Algebraic ABCs

Summary

ABCs[1] and type annotations are recent additions to the Python language. ABCs let programmers describe the kind of data they have in their programs. Type annotations associate these descriptions with particular variables. Together these let programmers add type–related metadata to their code and then use that information, through functions like `isinstance()` and `issubclass()`, when the program runs.

This paper describes work to provide more detailed metadata: instead of saying "this is a list" you can say "this is a list of integers". It also investigates how this information can be used: to check that the description matches the data, perhaps, or to indicate that different functions are called depending on the function arguments.

To do all this in a clear, consistent way, I use ideas from "type systems" in other languages. But that does not mean that I am adding such a system to Python; I am simply using the other languages to help decide how best to describe the data. The end result stays true to the idea that Python is "dynamically typed".

Author: Andrew Cooke (andrew@acooke.org)

Version: 1.0 / 2011-05-19 / `pytyp` v2.0

Latest: http://www.acooke.org/pytyp.pdf

Source: http://code.google.com/p/pytyp

Abstract

I explore what parametric, polymorphic, algebraic types might "mean" in Python, present a library that implements one approach, and suggest a few small changes to the base language as a consequence.

The result is a natural, expressive DSL[2], embedded in Python. Its semantics extend the ABC approach with registration of instances and iteration over values and types. The latter allows piecewise verification of values that cannot be registered.

The approach tries to be "pythonic": it adds type–related metadata to programs that can be used at run–time; it is optional, and builds on existing concepts. As with ABCs, correct use is not enforced by a static type system; no attempt is made to resolve conflicting specifications.

At the less pythonic extreme, the library also supports method overloading with dynamic dispatch and type–checking of functions via decorators.

[1]Abstract Base Classes

[2]Domain Specific Language.

Contents

Introduction

Some languages have static type systems, letting the compiler check for errors before the program is run. Often, in languages like Java, this seems more of a hindrance than a help; many people prefer Python for the freedom associated with a lack of type declarations.

But other languages — like Haskell and Scala — are using types in interesting ways. And even in Python we sometimes use commands like `isinstance()` and `issubclass()`.

So there are interesting questions to explore:

1. How do types currently work in Python? People often talk about classes as if they are types; more recently Python has gained ABCs. How do they fit together with "duck typing"?

2. Many "modern" uses of types are based on clean, orthogonal ideas. How do these fit with the data structures provided by Python? How do they fit with the existing support for types?

3. What core functions are needed in a library that extends Python's types? Instead of saying exactly how to do something, types gives us a way to describe what the results should look like. So, for example, how could we extend types so that it is easy to use them to write a library that can convert JSON data to a given set of Python classes?

Roadmap

In the first section, Current Status, I sketch Python's runtime type support. This shows how ABCs provide a clear, general model for duck typing.

The next section, Discussion, explores how new ideas can be added to Python. For example, we might extend the `Sequence` ABC, using `Sequence(int)` to describe sequences of integers. "Parametric ABCs" like this could support registration of instances as well as classes; for mutable containers that do not support hashing (and so cannot be registered) introspective, structural verification could be an option[3]:

```
>>> isinstance([1,2,None,4], Sequence(Option(int)))
True
```

A concrete implementation of all this (and more!) is described in The Pytyp Library (and Appendix: Further Details).

Finally, in Conclusions, I review the most import lessons from this work.

Terminology

Many terms used to discuss types have meanings related to the static verification of program properties. In this paper I am addressing a different subject. This means that I will often use the word "type" in a poorly defined way. When I need more precision I will use "(static) type system" (about which one can reliably reason without executing code), "type specification" (metadata using ABCs to describe Python data), and "duck types" (a model of runtime behaviour using available attributes).

Current Status

Python does not have a type system[4], but the language does have a notion of types.

[3]This particular example is not a valid `pytyp` specification. For practical reasons (the need to introduce a new metaclass, and the difficulty in modifying existing ABCs) the final library uses `Seq(Opt(int))`.

[4]In the sense defined in Terminology.

Classes and Attributes

The principal abstraction for structuring Python code is the class. This specifies a set of attributes (directly and through inheritance) for classes and their instances (objects). The class associated with an object is universally referred to as its type and available at runtime via the `type()` function[5].

However, the attributes associated with an object are not fixed — it is possible to modify objects through various mechanisms (including metaclasses and direct manipulation of the underlying dictionaries) — and the language runtime does not use the object's class to guide execution[6]. Instead, **each operation succeeds or fails depending on whether any necessary attribute is present on the instance in question**.

