The Python Architecture: Why It Works So Elegantly

From C-structs to elegant protocols in high-performance DSP.

Updated Oct 31, 2026 · @Roee Ronkin

Contents

1. The Core Philosophy: Fixed Syntax, Pluggable Behavior 2
2. Under the Hood: The C-Struct Architecture 3
3. The Grand Unification: Two Pillars and a Keystone 4
4. Deep Dive: First-Class Objects 8
5. Conceptual Paradigm: Python as a Plugin Architecture 9
6. The Data Model, Protocols and Dunder Methods 10
7. Python as an ICD: The Client/Supplier Dynamic 13
8. The Golden Rule of Dunders 13
9. Practical Synthesis: The MATLAB System Object in Python 13
10. IDE Integration and Essential Protocols: The Cheat Sheet 18
11. Performance Reality: Python vs. MATLAB 19
12. What the Design Costs 21
13. Epilogue: Why “Data Model”? 21

Much of Python's elegance comes from one design decision. The syntax is fixed, but what the syntax *does* is decided by your objects. This guide follows that decision and how CPython implements it, for engineers coming to Python from MATLAB, C and DSP work. Section 12 covers what the design costs.

The code examples assume `import numpy as np`. Where they use `time` or CuPy, they also assume `import time` and `import cupy as cp`.

Found a mistake? Open an issue.

1. The Core Philosophy: Fixed Syntax, Pluggable Behavior

Python's keywords and operators are hardcoded in the grammar, like any language. But most of that syntax is defined as a call to a named method on the object. These methods are the "dunder" (Double UNDERscore) methods. A set of dunders that together give an object a capability is called a **protocol**.

If you know operator overloading from C++ or MATLAB, think of dunder methods as that same idea extended beyond operators. Attribute access, `for` loops, `with` blocks and even class creation all call methods you can define.

Python did not invent this. C++ and C# overload operators, and Ruby and Smalltalk go further by treating nearly every operation as a method call. Python's strength is how much of the language it brings under one documented set of protocols, and how little a class must declare to join in.

So when the language needed to grow, its designers rarely added special cases. They added protocols.

- **The `for` loop:** Python does not loop over arrays specifically. It says: "If an object has `__iter__()`, and the iterator it returns has `__next__()`, then `for` works on it." An **iterator** is an object that hands out items one at a time. Each call to `__next__()` returns the next item, and raising `StopIteration` signals the end.
- **Context managers:** Python does not add a special statement for each kind of resource, such as files, locks or devices. It says: "If an object has `__enter__()` and `__exit__()`, it works with `with`."
- **Arithmetic:** `a * b` is not a fixed machine operation. It is a request to `a.__mul__(b)`, with a defined fallback to `b.__rmul__(a)`.

Example: two methods, eight features

```
class Ring:
    def __init__(self, data):
        self.data = list(data)
    def __len__(self):
        return len(self.data)
    def __getitem__(self, i):
        return self.data[i]

r = Ring([3, 1, 2])
len(r)              # 3
r[0], r[0:2]        # 3, [3, 1]
2 in r              # True
for x in r: ...     # works
list(reversed(r))   # [2, 1, 3]
sorted(r), max(r)   # [1, 2, 3], 3
```

`Ring` never mentions `for`, `in`, `reversed`, `sorted` or `max`. It implements the sequence protocol, and the language supplies the rest. Even `for` works without `__iter__`: Python falls back to calling `__getitem__` with 0, 1, 2, … until it raises `IndexError`.

Decorators are a different mechanism. They are not a protocol. They are syntax for passing a function object into another function, which is possible because functions are ordinary objects (Section 4).

The trade-off is **dynamic dispatch overhead**. Every `+` or `[]` has to find and call the right method at runtime, where compiled C often needs a single machine instruction. In return, a custom hardware-simulation block can behave exactly like a native Python type.

2. Under the Hood: The C-Struct Architecture

In CPython (the reference interpreter), there are no raw primitives. Every value lives on the heap as a C struct that begins with a common header, `PyObject`. A Python name is just a reference to such a struct, which in C is a `PyObject*` pointer.

```
typedef struct _object {
    Py_ssize_t ob_refcnt;          // Reference count
    struct _typeobject *ob_type;   // Pointer to the object's type
} PyObject;
```

This is the layout of the default build. The newer free-threaded build (3.13+) adds fields for thread safety, but the idea is the same.

Memory management. `ob_refcnt` counts how many references point to the object. When it drops to zero, the object is freed immediately. Reference counting cannot free objects that refer to each other in a loop. A separate cyclic garbage collector runs periodically to clean those up.

Struct inheritance by layout. `PyObject` is a header stamped on the front of every allocation. Each concrete type appends its own fields after it.

