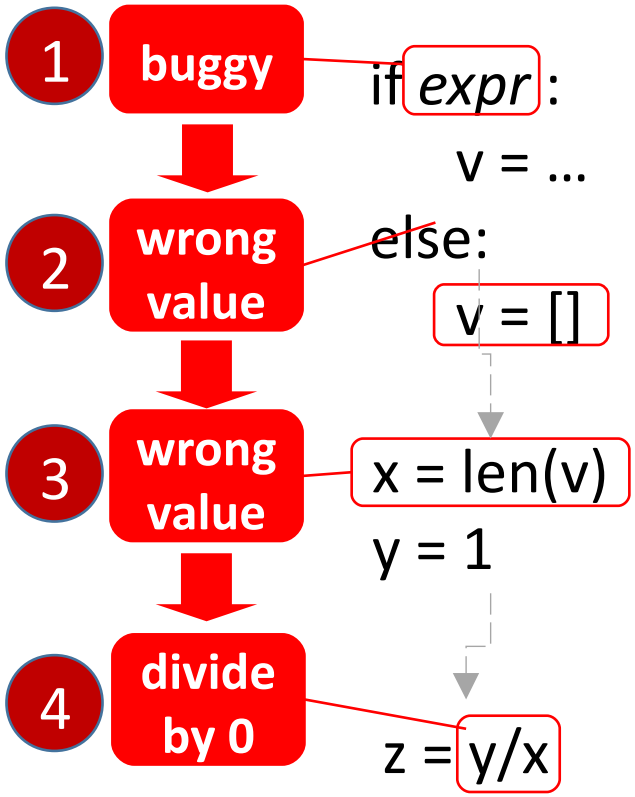
CSc 120
Introduction to Computer Programming II

`assert` and Invariants

Remember These?



Debugging deck
-
slide 21

Remember These?

```
def sumlist(L):
    print("Entered sumlist")
    sum = 0
    i = 0
    while i < len(L):
        i += 1
        sum += L[i]
    print("Leaving sumlist")
    return sum
```

```
> sumlist([1,2,3,4])
Entered sumlist

File "sumlist.py", line 9
        sum += L[i]
IndexError: list index out of range
```

Remember These?

```
def average(L):
      return sum(L)/len(L)
```

```
> x = [ ... ]
> average(x)

File "average.py", line 2
    return sum(L)/len(L)
ZeroDivisionError: division by zero
```

Debugging deck
-
slide 24

assert

- Wouldn't it be nice to have a way to detect errors earlier, when they were ***caused*** ?

- Wouldn't it be nice to be able to scan ***less*** code when you're debugging?

What If … ?

- Wouldn't it be nice to have a way to detect errors earlier, when they were ***caused*** ?
- Wouldn't it be nice to be able to scan ***less*** code when you're debugging?
- But often, we ***set*** a variable a long way away from where it is ***used*** . How to deal with this?

assert

- Many languages provide **assertions:** statements that you can make about what you believe to be true
- In Python, we do this with an `assert` statement:

```
x = ...
assert x > 0

y = foo(x)
assert y < 0
```

assert

Group Exercise:

We've already seen that the following code will sometimes hit a `ZeroDivisionError`. Add one or more `assert` statements to make the error easier to debug. Multiple answers are possible.

```
def average(L):
    return sum(L)/len(L)
```

assert

```
def average(L):
    assert len(L) > 0
    return sum(L)/len(L)
```

- or -

```
def average(L):
    assert L != []
    return sum(L)/len(L)
```

Discussion:

What are the tradeoffs between these two versions?

(They both are good, in different ways.)