Even so, the notion that an instance's type is its class, and that this describes how it will behave, is very useful in practice: experienced Python programmers still describe the behaviour of programs in terms of types and classes. This is because Python's extreme flexibility, although useful and powerful, is rarely exploited to the full.

Duck Typing

Some operations appear specific to certain class instances. For example, the function `float()` only works for numerical types (or strings that can be interpreted as numerical values). But such examples can generally be explained in terms of attribute access via "special" methods (in the case of `float()`, the method `__float__()` on the function's argument).

I do not know if *every* operation can be explained in terms of attributes, but my strong impression is that this is the intention: **Python's runtime behaviour can be modelled in terms of attribute access**. In this way it implements (and defines) duck typing.

Recent Extensions

Recent work extended the language in two interesting ways.

First, it addressed the conflict described above: on the one hand, programmers behave as though Python's behaviour can be reliably explained in terms of types; on the other, the runtime functions in terms of available attributes. **Abstract Base Classes (ABCs) resolve this by identifying collections of attributes, providing a class–like abstraction that is better suited to duck typing.**

In more detail: a programmer can identify a set of attributes, create an ABC that contains these, and then either subclass, or call the `register()` method, to associate a class with the ABC. The metaclass for ABCs, `ABCMeta`, modifies the behaviour of `isinstance()` and `issubclass()` to expose this relationship at runtime.

It is important to understand that Python does not support the runtime *verification* of arbitrary duck types[7]:

[5]Where it matters, I am discussing only Python 3.

[6]Except for immutable types, which exist partly so that the implementation *can* make such an assumption and so operate more efficiently.

```
>>> class MyAbc(metaclass=ABCMeta):
...     @abstractmethod
...     def foo(self): pass
>>> class MyExample:
...     def foo(self): return 42
>>> issubclass(MyExample, MyAbc)
False
```

Instead, `MyExample` must either subclass `MyAbc` or register itself, populating a lookup table used by `issubclass()`. The ABC acts only as a marker that signals the veracity of the registered (or subclass) type; it does not perform a runtime check of the attributes[8].

Second, Python 3 supports type annotations. These are metadata associated with functions[9]. For example, the following is syntactically valid:

```
def func(a:int, b:str) -> list:
    return [a, b]
```

Type annotations are not interpreted or enforced by the language runtime. They are added to the function metadata and exposed through Python's `inspect` package.

When used with ABCs, **type annotations associate variables with type–related metadata.**

Summary

A consistent, simple, global model of Python's runtime type system exists. It is called "duck typing" and, as described above, depends on the availability of object attributes.

Recent work has started to build on this foundation by reifying collections of attributes (ABCs) and allowing metadata (formatted in a manner traditionally associated with types) to be specified on functions. However, ABCs act only as an unverified marker; runtime checks are restricted to a few special cases. Nor are type annotations verified.

So **ABCs are type metadata; `ABCMeta` associates type metadata with values and provides access to the relationship via `isinstance()`; type annotations associate type metadata with variables.** The rest of this paper builds on this.

Discussion

Typed Collections

How can we define the type of a list of values? Or a dictionary?

Answering these questions with tools from the previous section would start with the appropriate container ABC. This defines the attributes used to access the data. To define the contents we could then add type annotations:

```
class IntSequence(Sequence):
    def __getitem__(index) -> int:
        return super().__getitem__(index)
    ...
```

This has some problems[10], but is, I hope, a fair extrapolation of Python's current approach.

One problem is easy to fix. We can define a simpler syntax: `[int]` or, more formally, `Seq(int)`. I will call this a *type specification*.

This can be extended to inhomogeneous collections: dictionaries would look like `{'a':int, 'b':str}`; tuples like `(int, str)`. A unified syntax is `Rec(a=int, b=str)` or `Rec(int, str)` (named arguments are assumed to be string keys; unnamed arguments have implicit integer indices: 0,1,2...).

But we have a problem: the step from sequences to maps was more significant than a simple change of syntax. **When we try to translate `Rec()` back into ABCs with type annotations we find that we need dependent types**. The type of the return value from `__getitem__(key)` depends on the argument, `key`.

Nice syntax; shame about the semantics.

Semantics

To improve the semantics we must consider how a type specification is used. For example, we might intend to enforce runtime checking of function arguments, or to specify how data can be transcoded.

On reflection (and experimentation) I can find three broad uses for type specifications: verification; identification; and expansion.

