goto; conference

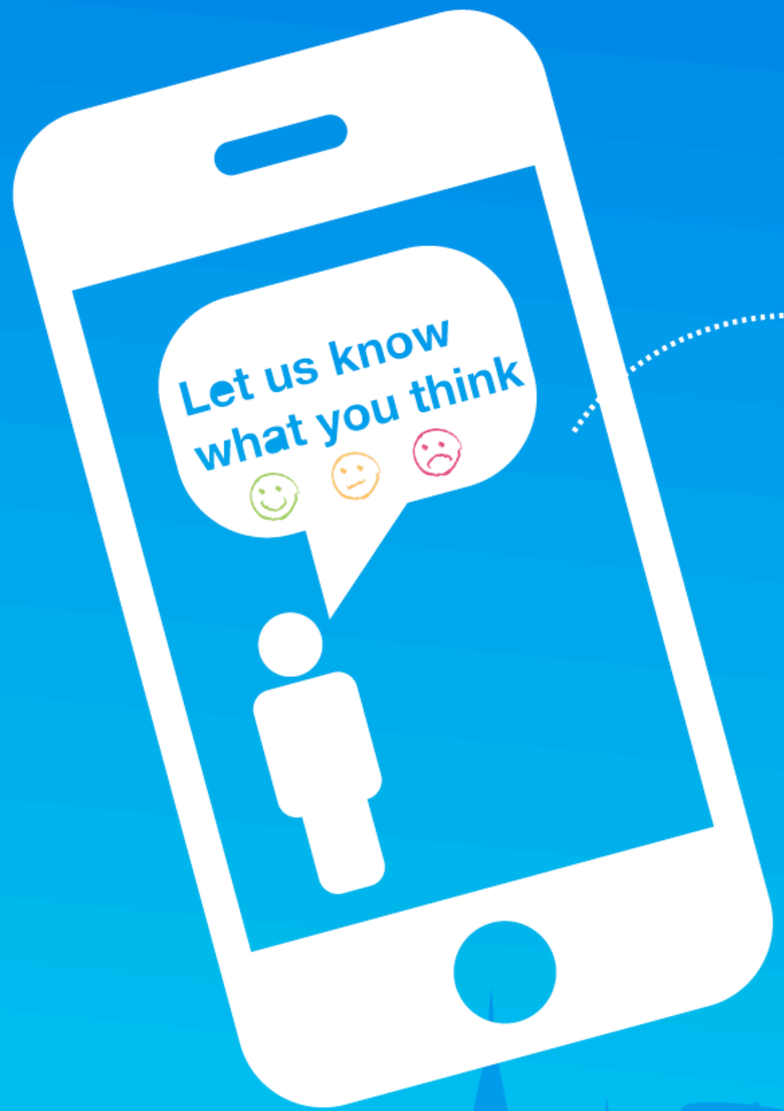
Mutation Testing in Python

Austin Bingham
@austin_bingham

SixtyNORTH

@sixty_north

goto; conference



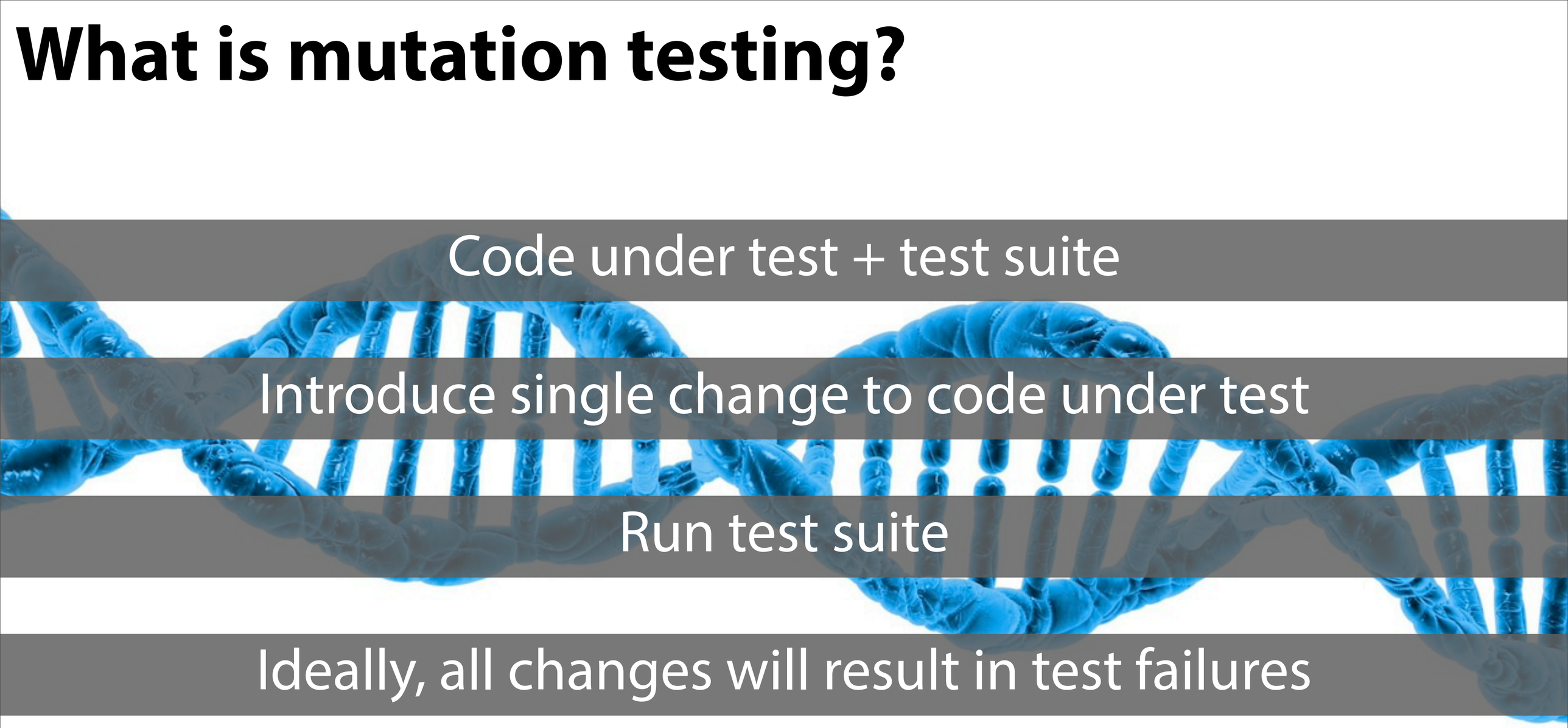
Click ‘engage’ to rate sessions and ask questions

