AexPy: Detecting API Breaking Changes in Python Packages

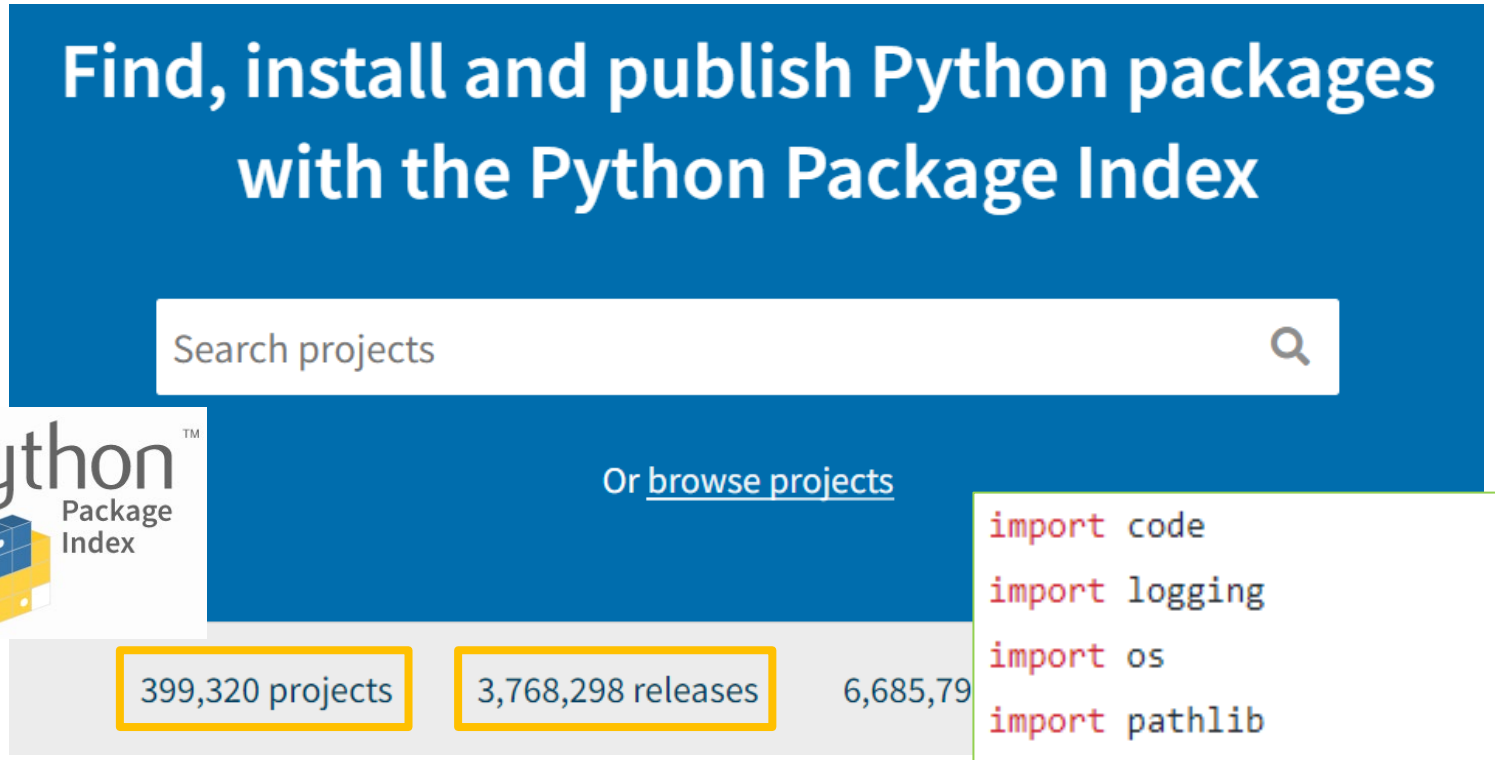
Xingliang Du, Jun Ma

State Key Laboratory for Novel Software Technology

Nanjing University

Motivation

The Popular Python



Create third-party packages

```
import code
import logging
import os
import pathlib

import click
import yaml
from click import BadArgumentUsage, BadOptionUsage, BadParameter
from click.exceptions import ClickException
```

