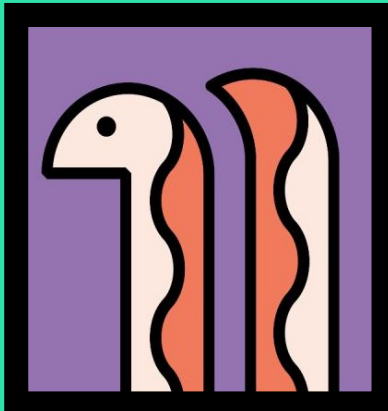
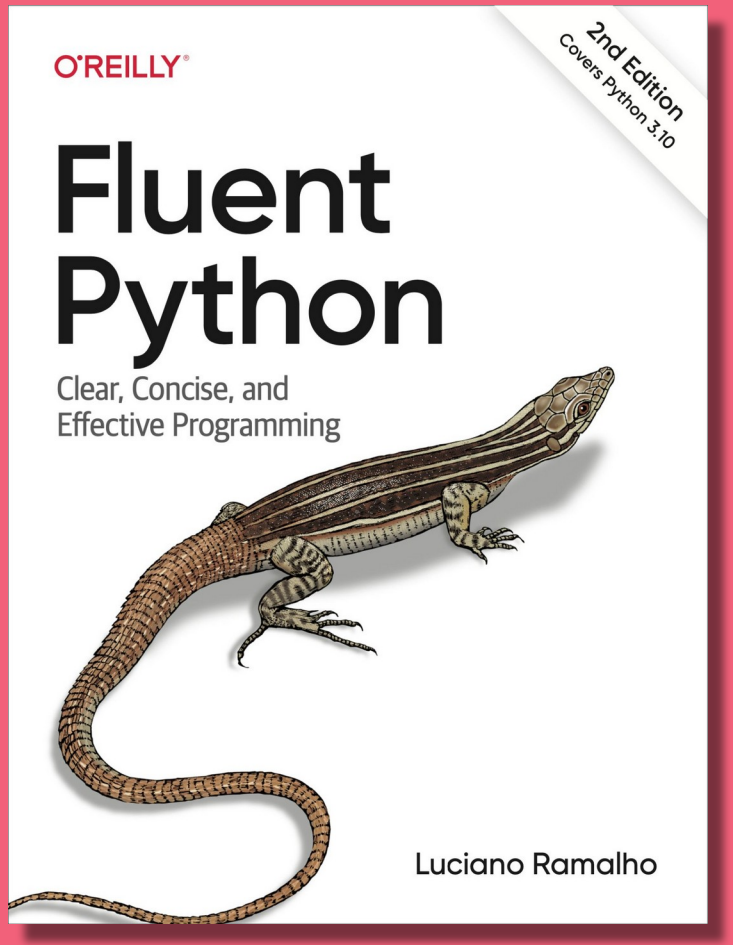
PYCON(IT) 24

Pythonic type hints with `typing.Protocol`

Luciano Ramalho
https://fosstodon.org/@ramgarlic





Fluent Python, Second Edition

- Covers **3.10**, including **pattern matching**
- 100+ pages about **type hints,** with many examples
- New coverage of **async/await**, with examples in **asyncio**, **FastAPI** and **Curio**
- OReilly.com: https://bit.ly/FluPy2e

Motivating Example

The first parameter is the **file-like object** to be sent...

To be considered “file-like”, the object supplied by the application must have a `read()` method that takes an optional size argument.

PEP 3333 – Python Web Server Gateway Interface

The words "file-like" appear with similar implied meaning in the Python 3.12 distribution:

- 148 times in the documentation;
- 92 times in code comments in .py or .c source files;
- 30 times across 21 PEPs:
  100, 214, 258, 282, 305, 310, 333, 368, 400, 441, 444, 578, 680, 691, 3116, 3119, 3143, 3145, 3154, 3156, 3333.

Protocol definition for “a file-like object”

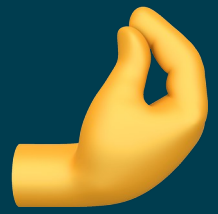
```python
from typing import Protocol

# ...

class _Readable(Protocol):
    def read(self, size: int = ..., /) -> bytes: ...
```

code from Lib/wsgiref/types.py

Agenda

1. What is a type?
2. The four modes of typing
3. typing.Protocol examples
4. Conclusion

What is a type?



There are many definitions of the concept of type in the literature. Here we assume that type is a set of values and a set of functions that one can apply to these values.

Guido van Rossum, Ivan Levkivskyi in PEP 483—The Theory of Type Hints

```
>>> i = 10**23
>>> i
100000000000000000000000
>>> f = float(i)
>>> f
1e+23
```

```
>>> i = 10**23
>>> i
100000000000000000000000
>>> f = float(i)
>>> f
1e+23
>>> i == f
False
```

```
>>> i = 10**23
>>> i
100000000000000000000000
>>> f = float(i)
>>> f
1e+23
>>> i == f
False
>>> from decimal import Decimal
>>> Decimal(i)
Decimal('100000000000000000000000')
>>> Decimal(f)
Decimal('99999999999999991611392')
```

The “set of values” definition is not useful: Python does not provide ways to specify types as sets of values, except for Enum.

We have very small sets (None, bool) or very large ones (int, str…).

Python type hints cannot define a `Quantity` type as the set of integers $0 < n \leq 1000$

or…

`AirportCode` as the set of all 17576 three-letter ASCII strings like “FLR”, ”LAX”, “BER” etc.

In practice, it's more useful to think that `int` is a subtype of `float` because it implements the same interface, and adds extra methods —not because `int` is a subset of `float`*

*** which it definitely isn't**

```
>>> i = 10**23
>>> i
100000000000000000000000
>>> f = float(i)
>>> f
1e+23
>>> i == f
False
>>> from decimal import Decimal
>>> Decimal(i)
Decimal('100000000000000000000000')
>>> Decimal(f)
Decimal('99999999999999991611392')
>>> i | 2
100000000000000000000002
>>> f | 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for |: 'float' and 'int'
```

There are many definitions of the concept of type in the literature. Here we assume that type is a set of values and a set of functions that one can apply to these values.