Join the conversation #gotocph

SixtyNORTH

Mutation Testing

What is mutation testing?



Code under test + test suite

Introduce single change to code under test

Run test suite

Ideally, all changes will result in test failures

Basic algorithm

A nested loop of mutation and testing

```
for operator in mutation-operators:
    for site in operator.sites(code):
        operator.mutate(site)
        run_tests()
```

What does mutation testing tell us?

Killed

Tests properly detected the mutation.

Survived

Tests failed to detect the mutant!

either

Tests are inadequate for detecting defects in necessary code

or

Mutated code is extraneous

Incompetent

Mutation produced code which is inherently flawed.

KILL ALL THE MUTANTS!

Goal #1: Coverage analysis

Do my tests *meaningfully* cover my code's functionality

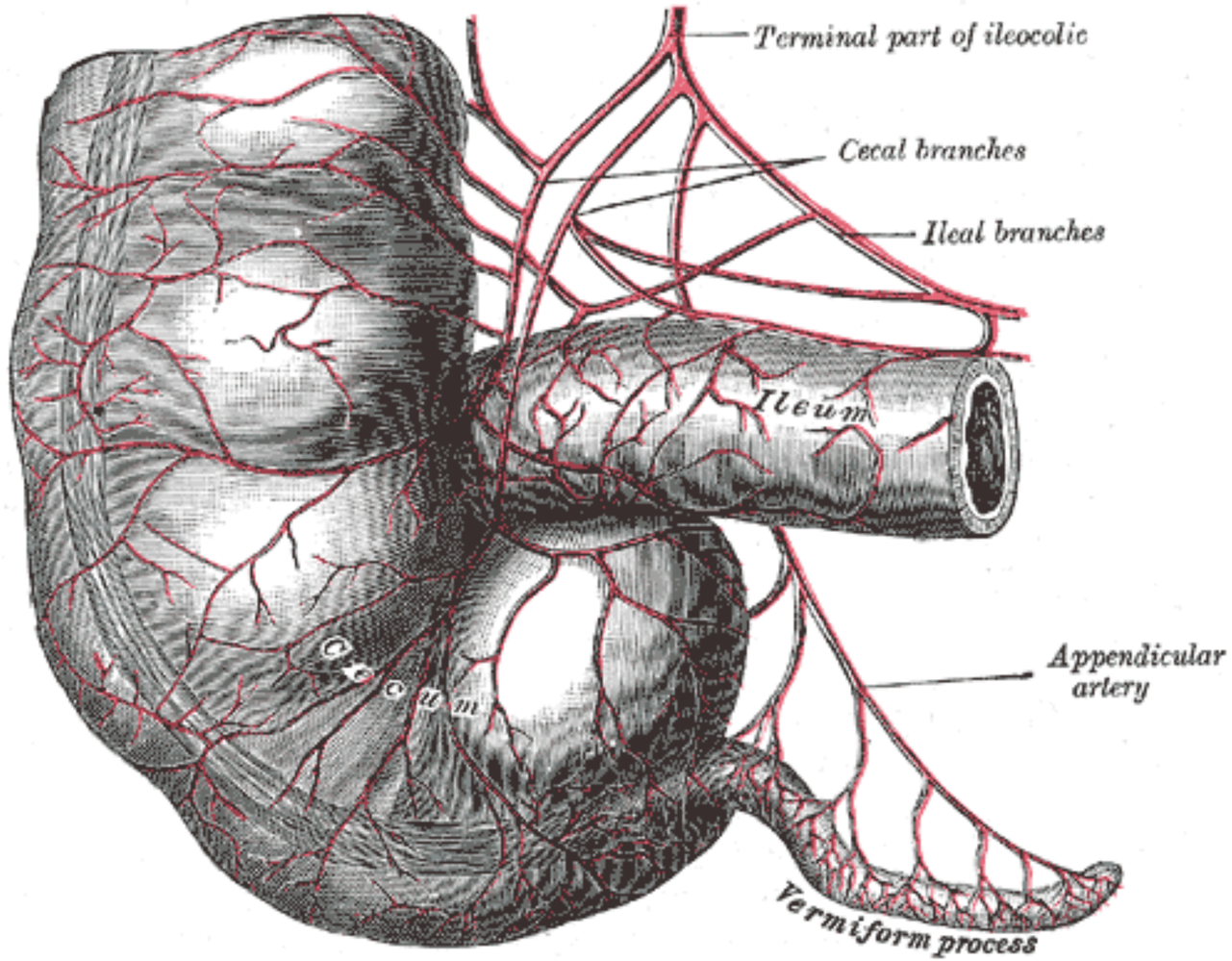
Is a line *executed*?

versus

Is functionality *verified*?