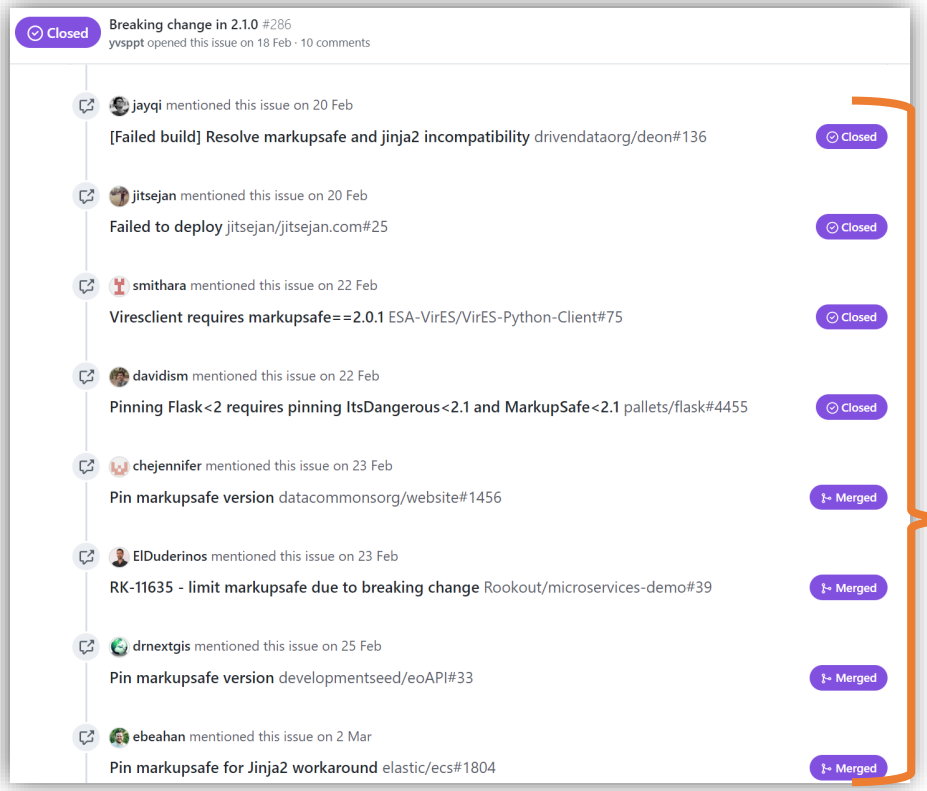
Use third-party packages

Motivation

Difficulty on Maintaining Packages

v1.0

{ API }



v1.1

{ API }

Affected clients

Breaking change in 2.1.0 · Issue #286 · pallets/markupsafe (github.com)

Motivation

Difficulty on Maintaining Packages

v1.0

{ API }

v1.1

{ API }

Breaking change in 0.5.5 · Issue #76 · sarugaku/resolvelib (github.com)

Motivation

Difficulty on Maintaining Packages

Backward Compatible / **Breaking**?

v1.0

{ API }

v1.1

{ API }

Breaking change in 0.5.5 #76

Closed lkollar opened this issue on 9 Mar 2021 · 2 comments

lkollar commented on 9 Mar 2021

We have a tool using `resolvelib` which implements the `AbstractProvider` interface. 0.5.5 broke the API with `915363f` changing the signature of `identify`.

Following semver, technically such changes are to be expected, but I also saw #74 warning about upcoming API breakage in 0.6.0, so I'm wondering what the policy around API changes is in `resolvelib`.

uranusjr commented on 9 Mar 2021 Member

I accidentally released the wrong branch (master) with the kwargs change 😖 I realised it just now as well (I'm pulling in changes in pypa/pip#9699).

My rule around versioning for now is to do breaking changes in the second digit (minor version), at least until 1.0 is reached. I think I'll just yank 0.5.5. Thanks for the report!

2

Breaking change in 0.5.5 · Issue #76 · sarugaku/resolvelib (github.com)

Motivation

Difficulty on Maintaining Packages

Manually check?

Backward Compatible / **Breaking**?

v1.0

{ API }

v1.1

{ API }

Breaking change in 0.5.5 #76

Closed lkollar opened this issue on 9 Mar 2021 · 2 comments

lkollar commented on 9 Mar 2021

We have a tool using `resolvelib` which implements the `AbstractProvider` interface. 0.5.5 broke the API with `915363f` changing the signature of `identify`.

Following semver, technically such changes are to be expected, but I also saw #74 warning about upcoming API breakage in 0.6.0, so I'm wondering what the policy around API changes is in `resolvelib`.

uranusjr commented on 9 Mar 2021 Member

I accidentally released the wrong branch (master) with the kwargs change 😖 I realised it just now as well (I'm pulling in changes in pypa/pip#9699).

My rule around versioning for now is to do breaking changes in the second digit (minor version), at least until 1.0 is reached. I think I'll just yank 0.5.5. Thanks for the report!

2

Breaking change in 0.5.5 · Issue #76 · sarugaku/resolvelib (github.com)

Motivation

Difficulty on Maintaining Packages

Automatically check by AexPy!

Backward Compatible / **Breaking**?

v1.0

{ API }

v1.1

{ API }

Breaking change in 0.5.5 #76

Closed lkollar opened this issue on 9 Mar 2021 · 2 comments

lkollar commented on 9 Mar 2021

We have a tool using `resolvelib` which implements the `AbstractProvider` interface. 0.5.5 broke the API with `915363f` changing the signature of `identify`.

Following semver, technically such changes are to be expected, but I also saw #74 warning about upcoming API breakage in 0.6.0, so I'm wondering what the policy around API changes is in `resolvelib`.

uranusjr commented on 9 Mar 2021 Member

I accidentally released the wrong branch (master) with the kwargs change 😖 I realised it just now as well (I'm pulling in changes in pypa/pip#9699).

My rule around versioning for now is to do breaking changes in the second digit (minor version), at least until 1.0 is reached. I think I'll just yank 0.5.5. Thanks for the report!

2

Breaking change in 0.5.5 · Issue #76 · sarugaku/resolvelib (github.com)

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
            self.x += 1
            return self.x
        else:
            raise StopIteration

def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```python
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
            self.x += 1
            return self.x
        else:
            raise StopIteration

def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```python
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
            self.x += 1
            return self.x
        else:
            raise StopIteration

def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Attributes in constructor

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```python
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
            self.x += 1
            return self.x
        else:
            raise StopIteration

def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Decorators to modify APIs at runtime

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```python
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
            self.x += 1
            return self.x
        else:
            raise StopIteration
```

Duck-typing: class A is a virtual subclass of Iterable

```python
def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```python
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
            self.x += 1
            return self.x
        else:
            raise StopIteration

def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Optional type annotations

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

Same API but different aliases

```
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
            self.x += 1
            return self.x
        else:
            raise StopIteration

def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

data._store
write

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10
```

