Dynamically Checked Deep Immutability in Python

FRIDTJOF STOLDT, Uppsala University, Sweden
SYLVAN CLEBSCH, Microsoft Azure Research, USA
MATTHEW A. JOHNSON, Microsoft Azure Research, United Kingdom
MATTHEW J. PARKINSON, Microsoft Azure Research, United Kingdom
TOBIAS WRIGSTAD, Uppsala University, Sweden

Immutability is common in the programming mainstream: deep immutability is the default in functional languages while imperative languages typically provide opt-in support for shallow immutability, usually enforced through static checking.

Python is a dynamic imperative language where mutability is inherent: not only are most objects mutable, but programs themselves—modules, classes, functions—are represented by mutable objects at run-time, and libraries routinely rely on this mutability. This makes adding immutability to Python a significant challenge.

This paper presents the design and implementation of deep immutability for Python. Our primary motivation is to permit multiple sub-interpreters to directly share object references, which currently requires costly serialisation. Sharing via immutability introduces a soundness challenge, as a violation could corrupt the interpreter's state.

We identify numerous challenges that stem from decades of design decisions that did not anticipate immutability, and show how they can be overcome through two complementary techniques: *detachment*, which severs run-time links that would cause immutability to propagate too widely, and *freezability*, which gives objects run-time control over whether and how they may become immutable. Together, these principles form a general design pattern for deep immutability in dynamic languages. We validate our design with an implementation on CPython 3.15 that is backwards-compatible with existing programs and enables direct, zero-copy sharing of immutable objects across sub-interpreters.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Computing methodologies → Concurrent programming languages**.

Additional Key Words and Phrases: Immutability, Concurrency Safety, Dynamic languages, Python

ACM Reference Format:
Fridtjof Stoldt, Sylvan Clebsch, Matthew A. Johnson, Matthew J. Parkinson, and Tobias Wrigstad. 2026. Dynamically Checked Deep Immutability in Python. *Proc. ACM Program. Lang.* 10, PLDI, Article 274 (June 2026), 24 pages. https://doi.org/10.1145/3808352

1 Introduction

Recent years have seen two parallel attempts to overcome the performance implications of Python's global interpreter lock (the GIL). The GIL only permits one Python thread at a time to execute bytecodes which causes all Python code to run sequentially. Python manages memory through a combination of reference counting and tracing garbage collection (GC), and the GIL is crucial for correctness of reference count manipulations. Without the GIL, both correctly and incorrectly

Authors' Contact Information: Fridtjof Stoldt, Uppsala University, Uppsala, Sweden, fridtjof.stoldt@it.uu.se; Sylvan Clebsch, Microsoft Azure Research, Austin, USA, sylvan.clebsch@microsoft.com; Matthew A. Johnson, Microsoft Azure Research, Cambridge, United Kingdom, matjoh@microsoft.com; Matthew J. Parkinson, Microsoft Azure Research, Cambridge, United Kingdom, mattpark@microsoft.com; Tobias Wrigstad, Uppsala University, Uppsala, Sweden, tobias.wrigstad@it.uu.se.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART274
https://doi.org/10.1145/3808352

synchronised programs can race via reference count manipulations., which in turn can lead to use-after-free errors, and memory corruption.

The on-going work on "Free-threaded Python" [12] aims to enable parallelism by removing the GIL completely. It makes all reference count manipulations atomic, using atomic exchanges to avoid racing field updates to cause reference counting to be applied to the wrong objects, and introduces per-object locks in container objects to ensure integrity of their operations. Free-threaded Python ensures the integrity of the Python interpreter, but user programs can break due to data races.

Python 3.12 permitted multiple isolated Python "sub-interpreters" to run with separate GILs [19] in the same Python process, which allows parallelism between threads in different sub-interpreters. From an implementation standpoint, this is a much simpler way to avoid the limitations of the GIL than free-threading, as each sub-interpreter is largely unmodified. As it keeps the GIL-based design, existing C extensions can continue to rely on the GIL for correctness. Isolation is enforced by giving each sub-interpreter its own completely separate state. Every interpreter maintains an independent `PyInterpreterState` containing its own module dictionary, memory allocator, and global variables, together with a dedicated GIL that guards only that interpreter's threads.

Rather than tracking ownership per object, CPython enforces isolation structurally: each interpreter manages its own object space and memory, and the interpreter prevents references from crossing between interpreters through strict API boundaries. As a result, memory management, reference counts, and object lifetimes remain confined within a single interpreter, and communication between interpreters must instead occur through explicit message-passing mechanisms such as channels introduced by Python Enhancement Proposal (PEP) 554 [20, 21].

Stoldt et al. [23] recently proposed a design for dynamically-enforced region-based ownership based on work by Arvidsson et al. [1] and Cheeseman et al. [8] as a way to add parallelism to Python. The design avoids the data races possible in Free-threaded Python without the strict isolation needed by multiple sub-interpreters. While their work focuses on efficient movement of *mutable objects* between threads or sub-interpreters, their approach relies on deep immutability to support safe sharing of objects which must be concurrently accessible without blocking. For example, the object representing Python's string type, but also user-defined objects and types. Notably, Python does not have explicit support for immutability, and a proposal for adding it was rejected in 2005 [28].

The work by Stoldt et al. [23] assumes that immutability can be added to Python. This is the inspiration for our work: exploring the design space of adding immutability to Python, a 35 year-old language without sound static type checking. Python programs are imperative and rely on mutability for many programming patterns including metaprogramming.

Contributions. The lack of sound static checking in Python, and reliance on mutable declarations and metaprogramming, means that well-understood static type-based approaches to immutability do not work. We have to move the *time* when modules, types, and objects become immutable from compile-time to run-time. This presents an especially hard challenge in languages like Python where types are represented as normal mutable objects in the object graph, and there is a surprising level of interconnectedness between objects in the run-time. This requires us to design ways to handle *freeze propagation*, *i.e.*, how far effects of making something immutable reaches. In this paper, we tame freeze propagation in two main ways: *detachment* and *freezability*. Together, they form a general design pattern for integrating deep immutability into dynamic, reflective languages.

Detachment is the act of severing links between objects in the run-time to stop immutability from propagating *too far*. We design five detachment techniques for immutable objects: detaching a function from the global scope; a nested function from its enclosing function's stack frame; a function from its stateful module import; a mutable sub-object from its immutable parent; and a mutable field from its enclosing immutable object. Without detachment, immutability in languages

like Python would be severely restricted as often, making an object immutable would incapacitate the entire system by making "everything" immutable.

Freezability is a new property of objects that controls if and how they may become immutable. This is required by the lack of a static type system that can propagate immutability at design-time, and check for internal consistency. Making types (and objects) freezable permits them to change (including metaprogramming) until becoming immutable. Explicit freezability permits objects to be made immutable directly, but not as a side-effect of making something else immutable.

One of our goals for immutability is to permit direct, zero-copy sharing of immutable objects across sub-interpreters. This enables data-race free parallelism in Python without having to give up the GIL. This requires rethinking how immutable objects' memory is managed as Python's memory management assumes a single sub-interpreter manages memory in isolation. We build on prior work [17] on managing immutable data by leveraging a strongly-connected component (SCC) analysis, adapting that work in non-trivial ways to handle external references into SCCs, rollback in case of failure, and adding support for weak references.

Outline. §2 gives a brief background on immutability. §3 states our goals for this work and discusses the challenges due to pervasive global state, objects' interconnectedness, and memory management design. §4 introduces a simple protocol for making objects immutable at run-time, §4.2 introduces detachment, and §4.3 introduces freezability. §5 discusses memory management without a GIL, SCCs, as well as noteworthy details of our implementation. §6 discusses our goal-fulfilment, compares the time it takes to make an object graph immutable with time it takes to serialise it, test coverage, and investigates the overhead of our implementation.

2 Background

Immutability is a well-known and well-explored concept in several settings [11, 18]. For example, immutability is the norm in functional programming languages and only special data types like "ref cells" (etc.) support operations that modify objects in-place. As shown in Table 1, many imperative languages support some notion of immutability. This immutability is typically *shallow* meaning that an object is considered immutable if its fields cannot change—however, the fields may reference objects which are mutable. *Deep* immutability in contrast means that sub-objects must also be immutable. Most scholarly works on immutability are concerned with deep immutability [5, 9, 15, 16, 25] but not all [6, 14]. None of the main dynamically typed imperative languages (including JavaScript, Python, Ruby, Lua, Perl, PHP, and Smalltalk) have built-in, general-purpose deep immutability. The dynamically typed languages Erlang and Clojure have deep immutability, stemming from their functional nature. Their approach is to eliminate mutation syntax rather than dynamically enforcing immutability. In contrast, most dynamically typed imperative languages require run-time enforcement because mutating operations are a fundamental part of the language. Ruby and JavaScript both provide a `freeze()` function that makes objects *shallow* immutable. This avoids propagation to *e.g.*, an object's class. Scheme, by contrast, encourages functional programming but retains pervasive mutable constructs. It does not provide a built-in notion of deep immutability. Immutability arises from programmer discipline, not semantic enforcement.

Immutable vs. Read-Only. It is important at this point to distinguish between *immutable* and *read-only*. Immutability is a property of an object—an immutable object cannot be modified; read-only is a property of a reference—a particular reference cannot be used to mutate its referent. Read-only references do not preclude the existence of other references through which mutation is permitted. Read-only references may witness the mutation through such references; it is thus a considerably weaker property. For a while, the term "reference immutability" was used synonymously with

Table 1. Comparison of immutability across languages. TS Enforced means enforced by a static type system. Legend: † – Uniqueness types; ‡ – Ownership system. ◊ – Via `final`; `val`; `let`; or `const`.

