Week 2: Internal DSLs in Python

April 10, 2025

Internal DSLs live in a host language

Internal DSLs...

- are embedded within a host language
  - like a library

Internal DSLs live in a host language

Internal DSLs...

- are embedded within a host language
  - like a library
- have syntax and semantics that are a subset of the host language's
  - ok: `sound @ Volume(2)`
  - not ok: `sound <> Volume(2)`

Internal DSLs live in a host language

Internal DSLs...

- are embedded within a host language
  - like a library
- have syntax and semantics that are a subset of the host language's
  - ok: `sound @ Volume(2)`
  - not ok: `sound <> Volume(2)`
- are generally more accessible
  - interoperability through host
  - metaprogramming (functions, classes, ...) through host
  - familiar syntax

Internal DSLs live in a host language

Internal DSLs...

- are embedded within a host language
  - like a library
- have syntax and semantics that are a subset of the host language's
  - ok: `sound @ Volume(2)`
  - not ok: `sound <> Volume(2)`
- are generally more accessible
  - interoperability through host
  - metaprogramming (functions, classes, ...) through host
  - familiar syntax
- **rely on the extensibility of the host**

Some Python Internal DSLs

Some Python Internal DSLs

```python
import tensorflow as tf

with tf.Session() as sess:
  # Phase 1: constructing the graph
  a = tf.constant(15, name="a")
  b = tf.constant(5, name="b")
  prod = tf.multiply(a, b, name="Multiply")
  sum = tf.add(a, b, name="Add")
  res = tf.divide(prod, sum, name="Divide")

  # Phase 2: running the session
  out = sess.run(res)
  print(out)
```

Some Python Internal DSLs

SymPy

```python
import tensorflow as tf

with tf.Session() as sess:
  # Phase 1: constructing the graph
  a = tf.constant(15, name="a")
  b = tf.constant(5, name="b")
  prod = tf.multiply(a, b, name="Multiply")
  sum = tf.add(a, b, name="Add")
  res = tf.divide(prod, sum, name="Divide")

  # Phase 2: running the session
  out = sess.run(res)
  print(out)
```

Some Python Internal DSLs

```python
import tensorflow as tf

with tf.Session() as sess:
  # Phase 1: constructing the graph
  a = tf.constant(15, name="a")
  b = tf.constant(5, name="b")
  prod = tf.multiply(a, b, name="Multiply")
  sum = tf.add(a, b, name="Add")
  res = tf.divide(prod, sum, name="Divide")

  # Phase 2: running the session
  out = sess.run(res)
  print(out)
```

Flask

Contents

Quickstart

Eager to get started? This page gives a good introductio
and install Flask first.

A Minimal Application

A minimal Flask application looks something like this:

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

Today

How can we **extend** Python
to create internal DSLs?

Agenda

Custom Operators

Custom Blocks

Custom Definitions

Deferred Execution

Custom Operators

How can this code

```
(A & B) - C
```

apply to sets instead of numbers?

Operator Overloading

In Python, operators on user-defined classes dispatch to specific methods.

The Python data model documents every operator and its method(s).

The expression `a + b` is evaluated as `a.__add__(b)`.

(If this is unimplemented, then Python tries `b.__radd__(a)`.)

Infix overloads

Arithmetic

- `+` `__add__`
- `-` `__sub__`
- `*` `__mul__`
- `/` `__truediv__`
- `//` `__floordiv__`
- `%` `__mod__`
- `@` `__matmul__`
- `**` `__pow__`

Arithmetic

- `+=` `__iadd__`
- ⋮
- `+` `__radd__`
- ⋮
- `+` `__pos__`
- `-` `__neg__`
- `~` `__invert__`

Bitwise

- `&` `__and__`
- `|` `__or__`
- `^` `__xor__`
- `<<` `__lshift__`
- `>>` `__rshift__`

Comparison

- `!=` `__ne__`
- `==` `__eq__`
- `>=` `__ge__`
- `>` `__gt__`
- `<=` `__le__`
- `<` `__lt__`

Special overloads

```
      xs[i]   ⟹   xs.__getitem__(i)
       if x   ⟹   if x.__bool__()
    x in xs   ⟹   xs.__contains__(x)
    len(xs)   ⟹   xs.__len__(x)
     f(*xs)   ⟹   f.__call__(*xs)
for x in xs   ⟹   for x in xs.__iter__()
```

Live example: multiset

Our goal:

```
>>> a = Multiset(1, 1, 2)
>>> b = Multiset(1, 4, 5)
>>> a + b
Multiset(1, 1, 1, 2, 4, 5)
>>> a | b
Multiset(1, 1, 2, 4, 5)
>>> a & b
Multiset(1)
>>> a - b
Multiset(1, 2)
```

Custom Blocks

Some compound statements can be customized

```
if condition:
  # code

for item in collection:
  # code

with open("out.txt", "w") as f:
  # code

# others: while, match, try
```

Some compound statements can be customized

```python
if condition:
  # code

for item in collection:
  # code

with open("out.txt", "w") as f:
  # code

# others: while, match, try
```

You can customize `for` by defining `__iter__` for `collection`.

Some compound statements can be customized

```
if condition:
  # code

for item in collection:
  # code

with open("out.txt", "w") as f:
  # code

# others: while, match, try
```

You can customize `for` by defining `__iter__` for `collection`.

You can also customize `with`...

With statements

```
with open("out.txt", "w") as f: # opens file

  # code (manipulates file)

  # file is implicitly closed
  # (even with an exception)
# post-close code
```

With statements

```python
with open("out.txt", "w") as f: # opens file

  # code (manipulates file)

  # file is implicitly closed
  # (even with an exception)
# post-close code
```

This works because `open("out.txt", "w")` is a **context manager**.

