SF Python Holiday 2017 – Documentation

Release 1.0

Raymond Hettinger

Dec 08, 2017

My Mission Train thousands of Python Programmers

Contact Info raymond dot hettinger at gmail dot com

Company Mutable Minds, Inc.

Training videos Free to user's of Safari Online: "Modern Python: Big Ideas, Little Code"

Twitter Account @raymondh

CHAPTER

ONE

CLASS GENERATORS: DATACLASSES AND NAMEDTUPLES

1.1 Problem to be Solved

- In the HSL color system, dark orange is 33 degrees on the color wheel at 100% saturation and 50% lightness.
- We want to pass around HSL values in a way is space-efficient, that has beautiful code, and is easy to debug.

1.1.1 There are many ways

```
from bunch import Bunch
from types import SimpleNamespace
from typing import NamedTuple
from dataclasses import dataclass

# tuple ######################
dark_orange = (33, 1.00, 0.5)
print(dark_orange)


# dict #######################
dark_orange = {'hue': 33, 'saturation': 1.0, 'lightness': 0.5}
print(dark_orange)


# write simple class #########
class Color:
     def __init__(self, hue, saturation, lightness=0.5):
       self.hue = hue
       self.saturation = saturation
       self.lightness = lightness

dark_orange = Color(33, 1.00, 0.5)
print(dark_orange)


# bunch recipe ###############
dark_orange = Bunch(hue=33, saturation=1.00, lightness=0.6)
print(dark_orange)


# simple namespace ###########
dark_orange = SimpleNamespace(hue=33, saturation=1.00, lightness=0.6)
print(dark_orange)
```

```
# named tuples ###############
class Color(NamedTuple):
    hue: int
    saturation: float
    lightness: float = 0.5

dark_orange = Color(hue=33, saturation=1.00)
print(dark_orange)

# record by george sakkis ####
'''
Color = record.recordtype('Color', ['hue', 'saturation', 'lightness'])

dark_orange = Color(hue=33, saturation=1.00, lightness=0.5)
'''

# dataclass ##########################################
@dataclass
class Color:
    hue: int
    saturation: float
    lightness: float = 0.5

dark_orange = Color(hue=33, saturation=1.00)
print(dark_orange)


# dataclass with slots but no defaults ###############
@dataclass
class Color:
    __slots__ = ['hue', 'saturation', 'lightness']
    hue: int
    saturation: float
    lightness: float

dark_orange = Color(hue=33, saturation=1.00, lightness=0.5)
print(dark_orange)


# dataclass with slots and defaults, mostly manual ####
@dataclass(init=False)    # provides repr and eq
class Color:
    __slots__ = ['hue', 'saturation', 'lightness']
    hue: int
    saturation: float
    lightness: float

    def __init__(self, hue, saturation, lightness=0.5):
        self.hue = hue
        self.saturation = saturation
        self.lightness = lightness

dark_orange = Color(hue=33, saturation=1.00)
print(dark_orange)
```

This outputs:

```
(33, 1.0, 0.5)
{'hue': 33, 'saturation': 1.0, 'lightness': 0.5}
<__main__.Color object at 0x10385ef28>
<bunch.Bunch object at 0x10385ef60>
namespace(hue=33, lightness=0.6, saturation=1.0)
Color(hue=33, saturation=1.0, lightness=0.5)
Color(hue=33, saturation=1.0, lightness=0.5)
Color(hue=33, saturation=1.0, lightness=0.5)
Color(hue=33, saturation=1.0, lightness=0.5)
```

1.2 Bunch and SimpleNamespace

1.2.1 Characteristics

- Underlying Store: Instance dict
- Size: 296 bytes
- Mutable

1.2.2 Virtues

- Fast attribute access

1.2.3 Vices

- Takes a lot of space
- Not unpackable
- Not orderable
- Doesn't create a class
- No default values

1.2.4 Doing it Manually

We use normal attribute access to extract the data.

```
class Color:
    def __init__(self, hue, saturation, lightness):
        self.hue = hue
        self.saturation = saturation
        self.lightness = lightness
```

1.2.5 Putting it to Work

```
dark_orange = Bunch(hue=33, saturation=1.00, lightness=0.5)
print(dark_orange)
```

1.2.6 Recipe

```
class Bunch(object):
    def __init__(self, **kwds):
        self.__dict__.update(kwds)
```

1.2.7 Optimization

The keyword dictionary can directly *replace* the instance dictionary:

```
class Bunch(object):
    def __init__(self, **kwds):
        self.__dict__ = kwds
```

1.2.8 C Version Built-in to Python

```
from types import SimpleNamespace

dark_orange = SimpleNamespace(hue=33, saturation=1.00, lightness=0.5)
print(dark_orange)
```

1.3 Named Tuples

1.3.1 Characteristics

- Underlying Store: tuple
- Size: 72 bytes
- Immutable / Hashable
- Iterable / Unpackable

1.3.2 Virtues

- Easy to use
- Substitutable for dictionary
- Fast indexed access
- Fast slicing
- Somewhat compact
- Orderable

1.3.3 Vices

- Attribute access is slower than native access
- No built-in support for typing

1.3.4 Doing it Manually

We rely on *property* and *itemgetter* to extract the data.

```
from operator import itemgetter

class Color(tuple):
    'Color(hue, saturation, lightness)'

    __slots__ = ()

    _fields = ['hue', 'saturation', 'lightness']

    def __new__(cls, hue, saturation, lightness):
        return tuple.__new__(cls, (hue, saturation, lightness))

    def __repr__(self):
        return f'{self.__class__.__name__}(hue={self.hue!r}, ' \
               f'saturation={self.saturation!r}, lightness={self.lightness})'

    hue = property(itemgetter(0))
    saturation = property(itemgetter(1))
    lightness = property(itemgetter(2))
```

1.3.5 Doing it with Code Generation

```
Color = namedtuple('Color', ['hue', 'saturation', 'lightness'])
```

1.3.6 Nicer way with typing.NamedTuple

Let's add typing and default values :-)

```
class Color(NamedTuple):
    __slots__ = ()
    hue: int
    saturation: float
    lightness: float = 0.5
```