Language	Immutability	TS Enforced	Comment
Haskell	Deep	Yes	Purely functional
Clean [7]	Deep	Yes†	Optimises mutation internally
ML / OCaml / F#	Default	Yes	Explicit mutable refs allowed
Erlang	Deep	Yes	Purely functional
Elm	Deep	Yes	No mutation
Rust	Deep	Yes‡	Mutation under borrow rules
Pony [10]	Shallow / Deep	Yes	Imperative actor language
Python	Shallow	No	Convention-based, named tuples
Java	Shallow	Partial◊	Deep immutability via discipline
Kotlin	Shallow / Deep	Partial◊	Encouraged via data classes
Scala	Deep	Partial	Functional collections
Clojure	Deep	Yes	Functional
Swift	Shallow / Deep	Partial◊	Strong for value types (structs)
TypeScript	Shallow	No	Libraries add deep immutability
JavaScript	Shallow	No	`Object.freeze(`*obj*`)`
Ruby	Shallow	No	*obj*`.freeze()`
Scheme	Shallow	No	Explicit, principled support built-in
C++	Shallow / Medium	Partial◊	Manually enforced
Go	Weak	No	Convention-based
Julia	Deep (optional)	Yes	`struct` vs `mutable struct`
Dart	Shallow / Deep	Partial◊	Deep immutability at compile time

read-only references [3, 18, 26]. This is an unfortunate misnomer as none of the strong immutability guarantees apply to "immutable references". Read-onlyness is not relevant in this work.

2.1 Support for Immutability in Python

The Python programming language does not support immutability as a language concept. A small number of core Python *type objects* including `bool`, `int`, `float`, `str`, `complex`, `tuple`, and `frozenset` do not permit mutation, and their *instances* are shallow immutable. Attempting to assign to attributes of these objects is rejected dynamically and gives rise to a readonly attribute error. Notably, the root class `object`'s type object is immutable. Except for references to their immutable type, instances of `bool`, `int` and `float` do not contain references to other objects so they are deeply immutable; instances of `str` and `complex` only have immutable sub-objects so they are deeply immutable too; shallow immutable tuples are deeply immutable if they store only deeply immutable arguments.

Immutability is not a reified concept in Python—it is not generally possible to introspect on an object's support for mutation; one can simply try to mutate it and fail. The closest to a reified notion of immutability is the "frozen" concept. For example, the shallow immutable frozen set mentioned above and data classes that define a frozen attribute that controls shallow immutability.

2.1.1 Strictness of Immutability. The discussion above implicitly assumed a non-strict definition of deep immutability. For example, the number 4711 has a mutable reference count which is manipulated implicitly by assigning from 4711 to variables and fields, and inspected through the `sys.getrefcount` API. Similarly, every type object maintains a list of all direct subclasses of the type, which can be extended by evaluating a new class definition that extend the type, and inspected through the `__subclasses__()` method that each type defines.

```
class Frozen:
  __slots__ = ("x", "y", "_frozen")

  def __init__(self, x, y):
    object.__setattr__(self, "x", x)
    object.__setattr__(self, "y", y)
    object.__setattr__(self, "_frozen", True)

  def __setattr__(self, name, value):
    if getattr(self, "_frozen", False):
      raise AttributeError("Obj is immutable")
    object.__setattr__(self, name, value)
```

Fig. 1. Implementing shallow immutability using `__slots__` and `__setattr__()`.

The implementation of, *e.g.*, reference counts and subclass lists as object attributes is a pragmatic choice. From a philosophical perspective, it makes sense to permit these attributes to stay mutable even though their

enclosing object is immutable. However, as our goal with this paper is to enable the direct sharing of objects across sub-interpreters this choice has far-reaching consequences. As we shall see in §3.6, permitting reference counts to stay mutable when objects are shared across sub-interpreters requires additional machinery for reference count manipulations to stay correct. Same for subclass lists.

This kind of pragmatism is not unique to Python. Many languages provide escape hatches for immutability, *e.g.*, C++ provides a `mutable` keyword that permits members of `const` objects to be modified, and a `const_cast` operation reinterpreting `const` pointers as mutable, thus permitting any mutation. Rust provides an `UnsafeCell<T>` which is used to implement smart pointers and synchronisation primitives like `RefCell` and `Mutex`.

```
class Frozen:
  def __init__(s, x, y):
    s._x = x
    s._y = y

  @property
  def x(self):
    return self._x

  @property
  def y(self):
    return self._y
```

Fig. 2. Shallow immutable via `@property`

2.1.2 Immutability and User-Defined Types. Python permits a user-defined type to implement shallow immutability by a combination of `__slots__` which prevents adding and removing fields of an object and `__setattr__()` which redefines field assignment. The class definition in Fig. 1 redefines field assignment to inspect the value of `_frozen`. If `_frozen` is `True`, assigning to fields raises an `AttributeError`. Another approach is shown in Fig. 2 that uses the `@property` decorator to prevent reassigning the `x` and `y` attributes, but new (mutable) fields can be added to the object because it does not use `__slots__`.

2.1.3 PEP351: The Freeze Protocol. PEP 351 [28] was published in 2005 and rejected based on a discussion on the python-dev mailing list that took place October 2005 to February 2006 [13]. It describes a protocol for requesting an immutable copy of a mutable object. It defines a new built-in function, `freeze()`, which uses this protocol to provide an immutable copy of cooperating objects.

The intended semantics of PEP351 was as follows: When a mutable object *o* is passed to the built-in `freeze()` function, it calls *o*'s `__freeze__()` method to get an immutable copy. (This is notably different from `freeze()` in Ruby or JavaScript, which make an object shallow immutable in-place.) If *o* does not have a `__freeze__()` method (*i.e.*, it is not cooperating), a `TypeError` is raised. Importantly, the `__freeze__()` method was intended as a programmer-defined method which could be implemented in any way. This level of trust in programmers to "do the right thing" is common in Python's design. A call to `__freeze__()` could in principle return the unmodified `self` object, or if using the design in Fig. 1 set the `_frozen` flag to `True` before returning `self`. (As opposed to returning a copy, aligning with Ruby and JavaScript

PEP351 was rejected due to a "general feeling" that there were no compelling use cases for immutability (the few motivating examples in the PEP could be implemented in other ways) and that freezing should not be applied to arbitrary objects but be an essential and explicit part of object design. In 2024, a discussion about a potential resurrection [24] of PEP351 motivated by a need for "a parallel-read-access dictionary and potentially other data structures to complement free-threaded Python" surfaced on `discuss.python.org` but no PEP proposal has yet come out of this discussion.

3 Challenges for Adding Immutability to Python

Let us first establish and motivate our goals for adding deep immutability to Python in the first place. We split these into two groups: functional and non-functional. These goals are compatible with—or inherited from—the goals of Stoldt et al. [23], which was the inspiration for this work.

Our functional goals are: (G1) a notion of immutability that is strong enough to enable direct (data-race free) sharing of immutable objects across Python sub-interpreters; (G2) not breaking any existing code; (G3) support existing programming patterns that are incompatible with a strict definition of immutability by providing escape hatches in a safe way; and to (G4) embrace Python's dynamic nature and only reject code when it actually violates immutability.

Our non-functional goals are: (G5) minimum performance impact on programs that either do not use immutability or use it without raising dynamic errors; (G6) immutable objects should enjoy the same promptness of reclamation as normal Python objects; and finally (G7) have a reasonable complexity budget as to not impact the burden of maintaining the language too negatively.

3.1 Reification and Interconnectedness

Python represents most things in a program as an object at run-time: types, functions, and modules are all represented as *mutable* objects. Fig. 3 shows a simple Python program, and Fig. 4 shows the resulting object graph. While the graph is a simplification[1], it shows that the Python object graph is considerably more complex than the naive object graph programmers may imagine. For illustrative purposes, the die class has two methods for rolling the die. The program also imports the `random` module to call the `randint` function via the module object as well as the directly imported `randint` function under the name `_randint`.

```
1  import random
2  from random import randint as _randint
3  class Die(object):
4    def __init__(self, sides):
5      self.sides = sides
6    def roll1(self):
7      return random.randint(1, self.sides)
8    def roll2(self):
9      return _randint(1, self.sides)
10
11 d = Die(6)
```

Fig. 3. A simple `Die` class and a 6-sided instance.

Note: From this point on, we will use immutable to mean *deeply* immutable.

In Fig. 4, the stack frame (`locals`) has the variables `d`, `Die`, `random`, and `_randint` pointing to a die object, a type object, a module object and a function object respectively. The die has a reference to its type object which has references to three function objects for its functions. The `roll1` function object has a reference to `random` whereas the `roll2` function object has a reference to `_randint`.

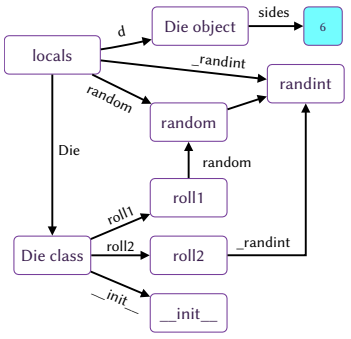


Fig. 4. Simplified overview of Python's object model. "Ice blue" = immutable.

We want immutability to enable direct sharing of objects across sub-interpreters. If we naively share the object referenced by `d`, all the objects above would become shared as a result. Below, we discuss the challenges that stem from this design: mutable type objects (§3.2), function objects and their interaction with enclosing mutable state (§3.3), mutable module state and its interplay with sub-interpreter isolation (§3.4), access to global state from extensions written in C (§3.5), and finally challenges due to Python's strategy for memory management (§3.6).

3.2 Programs Expect Mutable Type Objects

As also pointed out by Stoldt et al. [23], user-defined types in Python are represented at run-time by *mutable* objects. Type objects can be manipulated in a way that instantly becomes visible to all their instances. Many Python libraries rely on this ability to perform so-called "monkey-patching" of types. Common uses include testing, metaprogramming, and boilerplate code generation.

