

PYTHON ANTI-PATTERNS

Computing

Written by : EL BAROUDI Marouane

Section 199.1: 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: The except with no exception type (line 5) will catch even healthy exceptions, including KeyboardInterrupt. 1. That will prevent the program from exiting in some cases. The except block does not reraise the error, meaning that we won't be able to tell if the exception came from 2. within get_result or because res was an empty list. Worst of all, if we were worried about result being empty, we've caused something much worse. If 3. 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.

Section 199.2: 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