- **Built-ins:** A float stores its C `double` right after the header. A NumPy array stores a pointer to its data buffer, plus shape and strides.
- **Custom classes:** Instance attributes live in a per-object namespace, which is a table that maps names to objects (Section 3). Classically this is a `__dict__` hash table whose location is recorded in the type. Since 3.11, CPython often stores attributes inline in a compact form and only builds a real dict when needed.

Why this matters. Because every object starts with the same header, C code can handle any object through a single pointer type. That is why a `list` can hold an integer, a CuPy array (a NumPy-style array stored on a GPU; Section 11), and a custom delay module side by side. The list simply stores an array of `PyObject*` pointers. The `ob_type` pointer then tells the interpreter where to find each object's behavior.

3. The Grand Unification: Two Pillars and a Keystone

Strip away the syntax, libraries and ecosystem, and Python rests on two pillars. A third mechanism, descriptors, is the keystone that joins them. Namespaces underlie all of it. These mechanisms are defined in terms of each other, so some terms appear before their full explanation. This circularity is part of why the design is so compact: a few ideas, each built from the others.

Pillar 1: The Data Model (Protocols and Dunders) → Behavior

This is the structural contract. It defines how objects respond to native syntax: operators, indexing, iteration, calls, `with`, `len()`, and so on. It is the main way Python delivers object-oriented behavior.

Pillar 2: First-Class Objects → Flexibility

A value is first-class when it can be used like any ordinary value, such as an integer: stored, passed, returned and created at runtime. In Python, functions, classes and modules are all first-class objects. Section 4 explores this in depth.

Decorators are a direct consequence of this pillar. `@njit` above a DSP function means `f = njit(f)`. The function object is passed to Numba, a just-in-time (JIT) compiler that turns a Python function into machine code the first time it runs (Section 11). Numba returns a new callable object that dispatches to compiled machine code.

The Keystone: Descriptors

Descriptors solve a practical problem that starts with ordinary attributes. Assignment creates an attribute on one object:

```
class Car:
    pass

c = Car()
c.color = "red"   # only this object has color
```

To give every object the attribute, set it in `__init__`. Each object still holds its own value:

```
class Car:
    def __init__(self, wheels=4):
        self.wheels = wheels

c = Car()
c.wheels = -3     # accepted: none of your code runs
```

To run code on access, `property` handles a single attribute:

```
class Car:
    def __init__(self, wheels=4):
        self.wheels = wheels      # goes through the setter

    @property
    def wheels(self):
        return self._wheels

    @wheels.setter
    def wheels(self, value):
        if value < 0:
            raise ValueError("wheels must be non-negative")
        self._wheels = value
```

If `doors` and `seats` need the same rule, this getter and setter must be repeated for each one. A descriptor holds the logic once, so any attribute of any class can reuse it.

A **descriptor** is an object whose class defines `__get__`, `__set__` or `__delete__`. It takes effect only when it is stored as an attribute of another class. When you access that attribute, Python finds the descriptor on the class and lets it decide what happens.

```
class NonNegative:
    def __set_name__(self, owner, name):   # called once, when the class is built
        self.private = "_" + name

    def __get__(self, instance, owner=None):
        if instance is None:               # accessed on the class itself
            return self
        return getattr(instance, self.private)

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError(f"{self.private[1:]} must be non-negative")
        setattr(instance, self.private, value)

class Car:
    wheels = NonNegative()
    doors = NonNegative()

    def __init__(self, wheels=4, doors=4):
        self.wheels = wheels               # goes through __set__
        self.doors = doors

c1 = Car(wheels=4)
c2 = Car(wheels=6)
print(c1.wheels, c2.wheels)   # 4 6
c1.wheels = -3                # ValueError
```

`__get__` and `__set__` receive both `self` and `instance` because of where the descriptor lives. `Car` holds one `NonNegative` object for `wheels`, and every car shares it. `self` is that shared descriptor, and `instance` tells it which car is being accessed.

The logic therefore belongs on the descriptor and the data belongs on the instance. Storing the value on `self` is the classic bug:

```
class NonNegative:
    def __get__(self, instance, owner=None):
        return self.value            # bug: one value for all instances

    def __set__(self, instance, value):
        self.value = value           # bug: every car overwrites it

class Car:
    wheels = NonNegative()

c1 = Car()
c2 = Car()
c1.wheels = 4
c2.wheels = 6
print(c1.wheels)                     # 6, not 4
```

A descriptor controls access through one name, but it does not make the data private. `c1._wheels = -3` skips the check in the working version. Python has no enforced equivalent of MATLAB's `Access = private` or `Access = protected`. A leading underscore is only a convention

that says “internal, don’t touch”. A leading double underscore, as in `self.__x`, triggers **name mangling**: Python renames the attribute to `_Car__x`. This prevents accidental clashes with subclass attributes. It is still not access control, because `c._Car__x` works.