Developers' convention, but still accessible for clients

```
    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
            self.x += 1
            return self.x
        else:
            raise StopIteration

def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

Different visibilities of aliases

```
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
            self.x += 1
            return self.x
        else:
            raise StopIteration

def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

data._store (private)
write (public)

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```python
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
            self.x += 1
            return self.x
        else:
            raise StopIteration

def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
    test(1, 2)   Positional


def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
    test(1, 2)   Positional
    test(1, posOrKw=2, kw=3)   Keyword


def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

```
from os import path
from data import _store as write

class A:
    def __init__(self):
        self._x = 0
        self.bound = 10

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    def __iter__(self) -> "Optional[A]": return self

    def __next__(self):
        if self.x < self.bound:
    test(1, 2)   Positional
    test(1, posOrKw=2, kw=3)   Keyword
    test(1, 2, 3, other=4)   Variadic

def test(pos, /, posOrKw, *args, kw = None, **kwargs):
    print(pos, posOrKw, kw, args, kwargs)
```

Motivation

Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

Motivation
Challenges

Dynamic Language Features

- Programming behaviors at runtime
- Dynamic type system with duck-typing

Complex API References

- Importing and renaming
- Same API but different names

Fake Private Members

- Fuzzy API accessibility / visibility
- Custom naming-conventions in case of aliasing

