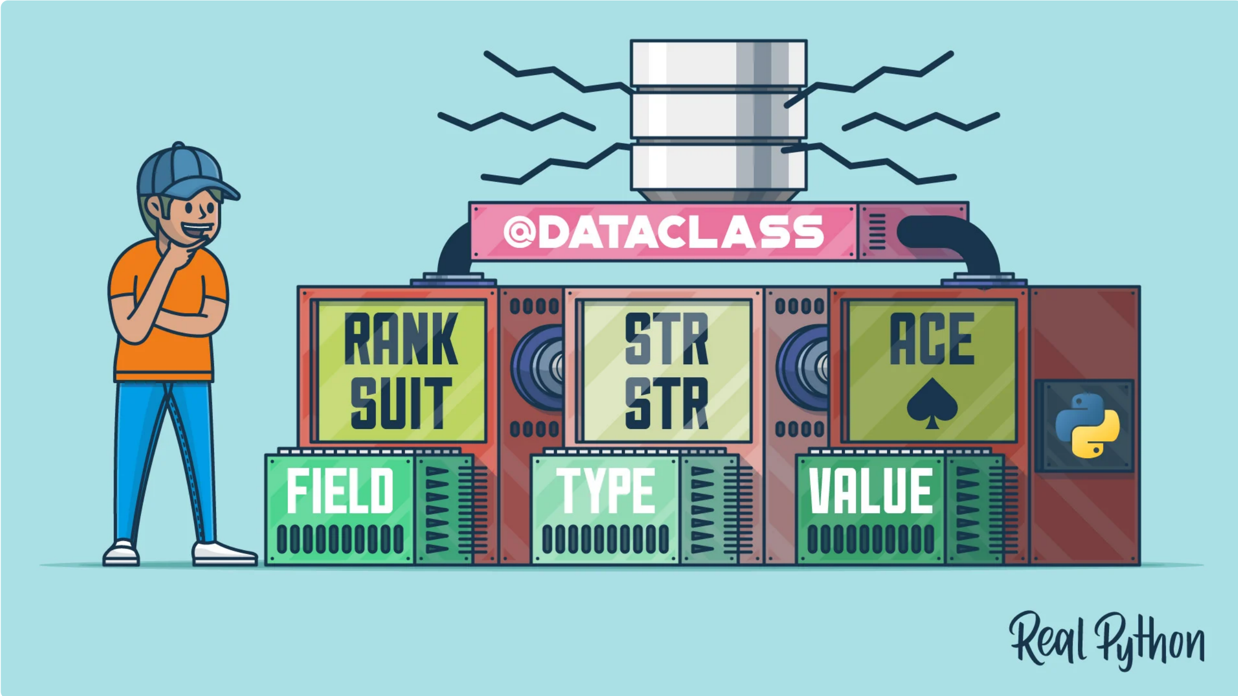

Data Classes in Python 3.7+ (Guide)

by Geir Arne Hjelle 27 Comments intermediate python

Mark as Completed

Tweet Share Email

Table of Contents

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Watch Now This tutorial has a related video course cre
Watch it together with the written tutorial to deepen yo
Classes in Python

One new and exciting feature coming in Python 3.7 is the d
typically containing mainly data, although there aren't reall
using the new `@dataclass` decorator, as follows:

Python

```python
from dataclasses import dataclass

@dataclass
class DataClassCard:
    rank: str
    suit: str
```

Note: This code, as well as all other examples in this tutorial, will only work in Python 3.7 and above.

A data class comes with basic functionality already implemented. For instance, you can instantiate, print, and compare data class instances straight out of the box:

Python >>>

```
>>> queen_of_hearts = DataClassCard('Q', 'Hearts')
>>> queen_of_hearts.rank
'Q'
>>> queen_of_hearts
DataClassCard(rank='Q', suit='Hearts')
>>> queen_of_hearts == DataClassCard('Q', 'Hearts')
True
```

Compare that to a regular class. A minimal regular class would look something like this:

Python

```
class RegularCard:
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit
```

While this is not much more code to write, you can already see signs of the boilerplate pain: `rank` and `suit` are both repeated three times simply to initialize an object. Furthermore, if you try to use this plain class, you'll notice that the representation of the objects is not very descriptive, and for some reason a queen of hearts is not the same as a queen of hearts:

Python >>>

```
>>> queen_of_hearts = RegularCard('Q', 'Hearts')
>>> queen_of_hearts.rank
'Q'
>>> queen_of_hearts
<__main__.RegularCard object at 0x7fb6eee35d30>
```

```
>>> queen_of_hearts == RegularCard('Q', 'Hearts')
False
```

Seems like data classes are helping us out behind the scenes. By default, data classes implement a `.__repr__()` method to provide a nice string representation and an `.__eq__()` method that can do basic object comparisons. For the `RegularCard` class to imitate the data class above, you need to add these methods as well:

```python
class RegularCard
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

    def __repr__(self):
        return (f'{self.__class__.__name__}'
                f'(rank={self.rank!r}, suit={self.suit!r})')

    def __eq__(self, other):
        if other.__class__ is not self.__class__:
            return NotImplemented
        return (self.rank, self.suit) == (other.rank, other.suit)
```

In this tutorial, you will learn exactly which conveniences data classes provide. In addition to nice representations and comparisons, you'll see:

- How to add default values to data class fields
- How data classes allow for ordering of objects
- How to represent immutable data
- How data classes handle inheritance

We will soon dive deeper into those features of data classes. However, you might be thinking that you have already seen something like this before.

Alternatives to Data Classes

For simple data structures, you have probably already used a `tuple` or a `dict`. You could represent the queen of hearts card in either of the following ways:

Python >>>

```
>>> queen_of_hearts_tuple = ('Q', 'Hearts')
>>> queen_of_hearts_dict = {'rank': 'Q', 'suit': 'Hearts'}
```

It works. However, it puts a lot of responsibility on you as a programmer:

- You need to remember that the `queen_of_hearts_...` variable represents a card.
- For the `tuple` version, you need to remember the order of the attributes. Writing `('Spades', 'A')` will mess up your program but probably not give you an easily understandable error message.
- If you use the `dict` version, you must make sure the names of the attributes are consistent. For instance `{'value': 'A', 'suit': 'Spades'}` will not work as expected.

Furthermore, using these structures is not ideal:

Python >>>

```
>>> queen_of_hearts_tuple[0]  # No named access
'Q'
>>> queen_of_hearts_dict['suit']  # Would be nicer with .suit
'Hearts'
```

A better alternative is the `namedtuple`. It has long been used to create readable small data structures. We can in fact recreate the data class example above using a `namedtuple` like this:

```python
from collections import namedtuple

NamedTupleCard = namedtuple('NamedTupleCard', ['rank', 'suit'])
```

This definition of `NamedTupleCard` will give the exact same output as our `DataClassCard` example did:

```python
>>> queen_of_hearts = NamedTupleCard('Q', 'Hearts')
>>> queen_of_hearts.rank
'Q'
>>> queen_of_hearts
NamedTupleCard(rank='Q', suit='Hearts')
>>> queen_of_hearts == NamedTupleCard('Q', 'Hearts')
True
```

So why even bother with data classes? First of all, data classes come with many more features than you have seen so far. At the same time, the `namedtuple` has some other features that are not necessarily desirable. By design, a `namedtuple` is a regular tuple. This can be seen in comparisons, for instance:

```python
>>> queen_of_hearts == ('Q', 'Hearts')
True
```

While this might seem like a good thing, this lack of awareness about its own type can lead to subtle and hard-to-find bugs, especially since it will also happily compare two different `namedtuple` classes:

Python

```
>>> Person = namedtuple('Person', ['first_initial', 'last_name']
>>> ace_of_spades = NamedTupleCard('A', 'Spades')
>>> ace_of_spades == Person('A', 'Spades')
True
```

The `namedtuple` also comes with some restrictions. For instance, it is hard to add default values to some of the fields in a `namedtuple`. A `namedtuple` is also by nature immutable. That is, the value of a `namedtuple` can never change. In some applications, this is an awesome feature, but in other settings, it would be nice to have more flexibility:

Python

```
>>> card = NamedTupleCard('7', 'Diamonds')
>>> card.rank = '9'
AttributeError: can't set attribute
```

Data classes will not replace all uses of `namedtuple`. For instance, if you need your data structure to behave like a tuple, then a named tuple is a great alternative!

