PEP 3124 - Overloading, Generic Functions, Interfaces, and Adaptation

Abeve Tayachow, Brandon Mikulka, Daniel Palmer

Introduction

There is no simple way to allow functions to deal with different argument types in Python. If there is a need to deal with varying arguments, the easiest tactic is with type checking using a number of if- statements. This is an inflexible solution. PEP 3124 proposes a standard library module that will implement dynamic overloading, which allows a function to react differently to distinct argument types, in essence introducing generic typing and interfacing to Python 3.0. Through use of decorators, we can easily control the ordering and selection of which code to run with certain inputs, making it much more versatile than the current implementation available in Python.

Overloading

One of the key aspects of the overloading api is the concept of the @overload decorator. When you have a function that could potentially take many types of arguments, the classic python way of dealing with this would be to create a series of if statements checking for the type of the argument. The @overload decorator allows you to redefine the function for a specific type as you can see in **Code block A**.

Code Block A: Example from PEP 3124

```
from overloading import overload
from collections import Iterable

def flatten(ob):
        """Flatten an object to its component iterables"""
        yield ob

@overload
```

```
def flatten(ob: Iterable):
        for o in ob:
                for ob in flatten(o):
                        yield ob

@overload
def flatten(ob: basestring):
        yield ob
```

Each following function checks for the type of the passed argument, and then executes the correct version of function based on the type. The code in **Code block A** would equate to the current python implementation in **Code block B**.

Code Block B: Example from PEP 3124

```
def flatten(ob):
        if isinstance(ob, basestring) or not isinstance(ob, Iterable):
                yield ob
        else:
                for o in ob:
                        for ob in flatten(o):
                                yield ob
```

Furthermore, the author Phillip J. Eby reccommends the use of a more general @when decorator for use when the function you are trying to overload is unbounded or bounded to a different namespace. In addition, the @when decorator will allow you to use a different function name for overriding by accepting a required function name and an optionional 'predicate' object to be passed into the decorator as shown in the last examples of **Code block C**.

Code Block C: Example from PEP 3124

```
from overloading import when

@overload
def flatten(ob: basestring):
        yield ob

@when(flatten)
def flatten(ob: basestring):
        yield ob

@when(flatten)
```

```
def flatten_basestring(ob: basestring):
        yield ob

@when(flatten, (basestring,))
def flatten_basestring(ob):
        yield ob
```

The main idea behind this functionality is to allow for expansion of code that you did not write, allowing for more generically typed functions. The most common use for this overloading, would be to accept parameters from an Api that doesn't necessarily give back the same object type each time it is called. This decorator would allow programmers to create an API that gives a uniform way to access functionality within libraries, frameworks, and applications, specifying what to do with what type of object.

A major objection to this implementation seems to be the lack of support for the concept of adding generic functions into python. Guido Van Rossen writes, "It's been excruciatingly hard to find anyone besides Phillip interested in GFs or able to provide use cases" (https://mail.python.org/pipermail/python-3000/2007-July/008831.html). Also, since according Eby the original reason for this Pep was to port the peak.rules.core library to Python 3.0 source code, there are problems with changes in the core of Python 3.0 that prevent this from being simple. Paul moore states the following:

> " The PEP states that the @overload decorator will work on any function, which requires in-place modification. By requiring overloadable functions to be declared somehow (for example, using a decorator) this requirement could possibly be removed." - Paul Moore (https://mail.python.org/pipermail/python-3000/2007-July/008829.html)

Combination and Overriding

If a function is being redeclared with overloading, we need a way to decide which to run and in which order they should do so. If a single method is to be used, the method that most accurately matches the calling parameters will be used, such as in **Code block D**.

Code Block D: Example from PEP 3124

```
def foo(bar:int, baz:object):
    pass

@overload
```

```python
def foo(bar:int, baz:int):
    pass
```

As you can see, one function takes an obj and an int as parameters, while the other takes two integers. If two integers are passed into the function, the second method will run as int, int is more specific than int, obj despite both technically being correct. If it is ambiguous which is more specific, an AmbiguousMethods error will be thrown.

In the case of ambiguous methods, order can be declared through use of the **proceed** function which simply returns a callable to the next-most specific method. In order to run certain methods every time a function is called, decorators such as @before, @after, and @around can be used. These are typically used for establishing pre-conditions and post-conditions. If a method is specified as @before, it will run before the primary function. An example of setting a pre-condition is shown in **code block E**, where @before is used to check if a database is available before the main function is called.

Code Block E: Example from PEP 3124

```python
def begin_transaction(db):
    print "Beginning the actual transaction"

@before(begin_transaction)
def check_single_access(db: SingletonDB):
    if db.inuse: raise TransactionError("Database already in use")
```

If there are multiple @before methods, they are run from most to least specific. To run methods after the primary function, the @after decorator is used. In contrast to @before, however, methods run after the primary function are ordered least to most specific. @around specifies methods to be run prior the first @before method. @around methods must use the **proceed** function to determine the next method to run. It will either return the next @around method, an error, or the first @before method. These can be used for things such as modifying input arguments that will impact future methods as well.

If a custom ordering of methods is required, we can use the always_overrides function. An example of this is shown in **code block F**.