Guido van Rossum, Ivan Levkivskyi in PEP 483—The Theory of Type Hints

Every object in Smalltalk, even a lowly integer, has a set of messages, a protocol, that defines the explicit communication to which that object can respond.

Dan Ingalls (Xerox PARC) in Design Principles Behind Smalltalk —BYTE Magazine, August 1981

To summarize:

- Types are defined by interfaces
- Protocol is a synonym of interface

Duck typing

Don’t check whether it is-a duck:
check whether it quacks-like-a duck, walks-like-a duck, etc, depending on exactly what subset of duck-like behavior you need...

Alex Martelli in comp-lang-python: ”polymorphism (was Re: Type checking in python?)” 2000-07-26

```
>>> def double(x):
...     return x * 2
...
>>> double(3)
6
>>> double(3.5)
7.0
>>> double(3j+4)
(8+6j)
>>> from fractions import Fraction
>>> double(Fraction(1, 3))
Fraction(2, 3)
>>> double('Spam')
'SpamSpam'
>>> double([1, 2, 3])
[1, 2, 3, 1, 2, 3]
>>>
```

```
>>> class Train:
...     def __init__(self, cars):
...         self.cars = cars
...     def __iter__(self):
...         for i in range(self.cars):
...             yield f'car #{i+1}'
...
>>> t = Train(4)
>>> for car in t:
...     print(car)
...
car #1
car #2
car #3
car #4
```

```
>>> class Train:
...     def __init__(self, cars):
...         self.cars = cars
...     def __getitem__(self, i):
...         if i < self.cars:
...             return f'car #{i+1}'
...         raise IndexError
...     def __len__(self):
...         return self.cars
...
>>> t = Train(4)
>>> len(t)
4
>>> t[0]
'car #1'
>>> for car in t:
...     print(car)
...
car #1
car #2
car #3
```

Example 1

A TextReader protocol

typing.Protocol allows (static) duck typing

```
14  from collections.abc import Sequence, Iterator
15  from typing import Any, Protocol, Callable, NoReturn
16
17  import lis

200
201 class TextReader(Protocol):
202     def read(self) -> str:
203         ...
204
205
206 def run_file(source_file: TextReader, env: lis.Environment | None = None) -> Any:
207     source = source_file.read()
208     return run(source, env)
209
```

typing.Protocol allows (static) duck typing

```
14  from collections.abc import Sequence, Iterator
15  from typing import Any, Protocol, Callable, NoReturn
16
17  import lis
```

```
200
201  class TextReader(Protocol):
202      def read(self) -> str:
203          ...
204
205
206  def run_file(source_file: TextReader, env: lis.Environment | None = None) -> Any:
207      source = source_file.read()
208      return run(source, env)
209
```

Protocol used in library code, not in application code

typing.Protocol allows (static) duck typing

```
14  from collections.abc import Sequence, Iterator
15  from typing import Any, Protocol, Callable, NoReturn
16
17  import lis
```

```
200
201  class TextReader(Protocol):
202      def read(self) -> str:
203          ...
204
205
206  def run_file(source_file: TextReader, env: lis.Environment | None = None) -> Any:
207      source = source_file.read()
208      return run(source, env)
209
```

Protocol often defined near API that requires it

Benefits of using typing.Protocol

Preserve the flexibility of duck typing

Let your clients know what is the minimal interface expected, regardless of class hierarchies

Benefits of using typing.Protocol

Preserve the flexibility of duck typing

Let your clients know what is the minimal interface expected, regardless of class hierarchies

Support static analysis

IDEs and linters can verify that an actual argument satisfies the protocol in the formal parameter

Benefits of using typing.Protocol

Preserve the flexibility of duck typing

Let your clients know what is the minimal interface expected, regardless of class hierarchies

Support static analysis

IDEs and linters can verify that an actual argument satisfies the protocol in the formal parameter

Reduce coupling

Client classes don’t need to subclass anything; just implement the protocol. This also makes testing easier.

