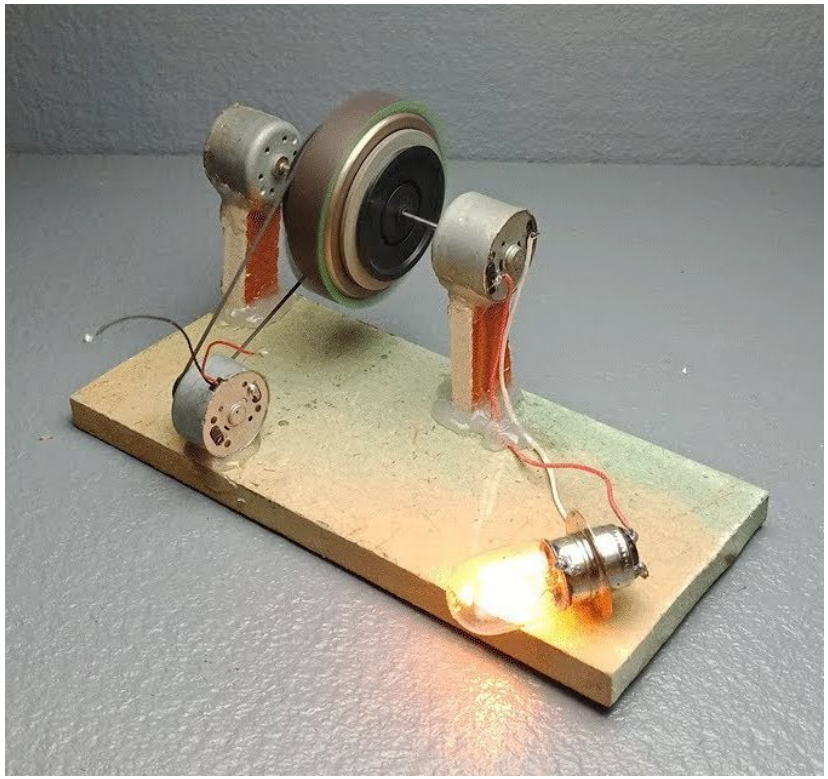
Python Generators.

The Idea,
a Test Case,
and a Subtle Bug.

