Data Classes

Mind-Numbing Tedium

```
class GroceryItem:
    '''
    Something sold in a grocery store.
    '''
    def __init__(self, name, size, price, brand, sku):
        self.name = name   # Name of the product
        self.size = size   # In grams
        self.price = price # In USD
        self.brand = brand # Brand name
        self.sku = sku     # "Stock Keeping Unit"

    def label(self):
        return f'{self.name}, {self.size}g - ${self.price:.2f} [{self.sku}]'
```

```
>>> corn = GroceryItem('Whole Kernel Sweet Corn', 432, 1.79,
...                    'Green Giant', '020000103907')
>>> corn.label()
'Whole Kernel Sweet Corn, 432g - $1.79 [020000103907]'
```

See how many times we repeat var names in the constructor. Boring!

Better

```python
from dataclasses import dataclass

@dataclass
class GroceryItem:
    '''
    Something sold in a grocery store.
    '''
    name: str   # Name of the product
    size: int   # In grams
    price: float # In USD
    brand: str   # Brand name
    sku: str   # "Stock Keeping Unit"

    def label(self):
        return f'{self.name}, {self.size}g - ${self.price:.2f} [{self.sku}]'
```

Same Result

This version of `GroceryItem` works the same:

```
>>> # Using the @dataclass version:
>>> corn = GroceryItem('Whole Kernel Sweet Corn', 432, 1.79,
...                    'Green Giant', '020000103907')
>>> corn.label()
'Whole Kernel Sweet Corn, 432g - $1.79 [020000103907]'
```

Same result! But variable names are not repeated in our code.

Bonus: we've documented their types.

How does this all work?

Data Classes

Python 3.7 introduces a welcome feature: the data class.

It's a way to automate certain tasks when you define your classes.

Benefits: Save time and drudgery, easier maintenance, and don't write error-prone code yourself.

Variable Type Hints

Starting in 3.6, you can annotate the variables in your class definitions, giving a hint on the intented type.

```
class Person:
    first: str
    last: str
    age: int

    def __init__(self, first, last, age):
        self.first = first
        self.last = last
        self.age = age
```

This is JUST documentation. That's it.

And that's useful. But by itself, it does nothing else at all.

We'll just cover the parts of this we need. For more info, read PEPs 484 and 526.

Using Type Annotations

The `@dataclass` decorator uses them to create a constructor.

```
# This...
@dataclass
class Person:
    first: str
    last: str
    age: int

# Is EXACTLY EQUIVALENT to this:
class Person:
    def __init__(self, first, last, age):
        self.first = first
        self.last = last
        self.age = age
```

- Order matters.
- The types are ignored.
- BUT WAIT! There's more.

str() and repr()

Python has two different ways to transform an object into a string:

- `str()` - For a user-friendly representation.
- `repr()` - For a machine-friendly representation.

You define two magic methods, `__str__` and `__repr__` for these:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    def __str__(self):
        return f'${self.dollars}.{self.cents:02d}'
    def __repr__(self):
        return f'Money({self.dollars},{self.cents})'
```

str() vs. repr()

```
>>> amount = Money(142,7)
>>>
>>> # For normal humans:
>>> str(amount)
'$142.07'
>>> # Used when printing
>>> print(amount)
$142.07
```

```
>>> # For programmers:
>>> repr(amount)
'Money(142,7)'
>>> # Used in logs, on the interactive prompt, etc.
>>> amount
Money(142,7)
```

Adds a `__repr__`

You get a `__repr__()` for free when you use `@dataclass`:

```
@dataclass
class Money:
    dollars: int
    cents: int
    # You still have to define __str__() yourself.
    # But __init__ and __repr__ are defined for you.
    def __str__(self):
        return f'${self.dollars}.{self.cents:02f}'

amount = Money(142,7)
```

```
>>> repr(amount)
'Money(dollars=142, cents=7)'
```

It's more usefully verbose than the one we made before!

Object Equality

What makes two instances equal to each other?

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents

in_pocket = Money(10, 42)
in_wallet = Money(120, 0)
in_card = Money(10, 42)
```

```
>>> in_pocket == in_wallet
False
>>> in_pocket == in_card
False
```

By default, equality in Python is based on object identity. So two different objects can never be equal.

You can alter that with the `__eq__` method.

__eq__

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    def __eq__(self, other):
        return self.dollars == other.dollars and self.cents == other.cents

in_pocket = Money(10, 42)
in_wallet = Money(120, 0)
in_card = Money(10, 42)
```

```
>>> in_pocket == in_wallet
False
>>> in_card == in_pocket
True
```

Better! More sensible.

Gives you `__eq__`

`@dataclass` gives you an `__eq__` method for free, too!

