CHAPTER 5
Data Class Builders

Data classes are like children. They are okay as a starting point, but to participate as a grownup object, they need to take some responsibility.

—Martin Fowler and Kent Beck[1]

Python offers a few ways to build a simple class that is just a collection of fields, with little or no extra functionality. That pattern is known as a "data class"—and `dataclasses` is one of the packages that supports this pattern. This chapter covers three different class builders that you may use as shortcuts to write data classes:

`collections.namedtuple`
: The simplest way—available since Python 2.6.

`typing.NamedTuple`
: An alternative that requires type hints on the fields—since Python 3.5, with `class` syntax added in 3.6.

`@dataclasses.dataclass`
: A class decorator that allows more customization than previous alternatives, adding lots of options and potential complexity—since Python 3.7.

After covering those class builders, we will discuss why *Data Class* is also the name of a code smell: a coding pattern that may be a symptom of poor object-oriented design.

1 From *Refactoring*, first edition, Chapter 3, "Bad Smells in Code, Data Class" section, page 87 (Addison-Wesley).

`typing.TypedDict` may seem like another data class builder. It uses similar syntax and is described right after `typing.NamedTuple` in the `typing` module documentation for Python 3.9.

However, `TypedDict` does not build concrete classes that you can instantiate. It's just syntax to write type hints for function parameters and variables that will accept mapping values used as records, with keys as field names. We'll see them in Chapter 15, "TypedDict" on page 526.

What's New in This Chapter

This chapter is new in the second edition of *Fluent Python*. The section "Classic Named Tuples" on page 169 appeared in Chapter 2 of the first edition, but the rest of the chapter is completely new.

We begin with a high-level overview of the three class builders.

Overview of Data Class Builders

Consider a simple class to represent a geographic coordinate pair, as shown in Example 5-1.

Example 5-1. class/coordinates.py

```
class Coordinate:

    def __init__(self, lat, lon):
        self.lat = lat
        self.lon = lon
```

That `Coordinate` class does the job of holding latitude and longitude attributes. Writing the `__init__` boilerplate becomes old real fast, especially if your class has more than a couple of attributes: each of them is mentioned three times! And that boilerplate doesn't buy us basic features we'd expect from a Python object:

```
    >>> from coordinates import Coordinate
    >>> moscow = Coordinate(55.76, 37.62)
    >>> moscow
    <coordinates.Coordinate object at 0x107142f10>  ❶
    >>> location = Coordinate(55.76, 37.62)
    >>> location == moscow  ❷
    False
    >>> (location.lat, location.lon) == (moscow.lat, moscow.lon)  ❸
    True
```

❶ `__repr__` inherited from `object` is not very helpful.

❷ Meaningless ==; the `__eq__` method inherited from `object` compares object IDs.

❸ Comparing two coordinates requires explicit comparison of each attribute.

The data class builders covered in this chapter provide the necessary `__init__`, `__repr__`, and `__eq__` methods automatically, as well as other useful features.

None of the class builders discussed here depend on inheritance to do their work. Both `collections.namedtuple` and `typing.NamedTuple` build classes that are `tuple` subclasses. `@dataclass` is a class decorator that does not affect the class hierarchy in any way. Each of them uses different metaprogramming techniques to inject methods and data attributes into the class under construction.

Here is a `Coordinate` class built with `namedtuple`—a factory function that builds a subclass of `tuple` with the name and fields you specify:

```
>>> from collections import namedtuple
>>> Coordinate = namedtuple('Coordinate', 'lat lon')
>>> issubclass(Coordinate, tuple)
True
>>> moscow = Coordinate(55.756, 37.617)
>>> moscow
Coordinate(lat=55.756, lon=37.617)  ❶
>>> moscow == Coordinate(lat=55.756, lon=37.617)  ❷
True
```

❶ Useful `__repr__`.

❷ Meaningful `__eq__`.

The newer `typing.NamedTuple` provides the same functionality, adding a type annotation to each field:

```
>>> import typing
>>> Coordinate = typing.NamedTuple('Coordinate',
...     [('lat', float), ('lon', float)])
>>> issubclass(Coordinate, tuple)
True
>>> typing.get_type_hints(Coordinate)
{'lat': <class 'float'>, 'lon': <class 'float'>}
```

A typed named tuple can also be constructed with the fields given as keyword arguments, like this:

```
Coordinate = typing.NamedTuple('Coordinate', lat=float, lon=float)
```

This is more readable, and also lets you provide the mapping of fields and types as `**fields_and_types`.

Since Python 3.6, `typing.NamedTuple` can also be used in a `class` statement, with type annotations written as described in PEP 526—Syntax for Variable Annotations. This is much more readable, and makes it easy to override methods or add new ones. Example 5-2 is the same `Coordinate` class, with a pair of `float` attributes and a custom `__str__` to display a coordinate formatted like 55.8°N, 37.6°E.

Example 5-2. typing_namedtuple/coordinates.py

```python
from typing import NamedTuple

class Coordinate(NamedTuple):
    lat: float
    lon: float

    def __str__(self):
        ns = 'N' if self.lat >= 0 else 'S'
        we = 'E' if self.lon >= 0 else 'W'
        return f'{abs(self.lat):.1f}°{ns}, {abs(self.lon):.1f}°{we}'
```

Although `NamedTuple` appears in the `class` statement as a superclass, it's actually not. `typing.NamedTuple` uses the advanced functionality of a metaclass[2] to customize the creation of the user's class. Check this out:

```
>>> issubclass(Coordinate, typing.NamedTuple)
False
>>> issubclass(Coordinate, tuple)
True
```

In the `__init__` method generated by `typing.NamedTuple`, the fields appear as parameters in the same order they appear in the `class` statement.

Like `typing.NamedTuple`, the `dataclass` decorator supports PEP 526 syntax to declare instance attributes. The decorator reads the variable annotations and automatically generates methods for your class. For comparison, check out the equivalent

2 Metaclasses are one of the subjects covered in Chapter 24, "Class Metaprogramming".

Coordinate class written with the help of the dataclass decorator, as shown in Example 5-3.

Example 5-3. dataclass/coordinates.py

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Coordinate:
    lat: float
    lon: float