1.3.7 Putting it to Work

```
dark_orange = Color(33, 1.00, 0.50)        # http://rgb.to/darkorange
print(dark_orange)
```

1.4 Record (mutable named tuple)

1.4.1 Characteristics

- Underlying Store: Instance slots
- Size: 64 bytes

- Mutable
- Iterable / Unpackable

1.4.2 Virtues

- Fastest attribute access
- Most compact

1.4.3 Vices

- No built-in support for typing
- Slow indexed access
- Not orderable

1.4.4 Doing it Manually

We use normal attribute access to extract the data.

```
class Color(object):
    'Color(hue, saturation, lightness)'

    __slots__ = ('hue', 'saturation', 'lightness')

    def __init__(self, hue, saturation, lightness):
        self.hue = hue
        self.saturation = saturation
        self.lightness = lightness

    def __len__(self):
        return 3

    def __iter__(self):
        yield self.hue
        yield self.saturation
        yield self.lightness

    def __getitem__(self, index):
        return getattr(self, self.__slots__[index])

    def __setitem__(self, index, value):
        return setattr(self, self.__slots__[index], value)

    def __repr__(self):
        return 'Color(hue=%r, saturation=%r, lightness=%r)' % (self.hue, self.
↪saturation, self.lightness)

    def __eq__(self, other):
        return isinstance(other, self.__class__) and self.hue==other.hue and self.
↪saturation==other.saturation and self.lightness==other.lightness

dark_orange = Color(33, 1.00, 0.50)        # http://rgb.to/darkorange
```

```
print(dark_orange)
        lightness = property(itemgetter(2))
```

1.4.5 Doing it with Code Generation

```
Color = recordtype('Color', ['hue', 'saturation', 'lightness'])
```

1.4.6 Putting it to Work

```
dark_orange = Color(33, 1.00, 0.50)        # http://rgb.to/darkorange
print(dark_orange)
```

1.5 Dataclasses

PEP 557 has landed. Woohoo!

1.5.1 What it does

Per PEP 557, Data Classes can be thought of as "mutable namedtuples with defaults" and "one of the main design goals of Data Classes is to support static type checkers."

Two ways of looking at it:

- attribute based data holder
- boilerplate for class

Roughly, dataclasses are a tool for auto-generating code for:

- class docstring
- __init__
- __repr__
- __eq__ with strong type checking
- rich comparisons
- hashing
- read-only enforced by __setattr__ __delattr__

1.5.2 Normal Dataclass without Slots

In-line call:

```
from dataclasses import make_dataclass

Color = make_dataclass('Color', [('hue', int), ('saturation', 'float'), ('lightness',
↪'float')])
dark_orange = Color(hue=33, saturation=1.00, lightness=0.5)
```

Nicer way:

```
from dataclasses import dataclass

@dataclass
class Color:
    hue: int
    saturation: float
    lightness: float = 0.5
```

1.5.3 Dataclass with Slots

In-line call:

```
from dataclasses import make_dataclass

Color = make_dataclass('Color',
                       [('hue', int), ('saturation', float), ('lightness', float)],
                        namespace={'__slots__': ['hue', 'saturation', 'lightness']})
dark_orange = Color(hue=33, saturation=1.00, lightness=0.5)
```

Nicer way:

```
from dataclasses import dataclass

@dataclass
class Color:
    __slots__ = ['hue', 'saturation', 'lightness']
    hue: int
    saturation: float
    lightness: float

dark_orange = Color(hue=33, saturation=1.00, lightness=0.5
```

1.6 Examples

1.6.1 Wrapper for priority queues

```
@dataclass(order=True)
class KeyedItem:
    key: int
    item: Any=field(compare=False)
```

1.6.2 Emulate a namedtuple

Auto-generates the repr. No other benefit. Could let you change equality relationship.

```
from operator import itemgetter

@dataclass(init=False, eq=False, order=False)
class Point(tuple):
    __slots__ = ()
```

```
    def __new__(cls, x, y):
        return tuple.__new__(cls, (x, y))
    x: Any = property(itemgetter(0))
    y: Any = property(itemgetter(1))
```

1.7 Summary

Let's see, how do I feel about all this?

I really like the new dataclasses!

If you need mutable named tuples today, try George Sakkis's *Record* class or use the attached dataclass code that runs fine on Python3.6 (but not prior).

'nuff said.

1.7.1 Pretty Table

Summary of your code generation options:

Code Generators	Store	Space	Mutable
Namedtuple	tuple	72	No
Record	instslots	64	Yes
Dataclasses	instdict	296	Optional
Dataclasses w/slots	instslots	64	Optional

1.7.2 Gains and Losses

Some things as named tuples (in typing module):

- attribute access
- introspectable fields list
- defaults values
- variable annotations
- conflict between slots and default values

Some nice improvements:

- slotted version sightly more compact by 1 field
- faster attribute access
- optional mutability
- more specific equality tests
- controllable init, hash, rich compares, and repr
- subclassable composition

Some things aren't as nice:

- slower creation time
- not substitutable for tuples or unpackable

- not great for subclassing immutable types
- much higher complexity and learning curve (esp. with fields() function and build flags)

1.8 Recipes for Immediate Use

1.8.1 Bunch

```
# Credit: Alex Martelli, Doug Hudgeon (Python Cookbook 2nd Edition)
# https://www.safaribooksonline.com/library/view/python-cookbook-2nd/0596007973/
↪ch04s19.html

class Bunch(object):
    def __init__(self, **kwds):
        self.__dict__.update(kwds)

if __name__ == '__main__':
    x = 15
    y = 29
    point = Bunch(datum=y, squared=y*y, coord=x)
    print(point.datum, point.squared, point.coord)

    dark_orange = Bunch(hue=33, saturation=1.00, lightness=0.5)
```

