# PurePy: A Functional Subset of Python for Scientific Computing

TOBIAS ANTENSTEINER, Karlsruhe Institute of Technology, Germany
CRISTINA DAVID, University of Bristol, UK
WEILIN HE, University of Bristol, UK
NIKOLAUS HUBER, University of Cambridge, UK
DOMINIC ORCHARD, University of Kent, UK and University of Cambridge, UK
JACOB PAKE, University of Kent, UK
ROLY PERERA, University of Cambridge, UK and University of Bristol, UK

The Python programming language has grown in popularity over the years and is now the common language of choice in data science and scientific computing applications. With machine-generated code, e.g. via LLMs, becoming increasingly common, the need for verification techniques applicable to Python is becoming more pressing and is widely sought after. Python however lacks an official formal semantics. Moreover, mutable state, dynamic binding, and order-dependent module loading make Python programs hard to reason about. Against this backdrop, we observe that Python is becoming a kind of shared, common notation reused by many new programming language efforts that seek to provide increased correctness guarantees and/or improved performance. Its widespread use, and hence prevalence in the training data of LLMs, means that adopting Python-like syntax may lower barriers to entry both for human programmers and LLMs. Such Python-inspired languages must make particular individual choices about what to include or exclude from Python. To bring some consensus and provide a strong foundation for further work, we introduce PurePy: a statically typed, lexically scoped, immutable subset of Python. We formally define both the static and dynamic semantics of PurePy, alongside a versioned language specification, and a verified reference interpreter and type checker. The type system accepts only programs that are well typed under mypy 2.3.1. The dynamic semantics is intended to produce the same result as Python 3.12 for every terminating run with defined behaviour. We test this agreement by comparing our reference type checker and interpreter against mypy and multiple Python implementations (CPython, GraalPy, Pyodide) via a suite of 896 PurePy programs. In some cases our approach identified bugs or points of disagreement amongst implementations. Overall, our goal is for PurePy to act as a platform for further programming language research, since any new languages and program analyses built on PurePy inherit its formal semantics and, with it, compatibility with Python. We also assess the claim that such a subset, as a basis for new language designs, can still be viably targeted by LLMs; we compare code generation into PurePy, Typed Python and Python on standard benchmarks. Lastly, we discuss adjacent language projects based on similar assumptions, including two of our own which build on PurePy and which are designed to better support computational science goals.

## 1 Introduction

Python remains the dominant language on GitHub for AI and data science, with scientific computing among the fields driving its continued use [GitHub 2024, 2025]. Moreover, scientific results increasingly rest on programs written with AI coding tools [O'Brien et al. 2025]. AI coding tools favour Python when the choice of language is left open, choosing it in most cases even where

Authors' Contact Information: Tobias Antensteiner, Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany; Cristina David, School of Computer Science, University of Bristol, UK; Weilin He, School of Computer Science, University of Bristol, UK; Nikolaus Huber, Department of Computer Science and Technology, University of Cambridge, UK; Dominic Orchard, School of Computing, University of Kent, UK, Department of Computer Science and Technology and University of Cambridge, UK; Jacob Pake, School of Computing, University of Kent, UK; Roly Perera, Department of Computer Science and Technology, University of Cambridge, UK, School of Computer Science and University of Bristol, UK.

2026. ACM 2475-1421/2026/6-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

another language would serve better [Twist et al. 2026]. Most of the code-generation benchmarks surveyed by Siddiq et al. [2024] are in Python. Models generate Python more accurately than languages with less training data, such as OCaml and Racket [Cassano et al. 2023], a gap which fine-tuning closes only in part [Cassano et al. 2024].

Code generated by a machine is seldom fully understood, or even read, by the people who depend on it, so the question of whether a program does what it should has become pressing. Recent benchmarks measure how often generated scientific code is correct [Chen et al. 2025; Duston et al. 2025; Siegel et al. 2024; Tian et al. 2024], but answering this question for a particular program requires detailed inspection of the code. More formal approaches to verification need a specification, which scientific programs seldom have.

Partly in response to these concerns about AI-generated code, the past few years have seen a growing number of new Python-like languages with a verification focus, leveraging Python's familiar syntax and widely available tooling: mandatory static typing [Lu et al. 2023], translation to program verifiers [Cheema 2021], and dependent types with machine-checked proofs to increase trust in generated code [BendLang 2026]. Python has also proved a popular foundation for other language efforts, including for teaching [Padhye et al. 2019; Sunderraman 2025], ahead-of-time compilation for performance [Ancona et al. 2007; Shajii et al. 2023], scientific computing [Deep Forest Sciences 2026], and immutability [Johnson et al. 2025]. These efforts adapt or build on Python in different ways, as approximate subsets, dialects, or new languages borrowing ideas and syntax.

One thing these efforts lack is a common core: a well-defined fragment of Python which each could adopt, and whose specification, conformance suite and guarantees each could inherit instead of establishing afresh. Such a core remains elusive. Python itself has no official formal semantics, and the language is sufficiently large and complex that the formalisations constructed by researchers [Guth 2013; Köhl 2021; Melançon et al. 2023; Politz et al. 2013] are usually limited to certain aspects or idealisations. A formal semantics of full Python would perhaps not be the best starting point for new languages, which might prefer a smaller and cleaner set of assumptions. Mutable state, dynamic binding, and sensitivity to module load order, among other features, make Python programs hard to reason about, for programmers and equally for researchers who want to explore new ideas in programming languages.

In this paper, we propose a language defined as a true subset of Python, omitting hard-to-reason about behaviours but otherwise behaving identically to Python. Our aim is for this to serve as a foundation for new developments which support emerging use cases such as AI-generated code and type systems for scientific computing. Throughout the paper, by "Python" we mean Python 3.12, as defined by the *Python Language Reference* [Python Software Foundation 2023] and as broadly realised by the leading implementations CPython, GraalPy, and Pyodide. PurePy is a statically typed, lexically scoped, and immutable subset, with a formal static and dynamic semantics. Every well-formed PurePy program is valid Python, and the dynamic semantics is intended to produce the same result as Python for every terminating run with defined behaviour (§2.1). A conformance suite tests this property on the three implementations (§2.1.1). The type system accepts only programs accepted by mypy 2.3.1, so that PurePy programs are checked using a workflow familiar to Python programmers. PurePy excludes Python idioms which rely on mutation, dynamic name resolution, or Python's lax typing (§2.2). Several of the excluded idioms, such as wildcard imports, are also discouraged by PEP 8,[1] so the exclusions partly follow established Python practice and the subset is thus lightly opinionated as well as more disciplined.

We take the position that machine-generated code changes the economics of language design. Features such as mutation in-place, implicit coercion, and late binding may seem convenient to a programmer, but the cost in reasoning falls on whoever must read the code or trust the result.

[1] https://peps.python.org/pep-0008/

When the author is a machine, the conveniences become less relevant, and strictness and formal guarantees become more important. A stricter language with a different syntax, however, forfeits the knowledge of Python already present in current models. The experimental evaluation of §6 measures the cost of the restrictions of PurePy to a code generator: how often it is still able to produce a correct solution, at what extra cost, and how much of that cost is due to the design of PurePy rather than to language features not yet included in the specification.

### 1.1 Contributions

Following an overview of the design principles (§2), we present our contributions as follows:

- **Specification**: We define PurePy as a versioned language specification,[2] available online. References to figures and sections therein are marked with ↗.
- **Syntax** (§3): We define the grammar of PurePy as a subset of the Python 3.12 grammar.
- **Well-formedness** (§4): We give a type system which accepts only programs well-typed under mypy, with annotations mandatory and names resolved statically.
- **Operational semantics** (§5): We give a semantics designed so that every terminating run with defined behaviour produces the same result under Python, and prove type soundness.
- **Mechanisation**: We mechanise the type system and semantics in Isabelle/HOL and generate the reference type checker and interpreter from the mechanisation (§5.2.2). Each definition and result in the paper names its Isabelle counterpart, listed by theory in `mechanisation/names.txt` of the artifact.
- **Playground**: We provide an online playground at https://pure-py.github.io, which runs every example in this paper, linked by the icon </> at the right of its first line.
- **Conformance suite**: We provide a suite of 896 programs (§2.1.1), run under CPython, GraalPy, Pyodide, mypy and the reference implementation. The tests are listed at https://pure-py.github.io/conformance-suite, each opening in the playground.
- **Evaluation** (§6): We compare AI code generation into PurePy, Typed Python, and Python on standard benchmarks.

The specification, mechanisation, reference implementation, conformance suite and playground all accompany the paper as the artifact and via the project website at https://pure-py.github.io. §7 presents two research languages of our own built on PurePy and discusses other related work, and §8 concludes with next steps.

## 2 Overview

This section relates PurePy to Python, defines compliance for implementations of PurePy and describes the conformance suite (§2.1), and sets out the design principles of PurePy, each framed as an aspect of Python's semantics which is ruled out by our design (§2.2). We then describe the Python features, such as exceptions and general-purpose classes, which are compatible with our design principles but deferred to a later version of the specification (§2.3).

### 2.1 Python compatibility and PurePy compliance

PurePy is a subset of Python, defined by static rules: a Python program is *a PurePy program* if it has the syntax of §3, and a PurePy program is *well-formed* if it satisfies the well-formedness rules of §4. Every well-formed program is also well-typed under mypy, except for programs rejected because of a reported mypy bug (Figure 1.2–1 ↗). Not every behaviour excluded by PurePy can be rejected statically (§2.2.5); runs which reach one of these *dynamically excluded* operations, such as comparison of two functions, have undefined behaviour, and abort with undef. Otherwise

[2] This paper describes version 0.27.9.

| Python | PurePy rejects | | | | PurePy accepts | |
|---|---|---|---|---|---|---|
| | Static | | | Dynamic | | |
| | *parser* | *type checker* | *not yet* | aborts (undef) | *defined* | *Total* |
| `SyntaxError` | **17** | **2** | **1** | | | 20 |
| Other exception | **4** | **100** | **3** | **3** | **29** | 139 |
| No exception | **29** | **311** | **74** | **7** | **316** | 737 |
| Total | 50 | 413 | 78 | 10 | 345 | 896 |

Python error prevented · Excluded by design · Same result as Python
Cannot occur · Omitted from this version · Well typed under mypy

Fig. 1. Conformance tests, by outcome under Python and PurePy

PurePy agrees with Python: every well-formed program is valid Python, in that it does not raise `SyntaxError` under Python, and every terminating run produces the same result.

