GERARD KEATING

PRESENTS

DATACLASSES:

WHAT DO THEY DO?

DO THEY DO THINGS?

LET'S FIND OUT!

A simple class

Create a simple user object with username and email

```
class User:
    def __init__(self, username, email):
        self.username = username
        self.email = email
User(username="ger", email="ger@example.com")
```

But what's the problem

- Printing it out looks awful

```
print(User(username="ger", email="ger@example.com"))
# <__main__.User object at 0x1103bb230>
```

- No equality

```
user =      User(username="ger", email="ger@example.com")
print(user==User(username="ger", email="ger@example.com"))
# False
```

Adding dunder methods

```
class User:
    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __repr__(self):
        return f"User(username={self.username!r}, email={self.email!r})"
        # User(username='ger', email='ger@example.com')
    def __eq__(self, other):
        if not isinstance(other, User):
            return False
        return (self.username, self.email) == (other.username, other.email)
```

Updating the class

Add a real_name attribute

```
class User:
    def __init__(self, username, email, real_name):
        self.username = username
        self.email = email
        self.real_name = real_name

    def __repr__(self):
        return f"User(username={self.username!r}, email={self.email!r}, real_name={self.real_name!r})"

    def __eq__(self, other):
        if not isinstance(other, User):
            return False
        return (self.username, self.email, self.real_name) == (other.username, other.email, other.real_name)
```

- So adding one more variable lead to changes in three functions not including tests

Solution dataclass

- Added in Python 3.7 PEP 557

```
from dataclasses import dataclass

@dataclass
class User:
    username: str
    email: str
    real_name: str
```

Testing out dataclasses

```
from dataclasses import dataclass

@dataclass
class User:
    username: str
    email: str
    real_name: str

user = User(username="ger", email="ger@example.com", real_name="Gerard Keating")
print(user)
# User(username='ger', email='ger@example.com', real_name='Gerard Keating')
print(user==User(username="ger", email="ger@example.com", real_name="Gerard Keating"))
# True
```

Breaking it Down

```
from dataclasses import dataclass
```

A class decorator

```
@dataclass
class User:
```

Type Annotation

```
    username: str
    email: str
```

Class Decorator

- Been in Python since Python 3.0
- Does not interfere with inheritance, nothing to do with metaclasses
- Simple to use and implement

```
def class_dec(cls):
    cls.foo = 42
    return cls

@class_dec
class MyClass:
    pass

print(MyClass.foo)  # 42
```

Type Annotation

```
class User:
    username: str
    email: str

print(User.__annotations__)
# {'username': <class 'str'>, 'email': <class 'str'>}
```

- Added in Python 3.6
- **Dataclass will not check your types** they just read annotations. Use a type checker (mypy/pyright) if you want enforcement.
- See my last PyconIE talk in 2023 *How I Learned to Stop Worrying and Love Type Annotation*

Default values (immutable)

```
from dataclasses import dataclass

@dataclass
class User:
    username: str
    email: str
    #Needs to be defined after the non default variables
    is_admin: bool = False

print(User('Ger', 'ger@example.com'))
# User(username='Ger', email='ger@example.com', is_admin=False)
```

Default value that is mutable

- When adding a mutable object (like a list, dict, set ...)) on the class, it’s shared by all instances

```
class Foo:
    bar: list = []

f = Foo()
f.bar.append('value')
print(f.bar) # ['value']
print(Foo.bar) # ['value']
b = Foo()
print(b.bar) # ['value']
```

- This can be useful for some user cases like having a registry but is usually a gotcha

default_factory

- Solution to having an mutable default value

  ```
  tags: list = dataclasses.field(default_factory=list)
  ```

- dataclasses.field lets you customize how a dataclass field behaves
- A default_factory is a callable that generates a fresh value per instance
- You can still have a class variable by using `typing.ClassVar`

Simple default factory code example

```
from dataclasses import dataclass, field
from typing import ClassVar

@dataclass
class User:
    username: str
    email: str
    tags: list[str] = field(default_factory=list)

    all_users_tags: ClassVar[list[str]] = []

print(User(username="ger", email="ger@example.com"))
# User(username='ger', email='ger@example.com', tags=[])
```

