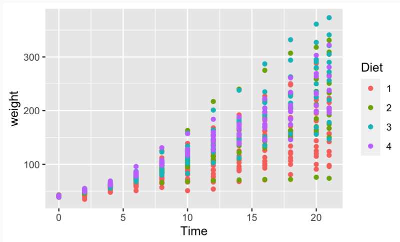
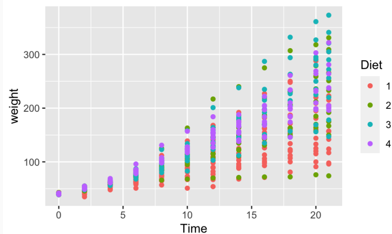
Week 2: Internal DSLs in Python

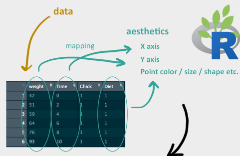
April 8, 2024

DSL of the day: ggplot2

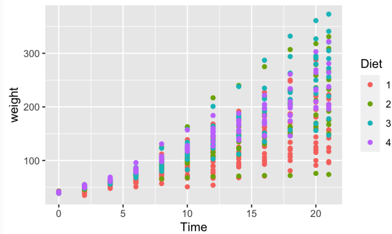


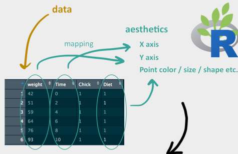
DSL of the day: ggplot2

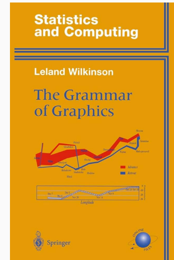




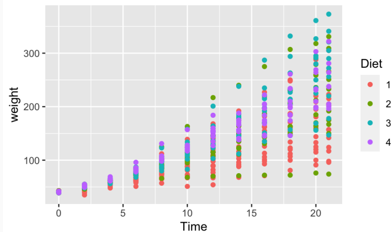
DSL of the day: ggplot2

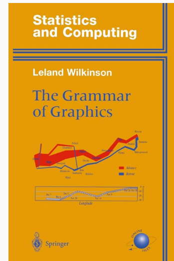






DSL of the day: ggplot2





ggplot2, simplified:

data

aesthetics

mapping

X axis

Y axis

Point color / size / shape etc.

	weight	Time	Chick	Diet
1	42	0	1	1
2	51	2	1	1
3	59	4	1	1
4	64	6	1	1
5	76	8	1	1
6	93	10	1	1

```
ggplot(data, aes(x=Time, y=weight)) +
  geom_point(aes(color=Diet))
```

```
df5 = dfmain %>%
  filter(country == "Singapore") %>%
  group_by(type) %>%
  mutate(cases7d = rollmean(cases, 7, na.pad = TRUE))

ggplot(df5, aes(date, cases, color = type)) +
  geom_point(size = 0.5) + geom_line(aes(y = cases7d)) +
  scale_x_date(date_breaks = "1 month", date_labels = "%d-%b") +
  scale_color_manual(values=c("darkorange2","firebrick","dodgerblue2")) +
  theme_classic(base_size = 24) +
  theme(axis.text.x = element_text(angle = 30, hjust = 1))
```

Layer	Function
Data	`ggplot(data)`
Aesthetics	`aes()`
Layers	`geom_*()` and `stat_*()`
Scales	`scale_*()`
Coordinate System	`coord_*()`
Facets	`facet_*()`
Visual Themes	`theme()` and `theme_*()`

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

```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
1 import tensorflow as tf
2
3 with tf.Session() as sess:
4   # Phase 1: constructing the graph
5   a = tf.constant(15, name="a")
6   b = tf.constant(5, name="b")
7   prod = tf.multiply(a, b, name="Multiply")
8   sum = tf.add(a, b, name="Add")
9   res = tf.divide(prod, sum, name="Divide")
10
11  # Phase 2: running the session
12  out = sess.run(res)
13  print(out)
```

Flask

Contents

Quickstart
A Minimal Application
Debug Mode
HTML Escaping
Routing

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)`.)

A laundry list

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

+= __iadd__
⋮
```

```
+ __radd__
⋮

+ __pos__
- __neg__
~ __invert__

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

```
if __bool__
() __call__
in __contains__
[] __getitem__
len __len__

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

Live example: multiset

Our goal:

```
1  >>> a = Multiset(1, 1, 2)
2  >>> b = Multiset(1, 4, 5)
3  >>> a + b
4  Multiset(1, 1, 1, 2, 4, 5)
5  >>> a | b
6  Multiset(1, 1, 2, 4, 5)
7  >>> a & b
8  Multiset(1)
9  >>> a - b
10 Multiset(1, 2)
```

Custom Blocks

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
```

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

```
1 with open("out.txt", "w") as f: # opens file
2
3   # code (manipulates file)
4
5   # file is implicitly closed
6   # (even with an exception)
7 # post-close code
```

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

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
```

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

With statements

`contextlib.contextmanager` is a convenience *decorator*[1] for implementing a context manager.

It converts a one-`yield` generator into a context manager.

```
@contextlib.contextmanager
def my_manager():
  # set up
  try:
    yield f # run block
  finally:
    # clean up
```

[1] We'll define this soon!

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.

- $\implies$ 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.

- $\implies$ 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.

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)
```

Decorators are widespread

My favorite stdlib decorator:

```
1 @dataclasses.dataclass
2 class Var(Expr):
3   name: str
```

Decorators are widespread

My favorite stdlib decorator:

```
1 @dataclasses.dataclass
2 class Var(Expr):
3   name: str
```

Other examples:

- `staticmethod` (method)
- `functools.total_ordering` (class)
- `functools.wraps` (function)
- `contextlib.contextmanager` (function)
- ... full list ...

Live example: terminal color

Our goal:

```
1  @rec_trace
2  def fib(n): return n if n < 2 else return fib(n - 1) +
       fib(n - 2)
3  >>> print(fib(3))
4  call fib(3)
5   call fib(2)
6    call fib(1)
7    ret  1 = fib(1)
8    call fib(0)
9    ret  0 = fib(0)
10  ret  1 = fib(2)
11  call fib(1)
12  ret  1 = fib(1)
13 ret  2 = fib(3)
14 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: design!

Bonus question

Is SQL an internal DSL?