A language implementation conforms to the PurePy specification if, on every run of a well-formed program with defined behaviour, it produces the Python result; on any other run it may either abort and report undef, following the semantics, or produce the Python result.

*2.1.1 Conformance suite.* The conformance suite consists of 896 tests, run under CPython, GraalPy and Pyodide, under mypy 2.3.1, and under the reference type checker and interpreter generated from the mechanised semantics (§5.2.2). Figure 1 counts the tests by outcome under Python (row) and under PurePy (column); the hatched cells are empty because every well-formed program is valid Python. CPython, GraalPy, and Pyodide give the recorded Python outcome of every test, and mypy accepts every well-formed test. The reference type checker must accept the well-formed tests and reject each other test for the reason named by its column. The reference interpreter must give the recorded Python result for each test in the *defined* column, and abort with undef for each test in the *aborts* column, so the *defined* column tests the agreement of §2.1 on three implementations of Python. Disagreements caused by bugs in mypy or a Python implementation were reported to the maintainers and recorded in the suite as expected failures.

The first line of the main module of each test of an ill-formed program is a comment citing the judgement at which the program fails and, if possible, a rule. The reference type checker names the judgement or rule in its reason for rejection, and the runner of the suite checks that the reason given by the type checker matches the citation in each test. The runner also checks that each of the 81 rules and 6 judgements under which the mechanisation can reject a program is named in the specification and cited by at least one test. These checks keep the specification, mechanisation and suite in correspondence. The specification defines citations precisely (§1.2.1).

## 2.2 Design principles

PurePy guarantees the following properties: every name has a unique lexical scope; variables, once assigned, denote fixed values; the members of a module are fixed once the module has loaded; and every name is bound before any reference to it is evaluated. Python programs, even well-typed ones, do not in general satisfy these properties, yet PurePy characterises a substantial subset of typed Python within which all four hold.

PurePy permits three kinds of effect: non-termination, aborting with a run-time error such as division by zero, and printing to the console. The first two are unavoidable in a general-purpose language. Errors of different kinds are reported differently, so a program with two possible failures reveals which failure is reached first. The order of evaluation, including the order in which modules

load, is therefore part of the semantics, and failing programs fail in the same way under every implementation (§5.2.3). Printing, in the form of Python's `print`, we consider to be a benign effect, since it reports on a computation without influencing its result.

*2.2.1 Static typing.* Unlike Python, PurePy is statically typed: a program is well-formed only if it is well-typed, under a type system compatible with mypy (§2.1). Function parameters, return types and dataclass fields must all carry type annotations (§4). References to attributes of objects must name fields of the class, and constructor calls and constructor patterns must match the arity of the class exactly. `AttributeError` and `TypeError` exceptions raised by Python for unknown attributes and arity mismatches are therefore statically ruled out in PurePy.

*2.2.2 Immutability.* PurePy variables must be declared before or together with their first assignment, must be assigned before use, and may not be reassigned. This is a key point of difference with Python, where variables are reassigned freely and uses of unbound variables are detected only at run time. Declarations are variables with a type annotation such as `x: int` (PEP 526[3]), optionally with an initial value. Similarly to Java's 'blank finals' [Gosling et al. 2025, §16], single assignment and assignment before use are checked along every control-flow path (*definite assignment*, §4.3). Variables assigned in only one branch of a conditional are therefore possibly unassigned afterwards, so may be neither used nor assigned, as on the left below.

```
def foo(b: bool) -> int:
    x: int
    if b:
        x = 3
        y: int = x + 1
    x = 0  # error: x possibly assigned
    return x
```

```
v: tuple[int, int] | None = (1, 2)
match v:
    case (x, y):
        print(x + y)
# error: x possibly unassigned
print(x)
```

Python assigns `x` twice when `b` is true. Python scopes pattern variables as it does assignments, so they remain bound after the `match` statement. PurePy treats each `case` as a branch and its pattern as an assignment of the pattern variables, so after the match a variable is definitely assigned only if every case binds it or returns, and has the join of its types in those cases. A match leaving some part of the scrutinee type unmatched may fall through, as on the right above; Python prints `1`, the binding of `x` having escaped the match.

Since variables denote fixed values once assigned, features which update variables or values in place are excluded: `global` and `nonlocal`, `del`, augmented assignment (`+=` and so on), the walrus operator (`:=`), `for` and `while` loops, and with them `break` and `continue`, as well as the generator constructs `yield` and `yield from`. Assignment targets are variables, so item assignment such as `xs[0] = v` and attribute assignment such as `p.x = v` are excluded, and methods updating values in place, such as `list.append`, are absent.

Iteration is instead expressed by recursion and comprehensions, the one context in which `for` remains. In Python comprehensions update the generator variables on each iteration, and closures capture bindings rather than values; in PurePy a lambda in the body or a later qualifier of a comprehension is therefore not permitted to capture a generator variable:

```
[lambda x: x**i for i in [1, 2, 3]] # error: lambda captures i
```

In Python, each lambda observes the updates to `i` when called, so all three compute `x**3`.

[3] https://peps.python.org/pep-0526/

*2.2.3 Statically-resolved names.* Python resolves names to scopes statically but checks only at run time that variables are bound. Within a function, which acts as a single scope, uses may precede variable declarations, and fail only when evaluated. PurePy scopes are module, function, lambda and comprehension bodies, as in Python, and the builtins form an outermost scope around every module. We mandate the following constraints of mypy: scopes declare each name at most once, including as a parameter, imported name or pattern variable, and declarations shadow the same names in enclosing scopes throughout the declaring scope. Definite assignment (§2.2.2) then forbids uses of variables before their declarations, as on the left below; Python raises `UnboundLocalError` when `f` is called. Module-level uses of builtin names before declarations of the same names are rejected for the same reason, as on the right, whereas mypy accepts such uses as uses of the builtins.

```
y: int = 7
def f() -> int:
    x: int = y  # error: y not yet bound
    y: int = 8
    return x
```

```
# error: len not yet bound
n: int = len([1])
len: int = 2
```

Function definitions require a different treatment, in order to support mutual recursion. Python resolves references to functions when they are evaluated; two functions may refer to each other in either order, provided both are bound when the references are evaluated. PurePy instead treats each maximal contiguous block of `def` statements, or *mutual def region* (§3.2), as a single binding of a set of simultaneously defined names, checking each body under the assumption that every name of the block is definitely assigned (§4.3). Intervening statements separate definitions into distinct regions, so the following is rejected statically:

```
1  def even(n: int) -> bool:
2      if n == 0:
3          return True
4      # error: odd not yet bound
5      return odd(n - 1)
6  x: bool = even(0)
7  def odd(n: int) -> bool:
8      if n == 0:
9          return False
10     return even(n - 1)
```

The definition of `even` (lines 1 to 5) forms a region by itself, in which `odd` is not yet bound. Python accepts this program, since the call `even(0)` never evaluates `odd`.

*2.2.4 Static reachability.* Python accepts statements and match cases which are statically unreachable; PurePy rejects both. A statement sequence returns only if its last statement returns and every earlier statement assigns, so a statement following a `return` is rejected (§4.3), as on the left below. Cases are unreachable if their patterns are covered by the patterns of earlier cases (§4.4), as on the right. Under mypy, unreachable code is only an optional warning, and the case on the right is not detected even then, since matching `42` does not narrow `int`.

```
def foo(x: int) -> int:
    return x + 1
    return x  # error: unreachable
```

```
n: int = 42
match n:
    case 42: print(1)
    # error: unreachable
    case 42: print(2)
```

*2.2.5 Strict typing of tests, equality and patterns.* Several Python constructs accept operands of types where a stricter type discipline would report a type error. Any value serves as the test of an `if` or `assert` or as an assertion message; `1 == "1"` is `False` rather than an error, and `True == 1` holds because Booleans are numbers; list and tuple patterns match list and tuple values interchangeably; and constructor patterns may omit fields.

PurePy rejects each of these statically. Tests must be `bool` and assertion messages `str`. The operands of == and != must have comparable types (see §4), and `bool` and `int` are not comparable. List and tuple patterns match only values of the respective type, and constructor patterns must match every field of the class. Python defines equality on functions by comparing 'identity' (effectively, pointer equality), so two textually identical lambdas are unequal. Identity is inexpressible in PurePy, so comparisons at function type are not permitted. Three cases are not decided by the types of the operands, so cannot be rejected statically and instead abort with `undef` at run time: at type `object`, equality between a Boolean and a number or between two functions; and equality of containers whose compared elements are both nan, which is decided by identity in Python.

*2.2.6 No load-dependent meaning.* In Python the meaning of a name depends on the sequence in which modules are loaded, which is a property of the whole program rather than of any one module. Imports may appear anywhere; submodules become attributes of their packages only when loaded, possibly by an import elsewhere in the program; and an attribute of a module may be set both by assignment and by loading a submodule of the same name, the later of the two overwriting the earlier. Failures surface at run time as `ModuleNotFoundError`, `ImportError` or `AttributeError`.

PurePy fixes the meaning of every name statically. We mandate the following constraints of mypy: imports may only name modules of the program, and from-imports only members of the source module. We additionally require that imports appear at the start of a module and do not form cycles, that qualified references resolve through the imports of the referring module to definitely assigned members, and that modules do not bind the names of their submodules. The following two-module program is accepted by Python and rejected by PurePy:

```
# module a              # module __main__
import pkg.sub          import pkg
                        import a
                        print(pkg.sub.y)  # error: pkg.sub is not imported here
```

Python accepts this because importing `a` loads `pkg.sub`; without that import, Python raises `AttributeError`. PurePy rejects the reference unless the main module itself imports `pkg.sub` (§4.5). §B.1.4 lists the excluded import forms, including wildcard imports.

### 2.3 Features not yet included

Python features absent from PurePy are either *excluded by design*, because they conflict with a principle of §2.2, or *omitted from this version*, meaning that they are compatible with the design and are likely to appear in a future version. The reference implementation reports which of the two kinds of exclusion applies to a rejected program. General-purpose classes are omitted; PurePy classes must be dataclasses consisting only of fields, with no methods. Subscripting, iteration, and equality have the fixed meanings of §5 since the special methods extending these operations in Python are absent too. Exceptions and `with` statements are omitted. First-class modules and classes are also omitted; PurePy module names may be used only to access members, and class names only in constructor calls or patterns. Figure B.2–1 lists the features omitted from this version.