Mutable type objects introduces a challenge for deep immutability as every object has a reference to its type object. Thus, to create an immutable instance of class *C*, the type object for *C* must also be immutable. Furthermore, since many Python programs rely on user-defined types being mutable, making type objects immutable by default is not possible, as this contradicts (G3). Making an immutable copy of a type to serve as the type for immutable objects does not scale: first it will

[1]This program gives rise to many more objects than shown, for example all functions in the `random` library, but also the super type of `Die`, the types of the function objects, the module object, and the `type` object — the type of all types.

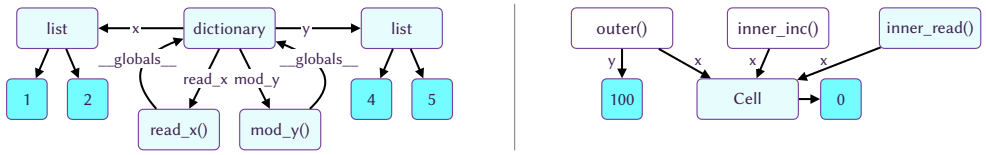


Fig. 6. Simplified object structures. Left (a)—the program in Fig. 5. Right (b)—the program in Fig. 7. Lightly shaded cells show propagation of immutability if we made `read_x()` or `inner_read()` immutable.

parallelise the entire type hierarchy as each type has a pointer to its super class; second if the same mutable type is modified n times and immutable objects created of each version of the type, we end up with n parallel hierarchies. This complicates the semantics of instance-of tests, not to mention serving as a likely source of confusion for programmers.

Occasionally, there is a need for Python types to contain mutable state, even for types which are designed to be effectively immutable. One such example is found in Python's `Fraction` class which uses a mutable `lru_cache` from the `functools` library to implement hashing efficiently, as a fraction such as $\frac{9}{3}$ should hash identically to 3. `Fraction` *instances* have no way to modify their numerator or denominator but because the `Fraction` class' `__hash__` function captures mutable state (*cf.*, §3.3.2), they are fundamentally incompatible with strict deep immutability.

3.3 Functions are Defined in an Inherently Mutable Context

Functions are a fundamental part of any Python program. Most types contain several methods, which are internally represented as function objects. Making a type immutable thus requires that function objects can be made immutable.

```
1 # Program 0
2 x = [1, 2]
3 y = [4, 5]
4
5 def read_x(idx):
6   return x[idx]
7
8 def mod_y(o):
9   y.append(o)
```

Fig. 5. Creates the object graph in Fig. 6a.

Functions in Python are surprisingly complex. When Python encounters a function definition it creates a function object which can be called at a later point in time. Function objects contain references to any external state captured by a function definition. There are different ways how external state is represented and captured, depending on whether the external state comes from the enclosing module or an enclosing function, in which case the captured state originates from a stack frame. For now, we confine ourselves to functions defined in Python. Functions defined in C pose additional challenges discussed in §3.5.

3.3.1 All Functions Capture Global State. We focus for now on the global state. Fig. 5 shows a program with two functions and Fig. 6a shows the object structure created by the running program. In Python, every function object has a reference to the globals dictionary of the scope it was defined in, stored in the `__globals__` attribute. This reference is used to resolve variables found in the local scope. The functions `read_x` and `mod_y` both use the reference to the dictionary of their defining scope to find the global variables `x` and `y` defined on Line 2 and Line 3 respectively. Note that the reference to globals is there, even if a function never accesses it.

Say that we want to make the `read_x()` function object immutable. This requires making its type immutable, and critically for this section, also the globals dictionary which is reachable from the function object. Since everything is reachable from globals this *makes all other objects in the program in Fig. 6a immutable.* In principle, not all of these need to be deeply immutable for`read_x()` to be immutable, *e.g.*, `mod_y()` and the list in `y` that are not accessed by `read_x()`.

It is not generally possible to analyse a Python function to see what parts of globals it accesses due to Python's powerful metaprogramming facilities. For example, if a function with parameters `f` and `v` performs `f(v)` this could in principle be a lookup from the global namespace if `f` is bound to the builtin function `globals()` and `v` to some variable name as a string. And even if we could tell exactly

what variables are captured and make their values immutable, the namespace is a normal dictionary and does not support preventing keys (the variables) from being reassigned. Second, in principle, it is possible to access anything in the global namespace through a function object even though the function it represents does not do so. (Meaning *e.g.*, `read_x.__globals__["y"].append(6)` is possible.)

Making all global functions immutable is not possible as many modules and even the Python interpreter make use of mutable global variables. This makes such an approach a non-starter.

3.3.2 Functions Capturing Shared Mutable State. In Python, functions that capture external state are represented differently at run-time depending on whether the external state comes from the enclosing module namespace (discussed in §3.3.1) or from an enclosing function. Fig. 7 shows a program with a function called `outer()` and two nested functions defined inside it. Fig. 6b shows how the object graph would look like on Line 13. The three functions share a cell object that contains the actual value which is used by Python. This indirection allows both functions to reassign `x` in a way that is visible to the other function. This allows the reassignment in Line 10 to be observed by the `outer()` and `inner_read()` functions. Also note that only variables captured from an enclosing function use this indirection. The variable `y` remains a direct reference to the contained object because neither nested function captures it.

```
1  def outer():
2    x = 0
3    y = 100
4
5    def inner_read():
6      print(x)
7
8    def inner_inc():
9      nonlocal x
10     x = x + 1
11     return x
12
13   return (inner_read,
14           inner_inc)
```

Fig. 7. `inner_inc()` captures `x` from the frame of `outer()`.

Making `inner_read` immutable requires all its reachable objects to be immutable which include the shared cell `x`. This is shown by the light shading in Fig. 6b. This prevents the reassignment on Line 10 in `inner_inc`, even though `inner_inc` is mutable.

3.4 Making Functions Immutable can Incapacitate Modules with Mutable State

Python modules are represented at run-time as mutable module objects with fields for each top-level declaration pointing to types, functions or ordinary Python objects. Stoldt et al. [23] exemplified mutable module state with the `math` and `random` modules: it is reasonable to turn mathematical constants such as `math.pi` into actual constants that cannot be reassigned, but the `random` module needs mutable state to function.

The leftmost code in Fig. 8 shows a simplified version of Python's `random` module. This module defines a `Random` class with internal state mutated on Line 9. The top-level function `random()` is a *bound method*, *i.e.*, a pair of a function object and its bound self: `(Random.next, _inst)`. The module also stores a reference to the `Random` class object. Line 1 states that instances of `Random` should not be made immutable. If that happens, calls to `next()` will fail due to the update on Line 9.

Python's `import` statements are versatile. Their location and usage influence the object graph and therefore also how immutability interacts with module objects. Fig. 8 shows the three main ways modules are usually imported and used.

- Program 1 imports the module at the top-level.[2] This defines a new global variable `random` referencing the module object. This module object is then captured by the `coin_1()` function, as discussed in §3.3.1, to be used to call the `random()` function in Line 6.
- Program 2 imports the module inside the `coin_2()` function. Each time the function is called, it imports the random module and store a reference to its module object in the local variable `random`. Thus, the function does not capture any external state.

[2]One can think of **`import`** `name` as desugaring to `name = __import__("name")`. The latter function is a builtin function importing modules by name as string, returning a module object.

```python
# Instances of Random should be mutable
class Random:
  def __init__(self, seed=1):
    self.state = seed
  # Return float in [0, 1)
  def next(self):
    s = self.state
    s = (110 * s + 123) % 0xffff
    self.state = s
    return (s & 0xff) / 0xff

_inst = Random()
random = _inst.next
```

```python
# Program 1

import random

def coin_1():
  r = random.random()

  if r < 0.5:
    return "Heads"
  else:
    return "Tails"
```

```python
# Program 2

def coin_2():
  import random

  r = random.random()

  if r < 0.5:
    return "Heads"
  else:
    return "Tails"
```

```python
# Program 3

from random import random

def coin_3():
  r = random()

  if r < 0.5:
    return "Heads"
  else:
    return "Tails"
```

Fig. 8. Three programs showing how a Random module can be imported and used in slightly different ways. All programs behave the same while the function and module objects are mutable.

– Program 3 imports only the `random()` function from the module and adds it to the globals of the program. The `coin_3()` function then captures the method object stored in `random` as part of its globals and uses it in Line 6.

To make the `coin_3()` function immutable we have to make the captured `random` method object immutable. This will propagate to `random._inst`, meaning that `random._inst.next()` calls will fail.

Before looking at Programs 1 and 2 we have to understand how module imports in work in CPython. When the interpreter encounters a `import module` statement, it checks if the requested module is already imported and cached in the `sys.modules` dictionary. If not, a new module object is created and the module is initialised. The module object is then added to the `sys.modules` dictionary to be used by following imports. Finally, the import statement then retrieves the module object from the `sys.modules` and adds it to the current scope. In other words: *CPython initialises each module only once. Subsequent imports are served directly from the* `sys.modules` *dictionary.*

In CPython, the import statement in `coin_2()` will always retrieve the same module object from the `sys.modules` dictionary (unless this has been overwritten by a user). The behaviour of Programs 1 and 2 is therefore equivalent. However, they behave differently when immutability is introduced. Program 2 remains working when `coin_2()` is made immutable, since the function does not capture anything and always loads the current module when called. Program 1, on the other hand, captures the `random` module object as part of the globals of the `coin_1()` function object, see §3.3.1. The resulting object graph is shown in Fig. 9. Thus, making `coin_1()` immutable propagates into the random module, incapacitating it by preventing the reassignment on Line 9 in `next()` in Fig. 8 on subsequent calls.

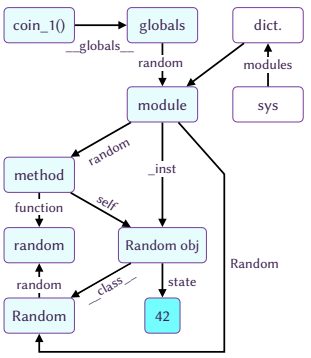


Fig. 9. Object graph from Program 1 in Fig. 8. For `coin_1()` to be immutable, all lightly shaded objects must be immutable too.