Flexible Argument Passing

- Required or optional parameters
- Positional, keyword, or variadic parameters

API Description

Extraction Method

Comparing Algorithm

Change Severities

AexPy's Approach

Overview

API Description

Extraction Method

Comparing Algorithm

Change Severities

AexPy's Approach

Overview

Detailed API Model

Extraction Method

Comparing Algorithm

Change Severities

AexPy's Approach

Overview

Detailed API Model

Hybrid Analysis

Comparing Algorithm

Change Severities

AexPy's Approach

Overview

Detailed API Model

Hybrid Analysis

Constraint Checking

Change Severities

AexPy's Approach

Overview

Detailed API Model

Hybrid Analysis

Constraint Checking

Breaking Levels

AexPy's Approach

Overview

Extraction

Detection

Grading

- API model
- Dynamic reflection
- Static analysis

- Change classification
- API and parameter pairing
- Constraint-based comparing

- Breaking levels
- API scope and change content
- Type compatibility

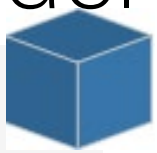
Detailed API Model

Hybrid Analysis

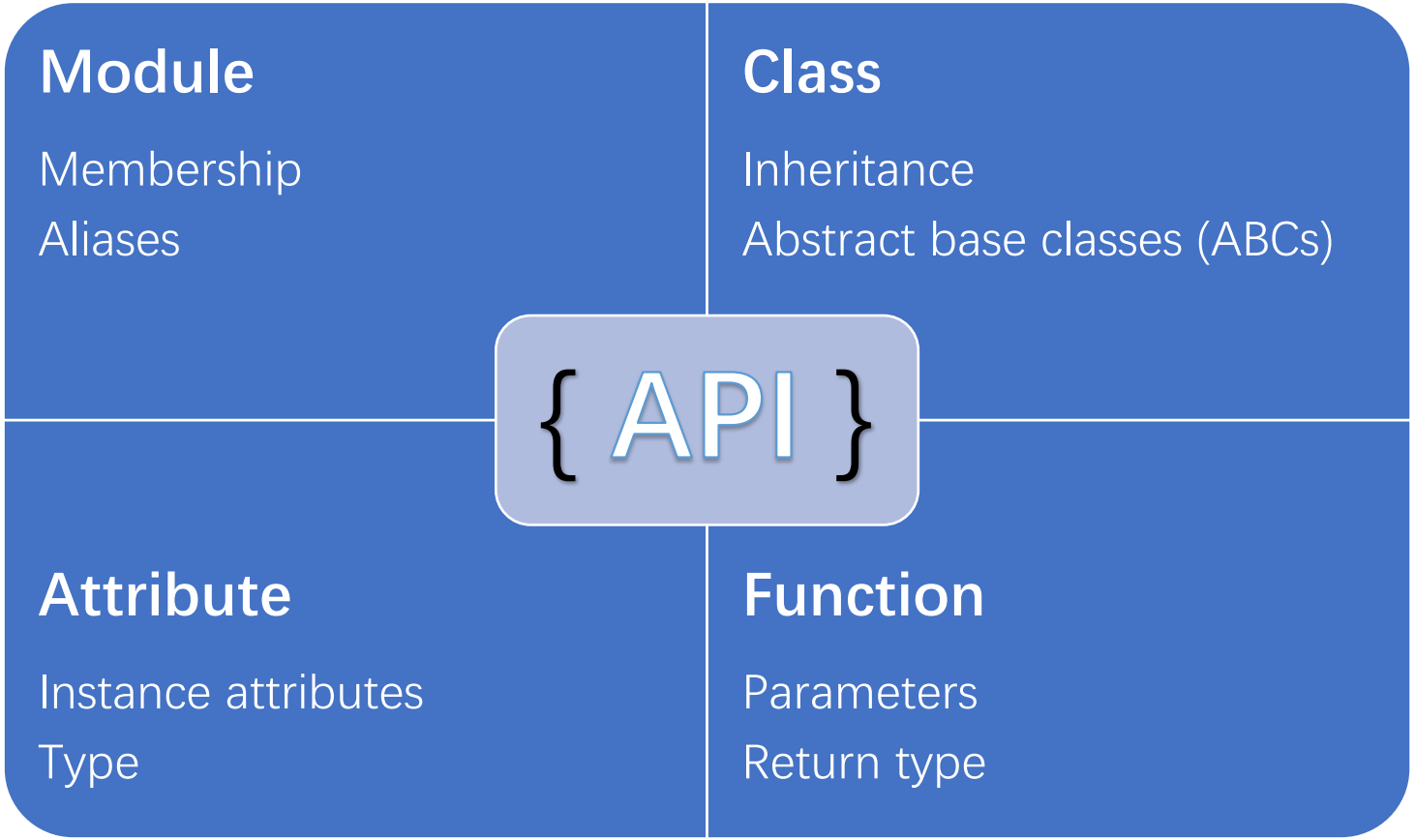
Constraint Checking

Breaking Levels

AexPy's Approach
Extraction – API Model

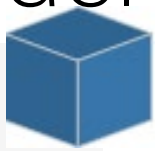


```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```



AexPy's Approach

Extraction – API Model



members(M).values

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Module

Membership

Aliases

Class

Inheritance

Abstract base classes (ABCs)

{ API }

Attribute

Instance attributes

Type

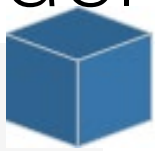
Function

Parameters

Return type

AexPy's Approach

Extraction – API Model



```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" =
        return c

    @property
    def x(self): return self
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

members(M).values

members(M).keys

```
{
    "opt",
    "func",
    "_share",
    "A",
    "B",
    "C"
}
```

Module

Membership

Aliases

Class

Inheritance

Abstract base classes (ABCs)

{ API }

Attribute

Instance attributes

Type

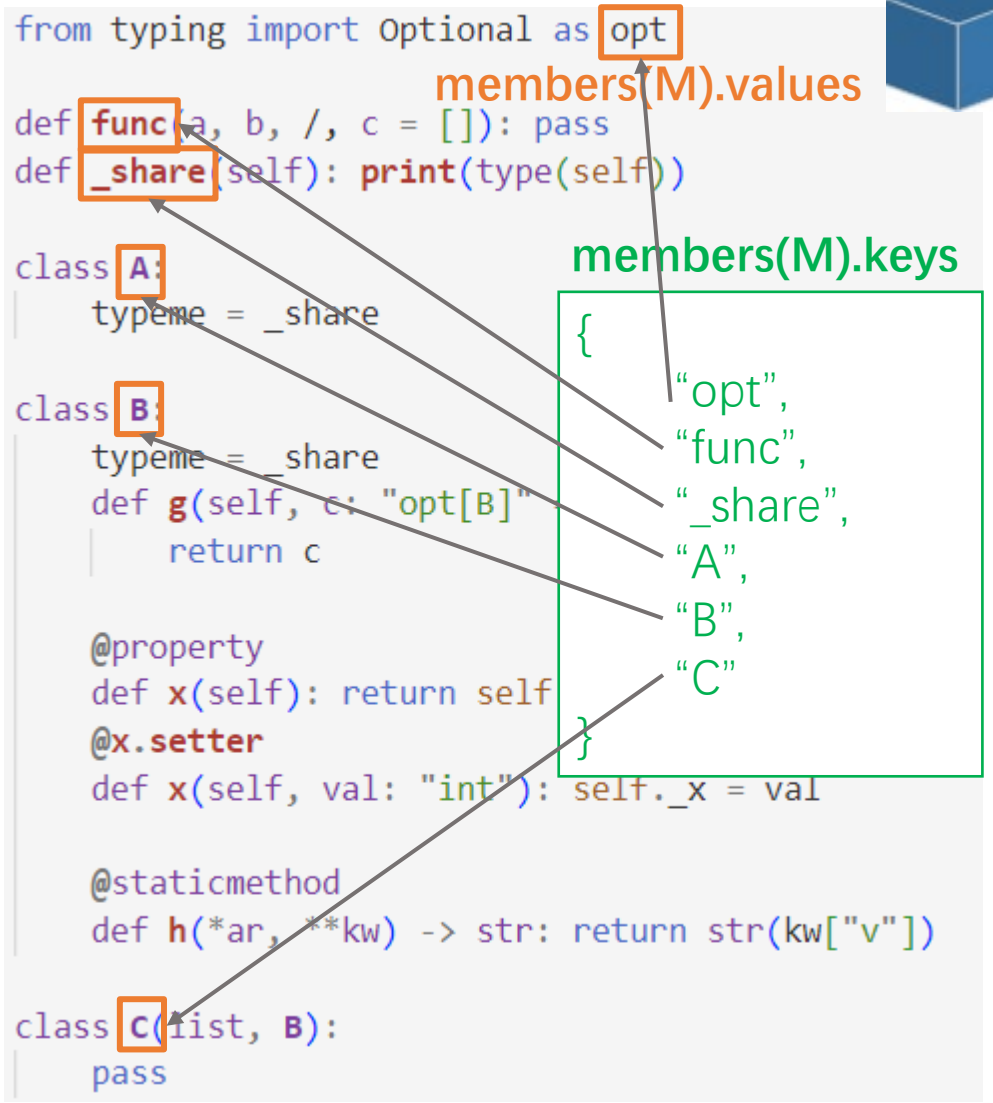
Function

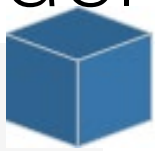
Parameters

Return type

AexPy's Approach

Extraction – API Model





Module

Membership

Aliases

Class

Inheritance

Abstract base classes (ABCs)