1.8.2 Namedtuple

```
# Extract from Python3.7 source for Lib/collections/__init__.py

from keyword import iskeyword as _iskeyword
import sys as _sys
from operator import itemgetter as _itemgetter

_nt_itemgetters = {}

def namedtuple(typename, field_names, *, rename=False, module=None):
    """Returns a new subclass of tuple with named fields.

    >>> Point = namedtuple('Point', ['x', 'y'])
    >>> Point.__doc__                   # docstring for the new class
    'Point(x, y)'
    >>> p = Point(11, y=22)             # instantiate with positional args or keywords
    >>> p[0] + p[1]                     # indexable like a plain tuple
    33
    >>> x, y = p                        # unpack like a regular tuple
    >>> x, y
    (11, 22)
    >>> p.x + p.y                       # fields also accessible by name
    33
    >>> d = p._asdict()                 # convert to a dictionary
    >>> d['x']
    11
    >>> Point(**d)                      # convert from a dictionary
    Point(x=11, y=22)
    >>> p._replace(x=100)               # _replace() is like str.replace() but␣
↪targets named fields
```

```
    Point(x=100, y=22)

    """

    # Validate the field names.  At the user's option, either generate an error
    # message or automatically replace the field name with a valid name.
    if isinstance(field_names, str):
        field_names = field_names.replace(',', ' ').split()
    field_names = list(map(str, field_names))
    typename = str(typename)
    if rename:
        seen = set()
        for index, name in enumerate(field_names):
            if (not name.isidentifier()
                or _iskeyword(name)
                or name.startswith('_')
                or name in seen):
                field_names[index] = f'_{index}'
            seen.add(name)
    for name in [typename] + field_names:
        if type(name) is not str:
            raise TypeError('Type names and field names must be strings')
        if not name.isidentifier():
            raise ValueError('Type names and field names must be valid '
                             f'identifiers: {name!r}')
        if _iskeyword(name):
            raise ValueError('Type names and field names cannot be a '
                             f'keyword: {name!r}')
    seen = set()
    for name in field_names:
        if name.startswith('_') and not rename:
            raise ValueError('Field names cannot start with an underscore: '
                             f'{name!r}')
        if name in seen:
            raise ValueError(f'Encountered duplicate field name: {name!r}')
        seen.add(name)

    # Variables used in the methods and docstrings
    field_names = tuple(map(_sys.intern, field_names))
    num_fields = len(field_names)
    arg_list = repr(field_names).replace("'", "")[1:-1]
    repr_fmt = '(' + ', '.join(f'{name}=%r' for name in field_names) + ')'
    tuple_new = tuple.__new__
    _len = len

    # Create all the named tuple methods to be added to the class namespace

    s = f'def __new__(_cls, {arg_list}): return _tuple_new(_cls, ({arg_list}))'
    namespace = {'_tuple_new': tuple_new, '__name__': f'namedtuple_{typename}'}
    # Note: exec() has the side-effect of interning the typename and field names
    exec(s, namespace)
    __new__ = namespace['__new__']
    __new__.__doc__ = f'Create new instance of {typename}({arg_list})'

    @classmethod
    def _make(cls, iterable):
        result = tuple_new(cls, iterable)
        if _len(result) != num_fields:
```

```
            raise TypeError(f'Expected {num_fields} arguments, got {len(result)}')
        return result

    _make.__func__.__doc__ = (f'Make a new {typename} object from a sequence '
                              'or iterable')

    def _replace(_self, **kwds):
        result = _self._make(map(kwds.pop, field_names, _self))
        if kwds:
            raise ValueError(f'Got unexpected field names: {list(kwds)!r}')
        return result

    _replace.__doc__ = (f'Return a new {typename} object replacing specified '
                        'fields with new values')

    def __repr__(self):
        'Return a nicely formatted representation string'
        return self.__class__.__name__ + repr_fmt % self

    def _asdict(self):
        'Return a new OrderedDict which maps field names to their values.'
        return OrderedDict(zip(self._fields, self))

    def __getnewargs__(self):
        'Return self as a plain tuple.  Used by copy and pickle.'
        return tuple(self)

    # Modify function metadata to help with introspection and debugging

    for method in (__new__, _make.__func__, _replace,
                   __repr__, _asdict, __getnewargs__):
        method.__qualname__ = f'{typename}.{method.__name__}'

    # Build-up the class namespace dictionary
    # and use type() to build the result class
    class_namespace = {
        '__doc__': f'{typename}({arg_list})',
        '__slots__': (),
        '_fields': field_names,
        '__new__': __new__,
        '_make': _make,
        '_replace': _replace,
        '__repr__': __repr__,
        '_asdict': _asdict,
        '__getnewargs__': __getnewargs__,
    }
    cache = _nt_itemgetters
    for index, name in enumerate(field_names):
        try:
            itemgetter_object, doc = cache[index]
        except KeyError:
            itemgetter_object = _itemgetter(index)
            doc = f'Alias for field number {index}'
            cache[index] = itemgetter_object, doc
        class_namespace[name] = property(itemgetter_object, doc=doc)

    result = type(typename, (tuple,), class_namespace)
```

```
    # For pickling to work, the __module__ variable needs to be set to the frame
    # where the named tuple is created.  Bypass this step in environments where
    # sys._getframe is not defined (Jython for example) or sys._getframe is not
    # defined for arguments greater than 0 (IronPython), or where the user has
    # specified a particular module.
    if module is None:
        try:
            module = _sys._getframe(1).f_globals.get('__name__', '__main__')
        except (AttributeError, ValueError):
            pass
    if module is not None:
        result.__module__ = module

    return result
```