## 3 Syntax

*Notation.* Sequences $\vec{x}$ denote $x_1, \ldots, x_n$ for some $n$. Inside the brackets of tuples, lists, type applications, and argument lists, $\vec{e}$ stands for comma-separated elements. The sequence $x \cdot \vec{x}$ has head $x$ and tail $\vec{x}$. Comma-separated elements inside set braces are pairwise distinct. Two sequences of equal length paired by a colon, $\vec{x}{:}\,\vec{\psi}$, denote the sequence of pairs $x_1{:}\,\psi_1, \ldots, x_n{:}\,\psi_n$, and judgements applied to sequences of equal length hold pointwise. Metavariables $x$, $y$, $z$, $f$, $c$ and $\alpha$ range over *variables*. Idiomatically, $f$ denotes a function, $c$ a class, and $\alpha$ a type parameter. *Names*

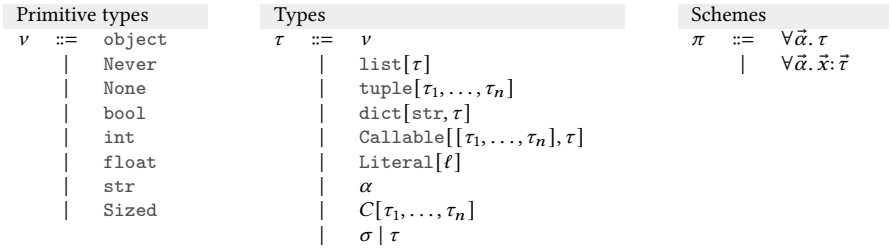



Fig. 2. Types and schemes

$q$ are non-empty sequences of variables separated by dots (Figure 3). A variable on its own is a *simple name* and a name with at least one dot is *qualified*. Names are *fully-qualified* when read from the root of the module hierarchy; modules (§3.3) and classes are identified by fully-qualified names.

Figures 2 and 3 define the syntax of PurePy. The grammar is an abstract syntax, so the metavariables $n$ and $w$, for example, range over numbers and strings rather than over numerals and string literals; the translation from Python's concrete grammar lies outside the specification, which assumes that each concrete form is translated to the form of the same shape in Figure 3. Calls of functions and of classes share the form $q(a)$, and are distinguished by type checking (§4). The grammar admits some programs which Python rejects as syntax errors, mostly duplicated names such as two parameters of the same name; Figure 2.3–4 lists these.

### 3.1 Types and type schemes

Figure 2 gives the syntax of types $\tau$, familiar from the Python typing specification [Python Typing Council 2025], and of schemes $\pi$. *Type expressions* $\psi$, omitted from the figure, are expressions which denote types; they have the form of types, except that where types have *instantiated classes* $C[\vec{\tau}]$ and type variables $\alpha$, type expressions have qualified names applied to type arguments $q[\vec{\psi}]$ naming a class, type alias, or type parameter, which are resolved to types during type checking (§4.1). Parameters, return types, and dataclass fields are annotated with type expressions, as are variable declarations $x\colon \psi$, with or without an initial assignment. Later assignments to a declared variable are not annotated. A *scheme* $\pi$ binds type parameters $\vec{\alpha}$, and is either a *type scheme*, binding type parameters in a type, for a function, or a *field scheme*, binding type parameters in a sequence of typed fields, for a class. A scheme without any type parameters is identified with its body.

### 3.2 Statements and expressions

Figure 3 gives the syntax of statements $s$, which are sequenced by juxtaposition, $s\,s'$. Function bodies, branches of conditionals, and bodies of cases are statements. *Type definitions* $\chi$ are either class definitions or `type` statements; the latter declare *type aliases*. *Top-level statements* $t$ are either statements or mutual type regions, also sequenced by juxtaposition; module bodies consist of imports followed by top-level statements, so type definitions occur only at the top level of a module. Function definitions, class definitions and `type` statements bind distinct *type parameters* $\vec{\alpha}$, written in brackets after the name as in Python 3.12. *Mutual def regions* $d_1 \ldots d_{n+1}$ are maximal contiguous blocks of `def` statements, and *mutual type regions* $\chi_1 \ldots \chi_{n+1}$ maximal contiguous blocks of type definitions; the well-formedness rules treat each as one mutually recursive binding, which replaces Python's late binding (§2.2.3).

Figure 3 also gives the syntax of expressions $e$. Metavariables $\odot$ and $\ominus$ range over binary and unary operators, whose precedence and associativity are given in Figure 2.3–5; `and` and `or` have dedicated forms, since they do not always evaluate both operands. *Argument lists* $a$ occur in calls

**Statements**

$$
\begin{array}{lcl}
s & ::= & s\ s' \\
& | & x\colon \psi \\
& | & x = e \\
& | & x\colon \psi = e \\
& | & e \\
& | & \texttt{print}(e) \\
& | & \texttt{assert}\ e, e' \\
& | & \texttt{return}\ e \\
& | & \texttt{if}\ e\texttt{:}\ s\ \texttt{elif}\ e_1\texttt{:}\ s_1\ \ldots\ \texttt{elif}\ e_n\texttt{:}\ s_n\ \texttt{else:}\ s' \\
& | & \texttt{match}\ e\colon b_1\ \ldots\ b_{n+1} \\
& | & d_1\ \ldots\ d_{n+1}
\end{array}
$$

**Top-level statements**

$$
\begin{array}{lcl}
t & ::= & s \\
& | & t\ t' \\
& | & \chi_1\ \cdots\ \chi_{n+1}
\end{array}
$$

**Type definitions**

$$
\begin{array}{lcl}
\chi & ::= & \texttt{@dataclass class}\ c[\vec{\alpha}](q[\vec{\psi}])\texttt{:}\ \phi \\
& | & \texttt{type}\ x[\vec{\alpha}] = \psi
\end{array}
$$

**Imports**

$$
\begin{array}{lcl}
\iota & ::= & \texttt{import}\ q \\
& | & \texttt{from}\ q\ \texttt{import}\ x_1, \ldots, x_{n+1}
\end{array}
$$

**Cases**

$$
b \quad ::= \quad \texttt{case}\ p\colon s
$$

**Field lists**

$$
\phi \quad ::= \quad x_1\colon \psi_1\ \ldots\ x_{n+1}\colon \psi_{n+1}
$$

**Names**

$$
q \quad ::= \quad x_1.\cdots.x_{n+1}
$$

**Function definitions**

$$
d \quad ::= \quad \texttt{def}\ f[\vec{\alpha}](\vec{x}\colon\vec{\psi})\ \texttt{->}\ \psi\colon s
$$

**Expressions**

$$
\begin{array}{lcl}
e & ::= & x \\
& | & \ell \\
& | & e(a) \\
& | & \{e_1\colon e'_1, \ldots, e_n\colon e'_n\} \\
& | & e.x \\
& | & e[e'] \\
& | & e \odot e' \\
& | & \ominus e \\
& | & e\ \texttt{and}\ e' \mid e\ \texttt{or}\ e' \\
& | & e_1\ \texttt{if}\ e\ \texttt{else}\ e_2 \\
& | & [e_1, \ldots, e_n] \\
& | & e_1, \ldots, e_n \\
& | & [\,e\ g_1\ \ldots\ g_n] \\
& | & \{e\colon e'\ g_1\ \ldots\ g_n\} \\
& | & \texttt{lambda}\ \vec{x}\colon e
\end{array}
$$

**Comprehension qualifiers**

$$
\begin{array}{lcl}
g & ::= & \texttt{if}\ e \\
& | & \texttt{for}\ x\ \texttt{in}\ e
\end{array}
$$

**Literals**

$$
\begin{array}{lcl}
\ell & ::= & n \\
& | & w \\
& | & \texttt{True} \mid \texttt{False} \\
& | & \texttt{None}
\end{array}
$$

**Argument lists**

$$
a \quad ::= \quad \vec{u}, \vec{x}{=}\vec{u}'
$$

**Patterns**

$$
\begin{array}{lcl}
p & ::= & \ell \\
& | & x \\
& | & _ \\
& | & [p_1, \ldots, p_n] \\
& | & p_1, \ldots, p_n \\
& | & q(a) \\
& | & \{\vec{w}\colon\vec{p}\} \\
& | & p\ \texttt{as}\ x
\end{array}
$$

Fig. 3. Statements and expressions

and in constructor patterns, and consist of positional arguments $\vec{u}$ followed by keyword arguments $\vec{x}{=}\vec{u}'$, where each argument $u$ is an expression in a call and a pattern in a constructor pattern.

### 3.3 Modules and programs

A PurePy *program* is a finite map $\mathcal{M}$ from *fully-qualified module names* (Figure 3) to *module bodies*, with a distinguished *main module* named `__main__`. A module body is a sequence of imports $\vec{\iota}$, each a *plain import* `import` $q$ or a *from-import* `from` $q$ `import` $\vec{x}$, followed by a top-level statement $t$; since imports are not statements, they can appear neither within a statement nor after one. After its imports, the body begins with the declaration `__name__: str` = name($q$), where name($q$) is the string literal naming $q$, binding the name of the module when the module loads (§4.5).

A second map $\mathcal{P}$ gives the signatures of the predefined modules `builtins`, `dataclasses`, `sys` and `typing`, which have no body and provide selected members of the Python modules of the same names. The two domains are disjoint, and their union is prefix-closed: the parent of any module is itself a module. A typical encoding of $\mathcal{M}$ would be a source tree where the `.py` files and directories

are the modules, the body of a directory module being its `__init__.py` file. The maps $\mathcal{M}$ and $\mathcal{P}$ are fixed and ambient in all definitions that follow.

## 4 Well-formedness

Well-formedness is the static part of PurePy, defined via judgements on programs. Such judgements give each expression a type, each statement a static outcome (which records the variables the statement assigns or that it definitely returns) and each module a signature. The judgements are mechanised in Isabelle/HOL [Nipkow et al. 2002] as functions which either compute a type, static outcome, or signature, or fail with a reason. Figures and results with counterparts in the mechanisation name the Isabelle definition/theorem in the caption or heading, typeset in code font. For example, the reference type checker is generated from `check_program` by the Isabelle code generator [Haftmann and Nipkow 2010] through a backend for Python developed for this work.

