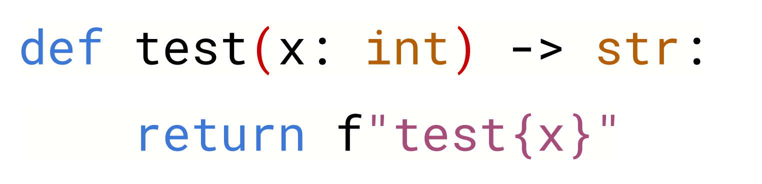
Pyrefly

A Python typechecker and language server

Neil Mitchell, Meta

Python has types!



```python
def test(x: int) -> str:
    return f"test{x}"
```

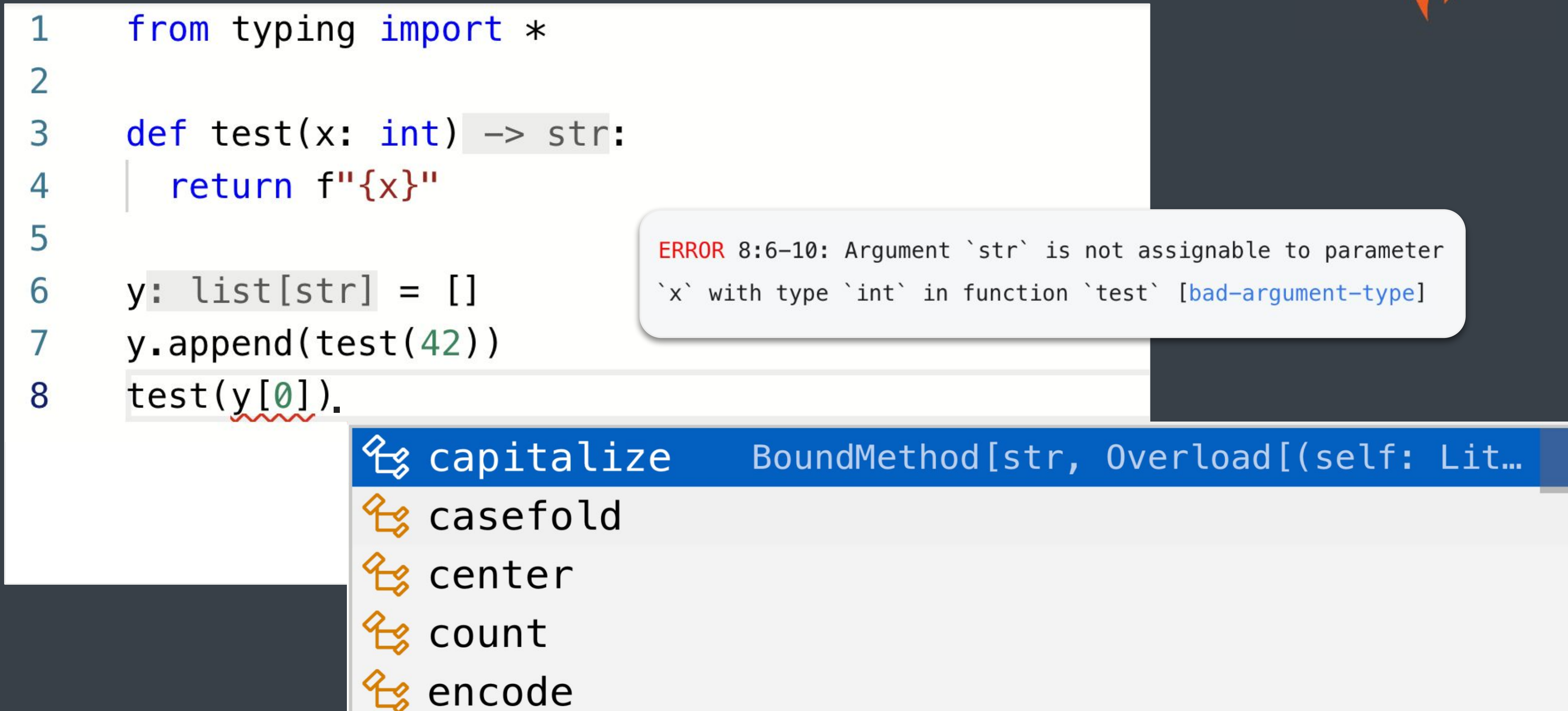
- Since version 3.5 (2015) in PEP 484
- Not checked by default - the interpreter skips* them
    - Runtime checking - typeguard, pydantic, enforce - runtime cost
    - Static checking - mypy, pyright, pyre, pytype, pyrefly, ty, zuban
- More complex than you might think!
    - Generics, literals, higher-order, dataclass transforms, narrowing
    - A standards document, conformance tests - constantly evolving

What is Pyrefly?

- A static checker and language server (IDE/LSP provider)
- An open-source standards-compliant Python type checker
- Fast and parallel (written in Rust)
- From the team behind Pyre (no code in common)
- Lots of type inference

pyrefly.org

Sandbox (pyrefly.org/sandbox)



```python
from typing import *

def test(x: int) -> str:
  return f"{x}"

y: list[str] = []
y.append(test(42))
test(y[0]).
```

ERROR 8:6-10: Argument `str` is not assignable to parameter `x` with type `int` in function `test` [bad-argument-type]

History of Pyrefly

- Meta develops Instagram which is a massive codebase of Python
    - Over 20 million lines of Python
    - 3B monthly active users
    - Over 3.3K daily Python developers across Meta

- In 2017 we started work on Pyre
    - MyPy was considered, but found to be too slow
    - Descendent of Hack (PHP) and Flow (Javascript)
    - Written in OCaml
    - Essential to our development flow

The problems with Pyre

OCaml wasn’t a great choice at Meta

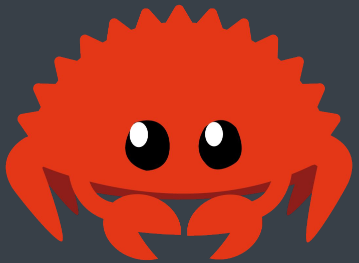
- Didn’t work on Windows
- Parallelism was hard (multiprocess)
- Barrier to open source contributors
- Some problems were solved later, outside Meta

But Pyre had issues too

- Started as abstract interpretation (fixed points, desugaring)
- Started as a command line, hard to pivot to IDE
- Started closed source, never focused on open source users

The birth of Pyrefly

- August 2024 two of us started prototyping MiniPyre
  - 7 prototypes written, constraints, subset based, abstract interpretation…
  - Using Rust (cross platform and fast)
  - Hard bits first: generics, recursion, overloads, `import *`
- October 2024 it was working well, so we started ~~Pyre2~~ Pyrefly
  - Implement features, following the typing spec
  - Implement LSP
- May 2025 we released an alpha at PyConUS
- October 2025 we are still improving, but give it a go (pyrefly.org)



The Python type system

Why types?

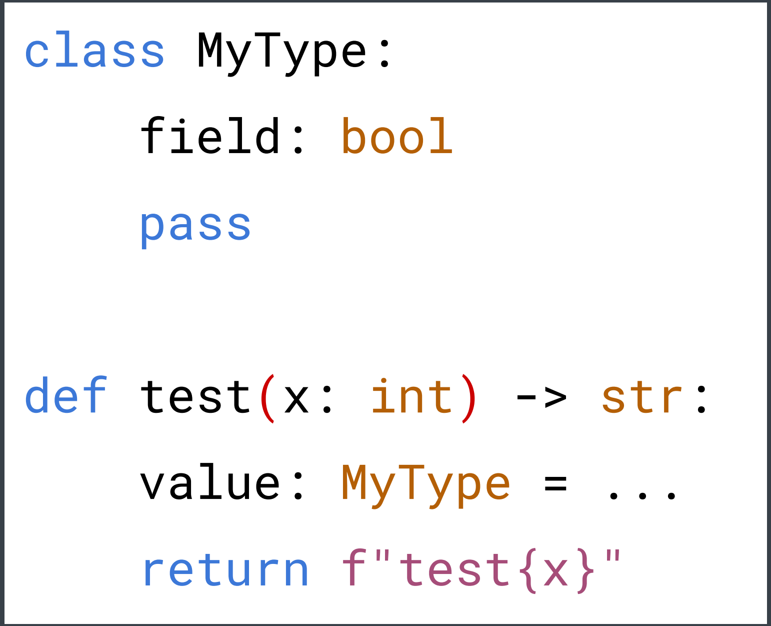
- Python developers are buying in more
    - “2025 is the year of type-checking for Python. I’m so excited.”
- Faster inner loop - run the code less
- Spot typos
- Make corner cases safer
- Understand the code better, documentation, goto-def
- LLM grader
- Write code faster - auto-completion
- Machine checked documentation
- Refactor with peace of mind

The aim

- Give a way to annotate existing code and detect bugs
- The trend has been increasingly complex type system features to model the code people actually write
- Typing conformance test (Pyrefly gets 67.3%, similar to mypy)
- Detailed specification (https://typing.python.org/)
    - But still lots of choices to make as to precise semantics
    - Inference left unspecified

Basic types

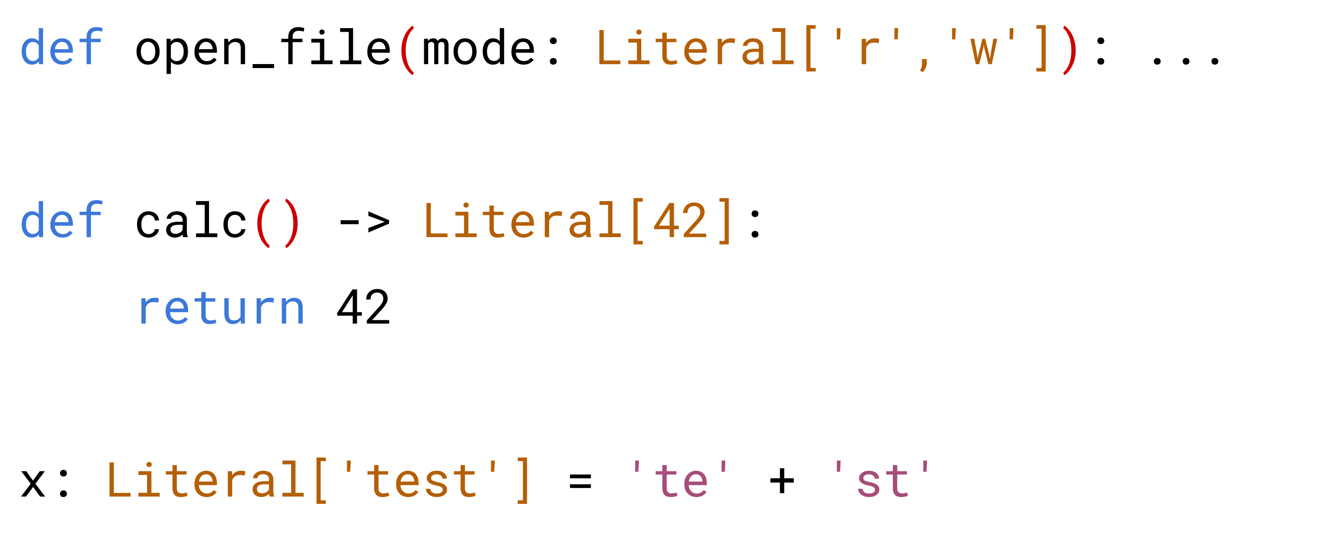
- A few places you can write types
- The basic primitives, plus class types



```
class MyType:
    field: bool
    pass


def test(x: int) -> str:
    value: MyType = ...
    return f"test{x}"
```

Literals

- For `int`, `str`, `bool`, `bytes`, you can write literals that restrict the type



```
def open_file(mode: Literal['r','w']): ...


def calc() -> Literal[42]:
    return 42


x: Literal['test'] = 'te' + 'st'
```

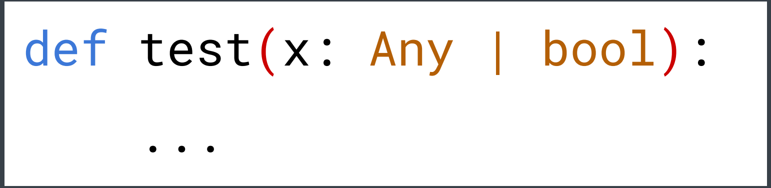
Aliases and forward refs

- You can define type aliases as values
- Types in strings are OK, to deal with execution order
- Is `z = "hello"` a type?

```python
def f(v: "X") -> "Y":
    return v + 1
X = int
print(X("42"))
type Y = "int"
```

Union and gradual types

```
Union[A, B] == A | B
Any ~= int | bool | MyType | …
```



```
def test(x: Any | bool):
    ...
```

- Any represents an unknown static type
- Every type is assignable to Any, and Any is assignable to every type.
- A function parameter without an annotation is assumed to be Any.
- Also Never/NoReturn for the empty union.

Is an unannotated variable implicitly annotated with Any? Systems vary.

Generics

Two forms, using `TypeVar` and generic syntax

`TypeVar` can specify variance, generic syntax infers it

Can specify constraints on both, e.g. must be iterable.

```python
X = TypeVar("X")
def box1(x: X) -> list[X]:
    return [x]


def box2[Y](y: Y) -> list[Y]:
    return [y]
```

Protocols

Structural subtyping

Standard types for iteration, collections, mutable collections, context managers etc

```
class SupportsClose(Protocol):
    def close(self) -> None: …
```

Overloads

A set of “fake signatures”, where only one matches.

```
@overload
def not(x: Literal[True]) -> Literal[False]: ...
@overload
def not(x: Literal[False]) -> Literal[True]: ...
def not(x: Any) -> bool:
    return False if x else True
```

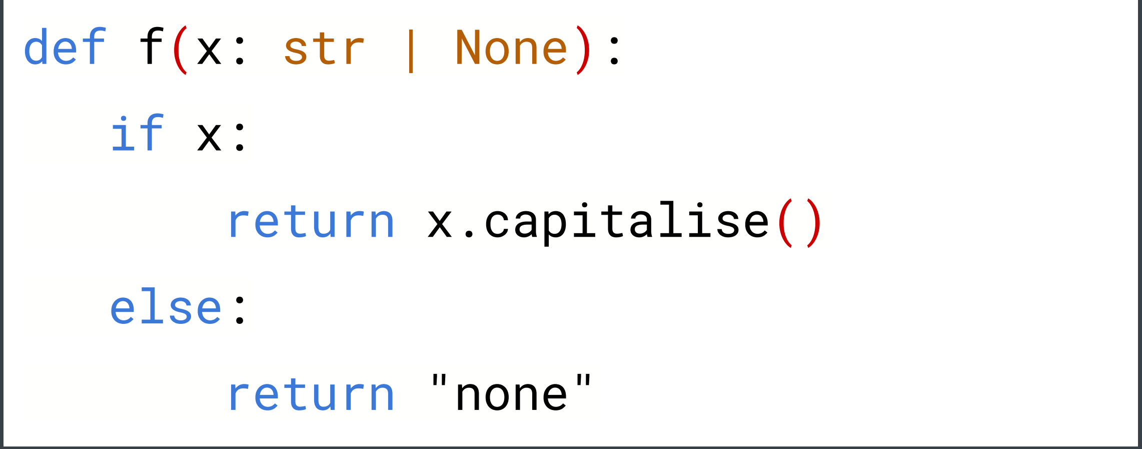
Callable

Specify functions as types. Versatile with generics and concat.
But awkward, since doesn't let you specify argument names.

```python
type Simple = Callable[[int, str], bool]
def f[**P, T](
    call: Callable[Concatenate[int, P], T],
    *args: P.args, **kwargs: P.kwargs) -> T:
    return call(42, *args, **kwargs)
```

Narrowing

Flow control refines the types in branches



```python
def f(x: str | None):
    if x:
        return x.capitalise()
    else:
        return "none"
```

Data class transforms

PEP-681, Dec 2021 - SQL/ORM style solution

Typeshed library

- Attempts to give types to everything in the standard library.
- A little awkward - the standard library was written without types.
- Often there is what the library does precisely, and what you would have made it do if you knew about types.
    - E.g. protocols are close, rather than precise.

The Pyrefly design

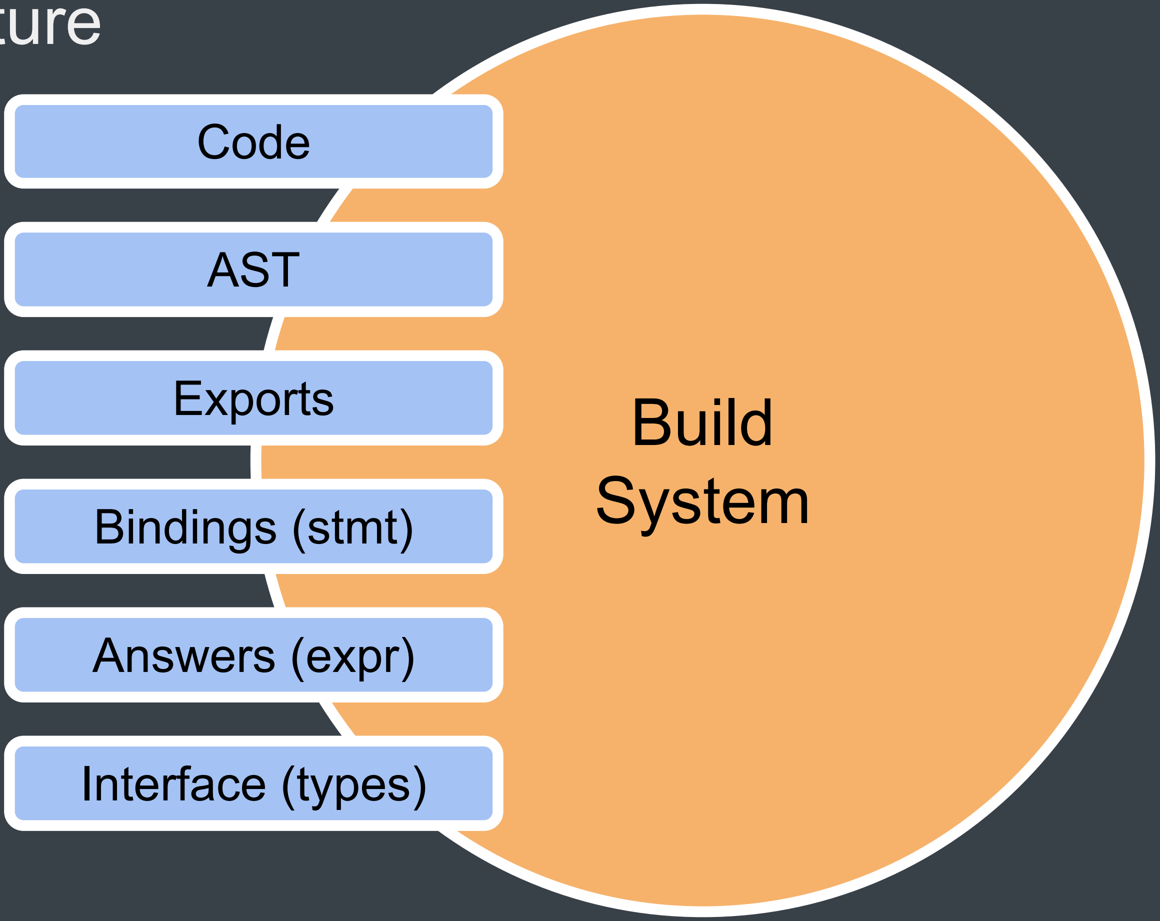
Design goals

- Efficient in time and memory. Incremental. Must cope with Instagram.
- Flexible. Command line. IDE/LSP. Buck. MCP. Dune?
- Hackable. Solid principles. Simple code.
- Good user experience.
- Deal with circular import graphs.

Conclusion*: Build system, operate at the *file* level, evict old data.
Implication: Each file must be super quick (a few ms)

* Caveat: I describe everything in my life as a build system

Architecture



Code

AST

Exports

Bindings (stmt)

Answers (expr)

Interface (types)

Build System

1. Code

- Read the code off disk (or if LSP, from LSP messages).

2. AST

- Parse the code into an AST.
- Uses the Ruff parser from Astral (which is amazing)
  - Always succeeds, error correcting parser
  - Which is like the ultimate fuzzer…

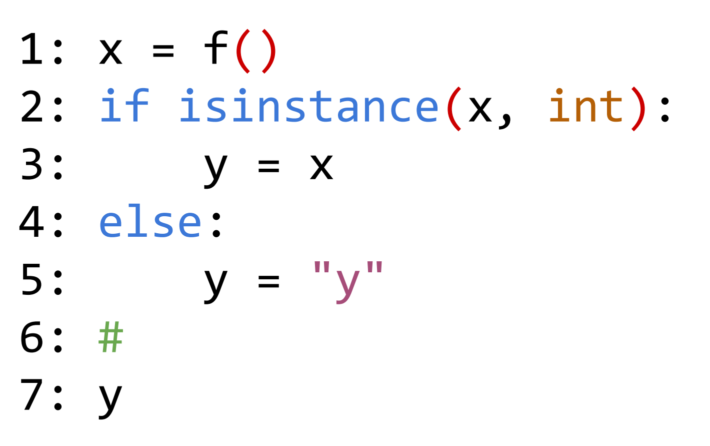
3. Exports

- What symbols does each module export
- Modules always export all their imports, apart from builtins
- Not trivial because of import * - can require a fixed point
- Required to know which module provides a given value

```
from module1 import *
from module2 import *
from module3 import y

x = z
```

4. Bindings (statements)

- How do statements relate to each other, where to variables flow
- Key -> Value mapping, where Value’s contain other Key’s



```
1: x = f()
2: if isinstance(x, int):
3:     y = x
4: else:
5:     y = "y"
6: #
7: y
```

```
Use(f@line1) => Forward(...)
Def(x@line1) => Expr(x, f())
Use(x@line2) => Forward(Def(x@line1))
Use(int@line2) => Forward(...)
N(x@line2, line3) => N(Use(x@line2), IsInstance(int))
N(x@line2, line4) => N(Use(x@line2), NotInstance(int))
Use(x@line3) => Forward(N(x@line2, line3))
Def(y@line3) => Expr(x)
Def(y@line5) => Expr(“y”)
Phi(y@line6) => Phi(Def(y@line3), Def(y@line5))
Use(y@line7) => Forward(Phi(y@line6))
```

5. Answers (expressions)

- Solve the bindings that were created to produce types.
- `Key -> Thunk<Type>` mapping
  - Allows cycles
  - Use Var as a placeholder for unknown types, and unification.
- Lots and lots of code for each special type object in Python.

6. Interface (types)

- Types of exported symbols only.
- Subset of the Answers (less memory)

What are the types of x?

```
x = [1]
x = [1, "test"]
x: Literal[1] = [1]
x = []
```

The magic Var (unification, inference, loops)

```
x = list[?1]
add = Callable[[list[?2], ?2], None]
list[?2] = list[?1] ⇒ ?2 = ?1
?2 = Literal[1] ⇒ ?2 = int (generalise), ?1 = int
```

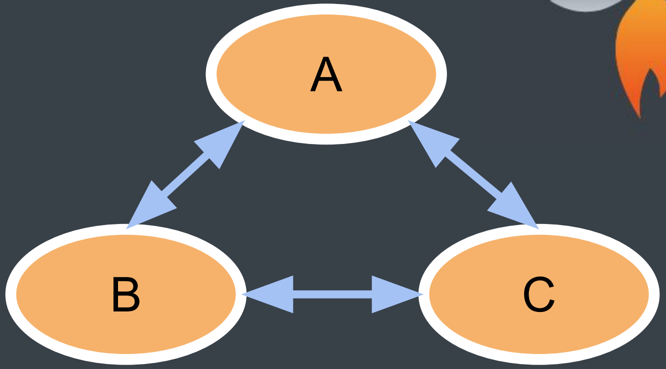
```
x = []
add(x, 1)


def add[T](a: list[T], b: T) -> None: ...
```

Eviction

- For most modules, we throw away AST, Bindings, Answers when done
- For files open in the IDE, we keep everything to support goto-def etc
- We always keep code around because the disk might change
- If you ask for the type of an export, we try the interface first, if not, the answers
    - Answers is required to deal with cycles
- Prioritise modules that are “nearly finished” to reduce memory

Eviction rules are simple because each module goes through sequential steps.

Incrementality (with cycles)



- A changes, what should we invalidate?
- Pyrefly says
  - Optimistically invalidate only A
  - Compute A using stale values of B/C
  - If the interface of A changes, invalidate B/C
  - Compute B/C using the last iteration of A
  - If A changes (since it changed before) invalidate the cycle
- Pros: Usually only one file invalidates
- Cons: Might compute a module more than once, and less parallelism

Performance

We want to check on every keystroke. We can check 1.85M lines/second.

Approximately Answers is 10x the cost of Bindings, which is 10x Exports.

We freely clone Type all the time. Should really have a heap…

Lots of profiling. Super easy to go quadratic.

With lots of threads, and Rust, the expensive things are:

- Thread communication
- Locks

Both are in the build system, which has been optimised a fair amount.

Transactions

An IDE does lots of things at once:

- Indexing (for find references)
- Checking a file that changed
- Answering queries

We have a transactional build system, with explicit commit. Never have to wait.

Can only have at most one mutable transaction at a time (so commit always succeeds)

Extensibility

- Model of features built in to the core
- Build systems like Buck/Bazel/Dune?
  - Buck integrated into the core, using queries and Bxl
  - Very open to further integrations
- Mypy plugins?
  - Supports Pydantic rules natively
  - Aiming to add some more special cases
  - Shape types one day?

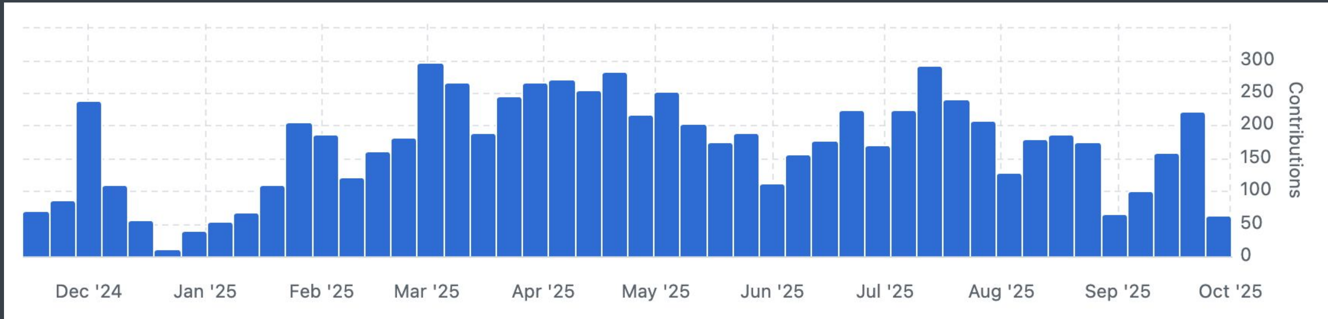
Recursion

```
struct Thunk<T>;
```

- If this thread is already calculating this Thunk, create a Var.
- If not, solve the binding.
- When the calculation completes, bind the Var.

Open source

- We have gained much from open source!
  - Python itself
  - Python typing specification, plus existing checkers (Pyright, Mypy etc)
  - Ruff parser
  - Open source Python projects, e.g. PyTorch (now has Pyrefly in shadow)
- MIT license, https://github.com/facebook/pyrefly
- Delighted to accept pull requests, all issues are on issue tracker



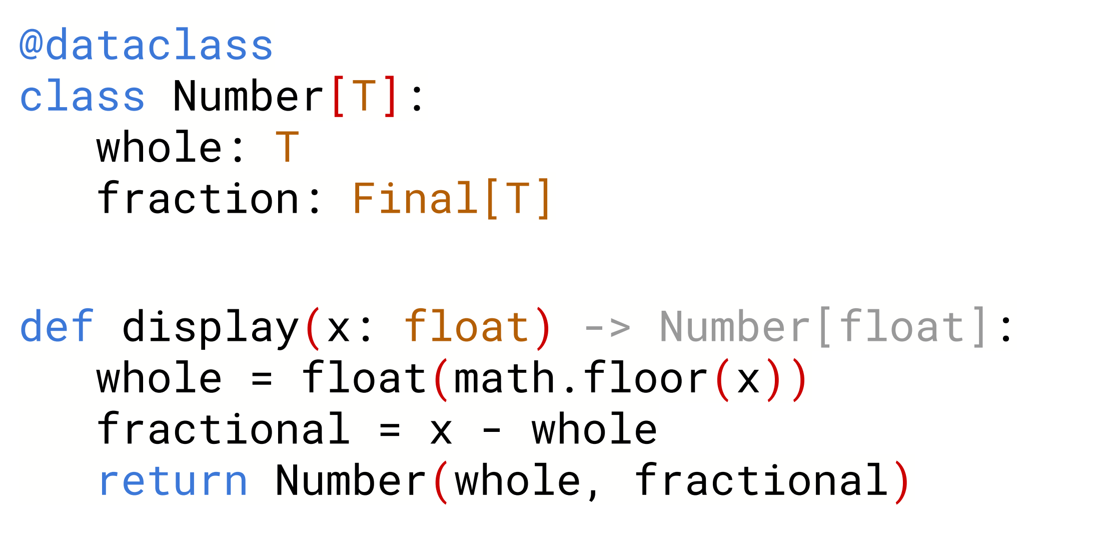
The journey of autocomplete

```
display(3.142).fraction
```

```
from typing import *
from numbers import *
```

- Find the type of `display(3.142).fraction`
- First, find `display`
  - Might come from `typing` or `numbers`
  - Figure out the export table from each
  - Which might require a fixed-point of recursive * imports…

The journey of autocomplete (2)

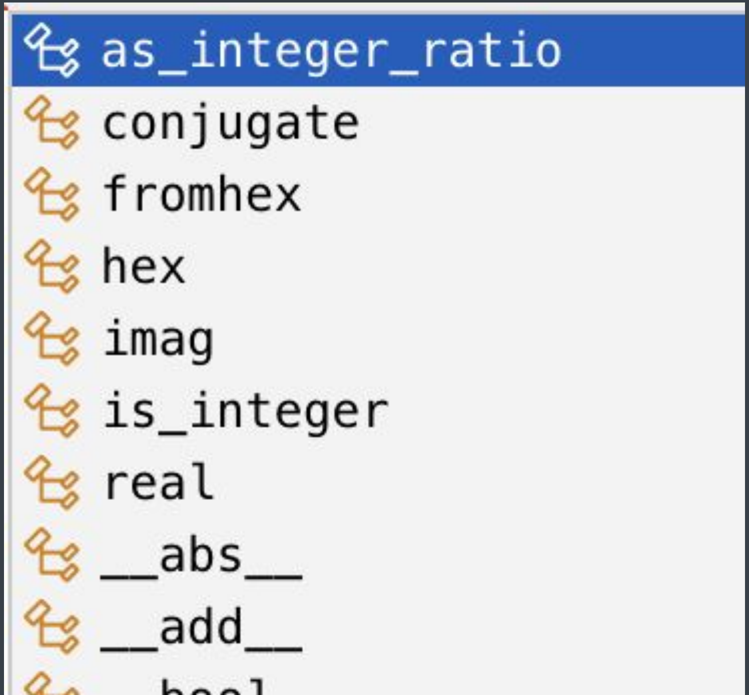


```
@dataclass
class Number[T]:
    whole: T
    fraction: Final[T]


def display(x: float) -> Number[float]:
    whole = float(math.floor(x))
    fractional = x - whole
    return Number(whole, fractional)
```

- Interpret **`@dataclass`**
- Infer types for each variable
- Infer the return type
- Instantiate some generics
- Understand **`Final`**

The journey of autocomplete (3)

```
display(3.142).fraction.
```

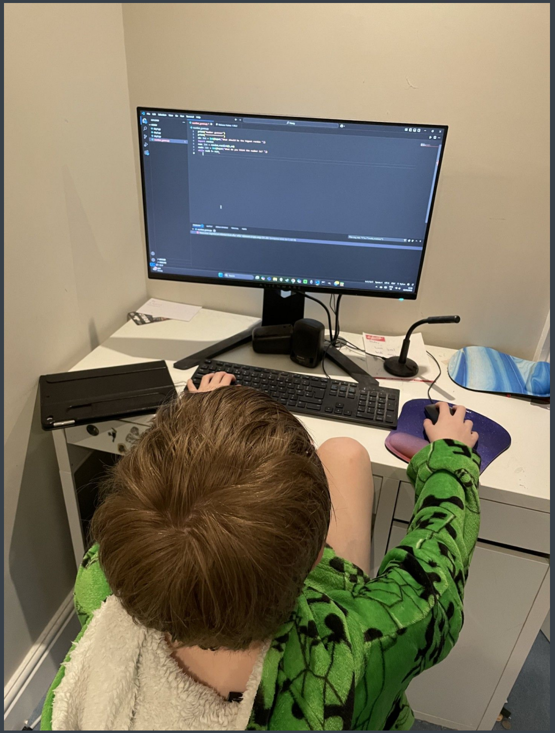


```
class float:
    def __new__(cls, x = ...) -> Self
    @classmethod
    def fromhex(cls, x: str) -> Self
    @property
    def real(self) -> float
    def conjugate(self) -> float
    def __add__(self, x: float) -> float
```

- Now we know we have `float`
- Figure out what methods it has

Why not Pyrefly?

- It is an alpha - lots of known bugs
- You will probably find bugs, most of which we will fix
- But you will get a sticker (if you are here)



The team (+ over 100 contributors)

Questions?

pyrefly.org