generic Documentation

Release 1.0.0

Andrey Popp

May 15, 2020

Contents

Generic is trying to provide a Python programmer with primitives for creating reusable software components by employing advanced techniques of OOP and other programming paradigms.

This documentation suits both needs in a tutorial and an API reference for generic:

CHAPTER 1

Multidispatching

Multidispatching allows you to define methods and functions which should behave differently based on arguments' types without cluttering `if-elif-else` chains and `isinstance` calls.

All you need is inside `generic.multidispatch` module. See examples below to learn how to use it to define multifunctions and multimethods.

First the basics:

```
>>> class Cat: pass
>>> class Dog: pass
>>> class Duck: pass
```

1.1 Multifunctions

Suppose we want to define a function which behaves differently based on arguments' types. The naive solution is to inspect argument types with `isinstance` function calls but generic provides us with `@multidispatch` decorator which can easily reduce the amount of boilerplate and provide desired level of extensibility:

```
>>> from generic.multidispatch import multidispatch

>>> @multidispatch(Dog)
```

(continues on next page)

(continued from previous page)

```
... def sound(o):
...   print("Woof!")

>>> @sound.register(Cat)
... def cat_sound(o):
...   print("Meow!")
```

Each separate definition of `sound` function works for different argument types, we will call each such definition *a multifunction case* or simply *a case*. We can test if our `sound` multifunction works as expected:

```
>>> sound(Dog())
Woof!
>>> sound(Cat())
Meow!
>>> sound(Duck())
Traceback (most recent call last):
  ...
TypeError: No available rule found for ...
```

The main advantage of using multifunctions over single function with a bunch of `isinstance` checks is extensibility – you can add more cases for other types even in separate module:

```
>>> @sound.register(Duck)
... def duck_sound(o):
...   print("Quack!")
```

When behaviour of multifunction depends on some argument we will say that this multifunction *dispatches* on this argument.

1.1.1 Multifunctions of several arguments

You can also define multifunctions of several arguments and even decide on which of first arguments you want to dispatch. For example the following function will only dispatch on its first argument while requiring both of them:

```
>>> @multidispatch(Dog)
... def walk(dog, meters):
...   print("Dog walks for %d meters" % meters)
```

But sometimes you want multifunctions to dispatch on more than one argument, then you just have to provide several arguments to `multidispatch` decorator and to subsequent `when` decorators:

```
>>> @multidispatch(Dog, Cat)
... def chases(dog, cat):
...   return True

>>> @chases.register(Dog, Dog)
... def chases_dog_dog(dog1, dog2):
...   return None

>>> @chases.register(Cat, Dog)
... def chases_cat_dog(cat, dog):
...   return False
```

You can have any number of arguments to dispatch on but they should be all positional, keyword arguments are allowed for multifunctions only if they're not used for dispatch.

1.2 Multimethods

Another functionality provided by `generic.multimethod` module are *multimethods*. Multimethods are similar to multifunctions except they are... methods. Technically the main and the only difference between multifunctions and multimethods is the latter is also dispatch on `self` argument.

Implementing multimethods is similar to implementing multifunctions, you just have to decorate your methods with `multimethod` decorator instead of `multidispatch`. But there's some issue with how Python's classes works which forces us to use also `has_multimethods` class decorator:

```
>>> class Vegetable: pass
>>> class Meat: pass

>>> from generic.multimethod import multimethod, has_multimethods

>>> @has_multimethods
... class Animal(object):
...
...   @multimethod(Vegetable)
...   def can_eat(self, food):
...     return True
...
...   @can_eat.register(Meat)
...   def can_eat(self, food):
...     return False
```

This would work like this:

```
>>> animal = Animal()
>>> animal.can_eat(Vegetable())
True
>>> animal.can_eat(Meat())
False
```

So far we haven't seen any differences between multifunctions and multimethods but as it have already been said there's one – multimethods use `self` argument for dispatch. We can see that if we would subclass our `Animal` class and override `can_eat` method definition:

```
>>> @has_multimethods
... class Predator(Animal):
...   @Animal.can_eat.register(Meat)
...   def can_eat(self, food):
...     return True
```

This will override `can_eat` on `Predator` instances but *only* for the case for `Meat` argument, case for the `Vegetable` is not overridden, so class inherits it from `Animal`:

```
>>> predator = Predator()
>>> predator.can_eat(Vegetable())
True
>>> predator.can_eat(Meat())
True
```

The only thing to care is you should not forget to include `@has_multimethods` decorator on classes which define or override multimethods.

You can also provide a "catch-all" case for multimethod using `otherwise` decorator just like in example for multifunctions.

