

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

Abeve Tayachow, Brandon Mikulka, Daniel Palmer

Team Mikulka Palmer Tayachow

PEP 3124 - The Problem

- Written by Phillip J. Eby
- Dealing varying arguments
- Inflexible libraries/applications

The Curent Solution

```
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
```

The Proposed solution

- overload library with 4 key functionalities:
- Overloading
- Combination and Overriding
- Overloading Classes
- Interface and adaptation

The @overload decorator

```
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
```

More @overload decorator

- '@when' decorator - function is unbounded/bounded to different namespace (more general)
- Optional predicate object
- Creating generically typed functions
- Adapting APIs for uniform ways to access functionality

Discussion on @overload

- Lack of support in Python community
- Guido Van Rossen:
- “It’s been excruciatingly hard to find anyone besides Phillip interested in GFs or able to provide use cases”

Issues with Python 3.0

- `@overload` decorator should work on any function
- requires in-place modification
- Possible solution:
- Another decorator to declare overload-ability and Remove this necessity

Combination and Overriding

A new problem:

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

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

Ordering

- `__proceed__` function returns a callable to the next-most specific method
- Specific methods will still be run first
- `@before`, `@after`, and `@around` decorators

@before Example

```
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")
```

Discussion

- Readability of code difficult
- Cluttered code
- Counter argument:
- Monkeypatching and code substitution already exist
- Bad practices will still remain bad practices
- This give more flexibility with current available libraries

Overloading Classes

- Can be applied to classes and methods
- Ordering classes and determining method order

Class Overloading Example

```
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 defined in local namespace

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

Discussion on Overloading Classes

- Python 3.0 Problems
- Issue from PEP 3115:
- 'requires that a class' metadata be determined before execution
- Hard to port peak.rules.core library

Interface and Adaptation

- Generation of interface typing
- Interface typing checked dynamically
- “adapts” as long as the instance does not raise `NoApplicableMethods`

Interface Example

```
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
```

@abstract decorator

- Creates an empty generic function
- Used in conjunction with an interface class
- Also can be used outside a class

@abstract Example

```
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
```

More about @abstract and casting

- Methods must be overloaded with the appropriate @when or @overload decorator
- In other words: @abstract is designed to be overloaded
- Interfaces can be used as type specifiers

Conclusion

- Good or bad for community?
- Will it make messy code?
- Can it be used responsibly?
- Current status: Deferred
- Not quite supported by the Python community yet