Mutation Testing in Python

Theory and Practice

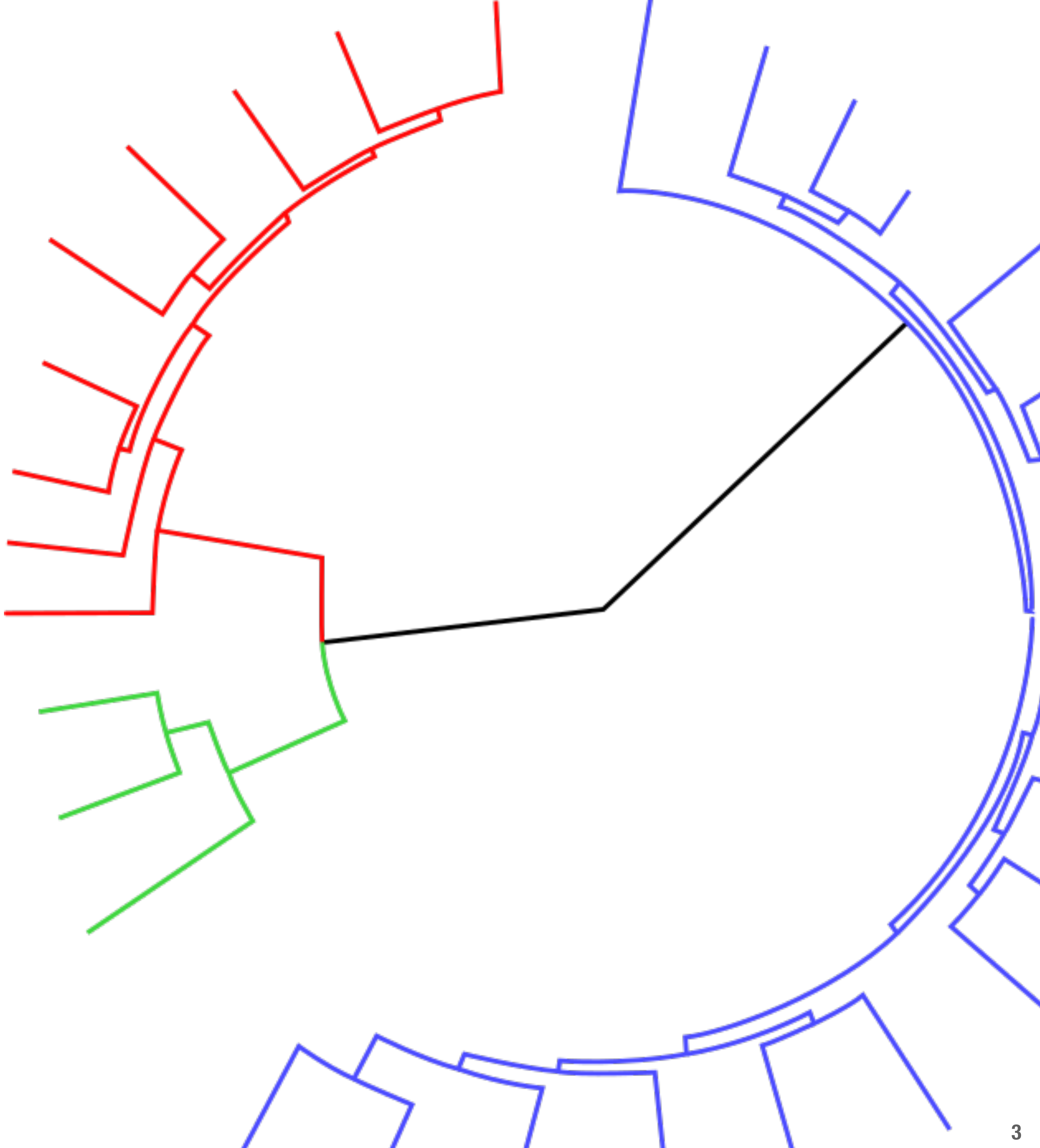
Austin Bingham

@austin_bingham


SixtyNORTH
Canada
Iceland
Norway
Denmark
United Kingdom
Ireland
Netherlands
Belgium
Germany
Czech Rep.
France
Switzerland
Austria
Hungary
Slovenia
Moldova
Romania
Cuba
Dominican Rep.
Guatemala
Honduras
El Salvador
Nicaragua
Costa Rica
Panama
Venezuela
Colombia
Suriname
Ecuador
Mali
Niger
Chad
Sudan
Eritrea
Yemen
Senegal
Gambia
Guinea-Bissau
Guinea
Sierra Leone
Liberia
Côte d'Ivoire
Ghana
Burkina Faso
Benin
Nigeria
Cameroon
Central African Rep.
S. Sudan
Ethiopia
Somaliland
Somalia
Uganda
Kenya
Gabon
Congo
Dem. Rep. Congo
Tanzania
CARTODB
CartoDB attribution

Agenda

1. Introduction to the theory of mutation testing
2. Overview of practical difficulties
3. Cosmic Ray: mutation testing for Python
4. Demo
5. Questions



Mutation Testing

“Mutation testing is **conceptually quite simple**.

Faults (or mutations) are automatically seeded into your code, then your **tests are run**. If your tests fail then the mutation is **killed**, if your tests pass then the mutation **lived**.

The **quality of your tests** can be gauged from the percentage of mutations killed.”

- *pitest.org*

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.

Incompetent

Mutation produced code which is inherently flawed.

Survived

Tests failed to detect the mutant!

either

Tests are inadequate for detecting defects in necessary code

or

Mutated code is extraneous


KILL ALL THE MUTANTS!

Goals of Mutation Testing

Goal #1: Coverage analysis

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

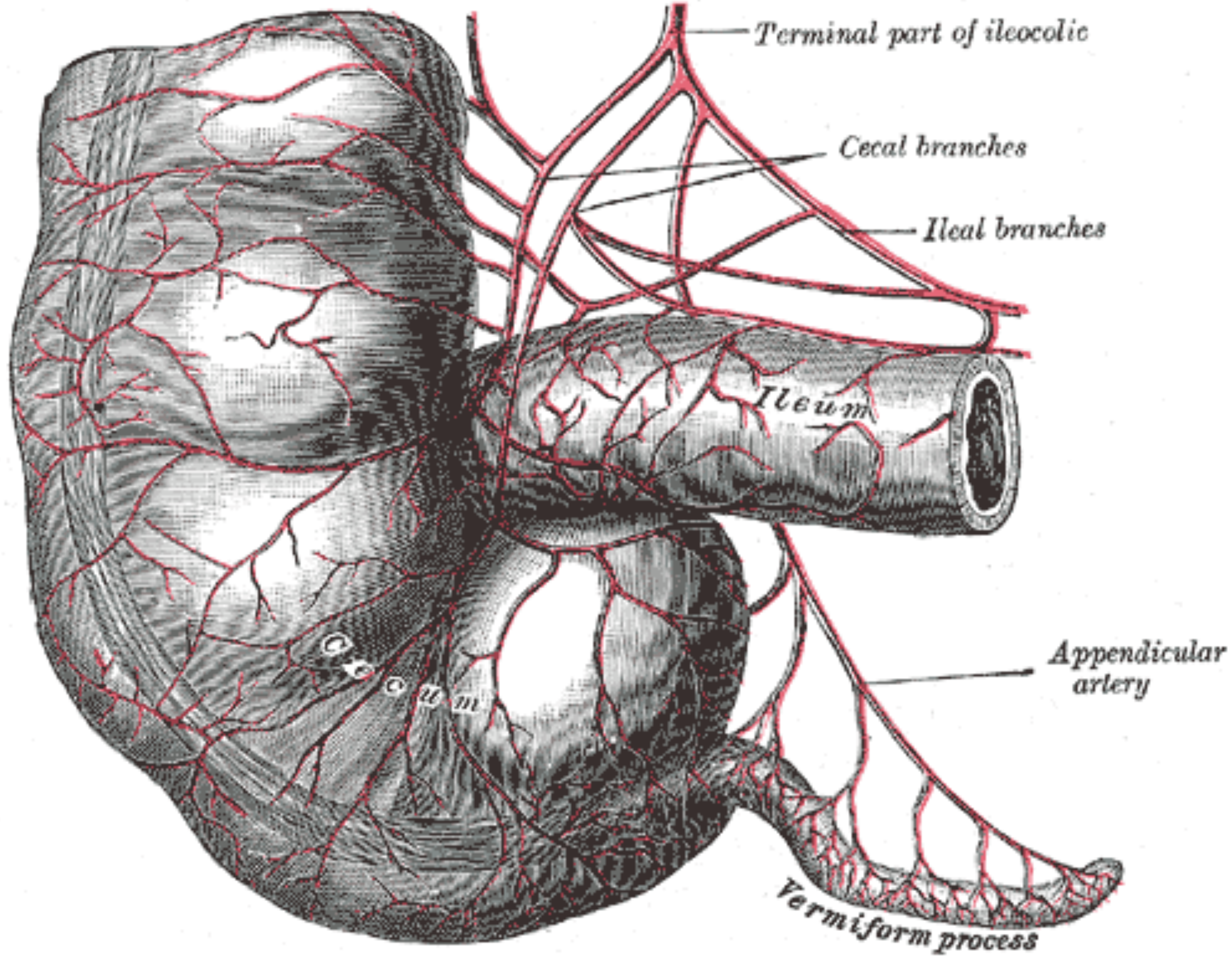
Is a line *executed*?