This is where the two pillars meet. Functions are first-class objects, and every function object has a `__get__` method. That single fact is how methods work:

```
class Gain:
    def apply(self, x):
        return 2 * x

g = Gain()
g.apply   # Python runs Gain.__dict__['apply'].__get__(g, Gain)
          # -> a "bound method" with self already filled in as g
```

Two classes are involved. The function type defines `__get__`. The function object `apply`, stored in `Gain`, is the descriptor.

There is no special “method” machinery in the language. `self` is filled in by a protocol. The same mechanism implements several built-in tools:

- `property` turns a method into an attribute that is computed when read, like a getter.
- `classmethod` binds a method to the class instead of an instance. It is often used for alternate constructors, such as `Signal.from_file(path)`.
- `staticmethod` turns binding off, so the method is a plain function stored on the class.
- `__slots__` replaces the per-object attribute dict with fixed storage (Section 10).

Section 9 uses a descriptor to build MATLAB-style nontunable properties in about a dozen lines. A nontunable property is one that is locked once the object starts running.

Underneath It All: Namespaces

Module globals, class bodies and ordinary instances store their names in dictionaries. You can see them with `vars(obj)`, and you can change them at runtime. For a class, `vars()` returns a read-only view, so write with `setattr(cls, name, value)` instead. This is what makes Python so malleable: a name is a string key that points to a first-class object.

```
globals()["gain"] = 4.0   # write straight into the module's dict
print(gain)               # 4.0: the name now exists
```

Function locals are the deliberate exception. The compiler resolves local names to fixed slots in an array, so local access is a fast index instead of a hash lookup. `locals()` inside a function returns a snapshot, not the real storage. This is also why a hot loop runs faster when it reads a local variable than a global one.

```
def f():
    x = 1
    locals()["x"] = 99    # writes into a snapshot
    return x              # 1: the real slot is unchanged
```

4. Deep Dive: First-Class Objects

What “first-class” means

An entity is **first-class** if it can be treated like any ordinary value, such as an integer. It must be possible to:

1. Assign it to a variable or store it in a data structure.
2. Pass it as an argument to a function.
3. Return it from a function.
4. Create it at runtime.

The simplest case is a dispatch table:

```
filters = {"lpf": lpf, "hpf": hpf}   # functions stored in a dict
y = filters[mode](x)                 # one chosen and called at runtime
```

Many modern languages now have first-class *functions*. In Python, functions, classes, modules, methods and types are all first-class, and they are all the same kind of thing: a `PyObject`. Python is not alone here. Smalltalk, Lisp, Ruby and JavaScript share this property. It is unusual among the languages DSP engineers typically use: C, C++, Java and MATLAB.

What sets Python apart from C, C++ and Java: `def` and `class` are executable statements

In Python, `def` and `class` are not declarations read by a compiler. They are statements that *run*, and running them builds a new object at that moment.

```
def make_fir(taps):
    def fir(x):                  # runs on every call: a new function object each time
        return np.convolve(x, taps, mode="same")
    return fir

lpf = make_fir(np.ones(8) / 8)   # a filter built at runtime

# `class` is shorthand for calling type(), which is itself a class
Block = type("Block", (), {"gain": 2.0})
```

The inner `fir` is a **closure**: a function that remembers variables (`taps`) from the scope where it was created.

The contrast

Language	Functions as values	Classes as runtime values
C	Function pointers only. No closures, no runtime creation.	No classes.
C++	Yes, since C++11: lambdas with captures and `std::function`.	No. Classes and templates exist only at compile time.

Language	Functions as values	Classes as runtime values
Java	Yes, since Java 8: lambdas and method references. Before that, anonymous wrapper classes.	Partly. Class objects exist via reflection (a program's ability to inspect its own classes at runtime), but classes are loaded from compiled code, not built by running a statement.
Python	Yes. Functions are full objects you can attach attributes to.	Yes. Classes are built by running code and can be created, passed and modified at runtime.

In almost all languages, control flow such as `if`, `while` and `for` is not first-class. Python is no exception. What Python does instead is let objects decide how `for` and `with` behave, which is the subject of the next section.

5. Conceptual Paradigm: Python as a Plugin Architecture

The Data Model turns the interpreter into a host application with fixed extension points. Native syntax is the set of features the host provides. Dunder methods are the sockets you plug into.

- Implementing `__getitem__` plugs your object into the bracket `[]` syntax.
- Implementing `__enter__` and `__exit__` plugs it into the `with` resource lifecycle.
- Implementing `__iter__` plugs it into `for`, unpacking, `list()`, `sum()` and every other consumer of iterables.