Failed `assert`s

- What happens if an assertion **fails?**
  - Program crashes
  - Prints out an error report

Failed `asserts`

- What happens if an assertion **fails?**
  - Program crashes
  - Prints out an error report

```
Traceback (most recent call last):
  File "foo.py", line 13, in <module>
    print(asdf(x,list(range(10))))
  File "foo.py", line 2, in asdf
    assert len(a) == len(b)
AssertionError
```

Failed `asserts`

This is what went wrong.

```
Traceback (most recent call last):
  File "foo.py", line 13, in <module>
    print(asdf(x,list(range(10))))
  File "foo.py", line 2, in asdf
    assert len(a) == len(b)
AssertionError
```

Failed `assert`s

This is the assertion that failed.

```
Traceback (most recent call last):
  File "foo.py", line 13, in <module>
    print(asdf(x,list(range(10))))
  File "foo.py", line 2, in asdf
    assert len(a) == len(b)
AssertionError
```

Failed `assert`s

This is where that line of code resides.

```
Traceback (most recent call last):
  File "foo.py", line 13, in <module>
    print(asdf(x, list(range(10))))
  File "foo.py", line 2, in asdf
    assert len(a) == len(b)
AssertionError
```

Failed `asserts`

And the name of the function that contains that line.

```
Traceback (most recent call last):
  File "foo.py", line 13, in <module>
    print(asdf(x,list(range(10))))
  File "foo.py", line 2, in asdf
    assert len(a) == len(b)
AssertionError
```

Failed `asserts`

This is the code which called that function.

```
Traceback (most recent call last):
  File "foo.py", line 13, in <module>
    print(asdf(x,list(range(10))))
  File "foo.py", line 2, in asdf
    assert len(a) == len(b)
AssertionError
```

Why `assert` ?

- What does an **assertion** actually mean?
- Where should we write them?

```
... code ...

assert such_and_such

... code ...
```

Why `assert` ?

- Sometimes, an **assertion** works like a **comment**: it communicates what you believe to be true at a given point in the code.

```
if x > 0 and y < 0:
    ...
elif y > 0:
    ...
else
    ...
```

Group Exercise:

What **assertion** could you write about `x` in the `else` block of this code?

```
if x > 0 and y < 0:
    ...
elif y > 0:
    ...
else
    assert x >= 0
    ...
```

This **assertion** doesn't change the code at all; it doesn't introduce any new information.

But it communicates to **humans,** so that we can see the consequences of the code that already exists.

```
if x > 0 and y < 0:
    ...
elif y > 0:
    ...
else
    assert x >= 0
    ...
```

Group Discussion:

Is there any reason to use an **assertion** here?

Wouldn't a comment work just as well?

Tradeoffs

Assertions are Good:

- Checked at runtime, useful for debug
- Unambiguous

Assertions are Bad:

- Runtime cost
- Difficult to check complex conditions
  - “List is sorted,” “number is prime,” etc.

Why `assert` ?

- Sometimes, an **assertion** states a simplifying assumption, or an interface requirement.

```
def copy_list_contents(dst,src):
    for i in range(len(dst)):
        dst[i] = src[i]
```

Group Exercise:

This code is making some implicit assumptions about the parameters that it was passed. Add one or more `assert` statements to make these assumptions explicit.

Why `assert` ?

```
def copy_list_contents(dst,src):
    assert len(dst) == len(src)
    for i in range(len(dst)):
        dst[i] = src[i]
```

Using `assert`-as-assumption

Group Discussion:

What are some common situations where using an **assertion** to state an assumption might be good programming practice?

Using `assert`-as-assumption

Good assumptions to check

- Did the caller pass you valid parameters?
- “I think X is impossible.”
- “Here's a key design point...”

and...

https://xkcd.com/1421/

Using `assert`-as-assumption

Good assumptions to check

- Did the caller pass you valid parameters?
- “I think X is impossible.”
- “Here's a key design point...”

- TODO marks, known design flaws
  - If the code's broken and you know it, it's often better to crash controllably than do an ***unpredictable, wrong*** thing.

Using `assert`s for Debugging

- Assertions help debugging in two ways:
  - Detecting errors when they occur
  - Reducing the scope of debug searches

Using `assert`s for Debugging

Group Discussion:

In a previous example, we added an assert which simply summarized what we knew would be true at a given point in the code.