Verification of a value's type (against some declaration) is traditionally performed by `isinstance()` and `issubclass()`. ABCs provide a mechanism for extending these, but still need an implementation for typed collections. We might examine the value structurally, comparing it against the type specification piece by piece. This approach is best suited to "data" types (lists, tuples and dictionaries) which are used in a polymorphic manner. Alternatively, we can use the existing registration and subclass mechanisms, which are more suited to user–defined classes.

Identification of a collection's type, although superficially similar to verification, is a harder problem. There is not always a single, well–defined answer. In some simpler cases we may have a set of candidate types, in which case we can verify them in turn, in other cases the instance's class may inherit from one or more ABCs. But I don't see a good, "pythonic" solution to the general problem. So this work does not extend `type()` to include typed collections.

Iteration over a collection by type covers a variety of uses where we want to process data in a manner informed by the associated types. One example is to automate mapping between `dict` and user–defined classes. Another is the structural type

[7]Excepting manual introspection and the "one trick pony" ABCs: `Hashable`, `Iterable`, `Iterator`, `Sized`, `Container` and `Callable`.

[8]This isn't completely true; when used with inheritance it is possible for ABCs to define abstract methods, which concrete implementations must supply.

[9]Python documentation calls them "function annotations", but the use cases in PEP3107 all refer to types.

[10]It is verbose, particularly when all methods are defined; type annotations don't exist for generators http://mail.python.org/pipermail/python-3000/2006-May/002103.html; it is unclear how to back-fit types to an existing API; type annotations are not "implemented"; it supports only homogenous sequences (as is normal with current type systems).

verification mentioned above. Handling ambiguous sum types is the most challenging point here.

Setting identification aside, we seem to have two possible semantics: one based on registration and subclassing of ABCs; the other iteration (similar to catamorphism or "folds").

A Little Formality

I will now explore how type specifications are related to various concepts from type theory. The aim here is not to directly emulate other languages, but to use common patterns to structure our approach.

Parametric Polymorphism

Since we started with data structures we have already addressed this: `Seq(x)` is polymorphic in `x`, for example. However, it is worth drawing attention to an important point: **polymorphism occurs naturally in Python data structures at the level of instances, not classes**. This contrasts with the current use of ABCs, which is at the class level.

So the idea that `isinstance([1,2,3], Seq(int))` evaluates as `True` implies a significant change to the language semantics: `isinstance()` depends on the *state* of an instance as well as its class. The relationship between `isinstance()` and `issubclass()` also shifts: the former cannot be expressed in terms of the latter (alone).

Product Types

The handling of maps above — `Rec(a=int, b=str)` — is close to the concept of product types: a record with a fixed number of values (referenced by label or index), each with a distinct type.

But there are some problems relating this to Python:

- The `Mapping` ABC does not include `tuple` or `list`, although these can be used as products.
- The `dict` class (and `list`, which can also function as a product, indexed by integers) has a variable number of entries. So `Rec()` should include a way to specify a type for "other" values.
- Class attributes look like products, but use `__getattr__()` rather than `__getitem__()`.

The final point can be addressed with a new specification, `Atr()`[11]. To avoid the need to specify all attributes on a class, `Atr()` should be open to additional entries (unlike `Rec()`, which is closed unless a default type is specified).

So Python appears to have two product types[12]; one associated with `__getitem__()`, `Rec()`; and one with `__getattr__()`, `Atr()`. Neither is closely associated with an existing ABC.

[11]`Atr()` has an advantage over `Rec()`: it does not require dependent types when reduced to ABCs with type annotations because each attribute would be described separately and so could have its own type.

[12]In comparison, Javascript's approach to attributes would require only a single type.

Sum Types

Although no Python feature maps directly to sum types — a value drawn from a set of types — there are various related ideas:

- Using `None` to indicate a missing value.
- The use of conditional code that either tests types — eg. `if isinstance()` — or returns multiple types from a single function.
- Subclassing and method dispatch.

This suggests a relationship between sum types, conditionals and dispatch; something that will become clearer in `pytyp`'s support for dynamic dispatch.

Because type specifications are metadata (and not a type system) we will not "know" the current type for a value associated with a sum. Iteration must attempt each possible type in turn, until one succeeds. With nested types this becomes a depth–first search over value / type combinations that backtracks on failures related to type errors.

I will use notation `Alt(a=int, b=str)` to describe sum types below. The optional labels might be used for dispatch by type, with a case–like syntax, for example.

Types as Sets

Types can be considered as [predicates that define] sets of values. This suggests two more type specifications: `And()`, which defines a type as the intersection of its arguments (so `And(MyClass, Seq(int))` would be the instances of `MyClass` that are also integer sequences); and `Or()` which is the union. Other set operations are possible, but don't appear to be very useful in practice[13].

