Automated Clone Elimination in Python Tests

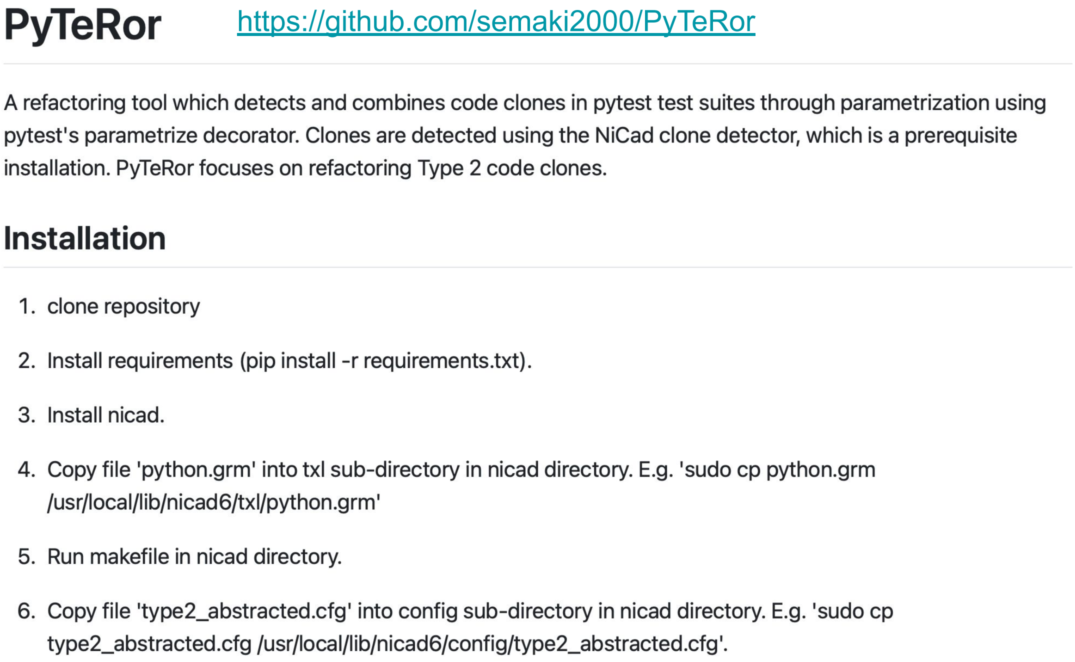
Sebastian Kingston (UiO), Volker Stolz (UiO & HVL),
Violet Ka I Pun (HVL)

UNIVERSITY OF OSLO

Western Norway University of Applied Sciences

Outline

- Motivation: code-clones
- Some background
- PyTeRor: an example
- Experimental results
- Implementation of PyTeRor
- Discussion
- Related work
- Future work & Conclusion



PyTeRor

https://github.com/semaki2000/PyTeRor

A refactoring tool which detects and combines code clones in pytest test suites through parametrization using pytest's parametrize decorator. Clones are detected using the NiCad clone detector, which is a prerequisite installation. PyTeRor focuses on refactoring Type 2 code clones.

Installation

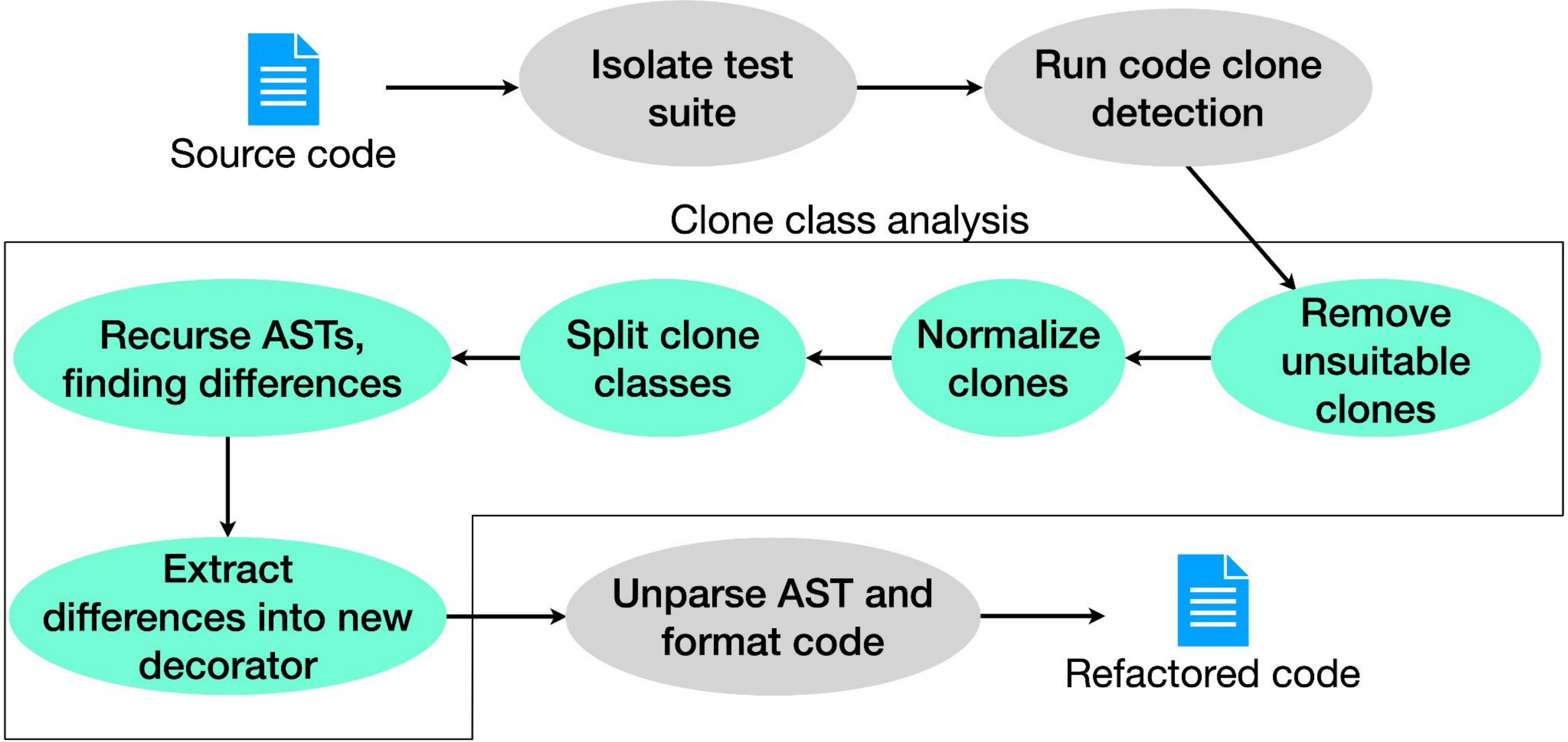
1. clone repository
2. Install requirements (pip install -r requirements.txt).
3. Install nicad.
4. Copy file 'python.grm' into txl sub-directory in nicad directory. E.g. 'sudo cp python.grm /usr/local/lib/nicad6/txl/python.grm'
5. Run makefile in nicad directory.
6. Copy file 'type2_abstracted.cfg' into config sub-directory in nicad directory. E.g. 'sudo cp type2_abstracted.cfg /usr/local/lib/nicad6/config/type2_abstracted.cfg'.

Motivation

- Eldh reports 30%-50% code clone overlap in test suites
    - Some suites containing up to 80% overlap
- Test code often given less attention
- Python - currently most “popular” programming language - TIOBE Index
    - Pytest - popular Python testing framework
    - Built-in parametrization
- Type 2 clones
    - Same structure, different values
    - Candidates for parametrization

Sigrid Eldh. ‘On Technical Debt in Software Testing - Observations from Industry’.
In: Leveraging Applications of Formal Methods, Verification and Validation.
Software Engineering. ISoLA 2022.

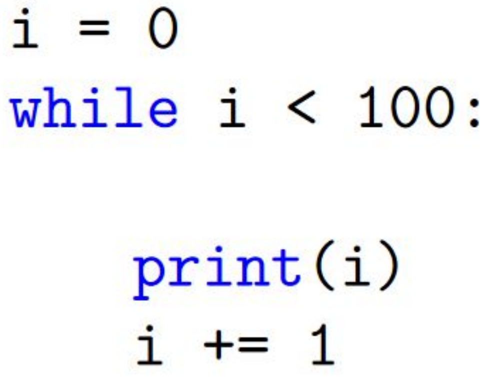
PyTeRor


Source code
Isolate test suite
Run code clone detection
Clone class analysis
Remove unsuitable clones
Normalize clones
Split clone classes
Recurse ASTs, finding differences
Extract differences into new decorator
Unparse AST and format code
Refactored code

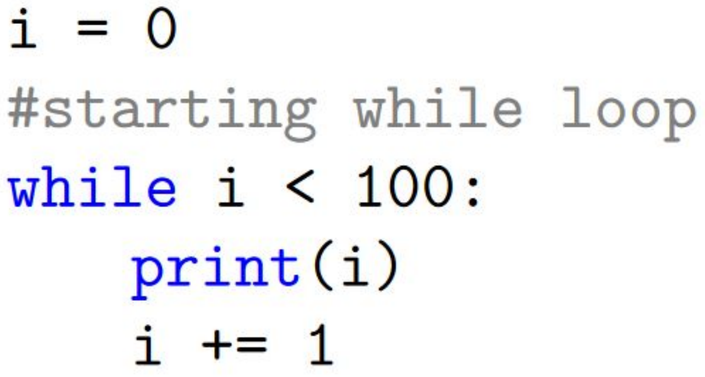
Background - Code clones

- Code fragment
  - Single continuous piece of code
- Clone pair
  - Two similar/duplicate code fragments
- Clone class
  - Group of similar/duplicate code fragments
- Code clone types
  - Defined by level of similarity between clones

Background - Type 1 code clones

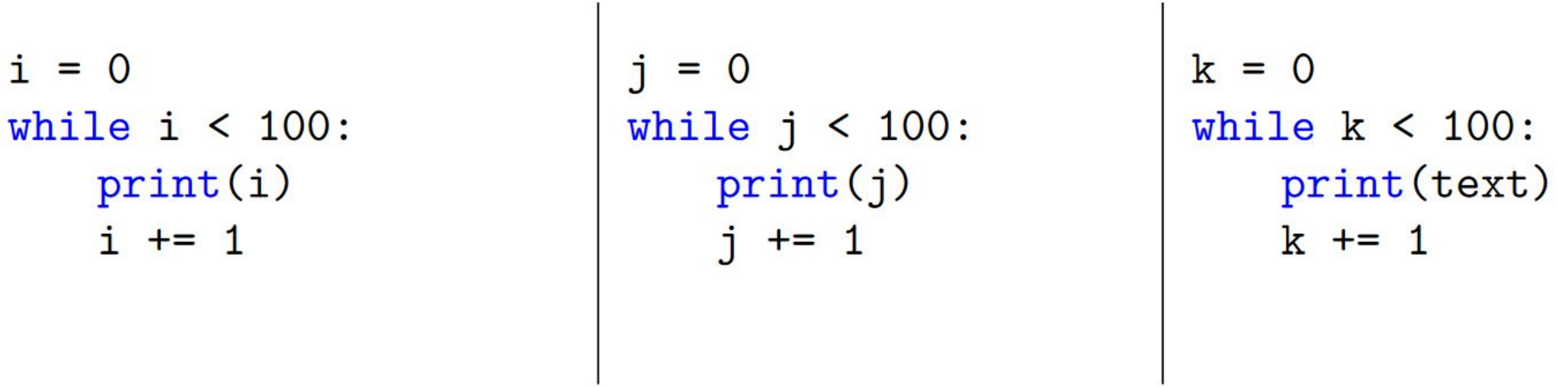


```
i = 0
while i < 100:

    print(i)
    i += 1
```



```
i = 0
#starting while loop
while i < 100:
    print(i)
    i += 1
```