3.5 C Extensions Have Access to Global Mutable State

One feature specific to CPython is a C interface which allows the addition of new built-in object types, calling C library functions and performing system calls. This feature is often used by modules that require hardware access or native execution speed, like NumPy, or avoid GIL limitations.

C Modules used to store global state in static variables which restricted them to be imported once per process. PEP 3121 [27] introduced per-module state which provides storage possibilities inside the module object. This allows a module to be imported several times with independent

states and is required by modules to support sub-interpreters. Per-module state is now the intended way of having global state in a module. This poses a few unique challenges regarding immutability:

(1) The per-module data from CPython's perspective is a `void*` pointer. The data inside this pointer is defined by the module it belongs to. The module is free to mutate its per-module state without any restrictions. If we would allow module objects to be immutable, there would be no way of externally ensuring that the per-module state stays unchanged. Thus, modules written in C could not be made immutable safely.
(2) If C modules can be made immutable, they can be accessed by multiple interpreters at a time without the protection of the GIL. Such modules would therefore need to implement correct synchronisation to update the per-module state in a thread-safe way.
(3) Function objects wrapping C functions and types defined in C store a pointer to the module that defined them. This pointer is needed to make the per-module state accessible inside functions and methods of the objects. However, it also means that every function or type object points to a module object which must be immutable for the object to be immutable.

3.6 Memory Management in Python Relies on the GIL

In Python 3.12, each sub-interpreter runs with its own GIL and maintains fully independent state. This includes separate loaded modules, distinct module search paths, independent built-in objects, and a private tracing garbage collector to manage cycles. Threads and Python objects belong to a single interpreter, and C extension state is generally not isolated unless explicitly designed to be. As a result, objects and modules cannot be safely shared across interpreters without explicit bridging mechanisms such as serialisation. This strict isolation provides memory and execution safety but introduces significant challenges when trying to coordinate work or share state between sub-interpreters, particularly for libraries not designed for concurrent use by the same Python process.

Fig. 10 illustrates two challenges when sharing objects that stem from sub-interpreter isolation. It shows sub-interpreters Sub 1 and Sub 2, each with its own sub-heap. Almost all objects in Python are part of a doubly linked list used by a per-sub-interpreter garbage collector to find unreachable cycles. These references are represented in grey as they are inaccessible from Python programs. References are annotated with the GIL used by a sub-interpreter when operating on the reference.

Notably, *Obj2* has incoming references whose accesses are synchronised using *different* GILs. This means that two assignments `x = Obj2` and `y = Obj2` executing in parallel in Sub 1 and Sub 2 respectively will not synchronise on a common GIL, meaning non-atomic reference count manipulations on *Obj2* can race.

Beyond the problem of thread-safe reference count manipulations lie thread-safe cycles as cycles can span sub-interpreters: unless Sub 2 is stopped while Sub 1 is performing garbage collection, concurrent manipulations from Sub 2 can destroy the correctness of cycle detection. This can lead to memory corruption and use-after-free. As the reference from *Obj3* to *Obj2* in Fig. 10 is not visible to sub-interpreter 1's garbage collector, sub-interpreter 1 does not know what other sub-interpreter to synchronise with when performing garbage collection.

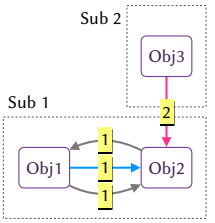


Fig. 10. Direct sharing of an object across two sub-interpreters.

4 Designing Deep Immutability for Python

Python's type system is not strict enough to soundly enforce immutability. This forces us to adapt a dynamic solution. To tame how immutability can propagate through the system, we introduce two complementary principles. *Detachment* (§4.2) addresses the structural problem: it severs specific runtime references—through globals, closures, modules, or object substructure—that would otherwise

cause immutability to spread far beyond what programmers intend. *Freezability* (§4.3), in contrast, provides the policy mechanism that determines whether and under what conditions an object may become immutable at all. Together, detachment constrains the reach of immutability, while freezability governs the permission to enforce it. These two principles form the foundation of our dynamic approach to deep immutability.

4.1 Freezing and Introspecting Mutability Dynamically

The two basic building blocks of our design are the two functions `freeze(`*obj*`)` and `is_frozen(`*obj*`)`. The former makes the transitive closure of objects reachable from *obj* immutable and the latter returns true if all objects transitively reachable from *obj* are immutable.

This notion of immutability is deceptively simple. As we have seen, there are a lot of hidden complexities in the design and implementation of Python that break this simple model.

For concreteness, let us return to Fig. 3 and append a line `freeze(d)`. The programmer's intention is, we speculate, to simply make the 6-sided die instance immutable, say to preserve the outcome of a good roll. Now consider Fig. 4. With our knowledge from the previous section, we know there is also a reference from the the die object referenced by `d` to the `Die` class object and references from the function objects back to the modules where they were defined, *e.g.*, from `__init__` back to locals and from `randint` back to its defining module object random. Consequently, the call to `freeze(d)` will end up *making all objects in Fig. 4 immutable.* This illustrates both the necessity of controlling propagation: by severing links between objects, and by preventing certain objects from ever becoming immutable.

4.1.1 Immutability by Construction. To avoid reasoning about how freezing arbitrary objects propagates through the object graph, immutability by construction is an appealing approach. As hinted to by the immutable numbers in our figures, we consider all objects of core Python which are inherently deeply immutable (§2.1) immutable in our system. Complex immutable structures must be constructed from the inside out, but immutable cyclic structures and structures involving user-defined types cannot be created. Fig. 11 shows typical cases.

```
t1 = (1, (2, (3, None))) # Immutable list
is_frozen(t1) # True

name = "Eric"
is_frozen(name) # True

from collections import namedtuple
Person = namedtuple('Person', ['name'])

# Create deeply immutable named tuple
eric = Person(name)
is_frozen(eric) # returns True
```

Fig. 11. Immutability by construction.

4.1.2 Dynamic Immutability. Instances of user-defined types as well as lists and dictionaries, mutable objects can be made immutable using the `freeze()` function introduced above. We sketch the implementation of `freeze()` in Fig. 12, using Python's metaprogramming facilities, restricting it to a single argument, and assuming the existence of the function `mark_as_immutable()`.

The `dir()` function returns a list of names of the attributes of the object, *e.g.*, `sides` in the case of a die. The function `getattr(`*o*`,`*a*`,`*v*`)` looks up the value of attribute *a* in object *o*, returning *v* if the attribute is does not exist. Given the existence of a `is_marked_as_immutable()` function, we could implement `is_frozen()` in a similar way, using type tests to identify (shallow) immutable objects.

```
def freeze(obj):
  mark_as_immutable(obj)
  for attr_name in dir(obj):
    value = getattr(obj, attr_name, False)
    if value and not is_frozen(value):
      freeze(value)
```

Fig. 12. Freeze function sketch.

4.1.3 Types are Made Immutable at Run-Time Through an Explicit `freeze()` Call. Making objects immutable through an explicit `freeze()` call extends naturally to dealing with monkey-patching.

Making types mutable on creation and optionally being made immutable later through an explicit `freeze()` operation in user code fits well with the design of Python as declarations of classes and functions are evaluated at run-time in the order they are discovered by the interpreter. While this delay gives the programmer the necessary flexibility to perform monkey-patching before making a type immutable, it also introduces challenges for reasoning about the effects of making a type immutable in a running program. We discuss a type's freezability in §4.3.

Type objects keep a list of references to their subclasses for pragmatic reasons which poses two problems for immutability: making a class (say `object`) immutable will propagate to its subclasses; it is not possible to subclass immutable classes. While this link between type objects is "incidental", the effects of it are real. We discuss how we can sever such links in §4.2.

4.1.4 Self-Containedness: A Non-Starter. An early question from a Python programmer suggesting that immutability should be tracked in Python's type system was: In this example, how can we tell that the call `func(x)` in the 2nd statement has not made `x` immutable causing the append to fail?

```
x = [1, 2, 3]1 ; func(x)2 ; x.append(4)3
```

To limit the need to reason about freeze propagation, we initially explored requiring that object graphs be *self-contained* to be frozen. Assume `freeze(x)` would cause all objects in the set G to become immutable. Self-containedness requires that every reference r to an object in G originates from within G, or is the reference stored in `x`. Otherwise, freezing with fail. A similar constraint is used in many statically checked type systems that permit (externally) unique objects to be made immutable [1, 4, 9, 16]. This is essentially the same as a copying solution (à la the Freeze Protocol in §2.1.3), except that the copying is not needed as all witnesses to objects changing from mutable to immutable are inside the graph that turns immutable. In our setting, requiring that object graphs are self-contained to be frozen would have prevented `func(x)` from freezing `x`, since the calling frame retains a reference (assuming the calling frame is not reachable from `x`).

We abandoned this idea for three reasons which have been recurring in this work, and is worth understanding before continuing with the rest of the paper:

- *There is more than one Python.* For our implementation to be efficient, we relied on reference counting, which is how memory is managed in CPython: counting the number of edges in the graph being frozen as we traverse it and comparing the result with the sum of the in-degrees as reported by each node's reference count. This implementation was rejected by Guido van Rossum (Python's creator) as not all versions of Python rely on reference counting, nor should we constrain future version of CPython to use reference counting.
- *Python is a moving target.* In parallel with our work, CPython adopted (coming in the next release) deferred reference counting [2]. This change was motivated by work on Free-threaded Python that uses atomic reference counts on shared objects. With deferred reference counting, reference counts are a safe under-approximation of the in-degrees of a node in the object graph. With deferred reference counting, some references become "borrowed" and do not cause the reference count to change, which means a loss of precision for our implementation of the self-contained check. In the call to `func(x)`, the `x` reference would be borrowed, meaning that func could now freeze `x`. This is in principle fixable, but points to the following underlying problem.
- *Python programmers are domain specialists, not programming language specialists.* Python programmers should not be expected to understand how reference counting works—or how memory management is implemented. In particular, they should not have to understand when deferred reference counting elides or does not elide reference count manipulations in order to debug an unexpected failure to freeze, or an unexpected success.