`Or()` is similar to `Alt()`[14]; the difference is the ability to name alternatives, which means that `Alt()` is not associative, while `Or()` is.

`And()` is similar to multiple inheritance. Creating a new type specification using a combinator rather than inheritance simplifies the implementation and feels more natural (to me).

Summary

This section introduced a syntax that can describe polymorphic, algebraic data types (roughly translated into Python's runtime context) within Python code, largely at the instance level:

```
Seq(a)         # Sequences of type a

# products
Rec(a,b,...) # Type a x b x ... via __getitem__ or []
Atr(a,b,...) # Type a x b x ... via __getattr__ or .

# sums
```

[13]An argument could be made for `Not()`, particularly when using dynamic dispatch.

[14]`And()` and `Or()` parallel the product and sum types in structural verification and so share common ancestors in `pytyp`.

```
Alt(a,b,...) # Type a + b + ...
Opt(a)       # Alias of Alt(value=a,none=type(None))

# sets
And(a,b,...) # Type a n b n ... (intersection)
Or(a,b,...)  # Type a u b u ... (union)
```

In addition, because the specifications above are built using classes, we need a syntax to distinguish classes used as types[15] and another to allow dispatch by type (see Dynamic Dispatch by Type below):

```
Cls(c)       # Instances of c
Sub(c)       # Subclasses of c
```

Relating the semantics for these type specifications to existing language features is more difficult. In particular, **adding type annotations to ABCs faces significant problems**. First, it is incomplete: attributes, generators and named tuples do not support annotations. Second, dependent types would be needed to handle `dict`. Third, it is verbose, particularly when using standard container classes which must be subclassed for every distinct use, but also because it ignores correlations between the types of different attributes.

Registration with ABCs (or subclassing) is more promising, but cannot handle all cases, even if extended to include instances; a general solution will also require a structural (piecewise inspection) approach.

The Pytyp Library

The previous section explored a variety of ideas. Now I will describe an implementation: the `pytyp` library.

ABCs

`Pytyp` provides "parametric ABCs". So, for example, `Seq(int)` is a call to the type sequence constructor `Seq` that returns a dynamically generated ABC representing sequences of integers. The ABC is cached in `Seq` so that subsequent calls with the same type argument receive the same instance.

Like ABCs that already exist in Python, you can subclass `Seq(int)`, or register a class. In addition, you can also register hashable instances:

```
>>> ilist = HashableList(1,2,3)
>>> Seq(int).register_instance(ilist)
>>> isinstance(ilist, Seq(int))
True
```

Class Hierarchy

The full class hierarchy is shown below (subclassed or registered to right; `Sequence`, `Container` and `Mapping` are all existing Python ABCs):

```
Product
+- Sequence
|  `- Seq -- Seq(*) +- Seq(X)  # Sequences (like [])
|                   +- X in Sequence.__subclasses__
|                   `- tuple
+- Container
|  `-Rec -- Rec() +- Rec(X)    # Records (like {})
|                 +- X in Mapping.__subclasses__
|                 `- tuple
+- Atr -- Atr(X)               # Attributes (like A.b)
`- And -- And(X)               # Intersection

Sum
+- Alt -- Alt(X)               # Alternatives
|  `- Opt -- Opt(X)            # Optional (or None)
`- Or -- Or(X)                 # Union

Cls -- Cls(*) -- Cls(X)        # Class
```

None of the ABCs have abstract or mixin methods. `Foo(*)` implies a default `object` argument, so `Seq()` is equivalent to `Seq(object)`.

Several additional classes modify behaviour. Classes with `NoNormalize` as an *immediate* superclass are considered to be type specifications during normalisation; other classes will be wrapped by `Cls()`. `NoStructural` identifies classes that inherit from type specifications and so do not need structural verification. Subclasses of `Atomic` are displayed without the `Cls()` wrapper.

Construction and Inheritance

`Cls(X)` is the type specification for class `X`. It is not always needed (eg. `Seq(int)` and `Seq(Cls(int))` are equivalent), but removes ambiguity when using classes that are themselves type specifications. For an example, consider the difference between `Seq(int)` and `Cls(Seq(int))`: the former represents a sequence; the latter represents the class `Seq(int)`.

Unfortunately, this leads to a problem: if the subclass relation is transitive then we cannot reliably test for the types of type specifications. Consider the following:

1. `issubclass(Cls(X), Cls)`
2. `issubclass(X, Cls(X))`
3. `issubclass(X, Cls)`

[1] is true because we sometimes need to group parameterised types by "family" (eg. we need to be able to test whether a sequence of some type is a sequence, rather than a record).