Goal #2: Detect unnecessary code

Survivors can indicate code which is no longer necessary



Examples of mutations

Replace relational operator

```
x > 1
```



```
x < 1
```

`break/continue` replacement

```
break
```

```
continue
```

- AOD - arithmetic operator deletion
- AOR - arithmetic operator replacement
- ASR - assignment operator replacement
- BCR - break continue replacement
- COD - conditional operator deletion
- COI - conditional operator insertion
- CRP - constant replacement
- DDL - decorator deletion
- EHD - exception handler deletion
- EXS - exception swallowing
- IHD - hiding variable deletion
- IOD - overriding method deletion
- IOP - overridden method calling position change
- LCR - logical connector replacement
- LOD - logical operator deletion
- LOR - logical operator replacement
- ROR - relational operator replacement
- SCD - super calling deletion
- SCI - super calling insert
- SIR - slice index remove

Complexity #1: It takes a *looooooooong* time

Long test suites, large code bases, and many operators can add up

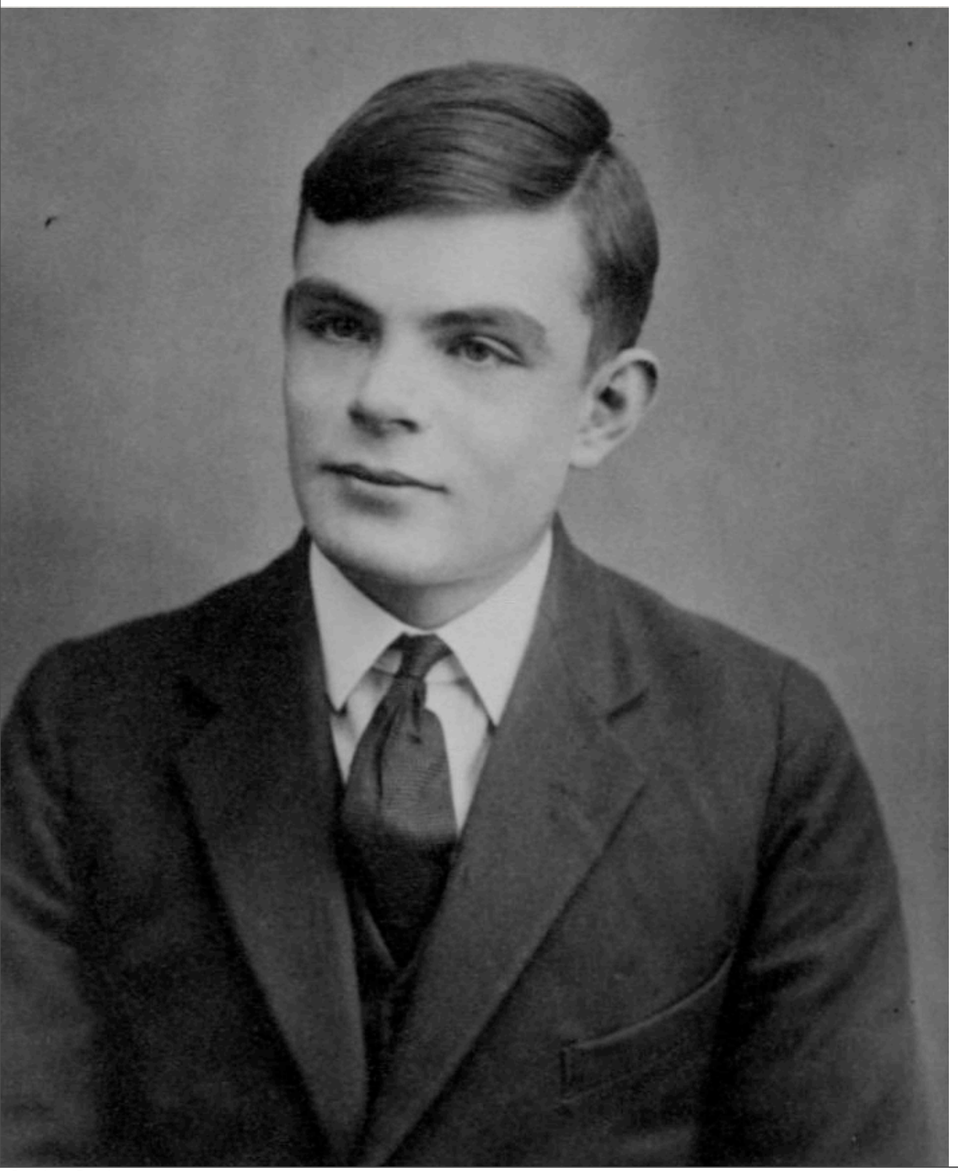
Image credit: John Mainstone (CC BY-SA 3.0)

What to do?

- Parallelize as much as possible!
- After baselining:
  - only run tests on modified code
  - only mutate modified code
- Speed up test suite

Complexity #2: Incompetence detection

Some incompetent mutants are harder to detect that others



"Good luck with that."