1.8.3 Record

This recipe is works on Python 2.2 to 2.7. I can easily be adapted to Python 3,

```
# RECORDS (PYTHON RECIPE) by George Sakkis
# http://code.activestate.com/recipes/576555-records/

__all__ = ['recordtype']

import sys
from textwrap import dedent
from keyword import iskeyword


def recordtype(typename, field_names, verbose=False, **default_kwds):
    '''Returns a new class with named fields.

    @keyword field_defaults: A mapping from (a subset of) field names to default
        values.
    @keyword default: If provided, the default value for all fields without an
        explicit default in `field_defaults`.

    >>> Point = recordtype('Point', 'x y', default=0)
    >>> Point.__doc__           # docstring for the new class
    'Point(x, y)'
    >>> Point()                 # instantiate with defaults
    Point(x=0, y=0)
    >>> p = Point(11, y=22)     # instantiate with positional args or keywords
    >>> p[0] + p.y              # accessible by name and index
    33
    >>> p.x = 100; p[1] =200    # modifiable by name and index
    >>> p
    Point(x=100, y=200)
    >>> x, y = p               # unpack
    >>> x, y
    (100, 200)
    >>> d = p.todict()         # convert to a dictionary
    >>> d['x']
    100
    >>> Point(**d) == p        # convert from a dictionary
    True
```

```
    '''
    # Parse and validate the field names.  Validation serves two purposes,
    # generating informative error messages and preventing template injection attacks.
    if isinstance(field_names, basestring):
        # names separated by whitespace and/or commas
        field_names = field_names.replace(',', ' ').split()
    field_names = tuple(map(str, field_names))
    if not field_names:
        raise ValueError('Records must have at least one field')
    for name in (typename,) + field_names:
        if not min(c.isalnum() or c=='_' for c in name):
            raise ValueError('Type names and field names can only contain '
                             'alphanumeric characters and underscores: %r' % name)
        if iskeyword(name):
            raise ValueError('Type names and field names cannot be a keyword: %r'
                             % name)
        if name[0].isdigit():
            raise ValueError('Type names and field names cannot start with a '
                             'number: %r' % name)
    seen_names = set()
    for name in field_names:
        if name.startswith('_'):
            raise ValueError('Field names cannot start with an underscore: %r'
                             % name)
        if name in seen_names:
            raise ValueError('Encountered duplicate field name: %r' % name)
        seen_names.add(name)
    # determine the func_defaults of __init__
    field_defaults = default_kwds.pop('field_defaults', {})
    if 'default' in default_kwds:
        default = default_kwds.pop('default')
        init_defaults = tuple(field_defaults.get(f,default) for f in field_names)
    elif not field_defaults:
        init_defaults = None
    else:
        default_fields = field_names[-len(field_defaults):]
        if set(default_fields) != set(field_defaults):
            raise ValueError('Missing default parameter values')
        init_defaults = tuple(field_defaults[f] for f in default_fields)
    if default_kwds:
        raise ValueError('Invalid keyword arguments: %s' % default_kwds)
    # Create and fill-in the class template
    numfields = len(field_names)
    argtxt = ', '.join(field_names)
    reprtxt = ', '.join('%s=%%r' % f for f in field_names)
    dicttxt = ', '.join('%r: self.%s' % (f,f) for f in field_names)
    tupletxt = repr(tuple('self.%s' % f for f in field_names)).replace("'",'')
    inittxt = '; '.join('self.%s=%s' % (f,f) for f in field_names)
    itertxt = '; '.join('yield self.%s' % f for f in field_names)
    eqtxt   = ' and '.join('self.%s==other.%s' % (f,f) for f in field_names)
    template = dedent('''
        class %(typename)s(object):
            '%(typename)s(%(argtxt)s)'

            __slots__  = %(field_names)r

            def __init__(self, %(argtxt)s):
                %(inittxt)s
```

```
            def __len__(self):
                return %(numfields)d

            def __iter__(self):
                %(itertxt)s

            def __getitem__(self, index):
                return getattr(self, self.__slots__[index])

            def __setitem__(self, index, value):
                return setattr(self, self.__slots__[index], value)

            def todict(self):
                'Return a new dict which maps field names to their values'
                return {%(dicttxt)s}

            def __repr__(self):
                return '%(typename)s(%(reprtxt)s)' %% %(tupletxt)s

            def __eq__(self, other):
                return isinstance(other, self.__class__) and %(eqtxt)s

            def __ne__(self, other):
                return not self==other

            def __getstate__(self):
                return %(tupletxt)s

            def __setstate__(self, state):
                %(tupletxt)s = state
    ''') % locals()
    # Execute the template string in a temporary namespace
    namespace = {}
    try:
        exec template in namespace
        if verbose: print template
    except SyntaxError, e:
        raise SyntaxError(e.message + ':\n' + template)
    cls = namespace[typename]
    cls.__init__.im_func.func_defaults = init_defaults
    # For pickling to work, the __module__ variable needs to be set to the frame
    # where the named tuple is created.  Bypass this step in enviroments where
    # sys._getframe is not defined (Jython for example).
    if hasattr(sys, '_getframe') and sys.platform != 'cli':
        cls.__module__ = sys._getframe(1).f_globals['__name__']
    return cls


if __name__ == '__main__':
    import doctest
    TestResults = recordtype('TestResults', 'failed, attempted')
    print TestResults(*doctest.testmod())
```