### 4.1 Contexts and types

Well-formedness is checked relative to a *context* $\Gamma$, a finite map from variables to *context entries* recording what each name in scope denotes, and a *class table* $\Sigma$, following Featherweight Java [Igarashi et al. 2001], which maps each class name to its definition, together with the context in which the definition appears. Context entries bind names to types or type schemes, classes, type parameters, type aliases, modules, or predefined names such as `int`. Variable entries also record *definite assignment* state [Gosling et al. 2025, §16]. Variables are *definitely assigned* when every path to the current point has assigned them, and may then be used; *definitely unassigned* when declared but assigned on no path, and may then be assigned but not used; and *possibly unassigned* when assigned on some paths but not all, and may then be neither used nor assigned. Variables declared later in the scope have entry Unbound, and their uses are rejected statically, where Python fails at runtime with `UnboundLocalError`. Contexts compose sequentially by *override* $\Gamma \otimes \Delta$, in which the entries of $\Delta$ replace those of $\Gamma$. Across branches of a conditional, contexts compose by *merge* $\Gamma \oplus_{\Sigma} \Delta$ indexed by class table $\Sigma$, where variables assigned in only one branch become possibly unassigned, and pattern variables bound at different types in two `case`s are bound at the join of those types. Figure 2.2–1 and Definitions 2.4 and 2.5 define contexts and their combining operations.

Programs mention types only through the type expressions in annotations (§3.1), so the typing rules must resolve these to types by looking up the names in $\Gamma$. Class names applied to type arguments resolve to instantiated classes, alias names to the bodies of the aliases at their type arguments, and type parameters to type variables (Figure 3.1–2).

*Subtyping.* Subtyping (Figure 3.1–5) is structural and follows (a subset of) mypy (§2.1). Instantiated classes are below their base classes, with type arguments invariant, as are lists and dictionaries. Tuples are covariant, and callables are contravariant in parameter types and covariant in result type. A literal type is below its base type, and `int` is below `float`, as PEP 484 requires, whereas `bool` is not below `int`, although in Python `bool` is a subclass of `int`. A union $\sigma \mid \tau$ is a least upper bound of its *disjuncts* $\sigma$ and $\tau$; `Never` is the least type and `object` the greatest. Subtyping is a preorder (Lemma 3.1), with distinct types such as $\sigma \mid \tau$ and $\tau \mid \sigma$ *equivalent*, $\sigma \equiv \tau$, when each is a subtype of the other. The notations $<:$ and $\equiv$ follow Pierce [2002].

Two types are *comparable* when either is a subtype of the other. Two comparable types *join* to the larger type and two incomparable types to their union. Two comparable types *meet* to the smaller type; a meet of incomparable types distributes over a union, takes tuples componentwise and callables with parameters joined and results combined using meet, and is `Never` otherwise, since no value belongs to both types. So joining a type with itself gives that type, joining a class with its base class gives the base class, and joining `int` with `str` gives `int | str`. A sequence of types is joined by folding binary join over the sequence, with `Never` as the unit. Types ordered by subtyping in

$\Sigma$, up to equivalence, form a bounded distributive lattice, with join as least upper bound, meet as greatest lower bound, `object` as the greatest type and `Never` as the least (`join_upper`).

### 4.2 Expressions

Expression typing is bidirectional, with two judgements. *Synthesis* $\Sigma; \Gamma \vdash e \triangleright \tau$ calculates a type for an expression (Figures 3.2–1, 3.2–2 and 3.2–3; `synth_expr`), and *checking* $\Sigma; \Gamma \vdash e \triangleleft \tau$ tests an expression against a type supplied by its surroundings (Figure 3.2–5; `check_expr`). As usual in bidirectional systems [Davies and Pfenning 2000; Dunfield and Krishnaswami 2021], introduction forms check and elimination forms synthesise. Python has no annotation form $e : \tau$ to let a checking form occur where synthesis is required, so PurePy gives each introduction form a synthesis rule wherever its parts determine the type, an approach also taken by Swift, Kotlin and Scala [Apple Inc. 2025; JetBrains 2025; Odersky et al. 2025]. Lambdas and empty lists have only checking rules, since lambda parameters cannot be annotated and empty lists do not determine the element type; a lambda takes its parameter types from the expected `Callable` type and checks its body against the return type (lambda). Literals other than floats synthesise to *singleton types*, e.g., $\vdash$ `3` $\triangleright$ `Literal[3]`, which are subtypes of their corresponding base type, e.g., `Literal[3]` <: `int`. Lists synthesise the join of the base types of their elements, since `list` is invariant and a literal element type would rule out every other element. Expressions that synthesise a type $\sigma$ check against any supertype of $\sigma$ by subsumption, which applies only when no other rule concludes $\Sigma; \Gamma \vdash e \triangleleft \tau$, so both judgements are syntax-directed.

Operators are overloaded, as specified in Figure 3.2–6, and the binop and unop rules synthesise the return type of the most specific overload at the operand types. Union types need dedicated rules at elimination forms, operators included: with `x` of type `int | str`, no overloading of `*` applies at `x * 2`, which is instead typed at `int` and at `str` and synthesises `int | str`. Calls in which the callee synthesises a union are typed at each disjunct, with the callee replaced by a fresh variable bound at that disjunct, synthesising the join of the two results (`synth_call`):

call-union

$$\frac{\Sigma; \Gamma \vdash e \triangleright \sigma \mid \sigma' \qquad \Sigma; \Gamma \otimes \{y{:}\,\sigma\} \vdash y(\vec{e}) \triangleright \tau \qquad \Sigma; \Gamma \otimes \{y{:}\,\sigma'\} \vdash y(\vec{e}) \triangleright \tau' \qquad y \notin \mathrm{fv}(\vec{e})}{\Sigma; \Gamma \vdash e(\vec{e}) \triangleright \tau \vee_\Sigma \tau'}$$

Subscripts and attribute references have analogous rules. A single rule eliminating a union at any subterm in evaluation position [Dunfield and Pfenning 2004] accepts more programs, such as a function of intersection type applied to an argument of union type, but the syntax of a term does not determine the subterm at which the rule applies, meaning that type checking requires search [Dunfield 2011]. PurePy lacks intersection types, and the rules above cover overloaded operators, their closest analogue, so union elimination stated for each elimination form keeps both judgements syntax-directed without rejecting any program which the single rule would accept.

Generic functions, bound in the context at type schemes rather than types, are typed only at calls or against expected types. A call instantiates the type parameters by matching each parameter type against the type synthesised by the corresponding argument, with a type parameter found at several positions taking the join of those types, and then checks every argument against its instantiated parameter type:

```
def keep[T](p: Callable[[T], bool], xs: list[T]) -> list[T]:
    return [x for x in xs if p(x)]
ys: list[int] = keep(lambda x: x > 0, [1, -2, 3])
```

Here the list argument fixes `T` at `int`, and the lambda is then checked against `Callable[[int], bool]`. All type arguments are determined before any argument is checked, so a call is rejected when every argument that could determine a type parameter is a lambda, or another expression that

does not synthesise a type. Such calls are accepted by mypy, which types each lambda once the other type arguments are known; the Python typing specification leaves the inference of type arguments unspecified, so neither behaviour is prescribed. Generic functions check against a type $\tau$ by instantiating their type parameters from $\tau$ in the same way, provided the instantiated type is a subtype of $\tau$; thus generic functions can be passed where `Callable` types are expected.

### 4.3 Statements

A well-typed statement has a *static outcome* $R$, either Returns, when the statement definitely returns or definitely reaches `assert False`, or Assigns($\Delta$), where $\Delta$ is a context with an entry for each variable declared or assigned by the statement, recording its assignment state afterwards. Override and merge lift to static outcomes (Definition 3.4), with Returns as zero for $\otimes$ and unit for $\oplus_\Sigma$. The statement judgement $\Sigma; \Gamma; \zeta \vdash s : R$ says that under $\Gamma$ and optional return type $\zeta$, statement $s$ has static outcome $R$ (Figure 3.3–1; `check_stmt`); every rule passes $\zeta$ down unchanged. A sequence $s\,s'$ types $s$, then types $s'$ under $\Gamma$ overridden by the context assigned by $s$. Conditionals combine their branches with $\oplus_\Sigma$, merging in Assigns($\varnothing$) when there is no `else` branch.

The defs rule (Figures 3.3–2 and 3.3–3) checks each definition of a mutual def region against the type scheme synthesised from its annotations, in a context in which each function name is definitely assigned (§2.2.3). The body is checked with the declared return type, and must have static outcome Returns unless that type is equivalent to `None`, so a function that declares a result definitely returns one. A return type such as `int | None` does not exempt the body from returning, nor permit a bare `return`, for compatibility with mypy (§2.1).

A `match` scrutinee synthesises a type $\sigma$. *Case checking* (§4.4) types the cases $\vec{b}$ against $\sigma$, merges their outcomes into $S$ and gives a final *remaining type*, an over-approximation of the values of $\sigma$ left unmatched by the cases. The match rule applies when the final remaining type is `Never`, meaning the match is exhaustive; the match-partial rule applies otherwise and also merges in Assigns($\varnothing$), the outcome of the path on which the scrutinee is not matched by any of the cases:

$$\text{match}\quad \frac{\Sigma; \Gamma \vdash e \triangleright \sigma \qquad \Sigma; \Gamma; \zeta \vdash \sigma, \mathrm{shapes}_\Sigma(\sigma, \varnothing), \vec{b} : S, \texttt{Never}, \varnothing}{\Sigma; \Gamma; \zeta \vdash \texttt{match}\ e\texttt{:}\ \vec{b} : S}$$

$$\text{match-partial}\quad \frac{\Sigma; \Gamma \vdash e \triangleright \sigma \qquad \Sigma; \Gamma; \zeta \vdash \sigma, \mathrm{shapes}_\Sigma(\sigma, \varnothing), \vec{b} : S, \tau, L \qquad \tau \neq \texttt{Never}}{\Sigma; \Gamma; \zeta \vdash \texttt{match}\ e\texttt{:}\ \vec{b} : S \oplus_\Sigma \mathrm{Assigns}(\varnothing)}$$

### 4.4 Pattern matching

For a `match` with scrutinee type $\sigma$, case checking $\Sigma; \Gamma; \zeta \vdash \tau, L, \vec{b} : S, \tau', L'$ (Figure 3.3–4; `match_cases`) threads through the cases a remaining type $\tau$, starting from $\sigma$, and a *residual* $L$, an ordered set of *shapes* (Figure 3.4–3a) each denoting a set of values of a single type, starting from the shapes of $\sigma$; both over-approximate the values of $\sigma$ left unmatched by earlier cases. The pattern of each case is checked against the remaining type, binding any pattern variables, and matched against the residual. A case is *reachable* if its pattern matches at least one shape of the residual, and a match with an unreachable case is ill-formed. For a reachable case, the body is checked under the pattern bindings and its outcome merged into $S$, and the shapes left after matching and the type remaining if the pattern fails become the residual and remaining type for the case that follows (match-case, match-cases). The residual corresponds to the term descriptions of Sestoft [1996], and reachability to the usefulness of Maranget [2007]; seeding the residual from the scrutinee type has a precedent in the uncovered sets of Karachalias et al. [2015] for GADT matches.