    def __str__(self):
        ns = 'N' if self.lat >= 0 else 'S'
        we = 'E' if self.lon >= 0 else 'W'
        return f'{abs(self.lat):.1f}°{ns}, {abs(self.lon):.1f}°{we}'
```

Note that the body of the classes in Example 5-2 and Example 5-3 are identical—the difference is in the class statement itself. The @dataclass decorator does not depend on inheritance or a metaclass, so it should not interfere with your own use of these mechanisms.[3] The Coordinate class in Example 5-3 is a subclass of object.

Main Features

The different data class builders have a lot in common, as summarized in Table 5-1.

Table 5-1. Selected features compared across the three data class builders; x stands for an instance of a data class of that kind

	namedtuple	NamedTuple	dataclass
mutable instances	NO	NO	YES
class statement syntax	NO	YES	YES
construct dict	x._asdict()	x._asdict()	dataclasses.asdict(x)
get field names	x._fields	x._fields	[f.name for f in dataclasses.fields(x)]
get defaults	x._field_defaults	x._field_defaults	[f.default for f in dataclasses.fields(x)]
get field types	N/A	x.__annotations__	x.__annotations__
new instance with changes	x._replace(…)	x._replace(…)	dataclasses.replace(x, …)
new class at runtime	namedtuple(…)	NamedTuple(…)	dataclasses.make_dataclass(…)

3 Class decorators are covered in Chapter 24, "Class Metaprogramming," along with metaclasses. Both are ways of customizing class behavior beyond what is possible with inheritance.

The classes built by `typing.NamedTuple` and `@dataclass` have an `__annotations__` attribute holding the type hints for the fields. However, reading from `__annotations__` directly is not recommended. Instead, the recommended best practice to get that information is to call `inspect.get_annotations(MyClass)` (added in Python 3.10) or `typing.get_type_hints(MyClass)` (Python 3.5 to 3.9). That's because those functions provide extra services, like resolving forward references in type hints. We'll come back to this issue much later in the book, in "Problems with Annotations at Runtime" on page 538.

Now let's discuss those main features.

Mutable instances

A key difference between these class builders is that `collections.namedtuple` and `typing.NamedTuple` build `tuple` subclasses, therefore the instances are immutable. By default, `@dataclass` produces mutable classes. But the decorator accepts a keyword argument `frozen`—shown in Example 5-3. When `frozen=True`, the class will raise an exception if you try to assign a value to a field after the instance is initialized.

Class statement syntax

Only `typing.NamedTuple` and `dataclass` support the regular `class` statement syntax, making it easier to add methods and docstrings to the class you are creating.

Construct dict

Both named tuple variants provide an instance method (`._asdict`) to construct a `dict` object from the fields in a data class instance. The `dataclasses` module provides a function to do it: `dataclasses.asdict`.

Get field names and default values

All three class builders let you get the field names and default values that may be configured for them. In named tuple classes, that metadata is in the `._fields` and `._fields_defaults` class attributes. You can get the same metadata from a `dataclass` decorated class using the `fields` function from the `dataclasses` module. It returns a tuple of `Field` objects that have several attributes, including `name` and `default`.

Get field types

Classes defined with the help of `typing.NamedTuple` and `@dataclass` have a mapping of field names to type the `__annotations__` class attribute. As mentioned, use the `typing.get_type_hints` function instead of reading `__annotations__` directly.

New instance with changes

Given a named tuple instance `x`, the call `x._replace(**kwargs)` returns a new instance with some attribute values replaced according to the keyword arguments given. The `dataclasses.replace(x, **kwargs)` module-level function does the same for an instance of a `dataclass` decorated class.

New class at runtime

Although the `class` statement syntax is more readable, it is hardcoded. A framework may need to build data classes on the fly, at runtime. For that, you can use the default function call syntax of `collections.namedtuple`, which is likewise supported by `typing.NamedTuple`. The `dataclasses` module provides a `make_dataclass` function for the same purpose.

After this overview of the main features of the data class builders, let's focus on each of them in turn, starting with the simplest.

Classic Named Tuples

The `collections.namedtuple` function is a factory that builds subclasses of `tuple` enhanced with field names, a class name, and an informative `__repr__`. Classes built with `namedtuple` can be used anywhere where tuples are needed, and in fact many functions of the Python standard library that are used to return tuples now return named tuples for convenience, without affecting the user's code at all.

Each instance of a class built by `namedtuple` takes exactly the same amount of memory as a tuple because the field names are stored in the class.

Example 5-4 shows how we could define a named tuple to hold information about a city.

Example 5-4. Defining and using a named tuple type

```
>>> from collections import namedtuple
>>> City = namedtuple('City', 'name country population coordinates')  ❶
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))  ❷
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722,
139.691667))
>>> tokyo.population  ❸
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

❶ Two parameters are required to create a named tuple: a class name and a list of field names, which can be given as an iterable of strings or as a single space-delimited string.

❷ Field values must be passed as separate positional arguments to the constructor (in contrast, the `tuple` constructor takes a single iterable).

❸ You can access the fields by name or position.

As a `tuple` subclass, `City` inherits useful methods such as `__eq__` and the special methods for comparison operators—including `__lt__`, which allows sorting lists of `City` instances.

A named tuple offers a few attributes and methods in addition to those inherited from the tuple. Example 5-5 shows the most useful: the `_fields` class attribute, the class method `_make(iterable)`, and the `_asdict()` instance method.

Example 5-5. Named tuple attributes and methods (continued from the previous example)

```
>>> City._fields  ❶
('name', 'country', 'population', 'location')
>>> Coordinate = namedtuple('Coordinate', 'lat lon')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, Coordinate(28.613889, 77.208889))
>>> delhi = City._make(delhi_data)  ❷
>>> delhi._asdict()  ❸
{'name': 'Delhi NCR', 'country': 'IN', 'population': 21.935,
'location': Coordinate(lat=28.613889, lon=77.208889)}
>>> import json
>>> json.dumps(delhi._asdict())  ❹
'{"name": "Delhi NCR", "country": "IN", "population": 21.935,
"location": [28.613889, 77.208889]}'
```

❶ `._fields` is a tuple with the field names of the class.

❷ `._make()` builds `City` from an iterable; `City(*delhi_data)` would do the same.

❸ `._asdict()` returns a `dict` built from the named tuple instance.

❹ `._asdict()` is useful to serialize the data in JSON format, for example.

The `_asdict` method returned an `OrderedDict` until Python 3.7. Since Python 3.8, it returns a simple `dict`—which is OK now that we can rely on key insertion order. If you must have an `OrderedDict`, the `_asdict` documentation recommends building one from the result: `OrderedDict(x._asdict())`.

Since Python 3.7, `namedtuple` accepts the `defaults` keyword-only argument providing an iterable of N default values for each of the N rightmost fields of the class. Example 5-6 shows how to define a `Coordinate` named tuple with a default value for a `reference` field.

Example 5-6. Named tuple attributes and methods, continued from Example 5-5

```
>>> Coordinate = namedtuple('Coordinate', 'lat lon reference', defaults=['WGS84'])
>>> Coordinate(0, 0)
Coordinate(lat=0, lon=0, reference='WGS84')
>>> Coordinate._field_defaults
{'reference': 'WGS84'}
```

In "Class statement syntax" on page 168, I mentioned it's easier to code methods with the class syntax supported by `typing.NamedTuple` and `@dataclass`. You can also add methods to a `namedtuple`, but it's a hack. Skip the following box if you're not interested in hacks.

Hacking a namedtuple to Inject a Method

Recall how we built the `Card` class in Example 1-1 in Chapter 1:

```
    Card = collections.namedtuple('Card', ['rank', 'suit'])
```

Later in Chapter 1, I wrote a `spades_high` function for sorting. It would be nice if that logic was encapsulated in a method of `Card`, but adding `spades_high` to `Card` without the benefit of a `class` statement requires a quick hack: define the function and then assign it to a class attribute. Example 5-7 shows how.

Example 5-7. frenchdeck.doctest: Adding a class attribute and a method to `Card`, the `namedtuple` from "A Pythonic Card Deck" on page 5

```
>>> Card.suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)  ❶
>>> def spades_high(card):                                              ❷
...     rank_value = FrenchDeck.ranks.index(card.rank)
...     suit_value = card.suit_values[card.suit]
...     return rank_value * len(card.suit_values) + suit_value
...
>>> Card.overall_rank = spades_high                         ❸
>>> lowest_card = Card('2', 'clubs')
>>> highest_card = Card('A', 'spades')
>>> lowest_card.overall_rank()                              ❹
0
>>> highest_card.overall_rank()
51
```

❶ Attach a class attribute with values for each suit.

❷ `spades_high` will become a method; the first argument doesn't need to be named `self`. Anyway, it will get the receiver when called as a method.

❸ Attach the function to the `Card` class as a method named `overall_rank`.