The four modes of typing

Static v. Dynamic Typing

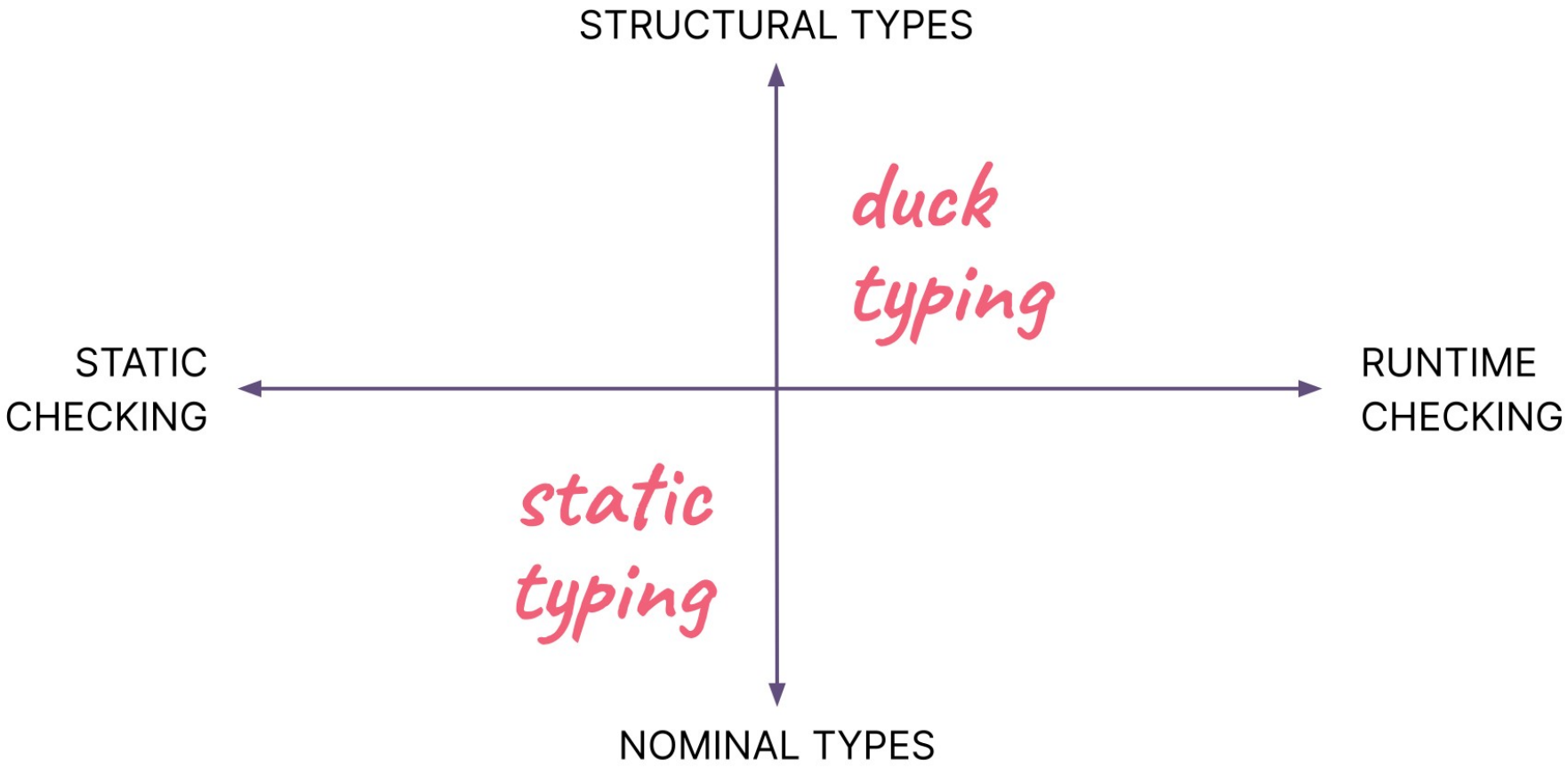
Static v. Dynamic Typing

a matter of *when*

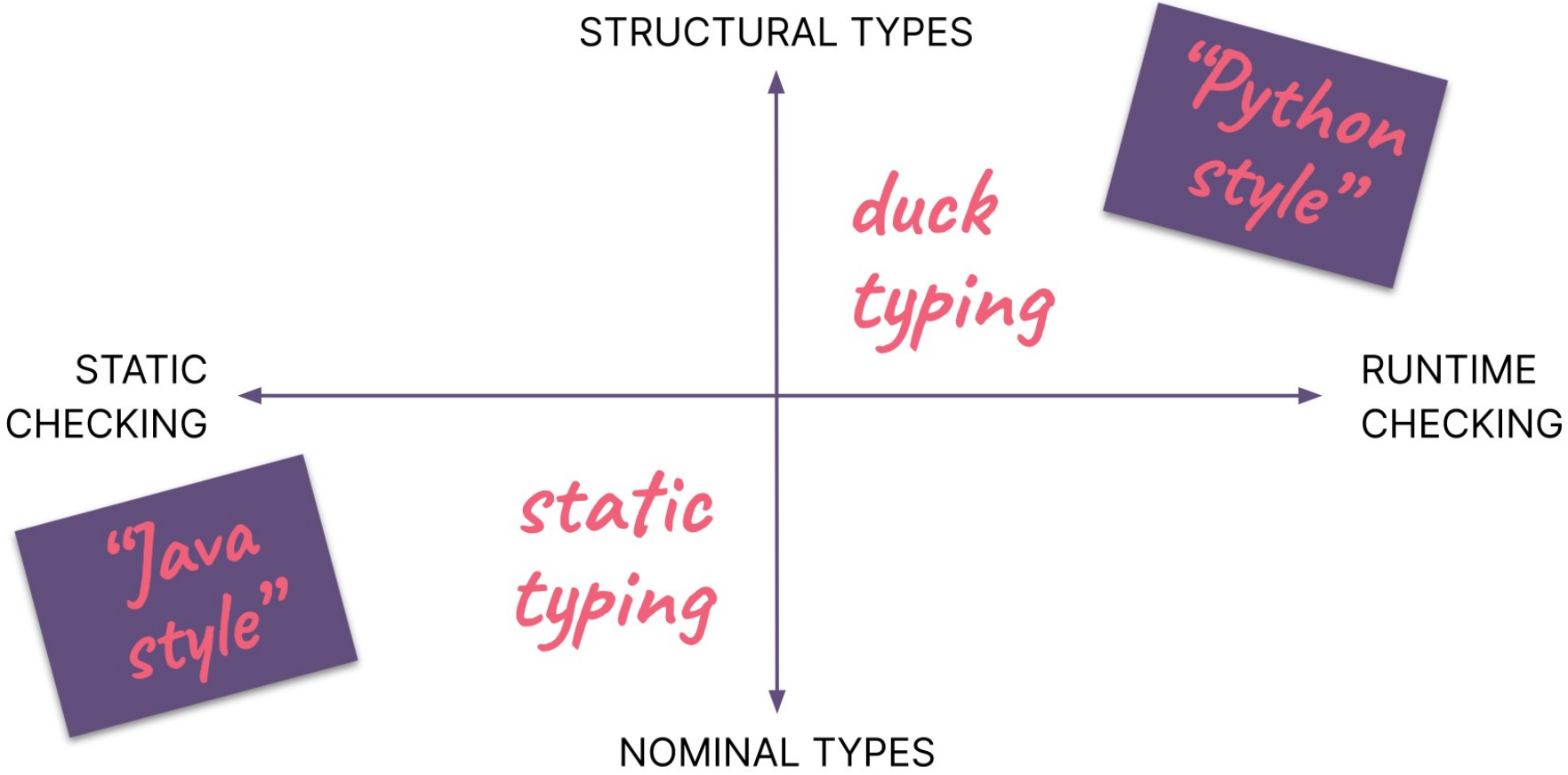
Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention.