Suppose that this assertion fails. What should you look for in your debugging? How might this bug have come about?

```
        ...
    else
        assert x >= 0
        ...
```

Using `assert`s for Debugging

- If a “this is definitely true” assertion fails, then:
  - Maybe your original logic was faulty
  - Maybe you didn't consider all possible inputs

    - or -

  - Maybe you changed something, elsewhere, which broke your assumptions

Using `assert`s for Debugging

- One of the purposes of assertions is to make it ***easier to alter your program.***
  - Your assumptions are clear, unambiguous
  - Automatically checked when you run the code

- Useful for development/debugging
- **Especially useful** when updating old code!
  - Usually, you don't remember how it works!

Using `asserts` for Debugging

Group Discussion:

This was an example of an assertion that encodes an assumption you're making.

If this fails, what might be wrong with your program?

```
def copy_list_contents(dst,src):
    assert len(dst) == len(src)
    for i in range(len(dst)):
        dst[i] = src[i]
```

Using `assert`s for Debugging

- If an “I'm assuming this” assertion fails, then:
  - Maybe the caller did something wrong
    - (Did you write good documentation?)

    - or -

  - Maybe your function is too limited, needs to be improved

Using `asserts` for Debugging

- One of the purposes of assertions is to prevent your program from doing something ***disastrous.***

```
name = ...

assert name != "important.txt"
delete_file(name)
```

Using `asserts` for Debugging

- Another purpose of assertions is to prevent it from doing something ***silly.***

```
def build_house(num_doors):
    assert num_doors > 0
    ...
```

Using `assert`s for Debugging

- Can we use assertions to make debugging easier?
  - Reduce scope of searches
  - Break code into smaller, independent chunks

Debugging: working backwards

```
# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps
    fmt =
```

v = …

```
if fmt[0]                                   rmat list recursively
    fmt =
    fmt_list = fmt.split(",")
    rexp = self.gen_output_fmt(fmt_list)
else:
    if fmt[0] in "iI":  # integer
        sz = fmt[1:]
        gen_fmt = "{}"
        cvt_fmt = "{:" + str(sz) + "d}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]
```

while v > 0:

y = …

```
                                n of the tuple (corresponding to
                                the field can be arbitrarily wide
    rexp = [(gen_fmt, "{}", "*")]

    elif fmt[0] in "eEfFgG":  # various floating point formats
        idx0 = fmt.find(".")
        sz = fmt[1:idx0]
        suffix = fmt[idx0 + 1 :]
        # The 'E' and G formats can optionally specify the width of
                                '.  For now we ignore any such
                                 it's there, we need to extract
                                e it.
                                )
                                er format? '{fmt}'"
        prec = m.group(1)
        gen_fmt = "{}"
        cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]

    elif fmt[0] in "pP":  # scaling factor
        # For now we ignore scaling: there are lots of other things we
        # need to spend time on.  To fix later if necessary.
                        1:]
                        tput_fmt_1(rest_of_fmt)
```

x = y + z

print(x)

```
                         # character string
                         # -2 for the quote at either end
        # escape any double-quotes in the string
        gen_fmt = fmt[1:-1].replace('"', '\\"')
        rexp = [(gen_fmt, None, None)]

    fmt[0] == "/":  # newlines
    gen_fmt = "\\n" * len(fmt)
```

root cause: wrong value of v

wrong value printed

Debugging deck
-
slide 74

Debugging: working backwards

```
# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps
    fmt =
```

v = ...

```
if fmt[0]                                   rmat list recursively
    fmt =
    fmt_list = fmt.split(",")
    rexp = self.gen_output_fmt(fmt_list)
else:
    if fmt[0] in "iI":  # integer
        sz = fmt[1:]
        gen_fmt = "{}"
        cvt_fmt = "{:" + str(sz) + "d}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]
```

while v > 0:

y = ...

```
                                n of the tuple (corresponding to
                                the field can be arbitrarily wide
    rexp = [(gen_fmt, "{}", "*")]

    elif fmt[0] in "eEfFgG":  # various floating point formats
        idx0 = fmt.find(".")
        sz = fmt[1:idx0]
        suffix = fmt[idx0 + 1 :]
        # The 'E' and G formats can optionally specify the width of
                                '.  For now we ignore any such
                                 it's there, we need to extract
                                e it.
                                )
                                er format? '{fmt}'"
```

x = y + z

```
        prec = m.group(1)
        gen_fmt = "{}"
        cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]

    elif fmt[0] in "pP":  # scaling factor
        # For now we ignore scaling: there are lots of other things we
        # need to spend time on.  To fix later if necessary.
                        1:]
                        tput_fmt_1(rest_of_fmt)
```

print(x)

```
                         # character string
                         # -2 for the quote at either end
        # escape any double-quotes in the string
        gen_fmt = fmt[1:-1].replace('"', '\\"')
        rexp = [(gen_fmt, None, None)]

        fmt[0] == "/":  # newlines
        gen_fmt = "\\n" * len(fmt)
```

Debugging deck
-
slide 74

This is a lot of code to search. Wouldn't it be nice if we could reduce the number of lines to search?

Debugging: working backwards

```
# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps
    fmt =
```

v = ...

```
if fmt[0]                                   rmat list recursively
    fmt =
    fmt_list = fmt.split(",")
    rexp = self.gen_output_fmt(fmt_list)
else:
    if fmt[0] in "iI":  # integer
        sz = fmt[1:]
        gen_fmt = "{}"
        cvt_fmt = "{:" + str(sz) + "d}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]
```

while v > 0:

y = ...

```
                                n of the tuple (corresponding to
                                the field can be arbitrarily wide
    rexp = [(gen_fmt, "{}", "*")]

    elif fmt[0] in "eEfFgG":  # various floating point formats
        idx0 = fmt.find(".")
        sz = fmt[1:idx0]
        suffix = fmt[idx0 + 1 :]
        # The 'E' and G formats can optionally specify the width of
                                '.  For now we ignore any such
                                 it's there, we need to extract
                                e it.
                                )
                                er format? '{fmt}'"
```

x = y + z

```
        prec = m.group(1)
        gen_fmt = "{}"
        cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
                                t(sz))]
```

assert ...

```
                                actor
                                there are lots of other things we
        # need to spend time on.  To fix later if necessary.
                        1:]
                        tput_fmt_1(rest_of_fmt)
```

print(x)

```
                                # character string
                                # -2 for the quote at either end
        # escape any double-quotes in the string
        gen_fmt = fmt[1:-1].replace('"', '\\"')
        rexp = [(gen_fmt, None, None)]

    fmt[0] == "/":  # newlines
    gen_fmt = "\\n" * len(fmt)
```

Debugging deck
-
slide 74

This is a lot of code to search. Wouldn't it be nice if we could reduce the number of lines to search?

Let's add an `assert`, which checks to make sure `x` is valid...

Debugging: working backwards

```
# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps
    fmt =
```

v = ...

```
if fmt[0]                                   rmat list recursively
    fmt =
    fmt_list = fmt.split(",")
    rexp = self.gen_output_fmt(fmt_list)
else:
    if fmt[0] in "iI":  # integer
        sz = fmt[1:]
        gen_fmt = "{}"
        cvt_fmt = "{:" + str(sz) + "d}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]
```

while v > 0:

y = ...

```
                                n of the tuple (corresponding to
                                the field can be arbitrarily wide
    rexp = [(gen_fmt, "{}", "*")]
```

```
                                ous floating point formats
```

assert ...

```
    suffix = fmt[idx0 + 1 :]
    # The 'E' and G formats can optionally specify the width of
                                '.  For now we ignore any such
                                 it's there, we need to extract
                                e it.
                                )
                                er format? '{fmt}'"
```

x = y + z

```
    prec = m.group(1)
    gen_fmt = "{}"
    cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
                                t(sz))]
```

assert ...

```
                                actor
    # need to spend time on.  there are lots of other things we
                                To fix later if necessary.
                                1:]
                                tput_fmt_1(rest_of_fmt)
```

print(x)

```
                                # character string
                                # -2 for the quote at either end
    # escape any double-quotes in the string
    gen_fmt = fmt[1:-1].replace('"', '\\"')
    rexp = [(gen_fmt, None, None)]

    fmt[0] == "/":  # newlines
    gen_fmt = "\\n" * len(fmt)
```

Debugging deck
-
slide 74

This is a lot of code to search. Wouldn't it be nice if we could reduce the number of lines to search?

