Go-Flavored Error Handling in Python

Contents

Introduction

Errors are typically indicated by a non-zero integer value. Within a program/library, which is the main focus here, error return values are negative. For a program, error exit values are positive. In both cases, a zero value indicates success.

Integers do not carry much information. And what information they do carry often needs to be looked up in a table (e.g., what is `errno` 11?). On the other hand, a (good) string is explanatory and can be tailored to specific situations. This is why error messages are often output to stderr, even when a non-zero exit code is returned. But error strings are not just useful for end users. Even with libraries, error strings can be useful for logging. But this is more than a discussion of error strings or not.

One of the alternatives to integer-based errors in Python is to use exceptions. This can be useful, but still, it is an exception rather than just an "error". The exception vs error debate is not addressed here, but see LBYL and EAFP for more on Python's philosophy re: errors and exceptions.

The point of this article is to present an `Error` class in the spirit of Go error handling and consider its use/application in Python from a personal perspective.

Error Class

I have used the following in a number of projects (see repo for full implementation):

errors.py

```
#
# errors.py
#

import sys

class Error:
    """Base error object.
    """
    def __init__(self, msg, ecode=None):
        self.msg = msg
        self.ecode = ecode

    def __str__(self):
        if self.ecode == None:
            s = str(self.msg)
        else:
            s = "%s (ecode=%s)" % (self.msg, self.ecode)
        return s

    def __repr__(self):
        return """<Error msg="%s" ecode="%s">""" % (self.msg, self.ecode)

    def get_ecode(self, default=None):
        if self.ecode == None:
            return default
        else:
            return self.ecode

def __perror(erro):
    sys.stderr.write("%s\n" % str(erro))

def is_error(erro):
    return isinstance(erro, Error)

def perror(erro):
    """On Error, write string to stderr.
    """
    if is_error(erro):
        __perror(erro)

def perror_exit(erro, exitcode=1):
    """Call perror(). Exit on non-zero exitcode.
    """
    if is_error(erro):
        __perror(erro)
        if exitcode != 0:
            sys.exit(exitcode)
```

Notes:

- `Error` indicates an error situation
- `Error` provides an arbitrary, but useful, text description of the error
- `Error` provide an optional error code (should be an integer)
- `Error` can be passed around as an error object rather than an undifferentiated integer
- `is_error()` is succinct and clear label for test
- `perror_exit()` supports commonly used functionality

Use Cases

Subroutine

The most basic case is when a callable acts like a subroutine (no value is returned explicitly; in which case Python returns `None`):

```
erro = myfunc()
if erro:
    ...
```

This is easy to use because a `None` return value will always be treated as success, as anything else will be an `Error`.

Function

When a callable acts like a function, a value is always returned. In such a case, an `Error` might be one possible values.

```
val = myfunc()
if isinstance(val, Error):
        ...
```

This situation requires a bit more effort to use, but Python makes it easy because `val` can be any value, including `Error`.

This can also be written using `is_error()`:

```
val = myfunc()
if is_error(val):
    ...
```

`is_error()` cannot be a method of `Error` since `myfunc()` can return a non-`Error` value.

Multivalue Return

Python can return multiple values from a callable:

```
val, erro = myfunc()
if erro:
    ...
```

This form is very familiar to Go programmers and avoids the overloading of the single return value case. The main difference is that Go provides a more efficient way to write this:

```
if val, err = myfunc(); err !=  nil {
        ...
}
```

Double Assignment

It is also possible to use a "double assignment":

```
val = erro = myfunc()
if is_error(erro):
    ...
# use the result bound to "val"
```

There are some clear benefits of this approach:

1. the code is succinct
2. that the function/method returns a non-error or an error result is documented by the double assignment
3. the result (error or not) can be accessed using a suitably named variable, as the code warrants

The only potential drawback is the cost of the double assignment. But given the use cases of Python, this likely has minimal overall performance impact while the readability and understandability of the code increases.

Extending Error

Of course, the `Error` class can be extended to provide a collection of classes to more precisely indicate the type of error. E.g.,

```
class ErrorOverflow(Error):
    pass

class ErrorUnderflow(Error):
    pass
```

These can then be tested for using `isinstance()` to match `Error` or the specific subclasses.

```
erro = myfunc()
if isinstance(erro, ErrorOverflow):
        ...
elif isinstance(erro, ErrorUnderflow):
    ...
elif isinstance(erro, Error):
    # catch all
    ...
```

In the original article, `Error` was subclassed to create `ErrorCode` which took an exit code. This functionality has been incorporated into the base `Error` class as an error code.

Some Pros and Cons

One of the main drawbacks is that `Error` is unique to my package. It would be nice if there was a canonical `Error` (as there is an `Exception`). However, it may not be as much of a problem as it first appears. If errors from one layer need to be passed to another, a conversion between different, unrelated `Error` implementations can be done trivially:

```
erro = myfunc()
if erro:
    erro = other.Error(str(erro))
    ...
```

After all, the programmer should be aware of interfaces/interactions between layers and take care of checking and returning appropriate values. And, as presented, `Error` always provides a simple text string.

A perceived drawback is that this approach encourages a lot of error testing, a practice which Go programmers typically defend as being necessary for code to be good: errors should be checked.

The primary benefit of `Error` over a simple error code is that useful, contextual information can percolate up from anywhere in the code and, if appropriate, be used to inform the user. Too often, a helpful error code is converted to a generic error code and returned, thus losing information. Code-wise, the `Error` type is useful; people-wise, the text is useful.

Conclusion

Even in Python, exceptions are not always the best way to do things. Sometimes returning an error code or similar is appropriate. In such situations, wherever one lands on the question of using something like `Error` instead of integer return values for errors, the approach is worth considering. `Error` is not always appropriate, but it does make one think about the merits of how and why things are done as they are.

Changes

- 2019-02-20: the base `Error` class has been enhanced to support an error code
- 2019-01-26: added `is_error()` and `perror_exit()`; other updates
- 2018-01-20: renamed `err` to `erro` to avoid possible name conflict
- 2017-11-08: the section on double assignment was added.