1.8.4 Dataclasses

This code is current as of 6 December 2017. It has been accepted and checked-in to the trunk for Python 3.7 but you should expect a few more tweaks before 3.7 goes final.

```
import sys
import types
from copy import deepcopy
import collections
import inspect

__all__ = ['dataclass',
           'field',
           'FrozenInstanceError',
           'InitVar',

           # Helper functions.
           'fields',
           'asdict',
           'astuple',
           'make_dataclass',
           'replace',
           ]

# Raised when an attempt is made to modify a frozen class.
class FrozenInstanceError(AttributeError): pass

# A sentinel object for default values to signal that a
#  default-factory will be used.
# This is given a nice repr() which will appear in the function
#  signature of dataclasses' constructors.
class _HAS_DEFAULT_FACTORY_CLASS:
    def __repr__(self):
        return '<factory>'
_HAS_DEFAULT_FACTORY = _HAS_DEFAULT_FACTORY_CLASS()

# A sentinel object to detect if a parameter is supplied or not.
class _MISSING_FACTORY:
    def __repr__(self):
        return '<missing>'
_MISSING = _MISSING_FACTORY()

# Since most per-field metadata will be unused, create an empty
#  read-only proxy that can be shared among all fields.
_EMPTY_METADATA = types.MappingProxyType({})

# Markers for the various kinds of fields and pseudo-fields.
_FIELD = object()                  # An actual field.
_FIELD_CLASSVAR = object()         # Not a field, but a ClassVar.
_FIELD_INITVAR = object()          # Not a field, but an InitVar.

# The name of an attribute on the class where we store the Field
#  objects. Also used to check if a class is a Data Class.
_MARKER = '__dataclass_fields__'

# The name of the function, that if it exists, is called at the end of
# __init__.
_POST_INIT_NAME = '__post_init__'
```

```
class _InitVarMeta(type):
    def __getitem__(self, params):
        return self

class InitVar(metaclass=_InitVarMeta):
    pass


# Instances of Field are only ever created from within this module,
#  and only from the field() function, although Field instances are
#  exposed externally as (conceptually) read-only objects.
# name and type are filled in after the fact, not in __init__. They're
#  not known at the time this class is instantiated, but it's
#  convenient if they're available later.
# When cls._MARKER is filled in with a list of Field objects, the name
#  and type fields will have been populated.
class Field:
    __slots__ = ('name',
                 'type',
                 'default',
                 'default_factory',
                 'repr',
                 'hash',
                 'init',
                 'compare',
                 'metadata',
                 '_field_type',  # Private: not to be used by user code.
                 )

    def __init__(self, default, default_factory, init, repr, hash, compare,
                 metadata):
        self.name = None
        self.type = None
        self.default = default
        self.default_factory = default_factory
        self.init = init
        self.repr = repr
        self.hash = hash
        self.compare = compare
        self.metadata = (_EMPTY_METADATA
                         if metadata is None or len(metadata) == 0 else
                         types.MappingProxyType(metadata))
        self._field_type = None

    def __repr__(self):
        return ('Field('
                f'name={self.name!r},'
                f'type={self.type},'
                f'default={self.default},'
                f'default_factory={self.default_factory},'
                f'init={self.init},'
                f'repr={self.repr},'
                f'hash={self.hash},'
                f'compare={self.compare},'
                f'metadata={self.metadata}'
                ')')
```

```
# This function is used instead of exposing Field creation directly,
#  so that a type checker can be told (via overloads) that this is a
#  function whose type depends on its parameters.
def field(*, default=_MISSING, default_factory=_MISSING, init=True, repr=True,
          hash=None, compare=True, metadata=None):
    """Return an object to identify dataclass fields.

    default is the default value of the field. default_factory is a
    0-argument function called to initialize a field's value. If init
    is True, the field will be a parameter to the class's __init__()
    function. If repr is True, the field will be included in the
    object's repr(). If hash is True, the field will be included in
    the object's hash(). If compare is True, the field will be used in
    comparison functions. metadata, if specified, must be a mapping
    which is stored but not otherwise examined by dataclass.

    It is an error to specify both default and default_factory.
    """

    if default is not _MISSING and default_factory is not _MISSING:
        raise ValueError('cannot specify both default and default_factory')
    return Field(default, default_factory, init, repr, hash, compare,
                 metadata)


def _tuple_str(obj_name, fields):
    # Return a string representing each field of obj_name as a tuple
    #  member. So, if fields is ['x', 'y'] and obj_name is "self",
    #  return "(self.x,self.y)".

    # Special case for the 0-tuple.
    if len(fields) == 0:
        return '()'
    # Note the trailing comma, needed if this turns out to be a 1-tuple.
    return f'({",".join([f"{obj_name}.{f.name}" for f in fields])},)'


def _create_fn(name, args, body, globals=None, locals=None,
               return_type=_MISSING):
    # Note that we mutate locals when exec() is called. Caller beware!
    if locals is None:
        locals = {}
    return_annotation = ''
    if return_type is not _MISSING:
        locals['_return_type'] = return_type
        return_annotation = '->_return_type'
    args = ','.join(args)
    body = '\n'.join(f' {b}' for b in body)

    txt = f'def {name}({args}){return_annotation}:\n{body}'

    exec(txt, globals, locals)
    return locals[name]


def _field_assign(frozen, name, value, self_name):
```

```
    # If we're a frozen class, then assign to our fields in __init__
    #  via object.__setattr__.  Otherwise, just use a simple
    #  assignment.
    # self_name is what "self" is called in this function: don't
    #  hard-code "self", since that might be a field name.
    if frozen:
        return f'object.__setattr__({self_name},{name!r},{value})'
    return f'{self_name}.{name}={value}'


def _field_init(f, frozen, globals, self_name):
    # Return the text of the line in the body of __init__ that will
    #  initialize this field.

    default_name = f'_dflt_{f.name}'
    if f.default_factory is not _MISSING:
        if f.init:
            # This field has a default factory.  If a parameter is
            #  given, use it.  If not, call the factory.
            globals[default_name] = f.default_factory
            value = (f'{default_name}() '
                     f'if {f.name} is _HAS_DEFAULT_FACTORY '
                     f'else {f.name}')
        else:
            # This is a field that's not in the __init__ params, but
            #  has a default factory function.  It needs to be
            #  initialized here by calling the factory function,
