Full Stack Type Safety

Szymon Pyżalski

Egnyte Inc.

Europython 2020
Outline

Premise

Typing basics

Our typical stack

Annotations and ORM

Enforcing the contract

Summary
Our goal

• Catch typing errors ASAP (not later than in CI)
• Catch typing errors that span layers of stack
Our goal

- Catch typing errors ASAP (not later than in CI)
Our goal

- Catch typing errors ASAP (not later than in CI)
- Catch typing errors that span layers of stack
Problems

• Type annotation system in Python is new and immature
• Various layers of stack feature different typing paradigms
• We tend to test layers in separation
Problems

- Type annotation system in Python is new and immature
Problems

- Type annotation system in Python is new and immature
- Various layers of stack feature different typing paradigms
Problems

• Type annotation system in Python is new and immature
• Various layers of stack feature different typing paradigms
• We tend to test layers in separation
Weak vs strong

Weak typing: A value can be misinterpreted unless we care about the type by ourselves.

Strong typing: We are protected from misinterpretations by the type system.
Weak vs strong

Weak typing  A value can be misinterpreted unless we care about the type by ourselves.
Weak vs strong

**Weak typing**  A value can be misinterpreted unless we care about the type by ourselves.

**Strong typing**  We are protected from misinterpretations by the type system.
Weak typing

```c
#include <stdio.h>
short int fun(int* x) {
    short int y = *(short int*)x;
    return y + 1;
}

int main(int argc, char** argv) {
    int a = -10;
    int b = 777777;
    printf("%u\n", a); // prints: 4294967286
    printf("%d\n", fun(&b)); // prints: -8654
}
```
Static vs dynamic
Static vs dynamic

**Static typing**  The types of objects can be determined during compile time.
Static vs dynamic

**Static typing**  The types of objects can be determined during compile time.

**Dynamic typing**  The types of objects are determined during runtime.
Dynamic typing in python

```python
def sum(xs, init):
    result = init
    for x in xs:
        result += x
    return result

print(sum([1, 2, 3], 0))  # prints 6
print(sum({'a': 'b', 'c': 'd'}, 'Keys: '))  # prints: Keys: ac
```
package main

import "fmt";

func fact(n int) int {
    result := 1
    for i := 1; i <= n; i++ {
        result *= i
    }
    return result
}

func main() {
    x := 10
    y := fact(5)
    fmt.Println(x)
    fmt.Println(y)
}
Strict vs loose
Strict vs loose

**Strict typing** Type conversions must be explicit. Type mismatch exceptions.
Strict vs loose

**Strict typing**  Type conversions must be explicit. Type mismatch exceptions.

**Loose typing**  Type conversions can be implicit.
Stricter than Python

```haskell
import Data.String.Utils (join)

list2Str :: [[[Char]]] -> [Char]
list2Str xs = if xs then "No elements" else (join "," xs) -

main = do
    putStrLn $ list2Str []
    putStrLn 10 -- Error
```
Looser than Python

\[
\begin{align*}
1 + 'a' & \quad // '1a' \\
{} + 2 & \quad // 0 \\
'abc' + ['d', 'e', 'f'] & \quad // "abcd,e,f" \\
{} + 'z' & \quad // NaN \\
{} + {} & \quad // NaN \\
{} + [] & \quad // 0 \\
[] + {} & \quad // "[object Object]"
\end{align*}
\]
Duck vs ???

Interfaces Protocols are implemented implicitly. Classes must inherit from a class in order to be compatible, or at least be marked as implementing the protocol.
Duck vs ???

**Duck typing** Interfaces Protocols are implemented implicitly. Object is compatible with a protocol if it implements required methods.
Duck vs ???

**Duck typing** Interfaces Protocols are implemented implicitly. Object is compatible with a protocol if it implements required methods.

??? Classes must inherit from a class in order to be compatible, or at least be marked as implementing the protocol.
Duck vs platonic

**Duck typing**  Interfaces Protocols are implemented implicitly. Object is compatible with a protocol if it implements required methods.

**Platonic typing**  Classes must inherit from a class in order to be compatible, or at least be marked as implementing the protocol.
Structural vs nominal

Structural typing  Interfaces Protocols are implemented implicitly. Object is compatible with a protocol if it implements required methods.