Patrick Schemitz, solute GmbH
Karlsruhe Python Meetup 2020-02-19

solute

Table of Contents

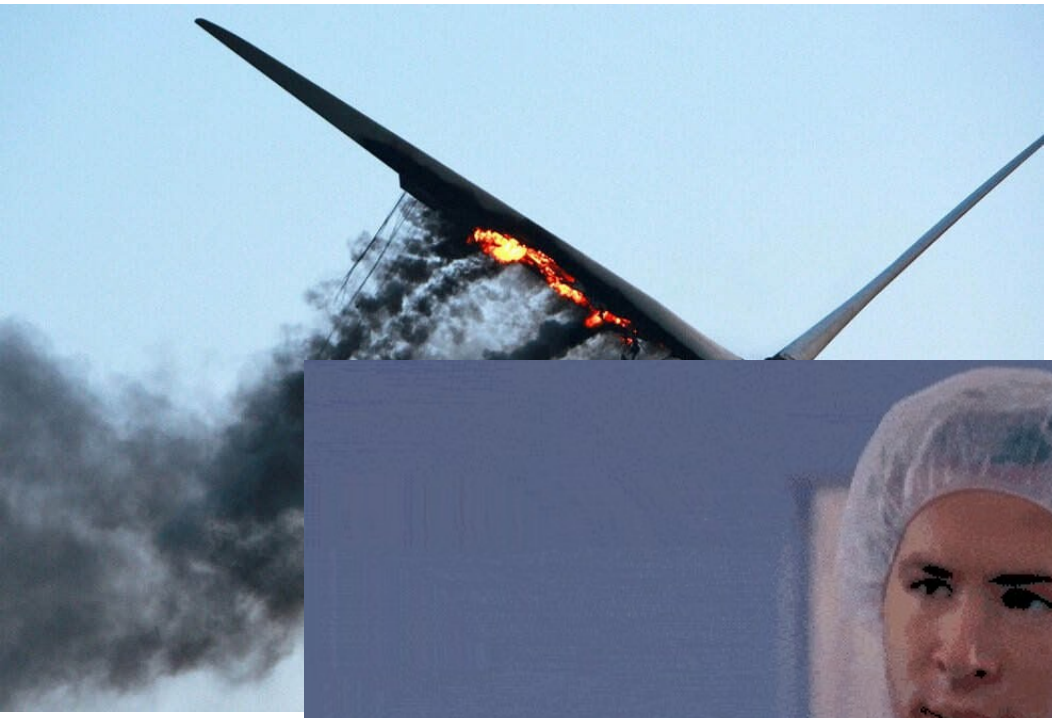


Prelude: A Subtle Bug...

```
def handle_unseen_exports():
    exports = get_exports()
    seen_exports = (
        ex["id"]
        for ex in exports
        if is_export_seen(ex["id"])
    )

    # ...

    for ex in exports:
        if ex["id"] in seen_exports:
            print("skipping seen export {}".format(ex))
            continue
        print("handling unseen export {}...".format(ex))
        if is_export_seen(ex["id"]):
            print("this should never happen")
```

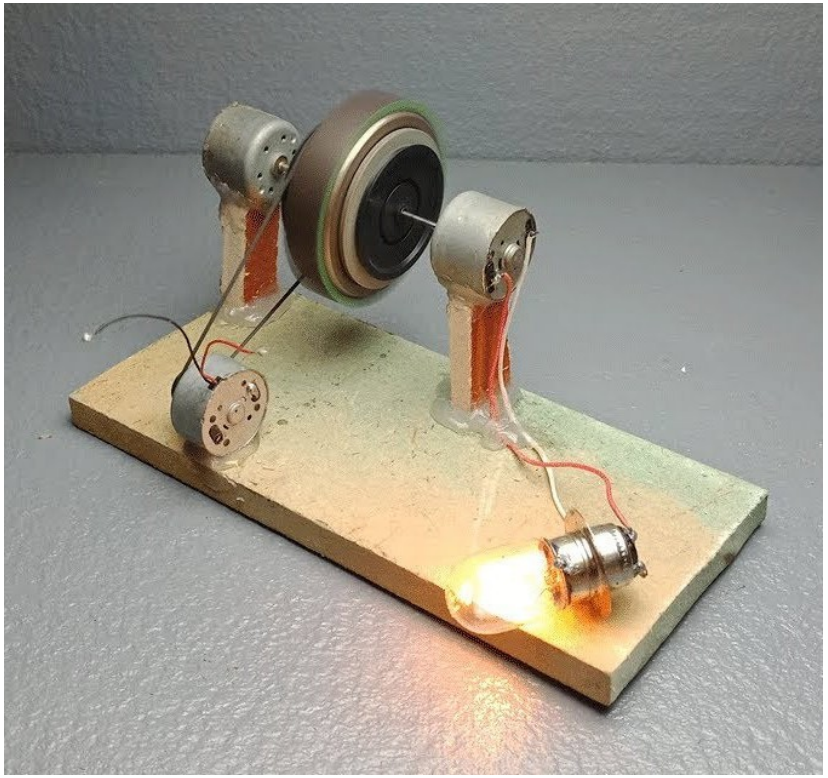
Prelude: A Subtle Bug...

Luckily this was on a staging system, not on production (yet)...



Generators

- Function that uses **`yield`** keyword
  - → Multiple exits
  - → Multiple entries
  - → Keeps state (locals, try)
- See PEP 255, 342



“Normal” function:

```
def multiply(a, b):
    result = a * b
    return result

c = multiply(a=3, b=5)
```

Generators vs. Functions

- Reflections on scopes, lifetime, and invokation/return:

```
def divide(a, b):
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        return math.inf

c = divide(a=3.0, b=2.0)
c = divide(a=3.0, b=0.0)
```

Generators vs. Functions

- Returning multiple values from a function:

```
def n_series(n):
    return [n*i for i in range(1, 11)]

def bounds(x):
    return math.floor(x), math.ceil(x)

for n in n_series(5):
    print(n)
```