{ API }

Attribute

Instance attributes

Type

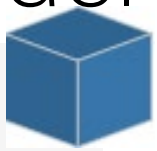
Function

Parameters

Return type

AexPy's Approach

Extraction – API Model



```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

bases(C)

abcs(C)

Sequence
Iterable

Module

Membership

Aliases

Class

Inheritance

Abstract base classes (ABCs)

{ API }

Attribute

Instance attributes

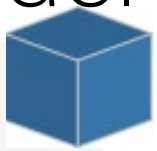
Type

Function

Parameters

Return type

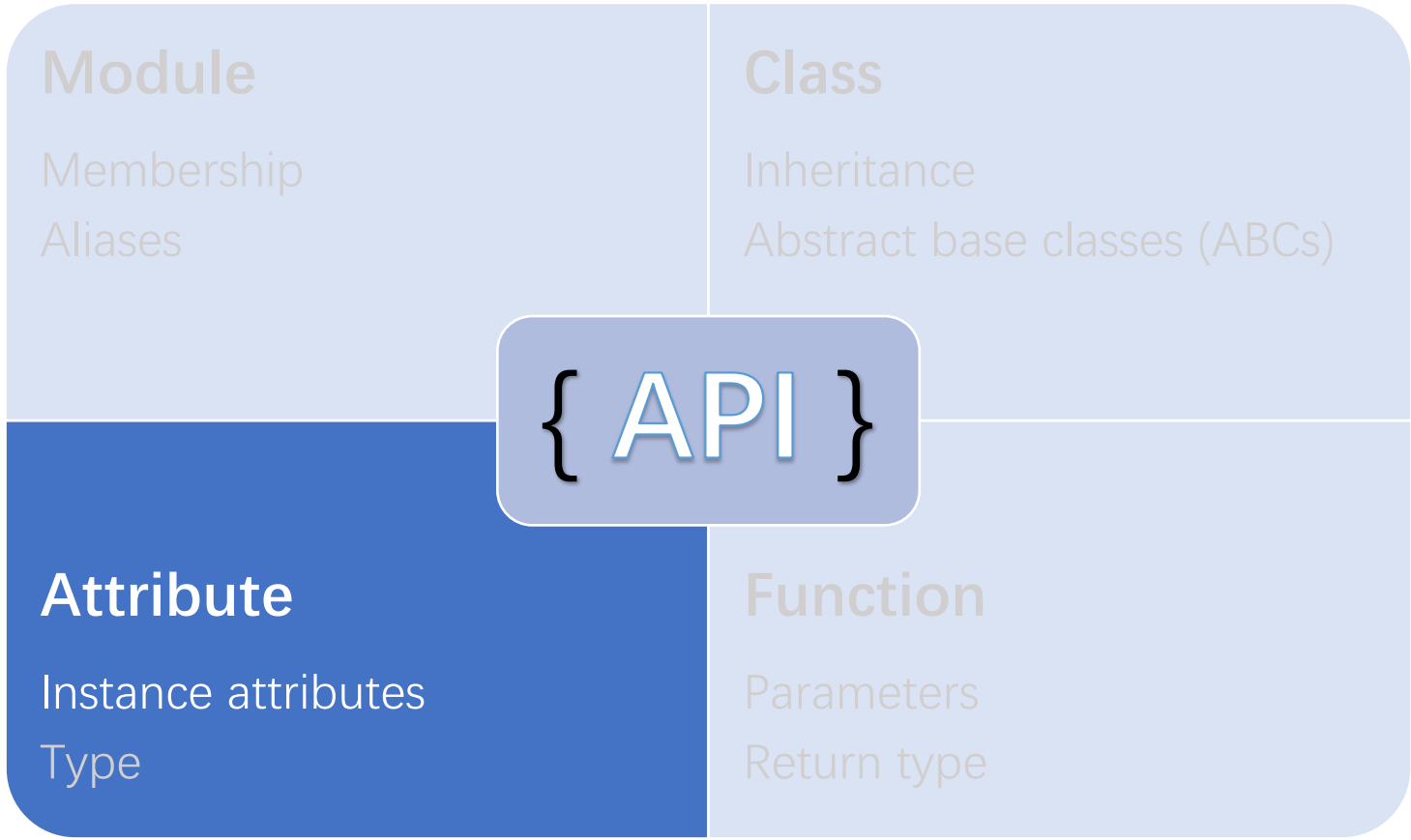
AexPy's Approach
Extraction – API Model



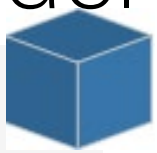
```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Normal attributes