Let's add an `assert`, which checks to make sure `x` is valid...then one for `y`...

Debugging: working backwards

```
# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps
    fmt =
```

v = ...

```
if fmt[0]                                   rmat list recursively
    fmt =
    fmt_list = fmt.split(",")
                                        st)
```

assert ...

```
    gen_fmt = "{}"
    cvt_fmt = "{:" + str(sz) + "d}"
    rexp = [(gen_fmt, cvt_fmt, int(sz))]
```

while v > 0:

y = ...

```
                                n of the tuple (corresponding to
                                the field can be arbitrarily wide
    rexp = [(gen_fmt, "{}", "*")]
                                ous floating point formats
```

assert ...

```
    suffix = fmt[idx0 + 1 :]
    # The 'E' and G formats can optionally specify the width of
                                '.  For now we ignore any such
                                 it's there, we need to extract
                                e it.
                                )
                                er format? '{fmt}'"
```

x = y + z

```
    prec = m.group(1)
    gen_fmt = "{}"
    cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
                                t(sz))]
```

assert ...

```
                                actor
    # need to spend time on.  there are lots of other things we
                                To fix later if necessary.
                        1:]
                                tput_fmt_1(rest_of_fmt)
```

print(x)

```
                                # character string
                                # -2 for the quote at either end
    # escape any double-quotes in the string
    gen_fmt = fmt[1:-1].replace('"', '\\"')
    rexp = [(gen_fmt, None, None)]

    fmt[0] == "/":  # newlines
    gen_fmt = "\\n" * len(fmt)
```

Debugging deck
-
slide 74

This is a lot of code to search. Wouldn't it be nice if we could reduce the number of lines to search?

Let's add an `assert`, which checks to make sure `x` is valid...then one for `y`...and also one for `v`.

Debugging: working backwards

```
# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps
    fmt =
```

v = ...

```
if fmt[0]                                   rmat list recursively
    fmt =
    fmt_list = fmt.split(",")
                                        st)
```

assert ...

```
        gen_fmt = "{}"
        cvt_fmt = "{:" + str(sz) + "d}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]
```

while v > 0:

y = ...

```
                                    n of the tuple (corresponding to
                                    the field can be arbitrarily wide
        rexp = [(gen_fmt, "{}", "*")]
                                    us floating point formats
```

assert ...

```
        suffix = fmt[idx0 + 1 :]
        # The 'E' and G formats can optionally specify the width of
                                    '.  For now we ignore any such
                                     it's there, we need to extract
```

x = y + z

```
                                    e it.
                                    )
                                    er format? '{fmt}'"
        prec = m.group(1)
        gen_fmt = "{}"
        cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
                                    t(sz))]
```

assert ...

```
                                    actor
                                    there are lots of other things we
        # need to spend time on.  To fix later if necessary.
                                1:]
                                tput_fmt_1(rest_of_fmt)
```

print(x)

```
                                    # character string
                                    # -2 for the quote at either end
        # escape any double-quotes in the string
        gen_fmt = fmt[1:-1].replace('"', '\\"')
        rexp = [(gen_fmt, None, None)]

        fmt[0] == "/":  # newlines
        gen_fmt = "\\n" * len(fmt)
```

Debugging deck
-
slide 74

Now, if this program prints out the wrong value for `x`, you only need to check a small range of code.