The remaining type determines the types of pattern variables and, via the match rule, whether a match is *exhaustive*, meaning that its cases match every value of $\sigma$. The Python typing specification

leaves narrowing largely unspecified.[4] Thus the remaining type emulates the narrowing of match scrutinees observed in mypy (§2.1), which subtracts a class from the scrutinee type only when a constructor pattern matches every value of the class, and narrows a tuple type only when exactly one sub-pattern of a tuple pattern can fail to match a value of the corresponding component type.

The residual and the remaining type can differ. In the following program, the two cases between them match every value of `C`:

```
@dataclass
class C:
    p: tuple[int, int] | None

# error: f does not always return
def f(c: C) -> int:
    match c:
        case C(None):
            return 0
        case C((a, b)):
            return a + b
```

The residual starts as $\{\texttt{C} \setminus \varnothing\}$, where the *rest shape* $\tau \setminus \varnothing$ denotes every value of $\tau$. The first case splits this shape into a constructor shape for the values of `C` with a tuple field and one for the values with a `None` field, and matches the shape with the `None` field; the second case matches the shape with the tuple field, leaving the empty residual, so a third case would be unreachable. Neither case matches every value of `C`, however, so the remaining type is still `C`, match-partial applies, and the function is rejected as not always returning, as under mypy. If the second case were `C(q)` instead, the remaining type after it would be `Never`, but `q` would be bound at `tuple[int, int] | None` rather than `tuple[int, int]`, because pattern checking uses the remaining type.

The cases needed to exhaust a match therefore depend on the type of the scrutinee: a union is exhausted by covering each disjunct, a class type by a pattern for the class itself, `bool` by `True` and `False`, a literal type `Literal[`$\ell$`]` by the literal pattern $\ell$ and a dictionary type by the empty dictionary pattern, while `int`, `float`, `str` and lists need a variable or wildcard, since literal patterns match single values and list patterns only match lists of a specific length.

### 4.5 Modules and programs

*Module checking*, the judgement $\Sigma \vdash_{\mathsf{mod}} q : \Gamma \dashv \Sigma'$ (Figure 3.5–3; `check_module`), types the body of module $q$ as a top-level statement and computes its signature $\Gamma$, extending $\Sigma$ to $\Sigma'$ with the classes of the imported modules and of $q$. A program is well-formed (`check_program`) when its main module is well typed. Checking a module first checks the modules it imports, so every module reachable via an import is checked and a module which is never imported is not checked; a program with an import cycle is ill-formed. The body is typed under the context of $\mathcal{P}$(`builtins`), overridden by the imported context and then, as in def-body, by an unbound entry for each name declared in the body or bound by a pattern in it, so that any names bound by the body shadow any builtins of the same name throughout the module (§2.2.3). The names declared in the module, including imported names, must be distinct from one another, from the implicit `__name__` (§3.3) and from any pattern variables occurring in its body. The signature $\Gamma$ comprises the classes, type aliases and variables defined by the module, together with `__name__` and a stub for each submodule.

## 5 Operational semantics

Evaluation is defined via a big-step relation on programs, producing a trace of printed outputs and a *termination kind* $\kappa$ (Figure 4) which records how the run ends. Termination kinds are named mainly after the exceptions raised by the same program under Python in the same situations, with SystemExit(0) recording a run which completes normally. The two termination kinds undef and

[4] https://typing.python.org/en/latest/spec/narrowing.html

**Outcomes**

$$
\begin{array}{lcl}
o & ::= & (\vec{v}, \omega) \\
\omega & ::= & \mathsf{assigns}(\rho) \\
& | & \mathsf{returns}(v) \\
& | & \mathsf{aborts}(\kappa)
\end{array}
$$

**Results**

$$
\begin{array}{lcl}
r & ::= & \mathsf{val}(v) \\
& | & \mathsf{aborts}(\kappa)
\end{array}
$$

**Match results**

$$
\begin{array}{lcl}
m & ::= & \mathsf{match}(\rho) \\
& | & \text{no-match} \\
& | & \kappa
\end{array}
$$

**Termination kinds**

$$
\begin{array}{lcl}
\kappa & ::= & \mathsf{TypeError} \mid \mathsf{IndexError} \mid \mathsf{KeyError} \\
& | & \mathsf{OverflowError} \mid \mathsf{ValueError} \mid \mathsf{ZeroDivisionError} \\
& | & \mathsf{AssertionError} \mid \mathsf{AssertionError}(w) \mid \mathsf{AttributeError} \\
& | & \mathsf{NameError} \mid \mathsf{SystemExit}(n) \mid \mathsf{undef} \mid \mathsf{excluded}
\end{array}
$$

Fig. 4. Outcomes, results, match results and termination kinds (`outcome`, `result`)

excluded are where evaluation departs from Python. A run aborts with excluded in a situation ruled out by the type system, such as a conditional test on a non-Boolean or a list pattern matched against a tuple; by type soundness (§5.2.3) these situations arise only in ill-formed programs.

A run aborts with undef in a situation which the type system cannot rule out (§2.2.5) but which cannot be defined in PurePy, e.g., a comparison between NaNs inside containers or an overflowing ** on floats. Python returns a result in these situations.

## 5.1 Expressions and statements

Values $v$ have the usual forms, plus special forms for predefined members and unbound variables, and for modules, classes and type aliases, which are bound to names by imports and type definitions although programs are not permitted to use them as values (§2.3). Environments $\rho$ are finite maps from variables to values (`val`, `env`). On entry to a scope, every variable declared in the scope is bound to Unbound, hiding any occurrences in enclosing scopes.

*5.1.1 Results and outcomes.* An expression evaluates to a *result* $r$ (Figure 4), either a value $\mathsf{val}(v)$ or an abort $\mathsf{aborts}(\kappa)$ with termination kind $\kappa$. For a statement $s$, outcome $(\vec{v}, \omega)$ represents $s$ printing the values $\vec{v}$, in order, and completing as $\omega$. Completion $\mathsf{assigns}(\rho)$ represents completing normally, with accumulated environment $\rho$; completion $\mathsf{returns}(v)$ represents returning the value $v$; and completion $\mathsf{aborts}(\kappa)$ represents the whole run aborting, with termination kind $\kappa$. Composing statements sequentially composes their outcomes according to the following monoid, analogous to sequential composition of static outcomes:

$$
\begin{aligned}
(\vec{v}, \mathsf{returns}(v)) \otimes o &= (\vec{v}, \mathsf{returns}(v)) \\
(\vec{v}, \mathsf{aborts}(\kappa)) \otimes o &= (\vec{v}, \mathsf{aborts}(\kappa)) \\
(\vec{v}, \mathsf{assigns}(\rho)) \otimes (\vec{v}', \mathsf{assigns}(\rho')) &= (\vec{v} +\!\!+ \vec{v}', \mathsf{assigns}(\rho \otimes \rho')) \\
(\vec{v}, \mathsf{assigns}(\rho)) \otimes (\vec{v}', \omega) &= (\vec{v} +\!\!+ \vec{v}', \omega) \quad \text{(otherwise)}
\end{aligned}
$$

*5.1.2 Evaluation judgements.* Throughout, $\Sigma$ is the class table produced by module checking of the program (§4.5) and, like $\mathcal{M}$ and $\mathcal{P}$, is ambient. The statement evaluation judgement $\rho, s \Rightarrow o$ says that $s$ evaluates to outcome $o$ in $\rho$ (Figures 4.3–2 and 4.3–3; `eval_stmt`), and the expression evaluation judgement $\rho, e \Rightarrow r$ says that $e$ evaluates to result $r$ in $\rho$ (Figures 4.3–5 and 4.3–7; `eval_expr`). Top-level statements $t$ have a separate judgement, $q, \rho, t \Rightarrow o$ (Figure 4.3–4; `eval_top`), where $q$ is the module under evaluation, used to produce qualified names for any type definitions.

*5.1.3 Accumulated bindings.* Assignment produces the same results as under Python (§2.1), but by a different mechanism: it extends the environment with a binding rather than updating the variable, and function bodies delimit these effects, so that as black boxes functions are pure. Print

statements in function bodies are omitted from this version, so the outcome of a body has empty output $\varepsilon$, and the auxiliary function result reads the outcome as the result of the call (`result_of`):

$$\begin{aligned}\mathrm{result}((\varepsilon, \mathrm{returns}(v))) &= \mathrm{val}(v)\\ \mathrm{result}((\varepsilon, \mathrm{assigns}(\rho))) &= \mathrm{val}(\texttt{None})\\ \mathrm{result}((\varepsilon, \mathrm{aborts}(\kappa))) &= \mathrm{aborts}(\kappa)\end{aligned}$$

Comprehensions likewise evaluate the body in a fresh extension of the environment for each element, where Python updates the comprehension variables in place; the two agree because lambdas are not permitted to capture generator variables (§2.2.2).