The sockets are on the class, not the object

Python looks up dunder methods on the object's **type**. It skips the instance's own attributes. Attaching a dunder to one object does nothing:

```
class Buf: pass
b = Buf()
b.__len__ = lambda: 4
len(b)                          # TypeError: object of type 'Buf' has no len()

Buf.__len__ = lambda self: 4
len(b)                          # 4
```

This is deliberate. The interpreter can cache each type's sockets in fixed C slots on the type struct (`ob_type` from Section 2). So `len(b)` goes straight to a C function pointer on the type and never searches the instance's dict. For a type written in C, that pointer is the implementation. For a class written in Python, it is a small wrapper that finds and calls your `__len__`, which is slower. Plugin flexibility and fast dispatch for built-in types come from the same design.

Implicit registration via duck typing

Duck typing means judging an object only by the methods it has. Its class and ancestry are ignored. The name comes from the saying "if it walks like a duck and quacks like a duck, it is a duck."

In most plugin systems, a plugin must explicitly register itself. It imports an SDK base class or edits a configuration file. Python skips this step. The interpreter does not care what your class inherits from. If the right dunder exists on the type, the plugin works.

The standard library takes this further. An **abstract base class** (ABC) is a class that defines an interface and is meant only to be inherited from. The single-method ABCs in `collections.abc`, such as `Iterable`, `Sized`, `Container` and `Callable`, recognize objects by their methods alone:

```
from collections.abc import Iterable

class Stream:
    def __iter__(self): ...

isinstance(Stream(), Iterable)  # True, with no inheritance
```

The result: your custom blocks extend the native vocabulary of the language itself.

6. The Data Model, Protocols and Dunder Methods

Python separates syntax from implementation. The Data Model is the API of the language itself.

- **Dunder methods (the hooks):** Reserved names such as `__init__`, `__add__` and `__len__`. When Python sees native syntax like `+`, it asks the object's type: "Do you have the method mapped to this syntax?"
- **Protocols (implicit interfaces):** An informal agreement. Implement the right set of dunders, and Python treats your object like a built-in type.

Core protocols in practice

- **Callable (`__call__`):** Use the object as a function. `filtered = my_filter(raw)`.
- **Container (`__len__`, `__getitem__`):** Index and slice custom buffers like native arrays. `sample = buffer[1]`.
- **Numeric (`__add__`, `__mul__`, ...):** Use math operators on your own signal or gain objects. `mixed = sig_1 * sig_2`.
- **Context manager (`__enter__`, `__exit__`):** Guarantee teardown when a block exits, even on an error.

The smallest useful context manager is a timer:

```
class Timer:
    def __enter__(self):
        self.t0 = time.perf_counter()
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"{time.perf_counter() - self.t0:.3f} s")
        return False          # do not swallow errors

with Timer():
    run_simulation()
```

If `run_simulation()` raises an error, the time is still printed and the error still propagates.

Operators are a negotiation

The numeric protocol shows how carefully the design is balanced. For `a * b`, Python runs a short negotiation:

1. Ask `a.__mul__(b)`.
2. If `a` returns the special value `NotImplemented`, ask `b.__rmul__(a)`. The `r` stands for "reflected": `b` is on the right.
3. If both decline, raise `TypeError`.

There is one exception to the order. If `b`'s type is a subclass of `a`'s type and overrides `__rmul__`, Python asks `b` first. This lets a specialized subclass take control of mixed operations with its parent.

Neither class needs to know about the other in advance. The smallest example:

```
class Gain:
    def __init__(self, g):
        self.g = g
    def __mul__(self, x):
        if isinstance(x, (int, float)):
            return self.g * x
        return NotImplemented      # "not mine, ask the other side"
    __rmul__ = __mul__             # gain is commutative

Gain(2) * 3    # Gain.__mul__ -> 6
3 * Gain(2)    # int declines, then Gain.__rmul__ -> 6
```

The `int` type was written decades before `Gain`, yet `3 * Gain(2)` works.

The DSP twist: NumPy arrays

NumPy arrays join the same negotiation, with one extra step. Extend the type check to arrays and add one line:

```
import numbers

class Gain:
    __array_ufunc__ = None             # tell NumPy to defer to us (see below)

    def __init__(self, g):
        self.g = g

    def __mul__(self, x):
        if isinstance(x, (numbers.Number, np.ndarray)):
            return self.g * x
        return NotImplemented

    __rmul__ = __mul__

x * Gain(2)    # ndarray declines, then Gain.__rmul__ -> float array
```

The `__array_ufunc__ = None` line matters in DSP code. A **ufunc** (universal function) is NumPy's name for an element-wise operation such as multiply, add or `exp`. `__array_ufunc__` is NumPy's own protocol for letting other objects take part in ufuncs.

