exceptive Documentation

Release 0.1.8

Eray Erdin

Jul 22, 2018

Contents:

CHAPTER 1

Basics

1.1 Installation

You can either use `pip` to install the package:

```
pip install exceptive
# you can use pip3 explicitly if you also have Python 2 in your development␣
↪environment
```

Or simply download the package, extract it and use `setup.py`:

```
python3 setup.py build
python3 setup.py install
```

1.2 Simple Usage

You can use `catch` decorator to simply define the callback method to run in case of a particular exception occurs:

```
def typeerror_fallback_function(exception):
    print("Invalid input.")

@catch(TypeError, typeerror_fallback_function)
def greet(name):
    print("Hello "+name+"!")

greet("world")
# Hello world!

greet(5)  # int value raises TypeError when concatenated with str directly
# Invalid input.
```

CHAPTER 2

Decorator Module

2.1 The Approach

Decorator-based approach is a good choice;

- if you do not want to use inheritance on your classes
- if you want to implement `exceptive` on a simpler case, like a simple method

Decorators are much more flexible than inheritance, in which you can define custom object method to run and custom object method to run on exception.

2.2 catch

`catch`(*exception*, *method*[, **args*[, ***kwargs*]])

You can import the `catch` decorator from `exceptive.decorators` module.

`catch` is a decorator which you can apply on an independent method as below:

```
def invalid_input(exception):
    print("The input is invalid.")

@catch(TypeError, invalid_input)
def greet(name):
    print("Hello "+name+"!")

greet("world")
# Hello world!

greet(5)
# The input is invalid.
```

First you provide which exception to catch and then a callback method, which might be a proper method or a lambda.

Notice you have `exception` on callback method? That's how you can further analysis the thrown exception and provide further custom behavior.

You can also proive positional or keyword arguments for callback method on decorator. Here is a code sample:

```
def invalid_input(exception, default_value):
    print("Hello "+default_value+"!")

@catch(TypeError, invalid_input, default_value="world")
def greet(name):
    print("Hello "+name+"!")

greet("Eray")
# Hello Eray!

greet(5)
# Hello world!
```

Warning: This decorator is not suitable for object methods. For object methods, see `catch_object` decorator below.

Note: You can also use multiple `catch` decorator to handle multiple exception types.

2.3 catch_object

`catch`(*exception*[, *method=None*[, **args*[, ***kwargs*]]])

`catch_object` decorator is specifically designed for *object methods*. It looks up for callback method in the object-level.

You can import `catch_object` decorator from `exceptive.decorators`.

2.3.1 Providing Default Callback Method

When you provide `catch_object` decorator with `YourException`, `except__YourException` is called in case the exception is thrown.:

```
class Hello:
    @catch_object(TypeError)
    def greet(self, name):
        print("Hello "+name+"!")

    def except__TypeError(self, exception):
        print("Invalid value!")
```

2.3.2 Providing Custom Callback Method

Default lookup for callback method may generate stylistic warnings by your intellisense or linters. So, you can also provide the method name to look up for as a `str` to handle exception.:

```
class Hello:
    @catch_object(TypeError, "handle_type_error")
    def greet(self, name):
        print("Hello "+name+"!")

    def handle_type_error(self, exception):
        print("Invalid value!")
```

`method` argument also accepts ‘‘callable‘‘s, so that you can pass an independent method.:

```
def handle_type_error(exception):
    print("Invalid value!")

class Hello:
    @catch_object(TypeError, handle_type_error)
    def greet(self, name):
        print("Hello "+name+"!")
```

Note: `catch_object` decorator's parameters are quite similar to `catch` decorator's. So you can also provide your own `args` and `kwargs` to it.

Note: You can also define multiple `catch_object` on a single object method.

CHAPTER 3

Inheritance Module

3.1 The Approach

Inheritance-based approach is a good choice;

- if you want to isolate your functionality in one class
- if you want to standardize the exception-handling

3.2 MethodicExceptive

`MethodicExceptive` is a class with `__call__` method. All you have to do is to provide a `run` method and `except__YourException` on it.

You can import `MethodicExceptive` from `exceptive.inheritance` module and inherit it on your class.:

```
class Hello(MethodicExceptive):
    def run(self, name):
        print("Hello "+name+"!")

    def except__TypeError(self, exception):
        print("Invalid input!")

    # Initialize your object.
    obj = Hello()

    # Call it.
    obj("world")
    # Hello world!

    obj(5)
    # Invalid input!
```

You can also provide `except__else` to handle exception that are not provided as method.:

```
class Hello(MethodicExceptive):
    def run(self, name):
        print("Hello world!")

    def except__TypeError(self, exception):
        pass  # do something with TypeError

    def except__else(self, exception):
        pass  # do something with any other exception
```

Exceptive is a Python library that makes exception handling more programmatic and debuggable. You can define custom behaviors based on exceptions which might be for logging, validation or any other purpose you desire.

Currently, exceptive bases on two different approaches, you can;

- either use decorator-based module
- or use inheritance-based module

Both approaches have their downfalls and uprises and might depend on your choice of architectural design. To have further detail, see them in their own sections.

Index