Nominal typing  Classes must inherit from a class in order to be compatible, or at lease be marked as implementing the protocol.
## The pythonish language

<table>
<thead>
<tr>
<th>They say</th>
<th>We say</th>
</tr>
</thead>
<tbody>
<tr>
<td>throw</td>
<td></td>
</tr>
<tr>
<td>raise</td>
<td></td>
</tr>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>list</td>
<td></td>
</tr>
<tr>
<td>deque</td>
<td></td>
</tr>
<tr>
<td>StopIteration</td>
<td></td>
</tr>
<tr>
<td>interfaces</td>
<td></td>
</tr>
<tr>
<td>protocols</td>
<td></td>
</tr>
</tbody>
</table>
## The pythonish language

<table>
<thead>
<tr>
<th>They say</th>
<th>We say</th>
</tr>
</thead>
<tbody>
<tr>
<td>throw</td>
<td>raise</td>
</tr>
</tbody>
</table>
The pythonish language

<table>
<thead>
<tr>
<th>They say</th>
<th>We say</th>
</tr>
</thead>
<tbody>
<tr>
<td>throw</td>
<td>raise</td>
</tr>
<tr>
<td>array</td>
<td>list</td>
</tr>
</tbody>
</table>
## The pythonish language

<table>
<thead>
<tr>
<th>They say</th>
<th>We say</th>
</tr>
</thead>
<tbody>
<tr>
<td>throw</td>
<td>raise</td>
</tr>
<tr>
<td>array</td>
<td>list</td>
</tr>
<tr>
<td>list</td>
<td>deque</td>
</tr>
</tbody>
</table>
# The pythonish language

<table>
<thead>
<tr>
<th>They say</th>
<th>We say</th>
</tr>
</thead>
<tbody>
<tr>
<td>throw</td>
<td>raise</td>
</tr>
<tr>
<td>array</td>
<td>list</td>
</tr>
<tr>
<td>list</td>
<td>deque</td>
</tr>
<tr>
<td>blatant abuse of exceptions</td>
<td>StopIteration</td>
</tr>
</tbody>
</table>
The pythonish language

<table>
<thead>
<tr>
<th>They say</th>
<th>We say</th>
</tr>
</thead>
<tbody>
<tr>
<td>throw</td>
<td>raise</td>
</tr>
<tr>
<td>array</td>
<td>list</td>
</tr>
<tr>
<td>list</td>
<td>deque</td>
</tr>
<tr>
<td>blatant abuse of exceptions</td>
<td>Stopleteration</td>
</tr>
<tr>
<td>interfaces</td>
<td>protocols</td>
</tr>
</tbody>
</table>
## The pythonish language

<table>
<thead>
<tr>
<th>They say</th>
<th>We say</th>
</tr>
</thead>
<tbody>
<tr>
<td>throw</td>
<td>raise</td>
</tr>
<tr>
<td>array</td>
<td>list</td>
</tr>
<tr>
<td>list</td>
<td>deque</td>
</tr>
<tr>
<td>blatant abuse of exceptions</td>
<td>StopIteration</td>
</tr>
<tr>
<td>interfaces</td>
<td>protocols</td>
</tr>
</tbody>
</table>
package main
import "fmt"

type Duck interface {
    swim(x int, y int)
    quack() string
}

type Mallard struct {
    x, y int
}

func (m *Mallard) swim(x, y int) {
    m.x += x
    m.y += y
}

func (m Mallard) quack() string {
    return "Quack quaaaack"
}

func swimThenQuack(d Duck) {
    d.swim(1, 1)
    fmt.Println(d.quack())
}

func main() {
    donald := Mallard{x: 0, y: 0}
    swimThenQuack(&donald)
    fmt.Println(donald)
}
Typing models

- Strong vs weak typing
- Static vs dynamic typing
- Strict vs loose typing
- Structural vs nominal typing
Typing models

- Strong vs weak typing
- Static vs dynamic typing
- Strict vs loose typing
- Structural vs nominal typing
- Free vs fixed attributes
Our typical stack

<table>
<thead>
<tr>
<th>Language</th>
<th>Strong</th>
<th>Very loose</th>
<th>Dynamic</th>
<th>Structural</th>
<th>Free attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Javascript</td>
<td>Strong</td>
<td>Very loose</td>
<td>Dynamic</td>
<td>Structural</td>
<td>Free attributes</td>
</tr>
<tr>
<td>Python</td>
<td>Strong</td>
<td>Strict</td>
<td>Dynamic</td>
<td>Structural</td>
<td>Free attributes</td>
</tr>
<tr>
<td>SQL</td>
<td>Weak (foreign keys)</td>
<td>Loose</td>
<td>Static</td>
<td>Nominal</td>
<td>Fixed attributes</td>
</tr>
</tbody>
</table>
## Our typical stack

<table>
<thead>
<tr>
<th>Javascript</th>
<th>Python</th>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong</td>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td>Very loose</td>
<td>Strict</td>
<td>Strict</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Static</td>
</tr>
<tr>
<td>Structural</td>
<td>Structural</td>
<td>Nominal</td>
</tr>
<tr>
<td>Free attributes</td>
<td>Free attributes</td>
<td>Fixed attributes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak (foreign keys)</td>
</tr>
<tr>
<td>Loose</td>
</tr>
<tr>
<td>Static</td>
</tr>
<tr>
<td>Nominal</td>
</tr>
<tr>
<td>Fixed attributes</td>
</tr>
</tbody>
</table>
Weakness of SQL foreign keys