Background - Type 2 code clones



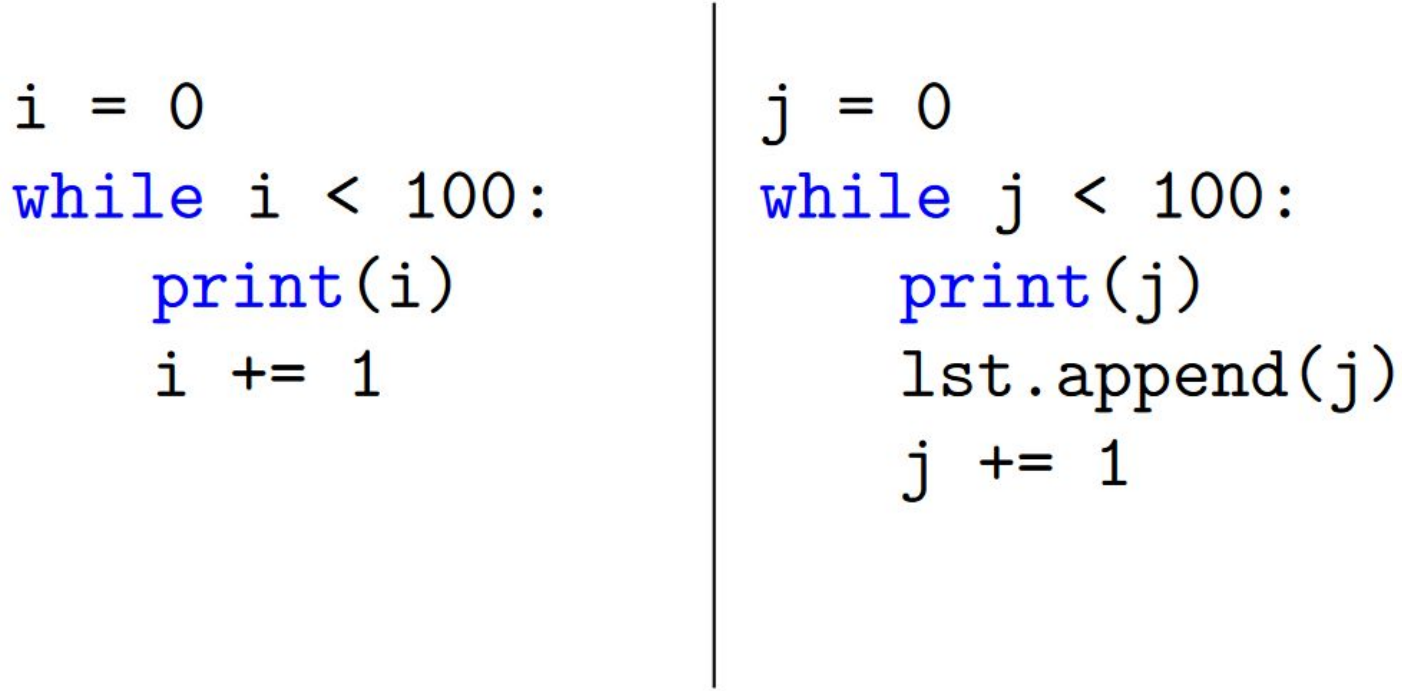
```
i = 0
while i < 100:
    print(i)
    i += 1
```

```
j = 0
while j < 100:
    print(j)
    j += 1
```

```
k = 0
while k < 100:
    print(text)
    k += 1
```

Consistent or *blind* clone: consistent renaming of identifiers?

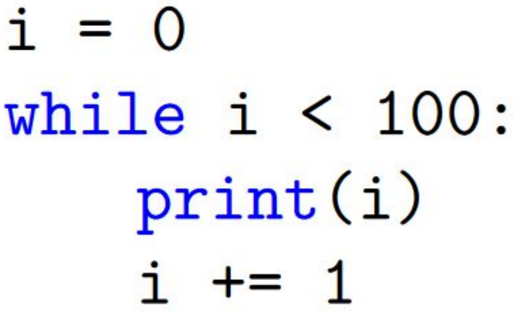
Background - Type 3 code clones



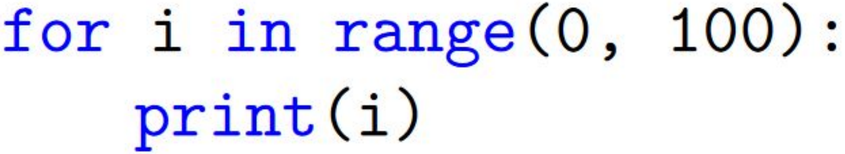
```
i = 0
while i < 100:
    print(i)
    i += 1
```

```
j = 0
while j < 100:
    print(j)
    lst.append(j)
    j += 1
```

Background - Type 4 code clones ("semantic clone")



```
i = 0
while i < 100:
    print(i)
    i += 1
```



```
for i in range(0, 100):
    print(i)
```

Background - NiCad Clone Detector

- Well-known clone detector, Python support
- Easily configurable
- Automated Detection of Near-Miss Intentional Clones (types 1, 2, 3)
- Steps:
  1. Parsing and extracting fragments at given granularity (functions, blocks)
  2. Renaming, filtering and normalization of extracted fragments
  3. Comparing extracted fragments to identify clones

txl NiCad clone detector

https://www.txl.ca/txl-nicaddownload.html

Background - Pytest

```python
def add(a, b):
    return a + b
```

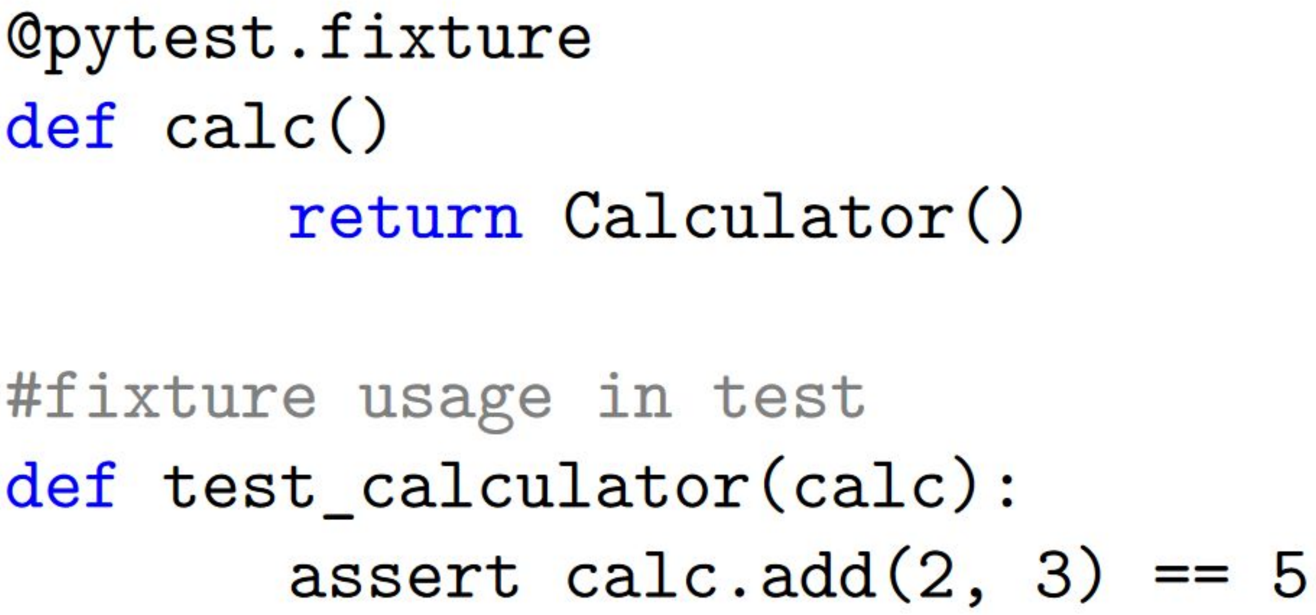
```python
def test_add():
    res = add(1, 2)
    assert res == 3
```

a) The function add

b) A simple pytest test of function add

Background - Pytest fixtures

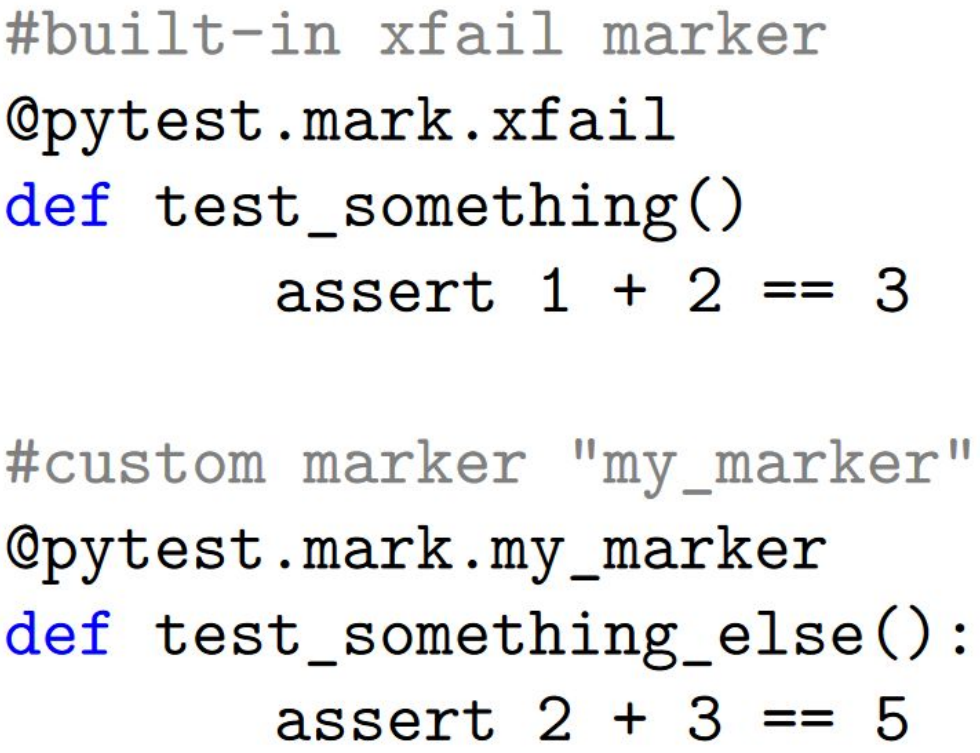
- Set-up functions
- Invocated by being supplied as formal parameter for pytest test



```
@pytest.fixture
def calc()
        return Calculator()

#fixture usage in test
def test_calculator(calc):
        assert calc.add(2, 3) == 5
```