Another alternative, and one of the inspirations for data classes, is the `attrs` project. With `attrs` installed (`pip install attrs`), you can write a card class as follows:

Python

```
import attr

@attr.s
class AttrsCard:
    rank = attr.ib()
    suit = attr.ib()
```

This can be used in exactly the same way as the `DataClassCard` and `NamedTupleCard` examples earlier. The `attrs` project is great and does support some features that data classes do not, including converters and validators. Furthermore, `attrs` has been around for a while and is supported in Python 2.7 as well as Python 3.4 and up. However, as `attrs` is not a part of the standard library, it does add an external dependency to your projects. Through data classes, similar functionality will be available everywhere.

In addition to `tuple`, `dict`, `namedtuple`, and `attrs`, there are many other similar projects, including `typing.NamedTuple`, `namedlist`, `attrdict`, `plumber`, and `fields`. While data classes are a great new alternative, there are still use cases where one of the older variants fits better. For instance, if you need compatibility with a specific API expecting tuples or need functionality not supported in data classes.

Basic Data Classes

Let us get back to data classes. As an example, we will create a `Position` class that will represent geographic positions with a name as well as the latitude and longitude:

Python

```python
from dataclasses import dataclass

@dataclass
class Position:
    name: str
    lon: float
    lat: float
```

What makes this a data class is the `@dataclass` decorator just above the class definition.

Beneath the `class Position:` line, you simply list the fields you want in your data class. The `:` notation used for the fields is using a new feature in Python 3.6 called variable

annotations. We will soon talk more about this notation and why we specify data types like `str` and `float`.

Those few lines of code are all you need. The new class is ready for use:

Python >>>

```
>>> pos = Position('Oslo', 10.8, 59.9)
>>> print(pos)
Position(name='Oslo', lon=10.8, lat=59.9)
>>> pos.lat
59.9
>>> print(f'{pos.name} is at {pos.lat}°N, {pos.lon}°E')
Oslo is at 59.9°N, 10.8°E
```

You can also create data classes similarly to how named tuples are created. The following is (almost) equivalent to the definition of `Position` above:

Python

```
from dataclasses import make_dataclass

Position = make_dataclass('Position', ['name', 'lat', 'lon'])
```

A data class is a regular Python class. The only thing that sets it apart is that it has basic data model methods like `.__init__()`, `.__repr__()`, and `.__eq__()` implemented for you.

Default Values

It is easy to add default values to the fields of your data class:

```
Python
from dataclasses import dataclass

@dataclass
class Position:
    name: str
    lon: float = 0.0
    lat: float = 0.0
```

This works exactly as if you had specified the default values in the definition of the `.__init__()` method of a regular class:

```
Python >>>
>>> Position('Null Island')
Position(name='Null Island', lon=0.0, lat=0.0)
>>> Position('Greenwich', lat=51.8)
Position(name='Greenwich', lon=0.0, lat=51.8)
>>> Position('Vancouver', -123.1, 49.3)
Position(name='Vancouver', lon=-123.1, lat=49.3)
```

Later you will learn about `default_factory`, which gives a way to provide more complicated default values.

Type Hints

So far, we have not made a big fuss of the fact that data classes support typing out of the box. You have probably noticed that we defined the fields with a type hint: `name: str` says that `name` should be a text string (`str` type).

In fact, adding some kind of type hint is mandatory when defining the fields in your data class. Without a type hint, the field will not be a part of the data class. However, if you do class. Without a type hint, the field will not be a part of the data class. However, if you do not want to add explicit types to your data class, use `typing.Any`:

```python
from dataclasses import dataclass
from typing import Any

@dataclass
class WithoutExplicitTypes:
    name: Any
    value: Any = 42
```

While you need to add type hints in some form when using data classes, these types are not enforced at runtime. The following code runs without any problems:

```python
>>> Position(3.14, 'pi day', 2018)
Position(name=3.14, lon='pi day', lat=2018)
```

This is how typing in Python usually works: Python is and will always be a dynamically typed language. To actually catch type errors, type checkers like Mypy can be run on your source code.