With statements

```python
with open("out.txt", "w") as f: # opens file

  # code (manipulates file)

  # file is implicitly closed
  # (even with an exception)
# post-close code
```

This works because `open("out.txt", "w")` is a **context manager**.
It implements `__enter__` and `__exit__`.

- `__enter__(self) -> Any`
  - return value is bound to `f` in “`as f`.”

With statements

```python
with open("out.txt", "w") as f: # opens file

  # code (manipulates file)

  # file is implicitly closed
  # (even with an exception)
# post-close code
```

This works because `open("out.txt", "w")` is a **context manager**.
It implements `__enter__` and `__exit__`.

- `__enter__(self) -> Any`
  - return value is bound to `f` in "`as f`."
- `__exit__(self, exception info) -> bool`
  - return value: whether to re-raise the exception

Live example: terminal color

Our goal:

```
>>> with(Color.RED): print("this is red")
this is red
>>> print("this is black")
this is black
>>> with(Color.BLUE): print("this is blue")
this is blue
```

Custom Definitions

Customizable assignment?

In Python, assignment (=) **cannot** be overloaded.

Customizable assignment?

In Python, assignment (`=`) **cannot** be overloaded.

- $\Longrightarrow$ DSLs override similar operators: `@=`, `<<=`, ...

Customizable assignment?

In Python, assignment (=) **cannot** be overloaded.

- $\Longrightarrow$ DSLs override similar operators: `@=`, `<<=`, ...
    - An example from Magma (a Python hardware DSL):

```
class BasicWhen(m.Circuit):
    io = m.IO(I=m.In(m.Bits[2]), S=m.In(m.Bit), O=m.Out(m.Bit))
    with m.when(io.S):
        io.O @= io.I[0]
    with m.otherwise():
        io.O @= io.I[1]
```

Customizable assignment?

In Python, assignment (=) **cannot** be overloaded.

- $\Longrightarrow$ DSLs override similar operators: `@=`, `<<=`, ...
  - An example from Magma (a Python hardware DSL):

```
class BasicWhen(m.Circuit):
    io = m.IO(I=m.In(m.Bits[2]), S=m.In(m.Bit), O=m.Out(m.Bit))
    with m.when(io.S):
        io.O @= io.I[0]
    with m.otherwise():
        io.O @= io.I[1]
```

But, *definitions* **can** be customized.

Customizable assignment?

In Python, assignment (=) **cannot** be overloaded.

- $\implies$ DSLs override similar operators: `@=`, `<<=`, ...
  - An example from Magma (a Python hardware DSL):

```
class BasicWhen(m.Circuit):
    io = m.IO(I=m.In(m.Bits[2]), S=m.In(m.Bit), O=m.Out(m.Bit))
    with m.when(io.S):
        io.O @= io.I[0]
    with m.otherwise():
        io.O @= io.I[1]
```

But, *definitions* **can** be customized.

- Function definitions: `def foo(..):`
- Class definitions: `class Foo(..):`

Decorator syntax

The following is an instance of a *decorator* applied to a function definition.

```
@my_decorator
def foo(..):
  # code
```

It is essentially equivalent to the following:

```
def foo(..):
  # code
foo = my_decorator(foo)
```

Live example: tracing

Our goal:

```
@rec_trace
def fib(n): return n if n < 2 else return fib(n - 1) +
    fib(n - 2)
>>> print(fib(3))
call fib(3)
 call fib(2)
  call fib(1)
  ret  1 = fib(1)
  call fib(0)
  ret  0 = fib(0)
 ret  1 = fib(2)
 call fib(1)
 ret  1 = fib(1)
ret  2 = fib(3)
2
```

Deferred Execution

Python's extensibility

Python is extensible. You can:

- customize operator semantics
- customize `with`-block entry/exit events
- wrap definitions

Python's extensibility

Python is extensible. You can:

- customize operator semantics
- customize `with`-block entry/exit events
- wrap definitions

Python's extensibility has limits.

- Evaluation order is fixed.
  - `A + B`, `A` always evaluates before `B` and before `+`.
- Precedence is fixed.
- Some operators are not overloadable: `=`, `and`, `or`, `not`.
- Lambdas are verbose and can't contain statements.
  - `lambda x, y: x + y`
- Evaluation is eager.

Breaking limits through external techniques

We can circumvent Python's limits with an *external* tool:

- an AST.

Breaking limits through external techniques

We can circumvent Python's limits with an *external* tool:

- an AST.

Two steps:

- Use Python's **evaluation semantics** to **build an AST**.
- Later, execute that AST using a custom interpreter.

Breaking limits through external techniques

We can circumvent Python's limits with an *external* tool:

- an AST.

Two steps:

- Use Python's **evaluation semantics** to **build an AST**.
- Later, execute that AST using a custom interpreter.

Some remarks:

- True execution is **deferred** until after Python's execution.
- The interpreter(/...) is often (but not always) in Python.
- This gives semantic flexibility of an external DSL.
- The does not improve syntactic flexibility very much.

Live example: auto-differentiation

Our goal:

```
@formula
def f(x, y):
  return x * x + y
  # derivative in x: 2 * x

>>> f(x=2, y=1)
5
>>> f.deriv("x")(x=2, y=1)
4
```

Recap

Custom operators (overloading)

Custom blocks (context managers)

Custom definitions (decorators)

Deferred execution (ASTs for internal DSLs)

The internal lab will exercise all of these skills.

Next class: examples of graphics DSLs!