Alan Turing (apocryphal)

Complexity #3: Equivalent mutants

Some mutants have no detectable differences in functionality

```
def consume(iterator, n):
    """Advance the iterator n-steps ahead.
       If n is none, consume entirely."""

    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)
```

Cosmic Ray: Mutation Testing for Python

Cosmic Ray operates on packages

Sub-packages and modules are discovered automatically

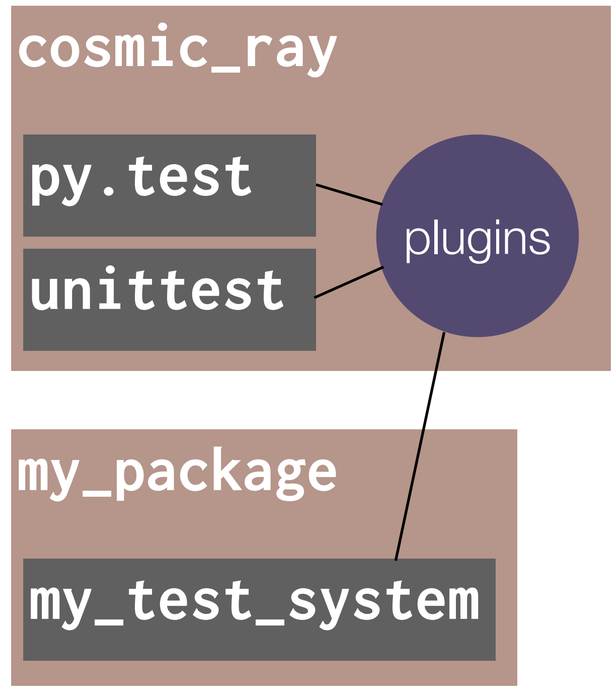
find_modules.py

```
def find_modules(name):
    module_names = [name]
    while module_names:
        module_name = module_names.pop()
        try:
            module = importlib.import_module(module_name)
            yield module
            if hasattr(module, '__path__'):
                for _, name, _ in pkgutil.iter_modules(module.__path__):
                    module_names.append('{}.{}'.format(module_name, name))
        except Exception:  # pylint:disable=broad-except
            LOG.exception('Unable to import %s', module_name)
```

Test system plugins

Support for tests systems are provided by dynamically discovered modules

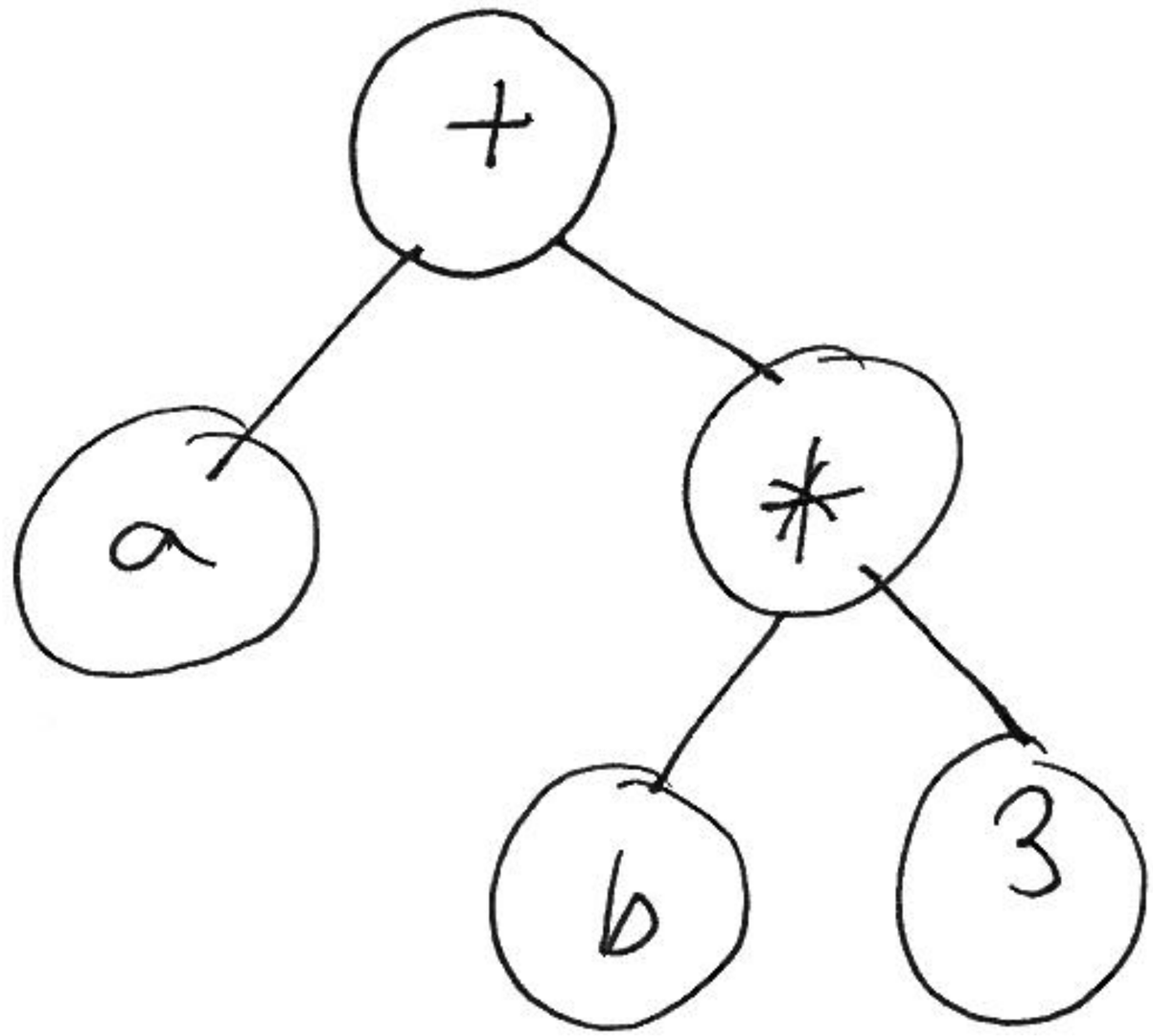
- **Using OpenStack's `stevedore` plugin system**
- **Plugins can come from external packages**
- **Define a `TestRunner` subclass and implement the `_run()` method**
  - Report a simple *success* or *failure* along with a printable object containing more information
- **`TestRunner`s are only given a "start directory" as context**



