Data-Oriented Django Deux

Adam Johnson

Most of your code does not exist

Most of your code does not exist at runtime

Most of your code does not exist at runtime in the same form

```
def funk():
    if x:
        return 1
    elif y:
        return 2
    else:
        return 3
```

Disassembled

```
>>> dis.dis(funk)
...
2           2 LOAD_FAST                0 (x)
            4 POP_JUMP_IF_FALSE        1 (to 8)

3           6 RETURN_CONST             1 (1)

4     >>    8 LOAD_FAST                1 (y)
           10 POP_JUMP_IF_FALSE        1 (to 14)

5          12 RETURN_CONST             2 (2)
...
```

Disassembled

```
>>> dis.dis(funk)
...
2           2 LOAD_FAST                0 (x)
            4 POP_JUMP_IF_FALSE        1 (to 8)

3           6 RETURN_CONST             1 (1)

4     >>    8 LOAD_FAST                1 (y)
           10 POP_JUMP_IF_FALSE        1 (to 14)

5          12 RETURN_CONST             2 (2)
...
```

Disassembled

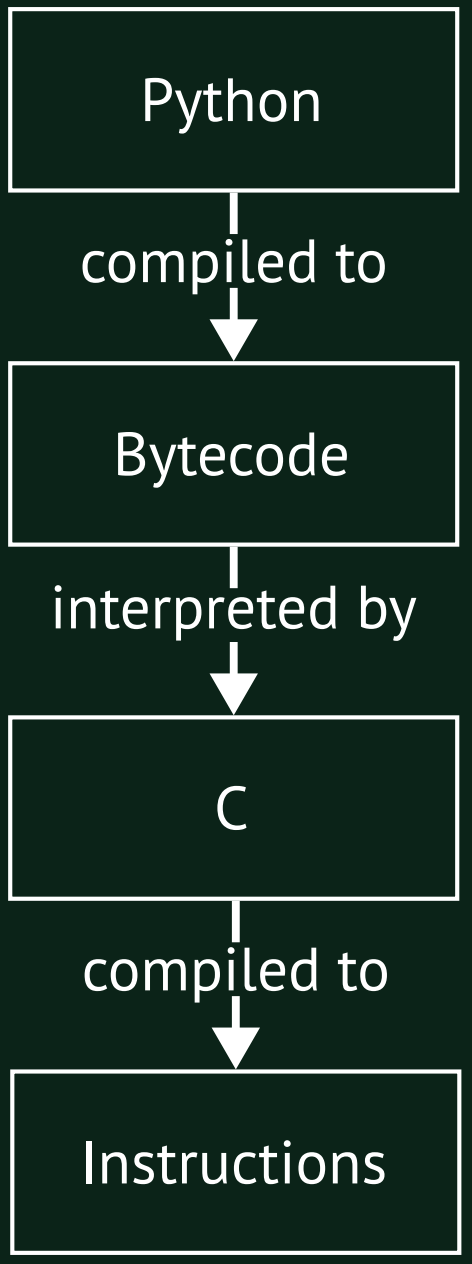
```
>>> dis.dis(funk)
...
2           2 LOAD_FAST                0 (x)
            4 POP_JUMP_IF_FALSE        1 (to 8)

3           6 RETURN_CONST             1 (1)

4     >>    8 LOAD_FAST                1 (y)
           10 POP_JUMP_IF_FALSE        1 (to 14)

5          12 RETURN_CONST             2 (2)
...
```

`if`, `elif`, and `else` gone!


Python
compiled to
Bytecode
interpreted by
C
compiled to
Instructions

Things that don’t exist in a processor

Things that don’t exist in a processor

Functions

Things that don’t exist in a processor

Functions

Classes

Things that don’t exist in a processor

Functions

Classes

Objects

Things that don’t exist in a processor

Functions

Classes

Objects

Programs

Things that do exist in a processor

Things that do exist in a processor

1

Things that do exist in a processor

1

0

Things that do exist in a processor

Things that do exist in a processor

Instructions

Things that do exist in a processor

Instructions

Data

Instruction optimizations

Constant folding	JIT compiling
Loop unrolling	Tail-call optimization
Inlining	Register allocation
Instruction scheduling	Parallel execution

...

