

Oh Python!

Where Did We Go Wrong?

Greg Ward <greg@gerg.ca>

ConFoo 2025, Montreal

27 Feb 2025



Once
upon
a
time...

There was a little programming language...

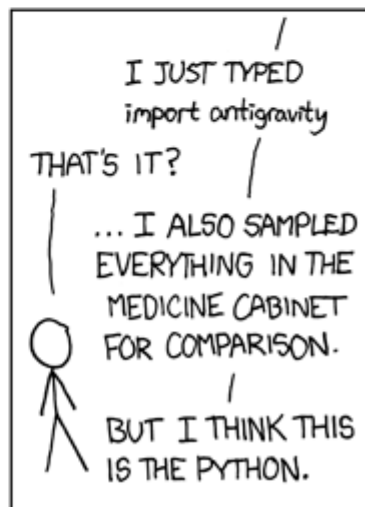
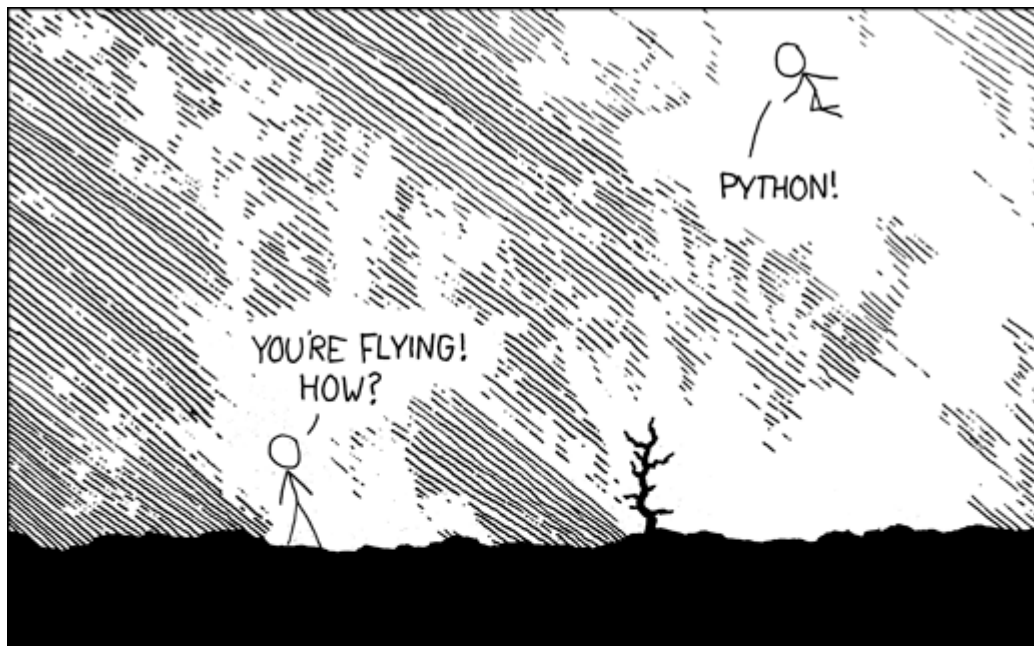
(we're talking late 90s, early 2000s here)

- it was simple
- it was straightforward
- it was *readable*
- “it's like executable pseudocode!” (said someone)

Python's secret weapons

- **concise:** much less typing than Java or C++
- **consistent:** == to compare, + to add—no need to learn different operators for different types
- **clear:** the code says what it does and does what it says

**The zeitgeist circa 2007
(when things were really heating up)**



Nobody's perfect

The perfect programming language does not exist! Even back in the day, Python had some tricky features...

- unconstrained operator overloading
- easy metaprogramming
- implicit execution at import time
- subclassing

My, how they grow up!

- closures
- generators
- iteration protocol
- decorators
- metaclasses
- properties
- list comprehensions
- dict/set comprehensions

And the community has innovated

- frameworks 😇
- frameworks on top of frameworks 😬
- serialization/validation libraries 👍
- “active record” ORMs (Django, SQLAlchemy) 👍👎