Instance attributes



AexPy's Approach
Extraction – API Model



```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Optional parameter

Variadic parameters

Return type

Module

Membership

Aliases

Class

Inheritance

Abstract base classes (ABCs)

{ API }

Attribute

Instance attributes

Type

Function

Parameters

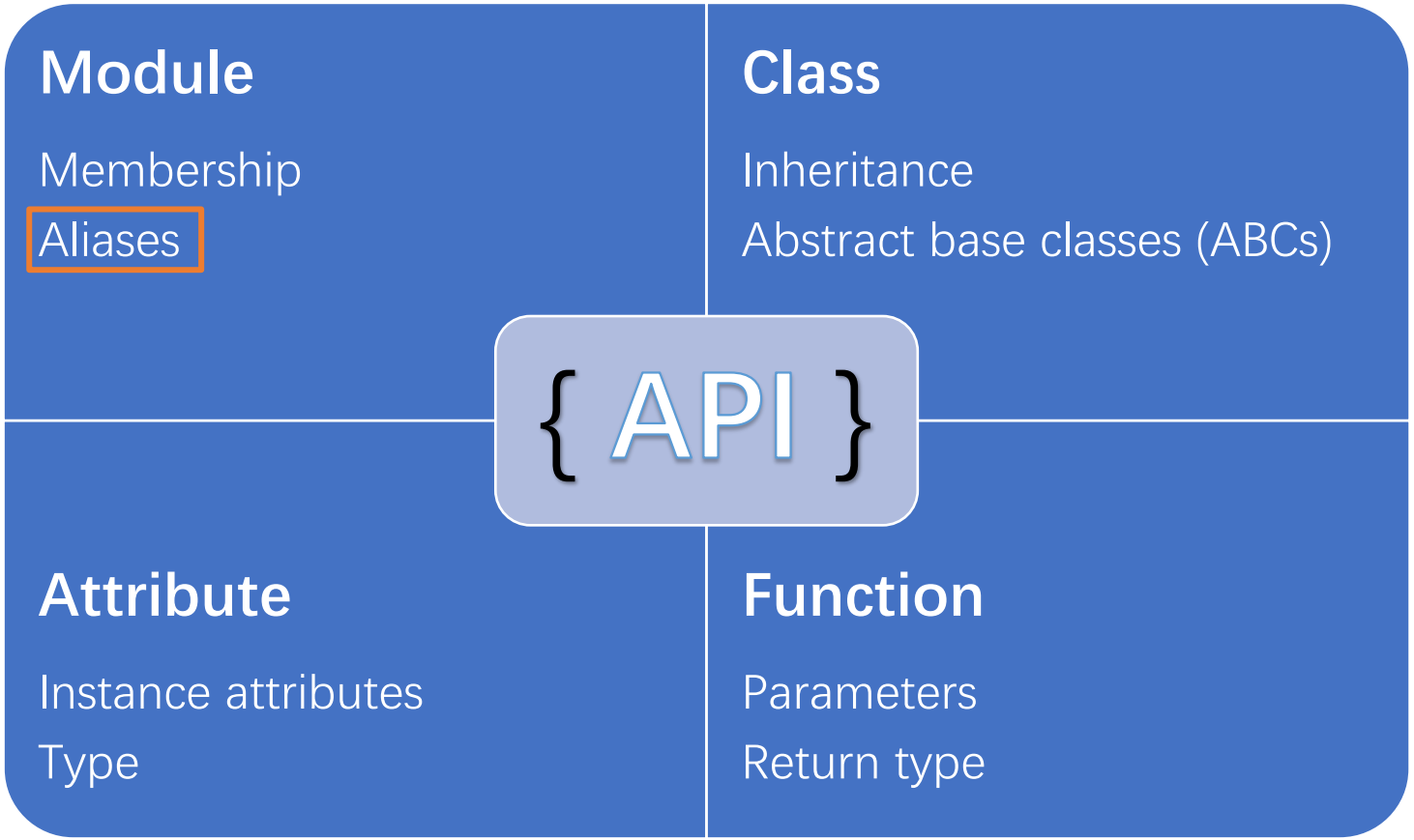
Return type

AexPy's Approach
Extraction – API Model

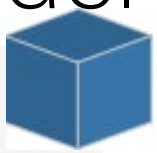
```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

aliases(_share)

A.typeme
B.typeme



AexPy's Approach
Extraction – API Model



```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Sum: $T_1 + T_2 + \cdots + T_n,$

Product: $T_1 \times T_2 \times \cdots \times T_n,$

Callable: $T_{args} \to T_{ret},$

Generic: $T_{base}(T_1, T_2, \ldots, T_n)$

Module

Membership

Aliases

Class

Inheritance

Abstract base classes (ABCs)

{ API }

Attribute

Instance attributes

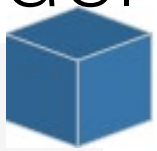
Type

Function

Parameters

Return type

AexPy's Approach
Extraction – API Model



```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

type(c) = **B** + **none**

Sum: $T_1 + T_2 + \cdots + T_n$,

Product: $T_1 \times T_2 \times \cdots \times T_n$,

Callable: $T_{args} \to T_{ret}$,

Generic: $T_{base}(T_1, T_2, \ldots, T_n)$

Module

Membership

Aliases

Class

Inheritance

Abstract base classes (ABCs)

{ API }

Attribute

Instance attributes

Type

Function

Parameters

Return type

AexPy's Approach
Extraction – Algorithm

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Dynamic Reflection

- Breadth-first search
- Inspect live objects

Static Analysis

- Traverse ASTs
- Gain types from Mypy

AexPy's Approach
Extraction – Algorithm

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Dynamic Reflection

Static Analysis

Breadth-first search

```
modules:

classes:

attributes:
```

```
functions:
```

AexPy's Approach
Extraction – Algorithm

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Dynamic Reflection

Breadth-first search

```
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:

attributes:
  M.B.x:
```

Static Analysis

```
functions:
  M.func:

  M._share:
    aliases: [M.A.typeme, M.B.typeme]

  M.B.g:
```

AexPy's Approach
Extraction – Algorithm

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Dynamic Reflection

Breadth-first search

```
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:

attributes:
  M.B.x:
```

Static Analysis

```
functions:
  M.func:

  M._share:
    aliases: [M.A.typeme, M.B.typeme]

  M.B.g:
```

AexPy's Approach
Extraction – Algorithm

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Dynamic Reflection

Breadth-first search

```
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:

attributes:
  M.B.x:
```