Wikipedia: Optimizing Compiler

Data optimizations

🤷 struct packing

Generally, compilers do not rearrange data.

Most of your code does not exist at runtime in the same form.

Most of your code does not exist at runtime

in the same form.

But your data does.

Data-Oriented Design

Data-Oriented Design

Focus on the data.

Everything else is secondary.

Three data considerations

1. Layout
2. Batching
3. Statistical distribution

1. Layout

Problem: find mean of 10,000 2D points

OOP / Row-oriented

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

from random import random
points = [
    Point(random(), random())
    for _ in range(10_000)
]
```

```
def mean_point():
    return Point(
        sum(p.x for p in points) / len(points),
        sum(p.y for p in points) / len(points),
    )
```

```
In [1]: %timeit mean_point()
499 µs ± 5.61 µs per loop (mean ± std. dev.)
```

Column-oriented

```
xs = []
ys = []

from random import random
for _ in range(10_000):
    xs.append(random())
    ys.append(random())
```

```
def mean_point2():
    return (
        sum(xs) / len(xs),
        sum(ys) / len(ys),
    )
```

```
In [2]: %timeit mean_point2()
54.9 µs ± 506 ns per loop (mean ± std. dev.)
```

10× faster

Column-oriented arrays

```
import polars as pl


# Generate 10,000 random 2D points
points_dataframe = pl.DataFrame({
    'x': [random() for _ in range(10_000)],
    'y': [random() for _ in range(10_000)]
})
```

```
def mean_point3():
    return (
        points_dataframe['x'].mean(),
        points_dataframe['y'].mean(),
    )
```