```
@dataclass
class Money:
    dollars: int
    cents: int
    # Just define __str__. You get __init__, __eq__ and __repr__ for free.
    def __str__(self):
        return f'${self.dollars}.{self.cents:02f}'

in_pocket = Money(10, 42)
in_wallet = Money(120, 0)
in_card = Money(10, 42)
```

```
>>> in_pocket == in_wallet
False
>>> in_card == in_pocket
True
```

Notice "==" behaves sensibly now.

Overriding Constructors

`@dataclass` only creates the constructor if you don't.

```
@dataclass
class Money:
    dollars: int
    cents: int
    def __init__(self, dollars, cents):
        # When you want to create your own constructor,
        # but receive the other benefits of using a data class.
        if cents > 100:
            dollars += cents // 100
            cents = cents % 100
        self.dollars = dollars
        self.cents = cents
```

Since you applied `@dataclass`, you get a `__repr__()` for free:

```
>>> Money(10, 273)
Money(dollars=12, cents=73)
```

Great! But there's an even better way to do this.

Post-init hook

You can add to the default constructor, instead of replacing it. Just define `__post_init__()`:

```
@dataclass
class Money:
    dollars: int
    cents: int

    def __post_init__(self):
        if self.cents > 100:
            self.dollars += self.cents // 100
            self.cents = self.cents % 100
```

This is run automatically, AFTER the normally-generated dataclass constructor.

```
>>> Money(10, 273)
Money(dollars=12, cents=73)
```

This is almost always a better choice, when practical.

Comparable

By default, Python class instances are NOT comparable (in 3.x):

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
```

```
>>> x = Money(20, 5)
>>> y = Money(10, 50)
>>> x > y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'Money' and 'Money'
```

(It's good this is the default: it makes certain horrible bugs impossible.)

But you can make your classes comparable by defining certain magic methods, similar to `__eq__`.

Ordering Methods

There are "magic methods" you can define for each comparison type:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    # for "greater than"
    def __gt__(self, other):
        return self.dollars > other.dollars and self.cents > other.cents
    # for "less than or equal"
    def __le__(self, other):
        return self.dollars <= other.dollars and self.cents <= other.cents
    # And so on for __ge__ (>=) and __lt__ (<)
```

Comparing

```
>>> x = Money(12, 73)
>>> y = Money(42, 1)
>>> z = Money(12, 73)
>>>
>>> # Triggers z.__le__(x)
>>> z <= x
True
>>> # Triggers z.__gt__(y)
>>> z > y
False
```

Dataclasses can be comparable

You might be expecting `@dataclass` to define these for you...

But it does NOT. Not by default.

But it will, if you pass in the option `order=True`:

```
@dataclass(order=True)
class Money:
    dollars: int
    cents: int
    # No methods needed!
```

```
>>> x = Money(12, 73)
>>> y = Money(42, 1)
>>> z = Money(12, 73)
>>>
>>> z <= x
True
>>> z > y
False
```

Comparing Tuples

```
# This:
@dataclass(order=True)
class Money:
    dollars: int
    cents: int

# ... is similar to this:

@dataclass
class Money:
    dollars: int
    cents: int
    def __gt__(self, other):
        left = (self.dollars, self.cents)
        right = (other.dollars, other.cents)
        return left > right
    # And likewise for __lt__, etc.
```

Why not on by default?

Basically, it would cause too many monstrous bugs.

You are required to declare that you want it. For many types, the idea of comparisons won't make sense:

```
from dataclasses import dataclass

@dataclass(order=True)
class Point:
    latitude: int
    longitude: int
```

```
>>> p1 = Point(40.768115, -73.960293)
>>> p2 = Point(39.898382, 116.572546)
>>> # It really would be better if this raised an error:
>>> p1 > p2
True
```

Frozen

By default, `@dataclass` creates regular member variables:

```
>>> pnt = Point(39.898382, 116.572546)
>>> pnt.latitude = 112
>>> pnt
Point(latitude=112, longitude=116.572546)
```

If you want them to be immutable, pass in `frozen=True`:

```
@dataclass(order=True, frozen=True)
class FrozenPoint:
    latitude: int
    longitude: int
```

Frozen == Immutable

Then variables can be initially set... but Python will then raise an error if code attempts to change them.

```
>>> pnt = FrozenPoint(39.898382, 116.572546)
>>> pnt.latitude = 112
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 3, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 'latitude'
>>> pnt
FrozenPoint(latitude=39.898382, longitude=116.572546)
```

Options

Option	Default	Description
repr	True	Create `__repr__`
eq	True	Create `__eq__`
order	False	Create comparison methods
frozen	False	Make fields immutable
init	True	Create `__init__`
unsafe_hash	False	Force auto-generated `__hash__`

- `init=False` is only if you want NO constructor. You'll more often define your own `__init__` or use `__post_init__` instead.
- If you think you need `__hash__` created, carefully read its section in the docs. There are hazards.

No such thing as "a data class"

Strictly speaking... there's no such thing as a data class. Just regular classes you use `@dataclass` to create.

We sometimes call that decorated class a "data class"...

But it's just a normal class.

All `@dataclass` does is automate creating useful methods, so you don't have to. But you could write those methods yourself, and get the exact same class.

Adding Methods

Remember you can add any method you want to a data class.