Generators

- Function that uses **`yield`** keyword
  - → Multiple exits
  - → Multiple entries
  - → Keeps state (locals, try)

```
def n_series(n):
    for i in range(1, 11):
        yield i*n

for n in n_series(5):
    print(n)
```

- Inspect `n_series()` in >>> REPL

Generators

Calling `n_series()` returns a “generator object”:

- has `__iter__` and `__next__` methods
  - → implements iterator protocol (`next()`)
  - → fits with `for...in`
- has `send()` method?!
- has `throw()` method?!! o_O

```
def position():
    pos = 0
    while True:
        delta = yield pos
        if delta:
            pos += delta

p = position()
next(p)
p.send(5)
p.send(-2)
```

Generators

- Generator object `throw()` method raises exception in current `yield`:

```
def n_series(n):
    for i in range(1, 11):
        try:
            yield i*n
        except ValueError:
            print(“that’s weird”)

gen = n_series(5)
next(gen)
gen.throw(ValueError, “val”, None)
next(gen)
gen.throw(KeyError, “val”, None)
next(gen)
```

Uncaught exception escalates, StopIteration

Why Generators?

- Generators (as iterators) help save memory:

```
def n_series(n):
    return [n*i for i in range(1, 11)]
```

vs.

```
def n_series(n):
    for i in range(1, 11):
        yield i*n

def json_reader(f):
    for line in f:
        yield json.loads(line)
```

→ csv.reader()

Example: Spool Dir Watcher

```
def file_finder(spooldir, interval):
    while True:
        new_files = os.listdir(
            os.path.join(spooldir, "new")
        )
        for filename in sorted(new_files):
            os.rename(
                os.path.join(spooldir, "new", filename),
                os.path.join(spooldir, "cur", filename),
            )
            yield os.path.join(spooldir, "cur", filename), True
        if not new_files:
            time.sleep(interval)


for mail_file, rm in file_finder(“/var/spool/incoming”, 10):
    try:
        process_mail(mail_file)
        if rm: os.remove(mail_file)
    except:
        # keep defective mail for analysis
```

Spool Dir Watcher – The Why

```
if len(sys.args) > 1:
    files = [(f, False) for f in sys.argv[1:]]
else:
    files = file_finder(“/var/spool/incoming”, 10)

for mail_file, rm in files:
    try:
        process_mail(mail_file)
        if rm: os.remove(mail_file)
    except:
        # keep defective mail for analysis
```

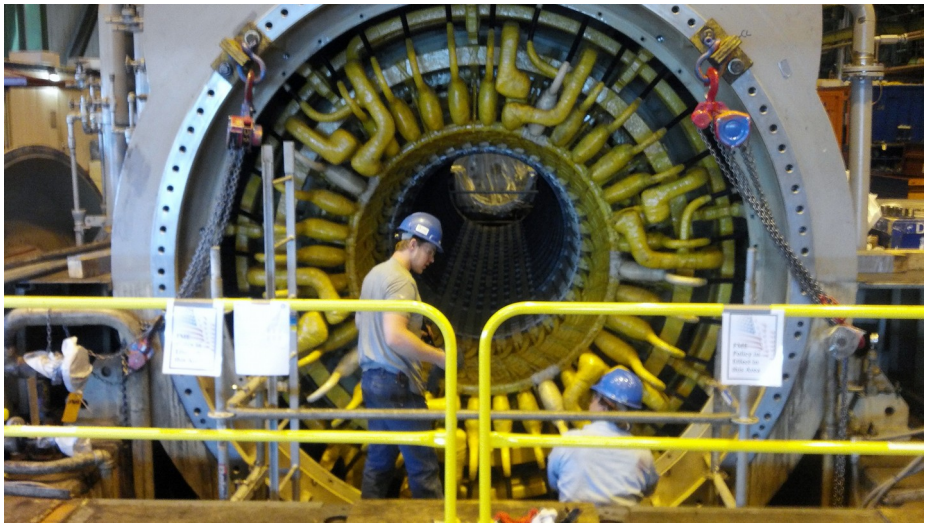
Generator Expressions

- Very much like list expressions written with ( ) instead of [ ]
- Saving all that memory...
- But with the hidden state:


```
docs = [{“id”: i} for i in range()]

docs = ({“id”: i} for i in range())

for doc in docs:
    print(doc)

for doc in docs:
    print(doc)
```