Static Analysis

```
functions:
  M.func:

  M._share:
    aliases: [M.A.typeme, M.B.typeme]

  M.B.g:
```

AexPy's Approach
Extraction – Algorithm

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Dynamic Reflection

Inspect live objects

```
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:

attributes:
  M.B.x:
```

Static Analysis

```
functions:
  M.func:

  M._share:
    aliases: [M.A.typeme, M.B.typeme]

  M.B.g:
```

AexPy's Approach
Extraction – Algorithm

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Dynamic Reflection — Inspect live objects

Static Analysis

```yaml
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:
    bases: [list, M.B]
    abcs: [Sequence, Iterable]
attributes:
  M.B.x:
    scope: instance
```

```yaml
functions:
  M.func:
    parameters:
      - name: a
        kind: Positional
      - name: b
        kind: Positional
      - name: c
        kind: PositionalOrKeyword
        optional: true
        default: <object>
  M._share:
    aliases: [M.A.typeme, M.B.typeme]
    parameters:
      - name: self
        kind: PositionalOrKeyword
  M.B.g:
    parameters:
      - name: self
        kind: PositionalOrKeyword
      - name: c
        kind: PositionalOrKeyword
        optional: true
```

AexPy's Approach
Extraction – Algorithm

Dynamic Reflection

Static Analysis

Traverse ASTs

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

```yaml
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:
    bases: [list, M.B]
    abcs: [Sequence, Iterable]
attributes:
  M.B.x:
    scope: instance
  M.B._x:
    scope: instance
```

```yaml
functions:
  M.func:
    parameters:
      - name: a
        kind: Positional
      - name: b
        kind: Positional
      - name: c
        kind: PositionalOrKeyword
        optional: true
        default: <object>
  M._share:
    aliases: [M.A.typeme, M.B.typeme]
    parameters:
      - name: self
        kind: PositionalOrKeyword
  M.B.g:
    parameters:
      - name: self
        kind: PositionalOrKeyword
      - name: c
        kind: PositionalOrKeyword
        optional: true
```

AexPy's Approach
Extraction – Algorithm

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

Dynamic Reflection → **Static Analysis**

Gain types from Mypy

```yaml
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:
    bases: [list, M.B]
    abcs: [Sequence, Iterable]
attributes:
  M.B.x:
    scope: instance
  M.B._x:
    scope: instance
```

```yaml
functions:
  M.func:
    parameters:
      - name: a
        kind: Positional
      - name: b
        kind: Positional
      - name: c
        kind: PositionalOrKeyword
        optional: true
        default: <object>
  M._share:
    aliases: [M.A.typeme, M.B.typeme]
    parameters:
      - name: self
        kind: PositionalOrKeyword
  M.B.g:
    parameters:
      - name: self
        kind: PositionalOrKeyword
      - name: c
        kind: PositionalOrKeyword
        optional: true
        type:
          category: union
          components: [B, none]
    return:
      category: union
      components: [B, none]
```

AexPy's Approach

Detection – Classification

	Module	Class	Function	Attribute	Parameter	Alias
Addition						
Removal						
Modification						

AexPy's Approach

Detection – Classification

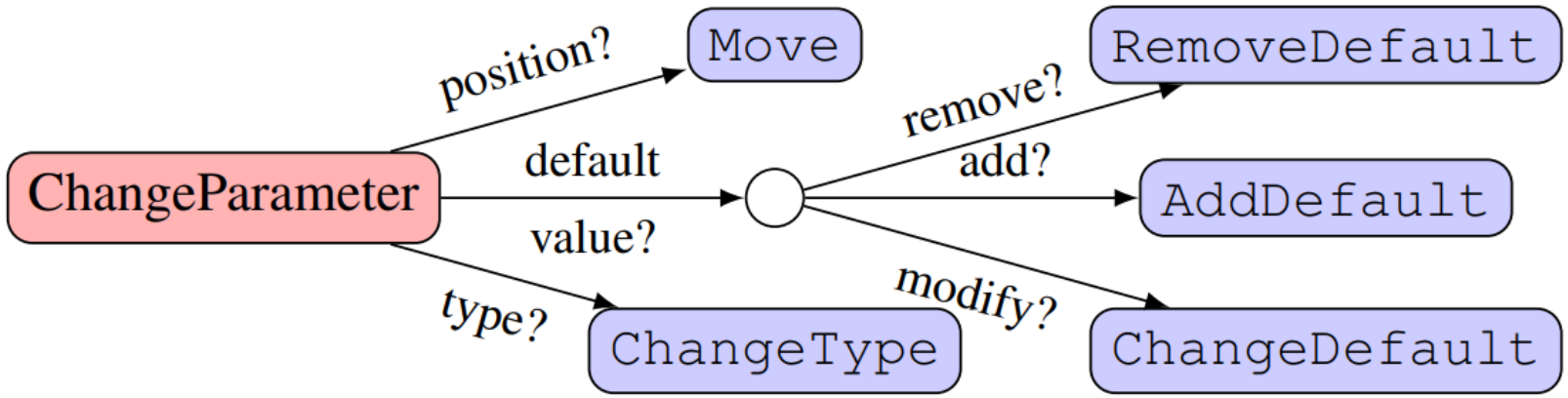
	Module	**Class**	**Function**	**Attribute**	**Parameter**	**Alias**
Addition	AddModule	AddClass	AddFunction	AddAttribute	*AddParameter*[*]	AddAlias
Removal	*RemoveModule*	*RemoveClass*	*RemoveFunction*	*RemoveAttribute*	*RemoveParameter*	*RemoveAlias*
Modification	-[†]	*ChangeInheritance*	*ChangeReturnType*	*ChangeAttributeType*	*ChangeParameter*	ChangeAlias