❹ It works!

For readability and future maintenance, it's much better to code methods inside a `class` statement. But it's good to know this hack is possible, because it may come in handy.[4]

This was a small detour to showcase the power of a dynamic language.

Now let's check out the `typing.NamedTuple` variation.

Typed Named Tuples

The `Coordinate` class with a default field from Example 5-6 can be written using `typing.NamedTuple`, as shown in Example 5-8.

Example 5-8. typing_namedtuple/coordinates2.py

```
from typing import NamedTuple

class Coordinate(NamedTuple):
    lat: float                ❶
    lon: float
    reference: str = 'WGS84'  ❷
```

4 If you know Ruby, you know that injecting methods is a well-known but controversial technique among Rubyists. In Python, it's not as common, because it doesn't work with any built-in type—`str`, `list`, etc. I consider this limitation of Python a blessing.

❶ Every instance field must be annotated with a type.

❷ The `reference` instance field is annotated with a type and a default value.

Classes built by `typing.NamedTuple` don't have any methods beyond those that `collections.namedtuple` also generates—and those that are inherited from `tuple`. The only difference is the presence of the `__annotations__` class attribute—which Python completely ignores at runtime.

Given that the main feature of `typing.NamedTuple` are the type annotations, we'll take a brief look at them before resuming our exploration of data class builders.

Type Hints 101

Type hints—a.k.a. type annotations—are ways to declare the expected type of function arguments, return values, variables, and attributes.

The first thing you need to know about type hints is that they are not enforced at all by the Python bytecode compiler and interpreter.

> This is a very brief introduction to type hints, just enough to make sense of the syntax and meaning of the annotations used in `typing.NamedTuple` and `@dataclass` declarations. We will cover type hints for function signatures in Chapter 8 and more advanced annotations in Chapter 15. Here we'll mostly see hints with simple built-in types, such as `str`, `int`, and `float`, which are probably the most common types used to annotate fields of data classes.

No Runtime Effect

Think about Python type hints as "documentation that can be verified by IDEs and type checkers."

That's because type hints have no impact on the runtime behavior of Python programs. Check out Example 5-9.

Example 5-9. Python does not enforce type hints at runtime

```
>>> import typing
>>> class Coordinate(typing.NamedTuple):
...     lat: float
...     lon: float
...
>>> trash = Coordinate('Ni!', None)
>>> print(trash)
Coordinate(lat='Ni!', lon=None)    ❶
```

❶ I told you: no type checking at runtime!

If you type the code of Example 5-9 in a Python module, it will run and display a meaningless `Coordinate`, with no error or warning:

```
$ python3 nocheck_demo.py
Coordinate(lat='Ni!', lon=None)
```

The type hints are intended primarily to support third-party type checkers, like Mypy or the PyCharm IDE built-in type checker. These are static analysis tools: they check Python source code "at rest," not running code.

To see the effect of type hints, you must run one of those tools on your code—like a linter. For instance, here is what Mypy has to say about the previous example:

```
$ mypy nocheck_demo.py
nocheck_demo.py:8: error: Argument 1 to "Coordinate" has
incompatible type "str"; expected "float"
nocheck_demo.py:8: error: Argument 2 to "Coordinate" has
incompatible type "None"; expected "float"
```

As you can see, given the definition of `Coordinate`, Mypy knows that both arguments to create an instance must be of type `float`, but the assignment to `trash` uses a `str` and `None`.[5]

Now let's talk about the syntax and meaning of type hints.

Variable Annotation Syntax

Both `typing.NamedTuple` and `@dataclass` use the syntax of variable annotations defined in PEP 526. This is a quick introduction to that syntax in the context defining attributes in `class` statements.

The basic syntax of variable annotation is:

```
var_name: some_type
```

The "Acceptable type hints" section in PEP 484 explains what are acceptable types, but in the context of defining a data class, these types are more likely to be useful:

- A concrete class, for example, `str` or `FrenchDeck`
- A parameterized collection type, like `list[int]`, `tuple[str, float]`, etc.

5 In the context of type hints, `None` is not the `NoneType` singleton, but an alias for `NoneType` itself. This is strange when we stop to think about it, but appeals to our intuition and makes function return annotations easier to read in the common case of functions that return `None`.

- `typing.Optional`, for example, `Optional[str]`—to declare a field that can be a `str` or `None`

You can also initialize the variable with a value. In a `typing.NamedTuple` or `@dataclass` declaration, that value will become the default for that attribute if the corresponding argument is omitted in the constructor call:

```
var_name: some_type = a_value
```

The Meaning of Variable Annotations

We saw in "No Runtime Effect" on page 173 that type hints have no effect at runtime. But at import time—when a module is loaded—Python does read them to build the `__annotations__` dictionary that `typing.NamedTuple` and `@dataclass` then use to enhance the class.

We'll start this exploration with a simple class in Example 5-10, so that we can later see what extra features are added by `typing.NamedTuple` and `@dataclass`.

Example 5-10. meaning/demo_plain.py: a plain class with type hints

```
class DemoPlainClass:
    a: int           ❶
    b: float = 1.1   ❷
    c = 'spam'       ❸
```

❶ `a` becomes an entry in `__annotations__`, but is otherwise discarded: no attribute named `a` is created in the class.

❷ `b` is saved as an annotation, and also becomes a class attribute with value `1.1`.

❸ `c` is just a plain old class attribute, not an annotation.

We can verify that in the console, first reading the `__annotations__` of the `DemoPlainClass`, then trying to get its attributes named `a`, `b`, and `c`:

```
>>> from demo_plain import DemoPlainClass
>>> DemoPlainClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoPlainClass.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'DemoPlainClass' has no attribute 'a'
>>> DemoPlainClass.b
1.1
>>> DemoPlainClass.c
'spam'
```

Note that the `__annotations__` special attribute is created by the interpreter to record the type hints that appear in the source code—even in a plain class.

The `a` survives only as an annotation. It doesn't become a class attribute because no value is bound to it.[6] The `b` and `c` are stored as class attributes because they are bound to values.

None of those three attributes will be in a new instance of `DemoPlainClass`. If you create an object `o = DemoPlainClass()`, `o.a` will raise `AttributeError`, while `o.b` and `o.c` will retrieve the class attributes with values `1.1` and `'spam'`—that's just normal Python object behavior.

Inspecting a typing.NamedTuple

Now let's examine a class built with `typing.NamedTuple` (Example 5-11), using the same attributes and annotations as `DemoPlainClass` from Example 5-10.

Example 5-11. meaning/demo_nt.py: a class built with `typing.NamedTuple`

```
import typing

class DemoNTClass(typing.NamedTuple):
    a: int           ❶
    b: float = 1.1   ❷
    c = 'spam'       ❸
```

❶ `a` becomes an annotation and also an instance attribute.

❷ `b` is another annotation, and also becomes an instance attribute with default value `1.1`.

❸ `c` is just a plain old class attribute; no annotation will refer to it.

Inspecting the `DemoNTClass`, we get:

```
>>> from demo_nt import DemoNTClass
>>> DemoNTClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoNTClass.a
<_collections._tuplegetter object at 0x101f0f940>
>>> DemoNTClass.b
<_collections._tuplegetter object at 0x101f0f8b0>
>>> DemoNTClass.c
'spam'
```

6 Python has no concept of *undefined*, one of the silliest mistakes in the design of JavaScript. Thank Guido!

Here we have the same annotations for `a` and `b` as we saw in Example 5-10. But `typing.NamedTuple` creates `a` and `b` class attributes. The `c` attribute is just a plain class attribute with the value `'spam'`.

