Obey the Testing Goat!

Oft-heard is the folorn cry...

Every so often you get bitten by a weird behaviour in one of your Selenium tests. You tell it to click a link, and then you ask it something about the new page, and it returns you something from the old page:

```
old_value = browser.find_element_by_id('my-id').text
browser.find_element_by_link_text('my link').click()
new_value = browser.find_element_by_id('my-id').text
assert new_value != old_value ## fails unexpectedly
```

You scratch your head, and eventually conclude Selenium must be fetching the element from the old page. "Why would it do that?!", you exclaim in a programmer-rage, "In real life, when you click on a link, you see the browser starts to load a new page, and you wait for it to load, right? That's obviously what you'd want Selenium to do too, and it should be totally trivial to implement!"

> "Selenium should just wait until the page has completed loading after you click!"

```
browser.find_element_by_link_text('my link').click() # should just block until the next page has loaded
```

... with a sane timeout perhaps. There's even a `document.readyState` API for checking on whether a page has loaded! Grrr...

The thing is that, from the Selenium point of view, it's not that simple (and I'm grateful for David from Mozilla (@AutomatedTester) for patiently explaining this to me, more than once.)

You see, Selenium has no way of telling whether you've asked it to click on a "real" hyperlink that goes to a new URL, or whether the link goes to the same page, or whether the click is going to be intercepted by some sort of JavaScript to do some rich UI stuff on the same page.

More than that, since Selenium webdriver has become more advanced, clicks are much more like "real" clicks, which has the benefit of making our tests more realistic, but it also means it's hard for Selenium to be able to track the impact that a click has on the browsers' internals -- it might try to poll the browser for its page-loaded status immediately after clicking, but that's open to a race condition where the browser was multitasking, hasn't quite got round to dealing with the click yet, and it gives you the `.readyState` of the old page.

So, instead, Selenium does its best. The `implicitly_wait` argument will at least put a little retry loop in if you try and fetch an element that doesn't exist on the old page:

```
browser.implicitly_wait(3)
old_value = browser.find_element_by_id('thing-on-old-page').text
browser.find_element_by_link_text('my link').click()
new_value = browser.find_element_by_id('thing-on-new-page').text # will block for 3 seconds until thing-on-new-page appears
assert new_value != old_value
```

But the problem comes when `#thing-on-new-page` also exists on the old page. So what to do?

The "recommended" solution is an explicit wait:

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions

old_value = browser.find_element_by_id('thing-on-old-page').text
browser.find_element_by_link_text('my link').click()
WebDriverWait(browser, 3).until(
    expected_conditions.text_to_be_present_in_element(
        (By.ID, 'thing-on-new-page'),
        'expected new text'
    )
)
```

Several problems with that though:

1. HIDEOUSLY UGLY *
2. It's not generic -- even if I do write a nice wrapper, it's tedious to have to call it every time I click on a thing, specifying a different other thing to wait for each time
3. And it won't work for the case when I want to check that some text stays the same between page loads.

Really, I just want a reliable way of waiting until the page has finished loading after I click on a thing. I totally understand that David and pals aren't going to provide that for me by default because they can't tell what's a Javascript click and what's a click that goes to a new page, but I know. But how to do it?

Some things that won't work

The naive attempt would be something like this:

```
def wait_for(condition_function):
    start_time = time.time()
    while time.time() < start_time + 3:
        if condition_function():
            return True
        else:
            time.sleep(0.1)
    raise Exception(
        'Timeout waiting for {}'.format(condition_function.__name__)
    )


def click_through_to_new_page(link_text):
    browser.find_element_by_link_text('my link').click()

    def page_has_loaded():
        page_state = browser.execute_script(
            'return document.readyState;'
        )
        return page_state == 'complete'

    wait_for(page_has_loaded)
```

The `wait_for` helper function is good, but unfortunately `click_through_to_new_page` is open to the race condition where we manage to execute the script in the old page, before the browser has started processing the click, and `page_has_loaded` just returns true straight away.

Our current working solution

Full credit to @ThomasMarks for coming up with this: if you keep some references to elements from the old page lying around, then they will become stale once the DOM refreshes, and stale elements cause selenium to raise a `StaleElementReferenceException` if you try and interact with them. So just poll one until you get an error. Bulletproof!

```
def click_through_to_new_page(link_text):
    link = browser.find_element_by_link_text('my link')
    link.click()

    def link_has_gone_stale():
        try:
            # poll the link with an arbitrary call
            link.find_elements_by_id('doesnt-matter')
            return False
        except StaleElementReferenceException:
            return True

    wait_for(link_has_gone_stale)
```

Or, here's a genericised, sanitized version of the same thing, based on comparing Selenium's internal "IDs" for an object, and made into a nice Pythonic context manager:

[update 2014-09-06 -- see the comments, it's possible that comparing ids is not as efffective as waiting for stale reference exceptions. Will investigate, but bewarned that YMMV for now.]

```
class wait_for_page_load(object):

    def __init__(self, browser):
        self.browser = browser

    def __enter__(self):
        self.old_page = self.browser.find_element_by_tag_name('html')
```

```
    def page_has_loaded(self):
        new_page = self.browser.find_element_by_tag_name('html')
        return new_page.id != self.old_page.id

    def __exit__(self, *_):
        wait_for(self.page_has_loaded)
```

And now we can do:

```
with wait_for_page_load(browser):
    browser.find_element_by_link_text('my link').click()
```

And I think that might just be bulletproof!

And for bonus points...

(credit to Tommy Beadle for this solution)

It turns out selenium has a built-in condition called `staleness_of`, as well as its own wait-for implementation. Use them, alongside the `@contextmanager` decorator and the magical-but-slightly-scary `yield` keyword, and you get:

```
from contextlib import contextmanager
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support.expected_conditions import \
    staleness_of

class MySeleniumTest(SomeFunctionalTestClass):
    # assumes self.browser is a selenium webdriver

    @contextmanager
    def wait_for_page_load(self, timeout=30):
        old_page = self.browser.find_element_by_tag_name('html')
        yield
        WebDriverWait(self.browser, timeout).until(
            staleness_of(old_page)
        )

    def test_stuff(self):
        # example use
        with self.wait_for_page_load(timeout=10):
            self.browser.find_element_by_link_text('a link')
            # nice!
```

Note that this solution only works for "non-javascript" clicks, ie clicks that will cause the browser to load a brand new page, and thus load a brand new HTML body element.

Let me know what you think!