Guido van Rossum, Jukka Lehtosalo, Łukasz Langa in PEP 484—Type Hints

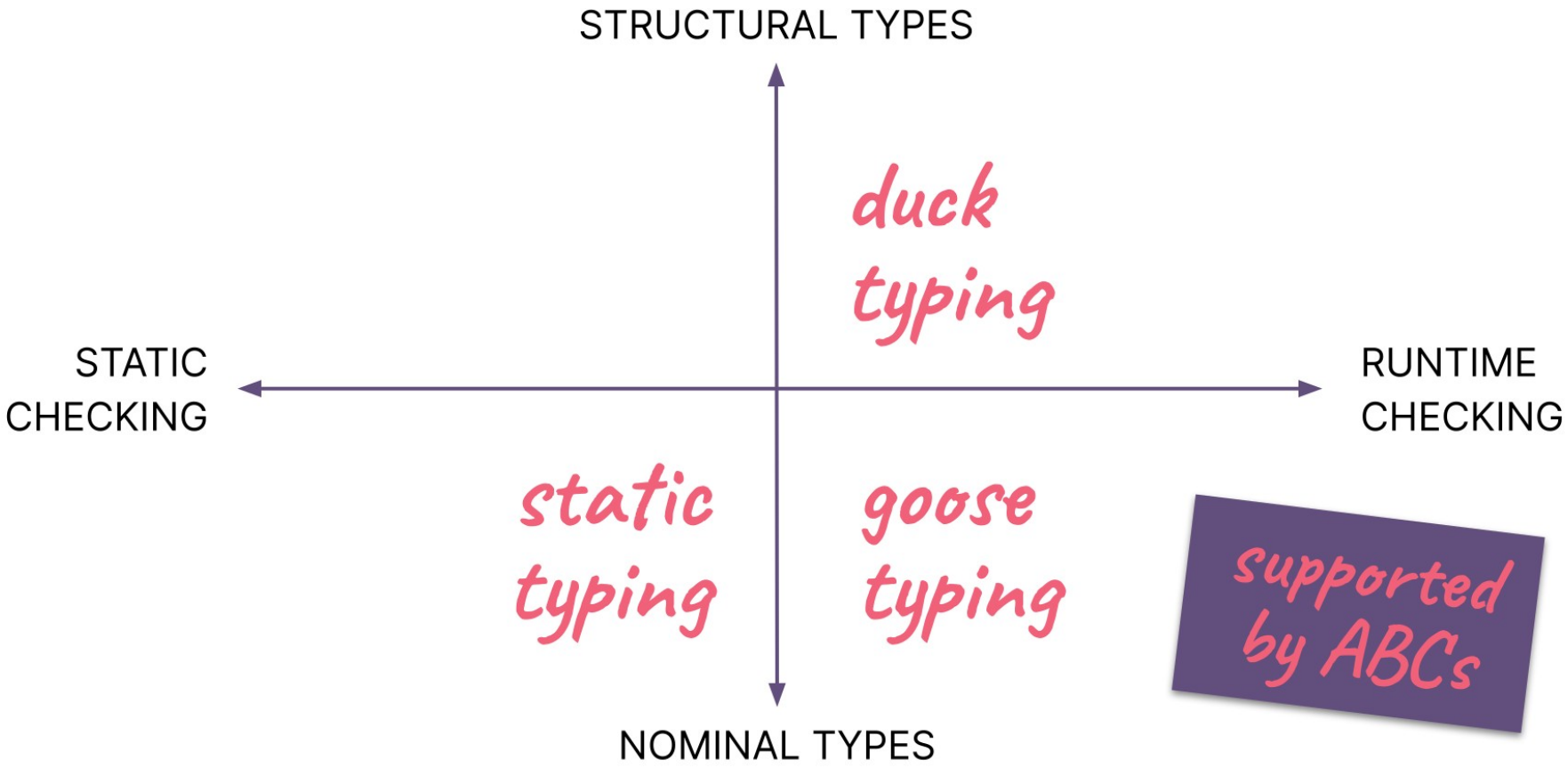
Typing Map


STRUCTURAL TYPES
duck typing
STATIC CHECKING
RUNTIME CHECKING
static typing
NOMINAL TYPES

Typing Map


STRUCTURAL TYPES
"Python style"
duck typing
STATIC CHECKING
RUNTIME CHECKING
"Java style"
static typing
NOMINAL TYPES

Typing Map


STRUCTURAL TYPES
duck typing
STATIC CHECKING
RUNTIME CHECKING
static typing
goose typing
supported by ABCs
NOMINAL TYPES

PEP 544 -- Protocols: Structur

https://www.python.org/dev/peps/pep-0544/

Python | PSF | Docs | PyPI | Jobs | Community

python™

Donate | Search | GO | Socialize

About | Downloads | Documentation | Community | Success Stories | News | Events

Tweets by @ThePSF

Python Software Foundation
@ThePSF

Your support helps us sustain our Developer-in-Residence program & the Packaging Project Manager position. Donate during our upcoming #PSFSpringFundraiser & help us reach our goal of $80K (only a small portion of what is needed to provide these critical programs). Stay tuned.