Adding Methods

You already know that a data class is just a regular class. That means that you can freely add your own methods to a data class. As an example, let us calculate the distance between one position and another, along the Earth's surface. One way to do this is by using the haversine formula:

You can add a `.distance_to()` method to your data class just like you can with normal classes:

Python

```python
from dataclasses import dataclass
from math import asin, cos, radians, sin, sqrt

@dataclass
class Position:
    name: str
    lon: float = 0.0
    lat: float = 0.0

    def distance_to(self, other):
        r = 6371  # Earth radius in kilometers
        lam_1, lam_2 = radians(self.lon), radians(other.lon)
        phi_1, phi_2 = radians(self.lat), radians(other.lat)
        h = (sin((phi_2 - phi_1) / 2)**2
             + cos(phi_1) * cos(phi_2) * sin((lam_2 - lam_1) / 2)**2)
        return 2 * r * asin(sqrt(h))
```

It works as you would expect:

Python >>>

```python
>>> oslo = Position('Oslo', 10.8, 59.9)
>>> vancouver = Position('Vancouver', -123.1, 49.3)
>>> oslo.distance_to(vancouver)
7181.7841229421165
```

More Flexible Data Classes

So far, you have seen some of the basic features of the data class: it gives you some convenience methods, and you can still add default values and other methods. Now you will learn about some more advanced features like parameters to the `@dataclass` decorator and the `field()` function. Together, they give you more control when creating a data class.

Let us return to the playing card example you saw at the beginning of the tutorial and add a class containing a deck of cards while we are at it:

Python

```
from dataclasses import dataclass
from typing import List

@dataclass
class PlayingCard:
    rank: str
    suit: str

@dataclass
class Deck:
    cards: List[PlayingCard]
```

A simple deck containing only two cards can be created like this:

Python >>>

```
>>> queen_of_hearts = PlayingCard('Q', 'Hearts')
>>> ace_of_spades = PlayingCard('A', 'Spades')
>>> two_cards = Deck([queen_of_hearts, ace_of_spades])
Deck(cards=[PlayingCard(rank='Q', suit='Hearts'),
            PlayingCard(rank='A', suit='Spades')])
```

Advanced Default Values

Say that you want to give a default value to the `Deck`. It would for example be convenient if `Deck()` created a regular (French) deck of 52 playing cards. First, specify the different ranks and suits. Then, add a function `make_french_deck()` that creates a list of instances of `PlayingCard`:

Python

```
RANKS = '2 3 4 5 6 7 8 9 10 J Q K A'.split()
SUITS = '♣ ♢ ♡ ♠'.split()

def make_french_deck():
    return [PlayingCard(r, s) for s in SUITS for r in RANKS]
```

For fun, the four different suits are specified using their Unicode symbols.

> **Note:** Above, we used Unicode glyphs like ♠ directly in the source code. We could do this because Python supports writing source code in UTF-8 by default. Refer to this page on Unicode input for how to enter these on your system. You could also enter the Unicode symbols for the suits using `\N` named character escapes (like `\N{BLACK SPADE SUIT}`) or `\u` Unicode escapes (like `\u2660`).

To simplify comparisons of cards later, the ranks and suits are also listed in their usual order.

Python >>>

```
>>> make_french_deck()
[PlayingCard(rank='2', suit='♣'), PlayingCard(rank='3', suit='♣'), ...
 PlayingCard(rank='K', suit='♠'), PlayingCard(rank='A', suit='♠')]
```

In theory, you could now use this function to specify a default value for `Deck.cards`:

Python

```python
from dataclasses import dataclass
from typing import List

@dataclass
class Deck:  # Will NOT work
    cards: List[PlayingCard] = make_french_deck()
```

Don't do this! This introduces one of the most common anti-patterns in Python: using mutable default arguments. The problem is that all instances of `Deck` will use the same list object as the default value of the `.cards` property. This means that if, say, one card is removed from one `Deck`, then it disappears from all other instances of `Deck` as well. Actually, data classes try to prevent you from doing this, and the code above will raise a `ValueError`.