4.2 Detachment

In this section we describe five techniques for detaching objects to stop the propagation of immutability. Detachment only affects immutable objects—the semantics of mutable objects is unaffected.

4.2.1 Detaching Globals Variables. We handle capturing of global state in two ways. First, when we make a function f immutable, we create a fresh dictionary and store it in the function's `__globals__` attribute in place of the old global's dictionary. We then populate the dictionary with the keys that the function object knows it has captured (their names are stored as strings in a tuple in f`.__code__.co_names`). Second we freeze the new dictionary along with its contents. This behaviour is shown in Fig. 13: immutability only propagates to the value of `x` while `y` and `mod_y` remain mutable. Giving each immutable function object a private dictionary has three main consequences:

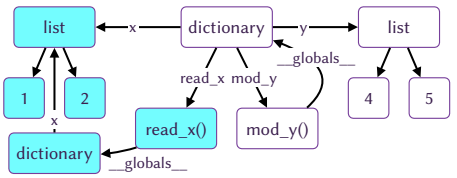


Fig. 13. The result of freezing `read_x()` from Fig. 5. Compare with the "naive" solution in Fig. 6a.

(1) This permits the original globals dictionary to stay mutable, but values which are shared with private dictionaries will become immutable.
(2) Because the immutable function is detached from the namespace where it was defined, code like Fig. 14 is possible. This may seem surprising at first, but it is actually quite intuitive: making a function immutable "locks" the values of its captured variables in place *in the private dictionary of the immutable function*, but not in its enclosing namespace.
(3) Metaprogramming accesses from within an immutable function can now fail when trying to access a variable that was not copied to the private dictionary. For example, evaluating the expression `globals("yx"[0])` inside the frozen `read_x()` function will fail trying to access `y` from its private dictionary since `y` was not "captured explicitly" by the function.

This solution is not "perfect"—but we believe that it is *good enough*. It preserves the intended behaviour of use cases that do not rely on metaprogramming. When names accessed dynamically are known ahead-of-time, explicitly mentioning the variable will cause it to be copied across. Functions that access global variables whose names are computed dynamically, and functions that need to update global variables should not be made immutable.

```
x = [1, 2, 3]
def tally():
  sum = 0
  for v in x:
    sum += v
  return sum

freeze(tally)
tally() # 6
x = [1, 2] # reassign x
tally() # still 6
```

Fig. 14. Freezing locks captured values.

This approach may make objects in the globals dictionary immutable, *e.g.*, the value of `x` in Fig. 6a. We assume that this is the intended behaviour when programmers freeze functions. In §4.3 we discuss ways to preventing objects from being frozen. In addition to the globals dictionary, function objects also store a reference to the `__builtins__` dictionary which behaves the same as the globals dictionary. We handle this dictionary using the same solution as globals.

4.2.2 Detaching Variables from Enclosing Namespaces. As we showed in §3.3.2, when a function f captures a variable x from an enclosing function g, the result is a shared mutable cell. Deep immutability of f requires the cell to be frozen, but this stops g for reassigning x. Also this problem can be solved using detachment. When a function is made immutable, we create private copies of its cells, including the pointer to its contents, and make the copies immutable along with their content. This is shown in Fig. 15. Compare with Fig. 6b where `inner_inc()` is also affected.

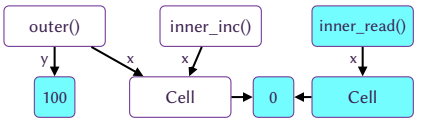


Fig. 15. Detaching a cell object from an enclosing stack frame stops propagation.

This permits the original cells to remain mutable, allowing other functions to reassign their captured variables. An immutable cell will reject any attempt at reassigning its value, raising an exception. Making the function `inner_read` immutable will "lock it" to the current values it captures and will not allow updates to these from the function.

Our solution freezes state which is reachable from other functions. This may lead to errors if functions rely on mutating shared state. We believe that this error is actually good, as it warns programmers that they froze state which is expected to be mutable. Because of detachment, freezing `inner_read()` permits the assignment on Line 10 in `inner_inc()`. The reassignment means that `inner_inc()` will remain working, while `inner_read()` retains the value from when it was frozen.

```
1 x = ...
2
3 def function(arg = x):
4   print(arg)
5
6 class A():
7   field = x
```

Fig. 16. A program showing how the variable x can be used as part of function and class definitions

Functions and classes can capture the *values* of external variables due to initialisation of default arguments and class attributes, see Lines 3 and 7 in Fig. 16. In both cases, the value of x is read when the declaration is evaluated and stored in a field in the resulting object. Freezing this object will propagate to this captured value.

4.2.3 Module Object Detachment. We now revisit the problem exemplified by Fig. 8 where freezing could lead to modules becoming incapacitated. Asking users to refactor their code from the style of Program 1 into Program 2 would be a massive undertaking, even if the change itself is trivial. Even with our detachment from globals introduced in §4.2, the random module would still be frozen if we freeze `coin_1()` since it is reachable from the function object.

Our solution is to make detachment stronger for module objects by replacing a module object with a proxy that causes the module object to be loaded from the current sub-interpreters when the function is called. The solution is shown in Fig. 17.

When we freeze a module object m with name n we first create a copy m' of m and store that copy in a new module dictionary in `sys`: `sys.mut_modules[`n`]`=m'. We then remove all module state from m and use metaprogramming to turn m into a proxy object that transparently delegates all accesses to m' via `sys.mut_modules[`n`]` on the current interpreter. The proxy m will also import the module if it is missing from the interpreter it is being accessed from. m can now be frozen without affecting the mutable state stored in m' since there is no direct reference from m to m' This mechanism is transparent to Python code interacting with the module and subsequent sharing of m is easy as it is deeply immutable.

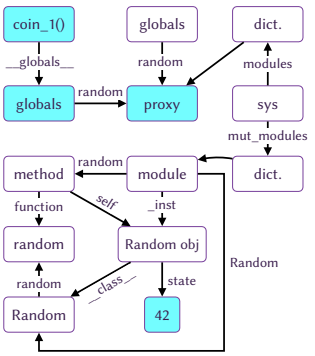


Fig. 17. Freezing `coin_1()` from Fig. 8 detaches the module object and replaces it with a proxy that stops freeze propagation.

The object graph in Fig. 17 shows module object detachment to avoid freeze propagation when a function is frozen, and how the module state remains mutable. The solution effectively turns Program 1 in Fig. 8 into Program 2 from the same figure. Program 2 imports the module inside the function and thus always loads the mutable module object on the current interpreter. This should also keep CPython and its external libraries working, as existing references continue being module objects—we only change their internal state.

This solution may not work for every module. Some modules may require that objects created on one interpreter are only ever used on the specific interpreter that it was created on. This could be, because the object is just a handle to some internal mapping. For such cases, modules can opt-out of this behaviour *cf.*, §4.3.

4.2.4 Detaching C Extensions' References to Global Mutable State. In §3.5 we discussed challenges with C-modules' access to module state. We considered multiple solutions to this problem, including preventing C-modules from being frozen. Such approaches add significant complexity to the implementation not to mention problems for the Python programmer. To our delight, the detachment in §4.2.3 handles all of these cases in an intuitive and explainable way, while supporting immutability for most C modules *automatically*. What started as a convenience feature turned into an integral component to support the freezing of types and functions defined in C.

4.2.5 Detaching Fields. As §4.1.3 exemplified, cases exist where an immutable object need to retain some mutability.

A *shared field* lets an immutable object keep a mutable field but restricts the field to only contain immutable objects. The latter restriction is necessary as the shared field becomes a channel through which sub-interpreters can access the same data, and sub-interpreters cannot safely share mutable state (§3.6). The field is implemented as an object indirection with methods `get()`, `set(`*val*`)` and `swap(`*old*`,`*new*`)` that read from, write to, and CAS on the field.

```
class Die:
  _count = shared_field(0)
  def __init__(self, sides):
    ... # old code from Fig. 3
    while True:
      c = self._count.get()
      if self._count.swap(c, c + 1):
        break
```

Fig. 18. An example shared field usage.

If we implemented the `__subclasses__` list in a type object using a shared field, we would be restricted to only subclassing immutable types with immutable classes. Apart from being restrictive, the inheritance relation is established at the creation of the type object, at which point the type object is mutable and constructed. A good example of use for shared fields is a class that counts how many times it has been instantiated. Fig. 18 shows how we can extend the die class with such a counter. The `shared_field()` constructor takes an immutable argument as the initial value for the field and is implemented using locks to ensure that all interactions are thread-safe. Building on shared fields to make *e.g.*, thread-safe counters with `inc()` and `dec()` methods is straightforward.

Technically, we could implement atomic reference counting for immutable objects using shared fields, but this would not be sufficiently efficient. We return to this in §5.1.

4.2.6 Detaching Mutable Sub-Objects. Permitting immutable objects to reference mutable objects has use cases, as exemplified by the `__subclasses__` list above. We can enable that through *interpreter-local fields*, which work like thread-local indirections but at the sub-interpreter level. It effectively implements one field in the object *per sub-interpreter* that is only accessible to that sub-interpreter. Because stores to such a field will only be read from the same sub-interpreter such fields can contain both mutable and immutable objects. An interpreter-local field is initialised with an immutable object that will be the initial value for any thread, or by a lambda function which will be run once per sub-interpreter on first access to compute the initial value. The field's constructor will freeze the lambda to ensure that it does not capture mutable state from its creator. The interpreter-local field is implemented as an immutable object with two methods, `get()` and `set(`*value*`)` that return or update the object stored in the field for the current interpreter.

```
class type:
  # Each sub-interpreter has its own list
  __subclasses__ =
    local(lambda _ : [])
  # Called by the inheriting subclass
  def subclass(self, typ):
    # Add self to typ's base classes
    typ.__bases__ += (self,)
    sub = self.__subclasses__.get()
    sub.append(typ)
```

Fig. 19. An example interpreter-local field usage.