Background - Pytest markers

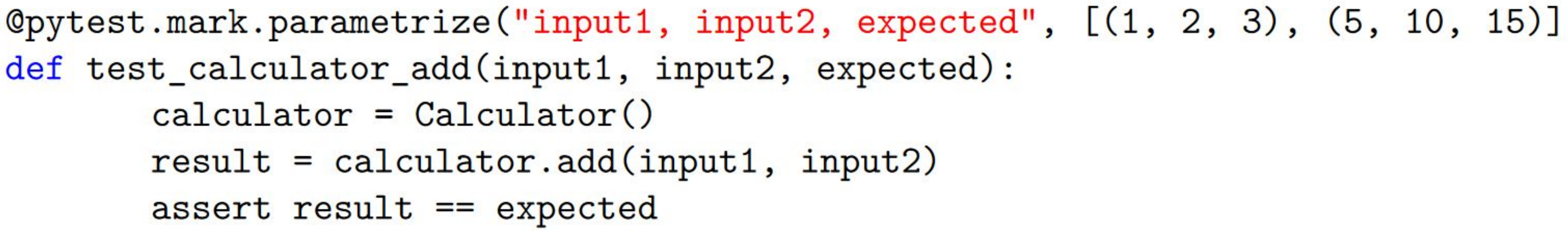
- Used to identify or run subset of test suite
- Tests can have multiple markers



```
#built-in xfail marker
@pytest.mark.xfail
def test_something()
        assert 1 + 2 == 3

#custom marker "my_marker"
@pytest.mark.my_marker
def test_something_else():
        assert 2 + 3 == 5
```

Background - `Pytest` parametrization

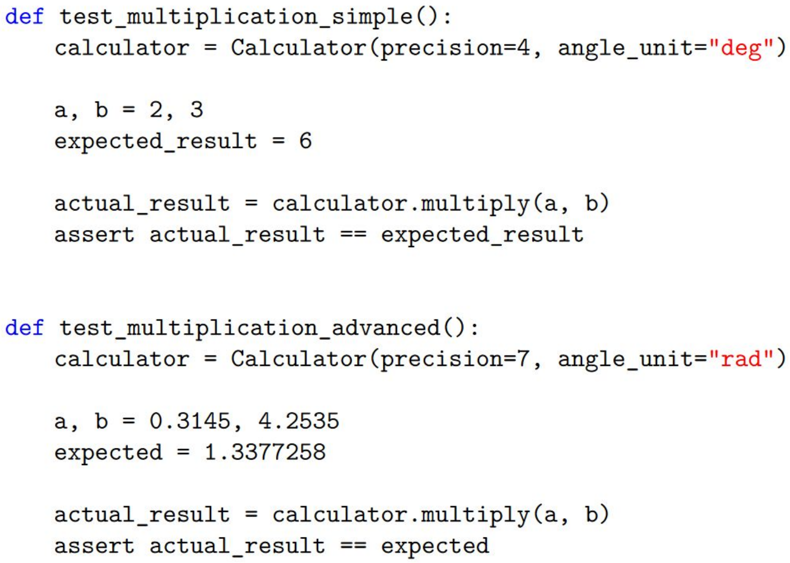
- Built-in `pytest` marker which takes arguments
- Supplied values are mapped to parameter names
- Each set of parentheses is parameters for single run of test



```
@pytest.mark.parametrize("input1, input2, expected", [(1, 2, 3), (5, 10, 15)]
def test_calculator_add(input1, input2, expected):
       calculator = Calculator()
       result = calculator.add(input1, input2)
       assert result == expected
```

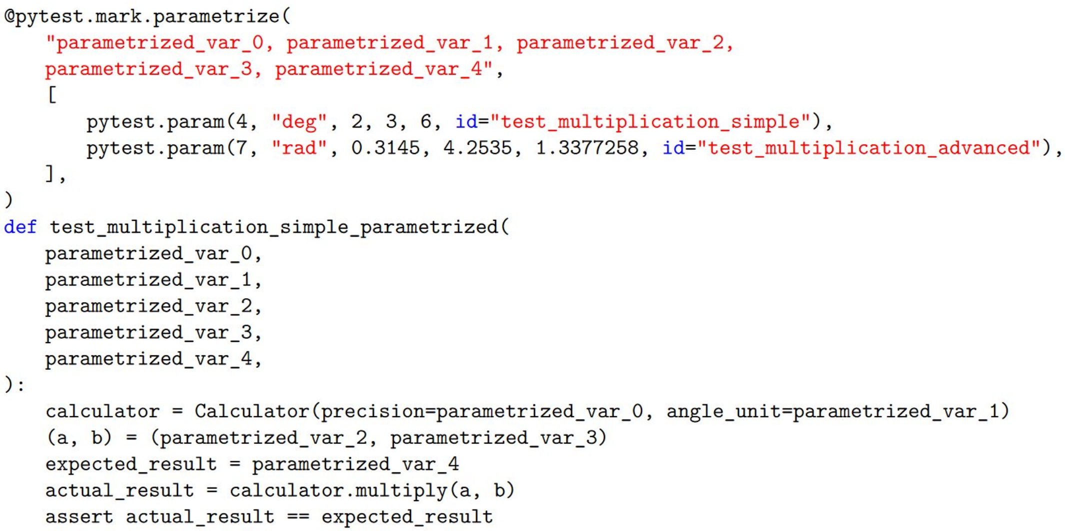
PyTeRor: an example

Pre-refactoring - clones



```
def test_multiplication_simple():
    calculator = Calculator(precision=4, angle_unit="deg")

    a, b = 2, 3
    expected_result = 6

    actual_result = calculator.multiply(a, b)
    assert actual_result == expected_result


def test_multiplication_advanced():
    calculator = Calculator(precision=7, angle_unit="rad")

    a, b = 0.3145, 4.2535
    expected = 1.3377258

    actual_result = calculator.multiply(a, b)
    assert actual_result == expected
```

Post-refactoring - target



```
@pytest.mark.parametrize(
    "parametrized_var_0, parametrized_var_1, parametrized_var_2,
    parametrized_var_3, parametrized_var_4",
    [
        pytest.param(4, "deg", 2, 3, 6, id="test_multiplication_simple"),
        pytest.param(7, "rad", 0.3145, 4.2535, 1.3377258, id="test_multiplication_advanced"),
    ],
)
def test_multiplication_simple_parametrized(
    parametrized_var_0,
    parametrized_var_1,
    parametrized_var_2,
    parametrized_var_3,
    parametrized_var_4,
):
    calculator = Calculator(precision=parametrized_var_0, angle_unit=parametrized_var_1)
    (a, b) = (parametrized_var_2, parametrized_var_3)
    expected_result = parametrized_var_4
    actual_result = calculator.multiply(a, b)
    assert actual_result == expected_result
```

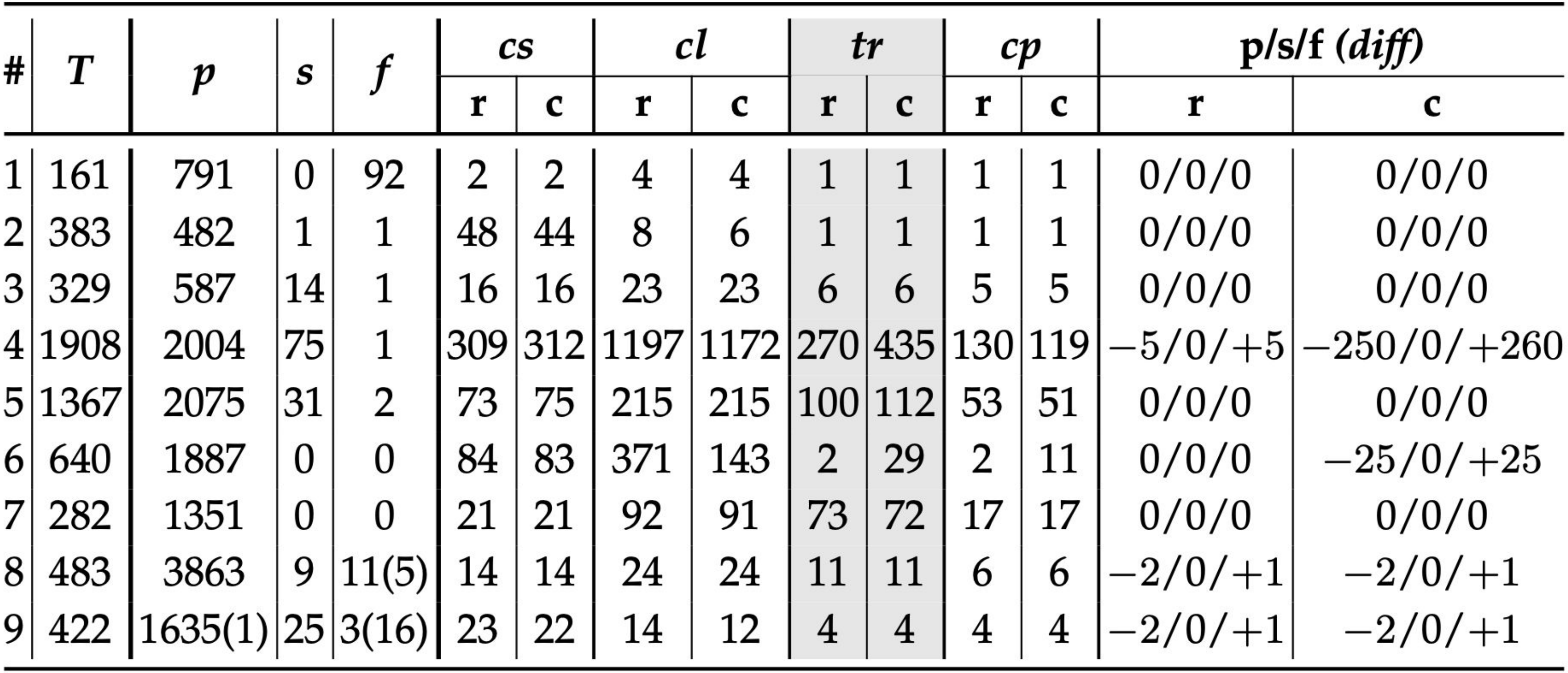
Regular parametrization vs cross-file parametrization

- Regular parametrization
    - Does not refactor clones between files
    - Instead, splits clone class based on scope

- Cross-file parametrization
    - Does not split clone class if spread over more than one file
    - Our limitation: currently no deep semantic analysis