Debugging: working backwards

```
# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps
    fmt =
```

v = ...

```
if fmt[0]                                   rmat list recursively
    fmt =
    fmt_list = fmt.split(",")
                                        st)
```

assert ...

```
        gen_fmt = "{}"
        cvt_fmt = "{:" + str(sz) + "d}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]
```

while v > 0:

y = ...

```
                                    n of the tuple (corresponding to
                                    the field can be arbitrarily wide
    rexp = [(gen_fmt, "{}", "*")]
```

assert ...

```
                                    us floating point formats
    suffix = fmt[idx0 + 1 :]
    # The 'E' and G formats can optionally specify the width of
                                    '.  For now we ignore any such
                                     it's there, we need to extract
                                    e it.
                                    )
                                    er format? '{fmt}'"
    prec = m.group(1)
    gen_fmt = "{}"
    cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
                                    t(sz))]
```

x = y + z

assert ...

```
                                    actor
                                    there are lots of other things we
    # need to spend time on.  To fix later if necessary.
                                    1:]
                                    tput_fmt_1(rest_of_fmt)
```

print(x)

```
                                     # character string
                                     # -2 for the quote at either end
    # escape any double-quotes in the string
    gen_fmt = fmt[1:-1].replace('"', '\\"')
    rexp = [(gen_fmt, None, None)]

    fmt[0] == "/":  # newlines
    gen_fmt = "\\n" * len(fmt)
```

Debugging deck
-
slide 74

Or maybe the assert about `x` failed?

You know that the assert about y passed, so you only have to check a small chunk.

Using `assert`s for Debugging

When debugging:

- Use `assert` statements as “starting assumptions” for a block of code
  - Even if you're not sure they are *always* true...
  - You know that the assertion didn't fail *this time.*

Using `assert`s for Debugging

But be warned:

- Not all “true things” are easy to check
  - Too complex
  - Too slow

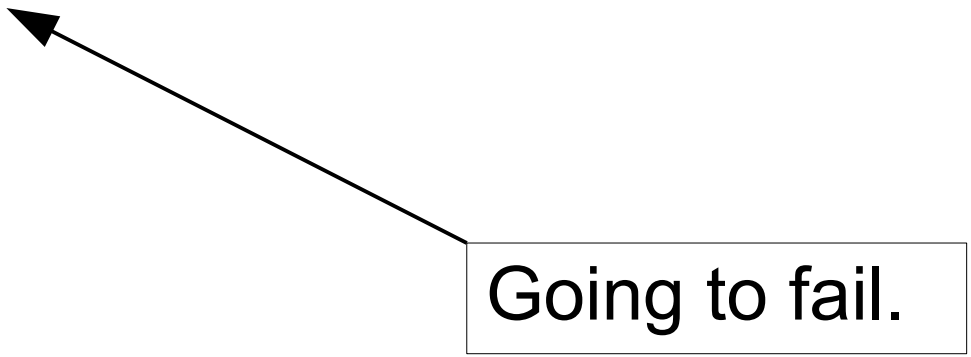
- Using assertions for debugging is *great when you can get it,* but doesn't work in all situations

More Detail...

Cool Trick:

- Python allows you to pass a value along with the check – if it fails, it can give you more debug detail

```
x = 2
assert x==3
```



More Detail...

Cool Trick:

- Python allows you to pass a value along with the check – if it fails, it can give you more debug detail

```
x = 2
assert x==3, x
```

Secondary info.

More Detail...

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: 3
```

```
x = 2
assert x==3, x
```

If assertion fails, it will print out the secondary info.

Using `assert`s for Coding

- Can we use assertions to write better code?

Some tricks:

- Use (cheap) assertions liberally
- Use comments when assertions not practical
- Break code into ***small logical chunks!***

Using `asserts` for Coding

```
... code ...
... code ...
... code ...

assert ... something about x ...
assert ... something about y ...
assert ... something about z ...

... code ...
... code ...
... code ...
```

We can mostly ignore this code...

...when we're writing this code.

(So long as the `assert`'s tell us enough.)