The type check uses `numbers.Number` instead of `(int, float, complex)`. NumPy scalars such as `np.float32` and `np.int64` do not inherit from Python's `float` and `int`. NumPy does register them with the `numbers` ABCs, so `numbers.Number` accepts them.

Without the line, NumPy treats your `Gain` object as a scalar and **broadcasts** it. Broadcasting stretches the smaller operand to the shape of the larger one, then applies the operation element by element. `x * Gain(2)` then returns an array of dtype `object`, meaning an array of generic Python objects instead of numbers. It is slow and easy to miss. Setting `__array_ufunc__ = None` tells NumPy: "let the other operand handle this."

Prefer protocols over invented method names

If you are inventing a method name for something a Python keyword or built-in already does, use the protocol instead.

Traditional OOP mindset	The Pythonic protocol
`def get_size(self):`	`def __len__(self):`
`def get_sample(self, i):`	`def __getitem__(self, index):`
`def contains_prn(self, x):`	`def __contains__(self, item):`

Discovery tip: Pass a built-in object to `dir()`, for example `dir(list)`, to see which dunders it implements.

7. Python as an ICD: The Client/Supplier Dynamic

The Data Model works like a two-way Interface Control Document (ICD) between two subsystems:

1. **Subsystem A:** The Python runtime (syntax, built-in functions, the standard library).
2. **Subsystem B:** Your custom DSP and hardware blocks.

Python acts as the systems integrator. It enforces the contract without caring how either side is implemented.

- **The supplier hat:** You write classes that meet the contract (`__call__`, `__len__`, `__repr__`). Inside, you manage registers, buffers and hardware state.
- **The client hat:** You write the main processing loop. Because the supplier side met the contract, the client code is short and abstract. It also works automatically with built-in functions, the debugger, and libraries like NumPy.

The same object can be supplier and client at once. A block that implements `__iter__` is a supplier to `for`. When it loops over another stream internally, it is a client of that stream's protocol.

8. The Golden Rule of Dunders

You should almost never call a dunder method directly. Writing `my_buffer.__len__()` or `sig_a.__add__(sig_b)` is a red flag. Define the dunders on the class, then use native syntax and let the interpreter call them: `len(my_buffer)` or `sig_a + sig_b`.

Calling through the syntax is also more correct. `sig_a + sig_b` runs the full negotiation from Section 6, including the fallback to `sig_b.__radd__`. A direct `sig_a.__add__(sig_b)` call skips it and can return `NotImplemented` instead of a result.

The exceptions are cases where you deliberately reach a specific implementation:

- Calling a parent class through `super()`, as in `super().__init__()`.
- Delegating inside your own dunder, for example a wrapper whose `__getitem__` forwards to an inner array's `__getitem__`.

9. Practical Synthesis: The MATLAB System Object in Python

Combining the MATLAB System object lifecycle with the Python Data Model gives a clean ICD for signal-processing blocks. This version uses all three mechanisms: protocols, first-class objects, and descriptors.

It follows the **template-method pattern**. The base class owns the public lifecycle (`__call__`, `reset`, `release`) and its bookkeeping. Subclasses only fill in the `...Impl` hooks. A subclass therefore cannot corrupt the setup state by forgetting to call `super()`.

The `abc` module enforces the contract. `SystemObject` inherits from `ABC` (abstract base class, Section 5), and its required hooks are marked `@abstractmethod`. Python refuses to create an instance of any subclass that leaves an abstract method unimplemented. A missing `stepImpl` therefore fails immediately at construction, instead of deep inside a simulation run.

A descriptor for MATLAB "Nontunable" properties

In MATLAB, a nontunable property is locked once the object is running. In Python, this is a descriptor of about a dozen lines (Section 3):

```
class Nontunable:
    """A property that is locked while the object is running."""

    def __set_name__(self, owner, name):       # called once, when the class is built
        self.name = name
        self.private = "_" + name

    def __get__(self, instance, owner=None):
        if instance is None:                    # accessed on the class itself
            return self
        return getattr(instance, self.private)

    def __set__(self, instance, value):
        if getattr(instance, "_is_setup", False):
            raise AttributeError(f"'{self.name}' is nontunable; call release() first")
        setattr(instance, self.private, value)
```

This is the two-class pattern from Section 3. `Nontunable` defines the methods, and the object stored in `DelayLine` as `delay` is the descriptor.

In MATLAB terms, the pieces map like this:

MATLAB	Python	How many
The attribute kind `Nontunable`	The class `Nontunable`	One, reusable in any class
The declaration `properties (Nontunable) delay`	The descriptor object `delay = Nontunable()`	One per property per class
Each object's value of `delay`	`instance._delay`, in the instance's own dict	One per object

MATLAB's set of property attributes is fixed. In Python, a new attribute kind is an ordinary class of about a dozen lines. Note the vocabulary clash: in MATLAB, an "attribute" is a modifier such as `Nontunable`. In Python, an "attribute" is any name reached with a dot, such as `d.delay`.