Fig. 19 shows a hypothetical implementation of the `__subclasses__` attribute using an interpreter-local field. Since the field is effectively private to a single sub-interpreter, which uses a GIL, there is no need for additional synchronisation and all objects in the list and the list itself can stay mutable. In this implementation, asking an immutable class for its subclasses may give different results on different sub-interpreters (which is consistent with mutable classes).

4.2.7 New Cyclic Garbage Challenges. A side-effect of "escape hatches", such as allowing mutable fields in immutable objects, is that it permits the creation of cycles in "immutable" objects after they were frozen. Such cycles will not be detected by our strongly connected component analysis (*cf.*, §5.2.3) and since immutable objects do not participate in cycle detection, such cycles can cause leaks. This can be solved by adding a stop-all-sub-interpreters GC event which is technically easy, and consistent with normal Python GC, including free-threading. Solving it without a global stop effectively needs concurrent GC. We leave this to future work and for now require programmers to manually break cycles involving escape hatches.

4.3 Freezability: Managing the Propagating Effects of Deep Dynamic Freezing

Freezing types and functions can have far-reaching consequences, even after detachment has been applied. To provide control to Python programmers, we introduce the notion of *freezability*. Objects can opt-out of freezing by setting `x.__freezable__ = No` which will cause `freeze()` to fail if called on `x` or propagating to `x`. We consider most objects to be freezable by default with a notable exception being type objects, which are unfreezable unless they have been explicitly marked as freezable by setting `x.__freezable__ = Yes`.

Types are special since they define the behaviour of objects in the form of fields and methods, and behaviour is a major factor in deciding if something can be frozen or not. Having a type unfreezable by default means that its instances are unfreezable until the type allows freezing. This is especially important for C-types, as incorrectly assuming that a C-type can be frozen and shared across sub-interpreters can lead to undefined behaviour, including crashes and memory corruption.

Subclasses and instances inherit freezability but can override it while keeping the base type freezable. Standard classes like `list`, `dict`, etc. are immutable and their instances freezable.

Not all objects allow the addition of attributes. The function `set_freezable(`*obj*`,` *val*`)` can set the attribute, falling back to storing the metadata outside of attributes, if setting the assignment fails.

The notion of freezability and ability to set it dynamically allows the construction of mechanisms like the one shown in Fig. 20. Using a context manager, it is possible to write code that temporarily prevents the freezing of an object, by setting and unsetting the `__freezable__` attribute. The `mutable` context manager has methods `__enter__()` and `__exit__()` called at the beginning and end of the "block" (Lines 2 and 4) that will set `x.__freezable__ = No` and then restore `x.__freezable__`.

```
1 x = [1,2,3]
2 with mutable(x):
3   func(x)
4 # x is guaranteed mutable
5 x.append(4)
```

Fig. 20. "Protecting" against freezing.

4.3.1 Explicit Propagation. We want propagation to make immutability work without calling `freeze()` on every single object in a graph. However, some objects may be freezable, but should not be frozen as part of propagation. This could be a type object or an object representing some configuration which supports immutability but should not be frozen accidentally by the programmer. *obj*`.__freezable__ = Explicit` supports this case. It allows calls to `freeze()` to succeed if *obj* is passed in *directly* but not if *obj* is frozen by propagation. Setting the value on the class will also require all instances to be explicitly frozen.

As an example, assume that the `Die` class from Fig. 3 is marked as explicitly freezable. To make the `d` instance immutable, we must explicitly freeze the type object first, or at the same time.

`freeze(Die)`$_1$; `freeze(d)`$_2$ *or* `freeze(Die, d)`$_3$ *or* `freeze(d, Die)`$_4$

The order between statements 1 and 2 matters, but not between arguments in statements 3 and 4. Forcing explicit freezing seems to strike a nice balance between being permissive by default and giving control to users (which is pythonic). Using explicit effectively requires that the programmer correctly declares the side effects regarding explicitly freezable objects as part of a `freeze()` call. If the programmer is misinformed about the reachability of such an object, freezing will fail.

4.3.2 Decorators: `@frozen`, `@freezable`, `@explicitly_freezable`, *and* `@unfreezable`. Python provides a decorator mechanism that can be used to write statically typed-*looking* code. A declaration

`@`*decorator* `class Foo: ...`$_1$

is semantically equivalent to (where the `del` operator removes the `_tmp` variable)

`class _tmp: ...`$_1$ `; Foo =` *decorator*`(_tmp)`$_2$ `; del(_tmp)`$_3$

To help programmers document immutability in source code we provide an `@frozen` decorator to use on class declarations to render type objects immutable immediately after creation. The `@frozen` decorator calls `set_freezable(`*class*`, Yes)` in addition to calling `freeze()`. The `@freezable`, `@unfreezable` and `@explicitly_freezable` decorators set `__freezable__` to `No`, `Yes`, or `Explicit` respectively, immediately after creation. The `@unfreezable` decorator should have been used in place of the comment on Line 1 in Fig. 9. Decorators allows freezing of types to be an "essential and explicit part of object design" (§2.1.3). *We expect that most Python classes can or will be* `@frozen`.

4.3.3 Immutable by Construction for C-Types. This section thus far has talked about freezability and freezing of mutable objects. §4.1.1 showed that some C-types, like tuples, are immutable by construction, and also introduced special handling for them. Shallow immutable objects which only contain other shallow immutable objects are implicitly frozen, when immutability is observed for the first time. This does not change the behaviour of these objects since they are already immutable by construction, but it allows them to be easily shared across sub-interpreters without the need of calling `freeze()` on them. We provide a `register_shallow_freezable(`*type*`)` function for C-modules which they can use to register such types. This status will propagate to instances of the type, but not to subclasses of the type.

4.3.4 Pre-Freeze Hook. The discussions about functions §3.3.2 and modules §3.4 show that some objects need to be prepared before they can be frozen properly. Our design adds support for this in the shape of a "pre-freeze hook". Objects can define a `__pre_freeze__()` method which is run *prior* to freezing an object. This hook can be used to optimise the internal representation or freeze explicitly freezable objects which are hidden from users of the type, as shown in Fig. 21. The pre-freeze hook can also abort freezing by throwing an exception.

```
1 class Wrapper:
2   ...
3   def __pre_freeze__(self):
4     # Ensure explicitly freezable
5     # data is frozen.
6     freeze(
7       self.explicit_data)
```

Fig. 21. An example of pre-freeze hook usage.

5 Implementation

Our implementation is on top of CPython 3.15.0a1 (sha `aeff92d8`) adding 6715 LOC and removing 230 LOC across 185 files. Below we describe how we implement safe memory management, how we track, check and set immutability and our changes to support direct sharing across sub-interpreters.

5.1 Thread-Safe Memory Management without a GIL

To ensure that reference count updates from different sub-interpreters are handled correctly, we make reference count manipulations on immutable objects atomic.

Mutable objects are unaffected by this change. This is less expensive than the solution adopted in Free-threaded Python: since the objects are immutable there is no need for atomic exchanges or per-object locks when accessing objects' contents. Thus, Free-threaded Python could use this work to optimise interactions with immutable objects. Atomic reference counting is implemented by inserting an additional branch in the existing reference counting macros that uses atomic operations if the target object is immutable. There is already a slow path in the reference counting macros that we can use for this operation which should minimise the performance impact. Together, this removes all needs for concurrent accesses to synchronise using a GIL. The implementation of

`sys.getrefcount()` that inspects the reference count of an object will use an atomic load when reading the reference count of an immutable object.

To overcome the problems related to cycle detection, we *remove* immutable objects from the remit of their owning sub-interpreter's garbage collector. Instead we integrate a technique proposed by Parkinson et al. [17] that performs a strongly connected component analysis of immutable data structures with the insight that all members of such immutable components will become garbage at exactly the same time. They can thus be managed by a single shared reference count. This allows us to collect cyclic immutable garbage without tracing, and without losing promptness of reclamation (actually, this solution has less floating garbage than tracing-based solutions), meeting (G6).

Once an immutable object becomes garbage, we insert the object back into the doubly linked list of any sub-interpreter's collector. This allows us to use the normal dealloc mechanism and lets the interpreter handle, *e.g.*, resurrection, in accordance with (G7). We expand on this further in §5.2.3, including how we have to adapt this work to our setting.

5.2 Tracking, Checking, and Setting Immutability

5.2.1 Tracking. To keep the object size invariant, we use two available bits in the object header: one to track immutability and a second to track whether an immutable object is a representative for a strongly connected component of immutable objects (*cf.*, *Setting* below).

To not impact performance of programs that do not make use of immutability, we do *not* inspect the sub-objects when a shallow immutable object is created to check if we should set the deep immutability bit. Instead we delay that check until `is_frozen()` or `freeze()` are called on the object. Following a `freeze()` call, `is_frozen()` is a constant-time operation.

5.2.2 Checking. To check immutability we use a write barrier macro: `Py_CHECKWRITE`. We manually added calls to the macro since there is no existing write barrier infrastructure. The macro inspects the immutability bit and fails if the bit is set. The failing function needs to emit an error using `PyErr_WriteToImmutable` (which we implement) and back out correctly, which means returning `-1` or `NULL` depending on the function (as C lacks an exception handling mechanism).

In total we added 148 checks to Python core. Immutability is checked as part of CPython's API before the set attribute request is passed to the object. This covers all Python code and attributes of C-types. C-modules which want to support immutability for their types therefore only need to insert checks at places that alter the state of their C-type without going through CPython's API.

5.2.3 Setting. The `freeze()` function sets immutability of an object graph given a set of roots. The basic concept is this: "*Walk the object graph, for each object call the pre-freeze hook followed by marking the object as immutable, and fail if an unfreezable object is found.*" In reality there are several complexities that need to be handled to protect the interpreter and fulfil our design goals.