*5.1.4 Pattern matching.* Matching a pattern against a value has three possible outcomes: a match, which binds the pattern variables; no match; or an abort. A literal pattern matches a value by equality with the literal, so matching aborts when that equality is undefined, and matching also aborts when a pattern is matched against a value ruled out by typing, such as a list pattern against a tuple (§5). A `match` statement then evaluates the body of the first case whose pattern matches the value, completes normally with no bindings if every case fails to match, and aborts if matching aborts. The rule eval-match passes the bindings of the selected case to its body and includes them in the outcome of the statement, so pattern variables remain bound after the match, in keeping with the static treatment of patterns as assignments (§2.2.2):

$$\text{eval-match}\quad \frac{\rho, e \Rightarrow \mathrm{val}(v) \qquad \mathrm{dispatch}(\rho, v, \vec{p}, \vec{s}) = (\rho', s') \qquad \rho \otimes \rho', s' \Rightarrow o}{\rho, \texttt{match } e\texttt{: case } p_1\texttt{: } s_1 \ \ldots\ \texttt{case } p_{n+1}\texttt{: } s_{n+1} \Rightarrow (\varepsilon, \mathrm{assigns}(\rho')) \otimes o}$$

## 5.2 Modules and programs

Module and program evaluation are defined in full in Figure 4.5–1. A program runs each module body once, after its imports have loaded, in *load order*. Load order is fixed by the imports alone, since imports precede statements and so cannot depend on the outcome of a statement. Module evaluation $q, \mathcal{L} \Rightarrow_{\mathrm{mod}} o$ runs the body of module $q$ given the *loaded module environments* $\mathcal{L}$, a finite map from module names to the environments of the modules loaded so far. The body runs in the environment corresponding to the context in which the body was checked.

Evaluation of a sequence of module bodies extends $\mathcal{L}$ with the environment of each module in turn. The environment of a module $q$ consists of a stub for each submodule of $q$, overridden by the environment assigned by the body of $q$. The program rules start from the environments $\mathcal{E}(\mathcal{A})$ of the predefined modules, given command-line arguments $\mathcal{A}$, a sequence of strings beginning with the file name of the main module, as per `sys.argv`.

*5.2.1 Program results and conformance.* A program evaluates to a *program result* $(\vec{v}, \kappa)$, pairing its output $\vec{v}$ with a termination kind $\kappa$ (Figure 4). The output is the concatenation of the values printed by the module bodies, and the termination kind is $\mathrm{SystemExit}(0)$ if every module body completes normally, and otherwise the kind of the first module body to abort, after which nothing more is printed. $\mathrm{SystemExit}(n)$ is a run that exited with status $n$. An implementation *conforms* to the semantics (§2.1) if, for every well-formed program $\mathcal{M}$, arguments $\mathcal{A}$ and program result $(\vec{v}, \kappa)$ with $\mathcal{M}, \mathcal{A} \Rightarrow \vec{v}, \kappa$ and $\kappa \neq \mathrm{undef}$, a run of the implementation on $\mathcal{M}$ with arguments $\mathcal{A}$ writes the values $\vec{v}$ in order, each as by Python's `str` followed by a newline, and reports termination kind $\kappa$. The form in which an implementation reports the termination kind is not prescribed.

An implementation also conforms if it meets this definition with the result of `**` on floats being the nearest float below or above the exact power, not necessarily the nearer of the two. Python implementations conform, since they compute `**` on floats with functions such as `pow` in C and `Math.pow` in Java, which meet this requirement.

*5.2.2 Reference interpreter.* The evaluation judgements are mechanised as inductive relations, which are then shown to be deterministic (`program_det`). The reference interpreter is generated from `run_program`, a recursive function with a fuel argument which, for a well-formed program, returns a result at some fuel if and only if the big-step semantics relates the program to that result (`program_soundness`, `checked_program_completeness`). A front end parses source files into the abstract syntax of the mechanisation using the `ast` module of the Python standard library.

*5.2.3 Soundness.* Typing rules out four of the termination kinds.

Theorem 1 (Type soundness; `type_soundness`). *If* $\vdash \mathcal{M}$ *and* $\mathcal{M}, \mathcal{A} \Rightarrow \vec{v}, \kappa$ *then* $\kappa$ *is not* TypeError, AttributeError, NameError *or* excluded.

The environment typing judgement $\vdash \rho : \Gamma$ says that every value in $\rho$ is well-typed at the corresponding entry in $\Gamma$ (Figure 4.3–1). The proof relies on the predefined environments $\mathcal{E}(\mathcal{A})$ being well-typed by construction and the environment loaded for a module with signature $\Gamma$ being well-typed at $\Gamma$ (`predefined_env`, `loaded_module`). The var rule requires $x$ to be definitely assigned; since no value other than Unbound is well-typed at Unbound or $\mathrm{DU}(\tau)$ (`wf_env_lookup`, `wf_unbound_ty`), well-formed programs never abort with NameError.

## 6 Experimental evaluation

We compare generating PurePy with generating Typed Python, which separates the effect of the further restrictions of PurePy from the effect of types alone, and Typed Python with Python. We ask two questions: how often a model produces a correct PurePy solution, at what cost, and how much of the extra cost is due to the design of PurePy rather than to features not yet in the specification (**RQ1**); and how correct PurePy solutions compare in size and running time (**RQ2**).

### 6.1 Experimental setup

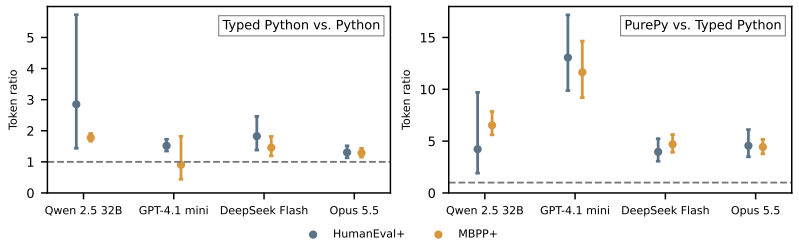
We use HumanEval+ and MBPP+ benchmarks from EvalPlus [Liu et al. 2023], providing problems over small functions, builtin types, and the standard library. 14 MBPP+ problems are excluded as they use unsupported types, e.g., sets, so no PurePy solution can exist. EvalPlus records no outcome for 1 HumanEval+ problem, so is also excluded, leaving 163 HumanEval+ and 364 MBPP+ problems. The models are `Qwen/Qwen2.5-Coder-32B-Instruct` (Qwen 2.5 32B), `gpt-4.1-mini-2025-04-14` (GPT-4.1 mini), `deepseek-flash` (DeepSeek Flash) and `claude-opus-5-5` (Opus 5.5), with reasoning at high effort for DeepSeek Flash and Opus 5.5, and an output limit of 16384 tokens.

A *scenario* is one of: PurePy, checked by our reference implementation; Typed Python, checked by mypy in strict mode; and Python, unchecked. The model is prompted with the problem and the rules of the target language (for PurePy a summary of its restrictions with examples, for Typed Python the mypy strict-mode requirements), and a program extracted from the response. If the static check fails, the first diagnostic is returned to the model and a new attempt made, for at most 3 rounds. The accepted program is run against the official tests, which the model does not see.

The token limit, output format and repair budget are the same in every scenario. A response which ends at the token limit or breaks the output format is a failure, neither checked nor repaired, and a failure of the model API, type checker or test harness invalidates the run, which is repeated. The reference implementation classifies each rejection as: excluded by design, such as a loop; omitted from this version, such as slicing or the builtin `ord`; ill-formed; or syntax error. Programs rejected for using a feature omitted from this version are repaired like any other, since a solution without the feature may exist. The prompt lists the builtins omitted from this version, such as `sorted`, so a model is effectively required to define them if needed. PurePy has no loops, so iteration must be expressed using recursion or a comprehension, and since CPython has no tail-call optimisation,

| | | Python | Typed Python | | PurePy | |
|---|---|---|---|---|---|---|
| Model | Benchmark | Correct | Accepted | Correct | Accepted | Correct |
| Qwen 2.5 32B | HumanEval+ | 85.9% | 98.8% | 82.8% | 54.0% | 30.1% |
| Qwen 2.5 32B | MBPP+ | 76.9% | 98.6% | 76.4% | 56.6% | 31.9% |
| GPT-4.1 mini | HumanEval+ | 92.0% | 100.0% | 87.1% | 58.3% | 35.6% |
| GPT-4.1 mini | MBPP+ | 79.1% | 99.7% | 77.5% | 67.9% | 39.0% |
| DeepSeek Flash | HumanEval+ | 95.7% | 100.0% | 93.9% | 92.0% | 82.8% |
| DeepSeek Flash | MBPP+ | 84.9% | 99.5% | 83.0% | 90.9% | 70.9% |
| Opus 5.5 | HumanEval+ | 95.1% | 100.0% | 93.3% | 98.8% | 89.0% |
| Opus 5.5 | MBPP+ | 89.8% | 100.0% | 87.9% | 97.8% | 75.5% |

(a) Problems with an accepted program and with a correct solution (%)




(b) Output-token ratios

Fig. 5. Acceptance and cost of generating solutions (RQ1)

recursion over a large input can exceed its recursion limit. Such failures count against the current design and are reported as a category of their own (§6.2).

## 6.2 Correctness and cost of PurePy solutions (RQ1)

A program is *accepted* if the type checker accepts it within the repair budget; every Python program is accepted. A *correct solution* is an accepted program which passes the official tests. Cost is measured in output tokens, including the tokens spent on reasoning, summed over the first attempt and the repairs up to acceptance or exhaustion of the budget. Token counts are only compared between scenarios for individual models, since models differ in verbosity and tokenisation.

*Correctness.* Figure 5a shows the percentage of problems with an accepted program and with a correct solution. The models fall into two groups. For the more capable models, DeepSeek Flash and Opus 5.5, at least 91% of the PurePy programs are accepted, and 71 to 89% of problems have a correct PurePy solution, at most 12 points fewer than for Typed Python. For the less capable models, Qwen 2.5 32B and GPT-4.1 mini, 54 to 68% of the PurePy programs are accepted, and 30 to 39% of problems have a correct PurePy solution, against 76 to 87% for Typed Python.

Of the PurePy failures, 49% are accepted programs which fail the tests, 45% are programs still rejected after 3 repairs, 96% of them from the less capable models, and 6% are responses which reach the token limit or break the output format and so are not checked, 72% of them from DeepSeek Flash. Of the 456 programs which fail the tests, 125 exceed the recursion limit of CPython, and 57 raise `IndexError`, 36 of them because the program gives a tuple type of fixed length, such as `tuple[int, int, int]`, to a parameter which takes tuples of any length; this version of the specification does not support variable-length tuple types.