Who is who in each call. Below, `D` is the one `Nontunable` object stored in `DelayLine.__dict__['delay']`, and `d` is a `DelayLine` instance.

Call Python makes	Triggered by	`self`	`instance`	Other arguments
`D.__set_name__(DelayLine, "delay")`	Class creation, once	`D`	—	`owner` = `DelayLine`, `name` = `"delay"`

Call Python makes	Triggered by	`self`	`instance`	Other arguments
`D.__set__(d, 10)`	`d.delay = 10`	`D`	`d`	`value` = 10
`D.__get__(d, DelayLine)`	Reading `d.delay`	`D`	`d`	`owner` = `DelayLine`
`D.__get__(None, DelayLine)`	Reading `DelayLine.delay`	`D`	`None`	`owner` = `DelayLine`

`self` is always the descriptor. The `DelayLine` object always arrives as `instance`. Per-instance values must therefore be stored on `instance`, never on `self`. One descriptor serves every `DelayLine`, so `self.value = value` would give all of them a single shared delay.

Why `__set__` matters. A descriptor whose class defines `__set__` is a **data descriptor**. Python gives it priority over the instance's own dict, so every write to `d.delay` goes through it, including the first one in `__init__`. A descriptor with only `__get__`, such as a function, is a **non-data descriptor**, and the instance dict wins. If `Nontunable` had no `__set__`, then `self.delay = delay` would write straight into the instance dict, and the lock would never run.

The base class

```python
from abc import ABC, abstractmethod
import numpy as np

class SystemObject(ABC):
    """MATLAB-style lifecycle exposed through the Python Data Model."""

    def __init__(self):
        self._is_setup = False

    # --- SUPPLIER CONTRACT: hooks that subclasses override ---
    @abstractmethod
    def setupImpl(self, x): ...

    @abstractmethod
    def stepImpl(self, x): ...

    def resetImpl(self): pass
    def releaseImpl(self): pass

    # --- PUBLIC LIFECYCLE: owned by the base class, never overridden ---
    def reset(self):
        if self._is_setup:
            self.resetImpl()

    def release(self):
        if self._is_setup:
            self.releaseImpl()
            self._is_setup = False              # always runs, whatever the subclass does

    # --- CLIENT INTEGRATION: the Python Data Model ---
    def __call__(self, x):
        """Callable protocol. First call runs setup then reset, as in MATLAB."""
        if not self._is_setup:
            self.setupImpl(x)
            self._is_setup = True
            self.resetImpl()
        return self.stepImpl(x)

    def __repr__(self):
        """Inspection protocol: what the debugger and hover tooltips show."""
        state = "Running" if self._is_setup else "Uninitialized"
        return f"<{type(self).__name__} | State: {state}>"

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Context manager protocol: guaranteed teardown, even on an error."""
        self.release()
        return False                            # never swallow the exception
```

A derived block: a delay line

The derived block adds the sequence protocol (`__len__`, `__getitem__`) on top of the lifecycle. Note how `setupImpl` sizes and types the buffer from the first input, just as MATLAB does.

```
class DelayLine(SystemObject):
    """Delays its input by a fixed number of samples."""

    delay = Nontunable()

    def __init__(self, delay):
        super().__init__()
        self.delay = delay      # right side: the parameter delay (a number)
                                # left side: attribute access, goes through
Nontunable.__set__

    def setupImpl(self, x):
        self._buf = np.zeros(self.delay, dtype=np.result_type(x))

    def resetImpl(self):
        self._buf[:] = 0
        self._head = 0                          # index of the oldest sample

    def stepImpl(self, x):
        out = self._buf[self._head]
        self._buf[self._head] = x
        self._head = (self._head + 1) % self.delay
        return out

    def releaseImpl(self):
        del self._buf

    # Sequence protocol: index the line, oldest sample first
    def __len__(self):
        return self.delay

    def __getitem__(self, index):
        if not self._is_setup:                  # no buffer before setup or after release
            raise RuntimeError("DelayLine is uninitialized; call it with data first")
        return np.roll(self._buf, -self._head)[index]
```

The two `delay`s in `self.delay = delay` are unrelated. The bare name on the right is the `__init__` parameter. Python never looks in the class body when resolving a bare name inside a method. Renaming the parameter to `n` and writing `self.delay = n` behaves identically.