The first challenge is that most CPython objects are part of a doubly linked list used by the GC. Each interpreter has its own GC that runs periodically to detect and break unreachable reference cycles, thereby deallocating unused objects.

Immutable objects have to be removed from this linked-list because the existing GC needs the GIL and does not support concurrent access or modification of the linked-list, that might happen if an immutable object is deallocated. The GC also cannot reason about reachability of immutable objects, since other sub-interpreters might have a reference to the immutable object. We therefore remove objects from this list as part of the freezing process.

Removing objects from this list means that we need an alternative way to detect cycles in the shared immutable object graphs. We adapt Parkinson et al.'s [17] algorithm for reference counting deeply immutable state. The algorithm is based on calculating the strongly connected components (SCC) of the object graph. It uses a path-based algorithm to calculate the SCCs. The reference

counting is then adapted to use a single shared reference count for each SCC rather than per-object reference counts. This allows cycles of immutable objects to be correctly reference counted without the need for backup cycle detection.

We adapted the algorithm in several ways for CPython. The original algorithm assumes no external references to the graph being frozen. We cannot make that assumption. Second, the original algorithm did not account for failure during the freezing process which may happen due to unfreezable objects (§4.3). It is common practice for failed operations to not have side effects, so we need to be able to undo the freezing process. Note that actions of pre-freeze hooks are not undone.

The algorithm follows the same structure as Parkinson et al.'s [17]. It performs a path-based SCC calculation of the object graph. Any SCC in the current path is considered *pending*. Any edge leading into the current pending path represents a cycle and must combine all the *pending* SCCs on the path into a single SCC. Once an SCC is fully explored, it is marked *frozen*.

The original algorithm calculated reference counts by finding all edges into each SCC and using that to calculate the reference count. In CPython, we cannot do this as we have to handle external references into this graph. Rather than counting up the incoming edges, we subtract the internal edges from the SCCs reference counts. Thus, the reference count of an SCC is the sum of the original reference counts of the objects in the SCC, minus the internal references in the SCC.

We repurpose the two fields in the cycle detector's doubly linked list during the freezing process. One is used to implement a union-find data structure to track if two objects are in the same pending SCC, and the other is used to implement a cyclic list of all the elements in a particular SCC. This allows the union of pending SCCs in almost linear time. Once an SCC is marked frozen, we no longer require the union-find pointer in the representative, so we repurpose this to create a list of all the SCCs during the freezing process. This list is used to undo the freezing process if it fails.

Deallocation of an SCC has to correctly interact with the various schemes of finalisation that Python has, such as an object being revived if it becomes reachable again. The cyclic list of an SCC is used to find all objects in the SCC which are then re-added to the linked list of the current sub-interpreter. They are then deallocated using existing CPython mechanisms.

Many objects mutate internal state during finalisation, which would trigger exceptions in the case of frozen objects. We avoid this, by marking objects as mutable again during finalisation. This is safe since these objects are no longer reachable and will be destroyed anyway.

5.3 Direct Object Sharing with Sub-Interpreters

The described design and implementation is set up to allow sharing across sub-interpreter with separate GILs. Immutability removes the need for synchronisation with a notable exception being the reference count which is updated atomically to ensure correctness, *cf.*, §3.6.

CPython already provides a way to send and receive objects between sub-interpreters. By default this is done by serialising, sending the data, and then deserialising the data back into Python objects[3] [21]. Our implementation adds an extra branch to the sending code. If the sent object is immutable it is sent directly by reference, thereby avoiding the overhead of serialisation.

6 Evaluation

6.1 Functional Goals

The immutability design presented in this paper makes possible direct sharing of objects across sub-interpreters (G1). The final benchmark program we run demonstrates the relative efficiency of sharing frozen objects by reference over pickling-based sharing. We weakly demonstrate that

[3]Sub-interpreters use the `Lib/pickle.py` module for serialisation. The module calls the serialisation process *pickling* and *unpickling* respectively. These are the terms used in the documentation of sub-interpreters.

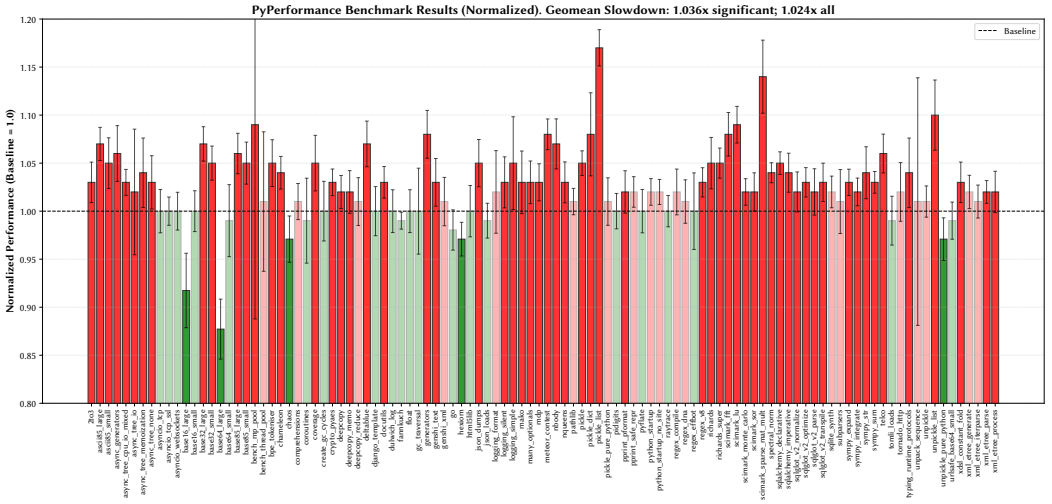


Fig. 22. PyPerformance benchmark results. Insignificant results are shaded light.

we do not break existing code (G2) by successfully running `make test` on our Linux test machine, an AMD Ryzen Threadripper 3970X (32 cores / 64 threads) with turbo boost enabled, 128 MiB L3 cache, and 256 GB ram running Ubuntu 22.04.5 LTS.

Before immutability lands (we hope) in the CPython main branch, it will not be possible to conclusively tell if our design provides sufficient support for *e.g.*, metaprogramming and inherent mutability (G3). We leave answering this question for future work. Finally, we embracing dynamic checking—in particular the only errors raised in our code are right before a violation of immutability would otherwise have occurred (G4). Tracking immutability in Python's type system is left as future work.

Test Compliance. We ran the full CPython test suite with our changes on top of it. The test suite contains 1126 Python files containing a total of 629 132 LOC. Our build ran *all* the passing tests of the baseline, suppressing one error due to missing documentation of the immutable module.

6.2 Non-Functional Goals

Our performance goal (G5) was to have "minimum performance impact on programs that either do not use immutability or use it without raising dynamic errors".

6.2.1 Running the PyPerformance Suite. The PyPerformance tests are a collection of microbenchmarks used by among others the Faster CPython project to monitor how the performance of Python changes over time, and to spot regressions or improvements. We ran all 106 microbenchmarks in the benchmark suite that work on CPython 3.15.0a1, our baseline, and show the results in Fig. 22. The suite performs its own statistical analysis, comparing repeated timings with a two-sample t-test, which concluded that 61 of the 106 microbenchmarks had a statistically significant *regression* and 5 had a statistically significant *improvement*. The worst regressions were 17 % (`pickle_list`) and 14 % (`scimark_sparse_mat_mult`), and the largest improvements was 16 % (`base64_large`). The geometric mean of the speedups and slowdowns across these 66 benchmarks is a 3.6 % *slowdown*.

We consider a 3.6 % slowdown, as indicated by the benchmark results, strong given that we have added a write-barrier and not yet optimised any of our code.

6.2.2 Freezing vs. Pickling and Unpickling. To understand the performance of our freeze function (which has yet to see any optimisation work), we compare the time required to successfully freeze data structures (lists, trees, dictionaries) with the time required to pickle and unpickle.

Our benchmark generates 1 000 000 strings with eight characters and inserts them into the respective data structures. We have two tests for dictionaries, both using random strings as keys, with one mapping to integers (dict-int) and the other mapping to user-defined student objects with a name and age field (dict-student). We then measured the time to freeze, pickle and unpickle the data structures over 10 runs.

The results in Table 2 show that freezing is always faster than pickling and unpickling *combined*, always faster than pickling, and sometimes slower than unpickling.

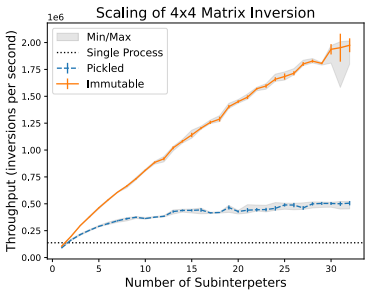
Table 2. Comparing time to freeze (F) with pickle (P) and unpickle (U) in ms.

Experiment	Op	Mean	GeoM.	StdDev
dict-int	F	346.23 (1.9×)	346.21	4.56
dict-int	P	177.64	177.55	6.15
dict-int	U	275.41 (1.6×)	275.25	9.93
dict-student	F	661.35	661.33	5.67
dict-student	P	1703.00 (2.6×)	1702.98	7.36
dict-student	U	1070.02 (1.6×)	1070.01	5.52
tuple	F	149.81 (2.7×)	149.78	3.50
tuple	P	153.28 (2.8×)	153.27	2.02
tuple	U	55.54	55.53	0.35
binary-tree	F	800.43	800.23	19.17
binary-tree	P	1940.95 (2.4×)	1940.93	9.61
binary-tree	U	924.78 (1.2×)	924.74	9.02

6.2.3 Direct Sharing Across Sub-interpreters. To demonstrate sharing across sub-interpreters, and what kinds of performance can be expected when using direct object sharing in comparison with pickling, we built a small benchmark that constructs (per worker) 200 000 4×4 matrices of random values, implemented as a user-defined Matrix class. A producer-interpreter shares lists of matrices with consumer-interpreters that process them and return how many of the matrices were invertible back to the producer. Thus, this benchmark mixes local compute with communication. Our goal is to reduce the overhead of communication by eliminating pickling. Thus, if communication takes C % of the time, we should expect less than C % improvement for the entire benchmark. Note that we do not time freezing itself since the "return on investment" in freezing will be dependent on object lifetimes. (Table 2 shows that freezing is faster than pickling and unpickling.)