What else can default factories do

- Let's say you want to create a uid for your user class
- Use default factory
- the `init=False` in the field means it doesn't turn up as an argument to your class init

```
from dataclasses import dataclass, field
from uuid import uuid4, UUID

@dataclass
class User:
    username: str
    email: str
    uid: UUID = field(init=False, default_factory=uuid4)

print(User('Ger', 'ger@example.com'))
# User(username='Ger', email='ger@example.com', uid=UUID('ddc36e5e-479d-4482-a56d-2dd38d903997'))
```

post_init

- Problem: You want to store your username in lower case
- Solution: You can have a `__post_init__` which is run after the `__init__`

```
@dataclass
class User:
    username: str
    email: str

    def __post_init__(self):
        self.username = self.username.lower()
print(User('GER', 'ger@example.com'))
# User(username='ger', email='ger@example.com')
```

dataclass.InitVar

- Problem: You want to pass a raw password to your user class but only store a hash
- Solution: `InitVar` is a value that the user can pass to the `__init__` of your dataclass
- It’s forwarded to `__post_init__`
- but not stored on the instance

dataclass.InitVar example

```python
from dataclasses import dataclass, InitVar, field

@dataclass
class User:
    username: str
    email: str

    raw_password: InitVar[str]
    # init False means this field is not in the __init__ signature
    hashed_password: str = field(init=False)


    def __post_init__(self, raw_password: str):
        # just an example, not the way to hash passwords
        self.hashed_password = str(hash(raw_password))

user = User('Ger', 'ger@example.com' , 'password123')
print(user)
# User(username='Ger', email='ger@example.com', hashed_password='8700902137681039529')
print(user.raw_password)
# AttributeError: 'User' object has no attribute 'raw_password'
```

Ordering

- Problem: You want to sort your user instances
- Solution: You can add `order=True` which adds all comparisons `__lt__`, `__ge__` etc.
- Can add `compare=False` to field to not include in comparison

```
from dataclasses import dataclass, field

@dataclass(order=True)
class User:
    username: str
    email: str = field(compare=False)
users = [User('Ger1', 'ger1@example.com'), User('Ger', 'ger@example.com')]
print(sorted(users))
# [User(username='Ger', email='ger@example.com'), User(username='Ger1', email='ger1@example.com')]
```

Immutability

- Problem: Want to have User be a key in a dictionary
- Solution: use the frozen keyword which makes the object immutable

```
from dataclasses import dataclass, field

@dataclass(frozen=True)
class User:
    username: str
    email: str

frozen_user = User('Elsa', 'elsa@example.com')
users = {frozen_user: ['some info']}
frozen_user.username = 'Anna'
# dataclasses.FrozenInstanceError: cannot assign to field 'username'
```

slots

- You are worried about how much memory your dataclass is using
- Solution `slots=true` which Eliminates per-instance `__dict__`

```
from dataclasses import dataclass

@dataclass(slots=True)
class User:
    username: str
    email: str
```

- `User` instance with `slots=False` is 552 bytes
- `User` instance with `slots=True` is 152 bytes

Convert to dict or tuple

```
from dataclasses import dataclass, asdict, astuple

@dataclass
class User:
    username: str
    email: str
user = User('Ger', 'ger@example.com')
print(asdict(user))
# {'username': 'Ger', 'email': 'ger@example.com'}
print(astuple(user))
# ('Ger', 'ger@example.com')
```

Alternative to dataclasses: typing.NamedTuple

- A NamedTuple is an (immutable) tuple subclass with field names and type annotations
- Has built-in readable `__repr__` and equality
- Does not have all the features of a dataclass like per field attributes, mutable defaults, default factories etc.

Alternative to dataclasses: typing.NamedTuple

```
from typing import NamedTuple

class User(NamedTuple):
    username: str
    email: str

user = User('ger', 'gerard@example.com')
print(user)
# User(username='ger', email='gerard@example.com')
print(user==User('ger', 'gerard@example.com'))
# True
```

Alternative to dataclasses: attrs

- attrs is not part of the standard library
- attrs was started in 2016, actually mentioned in the original Dataclass PEP

```
import attr # version 25.4.0

@attr.define
class User:
    username: str
    email: str

user1 = User('Ger', 'ger@example.com')
print(user1) # User(username='Ger', email='ger@example.com')
print(user1==User(username='Ger', email='ger@example.com'))
# True
```

attrs

Validation

attrs has built-in validators (`instance_of`, `min_len`, etc.) and lets you write your own

Converters

Automatically coerce values on init (e.g. age=str → int)

Ecosystem

Many tools builtin on top of attrs such as cattrs, attrs-strict, attrs-mate, hypothesis-attrs, attrs-jsonschema

Criticisms of Dataclasses

You should not have just dumb classes for storing data

- Agree and dataclasses can be used for so much more than just dumb data stores

They are too slow

- If raw speed is a major concern, you might want specialised tools (NumPy, Rust extensions, etc.) or libraries like attrs/pydantic that are tuned for particular workloads

They don't have *X* feature

- They are part of the standard library and can't come with every feature so either implement the feature you need or find a package that does

Why dataclasses

- Removes boilerplate code
- Makes for clearer code
- Easily adds advanced features (slots, ordering etc.)

Q&A

Thank you

gerardkeating.me

Slides will be up soon

@vfxGer