UPDATE books set author_id = (  
  SELECT id FROM publishers  
  WHERE name="Chilton Books"
);
ORM improving type safety

```
# b = Book.objects.get(id=1)
b.author = Publisher.objects.get(name='Chilton Books')
```
mypy enters the game

<table>
<thead>
<tr>
<th>Javascript</th>
<th>mypy</th>
<th>Python</th>
<th>Models</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong</td>
<td>Strong</td>
<td>Strong</td>
<td>Strong</td>
<td>Weak (foreign keys)</td>
</tr>
<tr>
<td>Very loose</td>
<td>String</td>
<td>Strict</td>
<td>Static</td>
<td>Loose</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Static</td>
<td>Dynamic</td>
<td>Static</td>
<td>Static</td>
</tr>
<tr>
<td>Structural</td>
<td>Preference for nominal</td>
<td>Structural</td>
<td>Nominal</td>
<td>Nominal</td>
</tr>
<tr>
<td>Free attributes</td>
<td>Fixed attributes</td>
<td>Free attributes</td>
<td>Fixed attributes</td>
<td>Fixed attributes</td>
</tr>
</tbody>
</table>

Preference for nominal
Demo 1

Django and mypy working together
mypy and Django pros and cons

- **Pro**: Recognizes the relationship between column types and Python types
- **Pro**: Recognizes the idea of null
- **Con**: Can't handle problems with incomplete data
- **Con**: Requires a mypy plugin
mypy and Django pros and cons

• Pro: Recognizes the relationship between column types and python types
mypy and Django pros and cons

- **Pro**: Recognizes the relationship between column types and python types
- **Pro**: Recognizes the idea of null
mypy and Django pros and cons

- Pro: Recognizes the relationship between column types and python types
- Pro: Recognizes the idea of null
- Con: Can’t handle problems with incomplete data
mypy and Django pros and cons

- **Pro**: Recognizes the relationship between column types and Python types
- **Pro**: Recognizes the idea of null
- **Con**: Can’t handle problems with incomplete data
- **Con**: Requires a mypy plugin
# Considering the JSON

<table>
<thead>
<tr>
<th></th>
<th>Javascript</th>
<th>JSON</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Strong</td>
<td>No typing above primitives</td>
</tr>
<tr>
<td></td>
<td>Very loose</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dynamic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Structural</td>
<td></td>
</tr>
<tr>
<td>Free attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mppy</td>
<td>Strong</td>
<td></td>
</tr>
<tr>
<td></td>
<td>String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Static</td>
<td></td>
</tr>
<tr>
<td>Preference for nominal</td>
<td>Fixed attributes</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>Strong</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Strict</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dynamic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Structural</td>
<td></td>
</tr>
<tr>
<td>Free attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL</td>
<td>Weak (foreign keys)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Loose</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Static</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Nominal</td>
<td></td>
</tr>
<tr>
<td>Fixed attributes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Preference for nominal: Fixed attributes
### One solution

<table>
<thead>
<tr>
<th>Typescript</th>
<th>JSON</th>
<th>OpenAPI3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong</td>
<td>No typing above primitives</td>
<td>Schema</td>
</tr>
<tr>
<td>Strict</td>
<td></td>
<td>Tests</td>
</tr>
<tr>
<td>Static</td>
<td></td>
<td>Code generation</td>
</tr>
<tr>
<td>Structural</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed attributes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mypy</th>
<th>Python</th>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong</td>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td>String</td>
<td>Strict</td>
<td>Strict</td>
</tr>
<tr>
<td>Static</td>
<td>Dynamic</td>
<td>Static</td>
</tr>
<tr>
<td>Preference for nominal</td>
<td>Structural</td>
<td>Nominal</td>
</tr>
<tr>
<td>Fixed attributes</td>
<td>Free attributes</td>
<td>Fixed attributes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak (foreign keys)</td>
</tr>
<tr>
<td>Loose</td>
</tr>
<tr>
<td>Static</td>
</tr>
<tr>
<td>Nominal</td>
</tr>
<tr>
<td>Fixed attributes</td>
</tr>
</tbody>
</table>
Demo 2

Enforcing the contract
Takeaways
Takeaways

- There are tools for code safety enforcement in a Python stack that are worth consideration
Takeaways

• There are tools for code safety enforcement in a Python stack that are worth consideration

• They are not yet perfect and we can’t expect to catch all errors
Future can bring

- Support for more patterns in type annotations without plugins
- Tools based on code annotations instead of descriptors (strawberry-graphql, pydantic, )
Future can bring

- Support for more patterns in type annotations without plugins
Future can bring

- Support for more patterns in type annotations without plugins
- Tools based on code annotations instead of descriptors (strawberry-graphql, pydantic, )
Tools used

- django-stubs A distribution of code annotations for django complete with a mypy plugin
- spectacular A schema generator for django-rest-framework
- openapi-generator Code generator that can create boilerplate code for several languages/frameworks based on OpenAPI3.