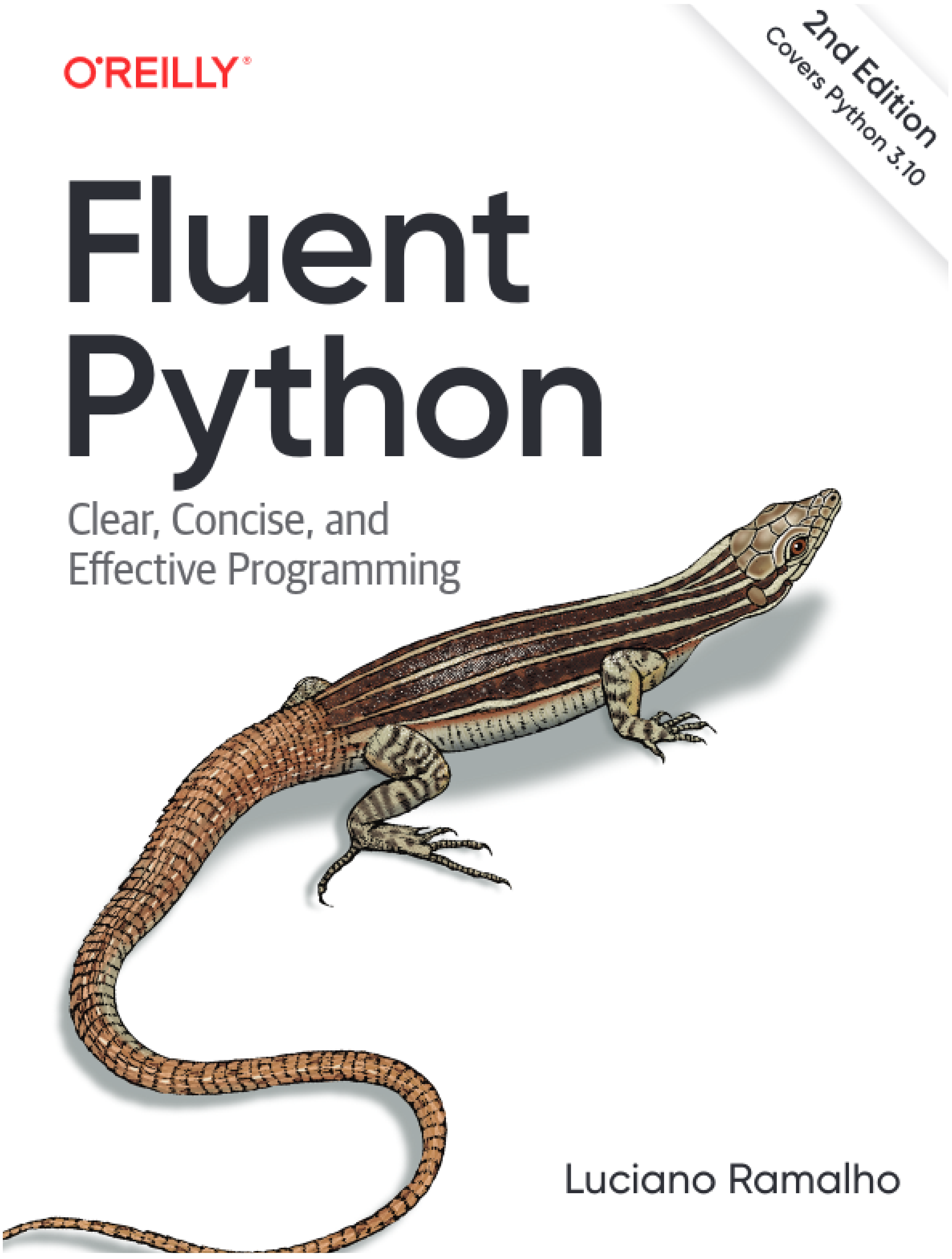
THE PYTHON DATA MODEL

ARIHARASUDHAN

What is this short book?

This short book series is a summarization of chapters in the so called “Fluent Python” book by Luciano Ramalho.



The Python Data Model

In Python, we use len(collection) but in most of the other languages, we use collection.len(). This oddity is the key to everything we call "Pythonic". The Python Data Model is basically a set of rules that govern how we interact with objects in Python.