Instead, data classes use something called a `default_factory` to handle mutable default values. To use `default_factory` (and many other cool features of data classes), you need to use the `field()` specifier:

Python

```python
from dataclasses import dataclass, field
from typing import List

@dataclass
class Deck:
    cards: List[PlayingCard] = field(default_factory=make_french_deck)
```

The argument to `default_factory` can be any zero parameter callable. Now it is easy to create a full deck of playing cards:

```python
>>> Deck()
Deck(cards=[PlayingCard(rank='2', suit='♣'), PlayingCard(rank='3', suit='♣'),
            PlayingCard(rank='K', suit='♠'), PlayingCard(rank='A', suit='♠')])
```

The `field()` specifier is used to customize each field of a data class individually. You will see some other examples later. For reference, these are the parameters `field()` supports:

- `default`: Default value of the field
- `default_factory`: Function that returns the initial value of the field
- `init`: Use field in `.__init__()` method? (Default is `True`.)
- `repr`: Use field in `repr` of the object? (Default is `True`.)
- `compare`: Include the field in comparisons? (Default is `True`.)
- `hash`: Include the field when calculating `hash()`? (Default is to use the same as for `compare`.)
- `metadata`: A mapping with information about the field

In the `Position` example, you saw how to add simple default values by writing `lat: float = 0.0`. However, if you also want to customize the field, for instance to hide it in the `repr`, you need to use the `default` parameter: `lat: float = field(default=0.0, repr=False)`. You may not specify both `default` and `default_factory`.

The `metadata` parameter is not used by the data classes themselves but is available for you (or third party packages) to attach information to fields. In the `Position` example, you could for instance specify that latitude and longitude should be given in degrees:

Python

```python
from dataclasses import dataclass, field

@dataclass
class Position:
    name: str
    lon: float = field(default=0.0, metadata={'unit': 'degrees'})
    lat: float = field(default=0.0, metadata={'unit': 'degrees'})
```

The metadata (and other information about a field) can be retrieved using the `fields()` function (note the plural *s*):

Python >>>

```python
>>> from dataclasses import fields
>>> fields(Position)
(Field(name='name',type=<class 'str'>,...,metadata={}),
 Field(name='lon',type=<class 'float'>,...,metadata={'unit': 'degrees'}),
 Field(name='lat',type=<class 'float'>,...,metadata={'unit': 'degrees'}))
>>> lat_unit = fields(Position)[2].metadata['unit']
>>> lat_unit
'degrees'
```


You Need Representation?

Recall that we can create decks of cards out of thin air:

```python
>>> Deck()
Deck(cards=[PlayingCard(rank='2', suit='♣'), PlayingCard(rank='3', suit='♣'),
            PlayingCard(rank='K', suit='♠'), PlayingCard(rank='A', suit='♠')])
```

While this representation of a `Deck` is explicit and readable, it is also very verbose. I have deleted 48 of the 52 cards in the deck in the output above. On an 80-column display, simply printing the full `Deck` takes up 22 lines! Let us add a more concise representation. In general, a Python object has two different string representations:

- `repr(obj)` is defined by `obj.__repr__()` and should return a developer-friendly representation of `obj`. If possible, this should be code that can recreate `obj`. Data classes do this.

- `str(obj)` is defined by `obj.__str__()` and should return a user-friendly representation of `obj`. Data classes do not implement a `.__str__()` method, so Python will fall back to the `.__repr__()` method.

Let us implement a user-friendly representation of a `PlayingCard`:

```python
from dataclasses import dataclass

@dataclass
class PlayingCard:
    rank: str
    suit: str

    def __str__(self):
        return f'{self.suit}{self.rank}'
```

The cards now look much nicer, but the deck is still as verbose as ever:

Python >>>

```python
>>> ace_of_spades = PlayingCard('A', '♠')
>>> ace_of_spades
PlayingCard(rank='A', suit='♠')
>>> print(ace_of_spades)
♠A
>>> print(Deck())
Deck(cards=[PlayingCard(rank='2', suit='♣'), PlayingCard(rank='3', suit='♣'),
            PlayingCard(rank='K', suit='♠'), PlayingCard(rank='A', suit='♠')])
```

To show that it is possible to add your own `.__repr__()` method as well, we will violate the principle that it should return code that can recreate an object. Practicality beats purity after all. The following code adds a more concise representation of the `Deck`:

Python

```python
from dataclasses import dataclass, field
from typing import List

@dataclass
class Deck:
    cards: List[PlayingCard] = field(default_factory=make_french_deck)

    def __repr__(self):
        cards = ', '.join(f'{c!s}' for c in self.cards)
        return f'{self.__class__.__name__}({cards})'
```