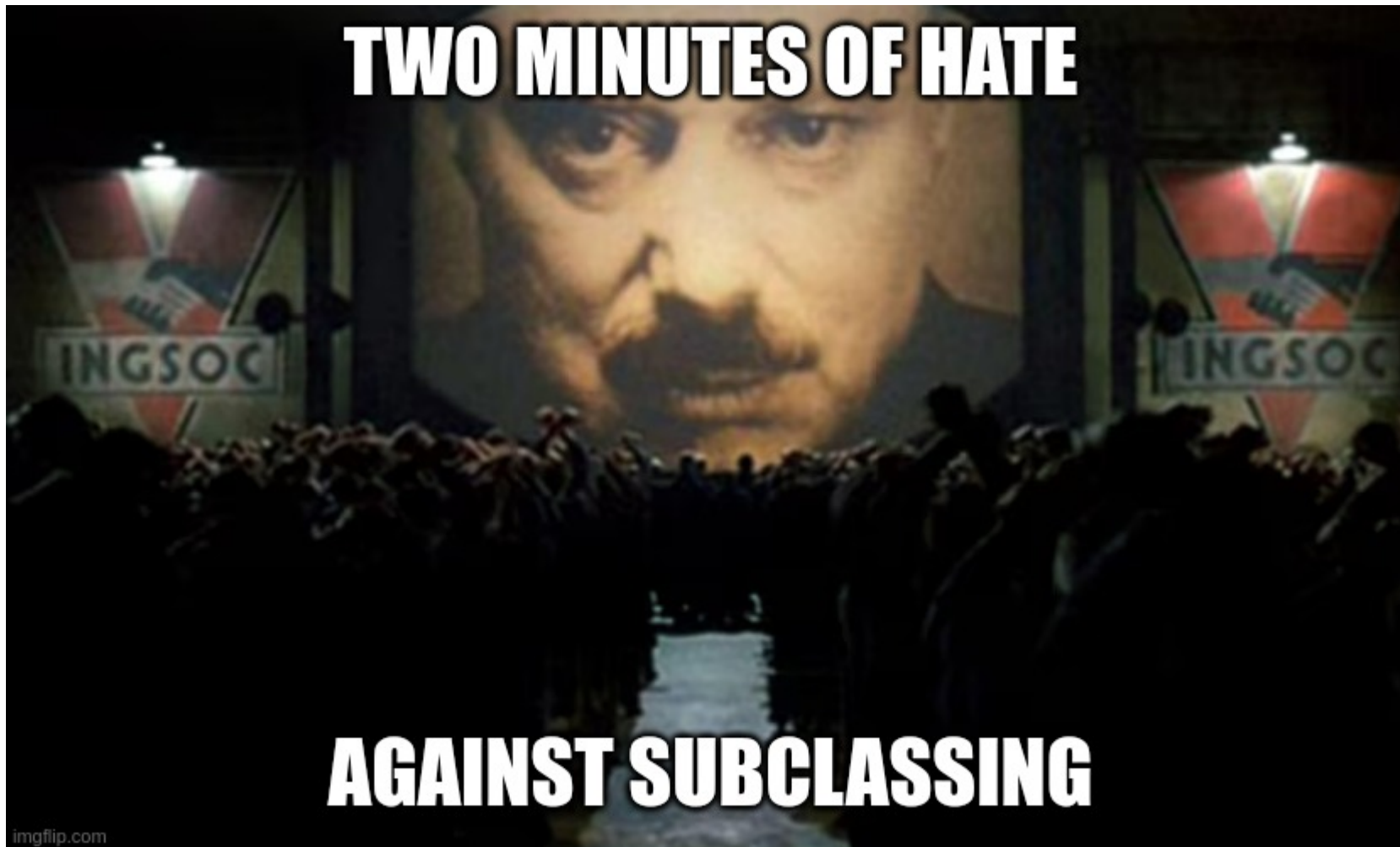
So, what's the problem?

- Are ALL of these bad features?
- Are ANY of these bad features?
- Or ... is it maybe about combining them?

So, what's the problem?

- Are ALL of these bad features? **NO**
- Are ANY of these bad features? **IMHO yes**
- Or ... is it maybe about combining them? **YES**

A brief digression to indulge in...



