- UsingAssertionsEffectively
- UsingAssert...Effectively
- FrontPage
- RecentChanges
- FindPage
- HelpContents
- UsingAssert...Effectively

Page

- Immutable Page
- Info
- Attachments
- More Actions: ▼

User

- Login

Python's `assert` statement helps you find bugs more quickly and with less pain. This note has some suggestions on good ways to use it.

Here are some observations about debugging:

- Practically all software has some bugs; it's a matter of frequency and severity rather than absolute perfection.
- The sooner you find a bug, the better: amongst other things, that it avoids wasting other people's time when they're bitten, and it makes schedules less likely to slip through extended debugging.
- When a bug does occur, you want to spend the minimum amount of time getting from the observed symptom to the root cause.

A few techniques can help shift the numbers in our favor, including good error logging, good testing, and internal self-checks (assertions). I wanted to write briefly about how assertions can help with Python code.

Assertions are a systematic way to check that the internal state of a program is as the programmer expected, with the goal of catching bugs. In particular, they're good for catching false assumptions that were made while writing the code, or abuse of an interface by another programmer. In addition, they can act as in-line documentation to some extent, by making the programmer's assumptions obvious. ("Explicit is better than implicit.")

Let me be explicit by giving an example. Suppose we have a simple database-like class which keeps an index from name to a numeric id, and from id to name.

Toggle line numbers

```
 1  class MyDB:
 2    def __init__(self):
 3      self._id2name_map = {}
 4      self._name2id_map = {}
 5
 6    def add(self, id, name):
 7      self._name2id_map[name] = id
 8      self._id2name_map[id] = name
 9
10    def by_name(self, name):
11      return self._name2id_map[name]
```

One of the "invariants" of this class is that the name->id map ought to always be in the inverse of the id->name map. You can see that the add() function tries to enforce that. However, it's possible that this invariant will be broken at some point: perhaps there is a typo in one of the methods, or perhaps some other bit of code gets in and fiddles with the private variables, etc. Perhaps it only happens for some values.

The symptoms are likely to be pretty confusing: code that uses the file will see results that are inconsistent. Perhaps what used to be a 1:1 map is no longer so.

A good place to add assertions to this class would be to check that the data-structure invariants are correct. For example:

Toggle line numbers

```
1    def by_name(self, name):
2      id = self._name2id_map[name]
3      assert self._id2name_map[id] == name
4      return id
```

This is not very hard to write or expensive to run, and follows naturally from the design of the class. It makes the programmer's assumptions clear to anyone else who works in the file in the future. If for any reason the code gets broken in the future, the failure will be obvious: the next time the key is accessed, the caller will get an `AssertionError` exception, which will typically go onto stderr or syslog.

Assertions are particularly useful in Python because of Python's powerful and flexible dynamic typing system. In the same example, we might want to make sure that ids are always numeric: this will protect against internal bugs, and also against the likely case of somebody getting confused and calling by_name when they meant by_id.

For example:

Toggle line numbers

```
1  from types import *
2  class MyDB:
3    ...
4    def add(self, id, name):
5      assert type(id) is IntType, "id is not an integer: %r" % id
6      assert type(name) is StringType, "name is not a string: %r" %
name
```

Note that the "types" module is explicitly "safe for import *"; everything it exports ends in "Type".

If you've been feeling (understandably) nervous about lack of parameter type checking in Python this can be a good way to address it. Don't make it too tight though: being able to flexibly use different types as appropriate is one of the strengths of the language.

You can also do this for classes, though the expression is a bit different.

Toggle line numbers

```
1  class PrintQueueList:
2    ...
3    def add(self, new_queue):
4      assert new_queue not in self._list, \
5        "%r is already in %r" % (self, new_queue)
6      assert isinstance(new_queue, PrintQueue), \
7        "%r is not a print queue" % new_queue
```

I realize that's not the exact way our function works but you get the idea: we want to protect against being called incorrectly. You can also see how printing the string representation of the objects involved in the error will help with debugging. ("How did a Kettle get there???"...)

Checking isinstance() should not be overused: if it quacks like a duck, there's perhaps no need to enquire too deeply into whether it really is. Sometimes it can be useful to pass values that were not anticipated by the original programmer.

Places to consider putting assertions:

- checking parameter types, classes, or values
- checking data structure invariants
- checking "can't happen" situations (duplicates in a list, contradictory state variables.)
- after calling a function, to make sure that its return is reasonable

The overall point is that if something does go wrong, we want to make it completely obvious as soon as possible.

It's easier to catch incorrect data at the point where it goes in than to work out how it got there later when it causes trouble.

Assertions are not a substitute for unit tests or system tests, but rather a complement. Because assertions are a clean way to examine the internal state of an object or function, they provide "for free" a clear-box assistance to a black-box test that examines the external behaviour.

Assertions should *not* be used to test for failure cases that can occur because of bad user input or operating system/environment failures, such as a file not being found. Instead, you should raise an exception, or print an error message, or whatever is appropriate. One important reason why assertions should only be used for self-tests of the program is that assertions can be disabled at compile time.

If Python is started with the -O option, then assertions will be stripped out and not evaluated. So if code uses assertions heavily, but is performance-critical, then there is a system for turning them off in release builds. (But don't do this unless it's really necessary. It's been scientifically proven that some bugs only show up when a customer uses the machine and we want assertions to help there too. )

CategoryDocumentation

UsingAssertionsEffectively (last edited 2014-05-24 22:52:34 by DaleAthanasias)