Note the `!s` specifier in the `{c!s}` format string. It means that we explicitly want to use the `str()` representation of each `PlayingCard`. With the new `.__repr__()`, the representation of `Deck` is easier on the eyes:

```python
>>> Deck()
Deck(♣2, ♣3, ♣4, ♣5, ♣6, ♣7, ♣8, ♣9, ♣10, ♣J, ♣Q, ♣K, ♣A,
     ♢2, ♢3, ♢4, ♢5, ♢6, ♢7, ♢8, ♢9, ♢10, ♢J, ♢Q, ♢K, ♢A,
     ♡2, ♡3, ♡4, ♡5, ♡6, ♡7, ♡8, ♡9, ♡10, ♡J, ♡Q, ♡K, ♡A,
     ♠2, ♠3, ♠4, ♠5, ♠6, ♠7, ♠8, ♠9, ♠10, ♠J, ♠Q, ♠K, ♠A)
```

This is a nicer representation of the deck. However, it comes at a cost. You're no longer able to recreate the deck by executing its representation. Often, you'd be better off implementing the same representation with `.__str__()` instead.

Comparing Cards

In many card games, cards are compared to each other. For instance in a typical trick taking game, the highest card takes the trick. As it is currently implemented, the `PlayingCard` class does not support this kind of comparison:

```python
>>> queen_of_hearts = PlayingCard('Q', '♡')
>>> ace_of_spades = PlayingCard('A', '♠')
>>> ace_of_spades > queen_of_hearts
TypeError: '>' not supported between instances of 'Card' and 'Card'
```

This is, however, (seemingly) easy to rectify:

Python

```python
from dataclasses import dataclass

@dataclass(order=True)
class PlayingCard:
    rank: str
    suit: str

    def __str__(self):
        return f'{self.suit}{self.rank}'
```

The `@dataclass` decorator has two forms. So far you have seen the simple form where `@dataclass` is specified without any parentheses and parameters. However, you can also give parameters to the `@dataclass()` decorator in parentheses. The following parameters are supported:

- `init`: Add `.__init__()` method? (Default is `True`.)
- `repr`: Add `.__repr__()` method? (Default is `True`.)
- `eq`: Add `.__eq__()` method? (Default is `True`.)
- `order`: Add ordering methods? (Default is `False`.)
- `unsafe_hash`: Force the addition of a `.__hash__()` method? (Default is `False`.)
- `frozen`: If `True`, assigning to fields raise an exception. (Default is `False`.)

See the original PEP for more information about each parameter. After setting `order=True`, instances of `PlayingCard` can be compared:

Python >>>

```python
>>> queen_of_hearts = PlayingCard('Q', '♡')
>>> ace_of_spades = PlayingCard('A', '♠')
>>> ace_of_spades > queen_of_hearts
```

```
False
```

How are the two cards compared though? You have not specified how the ordering should be done, and for some reason Python seems to believe that a Queen is higher than an Ace…

It turns out that data classes compare objects as if they were tuples of their fields. In other words, a Queen is higher than an Ace because `'Q'` comes after `'A'` in the alphabet:

```python
>>> ('A', '♠') > ('Q', '♡')
False
```

That does not really work for us. Instead, we need to define some kind of sort index that uses the order of `RANKS` and `SUITS`. Something like this:

```python
>>> RANKS = '2 3 4 5 6 7 8 9 10 J Q K A'.split()
>>> SUITS = '♣ ◊ ♡ ♠'.split()
>>> card = PlayingCard('Q', '♡')
>>> RANKS.index(card.rank) * len(SUITS) + SUITS.index(card.suit)
42
```

For `PlayingCard` to use this sort index for comparisons, we need to add a field `.sort_index` to the class. However, this field should be calculated from the other fields `.rank` and `.suit` automatically. This is exactly what the special method `.__post_init__()` is for. It allows for special processing after the regular `.__init__()` method is called:

Python

```python
from dataclasses import dataclass, field

RANKS = '2 3 4 5 6 7 8 9 10 J Q K A'.split()
SUITS = '♣ ♢ ♡ ♠'.split()

@dataclass(order=True)
class PlayingCard:
    sort_index: int = field(init=False, repr=False)
    rank: str
    suit: str

    def __post_init__(self):
        self.sort_index = (RANKS.index(self.rank) * len(SUITS)
                           + SUITS.index(self.suit))

    def __str__(self):
        return f'{self.suit}{self.rank}'
```

Note that `.sort_index` is added as the first field of the class. That way, the comparison is first done using `.sort_index` and only if there are ties are the other fields used. Using `field()`, you must also specify that `.sort_index` should not be included as a parameter in the `.__init__()` method (because it is calculated from the `.rank` and `.suit` fields). To avoid confusing the user about this implementation detail, it is probably also a good idea to remove `.sort_index` from the `repr` of the class.