The client loop

```
with DelayLine(delay=10) as delay_block:
    for sample in telemetry_stream:
        out = delay_block(sample)         # __call__ -> setup/reset once, then stepImpl
        oldest = delay_block[0]           # __getitem__

    try:
        delay_block.delay = 5             # locked while running
    except AttributeError as err:
        print(err)                        # 'delay' is nontunable; call release() first

print(delay_block)                        # <DelayLine | State: Uninitialized>
```

This code was run end to end. On a complex test stream, the output equals the input delayed by exactly 10 samples, with complex128 dtype inferred from the first sample.

Without the `try`, the `AttributeError` would propagate out of the `with` block. `__exit__` would still release the block, but the final `print` would never run.

This example processes one sample per call to keep the lifecycle easy to follow. In production code, pass a frame (a block of samples) to each call, as MATLAB System objects normally do. Per-sample calls hit the overhead wall described in Section 11.

10. IDE Integration and Essential Protocols: The Cheat Sheet

Modern IDEs are another client of your supplier code. Most IDE benefits come from protocols and **type hints** together. A type hint is an annotation such as `def __getitem__(self, i: int) -> complex:` that tells static tools what goes in and comes out. Type checkers read these hints: Pylance is the one built into VS Code, and mypy is a widely used command-line checker. Python itself ignores them at runtime.

Protocol family	Dunders / tools	Trigger syntax	Runtime DSP use case	IDE / VS Code benefit
Callable	`__call__`	`my_block(data)`	Core execution trigger; turns the object into a stateful function.	Hover shows the call signature. The output type is inferred from `__call__`'s return hint.
Inspection	`__repr__`	Debugger, hover, `print`	Shows internal state at a glance.	Sets the text shown in the Variables panel, hover tooltips and Debug Console.
Context manager	`__enter__`, `__exit__`	`with block as b:`	Guaranteed teardown, even when a simulation crashes.	The type of `b` is inferred from `__enter__`'s return hint.
Sequence	`__len__`, `__getitem__`	`len(buf)`, `buf[0:5]`	Ring buffers and delay lines that index like arrays.	The checker accepts `len()` and `[]` on the object and infers the element type.
Iterator	`__iter__`, `__next__` (or a generator with `yield`)	`for chunk in stream:`	Stream large binary GNSS files chunk by chunk to save RAM.	The loop variable's type is inferred from the iterator's hints.
Numeric	`__add__`, `__mul__`, `__rmul__`, ...	`sig_a * sig_b`	Mix signals and apply gains with natural math.	The result type of `a * b` is inferred from the return hint.
Memory bound	`__slots__`	Class attribute	Removes the per-object dict. Saves memory on many small objects; attribute access is slightly faster.	Pylance and mypy flag assignments to attributes not listed in `__slots__`.
Static typing	`typing.Protocol`	Type hint	No runtime effect; describes a duck-typed interface.	The checker verifies an object matches the interface at the call site, with no inheritance needed.

The iterator row in practice

A **generator** is a function containing `yield`. Python turns it into an object that implements the iterator protocol automatically. Each `yield` hands out one item and pauses the function, with all its local state, until the next item is requested.

```
def nco(step):
    phase = 0.0
    while True:                  # infinite, yet costs nothing until asked
        yield np.exp(1j * phase)
        phase += step

osc = nco(0.1)
next(osc), next(osc)             # samples on demand; phase survives between calls
```

Combined with a context manager, streaming a large recording takes four lines:

```
def read_chunks(path, n_bytes):
    with open(path, "rb") as f:                    # context manager protocol
        while chunk := f.read(n_bytes):
            yield np.frombuffer(chunk, dtype=np.int8)  # iterator protocol, for free

for block in read_chunks("capture.bin", 1 << 20):
    process(block)
```

The `:=` operator, nicknamed the “walrus”, assigns and tests in one step. `while chunk := f.read(n_bytes):` reads the next chunk into `chunk` and keeps looping while it is non-empty. At the end of the file, `f.read` returns an empty bytes object, which counts as false.

Three protocols cooperate here, and none of them is visible in the client loop.

11. Performance Reality: Python vs. MATLAB

The comparisons with MATLAB in this section are rules of thumb, not measurements. The Python timings are measured, on one machine, so treat them as orders of magnitude and time your own workload.

For vectorized work, NumPy and SciPy are generally competitive with MATLAB. Both are high-level wrappers around the same kinds of compiled linear-algebra libraries (BLAS, LAPACK, Intel MKL or OpenBLAS).

Where the overhead hides. Every NumPy call has a fixed cost on the order of a microsecond, for argument parsing and dispatch. On a 1M-sample array this is negligible. On a 4-sample array inside a loop, it dominates. Sample-by-sample GNSS code usually hits this wall before anything else.

Operation	Measured time
`np.add` on 4 samples	0.6 µs
`np.sum` on 4 samples	2.4 µs

Operation	Measured time
`np.add` on 1M samples	2.8 ms
One-pole IIR filter over 1M samples, plain Python loop	300 ms
The same loop with Numba's `@njit`, after compilation	4.5 ms

These are best-of-several runs on Windows with an Intel CPU, using Python 3.12.3, NumPy 2.5.4 and Numba 0.68.1.