For example, In Python, to find out how many things are in a collection, we use the len() function.

```
my_list = [1, 2, 3, 4, 5]
print(len(my_list))  # Output: 5
```

We can access items in a collection using indexing ([]).

```
my_list = [1, 2, 3, 4, 5]

print(my_list[0])  # Output: 1
```

We can loop through items in a box using a for loop.

```
my_list = [1, 2, 3, 4, 5]
for item in my_list:
    print(item)
```

Special / Dunder Methods

When we use a framework, we often have to write methods that the framework will call at various points during its execution. Similarly, in Python, when we leverage the Python Data Model to build custom classes, we can define special methods that the Python interpreter will call in response to specific operations on instances of our classes. These methods are invoked to perform basic object operations.

Special methods in Python are identified by their names, which are always written with leading and trailing double underscores. For example, if we have an object obj and we use the syntax obj[key], it is supported by the __getitem__ special method. Internally, the interpreter calls obj.__getitem__(key) to evaluate the expression.

These Special methods enable us to customize how our objects behave in various contexts, making our code more flexible and expressive.

By defining these special methods in our classes, you can make them behave like built-in types or integrate seamlessly with frameworks that rely on these operations. It provides a powerful mechanism for customization and abstraction in Python programming. For example,

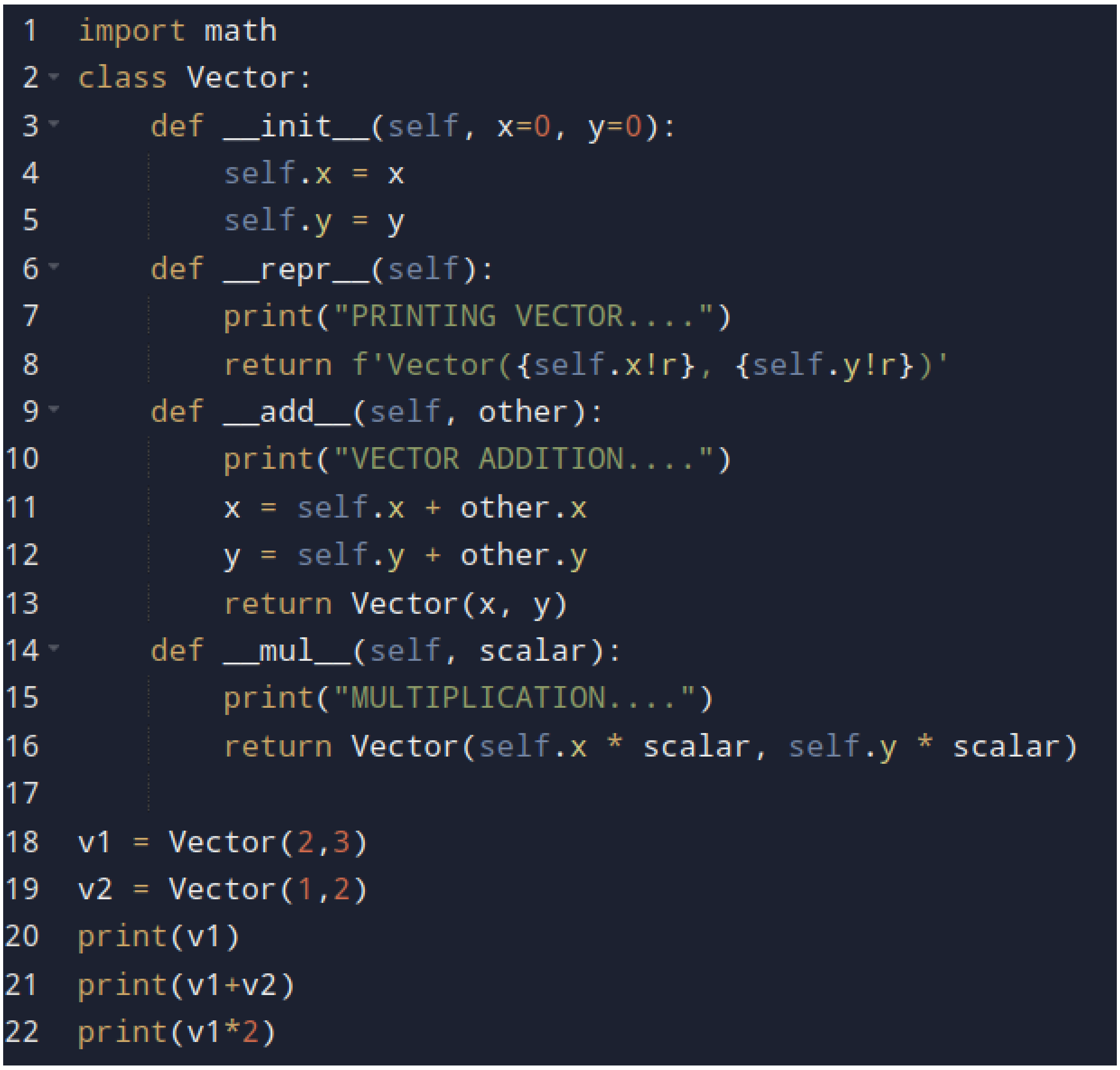
```
class CustomList:
    def __init__(self, *args):
        self._data = list(args)

    def __getitem__(self, index):
        return self._data[index]

    def __setitem__(self, index, value):
        self._data[index] = value

    def __len__(self):
        return len(self._data)

    def __str__(self):
        return str(self._data)

if __name__ == "__main__":
    custom_list = CustomList(1, 2, 3, 4, 5)
    # Access elements using indexing
    print(custom_list[0])  # Output: 1
    # Modify elements using indexing
    custom_list[2] = 10
    print(custom_list)  # Output: [1, 2, 10, 4, 5]
    # Get length of the custom list
    print(len(custom_list))  # Output: 5
    # String representation of the custom list
    print(custom_list)  # Output: [1, 2, 10, 4, 5]
```

Using Special Methods

Special methods in Python are designed to be invoked by the Python interpreter, not directly by us! Instead of calling my_object.__len__(), we typically use len(my_object), which automatically triggers the __len__ method we implemented, especially for user-defined classes. However, for built-in types like lists, strings, and NumPy arrays, Python optimizes by accessing internal attributes directly, such as the ob_size field in C-based collections, making operations like len(my_object) much faster. Implicit calls to special methods often occur, such as iter(x) being invoked by for i in x, which may internally call x.__iter__() or x.__getitem__(). Generally, our code should minimize direct calls to special methods, relying on built-in functions like len, as they often provide additional services and are faster, especially for built-in types. The only special method commonly called directly by user code is __init__ to invoke superclass initializers in our own __init__ implementation.

Emulating Numeric Types

Let’s discuss how special methods in Python are utilized to implement numeric types like vectors. Special methods such as `__repr__`, `__abs__`, `__add__`, and `__mul__` are utilized to enable operations like addition and scalar multiplication.

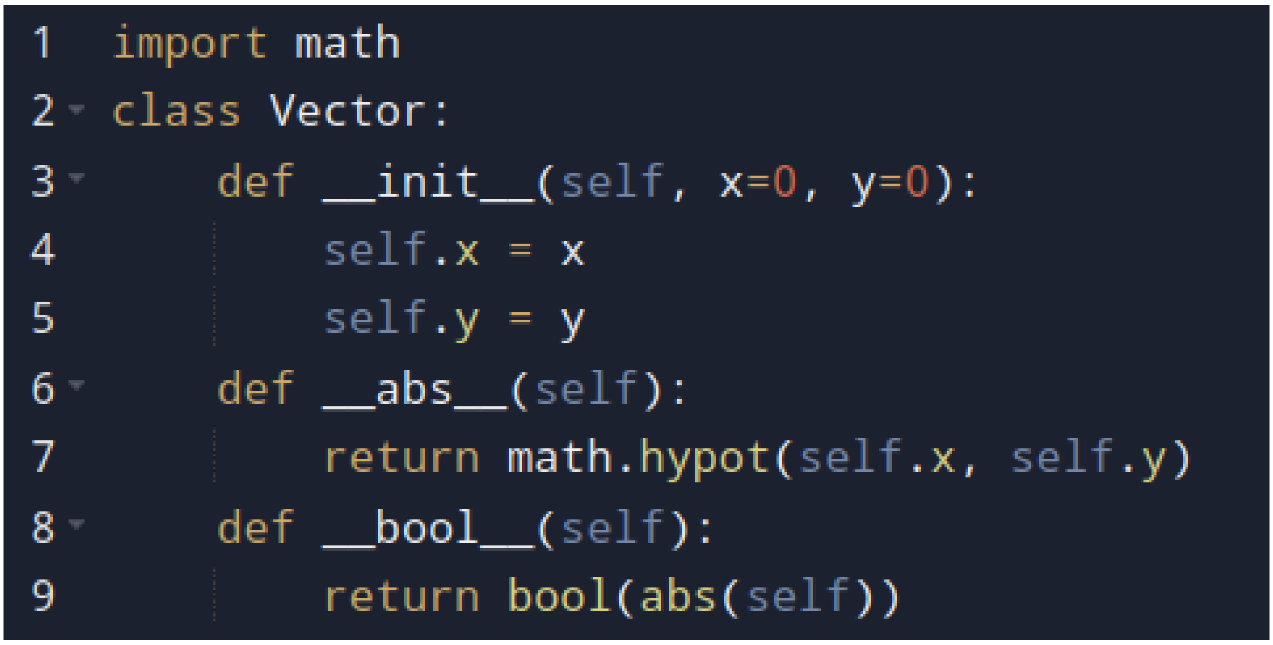


```python
import math
class Vector:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __repr__(self):
        print("PRINTING VECTOR....")
        return f'Vector({self.x!r}, {self.y!r})'
    def __add__(self, other):
        print("VECTOR ADDITION....")
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)
    def __mul__(self, scalar):
        print("MULTIPLICATION....")
        return Vector(self.x * scalar, self.y * scalar)

v1 = Vector(2,3)
v2 = Vector(1,2)
print(v1)
print(v1+v2)
print(v1*2)
```

```
PRINTING VECTOR....
Vector(2, 3)
VECTOR ADDITION....
PRINTING VECTOR....
Vector(3, 5)
MULTIPLICATION....
PRINTING VECTOR....
Vector(4, 6)
```

The implementation ensures that these operations return new instances of the vector class without modifying the original operands.

The __repr__ special method is called by the repr built-in to get the string representation of the object for inspection. Without a custom __repr__, Python's console
would display a Vector instance <Vector object at 0x10e100070>. __str__ is called by the str() built-in and implicitly used by the print function. It should return a string suitable for display to end users. Sometimes same string returned by __repr__ is user-friendly, and you don't need to code __str__ because the implementation inherited from the object class calls __repr__ as a fallback.



```python
import math
class Vector:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __abs__(self):
        return math.hypot(self.x, self.y)
    def __bool__(self):
        return bool(abs(self))
```

By default, instances of user-defined classes are considered truthy, unless either __bool__ or __len__ is implemented. Basically, bool(x) calls x.__bool__() and uses the result. If __bool__ is not implemented, Python tries to invoke x.__len__(), and if that returns zero, bool returns False. Otherwise bool returns True. Our implementation of __bool__ is conceptually simple: it returns False if the magnitude of the vector is zero, True otherwise. We convert the magnitude to a Boolean using bool(abs(self)) because __bool__ is expected to return a Boolean. Outside of __bool__ methods, it is rarely necessary to call bool() explicitly, because any object can be used in a Boolean context.

Some special methods are listed here :

Category	Method names
String/bytes representation	__repr__ __str__ __format__ __bytes__ __fspath__
Conversion to number	__bool__ __complex__ __int__ __float__ __hash__ __index__
Emulating collections	__len__ __getitem__ __setitem__ __delitem__ __contains__
Iteration	__iter__ __aiter__ __next__ __anext__ __reversed__
Callable or coroutine execution	__call__ __await__
Context management	__enter__ __exit__ __aexit__ __aenter__
Instance creation and destruction	__new__ __init__ __del__
Attribute management	__getattr__ __getattribute__ __setattr__ __delattr__ __dir__
Attribute descriptors	__get__ __set__ __delete__ __set_name__
Abstract base classes	__instancecheck__ __subclasscheck__
Class metaprogramming	__prepare__ __init_subclass__ __class_getitem__ mro entries

Operator category	Symbols	Method names
Unary numeric	- + abs()	__neg__ __pos__ __abs__
Rich comparison	< <= == != > >=	__lt__ __le__ __eq__ __ne__ __gt__ __ge__
Arithmetic	+ - * / // % @ divmod() round() ** pow()	__add__ __sub__ __mul__ __truediv__ __floordiv__ __mod__ __matmul__ __divmod__ __round__ __pow__
Reversed arithmetic	(arithmetic operators with swapped operands)	__radd__ __rsub__ __rmul__ __rtruediv__ __rfloordiv__ __rmod__ __rmatmul__ __rdivmod__ __rpow__
Augmented assignment arithmetic	+= -= *= /= //= %= @= **=	__iadd__ __isub__ __imul__ __itruediv__ __ifloordiv__ __imod__ __imatmul__ __ipow__
Bitwise	& | ^ << >> ~	__and__ __or__ __xor__ __lshift__ __rshift__ __invert__
Reversed bitwise	(bitwise operators with swapped operands)	__rand__ __ror__ __rxor__ __rlshift__ __rrshift__

Why len is not a method?

len(x) runs very fast when x is an instance of a built-in type. The length of such built-in type is simply read from a field in a C struct. Getting the number of items in a collection is a common operation and must work efficiently for such basic and diverse types as str, list, memoryview, and so on. In other words, len is not called as a method because it gets special treatment as part of the Python Data Model, just like abs. But thanks to the special method __len__, you can also make len work with your own custom objects. This is a fair compromise between the need for efficient built-in objects and the consistency of the language.

TO BE CONTINUED