[2] is true from the meaning of this particular type specification and the usual relationship between `isinstance()` and `issubclass()` (eg. both `isinstance(42, Cls(int))` and `issubclass(int, Cls(int))` are true).

[3] would be true if `issubclass()` were transitive. This is traditionally the case for any value of `X`, including `object` itself. So any class can be a subclass of `Cls`.

There is a conflict between "is X a type within the type specification Y?" and "is X a type specification of type Y?" To address this the library has the following structure:

[15] In `pytyp` this use of `Cls()` is optional in most cases; bare classes in type specifications will be automatically coerced to `Cls(...)`.

- **Type Specification Constructors** (eg. `Cls`, `Seq`) are ordinary classes whose `__new__` methods act as factories for type specifications.

- **Type Specifications** like `Cls(X)` and `Seq()` are[16] dynamically created classes, cached in the type constructor by type arguments, that have a `TSMeta` metaclass.

- **Type Specification Metaclass** (`TSMeta`) is a subclass of `ABCMeta` that extends registration to include instances, adds iteration and structural verification, etc.

This isolates the "magic" used to implement [2] (the logic in `ABCMeta` and `TSMeta` that `isinstance()` and `issubclass()` delegate to, and which is extended to make parametric polymorphism possible).

In summary: `issubclass(X, Cls)` asks if `X` is a subclass of the `Cls` constructor; `issubclass(X, Cls())` asks if `X` is described by the specification `Cls()`[17]. The first is resolved using normal Python subclassing; the second includes modified logic from `ABCMeta` and `TSMeta`. Since only the latter includes the support for parametric polymorphism we lose the unwanted transitivity.

This solution does not address the case where a type specification is subclassed, but those will be proper subclasses that are unlikely to be confusing during dispatch by type.

Instance Registration

`Pytyp` extend ABCs with an additional registry, for instances, populated by the `register_instance()` method.

`TSMeta` extends `__instancecheck__()`, called by `isinstance()`, to delegate to `__instancehook__()` on the class, if present. This parallels the use of `__subclasshook__()` within `__subclasscheck__()` (the standard ABC type extension mechanism).

Type specifications then implement `__instancehook__()` to check instances against the registry.

Structural Type Verification

Neither inheritance nor registration will help verify a list of integers, `[1,2,3]`: subclassing is not useful (`list` already exists, and anyway we need this to work at the instance level) and registration fails (the value cannot be hashed).

In cases like this we must fall back to structural verification: each entry is checked in turn (the mechanism is described in the next section, Iteration). This is inefficient, of course, so the programmer must consider whether it is appropriate. The alternative is a custom subclass:

```
>>> class IntList(list, Seq(int)): pass
>>> isinstance(IntList(), Seq(float))
False
```

[16]More exactly, "return".

[17]An instance of any class — `Cls()` is equivalent to `Cls(object)` — so the result is `True`.

Iteration

Iteration allows the type specification to guide processing of data. Each type specification implements `_for_each(data, callback)` and `_backtrack(data, callback)`. These both pass `callback` the current type specification and a generator that supplies `(value, spec, name)` for each sub–component of `data`.

So, for example, the call `Seq(int)._for_each([1, 2, 3], callback)` will provide `callback` with a generator that contains each list entry, in order, with a `spec` of `int`. In this case `name` will be None, but for `Rec()`, say, it will name the record.

The callback can recursively call `_for_each()` or `_backtrack()` on any sub–specifications, allowing the entire data structure to be processed.

The two methods differ in how they handle sum types (which have multiple possible types for a single value).

For Each

`_for_each()` passes `callback` each combination of type and value. For product types `callback` receives each value once, with a type; for sum types it receives each value multiple times, with a different type each time.

The callback must then handle the two cases appropriately. For example, the following code would implement structural type verification:

```
def callback(current, vsn):
    if isinstance(current, Product):
        for (value, spec, name) in vsn:
            if not isinstance(value, spec):
                return False
        return True
    else if isinstance(current, Sum):
        for (value, spec, name) in vsn:
            if isinstance(value, spec):
                return True
        return False
```

I have omitted many details, including the way that this would be called by `isinstance()`, but you can see how each case is handled separately.

Backtrack

In many cases, iteration over sum types means trying each type in turn until one works. For nested sum types this gives a depth first search of the possible value / type combinations. The `_backtrack()` routine makes this explicit: failure is indicated by raising an exception; the exception is caught and the next alternative tried.