2h

Read platform.twitter.com

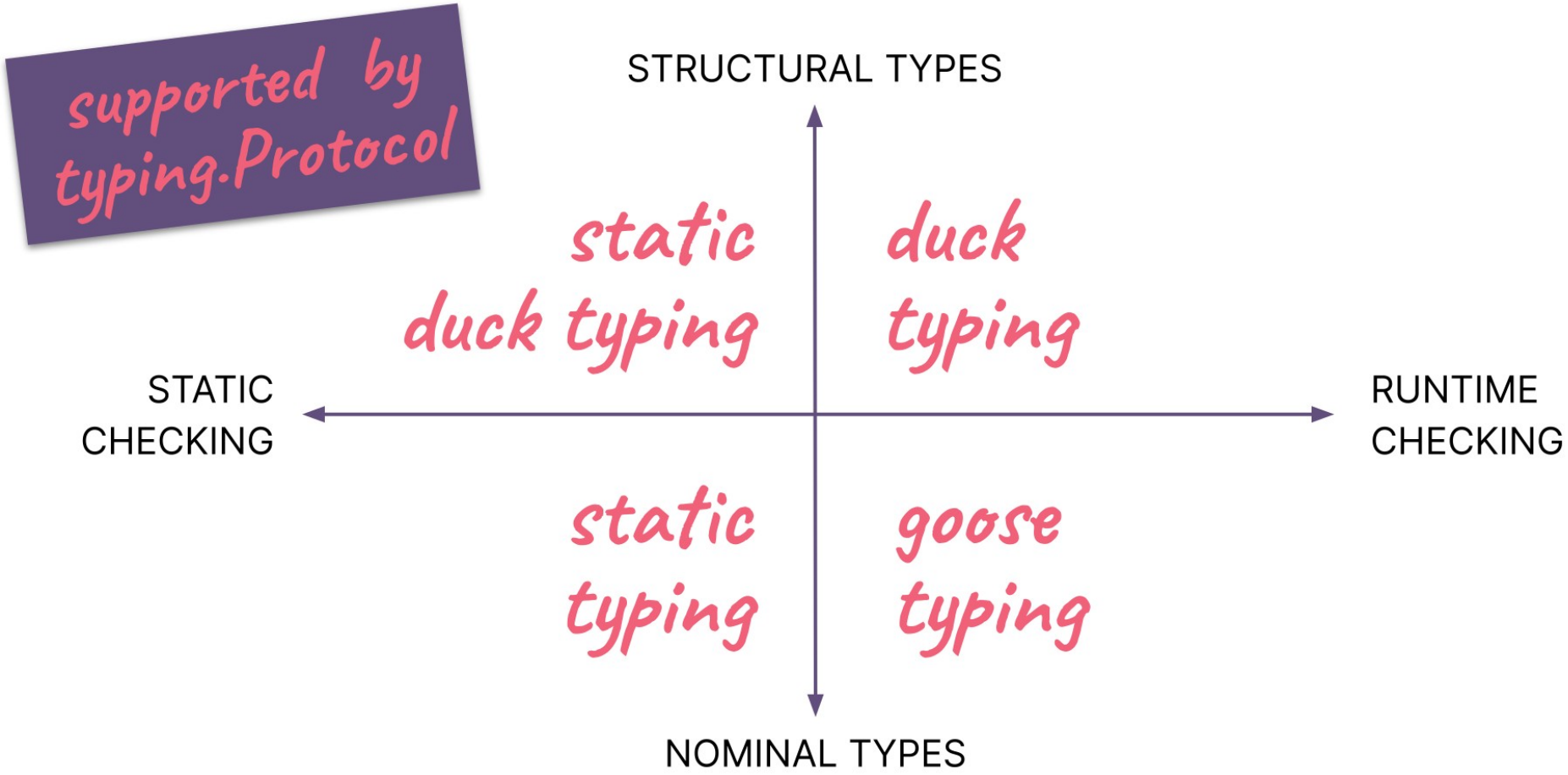
Python Software Foundation

Python >>> Python Developer's Guide >>> PEP Index >>> PEP 544 -- Protocols: Structural subtyping (static duck typing)

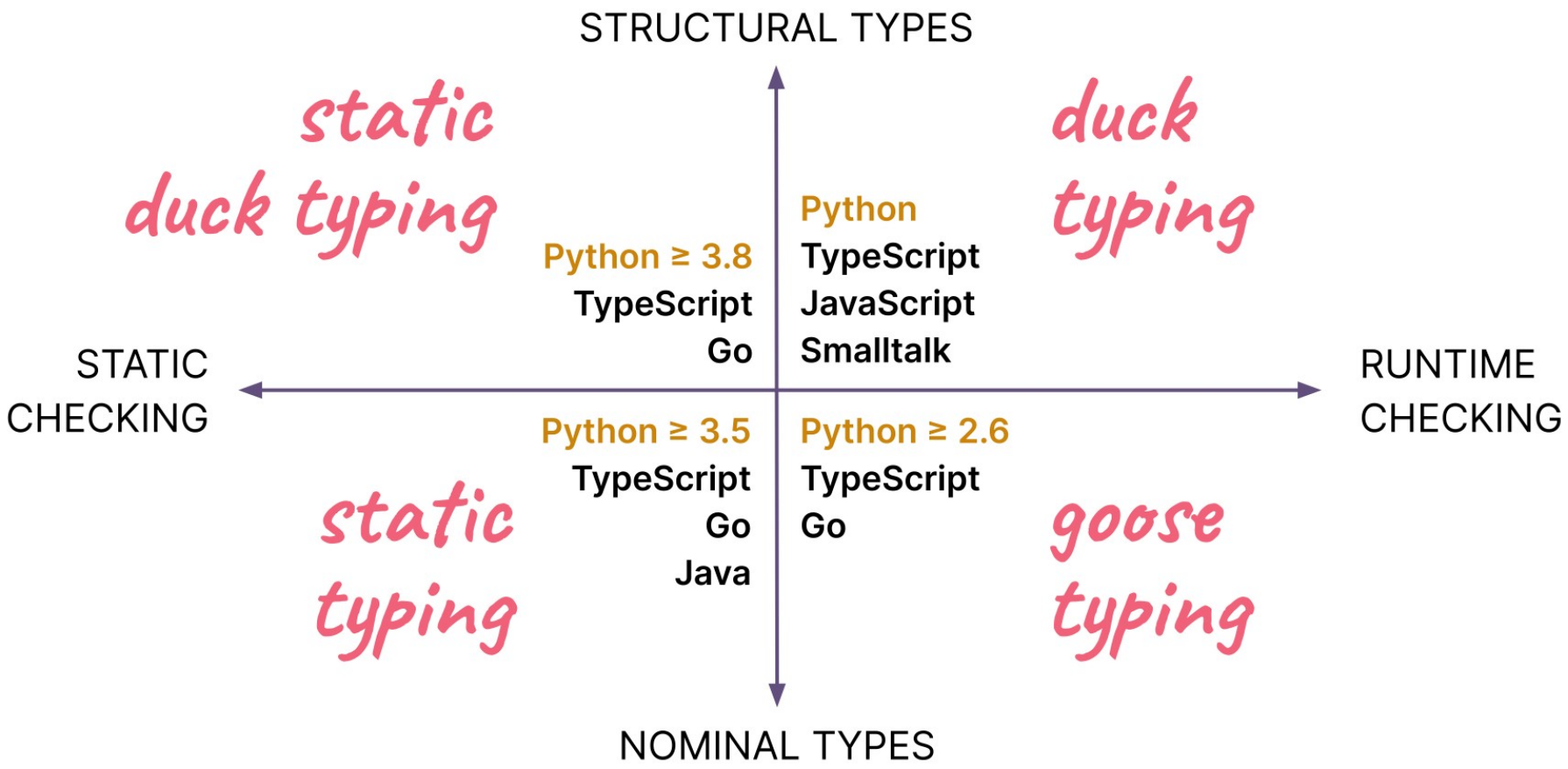
PEP 544 -- Protocols: Structural subtyping (static duck typing)