Communication between sub-interpreters is done via a Python C extension we implemented which exposes Erlang-like send and receive functions, which allows selective receive on a lockless message queue. The Pickled line shows the throughput of Python 3.15 when matrices are not frozen, in which case the data must be pickled to be sent, and unpickled upon delivery. The line marked Immutable represents the same operation but with a frozen Matrix class and frozen Matrix instances.

The code running on consumer-interpreters is shown right in Fig. 23. Mean values over ten trials are shown. One standard deviation is depicted as a vertical line. The throughput of a single process for the task is shown for comparison. At the peak 32 sub-interpreter mark, sharing is around 4× faster than sending using pickling. *This demonstrates that data-race free, direct sharing of immutable objects has the potential to improve Python's performance.*




```
running = True
matrix_inverse = Matrix()
send("started", True)
tags = set(["worker", "shutdown"])
while running:
  match receive(tags):
    case ["worker", values]:
      # values is immutable
      count = 0
      for matrix in values:
        if matrix.invert(matrix_inverse):
          count += 1

      send("result", count)

    case ["shutdown", _]:
      running = False
```

Fig. 23. Effect on performance from direct sharing of immutable objects.

6.2.4 Promptness of Reclamation. Regarding promptness of reclamation (G6), we provide the following argument: our implementation detects that an immutable object has become garbage immediately when its RC reaches 0—even cycles. At this point, the object is given to a sub-interpreter to deallocate. Thus, immutable objects are deallocated more promptly than mutable objects.

6.2.5 Staying within a Reasonable Complexity Budget. With respect to the (G7), we believe that we have consistently made design decisions in favour of keeping complexity down. The most complex part if this implementation is the freeze propagation §4.3 and SCC construction §5, which is contained in one location. The 148 write barriers add some complexity to existing types. However, it is usually easy to pin-point where an object is mutated (indicated by an assignment).

7 Conclusion

This work set out to make deep immutability practical in Python, a language whose highly dynamic object model, pervasive mutability, and reflective facilities appear to be fundamentally at odds with the notion of freezing arbitrary object graphs. Our implementation demonstrates that deep immutability can be integrated into Python without breaking its programming idioms, while enabling efficient and race-free sharing of data across sub-interpreters. It is backwards-compatible with existing Python programs, and passes all tests on our Linux test machine.

A broader lesson emerged through the design process: dynamic languages require two complementary principles to support deep immutability—*detachment and freezability*. These principles are not Python-specific; they describe design constraints that likely arise in any dynamic, reflective run-time. *Detachment* addresses the structural problem: immutability cannot propagate blindly across a graph whose edges arise from dynamic name lookup, captured environments, mutable modules, or type-level metadata. Python's execution model contains many such links, and freezing an object naively would cause "everything reachable" to become immutable. The repeated need to sever these links—between functions and their global namespaces, closures and their cells, modules and their mutable state, and between immutable parents and mutable children—revealed detachment as a general mechanism. Detachment reshapes the reachable graph around a frozen object so that immutability stays where it is intended rather than where the run-time happens to point. We believe this principle applies to any dynamic language with late binding, mutable meta-objects, or reflective scope resolution. *Freezability*, in contrast, solves the policy problem: without a static type system, a dynamic language requires a run-time contract governing whether an object may become immutable at all. Freezability provides that contract. Declaring a type or object as freezable permits it to evolve, including through metaprogramming, up to the moment it is frozen; declaring it unfreezable prevents accidental propagation when freezing other parts of the system; and declaring it explicitly freezable ensures that only intentional, programmer-directed freezes succeed. Freezability therefore replaces the compile-time discipline of typed languages with a run-time permission system that is equally expressive but much better aligned with dynamic semantics.

Taken together, detachment and freezability provide a transferable framework for introducing deep immutability into dynamic languages. Detachment isolates what must be immutable; freezability grants programmers control over when immutability is allowed. Our experience with Python suggests that these two principles are not optional conveniences, but necessary ingredients for a dynamic language that aims to support safe, shareable immutable data without compromising its design philosophy.

Acknowledgments

David Klement contributed the code for weak reference handling. We would like to thank the CPython community for discussing this proposal, and particularly Guido van Rossum, Brandt Bucher, Eric Snow, Michael Droettboom, Irit Katriel, and Mark Shannon. Finally, we thank the anonymous PLDI reviewers for their constructive feedback that vastly improved the paper.

This research was supported by the Swedish Research Council through the grant 2024-04565, "Data-race Freedom and Memory Safety for Untyped Languages".

Data-Availability Statement

Our artefact [22] consists of a docker container providing a local interactive website where users can write and run Python programs using our patched CPython codebase with immutability, and run all the tests and benchmarks that are in §6. The artefact also contains all source code with explanations and pointers to interesting parts of the implementation.

References

[1] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 270 (oct 2023), 31 pages. doi:10.1145/3622846

[2] Henry G. Baker. 1994. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *SIGPLAN Not.* 29, 9 (Sept. 1994), 38–43. doi:10.1145/185009.185016

[3] Adrian Birka and Michael D. Ernst. 2004. A practical type system and language for reference immutability. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, BC, Canada) *(OOPSLA '04)*. Association for Computing Machinery, New York, NY, USA, 35–49. doi:10.1145/1028976.1028980

[4] Chandrasekhar Boyapati. 2003. *Safejava: A Unified Type System for Safe Programming.* PhD Thesis. Massachusetts Institute of Technology.

[5] Chandrasekhar Boyapati and Martin Rinard. 2001. A Parameterized Type System for Race-Free Java Programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object -Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 56–69. doi:10.1145/504282.504287 event-place: Tampa Bay, FL, USA.

[6] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing. In *Proceedings of the 15th European Conference on Object- Oriented Programming ECOOP, Budapest, Hungary, June 18-22, 2001*. Springer Berlin Heidelberg, 2–27. doi:10.1007/3-540-45337-7_2

[7] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. 1987. Clean — A language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–384.

[8] Luke Cheeseman, Matthew J. Parkinson, Sylvan Clebsch, Marios Kogias, Sophia Drossopoulou, David Chisnall, Tobias Wrigstad, and Paul Liétar. 2023. When Concurrency Matters: Behaviour-Oriented Concurrency. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 276 (oct 2023), 30 pages. doi:10.1145/3622852

[9] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. 2008. Minimal Ownership for Active Objects. In *Programming Languages and Systems*, G. Ramalingam (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 139–154.

[10] Sylvan Clebsch. 2017. *Pony: Co-designing a Type System and a Runtime.* PhD Thesis. Imperial College London.

[11] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. 2016. Exploring language support for immutability. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 736–747. doi:10.1145/2884781.2884798

[12] Sam Gross. 2023. PEP 703 – Making the Global Interpreter Lock Optional in CPython. https://peps.python.org/pep-0703/.

[13] Raymond Hettinger. 2006. PEP 351 (Python-Dev mailing list). https://mail.python.org/pipermail/python-dev/2006-February/060793.html.

[14] Günter Kniesel-Wünsche and Dirk Theisen. 2001. JAC - Access right based encapsulation for Java. *Softw., Pract. Exper.* 31 (05 2001), 555–576. doi:10.1002/spe.372

[15] James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP*. 158–185.

[16] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. 2008. Ownership, uniqueness, and immutability. In *International Conference on Objects, Components, Models and Patterns*. Springer, Berlin, Heidelberg, 178–197.

[17] Matthew J. Parkinson, Sylvan Clebsch, and Tobias Wrigstad. 2024. Reference Counting Deeply Immutable Data Structures with Cycles: An Intellectual Abstract. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management* (Copenhagen, Denmark) *(ISMM 2024)*. Association for Computing Machinery, New York, NY, USA, 131–141. doi:10.1145/3652024.3665507

[18] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. 2013. Immutability. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 233–269. doi:10.1007/978-3-642-36946-9_9

[19] Eric Snow. 2022. PEP 684 – A Per-Interpreter GIL. https://peps.python.org/pep-0684/.

[20] Eric Snow. 2023. PEP 554 – Multiple Interpreters in the Stdlib. https://peps.python.org/pep-0554/.

[21] Eric Snow. 2023. PEP 734 – Multiple Interpreters in the Stdlib (supersedes [20]). https://peps.python.org/pep-0734/.
[22] Fridtjof Stoldt, Sylvan Clebsch, Matthew A. Johnson, Matthew J. Parkinson, and Tobias Wrigstad. 2026. *Artifact: Dynamically Checked Deep Immutability in Python.* doi:10.5281/zenodo.19486173
[23] Fridtjof Peer Stoldt, Brandt Bucher, Sylvan Clebsch, Matthew A. Johnson, Matthew J. Parkinson, Guido Van Rossum, Eric Snow, and Tobias Wrigstad. 2025. Dynamic Region Ownership for Concurrency Safety. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1565—-1590. doi:10.1145/3729313
[24] Michał Szołucha. 2024. Freezing data type for parallel access. https://discuss.python.org/t/freezing-data-type-for-parallel-access/62398/6.
[25] Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: adding reference immutability to Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 211–230. doi:10.1145/1094811.1094828
[26] Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: Adding Reference Immutability to Java. *SIGPLAN Not.* 40, 10 (oct 2005), 211–230. doi:10.1145/1103845.1094828
[27] Martin von Löwis. 2007. PEP 3121 – Extension Module Initialization and Finalization. https://peps.python.org/pep-3121/.
[28] Barry Warsaw. 2005. PEP 351 – The freeze protocol. https://peps.python.org/pep-0351/.

Received 2025-11-14; accepted 2026-04-03