            #  because there's no other way to initialize it.

            # For a field initialized with a default=defaultvalue, the
            #  class dict just has the default value
            #  (cls.fieldname=defaultvalue). But that won't work for a
            #  default factory, the factory must be called in __init__
            #  and we must assign that to self.fieldname. We can't
            #  fall back to the class dict's value, both because it's
            #  not set, and because it might be different per-class
            #  (which, after all, is why we have a factory function!).

            globals[default_name] = f.default_factory
            value = f'{default_name}()'
    else:
        # No default factory.
        if f.init:
            if f.default is _MISSING:
                # There's no default, just do an assignment.
                value = f.name
            elif f.default is not _MISSING:
                globals[default_name] = f.default
                value = f.name
        else:
            # This field does not need initialization. Signify that to
            #  the caller by returning None.
            return None

    # Only test this now, so that we can create variables for the
    #  default.  However, return None to signify that we're not going
    #  to actually do the assignment statement for InitVars.
    if f._field_type == _FIELD_INITVAR:
```

```
        return None

    # Now, actually generate the field assignment.
    return _field_assign(frozen, f.name, value, self_name)


def _init_param(f):
    # Return the __init__ parameter string for this field.
    #  For example, the equivalent of 'x:int=3' (except instead of 'int',
    #  reference a variable set to int, and instead of '3', reference a
    #  variable set to 3).
    if f.default is _MISSING and f.default_factory is _MISSING:
        # There's no default, and no default_factory, just
        #  output the variable name and type.
        default = ''
    elif f.default is not _MISSING:
        # There's a default, this will be the name that's used to look it up.
        default = f'=_dflt_{f.name}'
    elif f.default_factory is not _MISSING:
        # There's a factory function. Set a marker.
        default = '=_HAS_DEFAULT_FACTORY'
    return f'{f.name}:_type_{f.name}{default}'


def _init_fn(fields, frozen, has_post_init, self_name):
    # fields contains both real fields and InitVar pseudo-fields.

    # Make sure we don't have fields without defaults following fields
    #  with defaults.  This actually would be caught when exec-ing the
    #  function source code, but catching it here gives a better error
    #  message, and future-proofs us in case we build up the function
    #  using ast.
    seen_default = False
    for f in fields:
        # Only consider fields in the __init__ call.
        if f.init:
            if not (f.default is _MISSING and f.default_factory is _MISSING):
                seen_default = True
            elif seen_default:
                raise TypeError(f'non-default argument {f.name!r} '
                                'follows default argument')

    globals = {'_MISSING': _MISSING,
               '_HAS_DEFAULT_FACTORY': _HAS_DEFAULT_FACTORY}

    body_lines = []
    for f in fields:
        # Do not initialize the pseudo-fields, only the real ones.
        line = _field_init(f, frozen, globals, self_name)
        if line is not None:
            # line is None means that this field doesn't require
            #  initialization. Just skip it.
            body_lines.append(line)

    # Does this class have a post-init function?
    if has_post_init:
        params_str = ','.join(f.name for f in fields
                              if f._field_type is _FIELD_INITVAR)
```

```
        body_lines += [f'{self_name}.{_POST_INIT_NAME}({params_str})']

    # If no body lines, use 'pass'.
    if len(body_lines) == 0:
        body_lines = ['pass']

    locals = {f'_type_{f.name}': f.type for f in fields}
    return _create_fn('__init__',
                      [self_name] +[_init_param(f) for f in fields if f.init],
                      body_lines,
                      locals=locals,
                      globals=globals,
                      return_type=None)


def _repr_fn(fields):
    return _create_fn('__repr__',
                      ['self'],
                      ['return self.__class__.__qualname__ + f"(' +
                       ', '.join([f"{f.name}={{self.{f.name}!r}}"
                                  for f in fields]) +
                       ')"'])


def _frozen_setattr(self, name, value):
    raise FrozenInstanceError(f'cannot assign to field {name!r}')


def _frozen_delattr(self, name):
    raise FrozenInstanceError(f'cannot delete field {name!r}')


def _cmp_fn(name, op, self_tuple, other_tuple):
    # Create a comparison function.  If the fields in the object are
    #  named 'x' and 'y', then self_tuple is the string
    #  '(self.x,self.y)' and other_tuple is the string
    #  '(other.x,other.y)'.

    return _create_fn(name,
                      ['self', 'other'],
                      [ 'if other.__class__ is self.__class__:',
                       f' return {self_tuple}{op}{other_tuple}',
                        'return NotImplemented'])


def _set_eq_fns(cls, fields):
    # Create and set the equality comparison methods on cls.
    # Pre-compute self_tuple and other_tuple, then re-use them for
    #  each function.
    self_tuple = _tuple_str('self', fields)
    other_tuple = _tuple_str('other', fields)
    for name, op in [('__eq__', '=='),
                     ('__ne__', '!='),
                     ]:
        _set_attribute(cls, name, _cmp_fn(name, op, self_tuple, other_tuple))


def _set_order_fns(cls, fields):
```

```
    # Create and set the ordering methods on cls.
    # Pre-compute self_tuple and other_tuple, then re-use them for
    #  each function.
    self_tuple = _tuple_str('self', fields)
    other_tuple = _tuple_str('other', fields)
    for name, op in [('__lt__', '<'),
                     ('__le__', '<='),
                     ('__gt__', '>'),
                     ('__ge__', '>='),
                     ]:
        _set_attribute(cls, name, _cmp_fn(name, op, self_tuple, other_tuple))


def _hash_fn(fields):
    self_tuple = _tuple_str('self', fields)
    return _create_fn('__hash__',
                      ['self'],
                      [f'return hash({self_tuple})'])


def _get_field(cls, a_name, a_type):
    # Return a Field object, for this field name and type.  ClassVars
    #  and InitVars are also returned, but marked as such (see
    #  f._field_type).

    # If the default value isn't derived from field, then it's
    #  only a normal default value.  Convert it to a Field().
    default = getattr(cls, a_name, _MISSING)
    if isinstance(default, Field):
        f = default
    else:
        f = field(default=default)

    # Assume it's a normal field until proven otherwise.
    f._field_type = _FIELD

    # Only at this point do we know the name and the type. Set them.
    f.name = a_name
    f.type = a_type

    # If typing has not been imported, then it's impossible for
    #  any annotation to be a ClassVar. So, only look for ClassVar
    #  if typing has been imported.