Experimental results 1/2

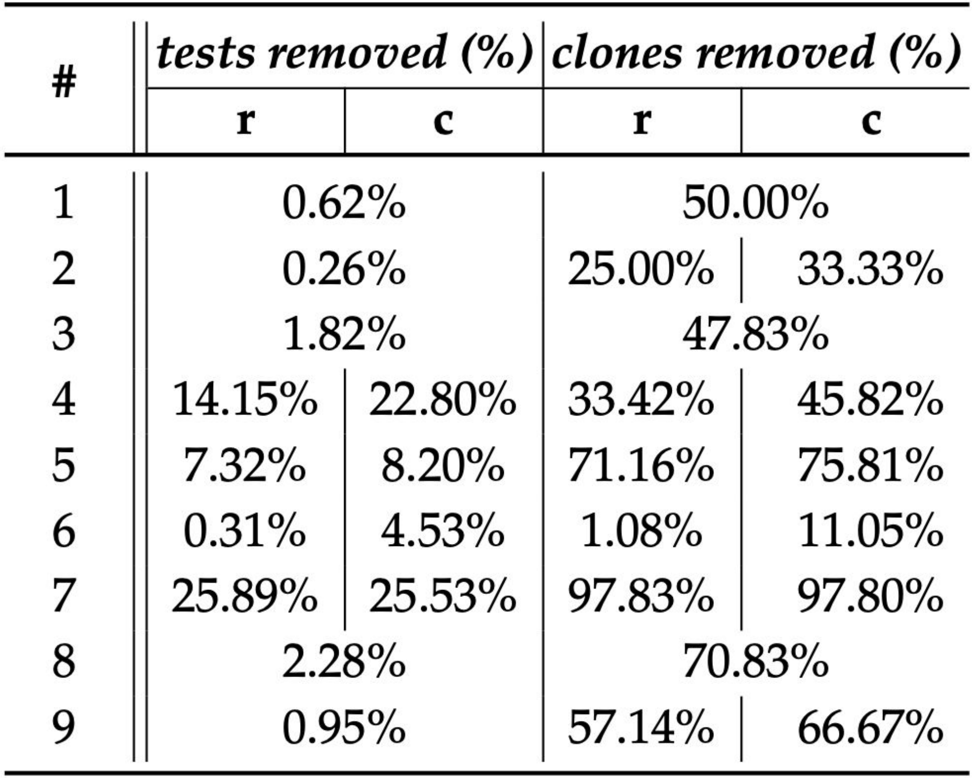
test definitions (T), number of (p)assing, (s)kipped or (f)ailed test runs
clone classes (cs), clones (cl) identified by PyTeRor, tests removed (tr),
parametrized clone classes (cp)



#	*T*	*p*	*s*	*f*	*cs*		*cl*		*tr*		*cp*		**p/s/f (*diff*)**	
					r	**c**	**r**	**c**	**r**	**c**	**r**	**c**	**r**	**c**
1	161	791	0	92	2	2	4	4	1	1	1	1	0/0/0	0/0/0
2	383	482	1	1	48	44	8	6	1	1	1	1	0/0/0	0/0/0
3	329	587	14	1	16	16	23	23	6	6	5	5	0/0/0	0/0/0
4	1908	2004	75	1	309	312	1197	1172	270	435	130	119	−5/0/+5	−250/0/+260
5	1367	2075	31	2	73	75	215	215	100	112	53	51	0/0/0	0/0/0
6	640	1887	0	0	84	83	371	143	2	29	2	11	0/0/0	−25/0/+25
7	282	1351	0	0	21	21	92	91	73	72	17	17	0/0/0	0/0/0
8	483	3863	9	11(5)	14	14	24	24	11	11	6	6	−2/0/+1	−2/0/+1
9	422	1635(1)	25	3(16)	23	22	14	12	4	4	4	4	−2/0/+1	−2/0/+1

Results of (r)egular/(c)rossfile parametrization

Experimental results 2/2

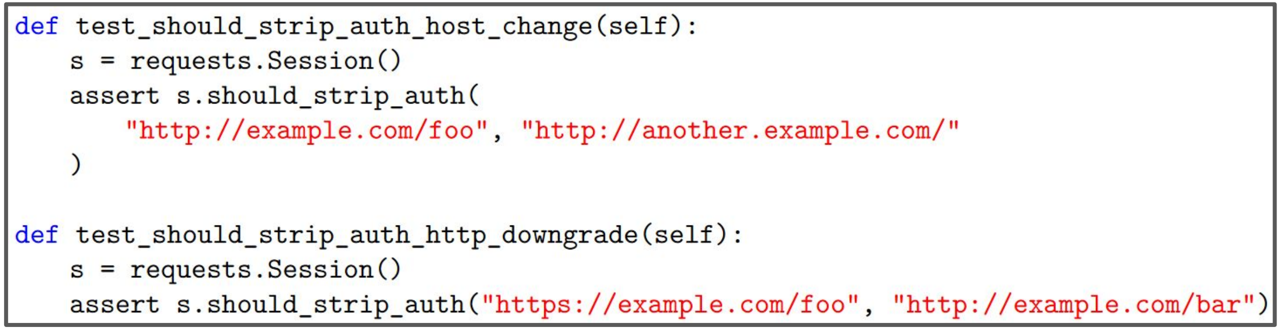


#	*tests removed (%)*		*clones removed (%)*	
	r	**c**	**r**	**c**
1	0.62%		50.00%	
2	0.26%		25.00%	33.33%
3	1.82%		47.83%	
4	14.15%	22.80%	33.42%	45.82%
5	7.32%	8.20%	71.16%	75.81%
6	0.31%	4.53%	1.08%	11.05%
7	25.89%	25.53%	97.83%	97.80%
8	2.28%		70.83%	
9	0.95%		57.14%	66.67%

Relative measure of the results of (r)egular/(c)rossfile parametrization

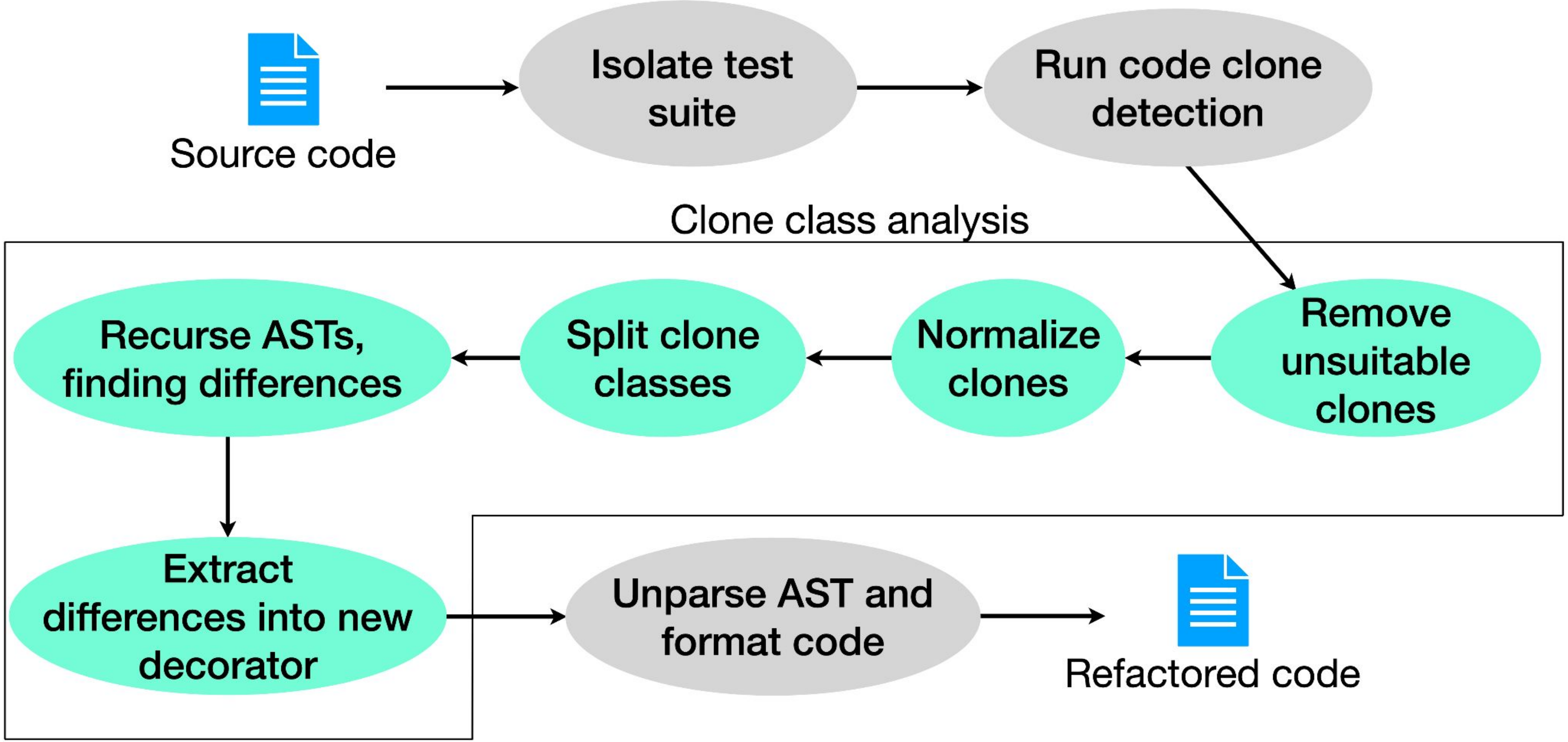
Threats to validity

- Dependent on results from clone detector
  - Certain clones are not found (different formatting)
- Unfamiliarity with projects we are testing
  - Characteristics which could bias results
- Only using open-source repositories
  - Results could be different for nine closed-source repositories
- Relatively low number of repositories
  - Cannot make any conclusions of significance



```
def test_should_strip_auth_host_change(self):
    s = requests.Session()
    assert s.should_strip_auth(
        "http://example.com/foo", "http://another.example.com/"
    )

def test_should_strip_auth_http_downgrade(self):
    s = requests.Session()
    assert s.should_strip_auth("https://example.com/foo", "http://example.com/bar")
```

PyTeRor


Source code
Isolate test suite
Run code clone detection
Clone class analysis
Remove unsuitable clones
Normalize clones
Split clone classes
Recurse ASTs, finding differences
Extract differences into new decorator
Unparse AST and format code
Refactored code

Identifying test files

- Isolate files following `pytest` file naming rules
- Run clone detection on isolated set of files
- Avoids clone pairs between test and non-test code

Code clone detection

- NiCad6
- Modified Type 2 configuration file
    - Literal abstraction
    - Blind clones
- Modified Python grammar file
    - Fix for bug in built-in grammar
    - Discrepancies between NiCad grammar and Python's **ast** module grammar

Code clone detection

- NiCad6
- Modified Type 2 configuration file
    - Literal abstraction
    - Blind clones
- Modified Python grammar file
    - Fix for bug in built-in grammar
    - Discrepancies between NiCad grammar and Python’s **ast** module grammar

Clone class analysis

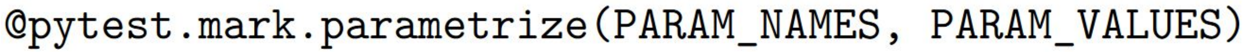
Analyse clone classes found in previous phase

Steps:

1. Processing clones
2. Normalization
3. Splitting clone classes
4. AST analysis
5. Extracting differences

Processing clones

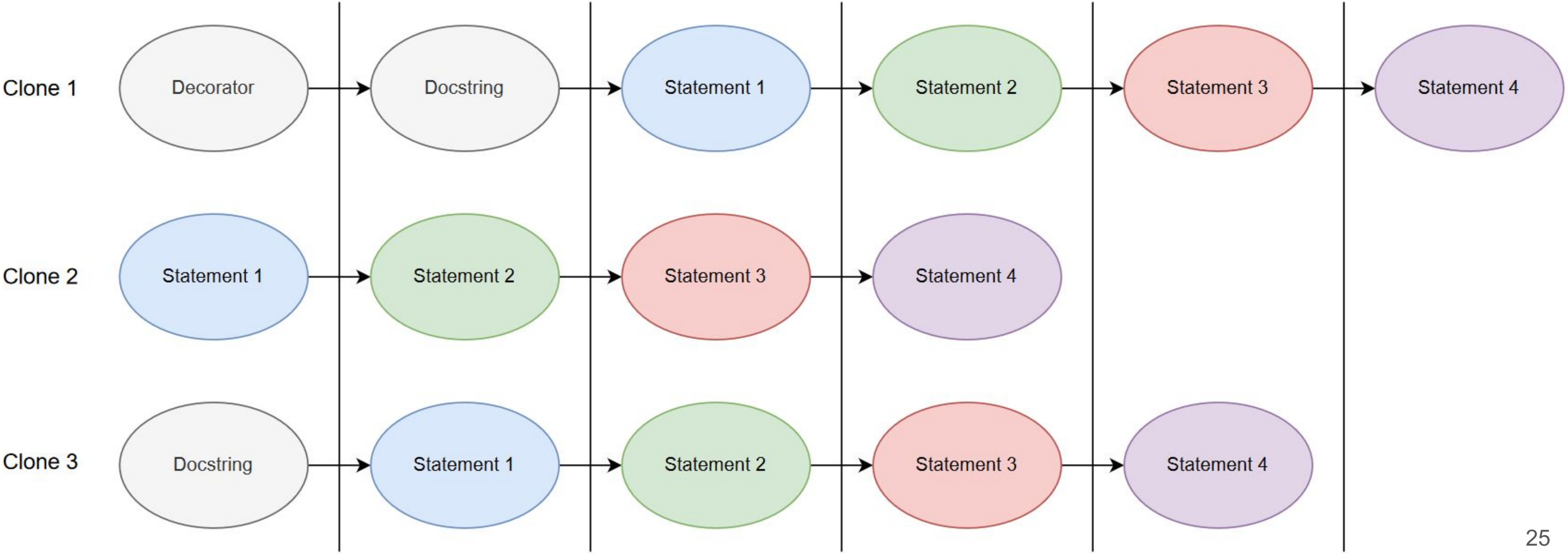
- Remove non-test clones
  - Fixtures
  - Other functions
- Remove clones with “bad” `parametrize` decorators
  - No direct access to parameter names, parameter values
  - Example:



```
@pytest.mark.parametrize(PARAM_NAMES, PARAM_VALUES)
```

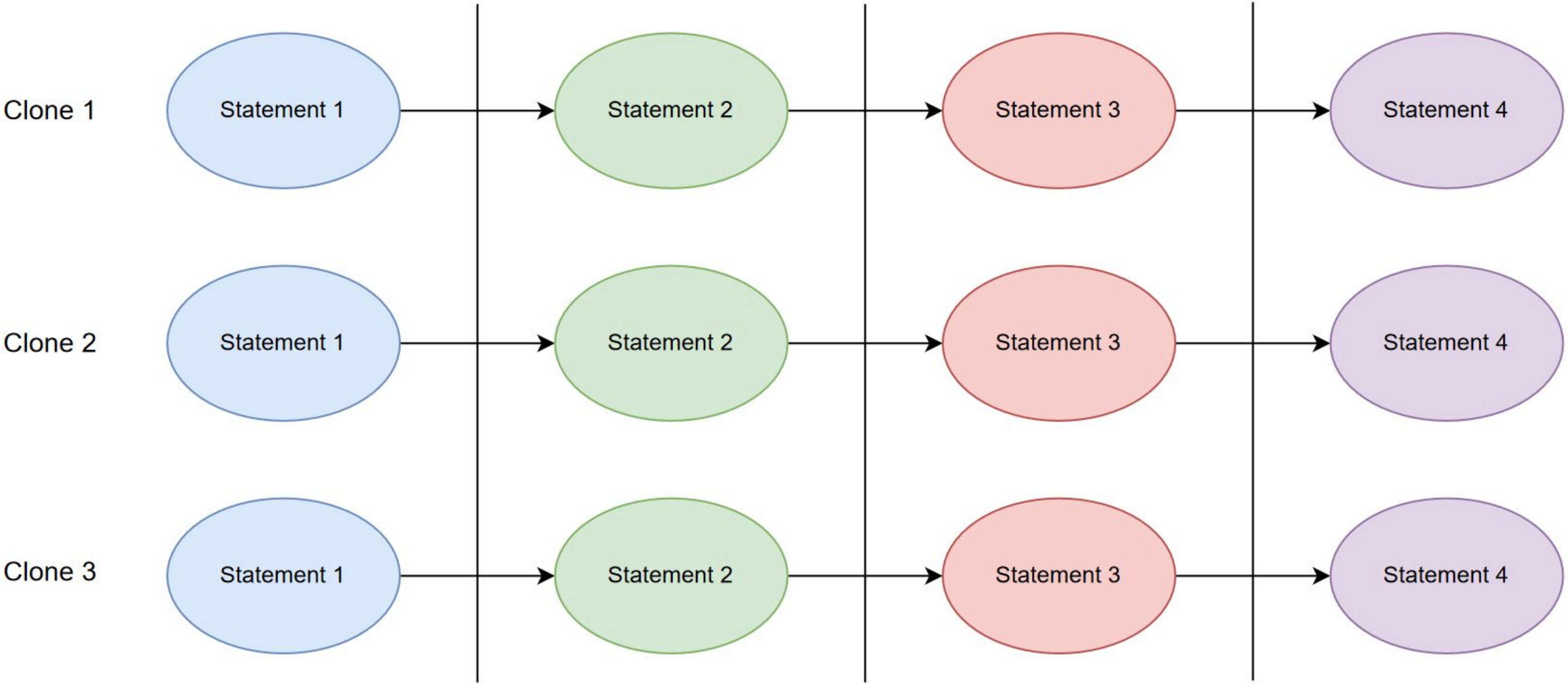
Normalization - Standardize AST of all clones

In order for simultaneous iteration



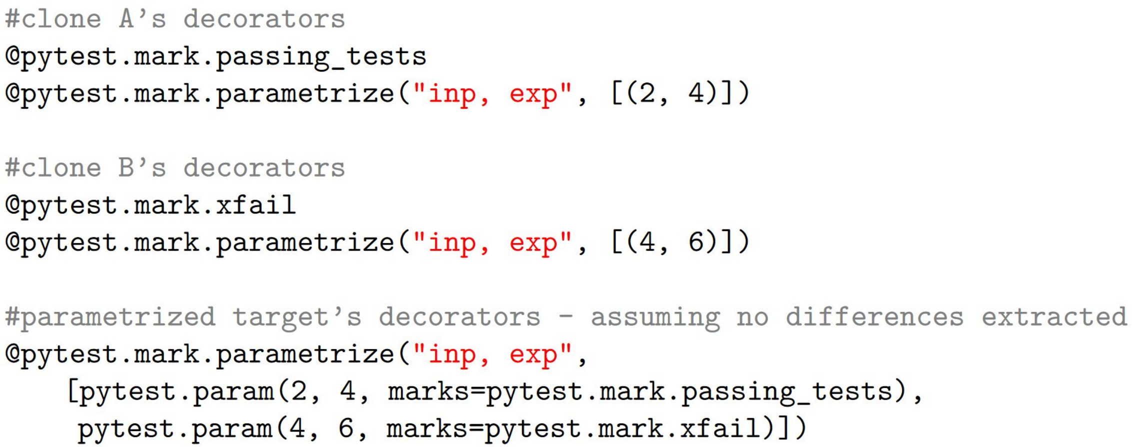
Normalization - Standardize AST of all clones

In order for simultaneous iteration



Splitting clone classes

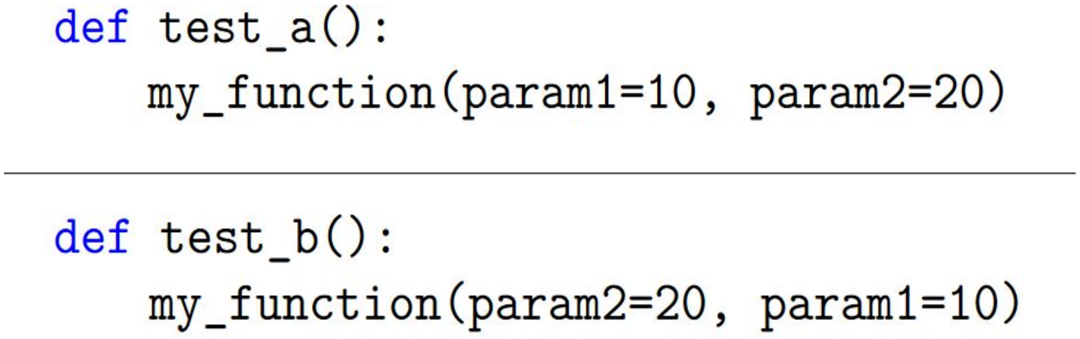
- Split on scope
  - Clones in different scopes cannot be parametrized
- Split on decorators
  - Clones with different decorators cannot be parametrized
  - Exception: certain built-in `pytest` decorators
    - Marker decorators, incl. parametrize decorator



```
#clone A’s decorators
@pytest.mark.passing_tests
@pytest.mark.parametrize("inp, exp", [(2, 4)])

#clone B’s decorators
@pytest.mark.xfail
@pytest.mark.parametrize("inp, exp", [(4, 6)])

#parametrized target’s decorators - assuming no differences extracted
@pytest.mark.parametrize("inp, exp",
    [pytest.param(2, 4, marks=pytest.mark.passing_tests),
     pytest.param(4, 6, marks=pytest.mark.xfail)])
```

AST analysis - simultaneous iteration over ASTs

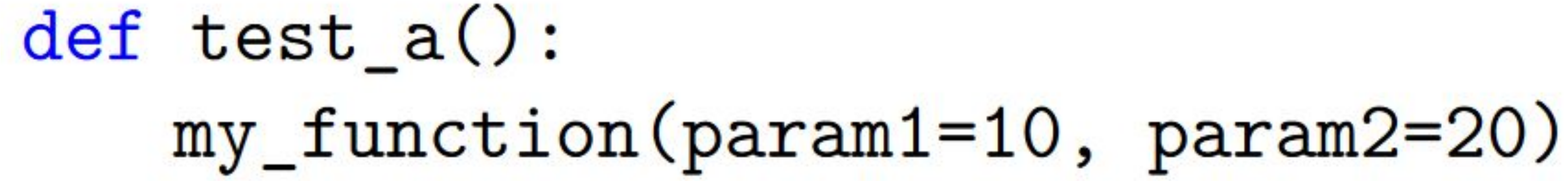
- Standardized ASTs
- Certain nodes may differ in values for Type 2 clones
  - ast.Constant
  - ast.Name
  - ast.Attribute
- Discovered differences are stored
- Some special cases
  - Inner functions with decorators
  - Import statements
  - Keywords in function call