Finally, aces are high:

Python >>>

```python
>>> queen_of_hearts = PlayingCard('Q', '♡')
>>> ace_of_spades = PlayingCard('A', '♠')
>>> ace_of_spades > queen_of_hearts
```

```
True
```

You can now easily create a sorted deck:

Python >>>

```
>>> Deck(sorted(make_french_deck()))
Deck(♣2, ♢2, ♡2, ♠2, ♣3, ♢3, ♡3, ♠3, ♣4, ♢4, ♡4, ♠4, ♣5,
     ♢5, ♡5, ♠5, ♣6, ♢6, ♡6, ♠6, ♣7, ♢7, ♡7, ♠7, ♣8, ♢8,
     ♡8, ♠8, ♣9, ♢9, ♡9, ♠9, ♣10, ♢10, ♡10, ♠10, ♣J, ♢J, ♡J,
     ♠J, ♣Q, ♢Q, ♡Q, ♠Q, ♣K, ♢K, ♡K, ♠K, ♣A, ♢A, ♡A, ♠A)
```

Or, if you don't care about sorting, this is how you draw a random hand of 10 cards:

Python >>>

```
>>> from random import sample
>>> Deck(sample(make_french_deck(), k=10))
Deck(♢2, ♡A, ♢10, ♣2, ♢3, ♠3, ♢A, ♠8, ♠9, ♠2)
```

Of course, you don't need `order=True` for that…

Immutable Data Classes

One of the defining features of the `namedtuple` you saw earlier is that it is immutable. That is, the value of its fields may never change. For many types of data classes, this is a great idea! To make a data class immutable, set `frozen=True` when you create it. For example, the following is an immutable version of the `Position` class you saw earlier:

Python

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class Position:
    name: str
    lon: float = 0.0
    lat: float = 0.0
```

In a frozen data class, you can not assign values to the fields after creation:

Python >>>

```python
>>> pos = Position('Oslo', 10.8, 59.9)
>>> pos.name
'Oslo'
>>> pos.name = 'Stockholm'
dataclasses.FrozenInstanceError: cannot assign to field 'name'
```

Be aware though that if your data class contains mutable fields, those might still change. This is true for all nested data structures in Python (see this video for further info):

Python

```python
from dataclasses import dataclass
from typing import List

@dataclass(frozen=True)
class ImmutableCard:
    rank: str
    suit: str

@dataclass(frozen=True)
class ImmutableDeck:
```

```
    cards: List[ImmutableCard]
```

Even though both `ImmutableCard` and `ImmutableDeck` are immutable, the list holding `cards` is not. You can therefore still change the cards in the deck:

Python >>>

```
>>> queen_of_hearts = ImmutableCard('Q', '♡')
>>> ace_of_spades = ImmutableCard('A', '♠')
>>> deck = ImmutableDeck([queen_of_hearts, ace_of_spades])
>>> deck
ImmutableDeck(cards=[ImmutableCard(rank='Q', suit='♡'), ImmutableCard(rank='A',
>>> deck.cards[0] = ImmutableCard('7', '♢')
>>> deck
ImmutableDeck(cards=[ImmutableCard(rank='7', suit='♢'), ImmutableCard(rank='A',
```

To avoid this, make sure all fields of an immutable data class use immutable types (but remember that types are not enforced at runtime). The `ImmutableDeck` should be implemented using a tuple instead of a list.

Inheritance

You can subclass data classes quite freely. As an example, we will extend our `Position` example with a `country` field and use it to record capitals:

```python
from dataclasses import dataclass

@dataclass
class Position:
    name: str
    lon: float
    lat: float

@dataclass
class Capital(Position):
    country: str
```

In this simple example, everything works without a hitch:

```python
>>> Capital('Oslo', 10.8, 59.9, 'Norway')
Capital(name='Oslo', lon=10.8, lat=59.9, country='Norway')
```

The `country` field of `Capital` is added after the three original fields in `Position`. Things get a little more complicated if any fields in the base class have default values:

```python
from dataclasses import dataclass

@dataclass
class Position:
    name: str
    lon: float = 0.0
    lat: float = 0.0

@dataclass
```

```python
class Capital(Position):
    country: str  # Does NOT work
```

This code will immediately crash with a `TypeError` complaining that "non-default argument 'country' follows default argument." The problem is that our new `country` field has no default value, while the `lon` and `lat` fields have default values. The data class will try to write an `.__init__()` method with the following signature:

Python

```python
def __init__(name: str, lon: float = 0.0, lat: float = 0.0, country: str):
    ...
```

However, this is not valid Python. If a parameter has a default value, all following parameters must also have a default value. In other words, if a field in a base class has a default value, then all new fields added in a subclass must have default values as well.

Another thing to be aware of is how fields are ordered in a subclass. Starting with the base class, fields are ordered in the order in which they are first defined. If a field is redefined in a subclass, its order does not change. For example, if you define `Position` and `Capital` as follows:

Python

```python
from dataclasses import dataclass

@dataclass
class Position:
    name: str
    lon: float = 0.0
    lat: float = 0.0

@dataclass
class Capital(Position):
    country: str = 'Unknown'
    lat: float = 40.0
```

Then the order of the fields in `Capital` will still be `name`, `lon`, `lat`, `country`. However, the default value of `lat` will be `40.0`.

Python

```
>>> Capital('Madrid', country='Spain')
Capital(name='Madrid', lon=0.0, lat=40.0, country='Spain')
```


Optimizing Data Classes

I'm going to end this tutorial with a few words about slots. Slots can be used to make classes faster and use less memory. Data classes have no explicit syntax for working with slots, but the normal way of creating slots works for data classes as well. (They really are just regular classes!)

```
Python
from dataclasses import dataclass

@dataclass
class SimplePosition:
    name: str
    lon: float
    lat: float

@dataclass
class SlotPosition:
    __slots__ = ['name', 'lon', 'lat']
    name: str
    lon: float
    lat: float
```

Essentially, slots are defined using `.__slots__` to list the variables on a class. Variables or attributes not present in `.__slots__` may not be defined. Furthermore, a slots class may not have default values.

The benefit of adding such restrictions is that certain optimizations may be done. For instance, slots classes take up less memory, as can be measured using Pympler:

```
Python                                                            >>>
>>> from pympler import asizeof
>>> simple = SimplePosition('London', -0.1, 51.5)
>>> slot = SlotPosition('Madrid', -3.7, 40.4)
>>> asizeof.asizesof(simple, slot)
(440, 248)
```

Similarly, slots classes are typically faster to work with. The following example measures the speed of attribute access on a slots data class and a regular data class using timeit from the standard library.

```
Python                                                                    >>>

>>> from timeit import timeit
>>> timeit('slot.name', setup="slot=SlotPosition('Oslo', 10.8, 59.9)", globals=
0.05882283499886398
>>> timeit('simple.name', setup="simple=SimplePosition('Oslo', 10.8, 59.9)", gl
0.09207444800267695
```

In this particular example, the slot class is about 35% faster.

Conclusion & Further Reading

Data classes are one of the new features of Python 3.7. With data classes, you do not have to write boilerplate code to get proper initialization, representation, and comparisons for your objects.

You have seen how to define your own data classes, as well as:

- How to add default values to the fields in your data class
- How to customize the ordering of data class objects
- How to work with immutable data classes
- How inheritance works for data classes

If you want to dive into all the details of data classes, have a look at PEP 557 as well as the discussions in the original GitHub repo.

In addition, Raymond Hettinger's PyCon 2018 talk Dataclasses: The code generator to end all code generators is well worth watching.

If you do not yet have Python 3.7, there is also a data classes backport for Python 3.6. And now, go forth and write less code!

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Using Data Classes in Python**

About **Geir Arne Hjelle**

Geir Arne is an avid Pythonista and a member of the Real Python tutorial team.

» More about Geir Arne

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren Dan Joanna

What Do You Think?

Rate this article:

Tweet Share in Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: intermediate python

Recommended Video Course: Using Data Classes in Python

❤️ Happy Pythoning!