PEP:	544
Title:	Protocols: Structural subtyping (static duck typing)
Author:	Ivan Levkivskyi <levkivskyi at gmail.com>, Jukka Lehtosalo <jukka.lehtosalo at iki.fi>, Łukasz Langa <lukasz at python.org>
BDFL-Delegate:	Guido van Rossum <guido at python.org>
Discussions-To:	Python-Dev <python-dev at python.org>
Status:	Accepted
Type:	Standards Track

Typing Map


supported by typing.Protocol
STRUCTURAL TYPES
static duck typing
duck typing
STATIC CHECKING
RUNTIME CHECKING
static typing
goose typing
NOMINAL TYPES

Typing Map: languages



More Examples

2.

double

```
>>> def double(x):
...     return x * 2
...
>>> double(3)
6
>>> double(3.5)
7.0
>>> double(3j+4)
(8+6j)
>>> from fractions import Fraction
>>> double(Fraction(1, 3))
Fraction(2, 3)
>>> double('Spam')
'SpamSpam'
>>> double([1, 2, 3])
[1, 2, 3, 1, 2, 3]
>>>
```

First take: object

```
def double(x: object) -> object:
    return x * 2
```

Error: object does not implement __mul__

Second take: Any

```
from typing import Any


def double(x: Any) -> Any:
    return x * 2
```

Useless: Any defeats type checking

Every Python value is of type object.
Every Python value is of type Any.

The object type implements a narrow interface, but Any is assumed to implement the widest possible interface: all possible methods!

Third take: Sequence[T]

```
from collections import abc
from typing import TypeVar

T = TypeVar('T')

def double(x: abc.Sequence[T]) -> Sequence[T]:
    return x * 2
```

Only works with sequences, not numbers

Fourth take: protocol misuse

```
from typing import TypeVar, Protocol

T = TypeVar('T')  # <1>

class Repeatable(Protocol):
    def __mul__(self: T, repeat_count: int) -> T: ...  # <2>

def double(x: Repeatable) -> Repeatable:  # <3>
    return x * 2
```

Not OK: Type checker assumes that result supports only `__mul__`, and no other method.

Solution: type variable bounded by protocol

```
from typing import TypeVar, Protocol

T = TypeVar('T')  # <1>

class Repeatable(Protocol):
    def __mul__(self: T, repeat_count: int) -> T: ...  # <2>

RT = TypeVar('RT', bound=Repeatable)  # <3>

def double(x: RT) -> RT:  # <4>
    return x * 2
```

3.

statistics.median_low

statistics.median_low and media | Should Mypy warn about potent | Differed Protocol / runtime_che

https://github.com/python/typeshed/issues/3894

Search or jump to... / Pull requests Issues Marketplace Explore

python / typeshed

Sponsor | Watch 61 | Star 2.3k | Fork 1.1k

Code | Issues 140 | Pull requests 19 | Actions | Projects | Security | Insights

statistics.median_low and median_high accept more than float, Decimal, Fraction #3894

Edit | New issue

Closed ramalho opened this issue on Mar 31, 2020 · 9 comments

ramalho commented on Mar 31, 2020 • edited — Contributor

Evidence:

```
>>> import statistics
>>> l = ['b', 'v', 'c', 'l']
>>> statistics.median_low(l)
'c'
```

However, the stub for `median_low` says:

```
def median_low(data: Iterable[_Number]) -> _Number: ...
```

The items must be comparable. I'll do some research and propose a pull request.

srittau commented on Mar 31, 2020 — Collaborator

Same for `median()` and `median_high()`.

Assignees

No one assigned

Labels

size-small type-false-positive

Projects

None yet

Milestone

No milestone

Linked pull requests

Successfully merging a pull request may close this issue.

Change stub for median_high and median_...

statistics.median_low and media | Should Mypy warn about potent | Differed Protocol / runtime_che

https://github.com/python/typeshed/issues/3894 200%

than float, Decimal, Fraction #3894

Closed **ramalho** opened this issue on Mar 31, 2020 · 9 comments

ramalho commented on Mar 31, 2020 • edited Contributor

Evidence:

```
>>> import statistics
>>> l = ['b', 'v', 'c', 'l']
>>> statistics.median_low(l)
'c'
```

However, the stub for `median_low` says:

```
def median_low(data: Iterable[_Number]) -> _Number: ...
```

The items must be comparable. I'll do some research and propose a pull request.