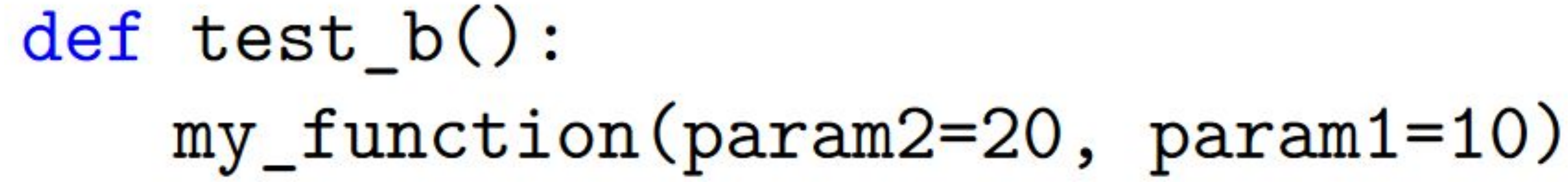
```python
def test_a():
    my_function(param1=10, param2=20)
```

```python
def test_b():
    my_function(param2=20, param1=10)
```

AST analysis - Keywords in function calls



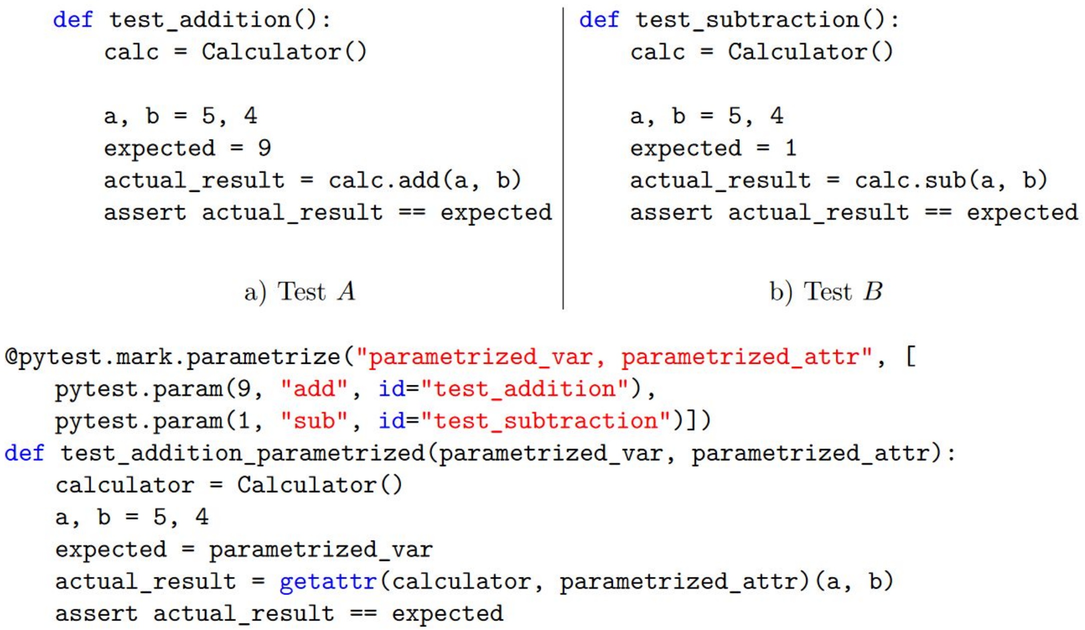
```python
def test_a():
    my_function(param1=10, param2=20)
```



```python
def test_b():
    my_function(param2=20, param1=10)
```

After AST analysis - split clone classes

- Split on attributes
  - We **do not** parametrize clones differing in attribute usage
  - Technically possible
- Split on fixtures
  - Cannot parametrize clones employing different fixtures
  - Not supported by `pytest`



```python
def test_addition():
    calc = Calculator()

    a, b = 5, 4
    expected = 9
    actual_result = calc.add(a, b)
    assert actual_result == expected
```

a) Test *A*

```python
def test_subtraction():
    calc = Calculator()

    a, b = 5, 4
    expected = 1
    actual_result = calc.sub(a, b)
    assert actual_result == expected
```

b) Test *B*

```python
@pytest.mark.parametrize("parametrized_var, parametrized_attr", [
    pytest.param(9, "add", id="test_addition"),
    pytest.param(1, "sub", id="test_subtraction")])
def test_addition_parametrized(parametrized_var, parametrized_attr):
    calculator = Calculator()
    a, b = 5, 4
    expected = parametrized_var
    actual_result = getattr(calculator, parametrized_attr)(a, b)
    assert actual_result == expected
```

c) Attribute parametrization of tests *A* & *B* using `getattr` method

Extracting differences

- Stored differences added to new `parametrize` decorator
- Generating variable names:
  - `parametrized_constant_N`
  - `parametrized_name_M`
  - N and M are incrementing numbers
  - Nodes appearing multiple times receive same name
- Variable names added to target clone/function and parameters
- Combining new parametrization with pre-existing parametrization

Unparsing

- Unparse target clone from AST, format, insert in file
- Remove other clones from file
- Preserves formatting and comments in file
  - Except in target clone

Discussion 1/2

- Exceptions to successful refactoring
  - Tests invoking (removed) tests
  - Parametrizing tests from other frameworks (unittest)
- Most test suites were not reduced by large amounts
  - Six repos: ~2% or fewer tests removed
  - Many of these repos had few clones
  - Repos with many clones had higher %
- Evaluating PyTeRor is difficult, no benchmark
  - How many clones are we failing to parametrize?

Discussion 2/2

- "Refactoring" - Certain behaviour is affected
  - `Pytest`'s *-k* option, with pre-set IDs (overwritten by PyTeRor)
  - `Pytest` does not support multiple IDs
  - Other behaviour is consistent pre- and post-refactoring
- Code quality - code becomes less legible
  - Generated variable names - "parametrized_constant_0"
  - Parametrizing function names
  - Comments removed
  - However: Reducing no. of clones is often tied to increased code quality
    - Maintainability

Alternative ideas for implementation

- Refactoring suggestions
  - Instead of actual refactorings
  - Manual refactoring -> higher code quality
  - Plug-in/extension to IDEs
- Extracting common initialisation code
  - Creating fixtures
  - Many non-clone tests contain common initialisations/set-up

Related work

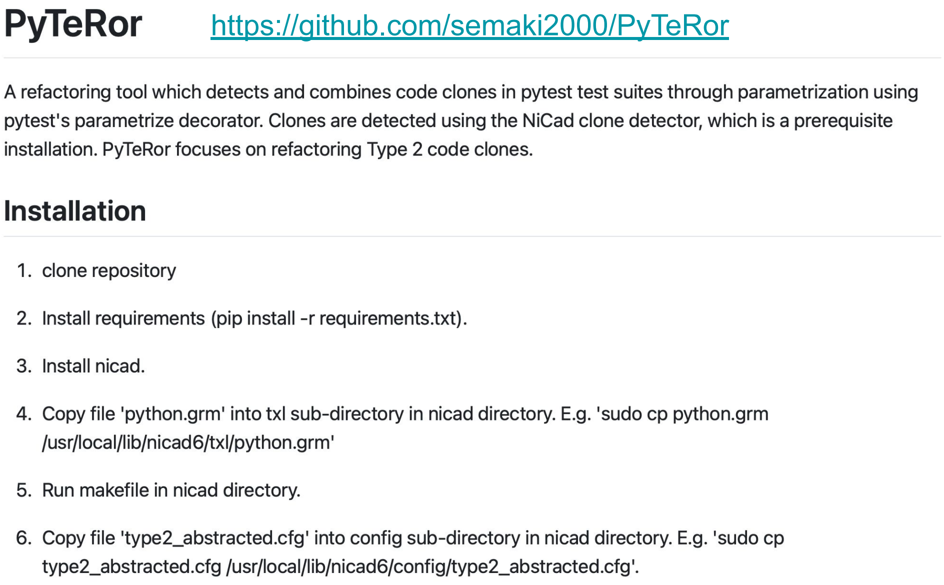
- Automated refactorings of Python code
    - Zhang et al., specifically targeted at transforming non-idiomatic code into idiomatic
- Test code refactoring
    - Meszaros, xUnit Test Patterns: Refactoring Test Code
    - Deursen et al, eleven test smells + six refactorings for these
    - Xuan et al, automatic test code refactoring, though intended to improve dynamic analysis
- Refactoring code clones
    - Baars and Oprescu, identifying refactorable clones
    - Tsantalis et al., clones in production code vs in test code
    - Baqais and Alshayeb cover multiple tools and techniques for automated detection and elimination

Future work

- Parametrizing test clones in other languages
    - Java, C++, C#
- Larger-scale experiment
    - Potentially automated
    - Could provide more interesting results for analysis
    - Measure coverage as well?
- Continuing work on PyTeRor
    - `Pytest` configuration files
    - Extracting common initialisation code?

Conclusion

- PyTeRor: reducing `pytest` test suites though refactoring Type 2 code clones
- PyTeRor does not parametrize certain code clones, e.g:
  - Clones with attribute differences
  - Clones with scope differences
- Successful refactoring except specific cases



PyTeRor

https://github.com/semaki2000/PyTeRor

A refactoring tool which detects and combines code clones in pytest test suites through parametrization using pytest's parametrize decorator. Clones are detected using the NiCad clone detector, which is a prerequisite installation. PyTeRor focuses on refactoring Type 2 code clones.

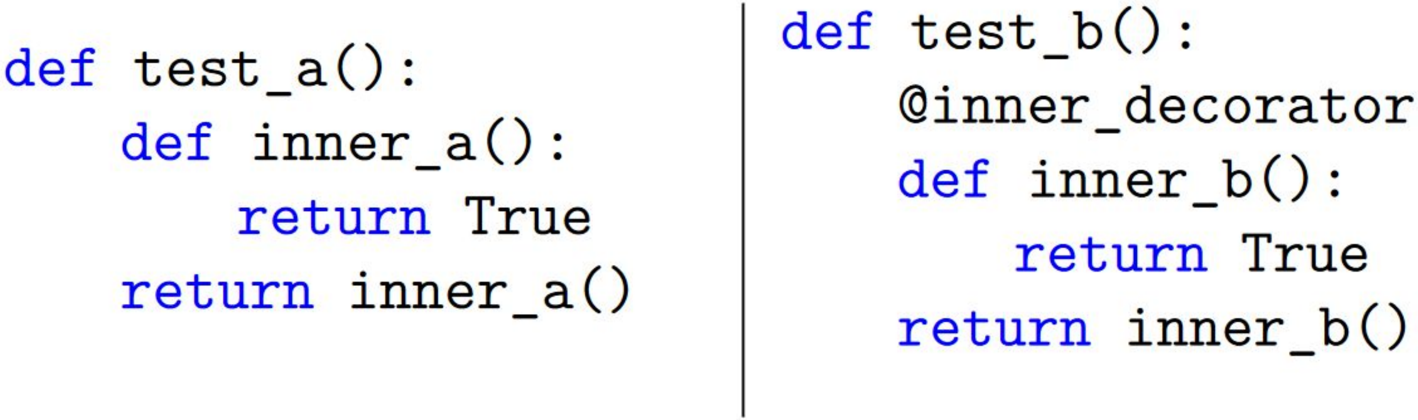
Installation

1. clone repository
2. Install requirements (pip install -r requirements.txt).
3. Install nicad.
4. Copy file 'python.grm' into txl sub-directory in nicad directory. E.g. 'sudo cp python.grm /usr/local/lib/nicad6/txl/python.grm'
5. Run makefile in nicad directory.
6. Copy file 'type2_abstracted.cfg' into config sub-directory in nicad directory. E.g. 'sudo cp type2_abstracted.cfg /usr/local/lib/nicad6/config/type2_abstracted.cfg'.