Code Block F: Example from PEP 3124

```python
# discounts have precedence over before/after/primary methods
always_overrides(Discount, Before)
always_overrides(Discount, After)
always_overrides(Discount, Method)
```

It takes as input two methods, the first will override the second regardless of which is more specific, or any decorators prefixing them. While nearly all cases can be handled with the standard usage of decorators, it can be necessary to use this override.

There was some discussion that these proposed changes could cause code to become very cluttered and difficult to read, as it allows a function to be used in multiple different ways that aren't necessarily declared together. The counter-argument is that bad code can be written anyway, and if used responsibly this proposed change can add utility to the language. If this proposed change is implemented, it is very clear that we require this system of method ordering for there to be any way to control the methods.

Overloading Classes

Overloading can also be used inside classes for overriding methods and determining method order. In the example in **Code block G**, the class method foo is being overloaded in the parent class A and within the child class B.

Code Block G: Example from PEP 3124

```
class A(object):
    def foo(self, ob):
        print "got an object"

    @overload
    def foo(__proceed__, self, ob:Iterable):
        print "it's iterable!"
        return __proceed__(self, ob)


class B(A):
    foo = A.foo     # foo must be defined in local namespace

    @overload
    def foo(__proceed__, self, ob:Iterable):
        print "B got an iterable!"
        return __proceed__(self, ob)
```

Given the foo method being called with an Iterable inside class B, the order of method calls would print the following: "B got an Itterable," "it's iterable!," "got an object." This is because foo inside class B uses the **proceed** call to call the 'Iterable' version of the class inside class A. This function also has a **proceed** call which then calls the base method 'foo' in class A.

Overloading classes in this manner is in contention because of new implementation in Python 3.0. Eby states, “The way things currently stand for 3.0, I actually *won’t* be able to make a GF implementation that handles the ‘first argument should be of the containing class’ rule without users having an explicit metaclass or class decorator that supports it” (https://mail.python.org/pipermail/python-3000/2007-July/008784.html). This issue in implementation spawns from PEP3115, that “requires that a class’ metaclass be determined before the class has executed,” which makes porting the Python 2.0 version of this implementation in the peak.rules.core library to Python 3.0 impossible. This issue is ongoing and there is a large thread of discussion on this topic in the forum about how to overcome this issue in a pythonic way.

Interface and adaptation

The overloading module also provides a convenient implementation of interfaces and adaptation. The interface class accepts an object as its argument and binds all its methods to that object. It allows to define interfaces that are checked dynamically. Hence, There is no need to explicitly indicate when an object or a class implements a given interface, as it shown in **code block H**; Person class is the interface of Developer class.

Code Block H: https://github.com/ceronman/typeannotations

```
class Person(Interface):
    name = str
    age = int
    def say_hello(name: str) -> str:
            pass

class Developer:
    def __init__(self, name, age):
            self.name = name
            self.age = age
    def say_hello(self, name: str) -> str:
            return 'hello ' + name

isinstance(Developer('bill', 20), Person) #Returns True
```

A class “adapts” to an interface so long as the instance does not raise a NoApplicableMethods within the methods of the interface.

An interface class can have an empty generic function as well as a full executable function without the @abstract decorator. @abstract decorator creates an empty generic function that could be used within an interface class, but also could be used outside the class. For instance, in the **Code block I** IWriteMapping is an interface that has a **setitem** generic function with @abstract decorator as well as an update method with a layout of its implementation.

Code Block I: https://github.com/ceronman/typeannotations

```
class IWriteMapping(Interface):     @abstract

def __setitem__(self, key, value):
    """This has to be implemented"""

def update(self, other:IReadMapping):
    for k, v in IReadMapping(other).items():
        self[k] = v
```

In the case a different implementation is wanted from the update method, an appropriate overload can be used as previously discussed.

In Interface class definition, @abstract decorator can be added in order to ensure that the particular function cannot be accessed directly through the interface class, but only through a class that implements the interface template. Functions who have @abstract decorator overloaded methods must be added with the appropriate decorator (@when, @before, etc.) in order to make them executable. Unlike the previous overloaded function, the @abstract decorator function is designed to be overloaded, since it doesn't have any implementation to begin with.

Furthermore, interfaces have the ability to be used as type specifiers. When defining arguments in a function, interface subclasses (any defined interface inheriting from the Interface class) can be used as argument annotations. It should be noted, however, that the actual arguments themselves are not changed in any way from this annotation. Arguments will aso need to be cast in order to be given access to the the interfaces methods as shown in **Code block J**.

Code Block J: Example from PEP 3124

```
@overload
def traverse(g: IGraph, s: IStack):
    g = IGraph(g)
    s = IStack(s)
    # etc....
```

Conclusion

Implementing the changes suggested by this PEP would have numerous effects. We will have increased flexibility with generic functions that may take different types of input. Control is increased over multi-use functions, and generic functions will be easier to write when the intended usage is unknown. However, there have been concerns expressed about how this will affect readability of code. PEP 3124 allows for a function to be declared differently in separate areas of the code, potentially making it tough to accurately determine what a function actually does – an area of much contention. While this concern is valid, there are still ways of accomplishing this with practices such as monkeypatching and code substitution. This PEP's purpos is to allow for the ability to extend a library or application to work with data types the author might not have contemplated. If used responsibly, this proposal can add functionality to the language without major downsides. PEP 3124 currently has a status of `Deferred` because it is not yet completed and it is not heavily supported by the community.