1.2.1 Providing "catch-all" case

There should be an analog to `else` statement – a case which is used when no matching case is found, we will call such case *a catch-all case*, here is how you can define it using `otherwise` decorator:

```
>>> @has_multimethods
... class Animal(object):
...
...   @multimethod(Vegetable)
...   def can_eat(self, food):
...     return True
...
...   @can_eat.register(Meat)
...   def can_eat(self, food):
...     return False
...
...   @can_eat.otherwise
...   def can_eat(self, food):
...     return "?"

>>> Animal().can_eat(1)
'?'
```

You can try calling `sound` with whatever argument type you wish, it will never fall with `TypeError` anymore.

1.3 API reference

generic.multidispatch.**multidispatch**(**argtypes*) → Callable[[T], generic.multidispatch.FunctionDispatcher[~T][T]]

Declare function as multidispatch

This decorator takes `argtypes` argument types and replace decorated function with *`FunctionDispatcher`* object, which is responsible for multiple dispatch feature.

generic.multimethod.**multimethod**(**argtypes*) → Callable[[T], generic.multimethod.MethodDispatcher[~T][T]]

Declare method as multimethod

This decorator works exactly the same as *`multidispatch()`* decorator but replaces decorated method with *`MethodDispatcher`* object instead.

Should be used only for decorating methods and enclosing class should have *`has_multimethods()`* decorator.

generic.multimethod.**has_multimethods**(*cls: Type[C]*) → Type[C]

Declare class as one that have multimethods

Should only be used for decorating classes which have methods decorated with *`multimethod()`* decorator.

class generic.multidispatch.**FunctionDispatcher**(*argspec: inspect.FullArgSpec*, *params_arity: int*)

Multidispatcher for functions

This object dispatch calls to function by its argument types. Usually it is produced by *`multidispatch()`* decorator.

You should not manually create objects of this type.

register(**argtypes*) → Callable[[T], T]
Decorator for registering new case for multidispatch

New case will be registered for types identified by `argtypes`. The length of `argtypes` should be equal to the length of `argtypes` argument were passed corresponding *`multidispatch()`* call, which also indicated the number of arguments multidispatch dispatches on.

class generic.multimethod.**MethodDispatcher**(*argspec: inspect.FullArgSpec*, *params_arity: int*)
Multiple dispatch for methods

This object dispatch call to method by its class and arguments types. Usually it is produced by *`multimethod()`* decorator.

You should not manually create objects of this type.

otherwise
Decorator which registeres "catch-all" case for multimethod

register(**argtypes*) → Callable[[T], T]
Register new case for multimethod for `argtypes`

CHAPTER 2

Event system

Generic library provides `generic.event` module which helps you implement event systems in your application. By event system I mean an API for *subscribing* for some types of events and to *handle* those events so previously subscribed *handlers* are being executed.

2.1 Basic usage

First you need to describe event types you want to use in your application, `generic.event` dispatches events to corresponding handlers by inspecting events' types, so it's natural to model those as classes:

```
>>> class CommentAdded(object):
...   def __init__(self, post_id, comment):
...     self.post_id = post_id
...     self.comment = comment
```

Now you want to register handler for your event type:

```
>>> from generic.event import Manager

>>> manager = Manager()

>>> @manager.subscriber(CommentAdded)
... def print_comment(ev):
...   print(f"Got new comment: {ev.comment}")
```

Then you just call `generic.event.handle` function with `CommentAdded` instance as its argument:

```
>>> manager.handle(CommentAdded(167, "Hello!"))
Got new comment: Hello!
```

This is how it works.

2.2 Event inheritance

2.3 Using per-application event API

2.4 API reference

class generic.event.**Manager**

Event manager

Provides API for subscribing for and firing events. There's also global event manager instantiated at module level with functions `subscribe()`, `handle()` and decorator `subscriber()` aliased to corresponding methods of class.

handle(*event: object*) → None

Fire `event`

All subscribers will be executed with no determined order.

subscribe(*handler: Callable[[object], None], event_type: Type[object]*) → None

Subscribe `handler` to specified `event_type`

subscriber(*event_type: Type[object]*) → Callable[[Callable[[object], None]], Callable[[object], None]]

Decorator for subscribing handlers

Works like this:

```
>>> mymanager = Manager()
>>> class MyEvent():
...     pass
>>> @mymanager.subscriber(MyEvent)
... def mysubscriber(evt):
...     # handle event
...     return
```

```
>>> mymanager.handle(MyEvent())
```

unsubscribe(*handler: Callable[[object], None], event_type: Type[object]*) → None

Unsubscribe `handler` from `event_type`

CHAPTER 3

Registry

CHAPTER 4

Installation

You can get generic by issuing *easy_install*:

```
% easy_install generic
```

or *pip* command:

```
% pip install generic
```

In case you find a bug or have a feature request, please file a ticket at GitHub Issues.

CHAPTER 5

Development process

Development takes place at GitHub, you can clone source code repository with the following command:

```
% git clone git://github.com/gaphor/generic.git
```

In case submitting patch or GitHub pull request please ensure you have corresponding tests for your bugfix or new functionality.

Index