Assignees

No one assigned

Labels

size-small type-false-positive

Projects

None yet

Milestone

No milestone

Linked pull requests 49

Successfully merging a pull request

median_low: fixed code

```python
from collections.abc import Iterable
from typing import Any, TypeVar, Protocol


class StatisticsError(ValueError):
    pass


class Sortable(Protocol):
    def __lt__(self, other: Any) -> bool: ...


SortableT = TypeVar('SortableT', bound=Sortable)


def median_low(data: Iterable[SortableT]) -> SortableT:
    """Return the low median value of data."""
    data = sorted(data)
    n = len(data)
```

etc.

First uses of the SupportsLessThan Protocol

Stub files for Python 3.9 standard library on typeshed in 2020

builtins:	list.sort
	max
	min
	sorted
statistics:	median_low
	median_high
functools:	cmp_to_key

bisect:	bisect_left
	bisect_right
	insort_left
	insort_right
heapq:	nlargest
	nsmallest
os.path:	commonprefix

Today* there are 120** Protocol definitions on typeshed/stdlib and nearly 100** on /stubs***

* 2024-05-24
** not counting network protocols
*** for external packages like Pillow, psycopg2, tensorflow, etc.

4.

max overload

The max() built-in function

Flexible and easy to use, but very hard to annotate

statistics.median_low and media | Should Mypy warn about potent | Differed Protocol / runtime_chec | Built-in Functions — Python 3.9

https://docs.python.org/3/library/functions.html#max 133%

max(*iterable*, **[, key, default]*)

max(*arg1*, *arg2*, **args[, key]*)

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a `ValueError` is raised.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc, reverse=True)[0]` and `heapq.nlargest(1, iterable, key=keyfunc)`.

New in version 3.4: The *default* keyword-only argument.

Changed in version 3.8: The *key* can be `None`.

statistics.median_low and media | Should Mypy warn about potent | Differed Protocol / runtime_che

https://github.com/python/typeshed/issues/4051 170%

Should Mypy warn about potential invalid arguments to `max`? #4051

New issue

Closed **willmcgugan** opened this issue on May 20, 2020 · 7 comments

willmcgugan commented on May 20, 2020

I was surprised that Mypy didn't flag an issue where I was calling `max` with an int and a None, which throws a TypeError at runtime.

For example, in the following code, should I not expect Mypy to warn that `top` is potentially None, which wouldn't work with `max` ?

```
from typing import Optional
top: Optional[int] = None
print(max(5, top))
```

JelleZijlstra commented on May 20, 2020 • edited Member

This is more of a typeshed issue. Currently `max()` is stubbed as accepting basically anything (https://github.com/python/typeshed/blob/master/stdlib/2and3/builtins.pyi#L1318). Perhaps it could be changed to accept only types that define `<`.

Assignees

No one assigned

Labels

size-medium type-false-negative

Projects

None yet

Milestone

No milestone

Linked pull requests

Successfully merging a pull request may close this issue.

max, min overloads with Protocol: f...

statistics.median_low and media | Should Mypy warn about potent | Differed Protocol / runtime_che

https://github.com/python/typeshed/issues/4051 170%

Should Mypy warn about potential invalid arguments to `max`? #4051

New issue

Closed **willmcgugan** opened this issue on May 20, 2020 · 7 comments

willmcgugan commented on May 20, 2020

I was surprised that Mypy didn't flag an issue where I was calling `max` with an int and a None, which throws a TypeError at runtime.

For example, in the following code, should I not expect Mypy to warn that `top` is potentially None, which wouldn't work with `max` ?

```
from typing import Optional
top: Optional[int] = None
print(max(5, top))
```

JelleZijlstra commented on May 20, 2020 • edited Member

This is more of a typeshed issue. Currently `max()` is stubbed as accepting basically anything (https://github.com/python/typeshed/blob/master/stdlib/2and3/builtins.pyi#L1318). Perhaps it could be changed to accept only types that define `<`.

False negative!

Labels

size-medium type-false-negative

Projects

None yet

Milestone

No milestone

Linked pull requests

Successfully merging a pull request may close this issue.

max, min overloads with Protocol: f...

max: old type hints

```
1358    @overload
1359    def max(__arg1: _T, __arg2: _T, *_args: _T, key: Callable[[_T], Any] = ...) -> _T: ...
1360    @overload
1361    def max(__iterable: Iterable[_T], *, key: Callable[[_T], Any] = ...) -> _T: ...
1362    @overload
1363    def max(__iterable: Iterable[_T], *, key: Callable[[_T], Any] = ..., default: _VT) -> Union[_T, _VT]: ...
```

max: fixed type hints

```
1119  @overload
1120  def max(
1121      __arg1: SupportsLessThanT, __arg2: SupportsLessThanT, *_args: SupportsLessThanT, key: None = ...
1122  ) -> SupportsLessThanT: ...
1123  @overload
1124  def max(__arg1: _T, __arg2: _T, *_args: _T, key: Callable[[_T], SupportsLessThan]) -> _T: ...
1125  @overload
1126  def max(__iterable: Iterable[SupportsLessThanT], *, key: None = ...) -> SupportsLessThanT: ...
1127  @overload
1128  def max(__iterable: Iterable[_T], *, key: Callable[[_T], SupportsLessThan]) -> _T: ...
1129  @overload
1130  def max(__iterable: Iterable[SupportsLessThanT], *, key: None = ..., default: _T) -> Union[SupportsLessThanT, _T]: ...
1131  @overload
1132  def max(__iterable: Iterable[_T1], *, key: Callable[[_T1], SupportsLessThan], default: _T2) -> Union[_T1, _T2]: ...
```

```python
def max(first, *args, key=None, default=MISSING):
    if args:
        series = args
        candidate = first
    else:
        series = iter(first)
        try:
            candidate = next(series)
        except StopIteration:
            if default is not MISSING:
                return default
            raise ValueError(EMPTY_MSG) from None
    if key is None:
        for current in series:
            if candidate < current:
                candidate = current
    else:
        candidate_key = key(candidate)
        for current in series:
            current_key = key(current)
            if candidate_key < current_key:
                candidate = current
                candidate_key = current_key
    return candidate
```