The problem with subclassing

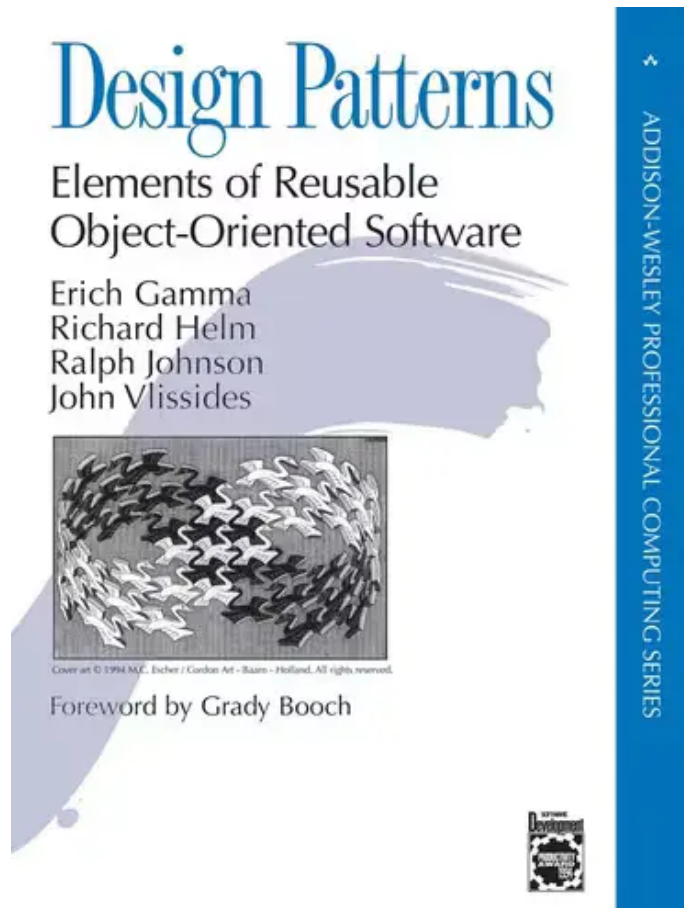
```
# package lcpf: a large, complex, published framework
class BaseThing:
    def action(self, a, b):
        self.pre_action()
        ...do some stuff...
        self.post_action()

    def pre_action(self, a):
        ...default implementation...

# package myapp: my little app using lcpf
class MyThing(BaseThing):
    def pre_action(self):
        super().pre_action()
        ...prepare for action...
```

This is NOT a new idea

These four authors figured it out by 1994:



Prefer composition over inheritance.

The answer to subclassing?

simple version: just... don't

nuanced version: it might be OK if

- you're using a library or framework that forces subclassing
- you're using it *solely* for code reuse, *not* for type polymorphism

...but really, you should pretend we live in a world where the Evil Overlord has banned all subclassing

</digression>

(from here on out, it's all nuance and subtlety)

Case Study 1: A Simple Web API

List, get, or create foobars

```
GET /foobar
{
  "foobars": [
    {"id": 2, "name": "ted", "colour": "blue", "num_eyes": 3},
    {"id": 1, "name": "jan", "colour": "green", "num_eyes": 6},
    {"id": 3, "name": "ann", "colour": "red", "num_eyes": 1}
  ]
}
```

```
GET /foobar/3
{"id": 3, "name": "ann", "colour": "red", "num_eyes": 1}
```

```
POST /foobar
{"name": "dan", "colour": "blue", "num_eyes": 2}
```

Implementation: model class

```
class Foobar:
    """Model object for a small furry creature

    Foobars come in various colours, with variable number of eyes.
    """
    id: t.Optional[int]
    name: str
    colour: str
    num_eyes: int

    def __init__(self, name: str, colour: str, num_eyes: int):
        self.name = name
        self.colour = colour
        self.num_eyes = num_eyes

    def as_dict(self) -> dict[str, t.Any]:
        return vars(self)
```

Implementation: list (Flask)

```
@app.get("/foobar")
def list_foobars():
    foobars = foobar_db.list()
    resp = {"foobars": [foobar.as_dict() for foobar in foobars]}
    return (resp, 200)
```

Implementation: get (Flask)

```
@app.get("/foobar/<int:foobar_id>")
def get_foobar(foobar_id: int):
    try:
        foobar = foobar_db.get(foobar_id)
    except db.NotFoundError:
        raise werkzeug.exceptions.NotFound()
    return (foobar.as_dict(), 200)
```

Implementation: create (Flask)

```
@app.post("/foobar")
def create_foobar():
    req = flask.request
    body = req.json
    foobar = common.Foobar(
        name=body["name"],
        colour=body["colour"],
        num_eyes=body["num_eyes"],
    )
    foobar_db.create(foobar)
    return (foobar.as_dict(), 201)
```

Flaws in this implementation

- Model class specifies fields twice: type hints and constructor
- Handling “not found” error like this will be a pain
- Input validation is incomplete and badly done
- Error responses are HTML, not JSON (Flask default)

I know! Let's use *two* frameworks

- ...this is a RESTful API
- ...using Flask
- ...so "Flask-RESTful" will solve all my problems?

Implementation: get (Flask-RESTful)

```
class FoobarResource(flask_restful.Resource):  
    def get_foobar(self, foobar_id: int):  
        try:  
            foobar = foobar_db.get(foobar_id)  
        except db.NotFoundError:  
            raise werkzeug.exceptions.NotFound()  
        return (foobar.as_dict(), 200)
```

- why do I need a class? where's the state?
- where's the URL? how do I connect this with HTTP requests?

