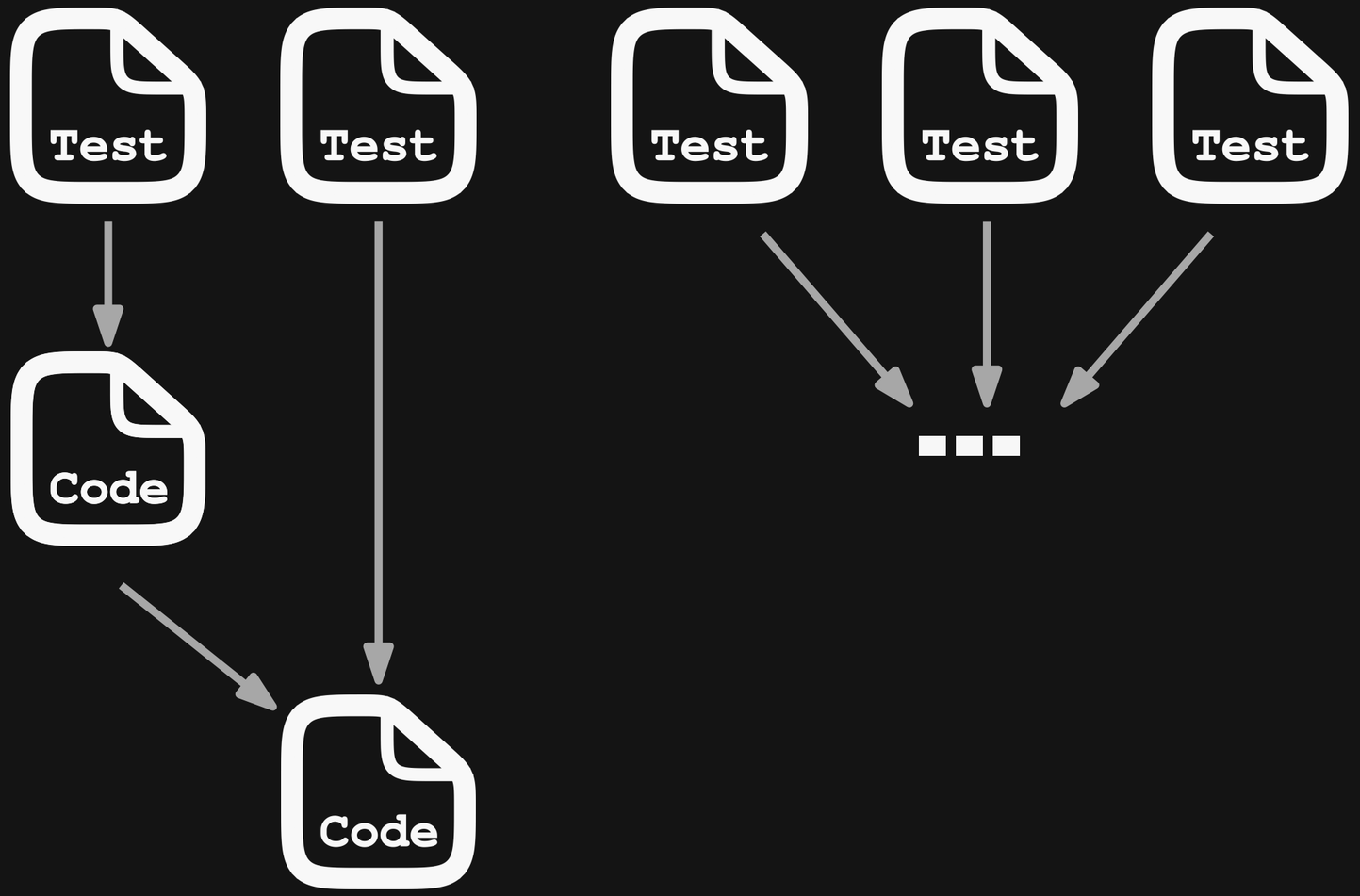
Names Are All You Need:

Effective and Safe Regression Test Selection for Python

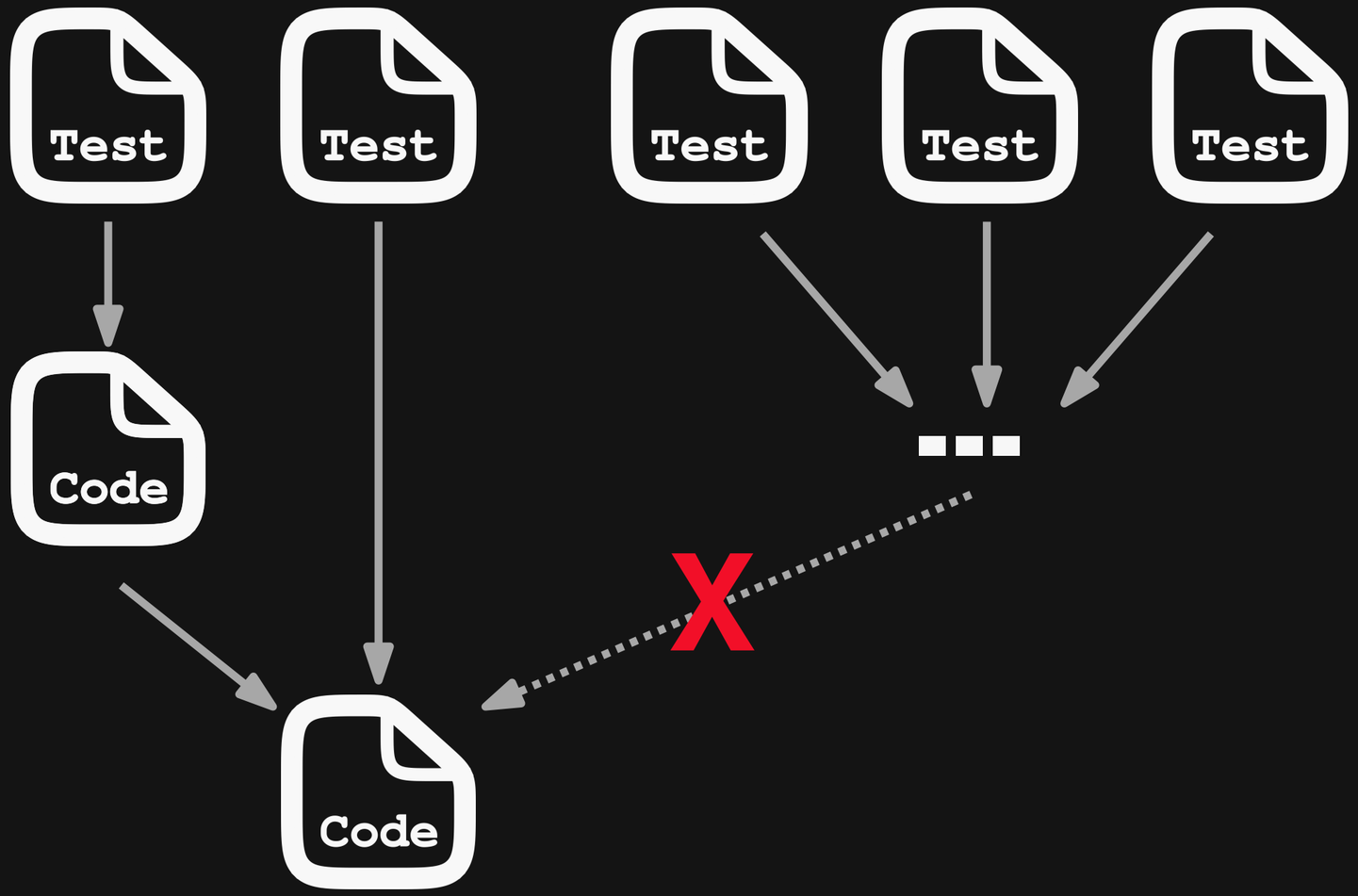
You Wang*, Michael Pradel#, Zhongxin Liu*

* Zhejiang University, # CISPA

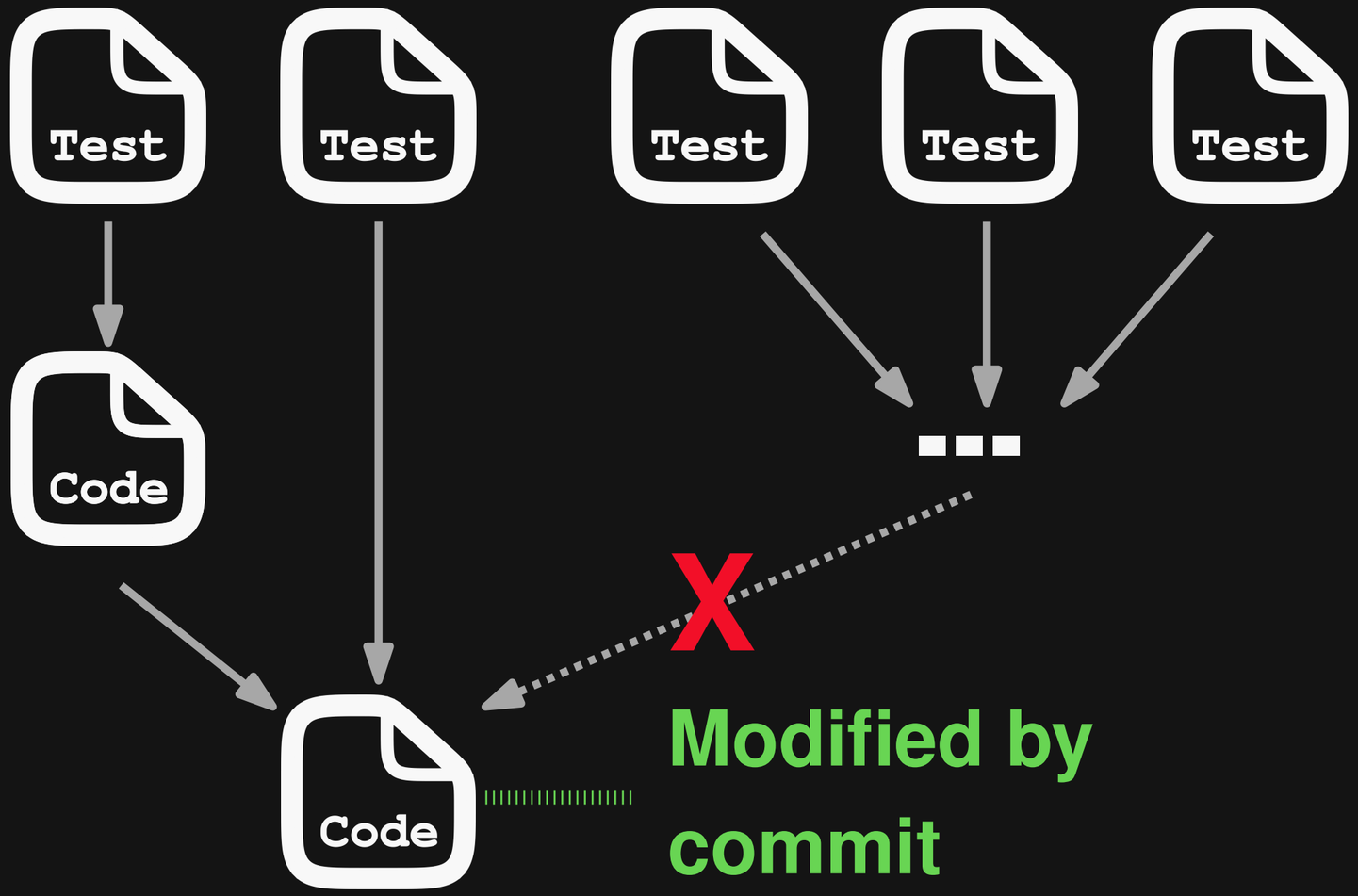
Regression Test Selection



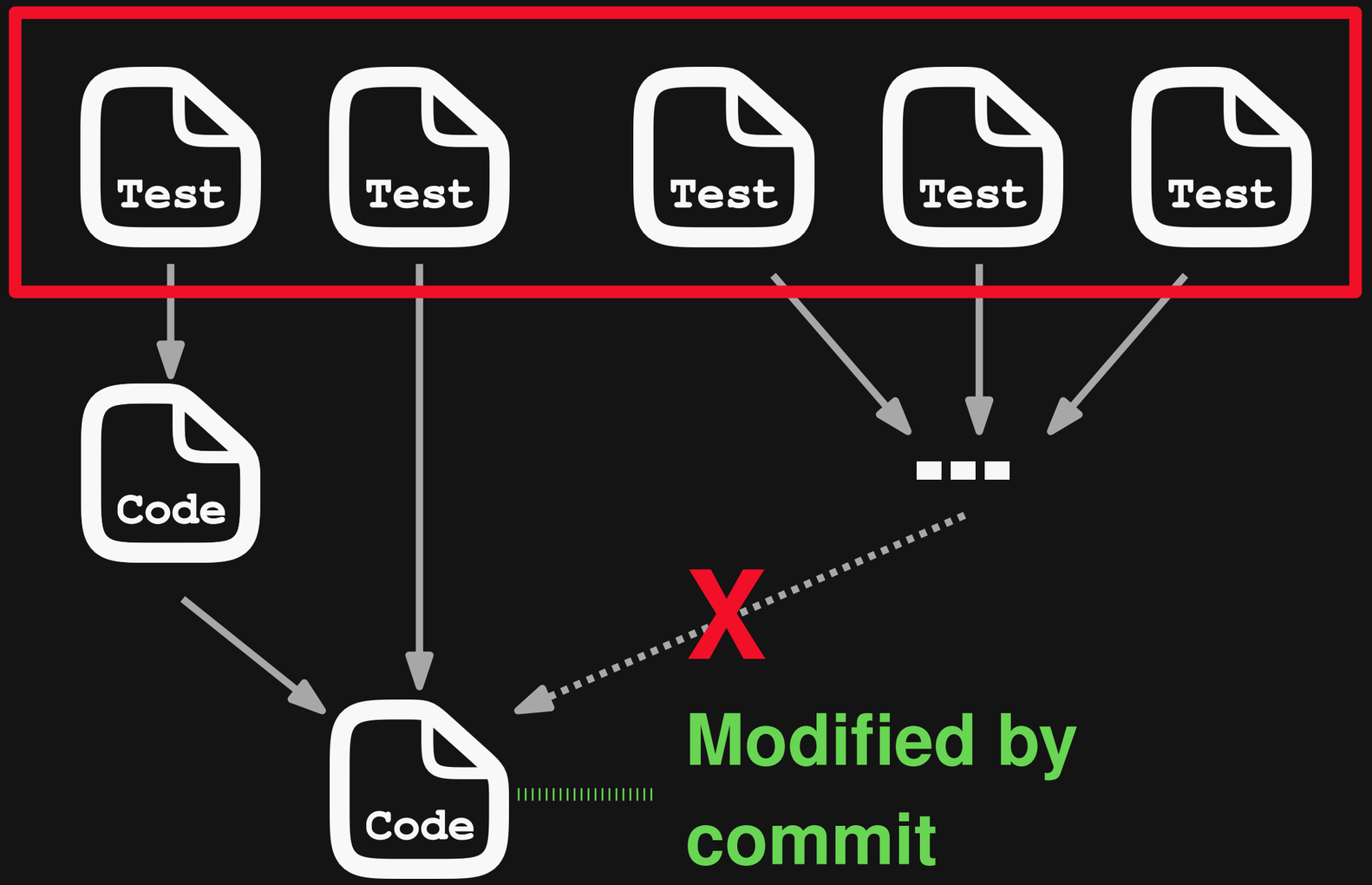
Regression Test Selection



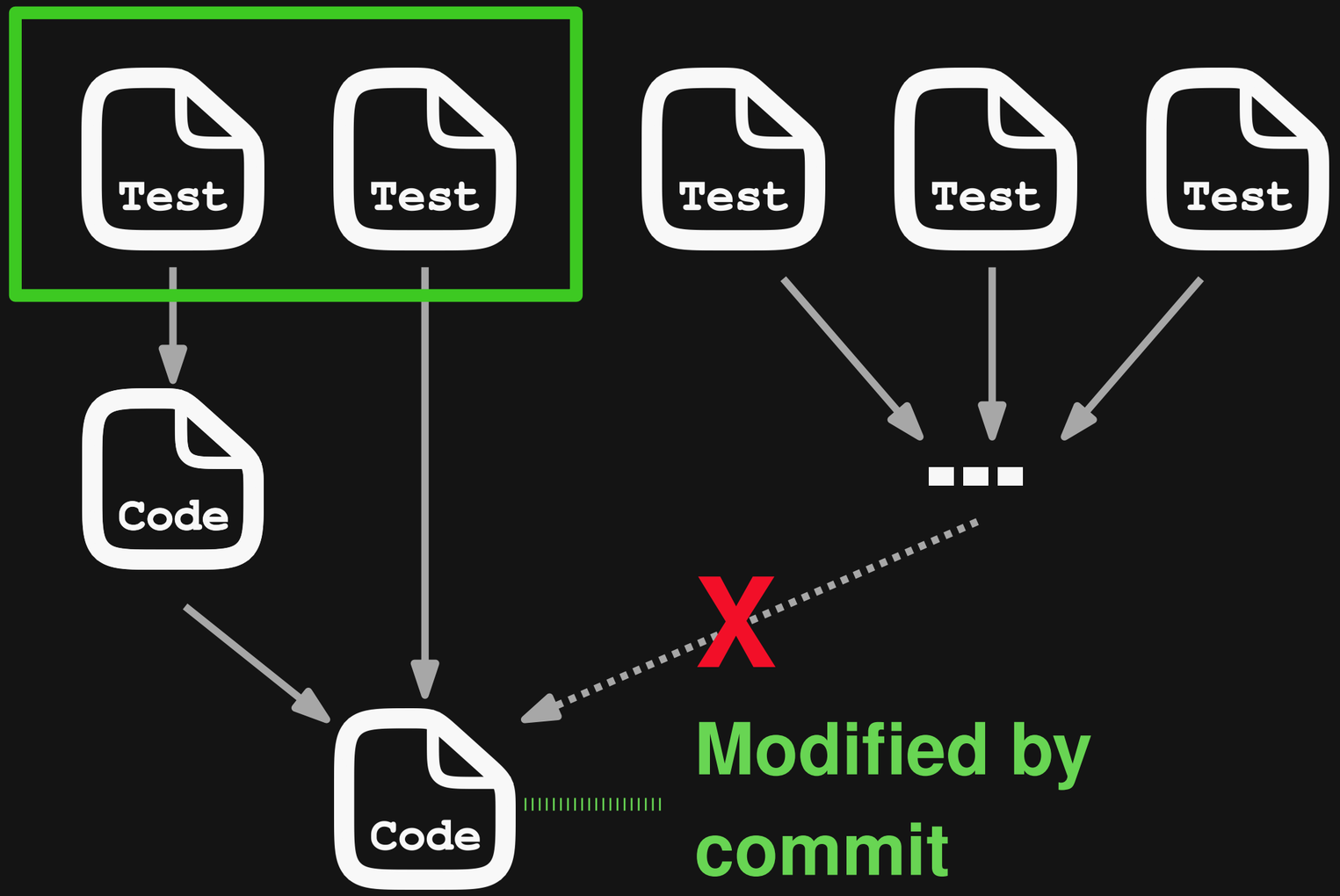
Regression Test Selection



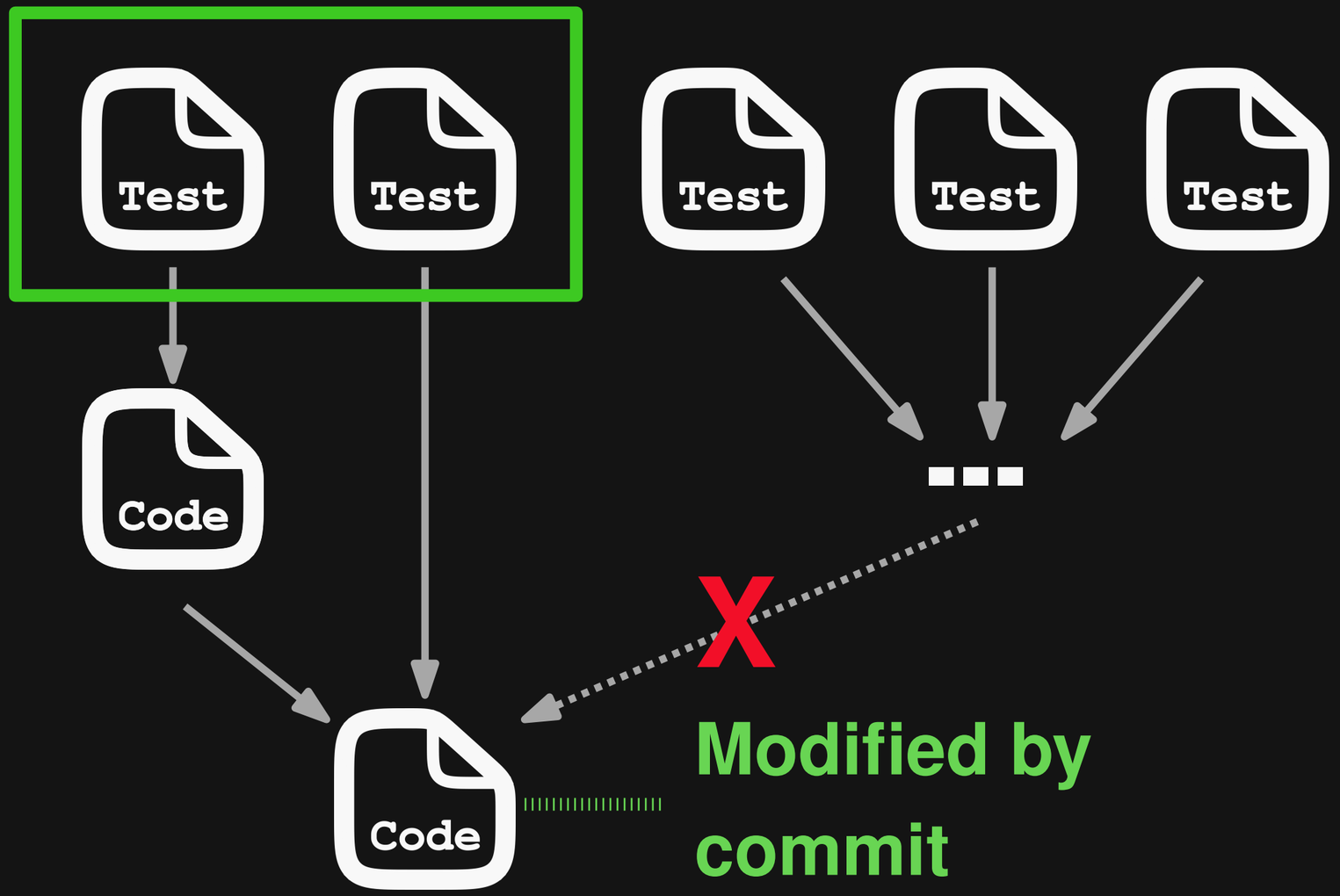
Regression Test Selection



Regression Test Selection



Regression Test Selection



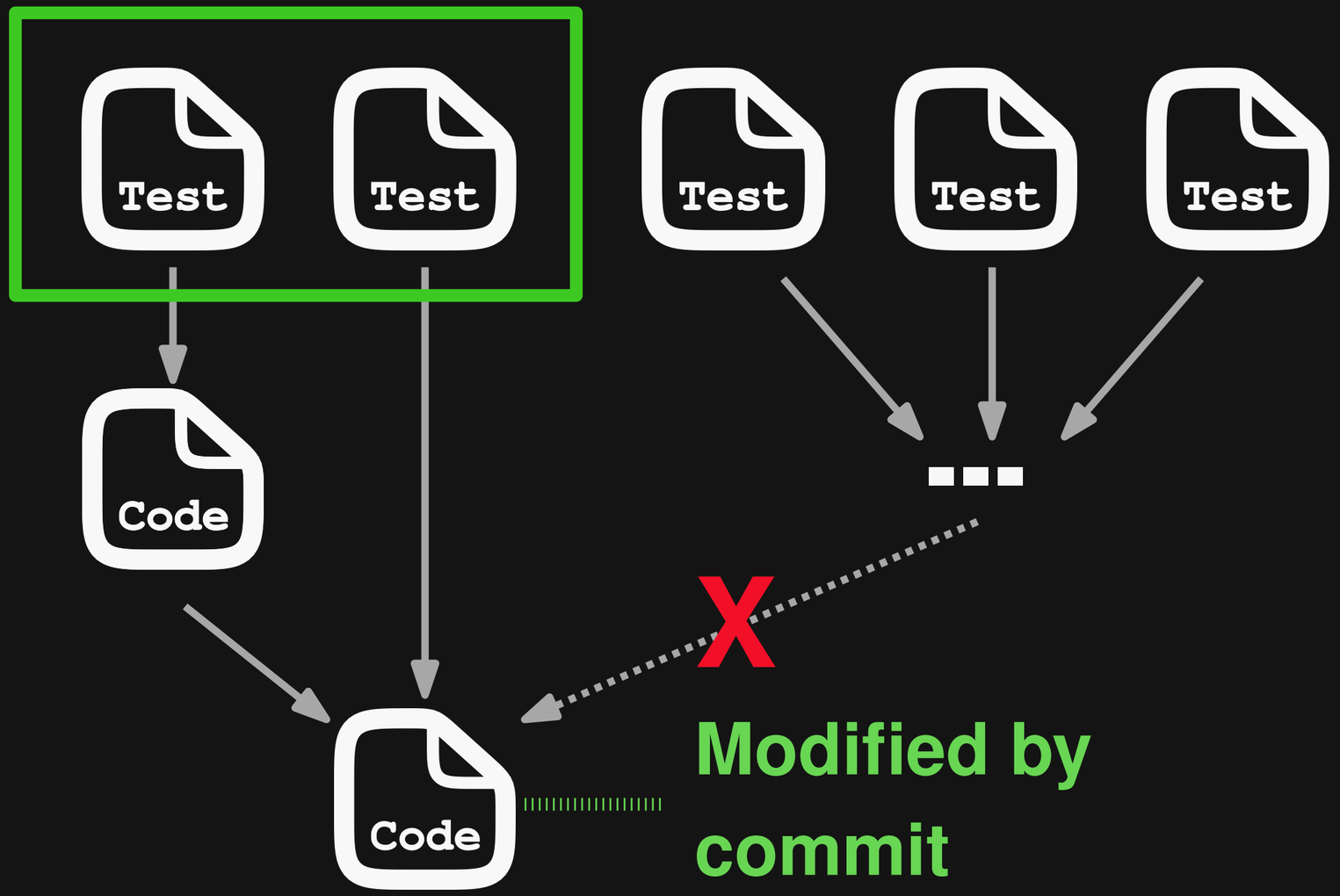
File-level:

- Cheaper
- Less effective

Function-level:

- More expensive
- More effective

Regression Test Selection



File-level:

- Cheaper
- Less effective

Function-level:

- More expensive
- More effective

+ Python-specific challenges

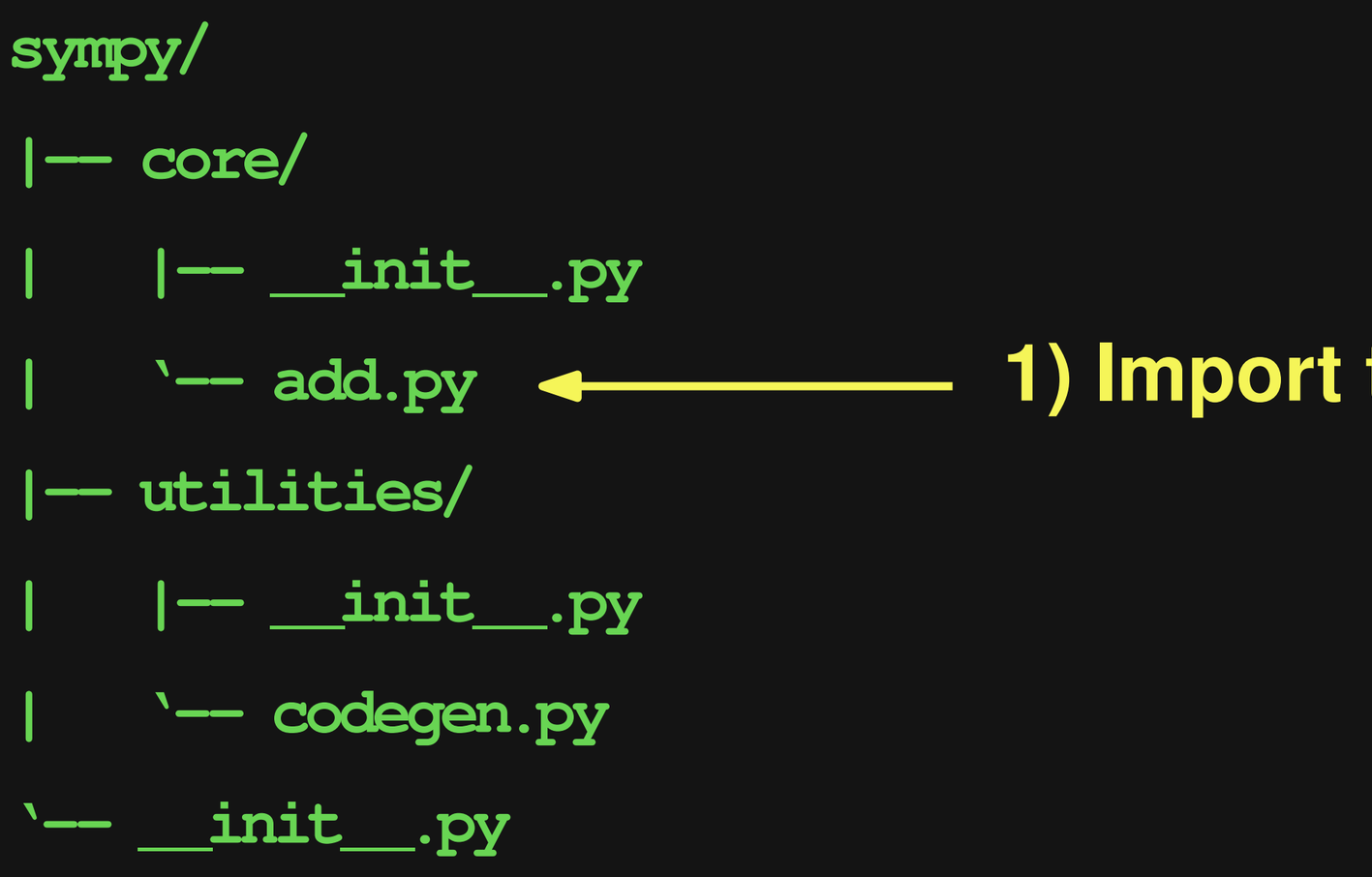
Challenge 1: Eager Importing in Python

Eager imports inflate file-level dependencies

```
sympy/
|-- core/
|    |-- __init__.py
|    `-- add.py
|-- utilities/
|    |-- __init__.py
|    `-- codegen.py
`-- __init__.py
```

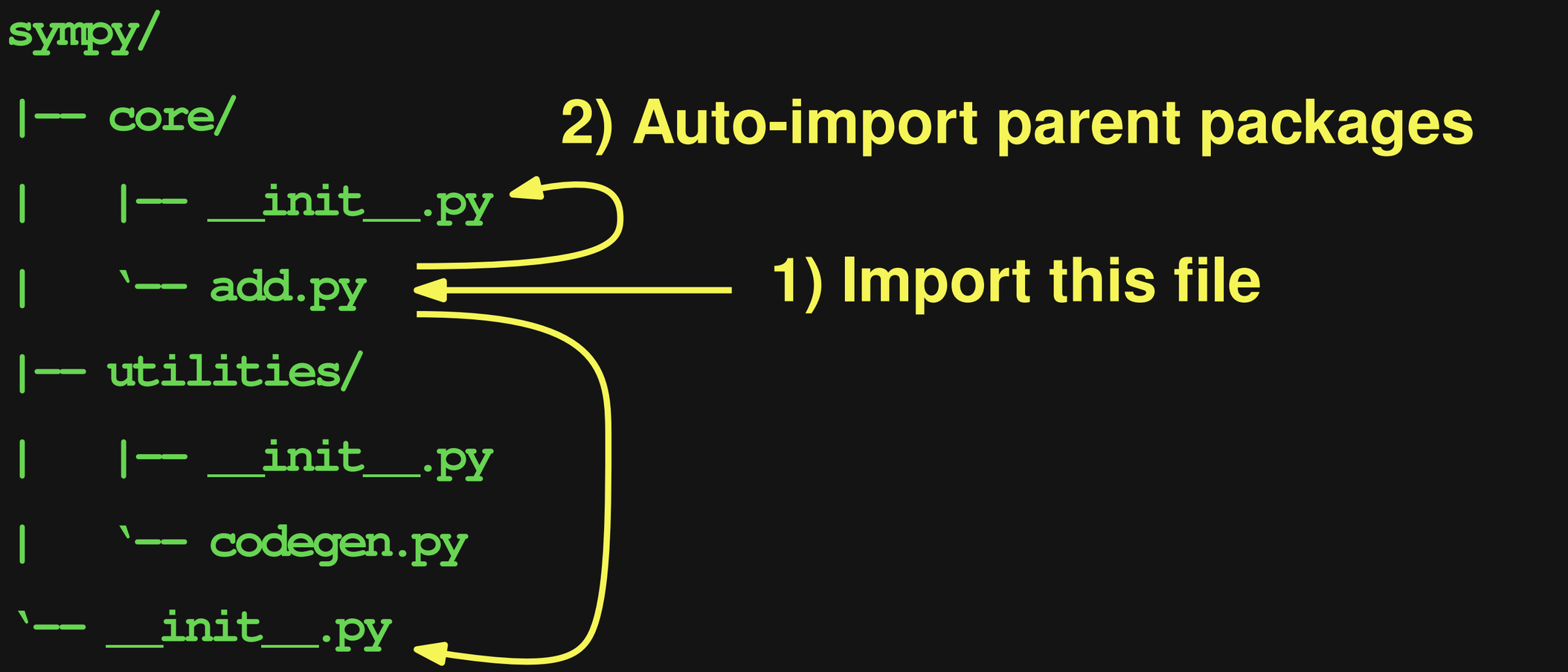
Challenge 1: Eager Importing in Python

Eager imports inflate file-level dependencies



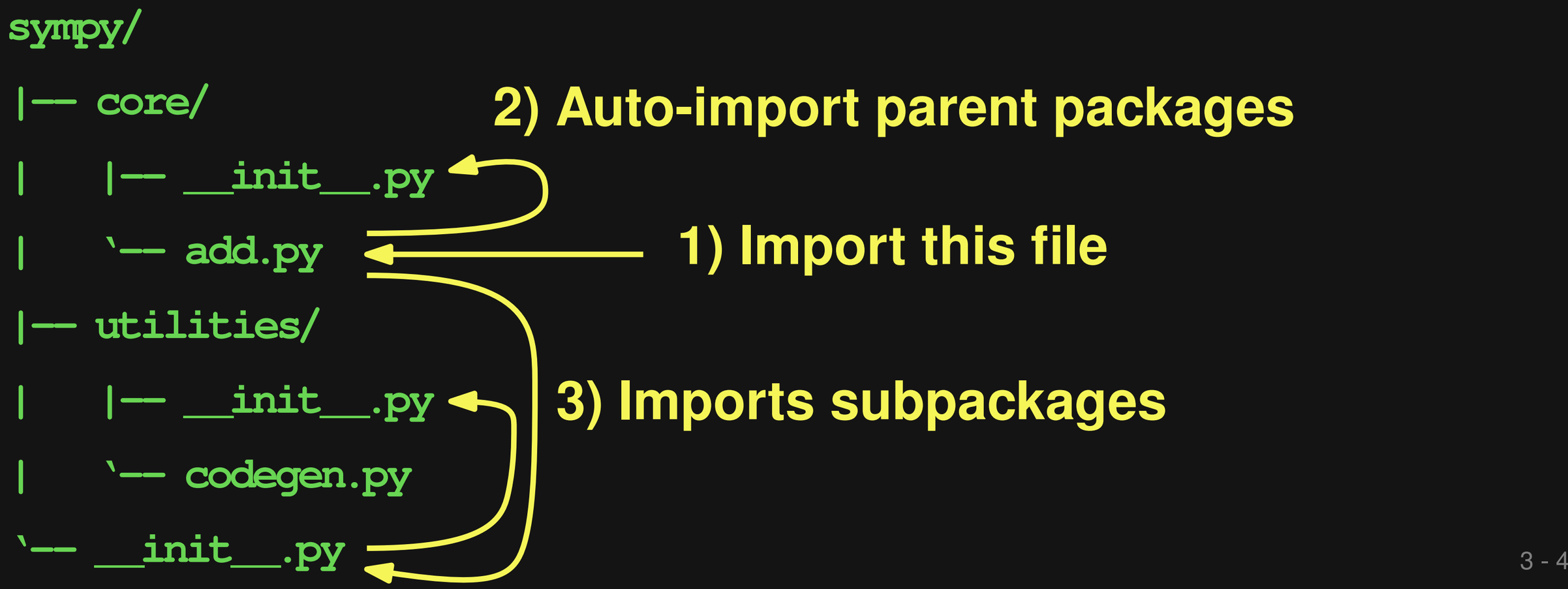
Challenge 1: Eager Importing in Python

Eager imports inflate file-level dependencies



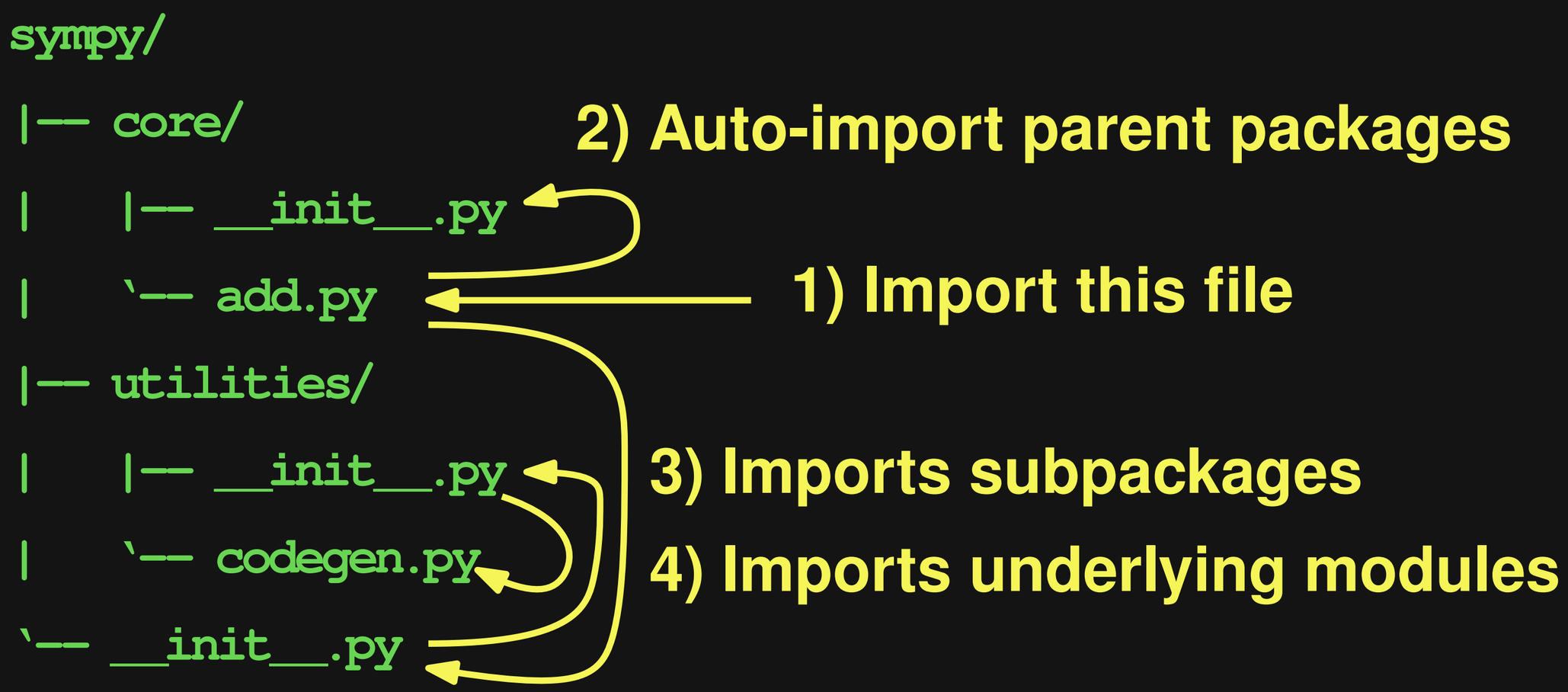
Challenge 1: Eager Importing in Python

Eager imports inflate file-level dependencies



Challenge 1: Eager Importing in Python

Eager imports inflate file-level dependencies



Challenge 2: Dynamic Typing

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```

Dynamic typing

$\Rightarrow$ Imprecise or incomplete call graph

$\Rightarrow$ Ineffective function-level dependency analysis

Challenge 2: Dynamic Typing

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```

Dynamic typing

$\Rightarrow$ Imprecise or incomplete call graph

$\Rightarrow$ Ineffective function-level dependency analysis

Do we really need to infer accurate types?

Idea: Name-Based Dependencies

- **Forget about types and call graphs**
- **Instead:**
  Fine-grained reasoning about dependencies between code elements based on ...
  - Names they define
  - Names they refer to

Back to Our Example

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```

After changes of which code elements do we need to run `test1`?

Back to Our Example

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```

Code elements:

Names:

Back to Our Example

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```

Code elements:

Names:

"A1"

"magnify"

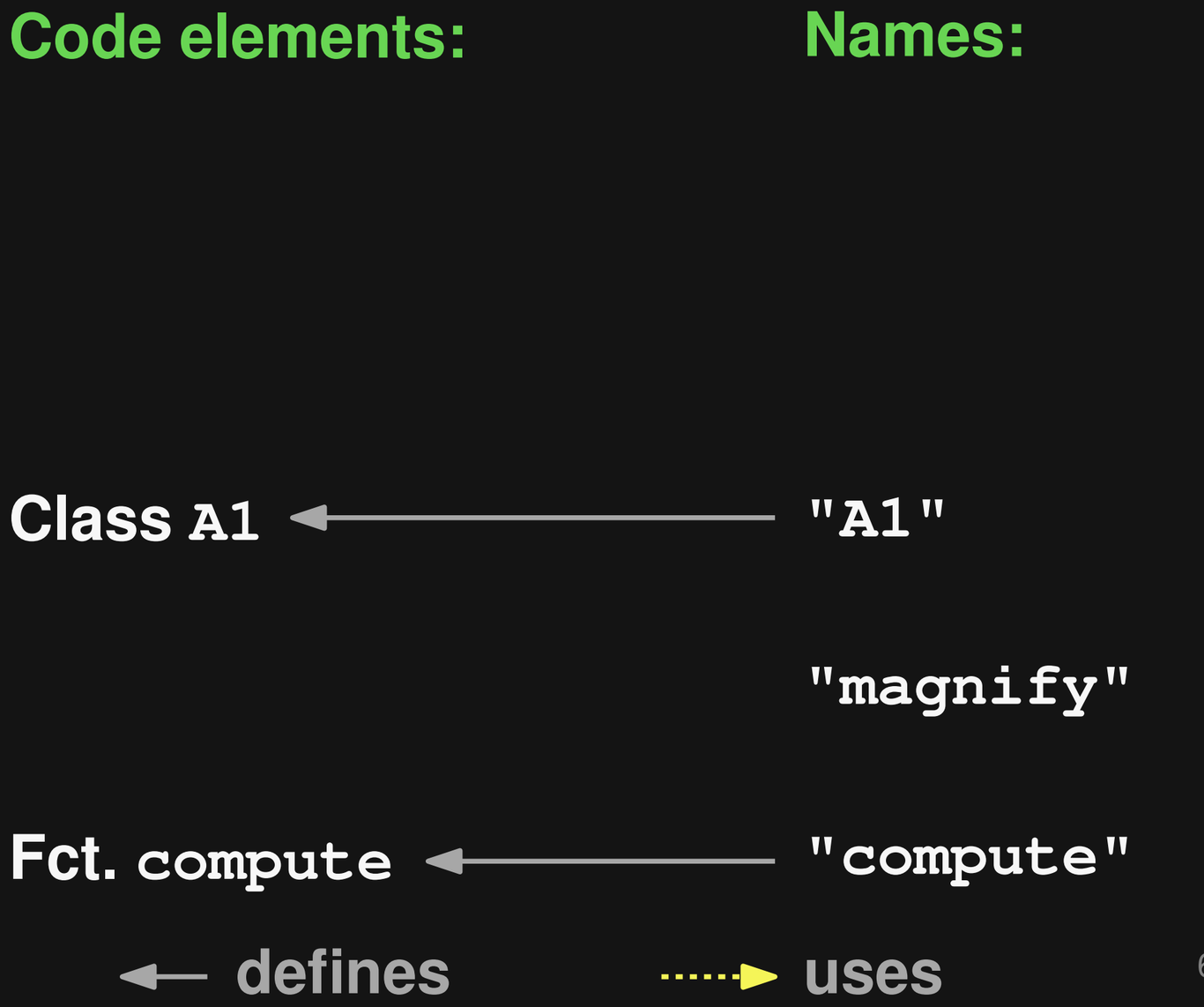
"compute"

← defines

⇢ uses

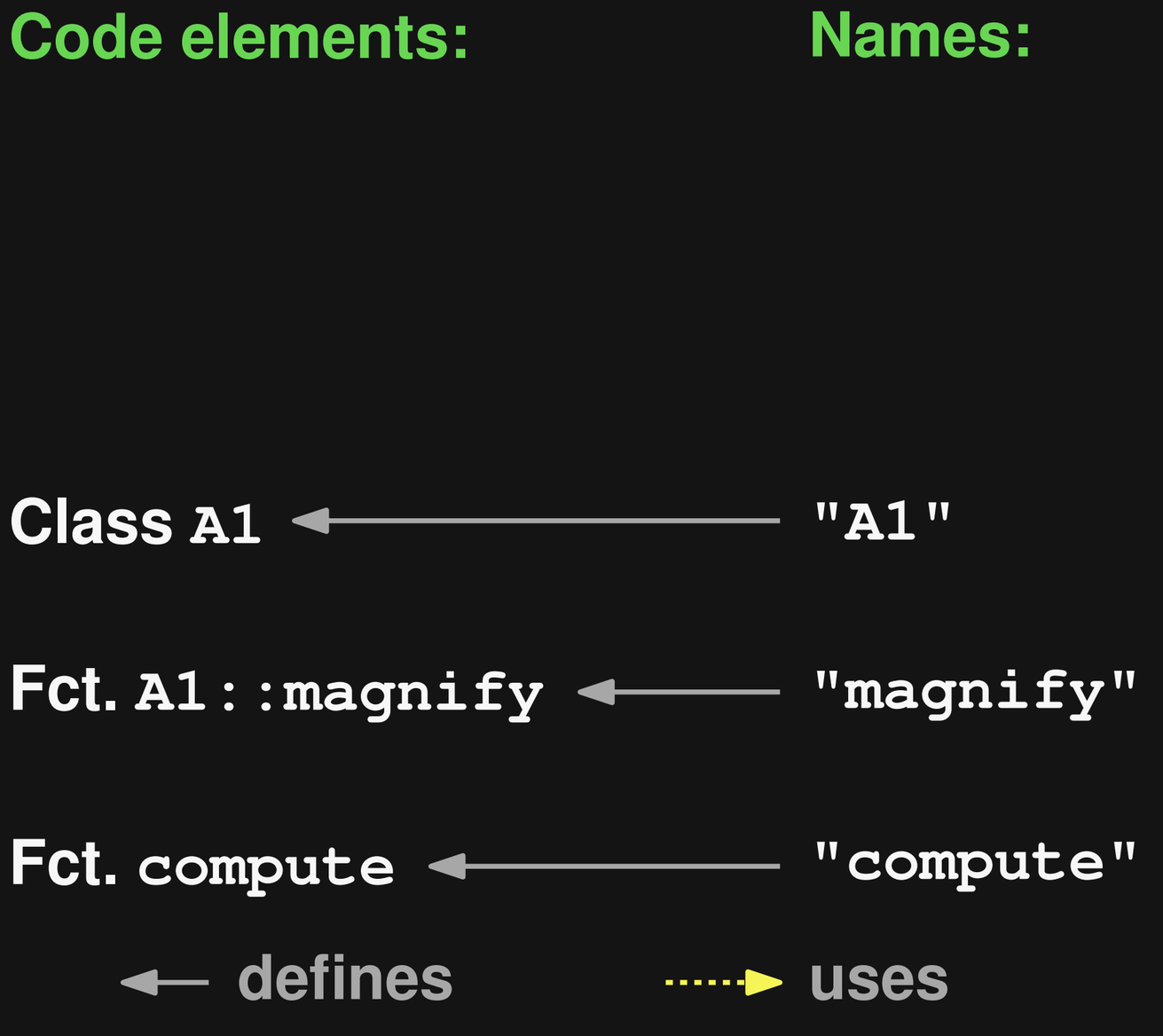
Back to Our Example

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```



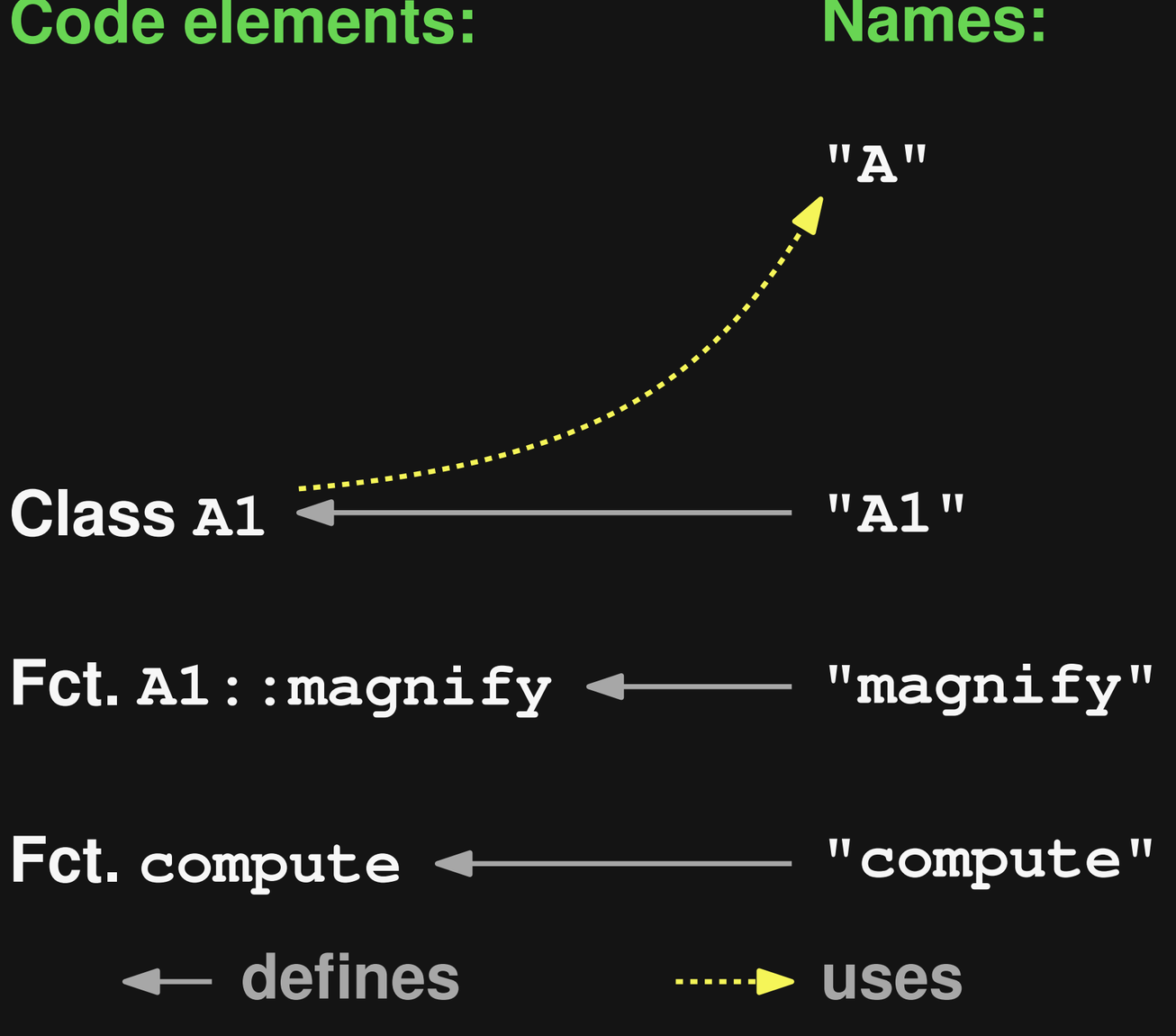
Back to Our Example

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```



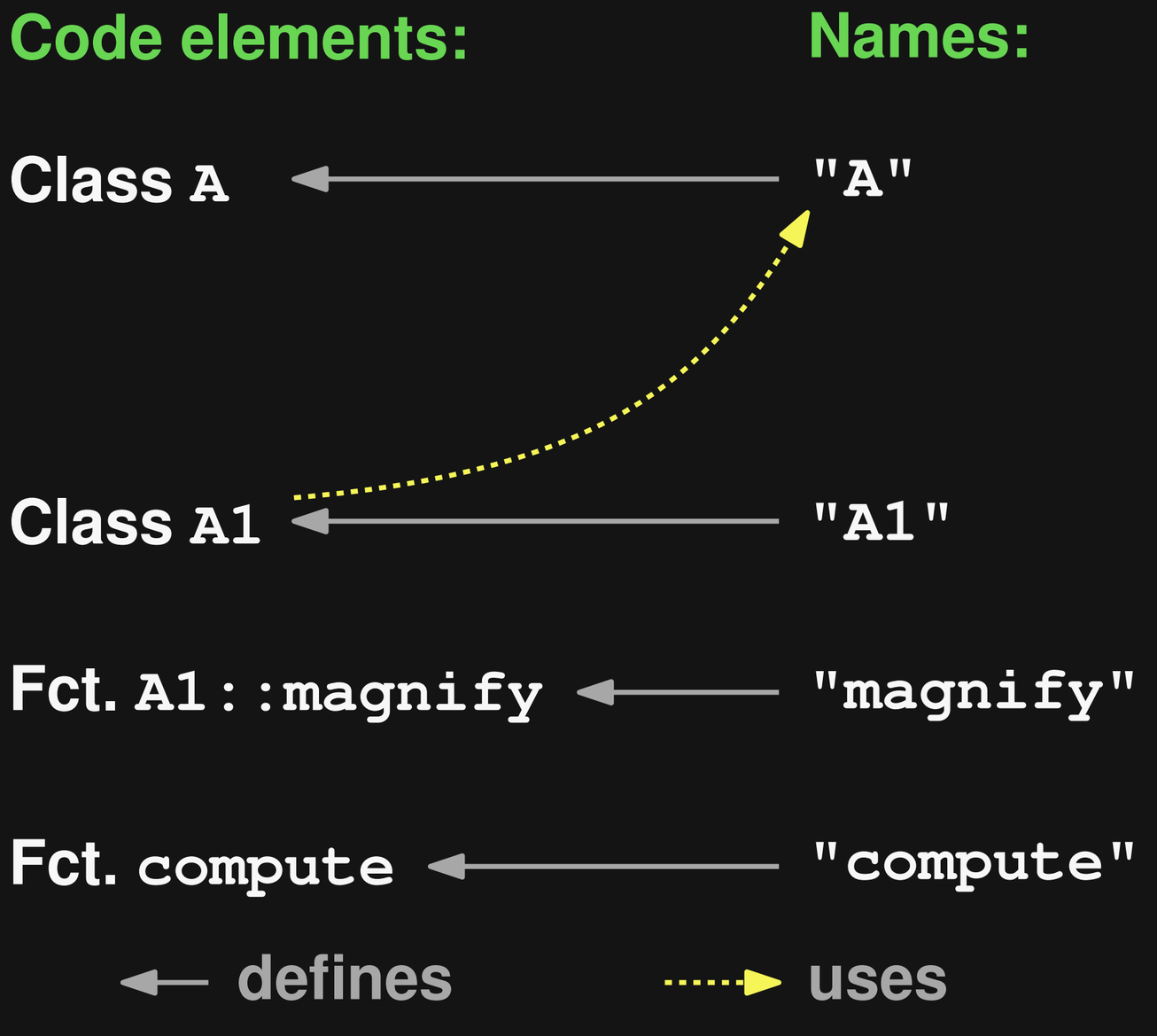
Back to Our Example

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```



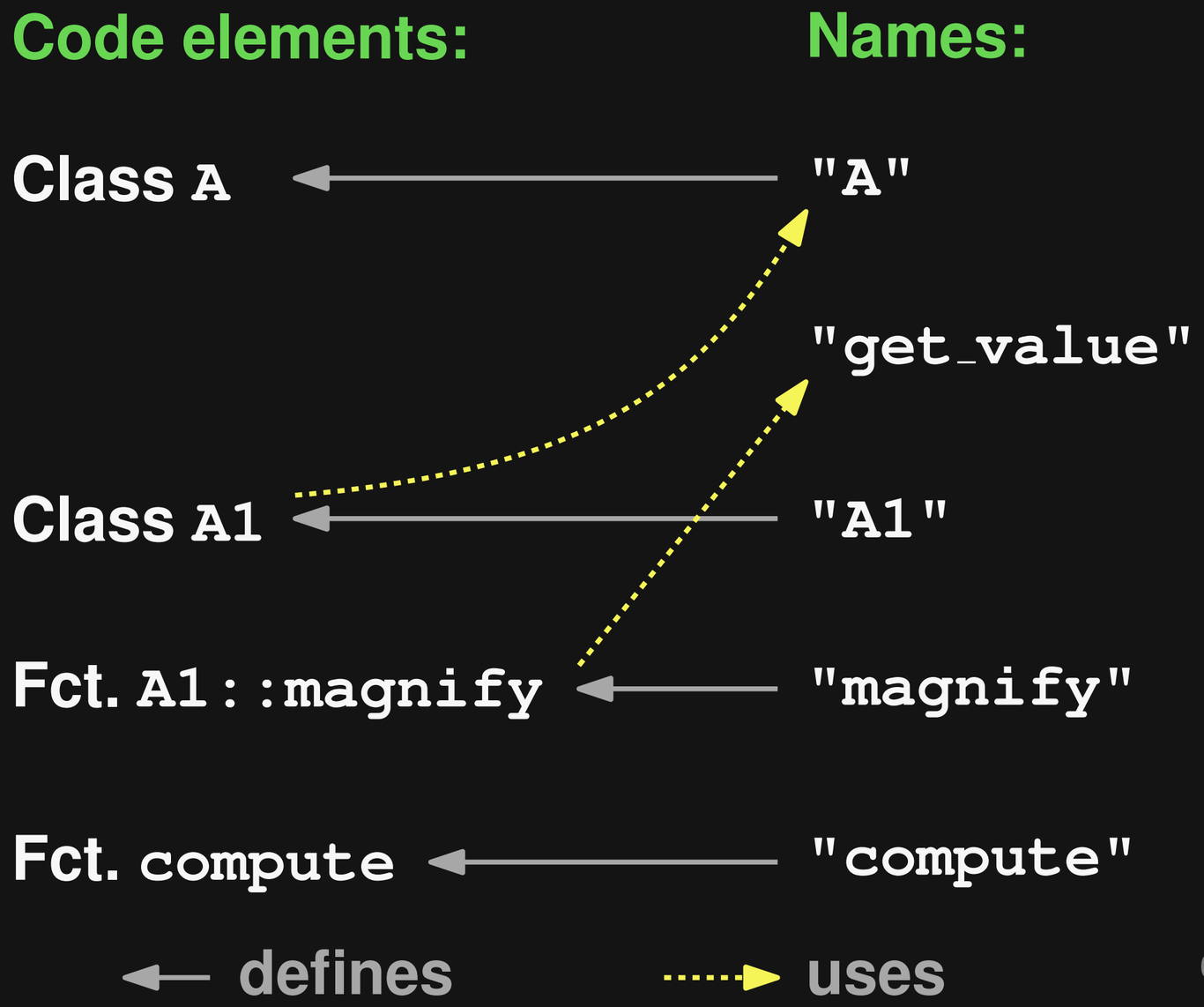
Back to Our Example

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```



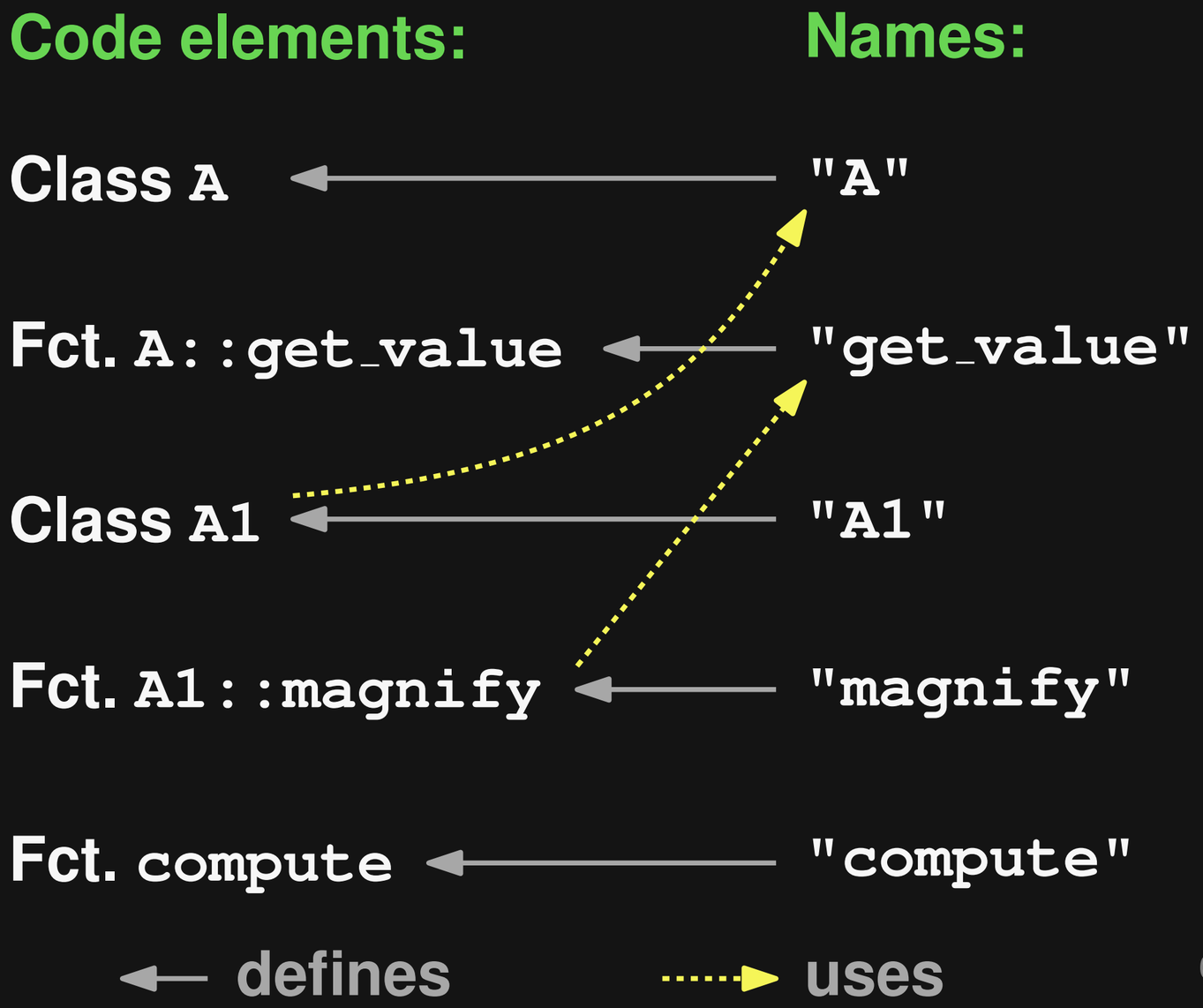
Back to Our Example

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```



Back to Our Example

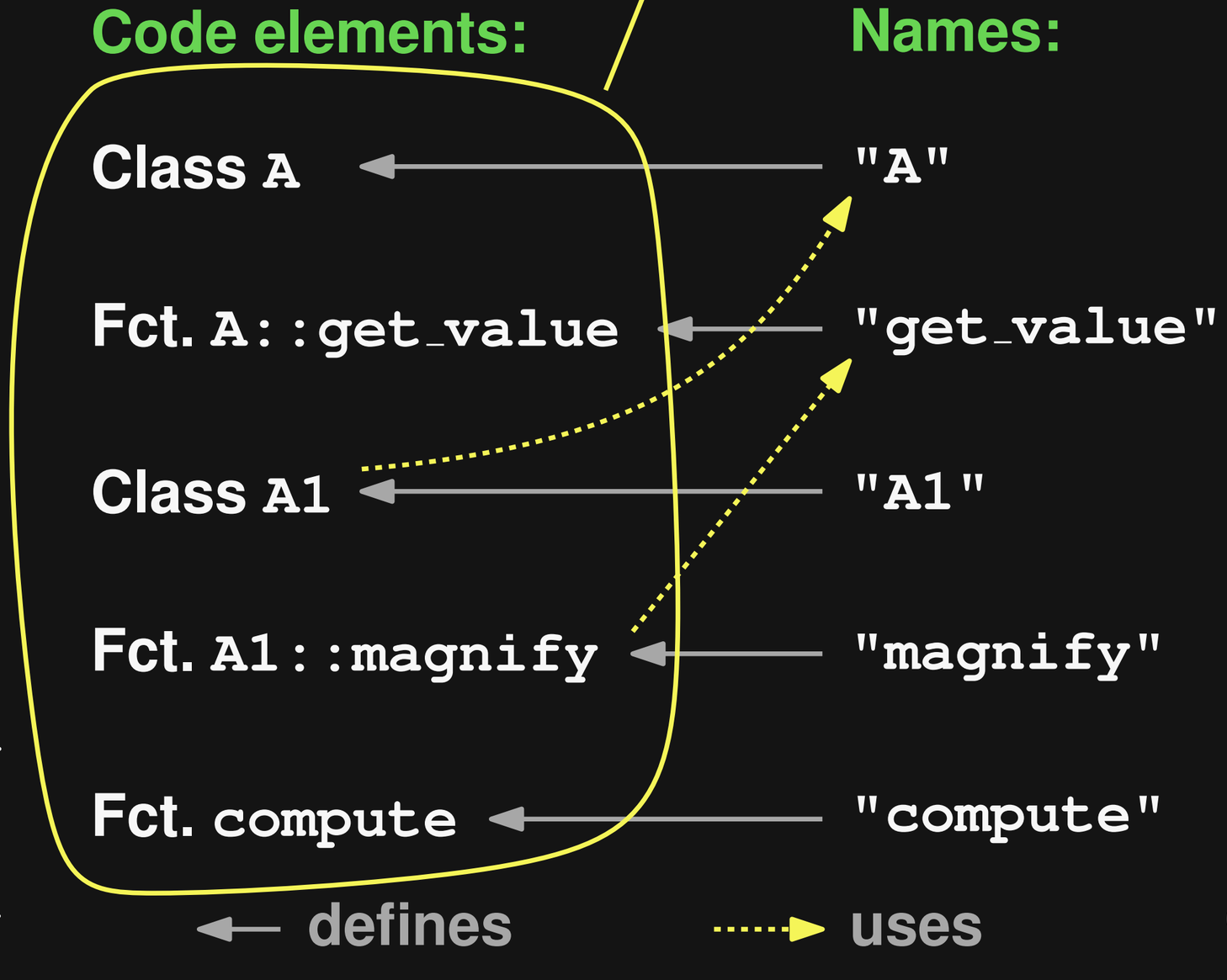
```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```



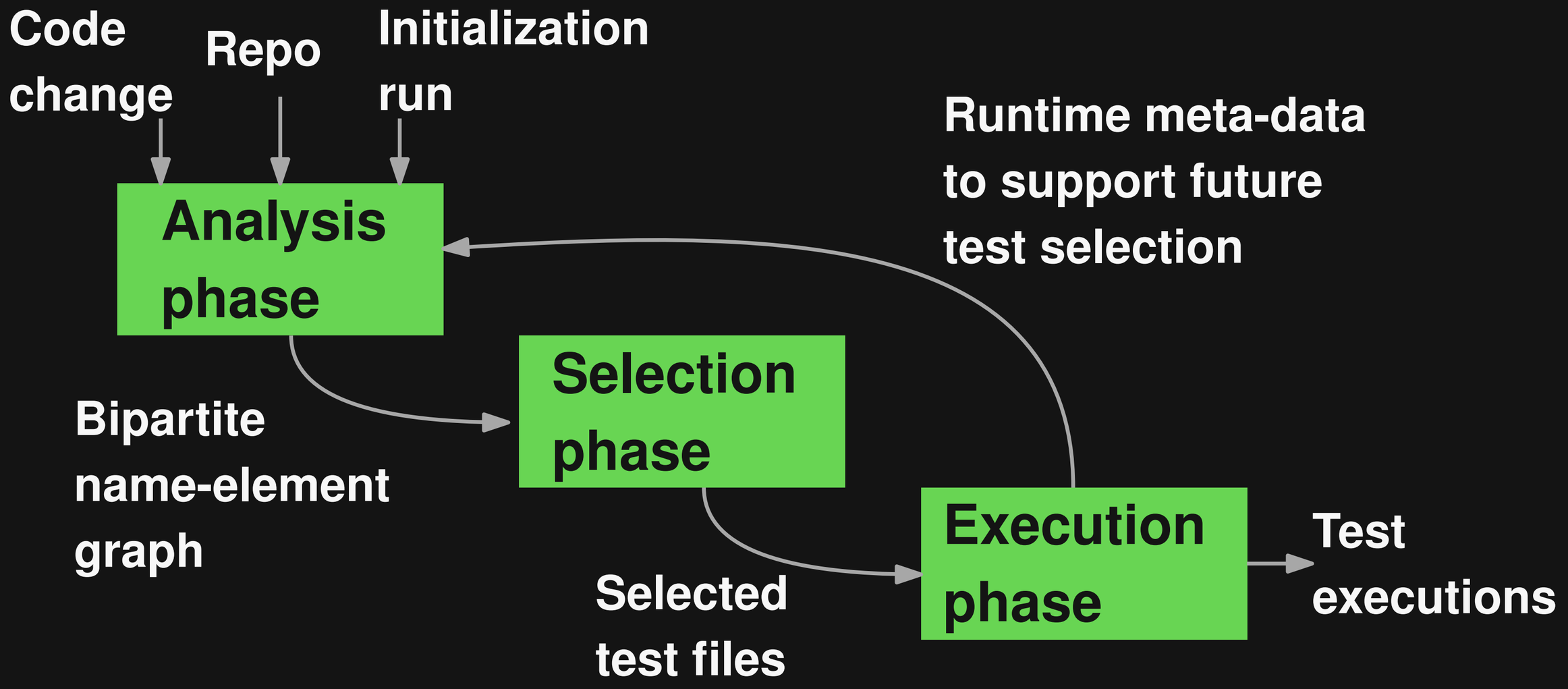
Back to Our Example

Answer: These code elements

```
class A:
  def get_value(self, i):
    return i

class A1(A):
  def magnify(self, i):
    return self.get_value(i) + 10

class A2(A):
  def magnify(self, i):
    return self.get_value(i) * 10

def compute(f, i):
  return f(i)

def test1():
  a1 = A1()
  assert compute(a1.magnify, 1) == 11

def test2():
  a = A()
  assert compute(a.get_value, 1) == 1
```



Overview of NameRTS



Step 1: Analysis Phase

Code element:

```
def test1():
    a1 = A1()
    assert compute(a1.magnify, 1) == 11
```

Bytecode analysis

Used external names:

`"A1", "compute", "magnify"`

Add to bipartite name-element graph

Fct. `test1` → `"A1"`, `"compute"`, `"magnify"`

Step 2: Selection Phase

Test selection

=

Reachability computation over the name-element graph

- **Computed via stack-based algorithm**
- **Test file gets selected if it can reach any modified code element**

Step 3: Execution Phase

- **Instrument** code
- **Execute** all selected tests
- Execution **collects meta-data** to be used for future test selection runs

Additional Optimizations

- Reason about **imports** via an import graph
- Handle **extremely common names** by pruning “critical functions”
- **Context-aware** name-element matching
- **Dynamic analysis** of operator overloading, reflection, and `importlib`-based imports

Additional Optimizations

- Reason about **imports** via an import graph
- Handle **extremely common names** by pruning "critical functions"
- **Context-aware** name-element matching
- **Dynamic anal**
  and `importl`

Distinguish three kinds of used names:

- Self attributes (e.g., `self.func()`)
- Non-attributes (e.g., `func()`)
- Ambiguous (e.g., `x.func()`)

Experimental Setup

- **500 commits from 10 projects (12k-400k LoC)**
  - Full test run takes 1–23 minutes
  - Graphs: 1k–24.4k name nodes + 1.7k–60.2k code-element nodes
- **Ground truth construction**
  - Instrument modified code elements
  - Execute all tests
  - Identify tests that exercise modified code

RQ1: Effectiveness and Safety

	Test reduction	Time reduction	Safe rate
NameRTS	69.9%	45.6%	99.6%

For comparison:
Perfect selection
achieves 80.7%

RQ1: Effectiveness and Safety

	Test reduction	Time reduction	Safe rate
EkstaP *	7.4%	3.1%	100.0%
BabelRTS #	28.4%	21.9%	76.6%
NameRTS	69.9%	45.6%	99.6%

For comparison:
Perfect selection
achieves 80.7%

* Python adaption of *Practical regression test selection with dynamic file dependencies*, Gligoric et al., 2015

\# *BabelRTS: Polyglot Regression Test Selection*, Maurina et al., 2025

RQ2: Computational Overhead

Project	RunAll(s)	Time (% of RunAll)				
		Test	Runtime	Select	Init	Total
sympy	69555.2	46.22%	10.64%	7.43%	3.05%	67.34%
sklearn	36277.0	27.04%	1.31%	2.33%	2.51%	33.19%
mpl	21548.0	53.83%	3.92%	2.95%	2.30%	63.00%
dask	43395.8	62.73%	4.32%	1.58%	2.22%	70.85%
xarray	36311.8	41.87%	5.72%	1.04%	2.25%	50.88%
sphinx	6647.6	66.95%	12.15%	8.24%	2.77%	90.11%
pylint	5132.8	63.12%	1.02%	3.35%	2.21%	69.70%
seaborn	28110.5	21.72%	0.20%	0.43%	1.96%	24.31%
pvlib	2832.3	12.17%	3.56%	3.16%	2.52%	21.41%
loguru	3845.3	48.07%	1.65%	1.66%	1.96%	53.34%
Avg.	25365.6	44.37%	4.45%	3.22%	2.38%	54.41%

Total time spent across 50 commits per project

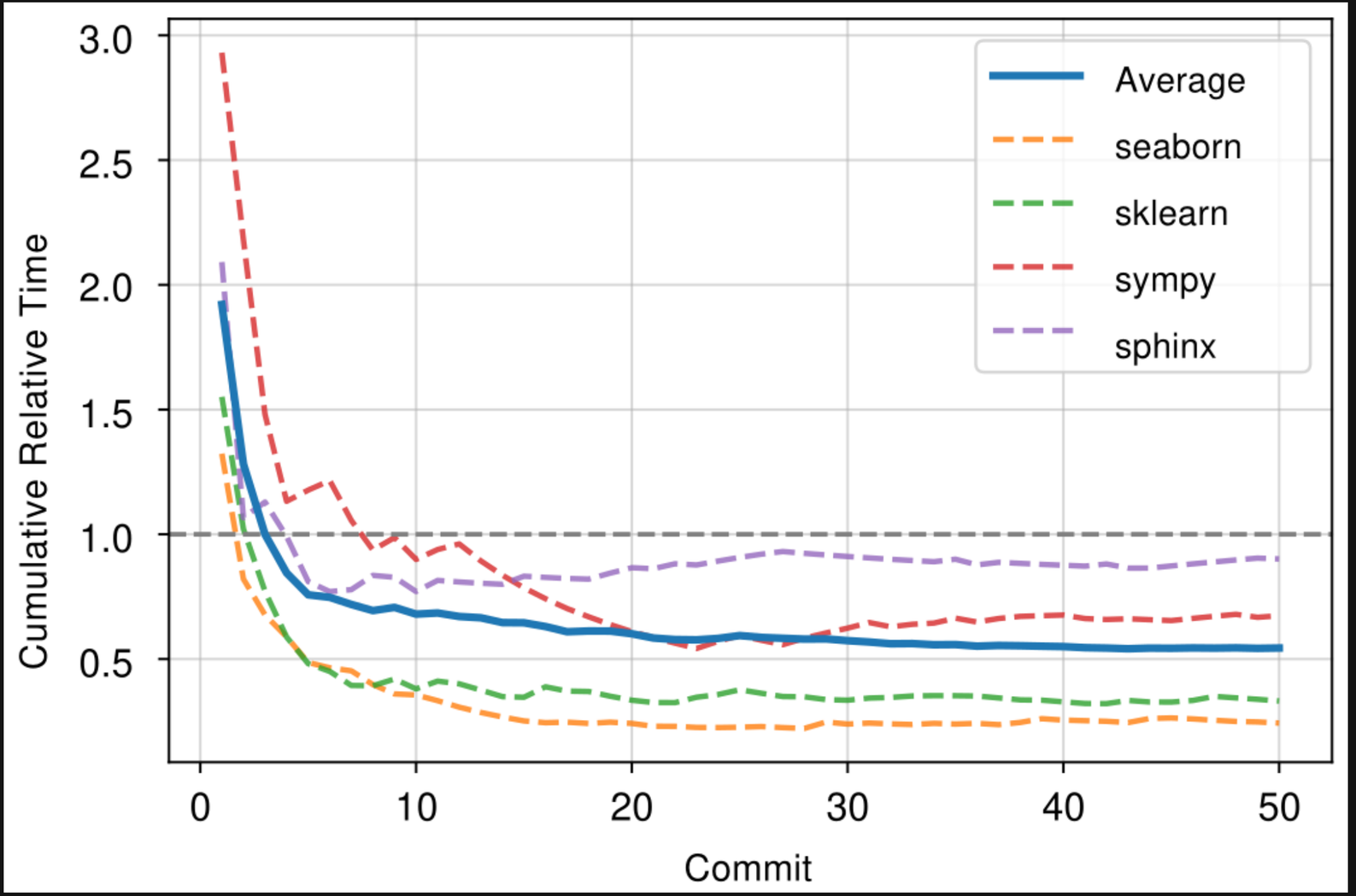
RQ2: Computational Overhead

Project	RunAll(s)	Time (% of RunAll)				
		Test	Runtime	Select	Init	Total
sympy	69555.2	46.22%	10.64%	7.43%	3.05%	67.34%
sklearn	36277.0	27.04%	1.31%	2.33%	2.51%	33.19%
mpl	21548.0	53.83%	3.92%	2.95%	2.30%	63.00%
dask	43395.8	62.73%	4.32%	1.58%	2.22%	70.85%
xarray	36311.8	41.87%	5.72%	1.04%	2.25%	50.88%
sphinx	6647.6	66.95%	12.15%	8.24%	2.77%	90.11%
pylint	5132.8	63.12%	1.02%	3.35%	2.21%	69.70%
seaborn	28110.5	21.72%	0.20%	0.43%	1.96%	24.31%
pvlib	2832.3	12.17%	3.56%	3.16%	2.52%	21.41%
loguru	3845.3	48.07%	1.65%	1.66%	1.96%	53.34%
Avg.	25365.6	44.37%	4.45%	3.22%	2.38%	54.41%

Total time spent across 50 commits per project

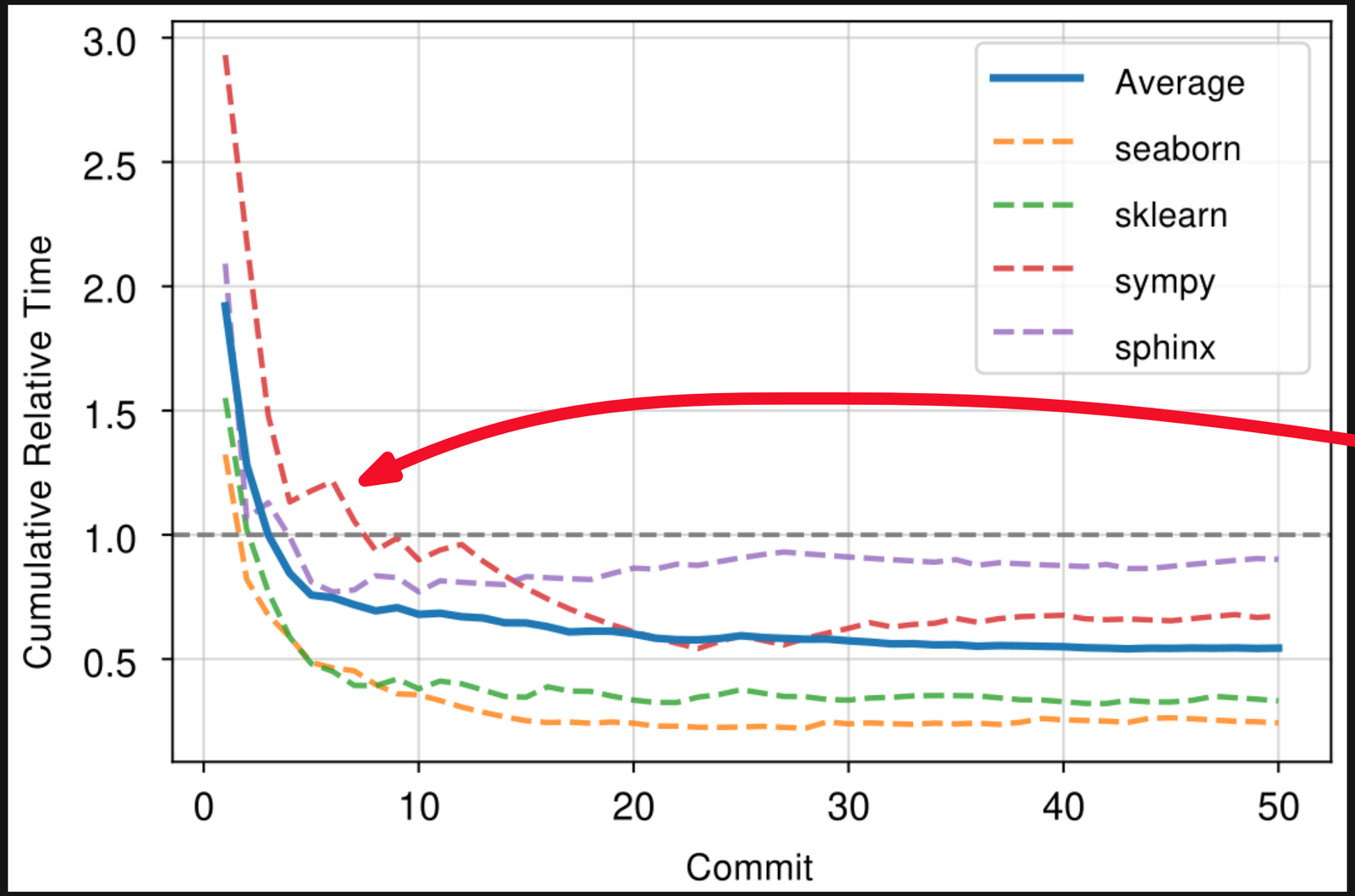
10.04% overhead relative to full test-suite execution

RQ2: Computational Overhead



Cumulative relative testing time

RQ2: Computational Overhead



Cumulative relative testing time

NameRTS pays for its initialization after 5 commits

Conclusion

- **First Python regression test selection technique based on fine-grained dependency analysis**
- **Name-based dependency propagation via name-element graph**
  - Avoids precise type inference and call graph construction
- **Saves 45.6% of test execution time while running all affected tests in 99.6% of commits**