ast

Standard library abstract syntax tree handling

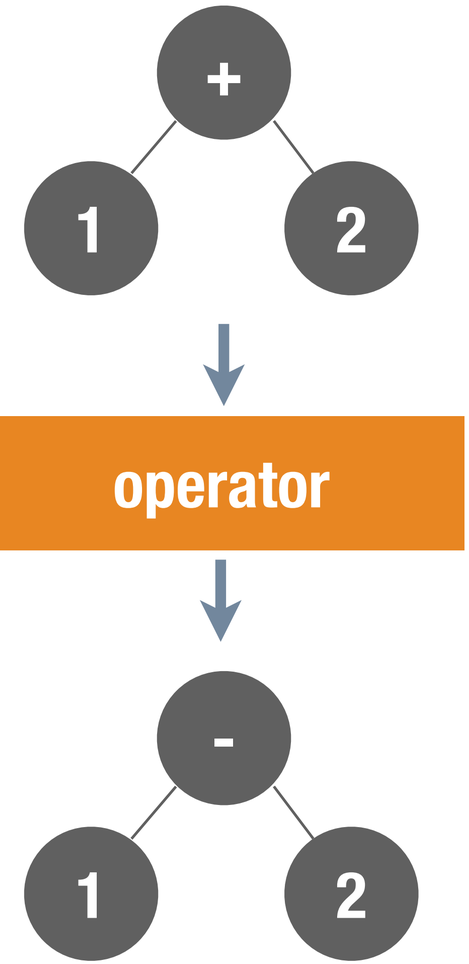
- **Generate AST from source code**
- **Modify copies of ASTs using `ast.NodeTransformer`**



Operators

Operators are responsible for actual AST modification

- **Operators inherit from `Operator` which in turn inherits from `ast.NodeTransformer`**
- **Operators are provided as plugins**
- **They have two primary jobs:**
  - Identify locations where a mutation can occur
  - Perform the mutation open request



Example operator: Reverse unary subtraction

Converts unary-sub to unary-add

```
class ReverseUnarySub(Operator):
    def visit_UnaryOp(self, node):
        if isinstance(node.op, ast.USub):
            return self.visit_mutation_site(node)
        else:
            return node

    def mutate(self, node):
        node.op = ast.UAdd()
        return node
```

Module management: overview

Python provides a sophisticated system for performing module imports

`finders`

Responsible for producing *loaders* when they recognize a module name

`loaders`

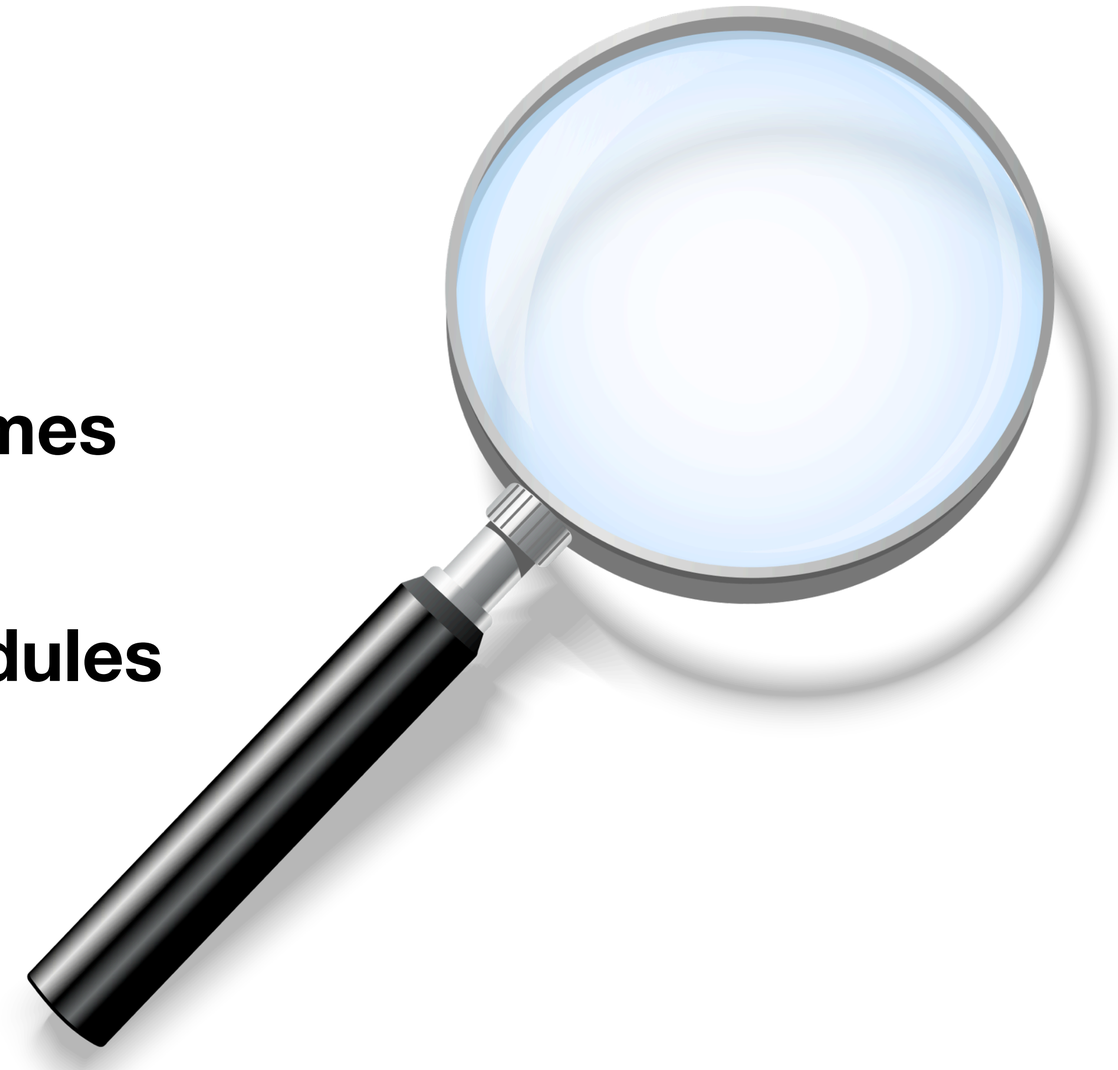
Responsible for populating module namespaces on import