In this approach, `callback()` receives only a single type for each value in a sum (other types are tried on alternative calls, if an exception is raised). The code for structural type verification becomes

```
def callback(current, vsn):
    for (value, spec, name) in vsn:
        if not isinstance(value, spec):
            raise TypeError
    return True
```

which is simpler than above because the logic for handling sum types is moved to `_backtrack()` itself.

Dynamic Dispatch by Type

Type specifications are metadata, implemented as ABCs, that can make APIs more declarative. Libraries that take this approach, like the JSON support in `pytyp`, must read and respond to that metadata. Unfortunately, this can result in code littered with calls to `isinstance()` and `issubclass()`.

The relationship between object types and program logic is usually implicit: OO method dispatch selects the correct action without explicit tests (the called method will have multiple implementations, depending on the type).

But this not possible with type specifications — it implies calling class methods on the ABCs themselves — so an alternative dispatch mechanism is needed. I have found dispatch by type, implemented as a decorator, to be extremely useful in these cases.

Here is a fragment of code used to encode data as JSON

```
class Encoder:

    @overload
    def __call__(self, value):
        return value

    @__call__.intercept
    def object(self, value):
        try:
            argspec = getfullargspec(init)
        except TypeError:
            return self.object.previous(value)
        ...

    @__call__.intercept
    def list(self, value:Sequence):
        return list(map(self.recurse, value))

    @__call__.intercept
    def map(self, value:Mapping):
        return dict((name, self.recurse(value))
                    for (name, value) in value.items())
```

which illustrates how the decorator is used. All the methods shown are accessed by calling an instance of `Encoder` as a function; ie. via `__call__()`. This works as follows:

- The target method (and default implementation) is marked with `@overload`.
- Other methods, one of which may be called *instead* of the target, are marked with `.intercept`. Methods are tried “from bottom to top”; arguments are checked against type annotations and the first successful match is invoked.
- Methods can explicitly pass the call up the chain by calling `.previous()` on the current method (see the `object()` method above).

The previous example, chosen for compactness, tests instances. When working with type specifications (ABCs) it is also useful to test subclasses. This explains the `Sub()` (pseudo–)type specification. For example,

```
@__call__.intercept
def rec(self, value, spec:Sub(Rec)):
    ....
```

will be invoked when the `spec` argument is a *subclass* of `Rec` — a type specification for a record.

Examples

The following examples build on the support for types described above to provide useful functionality.

Type Verification

The `checked` decorator verifies parameters and return values against the specification in the type annotation:

```
>>> @checked
... def int_list_len(s:[int]) -> int:
...     return len(s)
>>> int_list_len([1,2,3])
3
>>> int_list_len('abc')
Traceback (most recent call last):
  ...
TypeError: Type Seq(int) inconsistent with 'abc'.
```

JSON Decoding

Here JSON data, expressed using generic data–structures, are decoded into Python classes. Type specifications — in the call to `make_loads()` and via an annotation on the `Container()` constructor — are used to guide the decoding (implemented through nested iteration, as outlined earlier):

```
>>> class Example():
...     def __init__(self, foo):
...         self.foo = foo
...     def __repr__(self):
...         return '<Example({0})>'.format(self.foo)
>>> class Container():
...     def __init__(self, *examples:[Example]):
...         self.examples = examples
...     def __repr__(self):
...         return '<Container({0})>'.format(
...             ','.join(map(repr, self.examples)))
>>> loads = make_loads(Container)
>>> loads('[{"foo":"abc"}, {"foo":"xyz"}]')
<Container(<Example(abc)>,<Example(xyz)>)>
```

Conclusions

I have shown how type specifications — metadata using parameterised ABCs to describe Python data at the class and instance level — can be expressed within Python[18]. I have also provided an implementation with three operations: registration / subclassing; structural type verification; iteration.

Registration / subclassing and structural verification are complementary. The former allows classes and instances to be registered with, or inherit from, type specifications. This gives efficient verification of types. The latter is less efficient, but extends verification to mutable containers that cannot be registered. If performance is critical users can subclass and extend existing collections to make more efficient, registered classes.

`Pytyp` provides function decorators that verify arguments and implement dynamic dispatch by type.

Iteration is a general mechanism that can recursively explore a collection and the associated type specification. Because type specifications are not part of a static type system the concrete type of a value identified with a sum (ie. ambiguous or alternative) type is unknown; iteration must therefore support backtracking over the different possible combinations. This can be left to the client, or supported within the library.