```
In [3]: %timeit mean_point3()
3.11 µs ± 15.1 ns per loop (mean ± std. dev.)
```

~200× faster

Method	Speed
Row-oriented	500 µs
Column-oriented	50 µs
Column-oriented arrays	3 µs

Method	Speed
Row-oriented	500 µs
Column-oriented	50 µs
Column-oriented arrays	3 µs

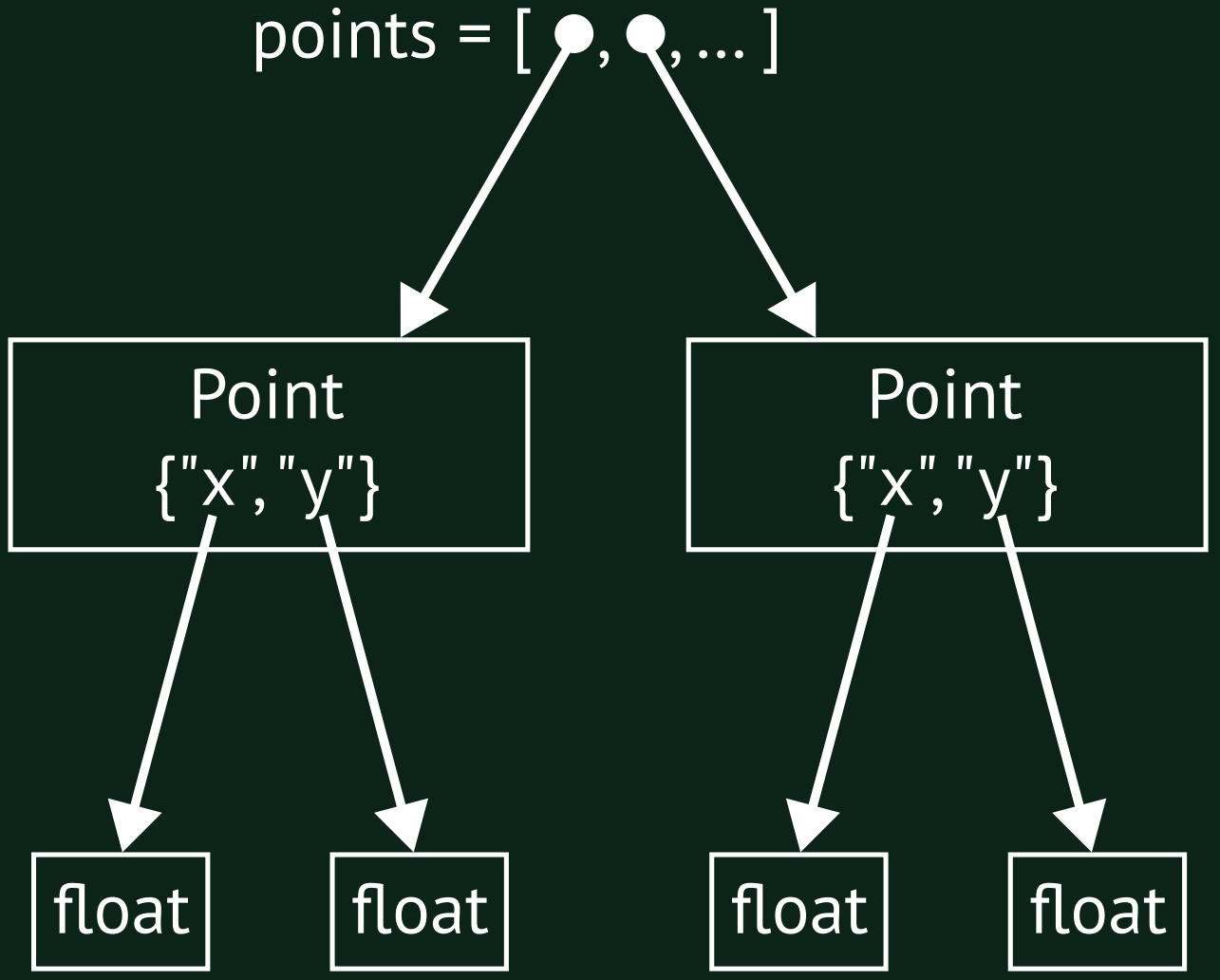
~200× faster

Method	Speed
Row-oriented	500 µs
Column-oriented	50 µs
Column-oriented arrays	3 µs

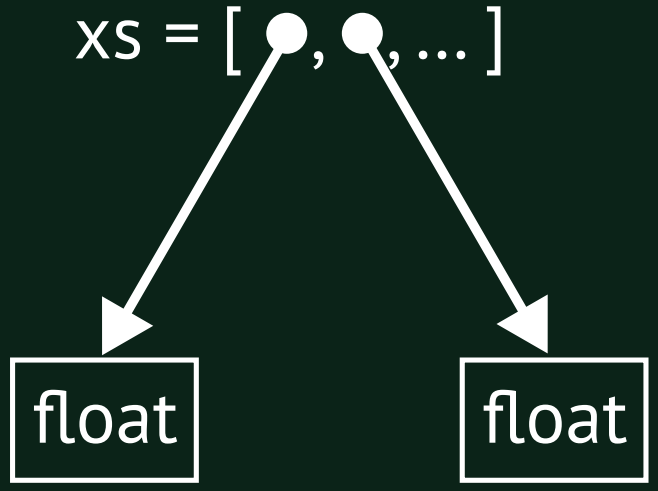
~200× faster

~15 years of hardware improvements

Row-oriented



Column-oriented



Column-oriented arrays

Layout techniques

1. Use column-oriented tools

Dataframes: Polars, Pandas, NumPy, ...

Databases: DuckDB, Parquet, Snowflake, ...

2. Do more in your database

```
def mean_point4():
    d = Point.objects.aggregate(
        x=Avg("x"),
        y=Avg("y"),
    )
    return Point(x=d['x'], y=d['y'])
```

3. Right-size your fields

```
class Book(models.Model):
    # Bloated, up to N bytes
    state = models.TextField(choices=...)
    # Just right, 1 byte
    state = models.PositiveSmallIntegerField(choices=
```

4. Learn data structures

- `frozenset`
- `collections: Counter, defaultdict, deque`
- `heapq`
- `queue`
- `graphlib`

2. Batching

`create_permissions()` optimization

Ticket #35408

`create_permissions()` optimization

Ticket #35408

2× speedup, from 5.2ms to 2.7ms

`create_permissions()` optimization

Ticket #35408

2× speedup, from 5.2ms to 2.7ms

8.5% to 4.7% of time to run Django's models tests

```
$ ./manage.py migrate
```

```
$ ./manage.py migrate
```

```
def emit_post_migrate_signal(...):
    for app_config in apps.get_app_configs():
        post_migrate.send(...)
```