`sys.meta_path`

A list of finders which are queried in order with module names when import is executed

Module management: Finder

Cosmic Ray implements a custom finder



- The finder associates module names with ASTs
- It produces loaders for those modules which are under mutation

Module management: Finder

Cosmic Ray implements a custom finder

```
class ASTFinder(MetaPathFinder):
    def __init__(self, fullname, ast):
        self._fullname = fullname
        self._ast = ast

    def find_spec(self, fullname, path, target=None):
        if fullname == self._fullname:
            return ModuleSpec(fullname,
                              ASTLoader(self._ast, fullname))
        else:
            return None
```

Module management: Loader

Cosmic Ray implements a custom loader

- **The loader compiles its AST in the namespace of a new module object**

Module management: Loader

Cosmic Ray implements a custom loader

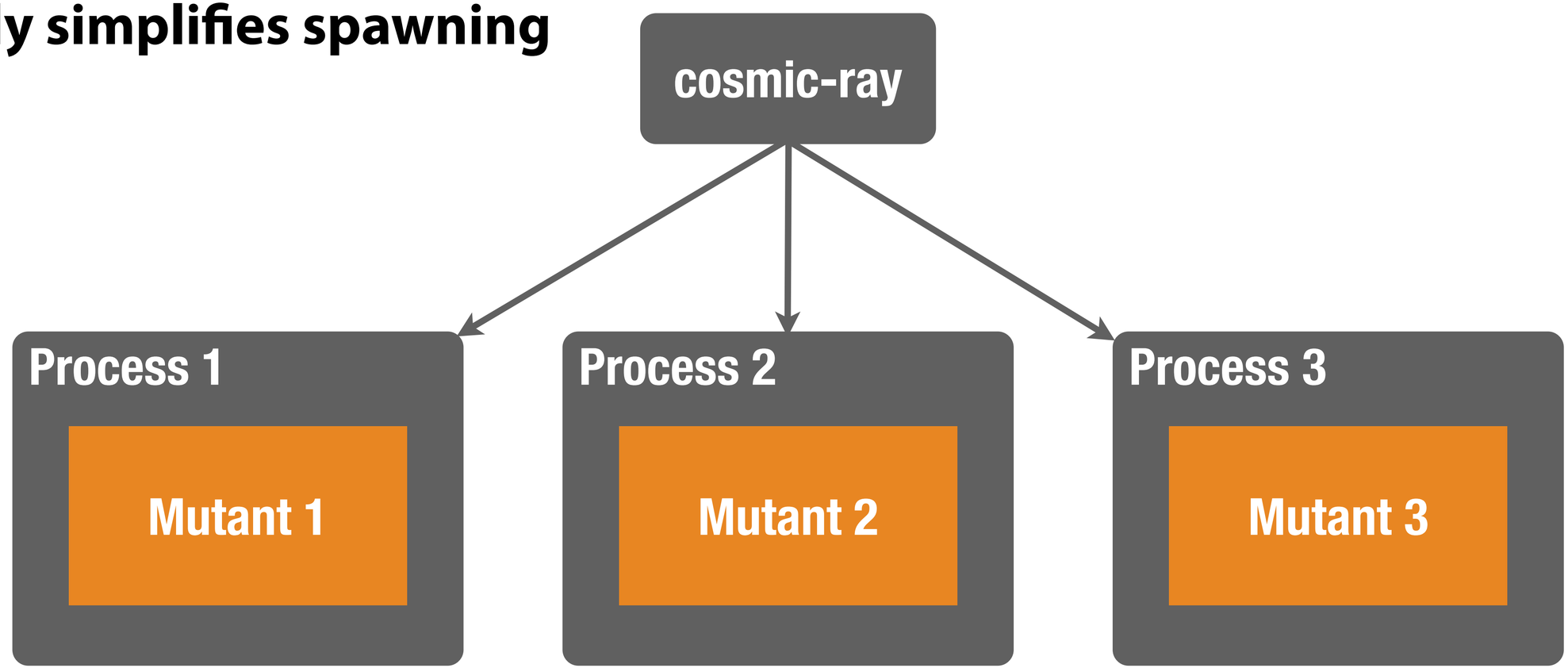
```
class ASTLoader:
    def __init__(self, ast, name):
        self._ast = ast
        self._name = name

def exec_module(self, mod):
    exec(compile(self._ast,
                 self._name,
                 'exec'),
         mod.__dict__)
```

