Python Anti-Patterns

Python 2.x 2.3–2.7 , Python 3.x 3.0–3.6

Examples

Overzealous except clause

Exceptions are powerful, but a single overzealous except clause can take it all away in a single line.

```
try:
    res = get_result()
    res = res[0]
    log('got result: %r' % res)
except:
    if not res:
        res = ''
    print('got exception')
```

This example demonstrates 3 symptoms of the antipattern:

1. The `except` with no exception type (line 5) will catch even healthy exceptions, including `KeyboardInterrupt` . That will prevent the program from exiting in some cases.
2. The except block does not reraise the error, meaning that we won't be able to tell if the exception came from within `get_result` or because `res` was an empty list.
3. Worst of all, if we were worried about result being empty, we've caused something much worse. If `get_result` fails, `res` will stay completely unset, and the reference to `res` in the except block, will raise `NameError` , completely masking the original error.

Always think about the type of exception you're trying to handle. Give the exceptions page a read and get a feel for what basic exceptions exist.

Here is a fixed version of the example above:

```
import traceback

try:
    res = get_result()
except Exception:
    log_exception(traceback.format_exc())
    raise
try:
    res = res[0]
except IndexError:
    res = ''

log('got result: %r' % res)
```

We catch more specific exceptions, reraising where necessary. A few more lines, but infinitely more correct.

Looking before you leap with processor-intensive function

A program can easily waste time by calling a processor-intensive function multiple times.

For example, take a function which looks like this: it returns an integer if the input `value` can produce one, else `None` :

```
def intensive_f(value): # int -> Optional[int]
   # complex, and time-consuming code
   if process_has_failed:
       return None
   return integer_output
```

And it could be used in the following way:

```
x = 5
if intensive_f(x) is not None:
    print(intensive_f(x) / 2)
else:
    print(x, "could not be processed")

print(x)
```

Whilst this will work, it has the problem of calling `intensive_f` , which doubles the length of time for the code to run. A better solution would be to get the return value of the function beforehand.

```
x = 5
result = intensive_f(x)
if result is not None:
```

```
    print(result / 2)
else:
    print(x, "could not be processed")
```

However, a clearer and possibly more pythonic way is to use exceptions, for example:

```
x = 5
try:
    print(intensive_f(x) / 2)
except TypeError: # The exception raised if None + 1 is attempted
    print(x, "could not be processed")
```

Here no temporary variable is needed. It may often be preferable to use a `assert` statement, and to catch the `AssertionError` instead.

Dictionary keys

A common example of where this may be found is accessing dictionary keys. For example compare:

```
bird_speeds = get_very_long_dictionary()

if "european swallow" in bird_speeds:
    speed = bird_speeds["european swallow"]
else:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

with:

```
bird_speeds = get_very_long_dictionary()

try:
    speed = bird_speeds["european swallow"]
except KeyError:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

The first example has to look through the dictionary twice, and as this is a long dictionary, it may take a long time to do so each time. The second only requires one search through the dictionary, and thus saves a lot of processor time.

An alternative to this is to use `dict.get(key, default)` , however many circumstances may require more complex operations to be done in the case that the key is not present

Syntax

Parameters

Remarks