`Pytyp` uses iteration to provide structural verification of types and the guided conversion of JSON data to Python classes.

Type specifications can help make APIs more declarative, but implementations must then be driven by the metadata. The resulting code is improved with dynamic dispatch by type, implemented as method decorators in `pytyp`.

Pythonic

The final decision on whether code is "pythonic" can only come from the community. And I suspect that they will not, in general, be supportive of the idea of "adding types" to Python.

However, the work described here does not implement, or advocate, a static type system. Instead, it builds on ideas already present in the language — ABCs, type annotations, `isinstance()` — to add optional features that respect the language semantics. For example, `Rec(int, str)` can describe a `dict` with keys `1` and `2`, a tuple, a even a list of length 2; no structure is imposed on the user beyond the attribute–based protocol (`__getitem__()` in this case) that already exists in the language.

Issues in Python

Type specifications describe parts of the Python language in a semi–formal way. So they highlight inconsistencies. That specifications are possible at all implies that Python is already a regularly structured language, but some irregularities have surfaced and I will describe them below.

[18]Implemented as an embedded, domain–specific language (EDSL).

Type Annotations

When developing `pytyp` my initial intention was for type specifications to be syntactic sugar that add type annotations to ABCs. This would make the type parameters explicit. Instead, the current implementation stores the parameters internally.

So type annotations are less central to this work than I expected. This is largely because **generators — which are particularly important for collections — do not allow for type annotations**. Which makes it difficult to extend ABCs with annotations in a consistent way.

The significance of the need for dependent types, when describing `Rec()` with ABCs and type annotations, is debatable. While type specifications are expressed in the language this may not be a serious problem (dependent types can be implemented as Python functions), but it might constrain future options to improve efficiency.

It's also worth noting that annotations are obscured by function decorators, although `functools.wraps` provides a `__wrapped__` attribute that can be used to chain to the original function.

Named Tuples, ABC Granularity

Named tuples are interesting because they so closely correspond to product types. Yet they are "bolted on" to the language and do not support type annotations. They also, confusingly, relate a `Rec()` over integer keys to an `Atr()` over different attribute names; more useful would be a relationship using the same names (ie. as between an object and the underlying dictionary). `Pytyp` provides `record()` for this.

A related issue is seen in the granularity of existing ABCs: **there is no abstraction between `Container` and `Mapping` / `Sequence` for `__getitem__()` and `__setitem__()`**. This muddies the connection between existing ABCs and product types.

Mutability

The idea of mutability in Python becomes more nuanced with the possibility of collections that have fixed types.

Mutability of an *individual* value in a collection is not addressed by the schema described here. In practice, Python's `tuple` type is immutable and can be used for both `Seq()` and `Rec()` (integer labels), while `namedtuple` also supports `Atr()`.

Type specifications do constrain the *type* or *number* of values in a container. User defined classes can support mutable values, while keeping fixed types, by verifying the input types. `Pytyp` provides the `checked` decorator to enforce type annotations. Curiously, **Python does not have a mutable collection of fixed size**. Again, `record` provides this in `pytyp`.

Registration of instances by `TSMeta` uses the Python hash. Strictly, only the number and type of the contents (and not the values themselves) should be used. But requiring a separate hash for types is over–ambitious.

More generally, functional programming suggests that accurately tracking mutability is important, but the runtime information for mutable types in Python is muddled: `Sequence` and

MutableSequence are distinguished by the *addition* of __setitem__(); the behaviour of mutable structures in Python depends on the *absence* of __hash__() and __eq__(). The pytyp library emphasises the latter; Seq is an ugly amalgam of the two ABCs that switches to structural verification when registration is impossible (ie. for unhashable instances).

Copy on write data structures[19][20] suggest an interesting way to address this issue. Their nature makes it easy to detect and record mutation. So hashing of mutable structures could be allowed, but "immutable references", in a similar way to weak references, would expire when the data change. This would remove, or at least reduce, the need for inefficient, structural verification of types.

AttributeError is a TypeError

In the context of duck typing, AtrributeError **should be a subclass of** TypeError. Or vice–versa?

Additional Issues in Pytyp

Efficiency

The issues above also affect pytyp. In addition, as with any pure–Python solution, there is a question of efficiency. For the occasional type check when debugging this is not an issue, but some of the features described are unsuitable for use across a Python application (eg. ubiquitous verification of type annotations).

How could performance be improved if some functionality was moved to the language run–time? What would minimal support require? Perhaps caching would be simplified by allowing arbitrary tags on (all) values? Is there a need for an intermediate conceptual level, between instances and types, that is somehow related to state? Are there useful parallels between type verification and the "unexpected path" handling of a JIT compiler?