`multiprocessing`

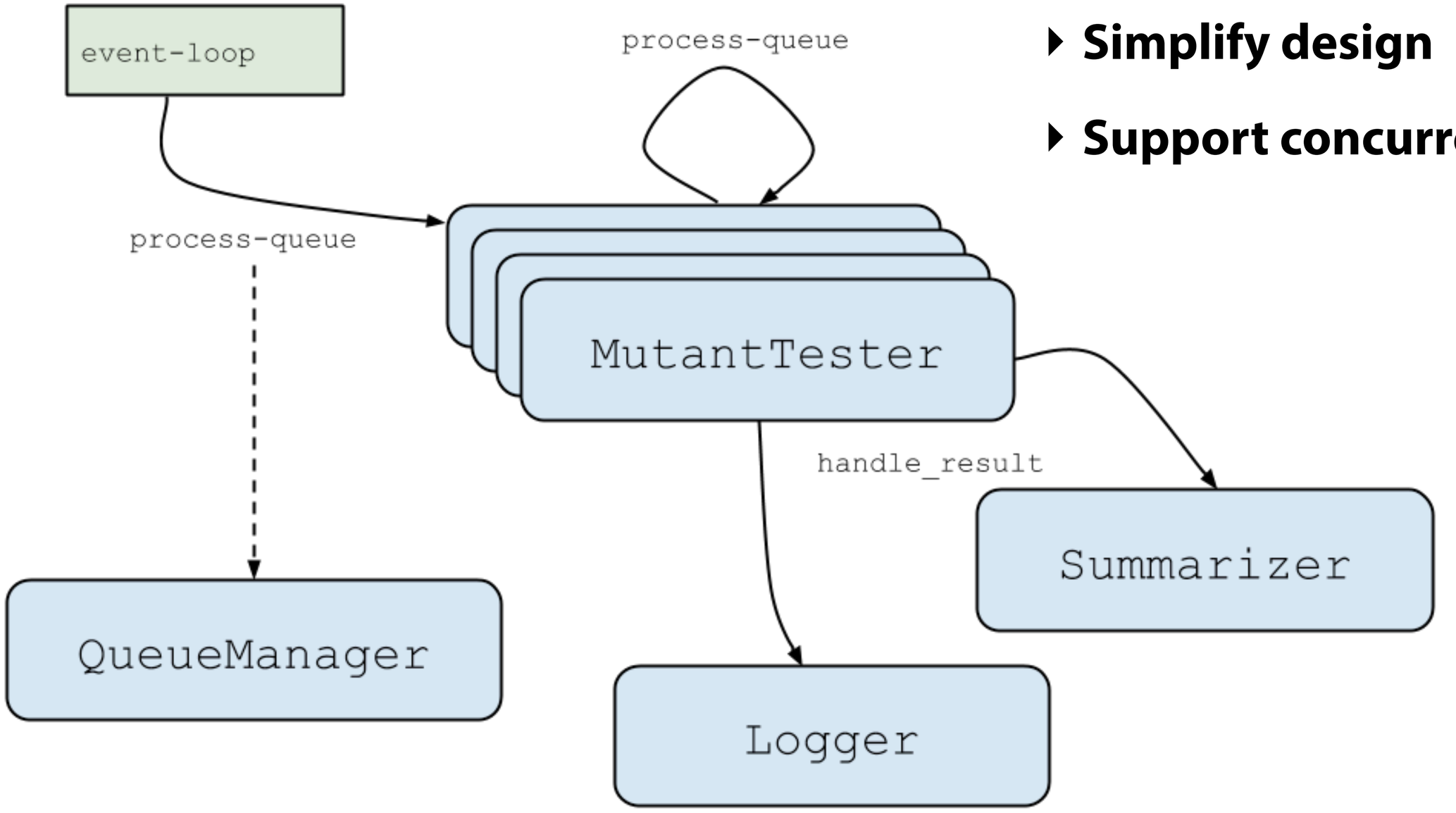
Mutant isolation with multiple processes

- **Avoid cross-mutant interference**
- **Significantly simplifies spawning processes**