https://zenodo.org/records/11145543

DOI 10.5281/zenodo.11145543

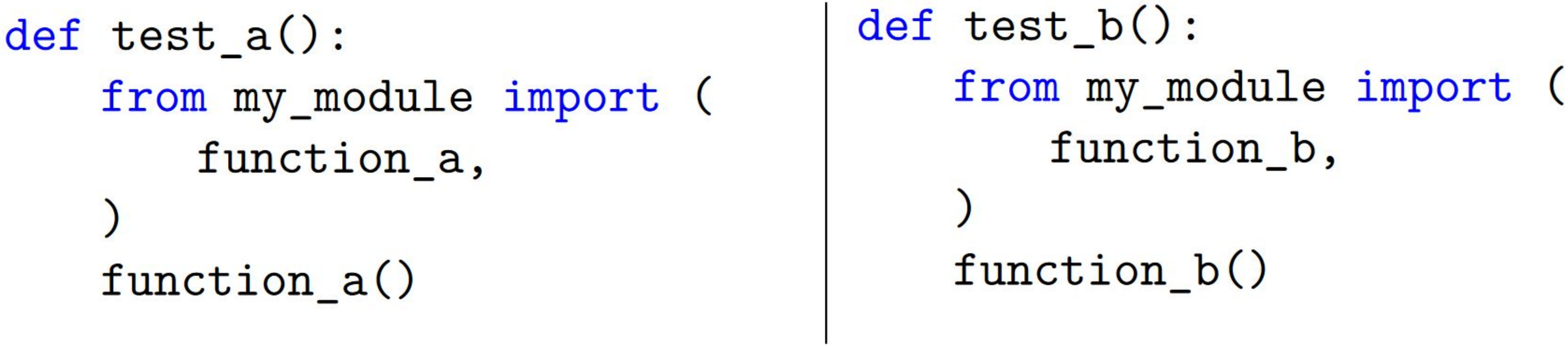
AST analysis - Inner functions with decorators



```python
def test_a():
    def inner_a():
        return True
    return inner_a()
```

```python
def test_b():
    @inner_decorator
    def inner_b():
        return True
    return inner_b()
```

AST analysis - Import statements



```
def test_a():
    from my_module import (
        function_a,
    )
    function_a()
```

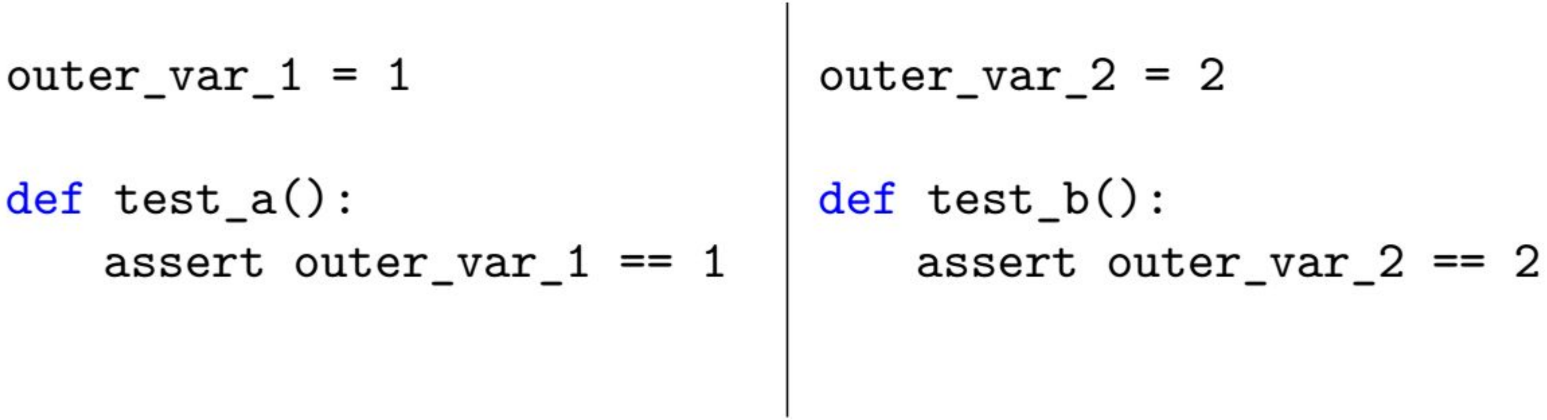
```
def test_b():
    from my_module import (
        function_b,
    )
    function_b()
```

Analysing differing identifiers

- Non-locally defined variables
- Locally defined variables
- Mix between local and non-local

Analysing differing identifiers - non-locally defined

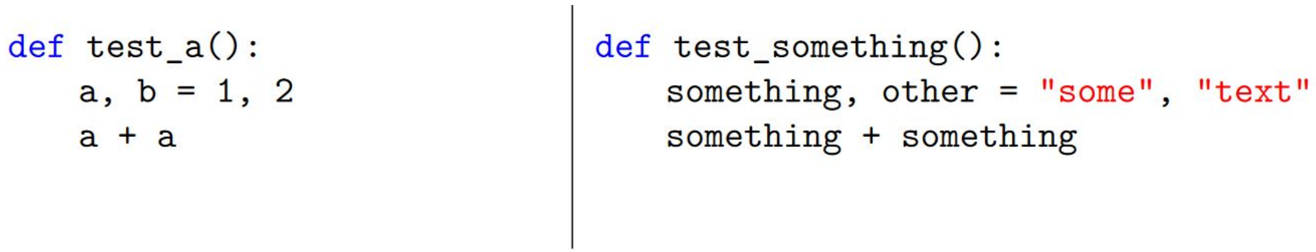
- Variable names extracted into new parametrize decorator
- Replaced with generated variable name in refactored code



```python
outer_var_1 = 1

def test_a():
    assert outer_var_1 == 1
```

```python
outer_var_2 = 2

def test_b():
    assert outer_var_2 == 2
```

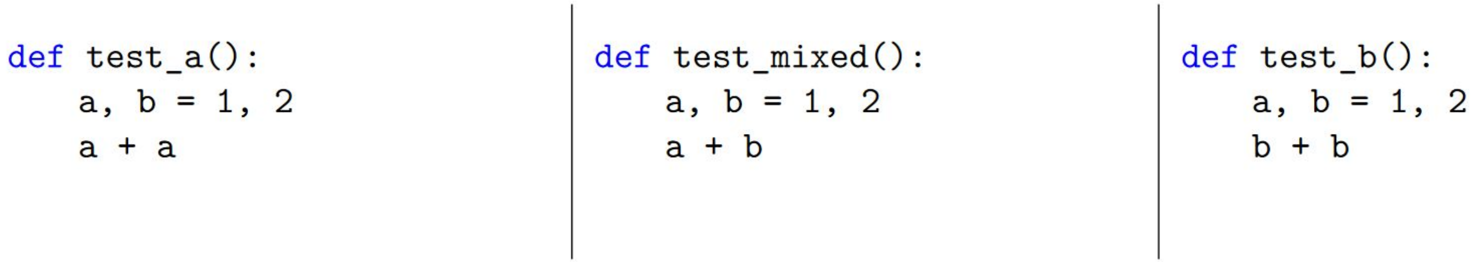
Analysing differing identifiers - locally defined



```python
def test_a():
    a, b = 1, 2
    a + a
```

```python
def test_something():
    something, other = "some", "text"
    something + something
```

Figure 3.14: Consistent local variables



```python
def test_a():
    a, b = 1, 2
    a + a
```

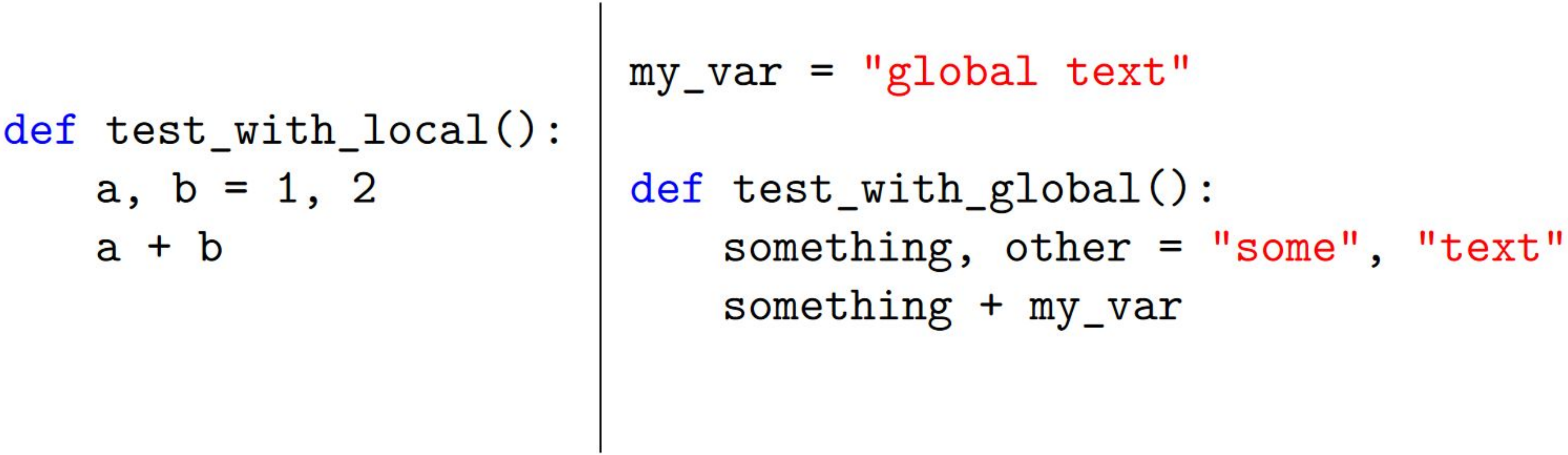
```python
def test_mixed():
    a, b = 1, 2
    a + b
```

```python
def test_b():
    a, b = 1, 2
    b + b
```

Figure 3.15: Inconsistent local variables

Analysing differing identifiers - mixed local/non-local

- Unparametrizable - cannot extract the local variables

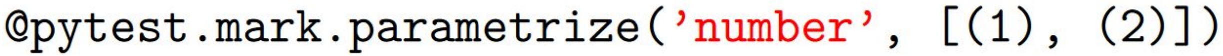


```python
def test_with_local():
    a, b = 1, 2
    a + b
```

```python
my_var = "global text"

def test_with_global():
    something, other = "some", "text"
    something + my_var
```

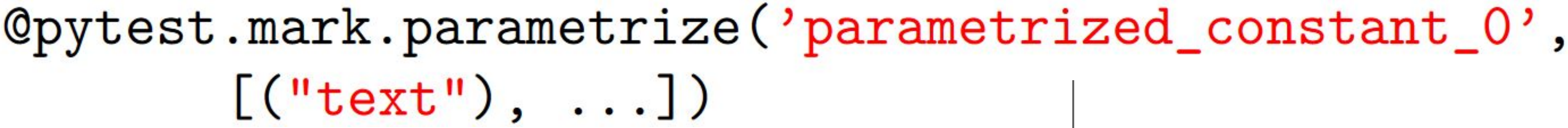
Combining pre-existing and new parametrizations