max implemented in Python, for testing

```python
from collections.abc import Callable, Iterable
from typing import Protocol, Any, TypeVar, overload, Union

MISSING = object()
EMPTY_MSG = 'max() arg is an empty sequence'

class SupportsLessThan(Protocol):
    def __lt__(self, other: Any) -> bool: ...

T = TypeVar('T')
LT = TypeVar('LT', bound=SupportsLessThan)
DT = TypeVar('DT')

@overload
def max(arg1: LT, arg2: LT, *_args: LT, key: None = ...) -> LT:
    ...
@overload
def max(arg1: T, arg2: T, *_args: T, key: Callable[[T], LT]) -> T:
    ...
@overload
def max(iterable: Iterable[LT], *, key: None = ...) -> LT:
    ...
@overload
def max(iterable: Iterable[T], *, key: Callable[[T], LT]) -> T:
    ...
@overload
def max(iterable: Iterable[LT], *, key: None = ..., default: DT) -> Union[LT, DT]:
    ...
@overload
def max(iterable: Iterable[T], *, key: Callable[[T], LT], default: DT) -> Union[T, DT]:
    ...
def max(first, *args, key=None, default=MISSING):
    if args:
        series = args
        candidate = first
    else:
        series = iter(first)
        try:
            candidate = next(series)
        except StopIteration:
            if default is not MISSING:
                return default
            raise ValueError(EMPTY_MSG) from None
    if key is None:
        for current in series:
            if candidate < current:
                candidate = current
    else:
        candidate_key = key(candidate)
        for current in series:
            current_key = key(current)
            if candidate_key < current_key:
                candidate = current
                candidate_key = current_key
    return candidate
```

29

Lines of code for type hints: 7 imports, 4 definitions, and 6 overloaded signatures

26

Lines of code to implement all the documented functionality, with 2 constants, no imports

max overload: postscript

After I contributed **SupportsLessThan** to typeshed, a few things happened:

- Edge cases were discovered where **SupportsGreaterThan** was needed
- **SupportsLessThan** was replaced with **SupportsGreaterThan**, but this exposed symmetric bugs in cases that previously worked
- Both were superseded by **SupportsDunderLT** and **SupportsDunderGT**
- Almost all of their uses were replaced with a new type—the union of both of them—named **SupportsRichComparison**
  - This means most functions that involve comparisons in the standard library now have type hints that accept objects implementing either **<** or **>**. No need to implement both.
- For details, see: https://github.com/python/typeshed/blob/master/stdlib/_typeshed/__init__.pyi

5.

Some protocols in the standard library

Protocols defined in the typing module

PEP 695 – Type | PEP 483 – The T | typing — Suppor | Wumbo | typeshed/__init_ | False-positive er | pylance - Google | Tipagem Pato, a | Duck Emoji | Introduce Mappi | GitHub - PyCQA/

https://docs.python.org/3/library/typing.html#protocols 133%

Protocols

These protocols are decorated with `runtime_checkable()`.

class typing.**SupportsAbs**

An ABC with one abstract method `__abs__` that is covariant in its return type.

class typing.**SupportsBytes**

An ABC with one abstract method `__bytes__`.

class typing.**SupportsComplex**

An ABC with one abstract method `__complex__`.

class typing.**SupportsFloat**

An ABC with one abstract method `__float__`.

class typing.**SupportsIndex**

An ABC with one abstract method `__index__`.

New in version 3.8.

class typing.**SupportsInt**

An ABC with one abstract method `__int__`.

class typing.**SupportsRound**

An ABC with one abstract method `__round__` that is covariant in its return type.

Example using SupportsIndex

```
117  class Vector:
118      typecode = 'd'
119
120      def __init__(self, components):
121          self._components = array(self.typecode, components)
122
149      @overload
150      def __getitem__(self: T, key: slice) -> T:
151          ...
152      @overload
153      def __getitem__(self, key: SupportsIndex) -> float:
154          ...
155      def __getitem__(self, key):
156          if isinstance(key, slice):  # <1>
157              cls = type(self)  # <2>
158              return cls(self._components[key])  # <3>
159          index = operator.index(key)  # <4>
160          return self._components[index]  # <5>
```

Summary

Protocol definition for “a file-like object”

```python
from typing import Protocol

# ...

class _Readable(Protocol):
    def read(self, size: int = ..., /) -> bytes: ...
```

Use `typing.Protocol` to build Pythonic APIs

Support duck typing with type hints

The essence of Python's
Data Model and standard library

Use `typing.Protocol` to build Pythonic APIs

Support duck typing with type hints

The essence of Python's
Data Model and standard library

**Follow the
Interface Segregation Principle**

Client code should not be forced to
depend on methods it does not use

Use `typing.Protocol` to build Pythonic APIs

Support duck typing with type hints

The essence of Python's
Data Model and standard library

**Follow the
Interface Segregation Principle**

Client code should not be forced to depend on methods it does not use

Prefer narrow protocols

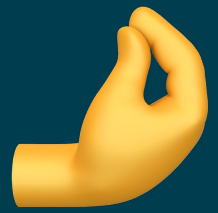
Single method protocols should be the most common. Sometimes, two methods. Rarely more.

Bonus slides: position statement on type hints

Being optional is not a bug of Python type hints.

Being optional is not a bug of Python type hints.

It’s a feature that gives us the power to cope with the inherent complexities, annoyances, flaws, and limitations of static types.



I don’t hesitate to use `# type: ignore` to avoid the limitations of static type checkers when submission to the tool would make the code worse or needlessly complicated.

I don’t hesitate to use `# type: ignore` to avoid the limitations of static type checkers when submission to the tool would make the code worse or needlessly complicated.

Me, in Fluent Python Second Edition

I don’t hesitate to use `# type: ignore` to avoid the limitations of static type checkers when submission to the tool would make the code worse or needlessly complicated.

Me, in Fluent Python Second Edition
@ramgarlic@fosstodon.org