5.3 - Week 5 - Extra Reading - Assertions & Exceptions

Errors and Exceptions

The week 4 lesson says that exceptions are used to deal with 'errors', but assertions are to guard against 'mistakes'. What gives? How are they different?

Errors and Bugs

The errors we talk about here are 'error conditions' - abnormal (but not necessarily unforeseen) situations that prevent a piece of code from doing its normal job.

This happens a lot when code interacts with the outside world. Consider some code that loads data from a given filename. The code has a 'happy path': it opens the given filename, reads the data, and returns the result. But this happy path may fail in a number of ways: maybe the file doesn't exist, or the file is malformed.

This doesn't mean the *code* is incorrect, but it does mean the code can't return data as expected. Instead, we need to signal to the *calling* code that something has gone wrong, and it needs to deal with the problem. This is what we use *Exceptions* to do.

Historically, the main alternative to Exceptions is to add 'error' information to the value that is returned. Sometimes this is done by using a different type (e.g. Rust's Result type), by picking an otherwise invalid value to return (often `NULL` values or similar) or using a separate error flag (which needs to be checked after the call). In Python, we will sometimes see code like this:

```
some_value = some_call(arguments)
if some_value is None:
  # some_call failed, fill in a default value
  some_value = handle_special_case()
```

Exceptions were introduced in the early 1960s, and programmers have been arguing (frequently, often at volume) about which of these approaches is better better ever since.

Assertions

Assertions, instead, are for internal sanity checking. Large code-bases and complex data structures often have lots of hidden invariants -- internal properties that are meant to be maintained, or restrictions on how a method is supposed to be used.

We use assertions to check these properties: to make sure that whatever properties we're relying on actually hold. If an assertion fails, it means either (a) our code is incorrect, (b) our *mental model* of the code is incorrect (so our assertion is wrong), or (c) our code is being *used* incorrectly. Unlike Exceptions, failed assertions aren't something we try to deal with *inside* the code: instead, it means the code itself is wrong and should be changed.

In many cases (some) assertions are turned off when code is deployed. So if we build our code *expecting* assertions to fail (and be detected/handled), we might later have a nasty surprise.

Choosing between Assertions and Exceptions

There are cases which should clearly be enforced by assertions, such as checking internal invariants of data structures. And there are those which should obviously be exceptions, such as anything which interacts with the untrusted, poorly behaved, world beyond the interpreter.

But just like any other allocation of responsibility, things in-between can get fuzzier. A serious point to consider is the convention (or culture) of your code-base -- how will other people *expect* your code to behave?

Pythonic convention

Just like its type handling, Python code and libraries tend to follow a 'try it and see' approach, rather than 'look before you leap'. This means things that might be enforced as assertions in other languages (we expect the calling code to check the operation can be done first) are instead handled using exceptions in Python.

For example, in C++ the `vector::pop_back()` method *asserts* that the `vector` is non-empty (the caller is responsible for checking if the `vector` is empty beforehand). Whereas the Python `list` class raises an `IndexError` if `pop()` is called on an empty list (so popping an empty list isn't considered *incorrect* code, though it still needs to be handled).

One tradeoff here is performance: the Python approach must check emptiness every time `pop()` is called. But in many cases, the calling code *already knows* the vector is nonempty, so the check is wasted.

Example: Internal invariants

These are two functions from the Python `datetime` library:

```
    def _check_date_fields(year, month, day):
         year = _check_int_field(year)
         month = _check_int_field(month)
         day = _check_int_field(day)
         if not MINYEAR <= year <= MAXYEAR:
            raise ValueError('year must be in %d..%d' % (MINYEAR, MAXYEAR), year)
         if not 1 <= month <= 12:
            raise ValueError('month must be in 1..12', month)
         dim = _days_in_month(year, month)
         if not 1 <= day <= dim:
            raise ValueError('day must be in 1..%d' % dim, day)
         return year, month, day
```

```
    def _ymd2ord(year, month, day):
        "year, month, day -> ordinal, considering 01-Jan-0001 as day 1."
        assert 1 <= month <= 12, 'month must be in 1..12'
        dim = _days_in_month(year, month)
        assert 1 <= day <= dim, ('day must be in 1..%d' % dim)
        return (_days_before_year(year) +
                _days_before_month(year, month) +
                day)
```

Both perform the same tests, and both are internal methods. But `_check_date_fields` raises Exceptions where `_ymd2ord` uses assertions. Why are they different?

Let's check where they're used:

```
    class date:
       # ...
       def __new__(cls, year, month=None, day=None):
           # ...
           year, month, day = _check_date_fields(year, month, day)
           self = object.__new__(cls)
           self._year = year
           self._month = month
           self._day = day
           self._hashcode = -1
           return self

       # ...
       def toordinal(self):
           """Return proleptic Gregorian ordinal for the year, month and day.
           January 1 of year 1 is day 1.  Only the year, month and day values
           contribute to the result.
           """
```

```
            return _ymd2ord(self._year, self._month, self._day)
```

`toordinal()` is converting an existing `date` object to a new form. It _should_ already be valid (we checked validity in `__new__`!) so the assertions should never fail. If they do, it means (a) the code for the `date` object is wrong somehow, or (b) the user reached inside the `date` object and broke it. In either case, the *program* is broken and needs to be fixed.

Whereas `_check_date_fields` is constructing a `date` object from external input. It *expects* to be given a `(year, month, day)` as input, but it might have been given any old garbage. If we're given an invalid date, we probably want the *program* to continue executing, but the caller needs to do something (likely ask for a different, valid date).

Note that `_check_date_fields` doesn't say *what* to do if the date we're given is invalid -- the correct behaviour may be to ask for a valid date, or fill in today's date, or many other things. Here we just signal the error, and it's expect the caller's job to handle it appropriately.

Example: Control flow

Here is a snippet of code from the Python `doctest` library, which checks the outcome of a user-provided test case:

```
        # <Runs user's code, and sets outcome>
        # ...
        if outcome is SUCCESS:
            if not quiet:
               self.report_success(out, test, example, got)
        elif outcome is FAILURE:
            if not quiet:
               self.report_failure(out, test, example, got)
            failures += 1
        elif outcome is BOOM:
            if not quiet:
                self.report_unexpected_exception(out, test, example,
                                                 exception)
            failures += 1
        else:
            assert False, ("unknown outcome", outcome)
```

The code has already set `outcome` to one of three values (based on the behaviour of the test case), and is updating some internal state based on the result. If we ever reach the final `else` case, it means something is wrong with our code: maybe we added a new outcome type but forgot to update this section; or maybe there is a code path where we never set `outcome`. In either case, the testing code is broken.

If we look carefully at the other parts of the module, we can see there are two versions of `report_unexpected_exception`. One is for normal testing, which simply records the exception as another failure:

```
        def report_unexpected_exception(self, out, test, example, exc_info):
            """
            Report that the given example raised an unexpected exception.
            """
            out(self._failure_header(test, example) +
                'Exception raised:\n' + _indent(_exception_traceback(exc_info)))
```

But there is another for debugging which throws its own exception:

```
        def report_unexpected_exception(self, out, test, example, exc_info):
            raise UnexpectedException(test, example, exc_info)
```

Here we use an Exception because we want the program to continue executing -- the *testing code* is fine, but there's a problem with the code being tested that needs to be dealt with (perhaps by

reporting it to the user, or starting a debugger).

If you look carefully at the code for *running* the test, you may notice that the `try ... except` block is written in a slightly unusual way. This is because *testing* is a rare exception to our rule against catching failed assertions, and the testing code needs to bypass the mechanism which normally stops us from doing so.