pykka

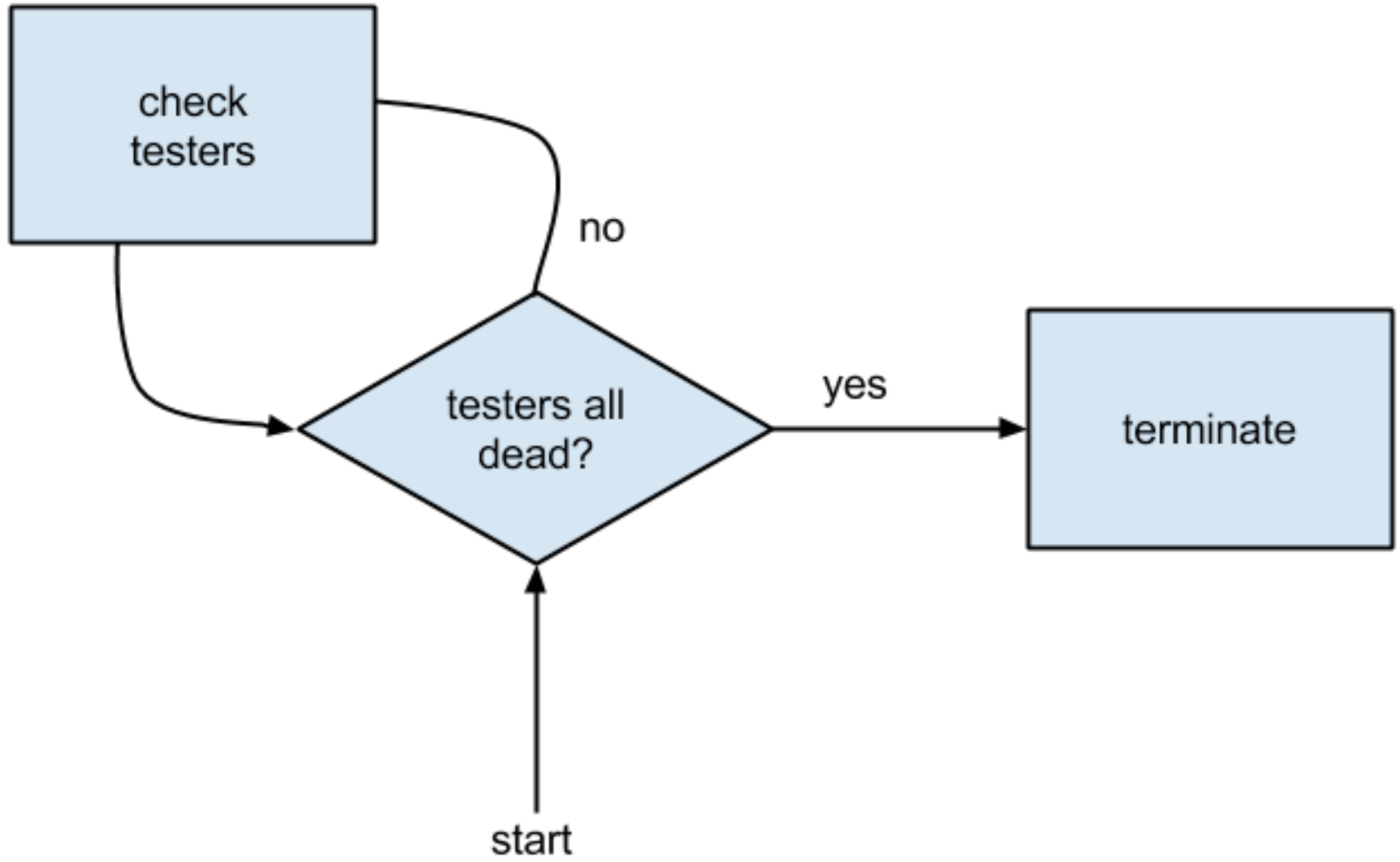
Actor model implementation in Python



- **Simplify design**
- **Support concurrency**

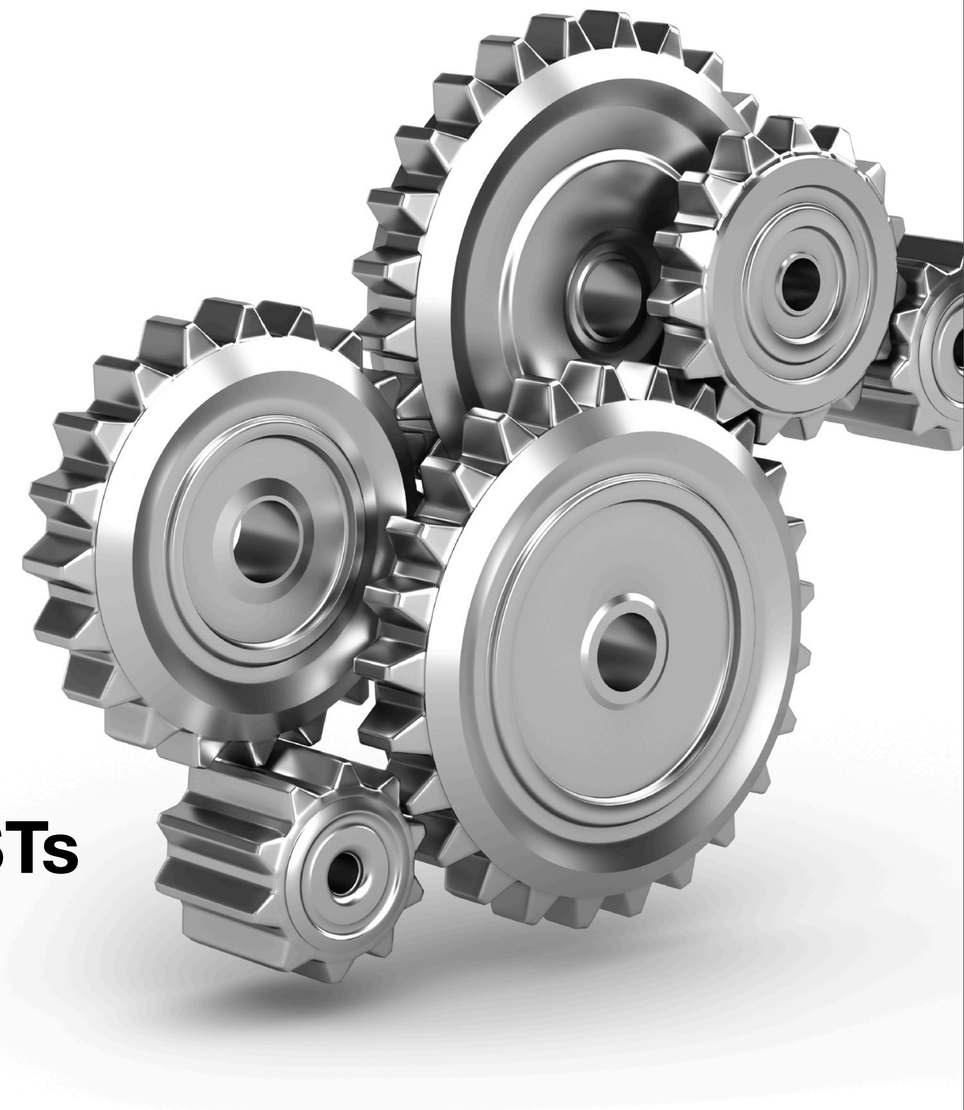
asyncio

Event loop to drive actors



Operational overview

Here's how we put all of these pieces together



1. **Find test-runner to use**
2. **Identify modules for mutation**
3. **Find all of the operators to use**
4. ***Lazily* generate a sequence of mutated ASTs**
5. **For each mutant, run the tests**
   1. Each test runs in a new process
   2. Each run manipulates its location `sys.meta_path` to inject the correct module

Demo

goto; conference

Please

Remember to rate this session

Thank you!

Join the conversation #gotocph

Thank you!

Austin Bingham

@austin_bingham

SixtyNORTH

@sixty_north