The `a` and `b` class attributes are *descriptors*—an advanced feature covered in Chapter 23. For now, think of them as similar to property getters: methods that don't require the explicit call operator `()` to retrieve an instance attribute. In practice, this means `a` and `b` will work as read-only instance attributes—which makes sense when we recall that `DemoNTClass` instances are just fancy tuples, and tuples are immutable.

`DemoNTClass` also gets a custom docstring:

```
>>> DemoNTClass.__doc__
'DemoNTClass(a, b)'
```

Let's inspect an instance of `DemoNTClass`:

```
>>> nt = DemoNTClass(8)
>>> nt.a
8
>>> nt.b
1.1
>>> nt.c
'spam'
```

To construct `nt`, we need to give at least the `a` argument to `DemoNTClass`. The constructor also takes a `b` argument, but it has a default value of `1.1`, so it's optional. The `nt` object has the `a` and `b` attributes as expected; it doesn't have a `c` attribute, but Python retrieves it from the class, as usual.

If you try to assign values to `nt.a`, `nt.b`, `nt.c`, or even `nt.z`, you'll get `AttributeError` exceptions with subtly different error messages. Try that and reflect on the messages.

Inspecting a class decorated with dataclass

Now, we'll examine Example 5-12.

Example 5-12. meaning/demo_dc.py: a class decorated with `@dataclass`

```
from dataclasses import dataclass

@dataclass
class DemoDataClass:
    a: int           ❶
    b: float = 1.1   ❷
    c = 'spam'       ❸
```

❶ a becomes an annotation and also an instance attribute controlled by a descriptor.

❷ b is another annotation, and also becomes an instance attribute with a descriptor and a default value 1.1.

❸ c is just a plain old class attribute; no annotation will refer to it.

Now let's check out __annotations__, __doc__, and the a, b, c attributes on DemoDataClass:

```
>>> from demo_dc import DemoDataClass
>>> DemoDataClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoDataClass.__doc__
'DemoDataClass(a: int, b: float = 1.1)'
>>> DemoDataClass.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'DemoDataClass' has no attribute 'a'
>>> DemoDataClass.b
1.1
>>> DemoDataClass.c
'spam'
```

The __annotations__ and __doc__ are not surprising. However, there is no attribute named a in DemoDataClass—in contrast with DemoNTClass from Example 5-11, which has a descriptor to get a from the instances as read-only attributes (that mysterious <_collections._tuplegetter>). That's because the a attribute will only exist in instances of DemoDataClass. It will be a public attribute that we can get and set, unless the class is frozen. But b and c exist as class attributes, with b holding the default value for the b instance attribute, while c is just a class attribute that will not be bound to the instances.

Now let's see how a DemoDataClass instance looks:

```
>>> dc = DemoDataClass(9)
>>> dc.a
9
>>> dc.b
1.1
>>> dc.c
'spam'
```

Again, a and b are instance attributes, and c is a class attribute we get via the instance.

As mentioned, DemoDataClass instances are mutable—and no type checking is done at runtime:

```
    >>> dc.a = 10
    >>> dc.b = 'oops'
```

We can do even sillier assignments:

```
    >>> dc.c = 'whatever'
    >>> dc.z = 'secret stash'
```

Now the `dc` instance has a `c` attribute—but that does not change the `c` class attribute. And we can add a new `z` attribute. This is normal Python behavior: regular instances can have their own attributes that don't appear in the class.[7]

More About @dataclass

We've only seen simple examples of `@dataclass` use so far. The decorator accepts several keyword arguments. This is its signature:

```
    @dataclass(*, init=True, repr=True, eq=True, order=False,
                  unsafe_hash=False, frozen=False)
```

The `*` in the first position means the remaining parameters are keyword-only. Table 5-2 describes them.

Table 5-2. Keyword parameters accepted by the `@dataclass` decorator

Option	Meaning	Default	Notes
`init`	Generate `__init__`	`True`	Ignored if `__init__` is implemented by user.
`repr`	Generate `__repr__`	`True`	Ignored if `__repr__` is implemented by user.
`eq`	Generate `__eq__`	`True`	Ignored if `__eq__` is implemented by user.
`order`	Generate `__lt__`, `__le__`, `__gt__`, `__ge__`	`False`	If `True`, raises exceptions if `eq=False`, or if any of the comparison methods that would be generated are defined or inherited.
`unsafe_hash`	Generate `__hash__`	`False`	Complex semantics and several caveats—see: dataclass documentation.
`frozen`	Make instances "immutable"	`False`	Instances will be reasonably safe from accidental change, but not really immutable.[a]

[a] `@dataclass` emulates immutability by generating `__setattr__` and `__delattr__`, which raise `dataclass.FrozenInstanceError`—a subclass of `AttributeError`—when the user attempts to set or delete a field.

7 Setting an attribute after `__init__` defeats the `__dict__` key-sharing memory optimization mentioned in "Practical Consequences of How dict Works" on page 102.

The defaults are really the most useful settings for common use cases. The options you are more likely to change from the defaults are:

`frozen=True`
: Protects against accidental changes to the class instances.

`order=True`
: Allows sorting of instances of the data class.

Given the dynamic nature of Python objects, it's not too hard for a nosy programmer to go around the protection afforded by `frozen=True`. But the necessary tricks should be easy to spot in a code review.

If the `eq` and `frozen` arguments are both `True`, `@dataclass` produces a suitable `__hash__` method, so the instances will be hashable. The generated `__hash__` will use data from all fields that are not individually excluded using a field option we'll see in "Field Options" on page 180. If `frozen=False` (the default), `@dataclass` will set `__hash__` to `None`, signalling that the instances are unhashable, therefore overriding `__hash__` from any superclass.

PEP 557—Data Classes has this to say about `unsafe_hash`:

> Although not recommended, you can force Data Classes to create a `__hash__` method with `unsafe_hash=True`. This might be the case if your class is logically immutable but can nonetheless be mutated. This is a specialized use case and should be considered carefully.

I will leave `unsafe_hash` at that. If you feel you must use that option, check the `dataclasses.dataclass` documentation.

Further customization of the generated data class can be done at a field level.

Field Options

We've already seen the most basic field option: providing (or not) a default value with the type hint. The instance fields you declare will become parameters in the generated `__init__`. Python does not allow parameters without defaults after parameters with defaults, therefore after you declare a field with a default value, all remaining fields must also have default values.

Mutable default values are a common source of bugs for beginning Python developers. In function definitions, a mutable default value is easily corrupted when one invocation of the function mutates the default, changing the behavior of further invocations—an issue we'll explore in "Mutable Types as Parameter Defaults: Bad Idea" on page 214 (Chapter 6). Class attributes are often used as default attribute values for instances, including in data classes. And `@dataclass` uses the default values in the type hints to generate parameters with defaults for `__init__`. To prevent bugs, `@dataclass` rejects the class definition in Example 5-13.

Example 5-13. dataclass/club_wrong.py: this class raises `ValueError`

```
@dataclass
class ClubMember:
    name: str
    guests: list = []
```

If you load the module with that `ClubMember` class, this is what you get:

```
$ python3 club_wrong.py
Traceback (most recent call last):
  File "club_wrong.py", line 4, in <module>
    class ClubMember:
  ...several lines omitted...
ValueError: mutable default <class 'list'> for field guests is not allowed:
use default_factory
```

The `ValueError` message explains the problem and suggests a solution: use `default_factory`. Example 5-14 shows how to correct `ClubMember`.