```
$ ./manage.py migrate
```

```
def emit_post_migrate_signal(...):
    for app_config in apps.get_app_configs():
        post_migrate.send(...)
```

```
# django.contrib.auth.apps
post_migrate.connect(create_permissions, ...)
```

```
def create_permissions(...):
    ...
    ctypes = set()
    for klass in app_config.get_models():
        ctype = ContentType.objects.get_for_model(
            klass
        )
        ctypes.add(ctype)
    ...
```

```
def create_permissions(...):
    ...
    ctypes = ContentType.objects.get_for_models(
        *models
    )
    ...
```

We're gonna need a bigger batch!

Maybe we could add:

```
def emit_post_migrate_signal(...):
    # All apps at once!
    post_migrate_all.send(...)

    # Legacy.
    for app_config in apps.get_app_configs():
        post_migrate.send(...)
```

Batching techniques

1. Avoid per-instance methods

```
 class User:
     ...
-    def send_notification(self, change):
-        ...

+def send_notification(change, users):
+    ...
```

2. Prefer larger functions

~~clean~~ **efficient** code

3. Use batch ORM methods

```
Widget.objects.bulk_create(widgets)

Widget.objects.bulk_update(widgets, fields=[...])

Widget.objects.bulk_create(
    widgets,
    update_conflicts=True,
    update_fields=[...],
)
```

4. Pre-emptively pluralize

```
 class Company(models.Model):
-    boss = models.ForeignKey(...)
+    bosses = models.ManyToManyField(...)
```

3. Statistical distribution

`_route_to_regex()` optimization

Ticket #35252

```
urlpatterns = [
    path("/p/<int:user_id>/", views.profile),
]
```

```
urlpatterns = [
    path("/p/<int:user_id>/", views.profile),
]
```

```
In [1]: _route_to_regex("/p/<int:user_id>/")
Out[1]:
('^/p/(?P<user_id>[0-9]+)/',
 {'user_id': <...IntConverter at 0x101f056d0>})
```

A surprising(?) optimization

```
def _route_to_regex(route, is_endpoint):
    ...
```

A surprising(?) optimization

```
+@functools.lru_cache
 def _route_to_regex(route, is_endpoint):
     ...
```

A surprising(?) optimization

```
+@functools.lru_cache
 def _route_to_regex(route, is_endpoint):
     ...
```

Repeat calls ~100x faster.

Data logging

```
import atexit, pprint

all_routes = []

@atexit.register
def print_routes():
    pprint.pprint(all_routes)


def _route_to_regex(route, is_endpoint):
    ...
    all_routes.append(route)
```

Data logging

```
'',
'add/',
'<path:object_id>/history/',
'<path:object_id>/delete/',
'<path:object_id>/change/',
'<path:object_id>/',
...
'',
'add/',
'<path:object_id>/history/',
'<path:object_id>/delete/',
'<path:object_id>/change/',
'<path:object_id>/',
```

ModelAdmin.get_urls()

```
def get_urls(self):
    ...
    return [
        path("", ...),
        path("add/", ...),
        path(
            "<path:object_id>/history/",
            ...
        ),
        ...
    ]
```

Statistical distribution techniques

1. Collect metadata

- `print()`
- Database metrics
- Production logs, APM tools
- Heck, even spreadsheets

2. Cache when repetition likely

- HTTP caching
- Django's cache framework
- `cachetools` - in-memory TTL cache

3. Check common conditions early

`AdminEmailHandler` - Ticket #35364

```
def emit(self, record):
    # Early return when no email will be sent
    if not settings.ADMINS:
        return

    # Render email
    ...

    # Send email to settings.ADMINS
    ...
```

Resources

- Casey Muratori - Clean code, horrible performance
- Cal Peterson - Take the tools out of 'Data', but don't take the data out of the tools
- Brandur - Atlanta, Job Queues, Batch-wise Operations
- swyx - Preemptive Pluralization is (Probably) Not Evil

Thank you! 🤗

- adamj.eu/contact
- github.com/adamchainz/talk-data-oriented-django-deux
- Books: **Boost Your Django DX**, **Boost Your Git DX**, **Speed Up Your Django Tests**