*Cost.* Figure 5b shows the tokens spent on Typed Python relative to Python and on PurePy relative to Typed Python. Typed Python takes 0.9 to 2.9 times the tokens of Python. PurePy uses

| PurePy against typed Python | Share (%) |
|---|---|
| First-attempt difference | 34.2 |
| Repairs: excluded by design | 9.3 |
| Repairs: excluded from version | 49.7 |
| Other-repair difference | 6.8 |
| Total | 100.0 |

Fig. 6. Sources of extra output tokens, PurePy vs. Typed Python (%, RQ1)

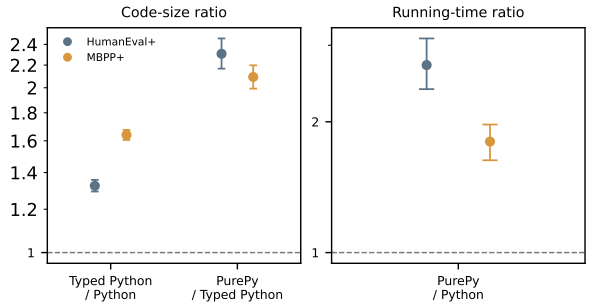



Fig. 7. Size and running time of correct solutions (RQ2), averaged over models; one dot per benchmark, with 95% confidence interval

considerably more tokens than Typed Python, but part of the difference is spent on problems with no accepted program. On the problems with an accepted PurePy program from every model, which are likely to be the easier problems, PurePy takes 3.2 for Qwen 2.5 32B, 5.6 for GPT-4.1 mini, 3.5 for DeepSeek Flash and 2.3 for Opus 5.5 times the tokens of Typed Python on HumanEval+, and 3.8 for Qwen 2.5 32B, 5.0 for GPT-4.1 mini, 2.4 for DeepSeek Flash and 2.7 for Opus 5.5 on MBPP+.

*Extra cost by kind of rejection.* Figure 6 attributes each repair, and its tokens, to the rejection which prompted it, after scaling the tokens of each model by its Typed Python total. Repairs of features excluded from this version, most often `typing.List` (494), slicing (493) and annotation that is not a type expression (253), are 50% of the extra cost. Longer first attempts, which include any definitions written for missing builtins such as `sorted`, are 34%. Repairs of features excluded by design, most often while loops (63), item assignment (48) and annotation that is not a type expression (45), are 9%, and repairs of ill-formed programs and syntax errors are the remaining 7%.

*Answer to RQ1.* The most capable models produce correct PurePy solutions on most problems, at a few times the tokens of Typed Python, whereas the less capable models fall well short. The design of PurePy accounts for 9% of the extra cost. Most of it is due to features missing from this version, and so is expected to fall as the specification grows.

### 6.3 Size and running time of correct solutions (RQ2)

The left panel of Figure 7 compares the sizes of correct solutions, in Python lexical tokens, over the problems with a correct solution in both languages. Typed Python solutions are 1.3 and 1.6 times the size of Python solutions on HumanEval+ and MBPP+, and PurePy solutions are 2.3 and 2.1 times the size of Typed Python solutions. The growth is from helper functions. Over the 1133 model-problem pairs correct in all three languages, a PurePy solution defines 2.5 functions on average and a Typed Python solution 1.1, mostly missing builtins and recursion in place of loops.

The second panel compares the running times of correct PurePy and Python solutions on the official tests, measured on an Apple M4 Pro under CPython 3.12, as the median of 5 repetitions of each program. PurePy solutions take 2.70 times as long as Python solutions on HumanEval+ (95% confidence interval 2.38 to 3.11) and 1.80 times as long on MBPP+ (1.63 to 1.97). The slowdown is concentrated in a few programs. The median PurePy solution is 1.6 times slower than the Python solution; the slowest tenth account for 91% of the extra running time. On geometric average, solutions with a directly recursive function are 2.9 times slower than the Python solutions, and those without 1.2 times slower. The slowest solutions also use less efficient algorithms, since comprehensions have no equivalent of `break` and builtins such as `sorted` are missing.

*Answer to RQ2.* Correct PurePy solutions are about twice the size of Typed Python solutions, mostly because of helper functions, and two to three times slower than Python solutions. Both costs have largely the same sources, the definitions written for missing builtins and recursion in place of loops. Adding the missing builtins is therefore expected to reduce both.

## 7 Related work

### 7.1 PurePy-based projects

Two research languages for scientific computing, developed in parallel by the authors, are built on PurePy: Cosi and Fluid. By being PurePy-compliant, a new user can take any PurePy program and, in principle, run it through Python or either of these languages and get the same result, providing a starting point for them learning the additional features that the respective languages provide, lowering barrier to entry.

*7.1.1 Cosi.* Cosi is a statically typed research language for scientific and numerical computing, with a Python-like syntax. Its distinctive feature is a system of extensible *graded numerical types*. Types are indexed and carry structure, so domain concepts such as dimensionality and units-of-measure (akin to the scheme of [McBride and Nordvall-Forsberg 2021]) and process/object identity are expressible as types and enforced by the type checker. For example, the program on the left is a valid PurePy program accepted by Cosi. On the right, is a Cosi program that goes outside of the PurePy subset, augmenting the types with additional information about the scientific intent:

```
def vel(x : float,
        t : float)
       -> float:
  return (x / t)
```

```
def vel[Obj](x, t) -> Float[Dimension[L / T] & Basis[Obj]]:
  x : Float[Dimension[L] & Basis[Obj]]
  t : Float[Dimension[T] & Basis[Obj]]
  return (x / t)
```

*7.1.2 Fluid.* Fluid is a functional language for transparent research outputs, built on PurePy, with a semantics extended to track the provenance of outputs such as figures and summaries. Fluid also extends the syntax in two ways. *Partial application* allows a function to be applied to fewer arguments than it declares, returning a function of the remaining arguments:

```
from typing import Callable
def add(x: int, y: int) -> int:
    return x + y
inc: Callable[[int], int] = add(1)
inc(5)     # 6
add(1)(2)  # 3
```

For `add(1)`, Python raises a `TypeError` and PurePy rejects it statically. In Fluid, a callee `Callable`$[[\sigma_1, \ldots, \sigma_n], \tau$ with $k < n$ arguments has a function type for the remaining parameters:

$$\text{call-partial}\quad \frac{\Sigma;\Gamma \vdash e, e_1, \ldots, e_k \triangleright \sigma_1, \ldots, \sigma_n, \tau \qquad k < n \qquad \Sigma;\Gamma \vdash e_i \triangleleft \sigma_i \quad (1 \le i \le k)}{\Sigma;\Gamma \vdash e(e_1, \ldots, e_k) \triangleright \texttt{Callable}[[\sigma_{k+1}, \ldots, \sigma_n], \tau]}$$

A call with more than $n$ arguments applies the result to the remaining arguments.

A *piecewise function definition* consists of contiguous clauses whose parameters are patterns; the clauses must agree in arity and annotations, and desugar to a single definition which matches the patterned parameters in a `match` statement, as on the right:

```
def f(k: int, [x]: list[int]) -> int:
    return k * x
def f(k: int, xs: list[int]) -> int:
    return 0
```

```
def f(k: int, $2: list[int]) -> int:
    match $2:
        case [x]: return k * x
        case xs:  return 0
```

### 7.2 Other Python-based language projects

*7.2.1 Language subsets.* Various Python subsets have been defined for specific use cases. For example, ChocoPy [Padhye et al. 2019] is a statically typed subset used in compiler-construction courses. RPython [Ancona et al. 2007] is a statically typed subset compiled to common runtimes by the PyPy project [Rigo and Pedroni 2006]. The tool dafny-of-python [Cheema 2021] verifies a typed subset by translation to the Dafny verifier [Leino 2010]. Each of these subsets is defined from scratch; PurePy offers a formally defined core on which future subsets of this kind can be built.

*7.2.2 Functional dialects.* Sunderraman [2025] defines a functional subset of Python for introductory courses which omits statements such as loops and conditionals; PurePy encloses more of the language by supporting conditional statements within a functional discipline. Related efforts sit outside the subset discipline. Coconut[5] adds pattern matching, partial application and pipeline operators to Python and compiles to it. Bend [BendLang 2026] is a pure, linear language with Python-like syntax and dependent types, in which the compiler checks proofs of stated laws so that generated code can be trusted without being read. Within Python itself, JAX [Bradbury et al. 2018], the leading framework for numerical computation, applies transformations such as differentiation and vectorisation to observationally pure Python functions. Functional pipelines have likewise been proposed for scientific computation [Zhang et al. 2024]. A recent proposal would add deeply immutable objects to the language itself [Johnson et al. 2025].

*7.2.3 Typed dialects.* Static Python [Lu et al. 2023] is a gradually typed Python dialect, enforcing type annotations at runtime. Codon [Shajii et al. 2023] and Mojo[6] are statically typed Python-like languages compiled ahead-of-time for performance. Mojo adds ownership and aims to become a superset of Python. PurePy mandates typing, requiring annotations and excluding `Any`; as a subset, a PurePy program is well-typed Python and runs unchanged on any Python implementation.

### 7.3 Formal semantics of Python

Several works formalise Python itself: an embedding [Guth 2013] in the $\mathbb{K}$ framework [Roșu and Șerbănuță 2010], a structural operational semantics [Köhl 2021], a reduction of much of the language to a core calculus [Politz et al. 2013], an executable semantics aimed at optimising compilers [Melançon et al. 2023], and a skeletal semantics of a fragment of Python [Andrieux and Schmitt 2024]. These formalisations describe Python behaviour, or of part of it, *as is*. PurePy instead prescribes a subset, excluding behaviour that is hard to reason about and defining a functional semantics for that subset, shown to be operationally equivalent to Python's imperative semantics. PurePy, like Clight [Blazy and Leroy 2009] and CakeML [Kumar et al. 2014], is a subset meant to be programmed in, and the reference implementation is generated from the mechanised semantics.