Example 5-14. dataclass/club.py: this `ClubMember` definition works

```
from dataclasses import dataclass, field

@dataclass
class ClubMember:
    name: str
    guests: list = field(default_factory=list)
```

In the `guests` field of Example 5-14, instead of a literal list, the default value is set by calling the `dataclasses.field` function with `default_factory=list`.

The `default_factory` parameter lets you provide a function, class, or any other callable, which will be invoked with zero arguments to build a default value each time an instance of the data class is created. This way, each instance of `ClubMember` will have its own `list`—instead of all instances sharing the same `list` from the class, which is rarely what we want and is often a bug.

It's good that `@dataclass` rejects class definitions with a `list` default value in a field. However, be aware that it is a partial solution that only applies to `list`, `dict`, and `set`. Other mutable values used as defaults will not be flagged by `@dataclass`. It's up to you to understand the problem and remember to use a default factory to set mutable default values.

If you browse the `dataclasses` module documentation, you'll see a `list` field defined with a novel syntax, as in Example 5-15.

Example 5-15. dataclass/club_generic.py: this `ClubMember` definition is more precise

```
from dataclasses import dataclass, field

@dataclass
class ClubMember:
    name: str
    guests: list[str] = field(default_factory=list)  ❶
```

❶ `list[str]` means "a list of str."

The new syntax `list[str]` is a parameterized generic type: since Python 3.9, the `list` built-in accepts that bracket notation to specify the type of the list items.

Prior to Python 3.9, the built-in collections did not support generic type notation. As a temporary workaround, there are corresponding collection types in the `typing` module. If you need a parameterized `list` type hint in Python 3.8 or earlier, you must import the `List` type from `typing` and use it: `List[str]`. For more about this issue, see "Legacy Support and Deprecated Collection Types" on page 272.

We'll cover generics in Chapter 8. For now, note that Examples 5-14 and 5-15 are both correct, and the Mypy type checker does not complain about either of those class definitions.

The difference is that `guests: list` means that `guests` can be a `list` of objects of any kind, while `guests: list[str]` says that `guests` must be a `list` in which every item is a `str`. This will allow the type checker to find (some) bugs in code that puts invalid items in the list, or that read items from it.

The `default_factory` is likely to be the most common option of the `field` function, but there are several others, listed in Table 5-3.

Table 5-3. Keyword arguments accepted by the `field` function

Option	Meaning	Default
`default`	Default value for field	`_MISSING_TYPE`[a]
`default_factory`	0-parameter function used to produce a default	`_MISSING_TYPE`
`init`	Include field in parameters to `__init__`	`True`
`repr`	Include field in `__repr__`	`True`

Option	Meaning	Default
compare	Use field in comparison methods `__eq__`, `__lt__`, etc.	True
hash	Include field in `__hash__` calculation	None[b]
metadata	Mapping with user-defined data; ignored by the @dataclass	None

[a] `dataclass._MISSING_TYPE` is a sentinel value indicating the option was not provided. It exists so we can set `None` as an actual default value, a common use case.

[b] The option `hash=None` means the field will be used in `__hash__` only if `compare=True`.

The `default` option exists because the `field` call takes the place of the default value in the field annotation. If you want to create an `athlete` field with a default value of `False`, and also omit that field from the `__repr__` method, you'd write this:

```
@dataclass
class ClubMember:
    name: str
    guests: list = field(default_factory=list)
    athlete: bool = field(default=False, repr=False)
```

Post-init Processing

The `__init__` method generated by `@dataclass` only takes the arguments passed and assigns them—or their default values, if missing—to the instance attributes that are instance fields. But you may need to do more than that to initialize the instance. If that's the case, you can provide a `__post_init__` method. When that method exists, `@dataclass` will add code to the generated `__init__` to call `__post_init__` as the last step.

Common use cases for `__post_init__` are validation and computing field values based on other fields. We'll study a simple example that uses `__post_init__` for both of these reasons.

First, let's look at the expected behavior of a `ClubMember` subclass named `HackerClubMember`, as described by doctests in Example 5-16.

Example 5-16. dataclass/hackerclub.py: doctests for `HackerClubMember`

```
"""
``HackerClubMember`` objects accept an optional ``handle`` argument::

    >>> anna = HackerClubMember('Anna Ravenscroft', handle='AnnaRaven')
    >>> anna
    HackerClubMember(name='Anna Ravenscroft', guests=[], handle='AnnaRaven')

If ``handle`` is omitted, it's set to the first part of the member's name::
    >>> leo = HackerClubMember('Leo Rochael')
    >>> leo
    HackerClubMember(name='Leo Rochael', guests=[], handle='Leo')

Members must have a unique handle. The following ``leo2`` will not be created,
because its ``handle`` would be 'Leo', which was taken by ``leo``::

    >>> leo2 = HackerClubMember('Leo DaVinci')
    Traceback (most recent call last):
      ...
    ValueError: handle 'Leo' already exists.

To fix, ``leo2`` must be created with an explicit ``handle``::

    >>> leo2 = HackerClubMember('Leo DaVinci', handle='Neo')
    >>> leo2
    HackerClubMember(name='Leo DaVinci', guests=[], handle='Neo')
"""
```

Note that we must provide `handle` as a keyword argument, because `HackerClubMember` inherits `name` and `guests` from `ClubMember`, and adds the `handle` field. The generated docstring for `HackerClubMember` shows the order of the fields in the constructor call:

```
>>> HackerClubMember.__doc__
"HackerClubMember(name: str, guests: list = <factory>, handle: str = '')"
```

Here, `<factory>` is a short way of saying that some callable will produce the default value for `guests` (in our case, the factory is the `list` class). The point is: to provide a `handle` but no `guests`, we must pass `handle` as a keyword argument.

The "Inheritance" section of the `dataclasses` module documentation explains how the order of the fields is computed when there are several levels of inheritance.

> In Chapter 14 we'll talk about misusing inheritance, particularly when the superclasses are not abstract. Creating a hierarchy of data classes is usually a bad idea, but it served us well here to make Example 5-17 shorter, focusing on the `handle` field declaration and `__post_init__` validation.

Example 5-17 shows the implementation.

Example 5-17. dataclass/hackerclub.py: code for `HackerClubMember`

```
from dataclasses import dataclass
from club import ClubMember
```

```
@dataclass
class HackerClubMember(ClubMember):                    ❶
    all_handles = set()                                ❷
    handle: str = ''                                   ❸

    def __post_init__(self):
        cls = self.__class__                           ❹
        if self.handle == '':                          ❺
            self.handle = self.name.split()[0]
        if self.handle in cls.all_handles:             ❻
            msg = f'handle {self.handle!r} already exists.'
            raise ValueError(msg)
        cls.all_handles.add(self.handle)               ❼
```

❶ `HackerClubMember` extends `ClubMember`.

❷ `all_handles` is a class attribute.

❸ `handle` is an instance field of type `str` with an empty string as its default value; this makes it optional.

❹ Get the class of the instance.

❺ If `self.handle` is the empty string, set it to the first part of `name`.

❻ If `self.handle` is in `cls.all_handles`, raise `ValueError`.

❼ Add the new `handle` to `cls.all_handles`.

Example 5-17 works as intended, but is not satisfactory to a static type checker. Next, we'll see why, and how to fix it.

