Thinking about Type Checking

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

A common complaint levied against Python (and other similar languages) is the dynamic nature of its type handling. Dynamic typing makes it difficult to optimize performance because code can't be compiled in the same way that it is in languages like C or Java. The lack of explicitly stated types can also make it difficult to figure out how the parts of a large application might fit together if you're simply looking at them in isolation. This difficulty also applies to tools that might analyze or try to check your program for correctness.

If you're using Python to write simple scripts, dynamic typing is not something you're likely to spend much time worrying about (if anything, not having to worry about types is a nice feature). However, if you're using Python to write a larger application, type-related issues might cause headaches. Sometimes programmers assume that these headaches are just part of using Python and that there isn't much that they can do about it. Not true. As an application developer, you actually have a variety of techniques that can be used to better control what's happening with types in a program. In this installment, we explore some of these techniques.

Dynamic Typing

To start, consider the following function:

```
def add(x, y):
    return x + y
```

In this function, there is nothing to indicate the expected types of the inputs. In fact, it will work with any inputs that happen to be compatible with the + operator used inside. This is dynamic typing in action. For example:

```
>>> add(2, 3)
5
>>> add('two', 'three')
'twothree'
>>> add([1,2], [3,4,5])
[1, 2, 3, 4, 5]
>>>
```

This kind of flexibility is both a blessing and curse. On one hand, you have the power to write very general-purpose code that works with almost anything. On the other hand, flexibility can introduce all sorts of strange bugs and usability problems. For instance, a function might accidentally "work" in situations where it might have been better to raise an error. Suppose, for example, you were expecting a mathematical operation, but strings got passed in by accident:

```
>>> add('2', '3')
'23'
>>>
```

You might look at something like that and say "but I would never do that!" Perhaps, but if you're working with a bunch of Web coders, you might never know what they're going to pass into your program. Frankly, it could probably be just about anything, so it's probably best to plan for the worst. I digress.

The lack of types in the source may make it difficult for someone else to understand code—especially as it grows in size and you start to think about the interconnections between components. As such, much of the burden is placed on writing good documentation strings—at least you can describe your intent to someone reading the source and hope for the best:

```
def add(x, y):
    '''
    Adds the numbers x and y
    '''
    return x + y
```

You might be inclined to explicitly enforce or check types using `isinstance()`. For example:

```
def add(x, y):
    '''
    Adds the numbers x and y
    '''
    assert isinstance(x, (int, float)), 'expected number'
    assert isinstance(y, (int, float)), 'expected number'
    return x + y
```

However, doing so typically leads to ugly non-idiomatic code and may make the code unnecessarily inflexible. For example, what if someone wants to use the above function with `Decimal` objects? Is that allowed?

```
>>> from decimal import Decimal
>>> x = Decimal('2')
>>> y = Decimal('3')
>>> add(x, y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in add
AssertionError: expected number
>>>
```

Alternatively, you might see a function written like this:

```
def add(x, y):
    '''
    Adds the integers x and y
    '''
    return int(x) + int(y)
```

This function will attempt to coerce whatever you give it into a specific type. For example:

```
>>> add(2, 3)
5
>>> add('2', '3')
5
>>> add('two', 'three')
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'two'
>>>
```

This too might have bizarre problems. For example, what if floats are given?

```
>>> add(2.5, 3.2)
5
>>>
```

Alas, the function runs but silently throws away the fractional part of the inputs. If that's what you expected, great, but if not, then you have a whole new set of problems to worry about. Needless to say, it can get complicated.

Do type-related issues really matter in real applications? Based on my own experience, I'd answer yes. As a developer, you often try to do your best in writing accurate code and in writing tests. However, if you're working on a team, you might not know every possible way that someone will interact with your program. As such, it can often pay to take a defensive posture in order to identify problems earlier rather than later. Frankly, I often think about such matters solely as a way to prevent myself from creating bugs.

Having better control over type handling in Python is mostly solved through techniques that add layers to objects and functions. For example, using properties to wrap instance attributes or using a decorator to wrap functions [4]. The next few sections have a few examples.

Managing Attribute Types on Instances

Suppose you have a class definition like this:

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

By default, the attributes of `Stock` can be anything. For example:

```
>>> s = Stock('IBM', 50, 91.1)
>>> s.shares = 75
>>> s.shares = '75'
>>> s.shares = 'seventyfive'
>>>
```

However, suppose you wanted to enforce some controls on the shares attribute. One approach is to define shares as a property. For example:

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property
    def shares(self):
        'Getter function. Return the shares attribute'
        return self.__dict__['shares']

    @shares.setter
    def shares(self, value):
        'Setter function. Set the shares attribute'
        assert isinstance(value, int), 'Expected int'
        self.__dict__['shares'] = value
```

A property is a pair of get/set functions that captures the dot (.) operator for a specific attribute. In this case, all access to the shares attribute routes through the two functions provided. These two functions merely access the underlying instance dictionary, but the setter has been programmed to make sure the value is a proper integer. The resulting class works in exactly the same way as it did before except that there is now type checking on shares:

```
>>> s = Stock('IBM', 50, 91.1)
>>> s.shares = 75
>>> s.shares = '75'
Traceback (most recent call last):
...
AssertionError: Expected int
>>>
```

The verbose nature of writing out code for a property is a bit annoying if you have to do it a lot. Thus, if type checking is something you might reuse in different contexts, you can actually make a utility function to generate the property code for you. For example:

```
def Integer(name):
    @property
    def intvalue(self):
        return self.__dict__[name]

    @intvalue.setter
    def intvalue(self, value):
        assert isinstance(value, int), 'Expected int'
        self.__dict__[name] = value
    return intvalue

# Example
class Point(object):
    x = Integer('x')
    y = Integer('y')
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Here is an example of using the type-checked attribute:

```
>>> p = Point(2,3)
>>> p.x = 4
>>> p.x = '4'
Traceback (most recent call last):
...
AssertionError: Expected int
>>>
```

Alternatively, you can implement special type-checked attributes directly using a "descriptor" like this:

```
class Integer(object):
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        assert isinstance(value, int), 'Expected int'
        instance.__dict__[self.name] = value
```

A descriptor is similar to a property in that it captures the dot (.) operation on selected attributes. Basically, if you add an instance of a descriptor to a class, access to the attribute will route through the __get__() and __set__() methods. You would use the descriptor in exactly the same way the Integer() function was used in the above example.

Managing Types in Function Arguments

You can manage the types passed to a function, but doing so usually involves putting a wrapper around it using a decorator. Here is an example that forces all of the arguments to integers:

```
from functools import wraps

def intargs(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
      iargs = [int(arg) for arg in args]
      ikwargs = { name: int(val) for name, val in kwargs.items() }
       return func(*iargs, **ikwargs)
    return wrapper

# Example use
@intargs
def add(x, y):
    return x + y
```

If you try the resulting decorator, you'll get this behavior:

```
>>> add(2,3)
5
>>> add('2', '3')
5
>>> add('two', 'three')
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'two'
>>>
```

In practice, you might want to define a decorator that is a bit more selective in its type checking. Here is an example of applying type checks selectively to only some of the arguments. Note: This example relies on the use of the `inspect.signature()`, which was only introduced in Python 3.3 [1]. It will probably require a bit of careful study.

```
from functools import wraps
from inspect import signature

def enforce(**types):
   def decorate(func):
      sig = signature(func)
      @wraps(func)
      def wrapper(*args, **kwargs):
         bound_values = sig.bind(*args, **kwargs)
         for name, value in bound_values.arguments.items():
          if name in types:
             expected_type = types[name]
             assert isinstance(bound_values.
arguments[name], \
                 expected_type), '%s expected %s' \

                 % (name, expected_type.__name__)
       return func(*args, **kwargs)
    return wrapper
 return decorate

# Example use
@enforce(x=int, z=str)
def spam(x, y, z):
   pass
```

In this example, the decorator works by obtaining the function's calling signature. In the wrapper, the `sig.bind()` operation binds the supplied arguments to argument names in the signature. The code that follows then iterates over the supplied arguments, looks up their expected type (if any), and asserts that it is correct. Here is an example of how the function would work:

```
>>> spam(1, 2, 'hello')
>>> spam(1, 'hello', 'world')
>>> spam('1', 'hello', 'world')
Traceback (most recent call last):
...
AssertionError: x expected int
>>> spam(1, 'hello', 3)
Traceback (most recent call last):
...
AssertionError: z expected str
>>>
```

A Word on Assertions

In these examples, the `assert` statement has been used to enforce type checks. One special feature of `assert` is that it can be easily disabled if you run Python with the `-O` option. For example:

```
bash % python -O someprogram.py
```

When you do this, all of the asserts simply get stripped from the program—resulting in faster performance because all of the extra checking will be gone. This actually opens up an interesting spin on the type-checking problem. If you have an application that executes in both a staging and production environment, you can do things like enable type checks in staging (where you hope all of the code is properly tested and errors would be caught), but turn them off in production.

There is also a global `__debug__` variable that is normally set to `True`, but it changes to `False` when `-O` is given. You might use this to selectively disable properties. For example:

```
class Point(object):
   if __debug__:
      x = Integer('x')
      y = Integer('y')
   def __init__(self, x, y):
      self.x = x
      self.y = y
```

The Future: Function Annotations?

The future of type checking may lie in the use of function annotations. First introduced in Python 3, functions can be annotated with additional metadata. For example:

```
def add(x:int, y:int) -> int:
   return x + y
```

These annotations are merely stored as additional information. For example:

```
>>> add.__annotations__
{'return': <class 'int'>, 'x': <class 'int'>, 'y': <class 'int'>}
>>>
```

To date, the use of function annotations in practice has been somewhat scanty. However, projects such as mypy [2] have renewed interest in their possible use for type checking. For example, here is a sample function annotated in the style of mypy:

```
def average(values: List[float]) -> float:
   total = sum(values)
   return total / len(values)
```

A recent email posting from Guido van Rossum indicated a renewed interest in using annotations for type checking and in adopting the mypy annotation style in particular [3]. Standardizing the use of annotations for types would be an interesting development. It's definitely something worth watching in the future.

References

[1] https://www.python.org/dev/peps/pep-0362 (Function Signature Object).

[2] http://mypy-lang.org.

[3] https://mail.python.org/pipermail/python-ideas/2014-August/028618.html.

[4] "Python 3: The Good, the Bad, and the Ugly" explains decorators and function wrappers: https://www.usenix.org/publications/login/april-2009-volume-34-number-2/python-3-good-bad-and-ugly.