Pre-existing parametrize decorator



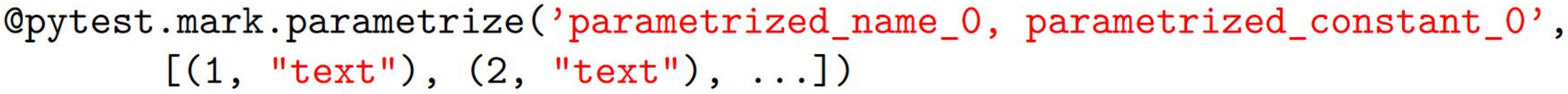
```
@pytest.mark.parametrize(’number’, [(1), (2)])
```

Extracting differences within clones



```
@pytest.mark.parametrize(’parametrized_constant_0’,
        [("text"), ...])
```

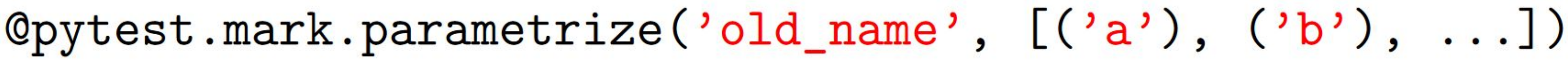
Adding pre-parametrization to new parametrization



```
@pytest.mark.parametrize(’parametrized_name_0, parametrized_constant_0’,
        [(1, "text"), (2, "text"), ...])
```

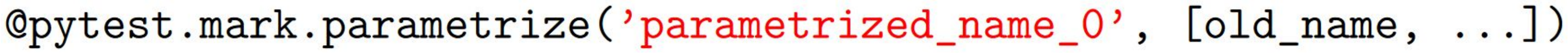
Extracting parametrized names

Pre-existing parametrize decorator



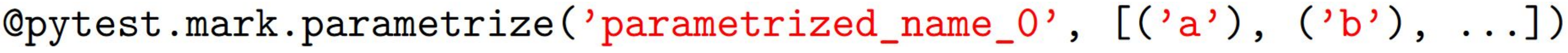
```
@pytest.mark.parametrize(’old_name’, [(’a’), (’b’), ...])
```

Extracting differences within clones



```
@pytest.mark.parametrize(’parametrized_name_0’, [old_name, ...])
```

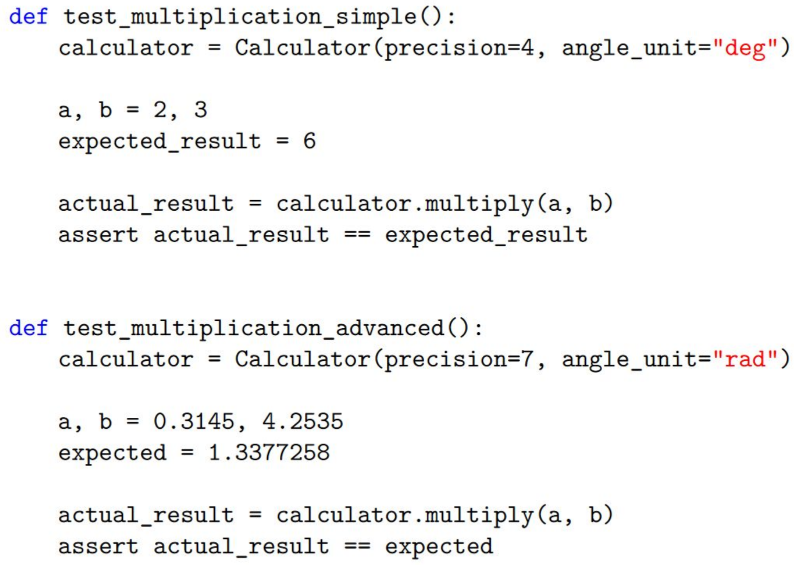
Replacing pre-parametrized names with values



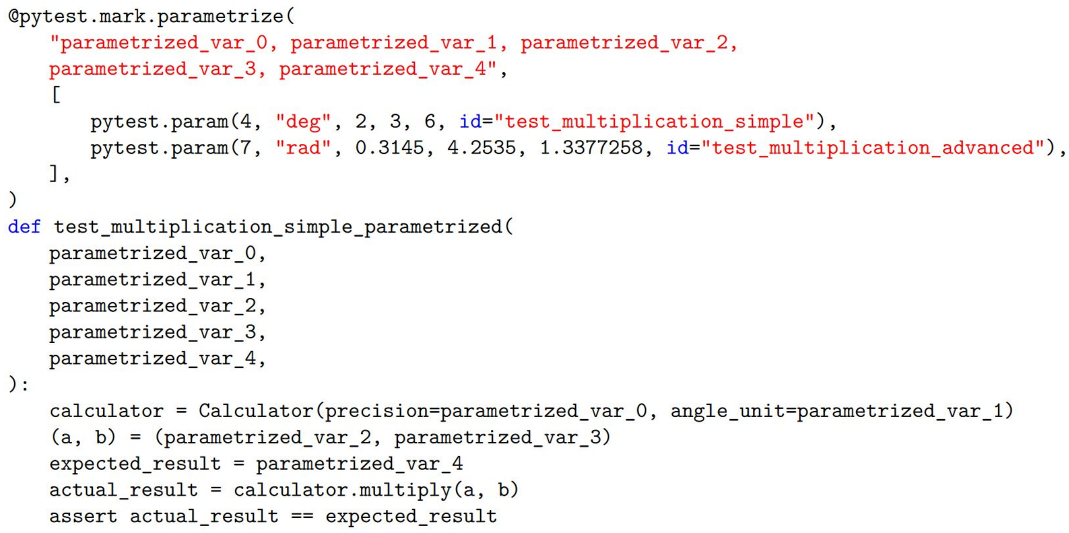
```
@pytest.mark.parametrize(’parametrized_name_0’, [(’a’), (’b’), ...])
```

Using pytest.param in parametrize decorator - ids

- Function names of tests are preserved in refactored code through *id* keyword
- Preserves behaviour for pytest’s *-k* option



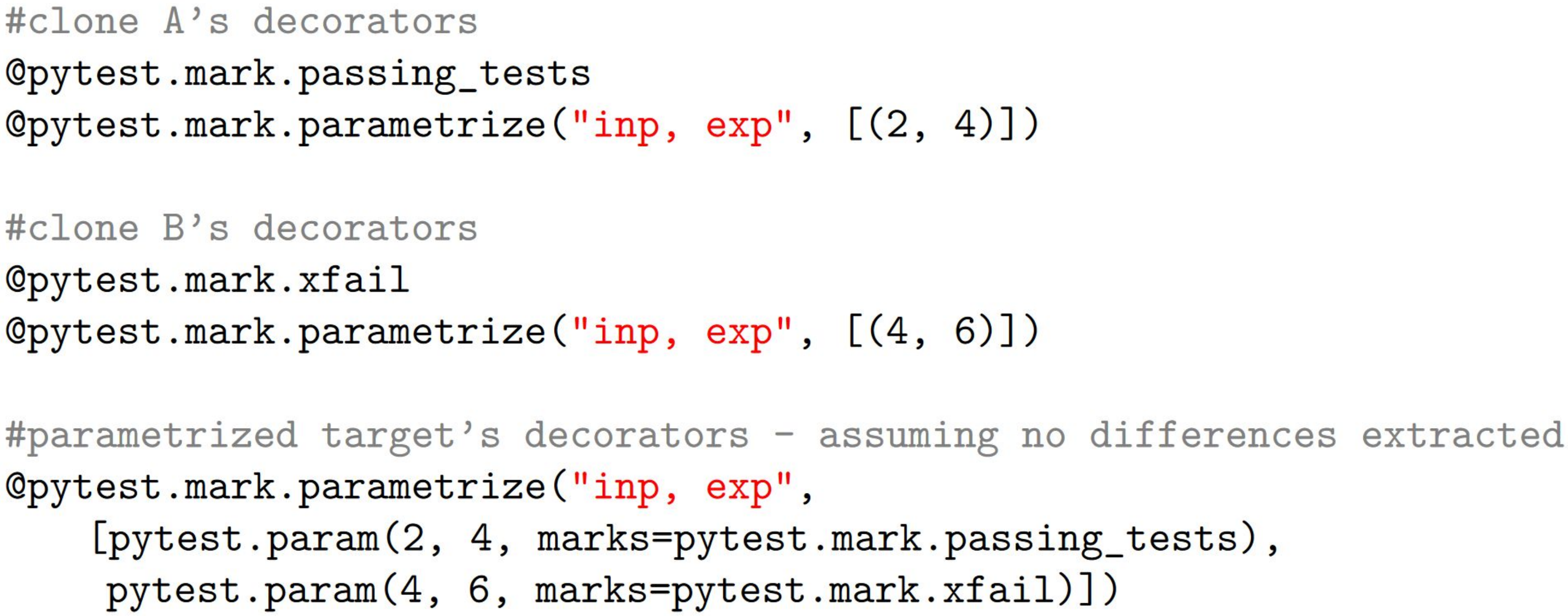
```python
def test_multiplication_simple():
    calculator = Calculator(precision=4, angle_unit="deg")

    a, b = 2, 3
    expected_result = 6

    actual_result = calculator.multiply(a, b)
    assert actual_result == expected_result


def test_multiplication_advanced():
    calculator = Calculator(precision=7, angle_unit="rad")

    a, b = 0.3145, 4.2535
    expected = 1.3377258

    actual_result = calculator.multiply(a, b)
    assert actual_result == expected
```



```python
@pytest.mark.parametrize(
    "parametrized_var_0, parametrized_var_1, parametrized_var_2,
    parametrized_var_3, parametrized_var_4",
    [
        pytest.param(4, "deg", 2, 3, 6, id="test_multiplication_simple"),
        pytest.param(7, "rad", 0.3145, 4.2535, 1.3377258, id="test_multiplication_advanced"),
    ],
)
def test_multiplication_simple_parametrized(
    parametrized_var_0,
    parametrized_var_1,
    parametrized_var_2,
    parametrized_var_3,
    parametrized_var_4,
):
    calculator = Calculator(precision=parametrized_var_0, angle_unit=parametrized_var_1)
    (a, b) = (parametrized_var_2, parametrized_var_3)
    expected_result = parametrized_var_4
    actual_result = calculator.multiply(a, b)
    assert actual_result == expected_result
```

Using pytest.param in parametrize decorator - markers

- Markers are preserved through the *marks* keyword
- Preserves behaviour for pytest's *-m* option



```
#clone A’s decorators
@pytest.mark.passing_tests
@pytest.mark.parametrize("inp, exp", [(2, 4)])

#clone B’s decorators
@pytest.mark.xfail
@pytest.mark.parametrize("inp, exp", [(4, 6)])

#parametrized target’s decorators - assuming no differences extracted
@pytest.mark.parametrize("inp, exp",
    [pytest.param(2, 4, marks=pytest.mark.passing_tests),
     pytest.param(4, 6, marks=pytest.mark.xfail)])
```