    typing = sys.modules.get('typing')
    if typing is not None:
        # This test uses a typing internal class, but it's the best
        #  way to test if this is a ClassVar.
        if type(a_type) is typing._ClassVar:
            # This field is a ClassVar, so it's not a field.
            f._field_type = _FIELD_CLASSVAR

    if f._field_type is _FIELD:
        # Check if this is an InitVar.
        if a_type is InitVar:
            # InitVars are not fields, either.
            f._field_type = _FIELD_INITVAR

    # Validations for fields.  This is delayed until now, instead of
```

```
    # in the Field() constructor, since only here do we know the field
    # name, which allows better error reporting.

    # Special restrictions for ClassVar and InitVar.
    if f._field_type in (_FIELD_CLASSVAR, _FIELD_INITVAR):
        if f.default_factory is not _MISSING:
            raise TypeError(f'field {f.name} cannot have a '
                            'default factory')
        # Should I check for other field settings? default_factory
        #  seems the most serious to check for. Maybe add others.  For
        #  example, how about init=False (or really,
        #  init=<not-the-default-init-value>)? It makes no sense for
        #  ClassVar and InitVar to specify init=<anything>.

    # For real fields, disallow mutable defaults for known types.
    if f._field_type is _FIELD and isinstance(f.default, (list, dict, set)):
        raise ValueError(f'mutable default {type(f.default)} for field '
                         f'{f.name} is not allowed: use default_factory')

    return f


def _find_fields(cls):
    # Return a list of Field objects, in order, for this class (and no
    #  base classes).  Fields are found from __annotations__ (which is
    #  guaranteed to be ordered).  Default values are from class
    #  attributes, if a field has a default.  If the default value is
    #  a Field(), then it contains additional info beyond (and
    #  possibly including) the actual default value.  Pseudo-fields
    #  ClassVars and InitVars are included, despite the fact that
    #  they're not real fields.  That's deal with later.

    annotations = getattr(cls, '__annotations__', {})

    return [_get_field(cls, a_name, a_type)
            for a_name, a_type in annotations.items()]


def _set_attribute(cls, name, value):
    # Raise TypeError if an attribute by this name already exists.
    if name in cls.__dict__:
        raise TypeError(f'Cannot overwrite attribute {name} '
                        f'in {cls.__name__}')
    setattr(cls, name, value)


def _process_class(cls, repr, eq, order, hash, init, frozen):
    # Use an OrderedDict because:
    #  - Order matters!
    #  - Derived class fields overwrite base class fields, but the
    #    order is defined by the base class, which is found first.
    fields = collections.OrderedDict()

    # Find our base classes in reverse MRO order, and exclude
    #  ourselves.  In reversed order so that more derived classes
    #  override earlier field definitions in base classes.
    for b in cls.__mro__[-1:0:-1]:
        # Only process classes that have been processed by our
```

```
        #  decorator.  That is, they have a _MARKER attribute.
        base_fields = getattr(b, _MARKER, None)
        if base_fields:
            for f in base_fields.values():
                fields[f.name] = f

    # Now find fields in our class.  While doing so, validate some
    #  things, and set the default values (as class attributes)
    #  where we can.
    for f in _find_fields(cls):
        fields[f.name] = f

        # If the class attribute (which is the default value for
        #  this field) exists and is of type 'Field', replace it
        #  with the real default.  This is so that normal class
        #  introspection sees a real default value, not a Field.
        if isinstance(getattr(cls, f.name, None), Field):
            if f.default is _MISSING:
                # If there's no default, delete the class attribute.
                #  This happens if we specify field(repr=False), for
                #  example (that is, we specified a field object, but
                #  no default value).  Also if we're using a default
                #  factory.  The class attribute should not be set at
                #  all in the post-processed class.
                delattr(cls, f.name)
            else:
                setattr(cls, f.name, f.default)

    # Remember all of the fields on our class (including bases).  This
    #  marks this class as being a dataclass.
    setattr(cls, _MARKER, fields)

    # We also need to check if a parent class is frozen: frozen has to
    #  be inherited down.
    is_frozen = frozen or cls.__setattr__ is _frozen_setattr

    # If we're generating ordering methods, we must be generating
    #  the eq methods.
    if order and not eq:
        raise ValueError('eq must be true if order is true')

    if init:
        # Does this class have a post-init function?
        has_post_init = hasattr(cls, _POST_INIT_NAME)

        # Include InitVars and regular fields (so, not ClassVars).
        _set_attribute(cls, '__init__',
                       _init_fn(list(filter(lambda f: f._field_type
                                                 in (_FIELD, _FIELD_INITVAR),
                                            fields.values())),
                                is_frozen,
                                has_post_init,
                                # The name to use for the "self" param
                                #  in __init__.  Use "self" if possible.
                                '__dataclass_self__' if 'self' in fields
                                    else 'self',
                                ))
```

```
    # Get the fields as a list, and include only real fields.  This is
    #  used in all of the following methods.
    field_list = list(filter(lambda f: f._field_type is _FIELD,
                             fields.values()))

    if repr:
        _set_attribute(cls, '__repr__',
                       _repr_fn(list(filter(lambda f: f.repr, field_list))))

    if is_frozen:
        _set_attribute(cls, '__setattr__', _frozen_setattr)
        _set_attribute(cls, '__delattr__', _frozen_delattr)

    generate_hash = False
    if hash is None:
        if eq and frozen:
            # Generate a hash function.
            generate_hash = True
        elif eq and not frozen:
            # Not hashable.
            _set_attribute(cls, '__hash__', None)
        elif not eq:
            # Otherwise, use the base class definition of hash().  That is,
            #  don't set anything on this class.
            pass
        else:
            assert "can't get here"
    else:
        generate_hash = hash
    if generate_hash:
        _set_attribute(cls, '__hash__',
                       _hash_fn(list(filter(lambda f: f.compare
                                                      if f.hash is None
                                                      else f.hash,
                                            field_list))))

    if eq:
        # Create and __eq__ and __ne__ methods.
        _set_eq_fns(cls, list(filter(lambda f: f.compare, field_list)))

    if order:
        # Create and __lt__, __le__, __gt__, and __ge__ methods.
        # Create and set the comparison functions.
        _set_order_fns(cls, list(filter(lambda f: f.compare, field_list)))

    if not getattr(cls, '__doc__'):
        # Create a class doc-string.
        cls.__doc__ = (cls.__name__ +
                       str(inspect.signature(cls)).replace(' -> None', ''))

    return cls


# _cls should never be specified by keyword, so start it with an
#  underscore. The presense of _cls is used to detect if this
#  decorator is being called with parameters or not.
def dataclass(_cls=None, *, init=True, repr=True, eq=True, order=False,
              hash=None, frozen=False):
```

```
    """Returns the same class as was passed in, with dunder methods
    added based on the fields defined in the class.

    Examines PEP 526 __annotations__ to determine fields.

    If init is true, an __init__() method is added to the class. If
    repr is true, a __repr__() method is added. If order is true, rich
    comparison dunder methods are added. If hash is true, a __hash__()
    method function is added. If frozen is true, fields may not be
    assigned to after instance creation.
    """

    def wrap(cls):
        return _process_class(cls, repr, eq, order, hash, init, frozen)

    # See if we're being called as @dataclass or @dataclass().
    if _cls is None:
        # We're called with parens.
        return wrap

    # We're called as @dataclass without parens.
    return wrap(_cls)


def fields(class_or_instance):
    """Return a tuple describing the fields of this dataclass.

    Accepts a dataclass or an instance of one. Tuple elements are of
    type Field.
    """

    # Might it be worth caching this, per class?
    try:
        fields =  getattr(class_or_instance, _MARKER)
    except AttributeError:
        raise TypeError('must be called with a dataclass type or instance')

    # Exclude pseudo-fields.
    return tuple(f for f in fields.values() if f._field_type is _FIELD)


def _isdataclass(obj):
    """Returns True if obj is an instance of a dataclass."""
    return not isinstance(obj, type) and hasattr(obj, _MARKER)


def asdict(obj, *, dict_factory=dict):
    """Return the fields of a dataclass instance as a new dictionary mapping
    field names to field values.

    Example usage:

      @dataclass
      class C:
          x: int
          y: int

      c = C(1, 2)
```

```
      assert asdict(c) == {'x': 1, 'y': 2}

    If given, 'dict_factory' will be used instead of built-in dict.
    The function applies recursively to field values that are
    dataclass instances. This will also look into built-in containers:
    tuples, lists, and dicts.
    """
    if not _isdataclass(obj):
        raise TypeError("asdict() should be called on dataclass instances")
    return _asdict_inner(obj, dict_factory)

def _asdict_inner(obj, dict_factory):
    if _isdataclass(obj):
        result = []
        for f in fields(obj):
            value = _asdict_inner(getattr(obj, f.name), dict_factory)
            result.append((f.name, value))
        return dict_factory(result)
    elif isinstance(obj, (list, tuple)):
        return type(obj)(_asdict_inner(v, dict_factory) for v in obj)
    elif isinstance(obj, dict):
        return type(obj)((_asdict_inner(k, dict_factory), _asdict_inner(v, dict_
↪factory))
                         for k, v in obj.items())
    else:
        return deepcopy(obj)


def astuple(obj, *, tuple_factory=tuple):
    """Return the fields of a dataclass instance as a new tuple of field values.

    Example usage::

      @dataclass
      class C:
          x: int
          y: int

    c = C(1, 2)
    assert asdtuple(c) == (1, 2)

    If given, 'tuple_factory' will be used instead of built-in tuple.
    The function applies recursively to field values that are
    dataclass instances. This will also look into built-in containers:
    tuples, lists, and dicts.
    """

    if not _isdataclass(obj):
        raise TypeError("astuple() should be called on dataclass instances")
    return _astuple_inner(obj, tuple_factory)

def _astuple_inner(obj, tuple_factory):
    if _isdataclass(obj):
        result = []
        for f in fields(obj):
            value = _astuple_inner(getattr(obj, f.name), tuple_factory)
            result.append(value)
        return tuple_factory(result)
```

```
    elif isinstance(obj, (list, tuple)):
        return type(obj)(_astuple_inner(v, tuple_factory) for v in obj)
    elif isinstance(obj, dict):
        return type(obj)((_astuple_inner(k, tuple_factory), _astuple_inner(v, tuple_
↪factory))
                         for k, v in obj.items())
    else:
        return deepcopy(obj)


def make_dataclass(cls_name, fields, *, bases=(), namespace=None):
    """Return a new dynamically created dataclass.

    The dataclass name will be 'cls_name'.  'fields' is an interable
    of either (name, type) or (name, type, Field) objects. Field
    objects are created by calling 'field(name, type [, Field])'.

      C = make_class('C', [('a', int', ('b', int, Field(init=False))], bases=Base)

    is equivalent to:

      @dataclass
      class C(Base):
          a: int
          b: int = field(init=False)

    For the bases and namespace paremeters, see the builtin type() function.
    """

    if namespace is None:
        namespace = {}
    else:
        # Copy namespace since we're going to mutate it.
        namespace = namespace.copy()

    anns = collections.OrderedDict((name, tp) for name, tp, *_ in fields)
    namespace['__annotations__'] = anns
    for item in fields:
        if len(item) == 3:
            name, tp, spec = item
            namespace[name] = spec
    cls = type(cls_name, bases, namespace)
    return dataclass(cls)


def replace(obj, **changes):
    """Return a new object replacing specified fields with new values.

    This is especially useful for frozen classes.  Example usage:

      @dataclass(frozen=True)
      class C:
          x: int
          y: int

      c = C(1, 2)
      c1 = replace(c, x=3)
      assert c1.x == 3 and c1.y == 2
```

```
    """

    # We're going to mutate 'changes', but that's okay because it's a new
    #  dict, even if called with 'replace(obj, **my_changes)'.

    if not _isdataclass(obj):
        raise TypeError("replace() should be called on dataclass instances")

    # It's an error to have init=False fields in 'changes'.
    # If a field is not in 'changes', read its value from the provided obj.

    for f in getattr(obj, _MARKER).values():
        if not f.init:
            # Error if this field is specified in changes.
            if f.name in changes:
                raise ValueError(f'field {f.name} is declared with '
                                 'init=False, it cannot be specified with '
                                 'replace()')
            continue

        if f.name not in changes:
            changes[f.name] = getattr(obj, f.name)

    # Create the new object, which calls __init__() and __post_init__
    #  (if defined), using all of the init fields we've added and/or
    #  left in 'changes'.
    # If there are values supplied in changes that aren't fields, this
    #  will correctly raise a TypeError.
    return obj.__class__(**changes)
```