versus

Is functionality *verified*?

Goal #2: Detect unnecessary code

Survivors can indicate code which is no longer necessary



Types of Mutations

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

Language-agnostic mutations

Some mutations are very widely applicable

- **Constant replacement**

  `0 → 4`

- **Constant for scalar variable replacement**

  `some_func(x) → some_func(42)`

- **Arithmetic operator replacement**

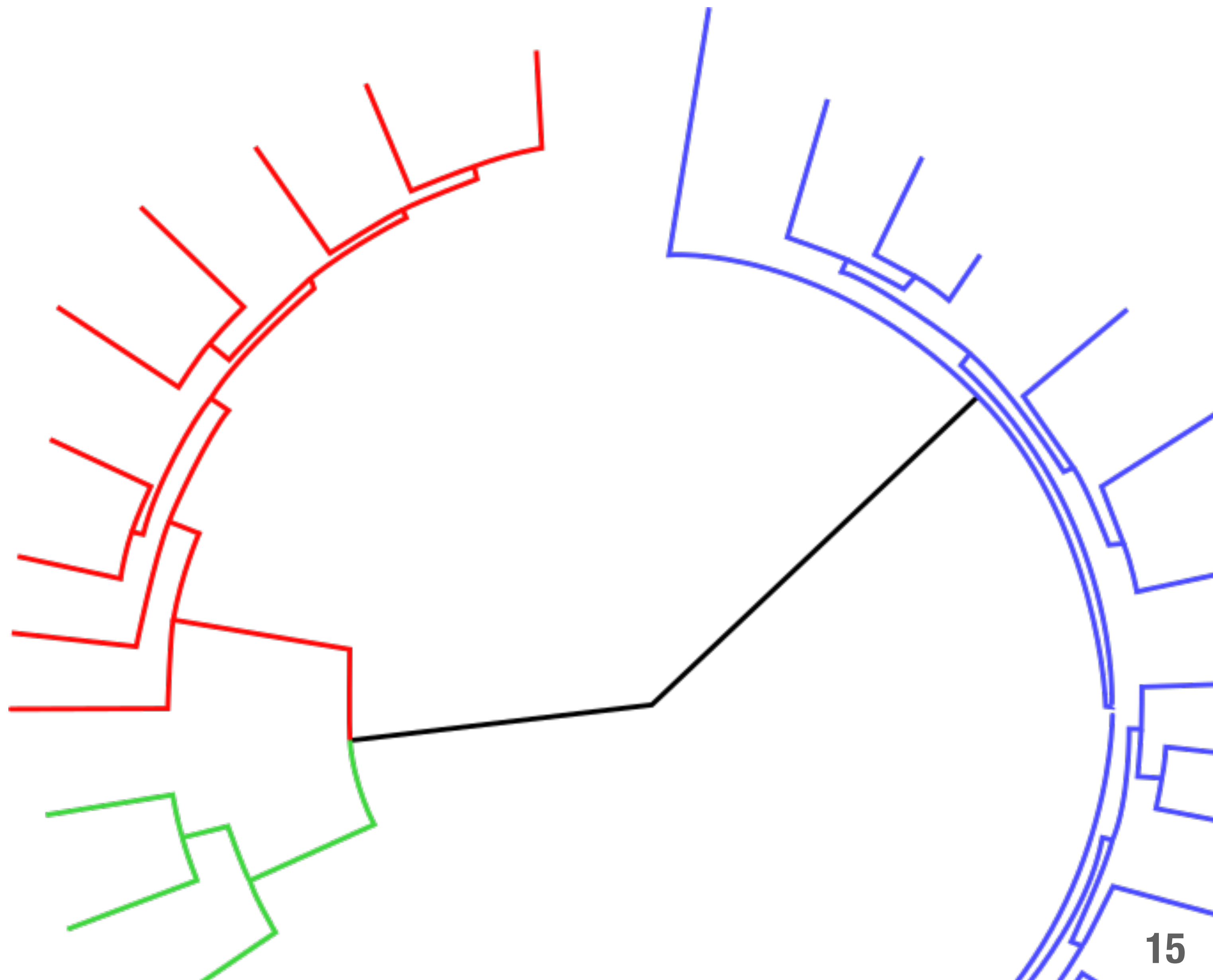
  `x + y → x * y`

- **Relational operator replacement**

  `x < y → x <= y`

- **Unary operator insertion**

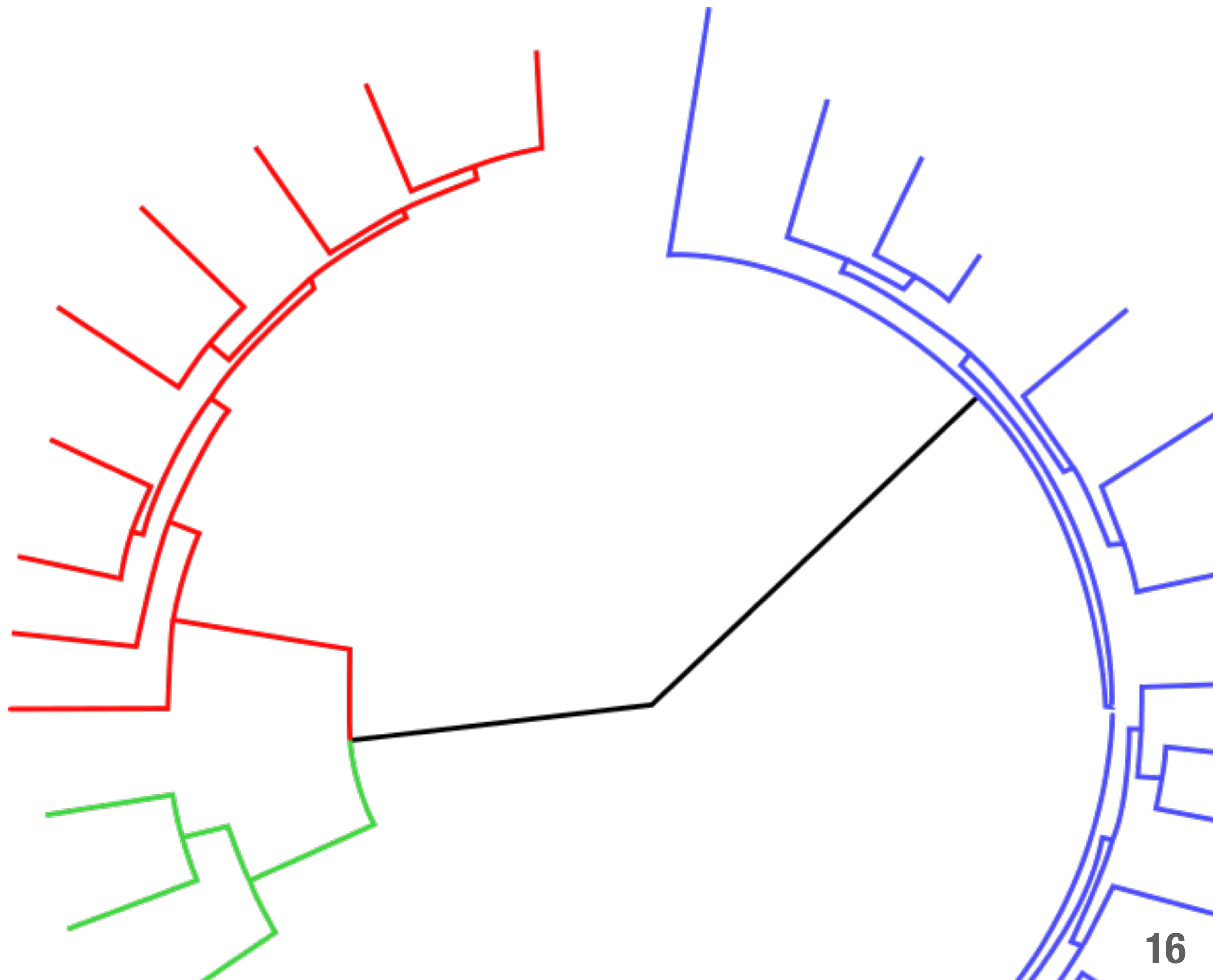
  `int x = 1 → int x = -1`



Object-oriented mutations

Mutations which only make sense for (some) OO languages

- **Changing an access modifier**
  `public int x → private int x`
- **Remove overloading method**
  `int foo() {} →` ~~`int foo() {}`~~
- **Change base class order**
  `class X(A, B) → class X(B, A)`
- **Change parameter order (?)**
  `foo(a, b) → foo(b, a)`



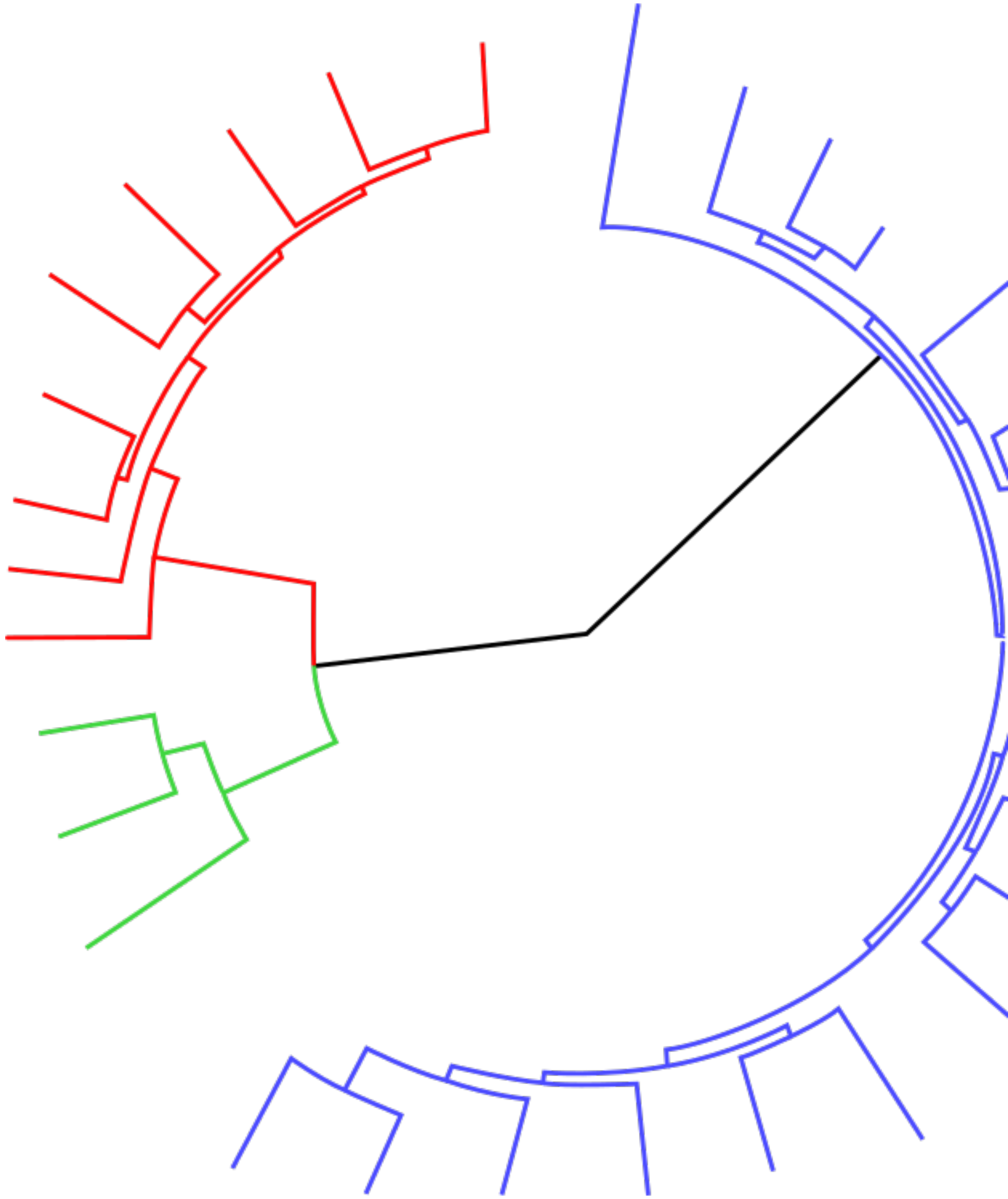
Lionel Brand - http://www.uio.no/studier/emner/matnat/ifi/INF4290/v10/undervisningsmateriale/INF4290-Mutest.pdf

Functional mutations

Mutations which only make sense for (some) functional languages

- **Change order of pattern matching**

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
                ↓
take _ [] = []
take 0 _ = []
take n (x:xs) = x : take’(n-1) xs
```



Duc Le, Mohammad Amin Alipour, Rahul Gopinath, Alex Groce - http://web.engr.oregonstate.edu/~alipourm/pub/fp_mutation.pdf

Complexities of Mutation Testing

Complexity #1: **It takes a *looooooooong* time**

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

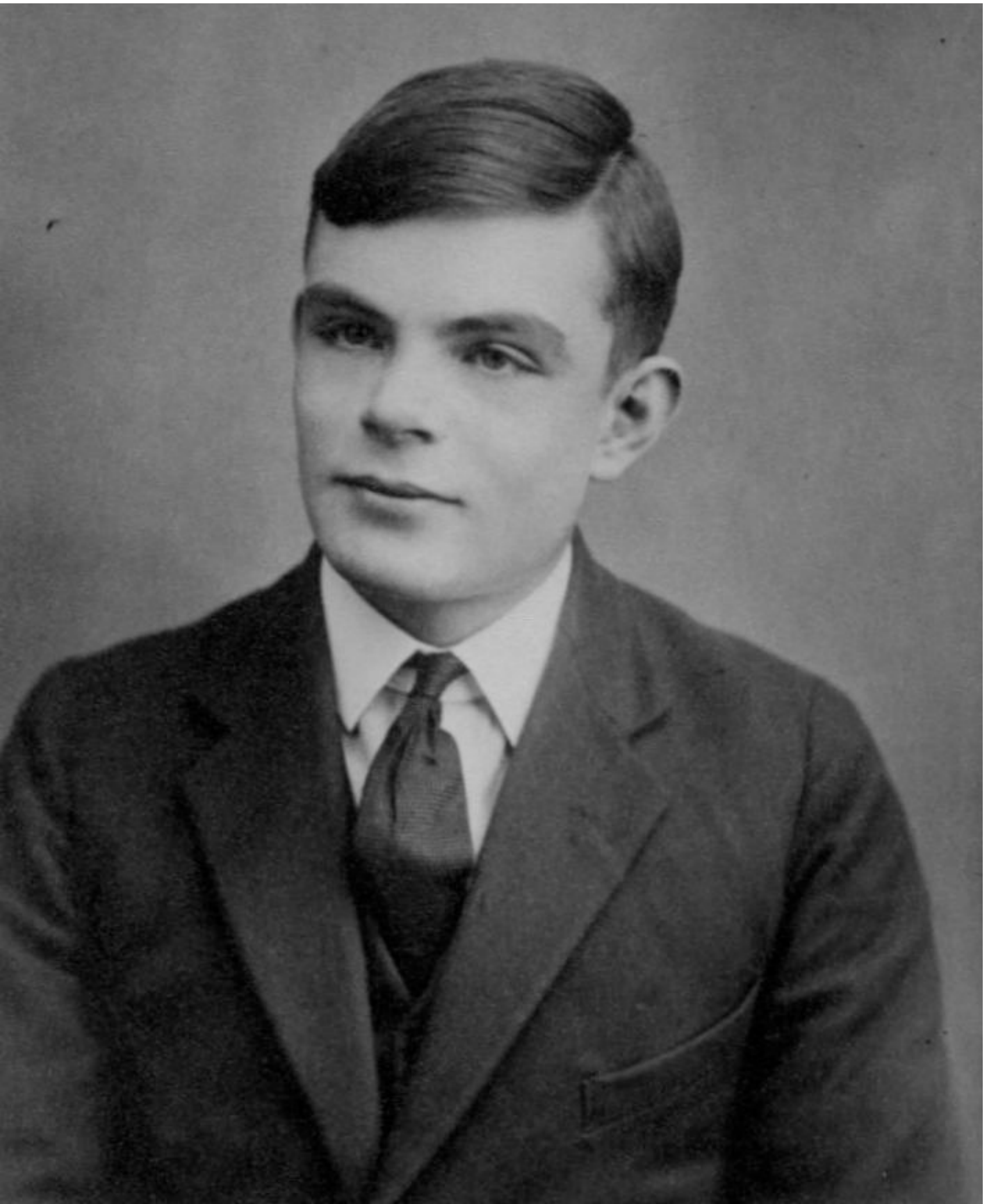
What to do?

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

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

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

```python
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)
```

Complexity #3: **Equivalent mutants**

Some mutants have no detectable differences in functionality

```
if __name__ == '__main__':
    run()
```

Cosmic Ray: Mutation Testing for Python

python™

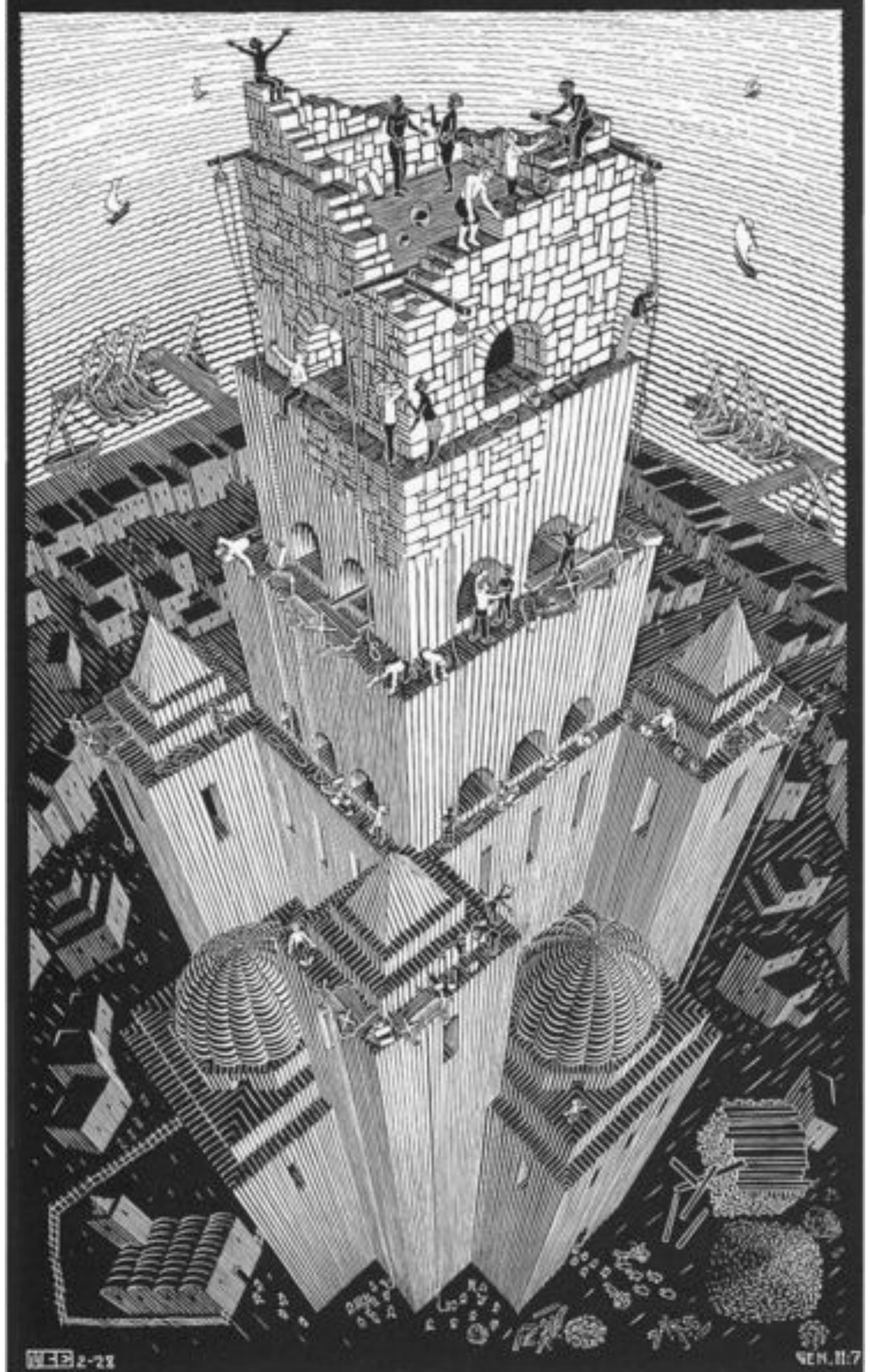
github.com/sixty-north/cosmic-ray

Implementation challenge

What do we need to do to make this work?

1. Determine which mutations to make.
2. Make those mutations one at a time.
3. Run a test suite against each mutant.

While also dealing with the complexities!



Operators

Core concept: *Operators*

Operators sit at the center of Cosmic Ray's…well…operations

Job #1:
Identify potential mutation sites

1 + 2

Job #2:
Perform mutations on request

1 – 2

- Not a job -
Decide when to perform mutations

Operator *cores*

Operator cores take action when a potential mutation site is detected

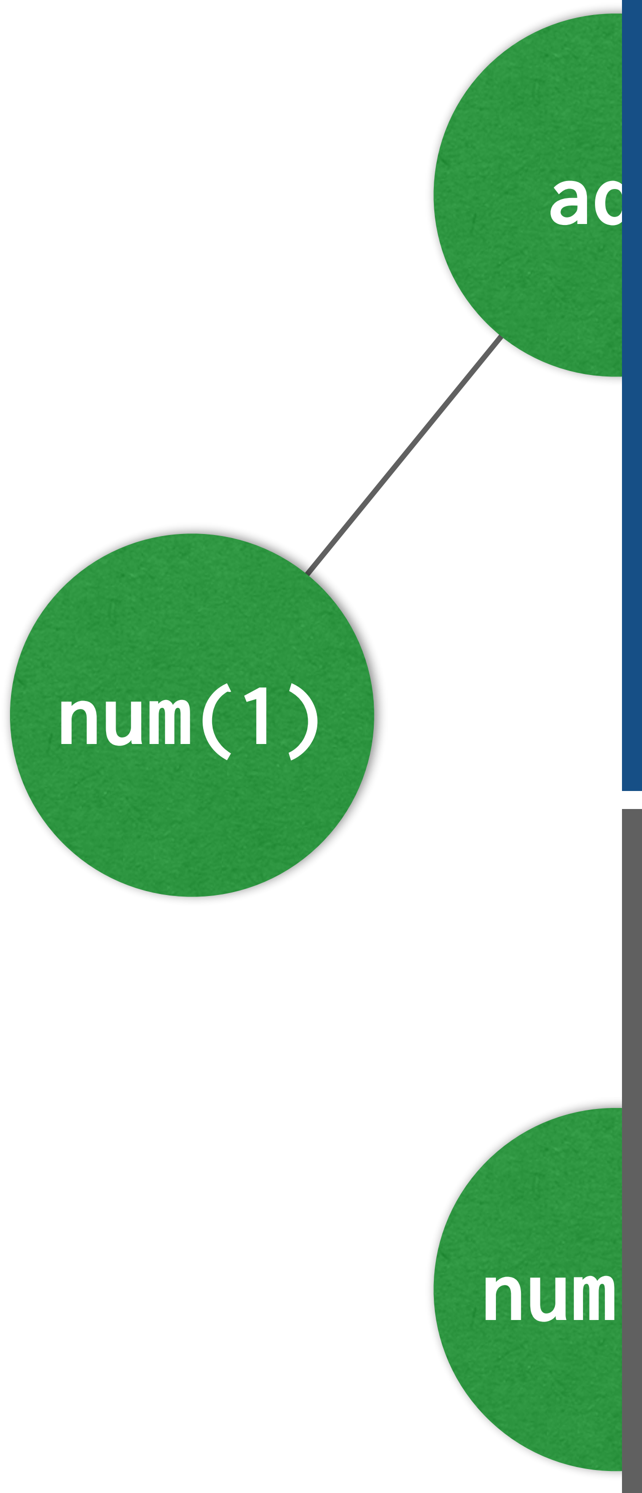
Current cores

1. *Counting*: counts number of mutations
2. *Mutating*: requests mutation at correct time

Python's standard `ast` module

Abstract syntax trees: the basis for Cosmic Ray's mutation operators

1 +



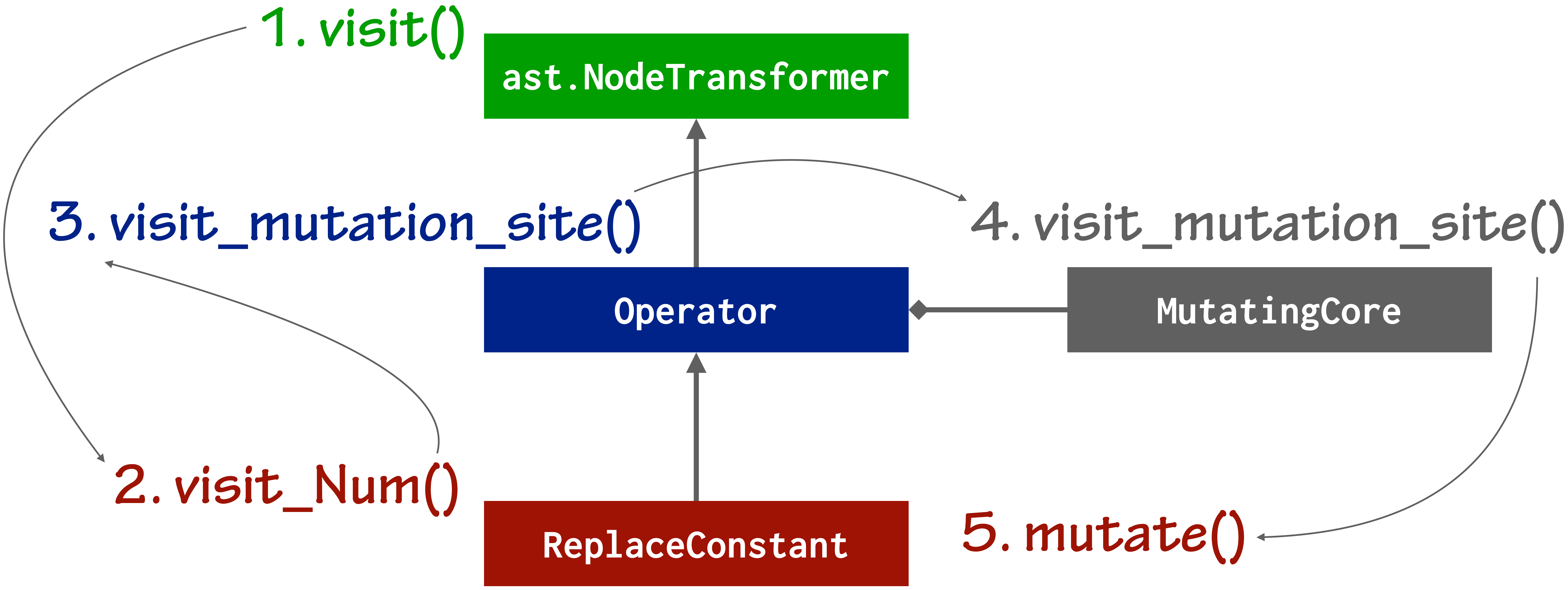
`ast` elements we use…

- Generating ASTs from Python source code
- Walking/transforming ASTs
- Manipulating AST nodes cleanly

Plus we use `compile()` to transform ASTs into code objects at runtime

Operators: putting it all together

The operator base class, subclasses, and cores all do a little dance



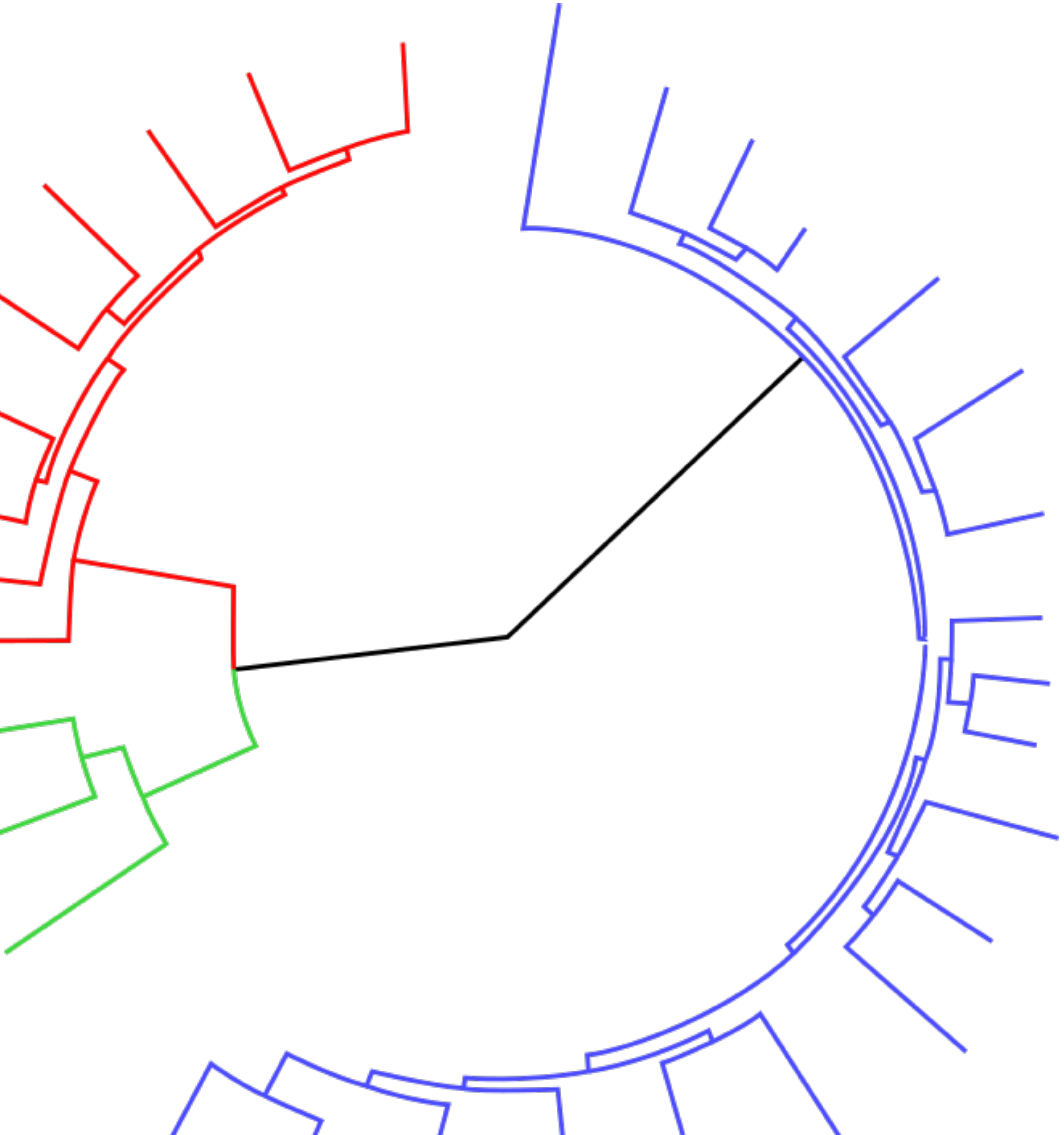
Example operator: Reverse unary subtraction

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
```

Operators summary



- Use `ast` to transform source code into abstract syntax trees.
- Implement operators which are able to detect mutation sites and perform mutations.
- Use different cores to control exactly what the operators are doing.

Installing modules

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**

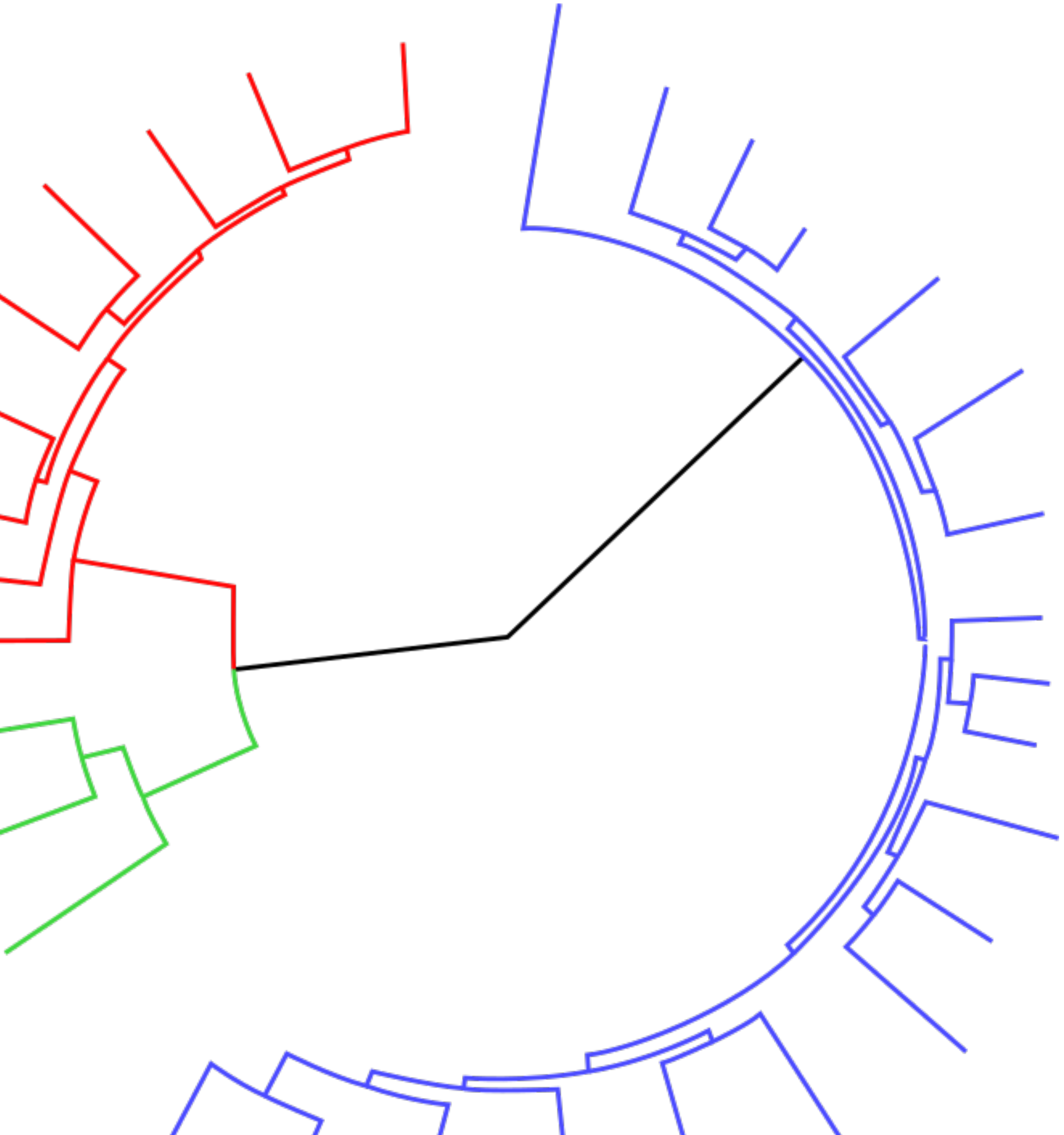
Module management: Loader

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__)
```

Module installation summary



- Use `MutatingCore` to generate mutated ASTs
- Use `compile()` to produce code objects from mutated ASTs
- Use *finders*, *loaders*, and `sys.meta_path` to advertise and install these mutated modules

Figuring out what to mutate

Cosmic Ray operates on a package

This seems like the natural boundary for mutation testing in the Python universe

- The user specifies a single package for mutation
- Cosmic Ray scans the package for all of its modules
- There are limitations to the kinds of modules it can mutate
- It is possible to exclude modules which should not be mutated

Finding modules

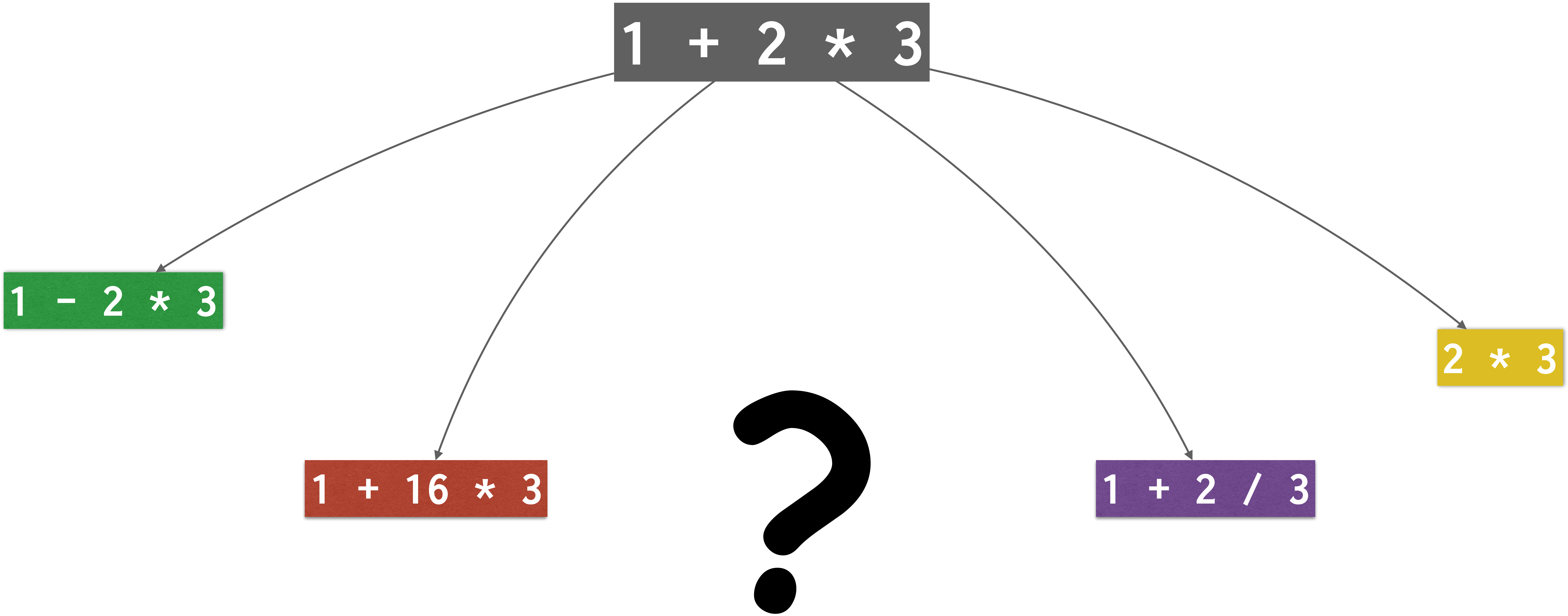
Sub-packages and modules are discovered automatically

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)
```

Counting potential mutants

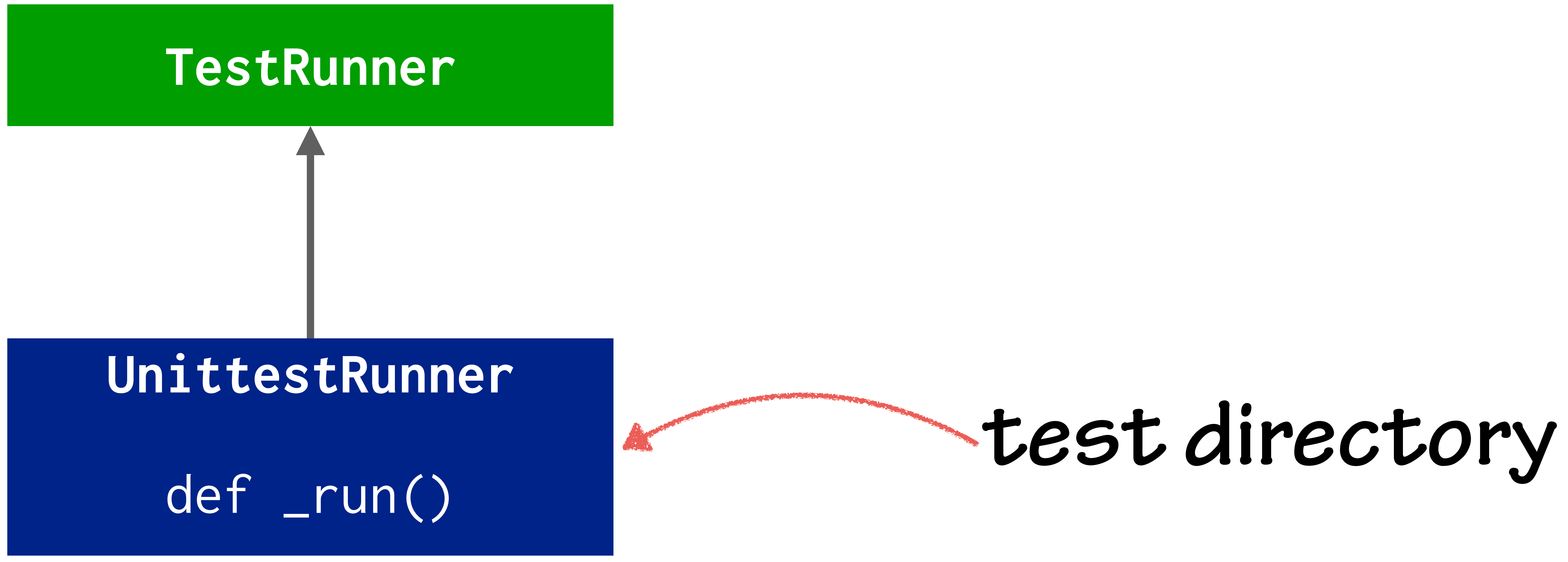
An interesting problem!



Running tests

Test runners

Encapsulate the differences between various testing systems



Testing overview

- Figure out what to mutate
- Create a mutant
- Install the mutant
- Tell `TestRunner` to run the tests

In a separate process

Dealing with incompetent mutants

There is no perfect strategy for detecting them

Absolute timeout

or

Based on a baseline

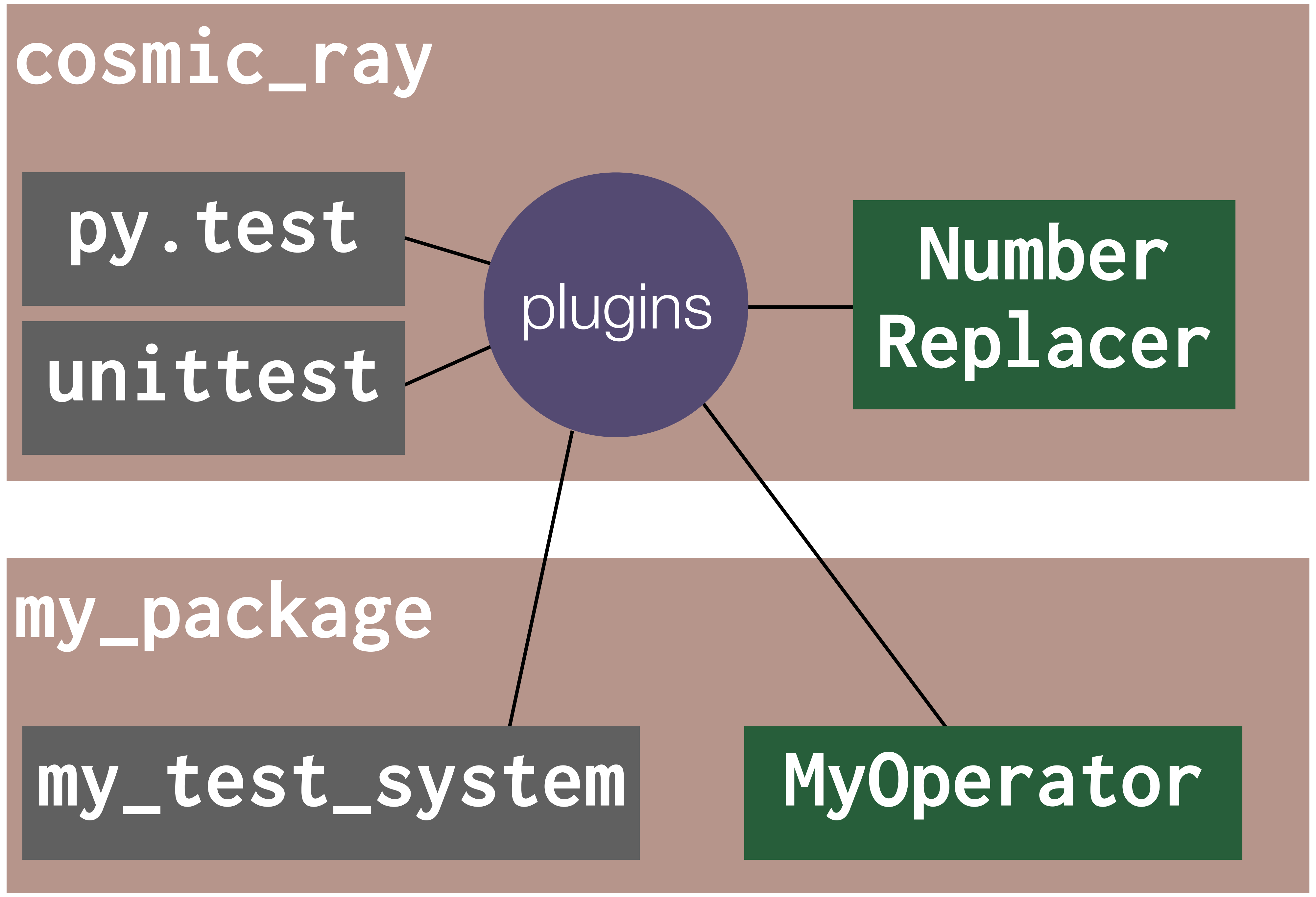
Image by o5com - https://www.flickr.com/photos/o5com/5488964999

The rest of the tech

Test system and operator plugins

Test runners and operators are provided by dynamically discovered modules

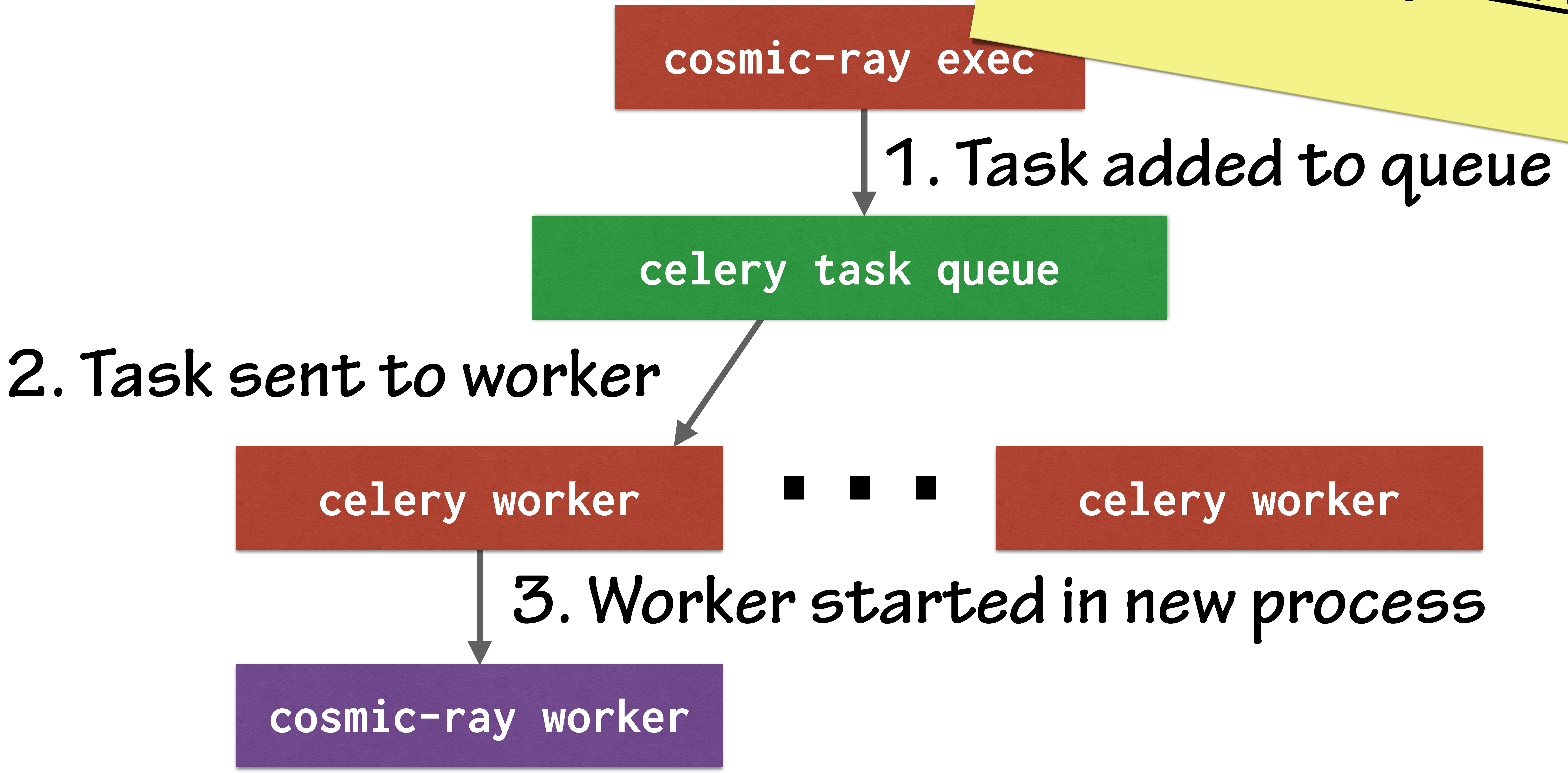
- **Using OpenStack's `stevedore` plugin system**
- **Plugins can come from external packages**



Celery: distributed task que

Used to distribute tasks to more than one machine

celeryproject.org



Staging of work

Use an embedded database to keep track of work and results

github.com/msiemens/tinydb

- **Use `CountingCore` to determine *work-to-be-done***
- **Only schedule work items that don't have results**
- **Allows interruption and resumption of runs**
- **Natural place for results**

docopt: command-line interface description language

Describe command-line syntax in comment strings…like magic!

```
"""usage: cosmic-ray counts [options] [--exclude-modules=P ...] <top-module>

Count the number of tests that would be run for a given testing configuration.
This is mostly useful for estimating run times and keeping track of testing
statistics.

options:
  --no-local-import   Allow importing module from the current d
  --test-runner=R     Test-runner plugin to use [default: unit
  --exclude-modules=P Pattern of module names to exclude from
"""
```

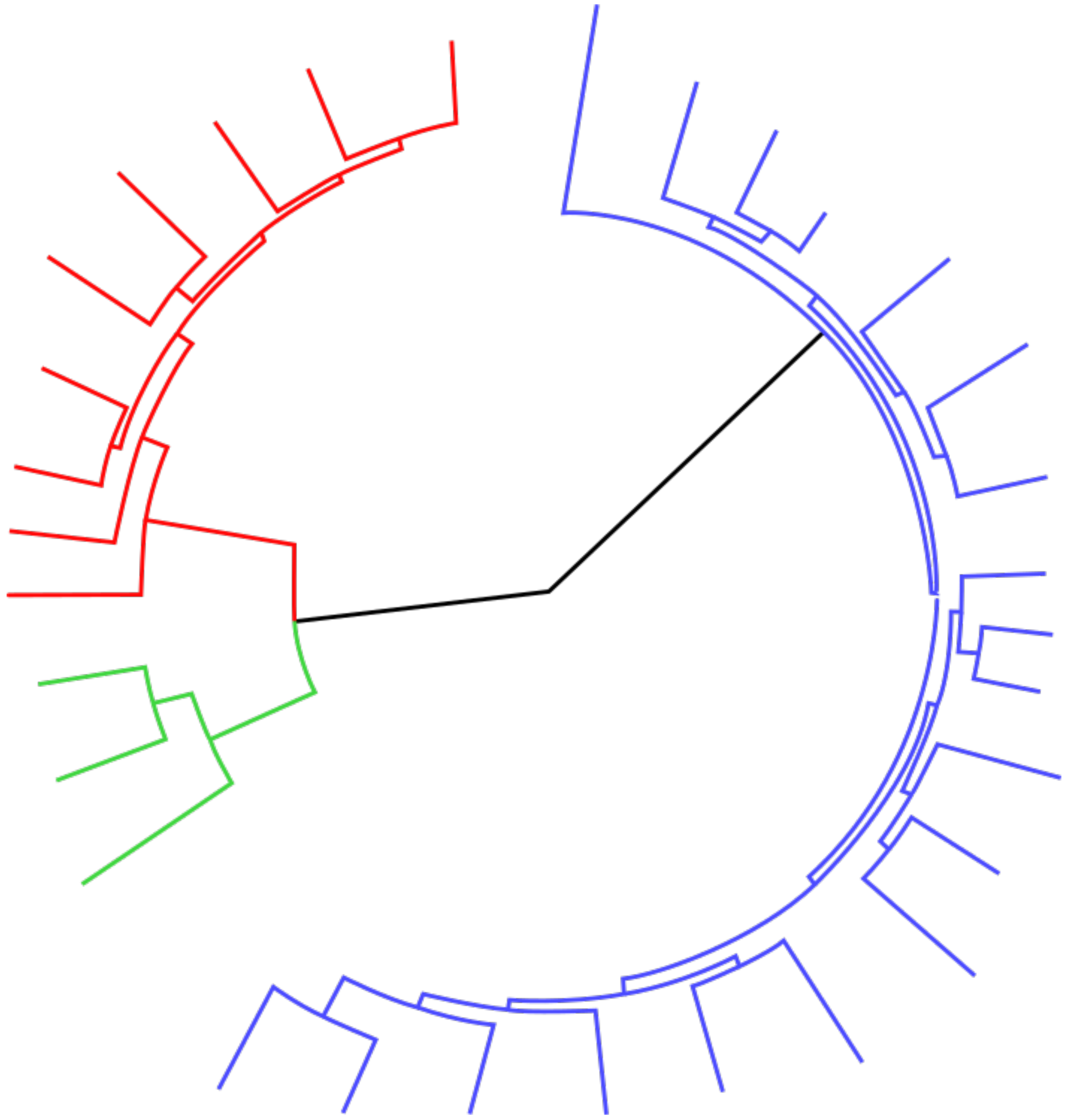
docopt.org

```
$ cosmic-ray —no-local-import —exclude-modules=“.*.test” foo
```

Remaining work

Remaining work

There's plenty left to do if you're interested!



- Properly implementing timeouts
- Exceptions and processing instructions
- Support for more kinds of modules
- Integration with coverage testing

github.com/sixty-north/cosmic-ray/issues

Demo

Thank you!

Austin Bingham

@austin_bingham

SixtyNORTH

@sixty_north