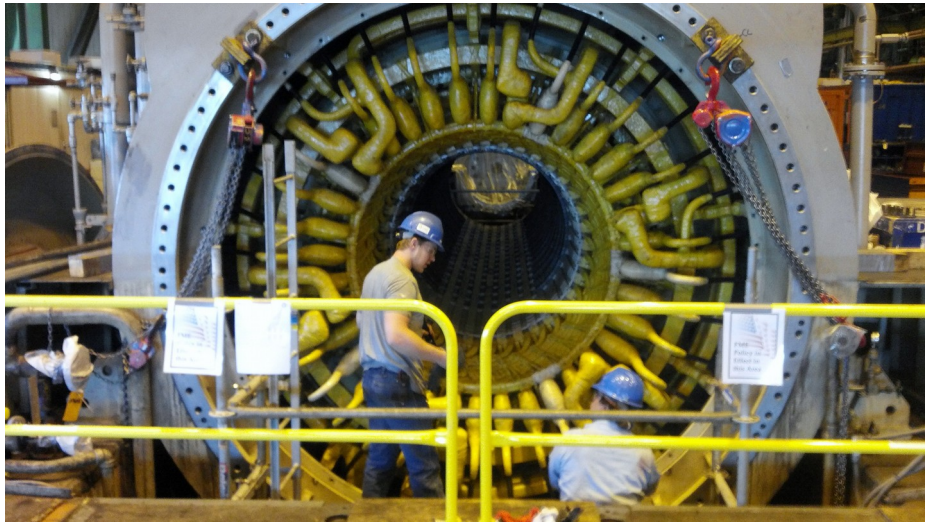
Testing a Generator

- Separate `<generator object>` creation and actual invokation: `gen = my_generator(foo, bar)`
- Use `next()` to step through generator
- Test for proper termination with `pytest.raises(StopIteration)`



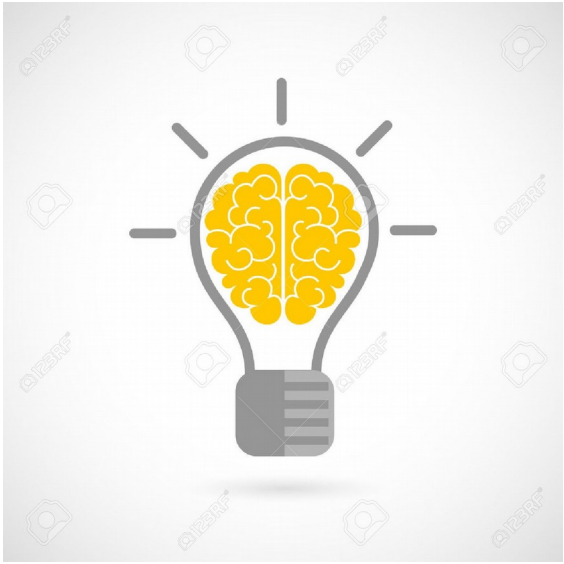
Testing a Generator

- Challenge: generators are sometimes used to wait for external events … (like, `file_finder()`)
- Test needs second thread to generate external events
- So production code is single threaded, but tests are multi threaded o_O
- Better than the other way around ...



A Subtle Bug, Resolved

```
def handle_unseen_exports():
    exports = get_exports()
    seen_exports = (
        ex["id"]
        for ex in exports
        if is_export_seen(ex["id"])
    )

    # ...

    for ex in exports:
        if ex["id"] in seen_exports:
            print("skipping seen export {}".format(ex))
            continue
        print("handling unseen export {}...".format(ex))
        if is_export_seen(ex["id"]):
            print("this should never happen")
```



“if-in” ate our generator!

Summary

- Generators are just cool functions with syntactic sugar, that return iterators
- You can `yield` data out and `send()` data in
- Generators help you save memory
- Generators have hidden state, which is always a fun source of bugs