## 8 Conclusions and future work

PurePy is already expressive enough to serve as the common core of two of our own research languages for scientific computing (§7.1), and our experimental study suggests that capable models produce correct PurePy solutions on most problems (§6). Our study also suggests that performance is harder to obtain as Python implementations are not optimised for purity. Recursion in place of loops is slow under CPython and exceeds its recursion limit on large inputs, so correct PurePy solutions can be two or three times slower than Python solutions. Ruling out loops and in-place update entirely is therefore likely to be unrealistic for a more performant future version of PurePy.

We therefore anticipate extending the PurePy dynamic semantics and type system so that selected functions may use local mutation, provided they can be verified statically to be observationally pure.

[5] https://coconut-lang.org [6] https://www.modular.com/mojo

This would allow PurePy libraries to implement key operations efficiently and PurePy programs to take advantage of platforms such as JAX [Bradbury et al. 2018], which are designed around pure functions in this sense.

We also plan to extend the scope of the specification substantially, so that the residual excluded features are mostly excluded by design. Scientific computing in Python rests on libraries such as NumPy and Pandas; both have substantial mutation-free subsets, which we have begun to formalise as PurePy libraries. Support for Python 3.14 is planned pending adoption by GraalPy, which remains on 3.12. Other priority features include exceptions and general-purpose classes, which provide the standard extension points of Python.

### AI Use

The AI coding agent Claude Code was used to draft and maintain technical LaTeX definitions, the conformance suite, the Isabelle/HOL mechanisation and the evaluation pipeline. All AI-generated content was subject to multiple rounds of human auditing and review before being incorporated.

## Data-Availability Statement

The artifact will be submitted for artifact evaluation and archived with a DOI. It comprises:

- PurePy specification, version 0.27.9
- Isabelle/HOL mechanisation of the static and dynamic semantics, release 0.27.9
- reference interpreter and type checker generated from the mechanisation
- conformance suite
- evaluation pipeline and result files of §6
- interactive playground

The specification is published at https://pure-py.github.io/pure-py-spec/v0.27.9/PurePy-spec.pdf, the mechanisation as release v0.27.9 of https://github.com/pure-py/isabelle-purepy, with the reference interpreter and type checker as a Python wheel attached to the release, and the conformance suite as a Python wheel attached to release v0.27.9 of https://github.com/pure-py/pure-py-spec; the playground is at https://pure-py.github.io.

## References


Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: A Step towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*. ACM, Montreal Quebec Canada, 53–64. doi:10.1145/1297081.1297091

Martin Andrieux and Alan Schmitt. 2024. Skeletal Semantics of a Fragment of Python. In *35es Journées Francophones Des Langages Applicatifs (JFLA 2024)*. https://inria.hal.science/hal-04406392

Apple Inc. 2025. *The Swift Programming Language*. Apple. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/

BendLang. 2026. Bend 2: A Fast Language That Blocks AI Mistakes via Proof. GitHub repository. https://github.com/bendlang/bend

Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (Oct. 2009), 263–288. doi:10.1007/s10817-009-9148-3

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: Composable Transformations of Python+NumPy Programs. http://github.com/google/jax

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q. Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 677–708. doi:10.1145/3689735

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3675–3691. doi:10.1109/TSE.2023.3267446

Arsalan Cheema. 2021. Dafny-of-Python: A Tool for Deductive Verification of Python Programs Based on Dafny. GitHub repository. https://github.com/arsalan0c/dafny-of-python

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. 2025. ScienceAgentBench: Toward Rigorous Assessment of Language Agents for Data-Driven Scientific Discovery. In *Proceedings of the 13th International Conference on Learning Representations (ICLR 2025)*.

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. 198–208. doi:10.1145/351240.351259

Deep Forest Sciences. 2026. Physika: A Physical Programming Language. GitHub repository. Retrieved November 12, 2026 from https://github.com/deepforestsci/physika

Jana Dunfield. 2011. Untangling Typechecking of Intersections and Unions. In *Proceedings of the Workshop on Intersection Types and Related Systems (ITRS 2010) (Electronic Proceedings in Theoretical Computer Science, Vol. 45)*. 59–70. doi:10.4204/EPTCS.45.5

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *Comput. Surveys* 54, 5 (2021), 1–38. doi:10.1145/3450952

Jana Dunfield and Frank Pfenning. 2004. Tridirectional Typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. 281–292. doi:10.1145/964001.964025

Titouan Duston, Shuo Xin, Yang Sun, Daoguang Zan, Aoyan Li, Shulin Xin, Kai Shen, Yixiao Chen, Qiming Sun, Ge Zhang, et al. 2025. AInsteinbench: Benchmarking Coding Agents on Scientific Repositories. *arXiv preprint arXiv:2512.21373* (2025). arXiv:2512.21373

GitHub. 2024. Octoverse: AI Leads Python to Top Language as the Number of Global Developers Surges. The GitHub Blog. https://github.blog/news-insights/octoverse/octoverse-2024/

GitHub. 2025. Octoverse: A New Developer Joins GitHub Every Second as AI Leads TypeScript to #1. The GitHub Blog. https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/

James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2025. *The Java Language Specification, Java SE 24 Edition*. Oracle. https://docs.oracle.com/javase/specs/jls/se24/html/index.html

Dwight Guth. 2013. *A Formal Semantics of Python 3.3*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

Florian Haftmann and Tobias Nipkow. 2010. Code Generation via Higher-Order Rewrite Systems. In *Functional and Logic Programming (FLOPS) (Lecture Notes in Computer Science, Vol. 6009)*. Springer, 103–117. doi:10.1007/978-3-642-12251-4_9

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (May 2001), 396–450. doi:10.1145/503502.503505

JetBrains. 2025. *Kotlin Language Documentation*. JetBrains. https://kotlinlang.org/docs/home.html

Matthew Johnson, Matthew Parkinson, Sylvan Clebsch, Fridtjof Stoldt, and Tobias Wrigstad. 2025. PEP 795 – Deep Immutability for Efficient Sharing and Concurrency Safety. https://pep-previews--4468.org.readthedocs.build/pep-0795/

Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. GADTs Meet Their Match: Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ACM, 424–436. doi:10.1145/2784731.2784748

Maximilian A. Köhl. 2021. An Executable Structural Operational Formal Semantics for Python. arXiv:2109.03139 doi:10.48550/arXiv.2109.03139

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. *ACM SIGPLAN Notices* 49, 1 (Jan. 2014), 179–191. doi:10.1145/2578855.2535841

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*. Springer, 348–370. doi:10.1007/978-3-642-17511-4_20

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*. doi:10.48550/arXiv.2305.01210

Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. 2023. Gradual Soundness: Lessons from Static Python. *The Art, Science, and Engineering of Programming* 7, 1 (2023). doi:10.22152/programming-journal.org/2023/7/2

Luc Maranget. 2007. Warnings for Pattern Matching. *Journal of Functional Programming* 17, 3 (2007), 387–421. doi:10.1017/S0956796807006223

Conor McBride and Fredrik Nordvall-Forsberg. 2021. Type Systems for Programs Respecting Dimensions. In *Advanced Mathematical and Computational Tools in Metrology and Testing XII*. World Scientific, 331–345.

Olivier Melançon, Marc Feeley, and Manuel Serrano. 2023. An Executable Semantics for Faster Development of Optimizing Python Compilers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, Cascais Portugal, 15–28. doi:10.1145/3623476.3623529

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. doi:10.1007/3-540-45949-9

Gabrielle O'Brien, Alexis Parker, Nasir Eisty, and Jeffrey Carver. 2025. A Survey of Generative AI Adoption and Perceived Productivity among Scientists Who Program. *arXiv preprint arXiv:2512.19644* (2025). arXiv:2512.19644

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. 2025. *The Scala Language Specification, Version 2.13*. EPFL. https://scala-lang.org/files/archive/spec/2.13/

Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: A Programming Language for Compilers Courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. ACM, Athens Greece, 41–45. doi:10.1145/3358711.3361627

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, Indianapolis Indiana USA, 217–232. doi:10.1145/2509136.2509536

Python Software Foundation. 2023. The Python Language Reference, Release 3.12. https://docs.python.org/3.12/reference/

Python Typing Council. 2025. Specification for the Python Type System. Python typing documentation. https://typing.python.org/en/latest/spec/

Armin Rigo and Samuele Pedroni. 2006. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. ACM, Portland Oregon USA, 944–953. doi:10.1145/1176617.1176753

Grigore Roșu and Traian Florin Șerbănuță. 2010. An Overview of the K Semantic Framework. *The Journal of Logic and Algebraic Programming* 79, 6 (Aug. 2010), 397–434. doi:10.1016/j.jlap.2010.03.012

Peter Sestoft. 1996. ML Pattern Match Compilation and Partial Evaluation. In *Partial Evaluation (Lecture Notes in Computer Science, Vol. 1110)*. Springer, 446–464. doi:10.1007/3-540-61580-6_23

Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jessica Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić. 2023. Codon: A Compiler for High-Performance Pythonic Applications and DSLs. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC 2023)*. 191–202. doi:10.1145/3578360.3580275

Mohammed Latif Siddiq, Simantika Dristi, Joy Saha, and Joanna C. S. Santos. 2024. The Fault in Our Stars: Quality Assessment of Code Generation Benchmarks. In *Proceedings of the 24th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2024)*. IEEE, 201–212. doi:10.1109/SCAM63643.2024.00028

Zachary S. Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. 2024. CORE-Bench: Fostering the Credibility of Published Research Through a Computational Reproducibility Agent Benchmark. doi:10.48550/arXiv.2409.11363

Rajshekhar Sunderraman. 2025. Functional Python Programming in Introductory Computer Science Courses. arXiv:2512.03492 doi:10.48550/arXiv.2512.03492

Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. 2024. SciCode: A Research Coding Benchmark Curated by Scientists. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Datasets and Benchmarks Track*. 30624–30650. doi:10.52202/079017-0963

Lukas Twist, Mark Harman, Don Syme, Joost Noppen, Helen Yannakoudakis, Detlef Nauck, and Jie M. Zhang. 2026. A Study of LLMs' Preferences for Libraries and Programming Languages. In *Findings of the Association for Computational Linguistics: ACL 2026*. doi:10.48550/arXiv.2503.17181

C. Zhang, L. Jia, W. Zhang, and N. Wen. 2024. Functional Programming Paradigm of Python for Scientific Computation Pipeline Integration. *arXiv preprint arXiv:2405.16956* (2024). arXiv:2405.16956 doi:10.48550/ARXIV.2405.16956