Typed Class Attributes

If we type check Example 5-17 with Mypy, we are reprimanded:

```
$ mypy hackerclub.py
hackerclub.py:37: error: Need type annotation for "all_handles"
(hint: "all_handles: Set[<type>] = ...")
Found 1 error in 1 file (checked 1 source file)
```

Unfortunately, the hint provided by Mypy (version 0.910 as I review this) is not helpful in the context of `@dataclass` usage. First, it suggests using `Set`, but I am using Python 3.9 so I can use `set`—and avoid importing `Set` from `typing`. More importantly, if we add a type hint like `set[…]` to `all_handles`, `@dataclass` will find that annotation and make `all_handles` an instance field. We saw this happening in "Inspecting a class decorated with dataclass" on page 177.

The workaround defined in PEP 526—Syntax for Variable Annotations is ugly. To code a class variable with a type hint, we need to use a pseudotype named typing.ClassVar, which leverages the generics [] notation to set the type of the variable and also declare it a class attribute.

To make the type checker and @dataclass happy, this is how we are supposed to declare all_handles in Example 5-17:

```
    all_handles: ClassVar[set[str]] = set()
```

That type hint is saying:

> all_handles is a class attribute of type set-of-str, with an empty set as its default value.

To code that annotation, we must import ClassVar from the typing module.

The @dataclass decorator doesn't care about the types in the annotations, except in two cases, and this is one of them: if the type is ClassVar, an instance field will not be generated for that attribute.

The other case where the type of the field is relevant to @dataclass is when declaring *init-only variables*, our next topic.

Initialization Variables That Are Not Fields

Sometimes you may need to pass arguments to __init__ that are not instance fields. Such arguments are called *init-only variables* by the dataclasses documentation. To declare an argument like that, the dataclasses module provides the pseudotype InitVar, which uses the same syntax of typing.ClassVar. The example given in the documentation is a data class that has a field initialized from a database, and the database object must be passed to the constructor.

Example 5-18 shows the code that illustrates the "Init-only variables" section.

Example 5-18. Example from the dataclasses module documentation

```
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

Note how the database attribute is declared. InitVar will prevent @dataclass from treating database as a regular field. It will not be set as an instance attribute, and the dataclasses.fields function will not list it. However, database will be one of the arguments that the generated __init__ will accept, and it will be also passed to __post_init__. If you write that method, you must add a corresponding argument to the method signature, as shown in Example 5-18.

This rather long overview of @dataclass covered the most useful features—some of them appeared in previous sections, like "Main Features" on page 167 where we covered all three data class builders in parallel. The dataclasses documentation and PEP 526—Syntax for Variable Annotations have all the details.

In the next section, I present a longer example with @dataclass.

@dataclass Example: Dublin Core Resource Record

Often, classes built with @dataclass will have more fields than the very short examples presented so far. Dublin Core provides the foundation for a more typical @dataclass example.

> The Dublin Core Schema is a small set of vocabulary terms that can be used to describe digital resources (video, images, web pages, etc.), as well as physical resources such as books or CDs, and objects like artworks.[8]
>
> —Dublin Core on Wikipedia

The standard defines 15 optional fields; the Resource class in Example 5-19 uses 8 of them.

Example 5-19. dataclass/resource.py: code for Resource, a class based on Dublin Core terms

```
from dataclasses import dataclass, field
from typing import Optional
from enum import Enum, auto
from datetime import date


class ResourceType(Enum):  ❶
    BOOK = auto()
    EBOOK = auto()
    VIDEO = auto()


@dataclass
```

8 Source: Dublin Core article in the English Wikipedia.

```
class Resource:
    """Media resource description."""
    identifier: str                                    ❷
    title: str = '<untitled>'                          ❸
    creators: list[str] = field(default_factory=list)
    date: Optional[date] = None                        ❹
    type: ResourceType = ResourceType.BOOK             ❺
    description: str = ''
    language: str = ''
    subjects: list[str] = field(default_factory=list)
```

❶ This `Enum` will provide type-safe values for the `Resource.type` field.

❷ `identifier` is the only required field.

❸ `title` is the first field with a default. This forces all fields below to provide defaults.

❹ The value of `date` can be a `datetime.date` instance, or `None`.

❺ The `type` field default is `ResourceType.BOOK`.

Example 5-20 shows a doctest to demonstrate how a `Resource` record appears in code.

Example 5-20. dataclass/resource.py: code for `Resource`, a class based on Dublin Core terms

```
>>> description = 'Improving the design of existing code'
>>> book = Resource('978-0-13-475759-9', 'Refactoring, 2nd Edition',
...     ['Martin Fowler', 'Kent Beck'], date(2018, 11, 19),
...     ResourceType.BOOK, description, 'EN',
...     ['computer programming', 'OOP'])
>>> book  # doctest: +NORMALIZE_WHITESPACE
Resource(identifier='978-0-13-475759-9', title='Refactoring, 2nd Edition',
creators=['Martin Fowler', 'Kent Beck'], date=datetime.date(2018, 11, 19),
type=<ResourceType.BOOK: 1>, description='Improving the design of existing code',
language='EN', subjects=['computer programming', 'OOP'])
```

The `__repr__` generated by `@dataclass` is OK, but we can make it more readable. This is the format we want from `repr(book)`:

```
>>> book  # doctest: +NORMALIZE_WHITESPACE
Resource(
    identifier = '978-0-13-475759-9',
    title = 'Refactoring, 2nd Edition',
    creators = ['Martin Fowler', 'Kent Beck'],
    date = datetime.date(2018, 11, 19),
    type = <ResourceType.BOOK: 1>,
    description = 'Improving the design of existing code',
    language = 'EN',
    subjects = ['computer programming', 'OOP'],
)
```

Example 5-21 is the code of `__repr__` to produce the format shown in the last snippet. This example uses `dataclass.fields` to get the names of the data class fields.

Example 5-21. `dataclass/resource_repr.py`: code for `__repr__` method implemented in the `Resource` class from Example 5-19

```
    def __repr__(self):
        cls = self.__class__
        cls_name = cls.__name__
        indent = ' ' * 4
        res = [f'{cls_name}(']                            ❶
        for f in fields(cls):                             ❷
            value = getattr(self, f.name)                 ❸
            res.append(f'{indent}{f.name} = {value!r},')  ❹