Loops. MATLAB's JIT compiles plain loops reasonably well. CPython has a specializing interpreter (3.11+) and an experimental JIT (first shipped in 3.13), but neither comes close to compiled code on numeric loops. In the table above, Numba runs the same loop about 65 times faster. Tight per-sample loops therefore need explicit compilation:

- **Numba (`@njit`):** Just-in-time (JIT) compilation of a Python function to machine code. The function is compiled on its first call, specialized to the actual argument types. Best for CPU loops over NumPy arrays.
- **CuPy:** A near drop-in NumPy replacement that runs on NVIDIA GPUs through CUDA.
- **Cython / pybind11:** Ahead-of-time compilation. Cython compiles annotated Python to C. pybind11 exposes existing C++ code to Python.

Note how each of these tools plugs in through the architecture already described. `@njit` is a decorator (Pillar 2). CuPy arrays mimic NumPy through the same protocols (Pillar 1).

CuPy is the strongest proof of the thesis. CuPy arrays implement two NumPy protocols: `__array_ufunc__` (Section 6) for element-wise operations, and `__array_function__`, which lets an array type take over any NumPy function called on it. The same client function then runs on either device:

```
def power(x):
    return np.sum(np.abs(x) ** 2)

power(np.ones(1000))     # runs on the CPU, returns a NumPy result
power(cp.ones(1000))     # runs on the GPU, returns a CuPy array
```

`power` never mentions the GPU. NumPy sees a CuPy array and hands the call to CuPy. This works only for functions CuPy implements; for the rest, NumPy raises an error instead of silently copying to the CPU.

A rewrite in C++ or Rust is usually only needed for bare-metal embedded targets, hard real-time latency, or tight RAM limits where per-object overhead matters. Offline GNSS simulation and signal generation can often be done entirely in Python.

12. What the Design Costs

Everything in this guide has a price, and other languages make different trades for good reasons.

- **Speed.** Runtime dispatch on every operation is why plain Python loops are slow (Section 11).
- **Memory.** Every value is a full object (Section 2). A Python `float` takes 24 bytes to hold 8 bytes of data. A list of 1M floats takes about 32 MB, while a NumPy array of the same values takes 8 MB.
- **Threads and the GIL.** The default CPython build has a Global Interpreter Lock (GIL), so only one thread runs Python bytecode at a time. Among other things, it keeps updates to `ob_refcnt` safe. Threads therefore do not speed up CPU-bound Python code. NumPy releases the GIL during many operations, and `multiprocessing` sidesteps it at the cost of moving data between processes. The free-threaded build (3.13+) removes the GIL, but it is slower on a single thread and depends on extension modules supporting it.
- **Late errors.** With duck typing, a missing method or wrong type surfaces only when that line runs, possibly hours into a simulation. Type checkers catch many of these errors (Section 10), but nothing enforces them at runtime.
- **Hidden behavior.** `a * b` can do anything: allocate memory, run a slow Python loop, or move work to a GPU. The syntax does not reveal the cost. Because almost anything can change at runtime (Section 3), compilers and analysis tools can assume very little either.
- **Limits of the plugin model.** `is` and plain name assignment cannot be overloaded. `and`, `or` and `not` can only consult `__bool__`. That is why NumPy uses `&`, `|` and `~` for element-wise logic, and why `a and b` on multi-element arrays raises `ValueError`.
- **Accumulated rules.** The design is compact, but its edge cases are real. The `__getitem__` fallback for `for` (Section 1), the subclass exception in operator dispatch (Section 6) and data versus non-data descriptors (Section 9) all have to be learned.

This guide also leaves out two related topics: the method resolution order, which is how Python searches base classes under multiple inheritance, and metaclasses, which are classes whose instances are classes. For those and for more depth on the data model, see the “Data model” chapter of the Python Language Reference, the official Descriptor Guide, and Luciano Ramalho’s *Fluent Python*.

13. Epilogue: Why “Data Model”?

The term comes from the title of the chapter in the official Python Language Reference that defines all of this. The core developers never wrote down why they chose it. What follows is an interpretation, but it fits the design well.

- **Infrastructure** would suggest plumbing: compilers and memory allocators. Python’s philosophy is that you should not need to think about those.
- **Interface** would suggest a thin boundary between components.

- **Data Model** describes what kinds of things exist in the language and how they behave: a sequence, a number, an iterator, a callable, a descriptor. Python does not dictate how your algorithm is written. It only requires that your data declare what it is, so the C engine underneath knows how to run it.

That is the core of the design. The language stays small, and your objects tell it the rest.