```
import math
@dataclass(order=True)
class Point:
    latitude: int
    longitude: int

    def distance_to(self, other_point):
        return math.sqrt(
            (self.latitude - other_point.latitude) ** 2
            + (self.longitude - other_point.longitude) ** 2)
```

```
>>> p1 = Point(40.768115, -73.960293)
>>> p2 = Point(39.898382, 116.572546)
>>> p1.distance_to(p2)
190.53482404246006
```

Even add class and static methods. Because a "data class" is just a class.

Default Values

When you define your fields, you can specify a default:

```
@dataclass
class Money:
    dollars: int
    cents: int = 0
```

```
>>> Money(10)
Money(dollars=10, cents=0)
>>> Money(20, 1)
Money(dollars=20, cents=1)
```

Essentially equivalent to this:

```
class Money:
    def __init__(self, dollars, cents=0):
        self.dollars = dollars
        self.cents = cents
```

Fields

With data classes, each line that defines a member variable is called a "field".

```
@dataclass
class Money:
    dollars: int
    cents: int = 0
```

This lets you set the name of the variable; document its type; and, optionally, set a default value.

To do more than this, you can use a function called `field()`.

dataclasses.field()

The `field()` function lets you express more information.

Call this function, and assign to the field:

```
from dataclasses import dataclass, field

@dataclass
class Money:
    # Like "dollars: int"
    dollars: int = field()
    # Like "cents: int = 0"
    cents: int = field(default=0)
```

```
>>> Money(10)
Money(dollars=10, cents=0)
>>> Money(20, 1)
Money(dollars=20, cents=1)
```

So far, just like on previous slide.

Interlude: Default Collections

Look at this non-data class:

```
class Department:
    def __init__(self, name, code, inventory=[]):
        self.name = name
        self.code = code
        self.inventory = inventory
```

There's a hidden trap. Let's run some code:

```
produce_dept = Department('Produce', 14)
auto_dept = Department('Automotive', 71)

produce_dept.inventory.append('carrots')
produce_dept.inventory.append('pineapple')
auto_dept.inventory.append('wiper blade')
auto_dept.inventory.append('tire cleaner')
```

What's the trap? Where's the bug?

Empty List

The `inventory=[]` creates an empty list, at byte compile time, tied to the class.

So EVERY INSTANCE gets the same list!

```
>>> produce_dept.inventory
['carrots', 'pineapple', 'wiper blade', 'tire cleaner']
>>> auto_dept.inventory
['carrots', 'pineapple', 'wiper blade', 'tire cleaner']
```

Workaround

The common workaround: use a sentinal, and create a fresh list in the constructor.

```
class Department:
    def __init__(self, name, code, inventory=None):
        if inventory is None:
            # creates separate list each time ctor is called
            inventory = []
        self.name = name
        self.code = code
        self.inventory = inventory
```

What's the equivalent data class?

Not Allowed

To protect against this trap, you can't create a data class with a field initialized in this way:

```
>>> @dataclass
... class Department:
...     name: str
...     code: int
...     inventory: list = []
...
Traceback (most recent call last):
    ...
    raise ValueError(f'mutable default {type(f.default)} for field '
ValueError: mutable default <class 'list'> for field inventory is not
allowed: use default_factory
```

See the error message? It says to use `default_factory`.

Default Factories

For collections and certain other types, you will need to use `field()` and the `default_factory` parameter.

```
@dataclass
class Department:
    name: str
    code: int
    inventory: list = field(default_factory=list)
```

This defines a zero-argument callable to initialize the field. Similar to:

```
class Department:
    def __init__(self, name, code, inventory = None):
        if inventory == None:
            inventory = list()
        # Then set the fields...
```

Other `field()` arguments

Param	Description
`default_factory`	Generate default value
`default`	Literal default value
`init`	Whether to include in `__init__()`
`repr`	Whether to include in `__repr__()`
`compare`	Whether to include in comp methods
`hash`	Controls use in `__hash__()`
`metadata`	3rd party extention

- You mainly need to know about `default_factory`; the rest, you're not as likely to need.
- Don't confuse `field()` with `fields()` - you'll rarely need the second one.

Learning more

The examples I've shown you demonstrate 95% of what you're likely to need.

Most important next step: start using dataclasses. You'll quickly learn a lot by doing.

More information and reference:

- `dataclasses` module docs - https://docs.python.org/3/library/dataclasses/
- PEP 557 - "Data Classes" - https://www.python.org/dev/peps/pep-0557/