Not a Type System

Pytyp **is not a type system; it does not support static reasoning about program correctness.** It is *only* a library for expressing and interpreting metadata at run–time. This fits within the Python ethos, but means, for example, that inconsistencies and errors are not flagged to the user, nor is the current type known for a value that has several alternatives (sum types). The last point implies that **type–guided iteration over data requires backtracking when inconsistencies are found.**

One way to move pytyp closer to a type system would be to add type inference. This could be a function, called at runtime, that uses type annotations to connect different type specifications together. For example, it could answer questions like "if I call function X with types Y and Z, what will the type of the result be?" The additional information Y and Z may help constrain the type of the result (resolving sum types, for example).

[19]http://pypi.python.org/pypi/blist

[20]http://www.python.org/dev/peps/pep-3128/

Negative Cache

ABCMeta contains both a class register and a negative cache (the cache tracks classes that are known not to be subclasses). TSMeta is a minimal extension of that code, which adds a register for instances, but does not include a corresponding negative cache. It is possible that a more careful implementation would be more efficient.

Inheritance, Types as Sets

No attempt is made to resolve multiple inheritance of type specifications. And() will merge the structural verification, so inheriting from And(X,Y) is preferable to subclassing both X and Y separately[21].

It has already been noted (in Types as Sets) that Or() is very close in meaning to Alt(). Since And() is similar to inheritance it may be better to drop both. This would simplify the library, but make it harder to use: the DSL approach to describing data is compact and readable; requiring the user to define new classes instead of writing And() would make it much more intrusive.

Acknowledgments

Thanks to Matthew Willson for useful comments.

Appendix: Further Details

Abbreviations and Normalisation

Pytyp supports the "abbreviated" syntax described above, but the normalize() function may be necessary when used in contexts that require a subclass of type:

```
>>> isinstance([1,2,3], normalize([int]))
True
>>> normalize([int, str])
Rec(int,str)
```

Optional Records

Optional records can be specified with a leading double underscore[22], which can be useful mapping between dict and function parameters (default values make certain names optional):

```
>>> isinstance({'a':1}, Rec(a=int, __b=str))
True
>>> isinstance({'a':1, 'b':'two'},
...             Rec(a=int, __b=str))
True
```

Similarly, a double underscore with no following name indicates a default type for additional values:

[21]The same logic might be implemented in the TSMeta metaclass.

```
>>> isinstance({'num':42, 'a':'foo', 'b':'bar'},
...            Rec(num=int, __=str))
True
```

To avoid syntax–related restrictions, `Rec()` can take a `dict` as a direct argument, via the `_dict` parameter. `Rec.OptKey()` can then mark optional records:

```
>>> isinstance({1:1},
...            Rec(_dict={1:int, Rec.OptKey('b'):str}))
True
```

Class and Attributes Shorthand

The `Cls()` constructor provides a shorthand for specifications that include a class and attributes:

```
>>> class Foo:
...     def __init__(self, x):
...         self.x = x
>>> isinstance(Foo(1), Cls(Foo, x=int))
True
>>> isinstance(Foo('one'), Cls(Foo, x=int))
False
>>> Cls(Foo, x=int)
And(Cls(Foo),Atr(x=int))
```

Circular References

These are defined using `Delayed()` which allows references to a type before it is known:

```
>>> d = Delayed()
>>> d.set(Alt(int, d, str))
>>> d
Delayed(Alt(int,...,str))
```

`Isinstance()` will raise a `RecursiveType` exception on recursive verification of a recursive type; typically this is handled by backtracking in `Alt()`.

Record

In a similar manner to `namedtuple()`, the function `record()` extends `dict` to construct classes that implement both `Rec()` and `Atr()`, providing unified access to named values:

```
>>> Simple = record('Simple', 'a,b,c=3')
>>> simple = Simple(1,'two')
>>> simple.b
'two'
>>> simple['c']
3

>>> Typed = record('Typed', 'a:int,b:str', mutable=True)
>>> typed = Typed(1, 'one')
>>> typed.a = 2
>>> typed['a']
2
>>> typed.b = 3
Traceback (most recent call last):
  ...
TypeError: Type str inconsistent with 3.

>>> StrTuple = record('StrTuple', ':str,:str')
>>> stuple = StrTuple('foo','bar')
>>> stuple[0]
'foo'
>>> stuple._1
'bar'
```

[22]It is hard to find something that is readable, an acceptable parameter name, and unlikely to clash with existing code.