        res.append(')')                                   ❺
        return '\n'.join(res)                             ❻
```

❶ Start the `res` list to build the output string with the class name and open parenthesis.

❷ For each field `f` in the class…

❸ …get the named attribute from the instance.

❹ Append an indented line with the name of the field and `repr(value)`—that's what the `!r` does.

❺ Append closing parenthesis.

❻ Build a multiline string from `res` and return it.

With this example inspired by the soul of Dublin, Ohio, we conclude our tour of Python's data class builders.

Data classes are handy, but your project may suffer if you overuse them. The next section explains.

Data Class as a Code Smell

Whether you implement a data class by writing all the code yourself or leveraging one of the class builders described in this chapter, be aware that it may signal a problem in your design.

In *Refactoring: Improving the Design of Existing Code*, 2nd ed. (Addison-Wesley), Martin Fowler and Kent Beck present a catalog of "code smells"—patterns in code that may indicate the need for refactoring. The entry titled "Data Class" starts like this:

> These are classes that have fields, getting and setting methods for fields, and nothing else. Such classes are dumb data holders and are often being manipulated in far too much detail by other classes.

In Fowler's personal website, there's an illuminating post titled "Code Smell". The post is very relevant to our discussion because he uses *data class* as one example of a code smell and suggests how to deal with it. Here is the post, reproduced in full.[9]

Code Smell

By Martin Fowler

A code smell is a surface indication that usually corresponds to a deeper problem in the system. The term was first coined by Kent Beck while helping me with my *Refactoring* book.

The quick definition above contains a couple of subtle points. Firstly, a smell is by definition something that's quick to spot—or sniffable as I've recently put it. A long method is a good example of this—just looking at the code and my nose twitches if I see more than a dozen lines of Java.

The second is that smells don't always indicate a problem. Some long methods are just fine. You have to look deeper to see if there is an underlying problem there—smells aren't inherently bad on their own—they are often an indicator of a problem rather than the problem themselves.

The best smells are something that's easy to spot and most of the time lead you to really interesting problems. Data classes (classes with all data and no behavior) are good examples of this. You look at them and ask yourself what behavior should be in this class. Then you start refactoring to move that behavior in there. Often simple questions and initial refactorings can be the vital step in turning anemic objects into something that really has class.

9 I am fortunate to have Martin Fowler as a colleague at Thoughtworks, so it took just 20 minutes to get his permission.

One of the nice things about smells is that it’s easy for inexperienced people to spot them, even if they don’t know enough to evaluate if there’s a real problem or to correct them. I’ve heard of lead developers who will pick a “smell of the week” and ask people to look for the smell and bring it up with the senior members of the team. Doing it one smell at a time is a good way of gradually teaching people on the team to be better programmers.

The main idea of object-oriented programming is to place behavior and data together in the same code unit: a class. If a class is widely used but has no significant behavior of its own, it’s possible that code dealing with its instances is scattered (and even duplicated) in methods and functions throughout the system—a recipe for maintenance headaches. That’s why Fowler’s refactorings to deal with a data class involve bringing responsibilities back into it.

Taking that into account, there are a couple of common scenarios where it makes sense to have a data class with little or no behavior.

Data Class as Scaffolding

In this scenario, the data class is an initial, simplistic implementation of a class to jump-start a new project or module. With time, the class should get its own methods, instead of relying on methods of other classes to operate on its instances. Scaffolding is temporary; eventually your custom class may become fully independent from the builder you used to start it.

Python is also used for quick problem solving and experimentation, and then it’s OK to leave the scaffolding in place.

Data Class as Intermediate Representation

A data class can be useful to build records about to be exported to JSON or some other interchange format, or to hold data that was just imported, crossing some system boundary. Python’s data class builders all provide a method or function to convert an instance to a plain `dict`, and you can always invoke the constructor with a `dict` used as keyword arguments expanded with `**`. Such a `dict` is very close to a JSON record.

In this scenario, the data class instances should be handled as immutable objects—even if the fields are mutable, you should not change them while they are in this intermediate form. If you do, you’re losing the key benefit of having data and behavior close together. When importing/exporting requires changing values, you should implement your own builder methods instead of using the given “as dict” methods or standard constructors.

Now we change the subject to see how to write patterns that match instances of arbitrary classes, and not just the sequences and mappings we've seen in "Pattern Matching with Sequences" on page 38 and "Pattern Matching with Mappings" on page 81.

Pattern Matching Class Instances

Class patterns are designed to match class instances by type and—optionally—by attributes. The subject of a class pattern can be any class instance, not only instances of data classes.[10]

There are three variations of class patterns: simple, keyword, and positional. We'll study them in that order.

Simple Class Patterns

We've already seen an example with simple class patterns used as subpatterns in "Pattern Matching with Sequences" on page 38:

```
        case [str(name), _, _, (float(lat), float(lon))]:
```

That pattern matches a four-item sequence where the first item must be an instance of `str`, and the last item must be a 2-tuple with two instances of `float`.

The syntax for class patterns looks like a constructor invocation. The following is a class pattern that matches `float` values without binding a variable (the case body can refer to `x` directly if needed):

```
    match x:
        case float():
            do_something_with(x)
```

But this is likely to be a bug in your code:

```
    match x:
        case float:  # DANGER!!!
            do_something_with(x)
```

In the preceding example, `case float:` matches any subject, because Python sees `float` as a variable, which is then bound to the subject.

The simple pattern syntax of `float(x)` is a special case that applies only to nine blessed built-in types, listed at the end of the "Class Patterns" section of PEP 634—Structural Pattern Matching: Specification:

```
bytes   dict   float   frozenset   int   list   set   str   tuple
```

10 I put this content here because it is the earliest chapter focusing on user-defined classes, and I thought pattern matching with classes was too important to wait until Part II of the book. My philosophy: it's more important to know how to use classes than to define classes.

In those classes, the variable that looks like a constructor argument—e.g., the `x` in `float(x)`—is bound to the whole subject instance or the part of the subject that matches a subpattern, as exemplified by `str(name)` in the sequence pattern we saw earlier:

```
case [str(name), _, _, (float(lat), float(lon))]:
```

If the class is not one of those nine blessed built-ins, then the argument-like variables represent patterns to be matched against attributes of an instance of that class.

Keyword Class Patterns

To understand how to use keyword class patterns, consider the following `City` class and five instances in Example 5-22.

Example 5-22. `City` class and a few instances

```
import typing

class City(typing.NamedTuple):
    continent: str
    name: str
    country: str


cities = [
    City('Asia', 'Tokyo', 'JP'),
    City('Asia', 'Delhi', 'IN'),
    City('North America', 'Mexico City', 'MX'),
    City('North America', 'New York', 'US'),
    City('South America', 'São Paulo', 'BR'),
]
```

Given those definitions, the following function would return a list of Asian cities:

```
def match_asian_cities():
    results = []
    for city in cities:
        match city:
            case City(continent='Asia'):
                results.append(city)
    return results
```

The pattern `City(continent='Asia')` matches any `City` instance where the `continent` attribute value is equal to `'Asia'`, regardless of the values of the other attributes.

If you want to collect the value of the `country` attribute, you could write:

```
def match_asian_countries():
    results = []
    for city in cities:
            match city:
                case City(continent='Asia', country=cc):
                    results.append(cc)
        return results
```

The pattern `City(continent='Asia', country=cc)` matches the same Asian cities as before, but now the `cc` variable is bound to the `country` attribute of the instance. This also works if the pattern variable is called `country` as well:

```
            match city:
                case City(continent='Asia', country=country):
                    results.append(country)
```

Keyword class patterns are very readable, and work with any class that has public instance attributes, but they are somewhat verbose.

Positional class patterns are more convenient in some cases, but they require explicit support by the class of the subject, as we'll see next.

Positional Class Patterns

Given the definitions from Example 5-22, the following function would return a list of Asian cities, using a positional class pattern:

```
    def match_asian_cities_pos():
        results = []
        for city in cities:
            match city:
                case City('Asia'):
                    results.append(city)
        return results
```

The pattern `City('Asia')` matches any `City` instance where the first attribute value is `'Asia'`, regardless of the values of the other attributes.

If you want to collect the value of the `country` attribute, you could write:

```
    def match_asian_countries_pos():
        results = []
        for city in cities:
            match city:
                case City('Asia', _, country):
                    results.append(country)
        return results
```

The pattern `City('Asia', _, country)` matches the same cities as before, but now the `country` variable is bound to the third attribute of the instance.

I've mentioned "first" or "third" attribute, but what does that really mean?

What makes `City` or any class work with positional patterns is the presence of a special class attribute named `__match_args__`, which the class builders in this chapter automatically create. This is the value of `__match_args__` in the `City` class:

```
>>> City.__match_args__
('continent', 'name', 'country')
```

As you can see, `__match_args__` declares the names of the attributes in the order they will be used in positional patterns.

In "Supporting Positional Pattern Matching" on page 377 we'll write code to define `__match_args__` for a class we'll create without the help of a class builder.

You can combine keyword and positional arguments in a pattern. Some, but not all, of the instance attributes available for matching may be listed in `__match_args__`. Therefore, sometimes you may need to use keyword arguments in addition to positional arguments in a pattern.

Time for a chapter summary.

Chapter Summary

The main topic of this chapter was the data class builders `collections.namedtuple`, `typing.NamedTuple`, and `dataclasses.dataclass`. We saw that each generates data classes from descriptions provided as arguments to a factory function, or from `class` statements with type hints in the case of the latter two. In particular, both named tuple variants produce `tuple` subclasses, adding only the ability to access fields by name, and providing a `_fields` class attribute listing the field names as a tuple of strings.

Next we studied the main features of the three class builders side by side, including how to extract instance data as a `dict`, how to get the names and default values of fields, and how to make a new instance from an existing one.

This prompted our first look into type hints, particularly those used to annotate attributes in a `class` statement, using the notation introduced in Python 3.6 with PEP 526—Syntax for Variable Annotations. Probably the most surprising aspect of type hints in general is the fact that they have no effect at all at runtime. Python remains a dynamic language. External tools, like Mypy, are needed to take advantage of typing information to detect errors via static analysis of the source code. After a basic overview of the syntax from PEP 526, we studied the effect of annotations in a plain class and in classes built by `typing.NamedTuple` and `@dataclass`.

Next, we covered the most commonly used features provided by `@dataclass` and the `default_factory` option of the `dataclasses.field` function. We also looked into the special pseudotype hints `typing.ClassVar` and `dataclasses.InitVar` that are important in the context of data classes. This main topic concluded with an example based on the Dublin Core Schema, which illustrated how to use `dataclasses.fields` to iterate over the attributes of a `Resource` instance in a custom `__repr__`.

Then, we warned against possible abuse of data classes defeating a basic principle of object-oriented programming: data and the functions that touch it should be together in the same class. Classes with no logic may be a sign of misplaced logic.

In the last section, we saw how pattern matching works with subjects that are instances of any class—not just classes built with the class builders presented in this chapter.

Further Reading

Python's standard documentation for the data class builders we covered is very good, and has quite a few small examples.

For `@dataclass` in particular, most of PEP 557—Data Classes was copied into the `dataclasses` module documentation. But PEP 557 has a few very informative sections that were not copied, including "Why not just use namedtuple?", "Why not just use typing.NamedTuple?", and the "Rationale" section, which concludes with this Q&A:

> Where is it not appropriate to use Data Classes?
>
> API compatibility with tuples or dicts is required. Type validation beyond that provided by PEPs 484 and 526 is required, or value validation or conversion is required.
>
> —Eric V. Smith, PEP 557 "Rationale"

Over at *RealPython.com*, Geir Arne Hjelle wrote a very complete "Ultimate guide to data classes in Python 3.7".

At PyCon US 2018, Raymond Hettinger presented "Dataclasses: The code generator to end all code generators" (video).

For more features and advanced functionality, including validation, the *attrs* project led by Hynek Schlawack appeared years before `dataclasses`, and offers more features, promising to "bring back the joy of writing classes by relieving you from the drudgery of implementing object protocols (aka dunder methods)." The influence of *attrs* on `@dataclass` is acknowledged by Eric V. Smith in PEP 557. This probably includes Smith's most important API decision: the use of a class decorator instead of a base class and/or a metaclass to do the job.

Glyph—founder of the Twisted project—wrote an excellent introduction to *attrs* in "The One Python Library Everyone Needs". The *attrs* documentation includes a discussion of alternatives.

Book author, instructor, and mad computer scientist Dave Beazley wrote *cluegen*, yet another data class generator. If you've seen any of Dave's talks, you know he is a master of metaprogramming Python from first principles. So I found it inspiring to learn from the *cluegen README.md* file the concrete use case that motivated him to write an alternative to Python's `@dataclass`, and his philosophy of presenting an approach to solve the problem, in contrast to providing a tool: the tool may be quicker to use at first, but the approach is more flexible and can take you as far as you want to go.

Regarding *data class* as a code smell, the best source I found was Martin Fowler's book *Refactoring*, 2nd ed. This newest version is missing the quote from the epigraph of this chapter, "Data classes are like children…," but otherwise it's the best edition of Fowler's most famous book, particularly for Pythonistas because the examples are in modern JavaScript, which is closer to Python than Java—the language of the first edition.

The website *Refactoring Guru* also has a description of the data class code smell.

Soapbox

The entry for "Guido" in "The Jargon File" is about Guido van Rossum. It says, among other things:

> Mythically, Guido's most important attribute besides Python itself is Guido's time machine, a device he is reputed to possess because of the unnerving frequency with which user requests for new features have been met with the response "I just implemented that last night…"

For the longest time, one of the missing pieces in Python's syntax has been a quick, standard way to declare instance attributes in a class. Many object-oriented languages have that. Here is part of a `Point` class definition in Smalltalk:

```
Object subclass: #Point
    instanceVariableNames: 'x y'
    classVariableNames: ''
    package: 'Kernel-BasicObjects'
```

The second line lists the names of the instance attributes `x` and `y`. If there were class attributes, they would be in the third line.

Python has always offered an easy way to declare class attributes, if they have an initial value. But instance attributes are much more common, and Python coders have been forced to look into the `__init__` method to find them, always afraid that there may be instance attributes created elsewhere in the class—or even created by external functions or methods of other classes.

Now we have `@dataclass`, yay!

But they bring their own problems.

First, when you use `@dataclass`, type hints are not optional. We've been promised for the last seven years, since PEP 484—Type Hints that they would always be optional. Now we have a major new language feature that requires them. If you don't like the whole static typing trend, you may want to use `attrs` instead.

Second, the PEP 526 syntax for annotating instance and class attributes reverses the established convention of `class` statements: everything declared at the top-level of a `class` block was a class attribute (methods are class attributes, too). With PEP 526 and `@dataclass`, any attribute declared at the top level with a type hint becomes an instance attribute:

```
@dataclass
class Spam:
    repeat: int  # instance attribute
```

Here, `repeat` is also an instance attribute:

```
@dataclass
class Spam:
    repeat: int = 99  # instance attribute
```

But if there are no type hints, suddenly you are back in the good old times when declarations at the top level of the class belong to the class only:

```
@dataclass
class Spam:
    repeat = 99  # class attribute!
```

Finally, if you want to annotate that class attribute with a type, you can't use regular types because then it will become an instance attribute. You must resort to that pseudotype `ClassVar` annotation:

```
@dataclass
class Spam:
    repeat: ClassVar[int] = 99  # aargh!
```

Here we are talking about the exception to the exception to the rule. This seems rather unPythonic to me.

I did not take part in the discussions leading to PEP 526 or PEP 557—Data Classes, but here is an alternative syntax that I'd like to see:

```
@dataclass
class HackerClubMember:
    .name: str                                          ❶
    .guests: list = field(default_factory=list)
    .handle: str = ''

    all_handles = set()                                 ❷
```

❶ Instance attributes must be declared with a `.` prefix.

❷ Any attribute name that doesn't have a `.` prefix is a class attribute (as they always have been).

The language grammar would have to change to accept that. I find this quite readable, and it avoids the exception-to-the-exception issue.

I wish I could borrow Guido's time machine to go back to 2017 and sell this idea to the core team.