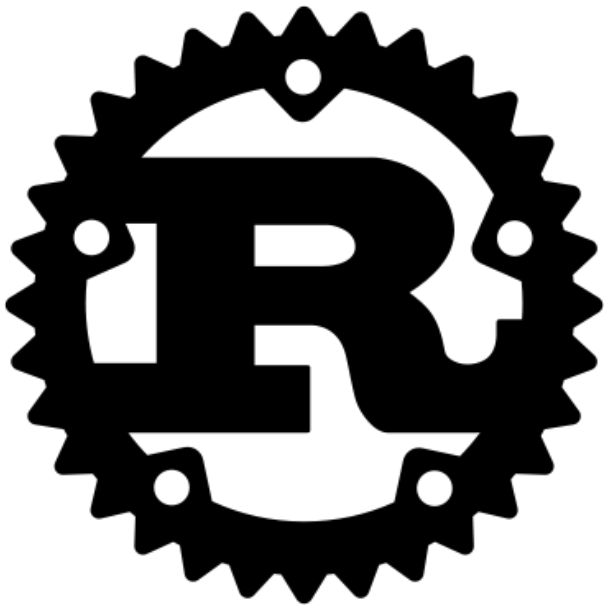
Algebraic sum types in Python

Simon Sapin
PyCon UK, 2013-09-22

Hi! I'm Simon.

(These notes expand a bit more than the actual five minutes lightning talk.)



rust-lang.org

It's nice!

I recently joined Mozilla, and I'm not doing as much Python anymore. I do Rust!

One of Rust's features that I miss in Python is its `enum` types.

Type theory

```
int
float
str
NoneType
```

Let's take a step back and talk about type theory for a minute.

Assuming you have some "basic" data types, one thing you can do is *compose* them into more complex types.

Composition

Product types: `A × B`

C, Rust:

```
struct Point { x: float, y: float }
```

Python: tuple, namedtuple, objects, ...

There are two fundamental ways to compose types. One of them is the *product,* which basically means that you take two (or more) things and put them together.

To do this C and Rust have structs, Rust and Python have tuples, and Python also has namedtuple, objects with attributes, etc. This is all well understood.

Composition

Sum type: `A + B`

Type algebra:

```
NoneType = 1
A × 1 = A
bool = 1 + 1 = 2
A + A = A × 2
```

The other way to compose things is the *sum type*. This means that your thing is one of several things, and only one at a time. (E.g. either a string or `None`.)

This product and sum for an *algebra* on types, much like the one you know on numbers even though these are not numbers at all. Algebraic types are fun but we don't have much time :)

C: *tagged union*

```
enum ShapeKind { Circle, Rectangle };
struct Shape {
  enum ShapeKind kind;
  union {
    struct {Point center; float radius}
      circle;
    struct {Point tl; Point br}
      rectangle
}};
```

C's *enum* types a special case of sum types where each term is `1`, the unit type (which only has one value.)

C's pattern for doing "real" sum types is the *tagged union*. It's not pretty.

Rust: enum

```
enum Shape {
    Circle(Point, float),
    Rectangle(Point, Point)
}
```

```
match shape {
  Circle(center, 0) => {...},
  Circle(center, radius) => {...},
  Rectangle(tl, br) => {...},
}
```

Rust on the other hand has built-in sum types, and calls them `enum`. Note how unlike in a C `enum`, each variant here can contain stuff.

Quite importantly, Rust also has a `match` pattern that does pattern matching, dispatching to different code branches (like C's `switch ... case`), and desconstruction (assigning fields to new local variables) all at once. This is very pleasant to use.

Python?

- PEP 435 Enum: like C, not like Rust
- Dynamic typing
- Object oriented: class hierachy, `isinstance()`, `.type` class attr
- Tuples: `('circle', x, y, r)`

  `('rectangle', x1, y1, x2, y2)`

We have a few options in Python, but none of them are quite as nice and general as Rust's `enum` with `match`.

Dymanic typing (eg. "pass either a string or a list") only helps when your variants are represented by different types, while series of `elif isinstance(…):` statements are just a pain to write.

Can we do better?

Another pattern in current Python?
Adding a `match` statement?

Discuss :)

Twitter: *@SimonSapin*
Email: *simon@exyr.org*

The point of this talk is to ask you, dear audience/reader: is there a better way?

Is there another (better) pattern in current Python? Or is it worth adding a new `match` statement in future versions of Python, possibly with a generalized Enum types?