Implementation: list (Flask-RESTful)

```
class FoobarListResource(flask_restful.Resource):  
    def get(self):  
        foobars = foobar_db.list()  
        resp = {"foobars": [foobar.as_dict() for foobar in foobars]}  
        return (resp, 200)  
  
[...class FoobarListResource continues on next slide...]
```

Implementation: create (Flask-RESTful)

```
[...class FoobarListResource continued...]
    def post(self):
        req = flask.request
        body = req.json
        foobar = common.Foobar(
            name=body["name"],
            colour=body["colour"],
            num_eyes=body["num_eyes"],
        )
        foobar_db.create(foobar)
        return (foobar.as_dict(), 201)
```

Flaws in this implementation

- Model class specifies fields twice: type hints and constructor
- Handling “not found” error like this will be a pain
- Input validation is incomplete and badly done
- ~~Error responses are HTML, not JSON (Flask default)~~
- More code to achieve the same result
- One resource type, three endpoints ... but *two* classes?
- Loss of explicit URL paths (they're now implicit)

The case for Flask-RESTful

- You probably have many resource types: users, groups, things, foobars
- Similar endpoints have similar implementations:
 - GET: query database, return result
 - POST: parse body, update database, return new data
- Surely you can factor that commonality out and put it in a base class?

The flaw in this reasoning

- No real-world API service stays this simple for long
- There's always business logic between the database and the API
- Thus, layered design: database on bottom, business logic in the middle, API on top
- Very little opportunity for code sharing in the API layer
- Finally... The Evil Overlord wants to ban base classes and subclassing

Decorators

Gone

Mild

Everybody loves this, right?

```
@app.post("/foobar")
def create_foobar():
    [...]

@app.get("/foobar/<int:foobar_id>")
def get_foobar(foobar_id: int):
    [...]
```


Recap: what is a decorator, again?

```
def create_foobar():  
    [...]  
create_foobar = app.post(create_foobar, "/foobar")  
  
def get_foobar(foobar_id: int):  
    [...]  
get_foobar = app.get(get_foobar, "/foobar/<int:foobar_id>")
```

Decorators are syntactic sugar for something you could do anyways!

Decorators: what are they good for?

- good: declare, record, register, announce
- iffy: inject arguments
- bad: replace a single line of code
- bad: replace return value

Decorators
Gone
Wild

If 1 decorator is good, then 2 must be better...

what does *this* do?

```
@app.post("/")
@authn_require(auth.API_AUTH)
@authz_require(PERM_WRITE_FOOBAR)
@need_feature(FoobarEnabled)
@parse_with(FoobarSchema, arg_name="foobar")
@serialize_with(FoobarSchema)
@handle_errors
def create_foobar(foobar):
    [...]
```

Why are these even decorators?

why not this?

```
@app.post("/")
@parse_with(FoobarSchema, arg_name="foobar")
@serialize_with(FoobarSchema)
@handle_errors
def create_foobar(foobar):
    # each one succeeds silently or raises an exception
    authn_require(auth.API_AUTH)
    authz_require(PERM_WRITE_FOOBAR)
    need_feature(FoobarEnabled)
    [...]
```

Fun fact #1: functions can return a value

```
@app.post("/")
@handle_errors
def create_foobar(foobar):
    # each one succeeds silently or raises an exception
    authn.require(auth.API_AUTH)
    authz.require(PERM_WRITE_FOOBAR)
    restrict_endpoint(FoobarEnabled)





    foobar = parse_with(FoobarSchema, flask.request)
    [...validate and save the foobar...]
    return serialize_with(FoobarSchema, foobar)
```

Fun fact #2: Flask can handle errors

```
@app.post("/")
def create_foobar(foobar):
    # each one succeeds silently or raises an exception
    authn.require(auth.API_AUTH)
    authz.require(PERM_WRITE_FOOBAR)
    restrict_endpoint(FoobarEnabled)

    foobar = parse_with(FoobarSchema, flask.request)
    [...validate and save the foobar...]
    return serialize_with(FoobarSchema, foobar)
```

Laundry list of possible topics

- unconstrained operator overloading
- easy metaprogramming
- implicit execution at import time
- subclassing 
- closures
- generators
- iteration protocol
- decorators 
- metaclasses
- properties
- list comprehensions
- dict/set comprehensions
- frameworks 
- frameworks on frameworks 
- serialization/validation libraries
- “active record” ORMs

This is what we've forgotten

Code is written once, and read many times

- Strive for readability! clarity, consistency above all
- Conciseness is nice, but don't go overboard
- Saving 10 minutes in the writing is not worth it when it costs 30 minutes to understand

Final thought

- Does your code do what it says?
- Does your code say what it does?
- If not ... I suggest you rethink your approach

Thanks!



5 / 50

Go to slide: