CS3430 S26: Scientific Computing
Why Type Annotations Matter in Python

Vladimir Kulyukin
SoC – CoE – USU

Spring 2026

Motivation

In this course, all provided Python source code uses type annotations. This choice is intentional. It is not about turning Python into a statically typed language, nor is it about enforcing rigid type rules.

Instead, annotations support clarity, correctness, and long-term maintainability, especially in scientific computing, where misunderstood data flows and silent errors are common.

This reading explains why annotations are useful today and how they align with Python's long-term design direction, as reflected in PEP 563 and PEP 649.

What Type Annotations Are (and Are Not)

Type annotations allow us to describe the *intended* types of variables, function parameters, and return values.

For example:

```
def central_diff(f: Callable[[float], float],
                 x: float,
                 h: float) -> float:
    ...
```

Annotations do *not* change how Python executes code. They are ignored at runtime unless explicitly inspected. Their purpose is to:

- document intent,
- support reasoning about code,
- enable static analysis tools,
- catch conceptual errors early.

Python remains dynamically typed. Annotations add information; they do not impose constraints unless we choose to use tools that act on them.

Why Annotations Matter in Scientific Computing

Scientific computing code often manipulates:

- numerical scalars,
- vectors and arrays,
- functions as first-class objects,
- approximations with implicit assumptions.

Errors in such code are frequently *semantic*, not syntactic. The code runs, but computes the wrong thing.

Annotations help by making assumptions explicit:

- Is a value a scalar or a vector?
- Is a function expected to be vectorized?
- Does a function return a number or a table?
- Is a parameter dimensionless or a step size?

In CS3430, annotations are part of how we make numerical reasoning visible.

The Problem with Eager Annotation Evaluation

Originally, Python evaluated annotations eagerly at function definition time. This caused practical problems:

- forward references required string hacks,
- importing modules could trigger heavy dependencies,
- annotations could slow startup or fail unexpectedly.

As Python typing grew more expressive, these issues became more severe. This led to PEP 563.

PEP 563: Postponed Evaluation of Annotations

PEP 563 proposed that annotations be stored as *unevaluated expressions* rather than immediate objects. Conceptually:

- annotations become descriptions, not runtime values;
- evaluation is deferred until explicitly requested;
- forward references work naturally.

This change emphasized an important idea: annotations are *metadata*, not execution logic. However, PEP 563 introduced complexity and ambiguity about when and how annotations should be evaluated.

PEP 649: Deferred Evaluation (The Long-Term Direction)

PEP 649 refines the idea of postponed evaluation by defining a clear, well-specified mechanism for *deferred* annotation evaluation. The key goals are:

- preserve runtime performance,
- avoid import-time failures,
- make annotation access explicit and predictable,
- support increasingly rich type systems.

PEP 649 represents Python's long-term commitment to annotations as a first-class language feature, even in a dynamically typed setting.

Why This Matters for Us

By writing annotated code now, we are:

- aligning with Python's future, not fighting it;
- learning to express assumptions precisely;
- producing code that is easier to test, review, and maintain;
- preparing for tools that reason about correctness.

In this course, annotations are part of how we avoid "cargo cult programming": they force us to think about what our code *means*, not just whether it runs.

Bottom Line

Type annotations are not about rigidity. They are about communication — with our future self, with collaborators, and with analysis tools.

Scientific computing depends on clarity of assumptions. Annotations help make those assumptions explicit.

That is why we will use them as much as possible in CS3430 S26.