AexPy's Approach

Detection – Classification

42 change patterns

	Module	Class	Function	Attribute	Parameter	Alias
Addition	AddModule	AddClass	AddFunction	AddAttribute	*AddParameter**	AddAlias
Removal	*RemoveModule*	*RemoveClass*	*RemoveFunction*	*RemoveAttribute*	*RemoveParameter*	*RemoveAlias*
Modification	-†	*ChangeInheritance*	*ChangeReturnType*	*ChangeAttributeType*	*ChangeParameter*	ChangeAlias



AexPy's Approach
Detection – Algorithm

Paring → Comparing

Simulate name resolution and argument passing

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) ->
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x =

    @staticmethod
    def h(*ar, **kw) -> str: return st

class C(list, B):
    pass
```

```yaml
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:
    bases: [list, M.B]
    abcs: [Sequence, Iterable]
attributes:
  M.B.x:
    scope: instance
  M.B._x:
    scope: instance
```

```python
from typing import Optional as opt

def func(b, a, /, c = []): pass
def _share(self, obj = None): print(type(

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = va

    @staticmethod
    def h(*ar, **kw) -> str: return str(k

class C(B):
    def g(self, c: "opt[C]" = None) -> "C
```

```yaml
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:
    bases: [M.B]
    abcs: []
attributes:
  M.B.x:
    scope: instance
  M.B._x:
    scope: instance
```

AexPy's Approach
Detection – Algorithm

Paring

Comparing

Simulate name resolution and argument passing

```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) ->
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x =

    @staticmethod
    def h(*ar, **kw) -> str: return st

class C(list, B):
    pass
```

```yaml
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:
    bases: [list, M.B]
    abcs: [Sequence, Iterable]
attributes:
  M.B.x:
    scope: instance
  M.B._x:
    scope: instance
```

```python
from typing import Optional as opt

def func(b, a, /, c = []): pass
def _share(self, obj = None): print(type(

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = va

    @staticmethod
    def h(*ar, **kw) -> str: return str(k

class C(B):
    def g(self, c: "opt[C]" = None) -> "C
```

```yaml
modules:
  M:
    members:
      opt: <external>typing.Optional
      func: M.func
      _share: M._share
      A: M.A
      B: M.B
      C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:
    bases: [M.B]
    abcs: []
attributes:
  M.B.x:
    scope: instance
  M.B._x:
    scope: instance
```

AexPy's Approach
Detection – Algorithm

Paring → Comparing

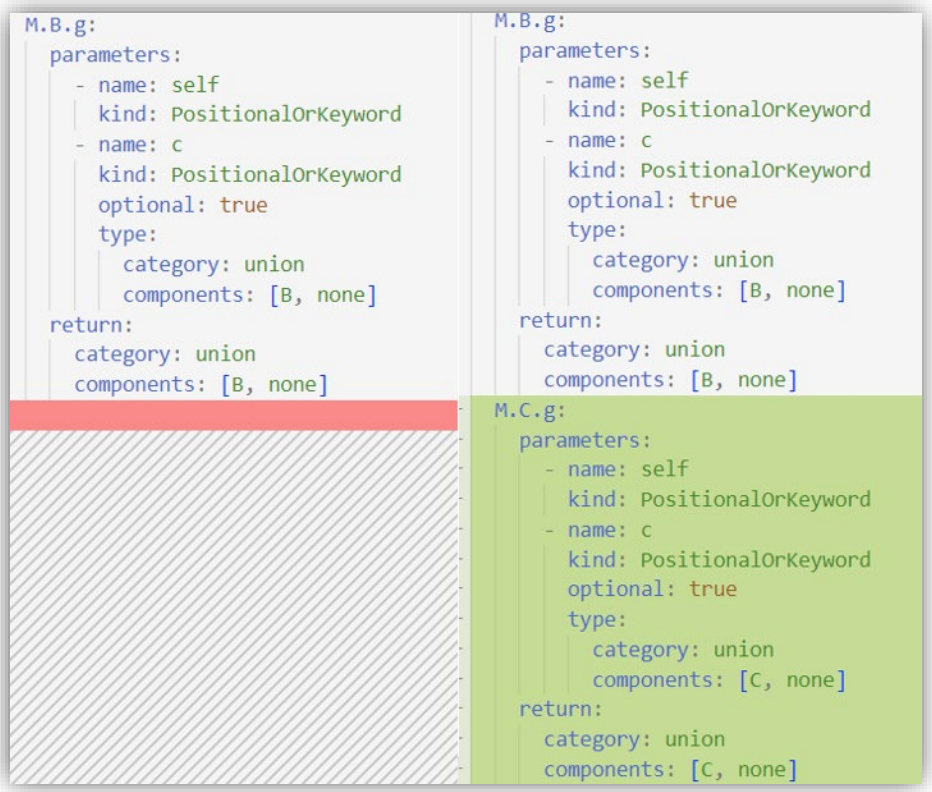
Simulate name resolution and argument passing

```python
def func(a, b, /, c = []): pass
def _share(self): print(type(self))
```

```python
def func(b, a, /, c = []): pass
def _share(self, obj = None): print(type(obj or self))
```

```python
class C(list, B):
    pass
```

```python
class C(B):
    def g(self, c: "opt[C]" = None) -> "C | None": return c
```



AexPy's Approach
Detection – Algorithm

Paring → Comparing

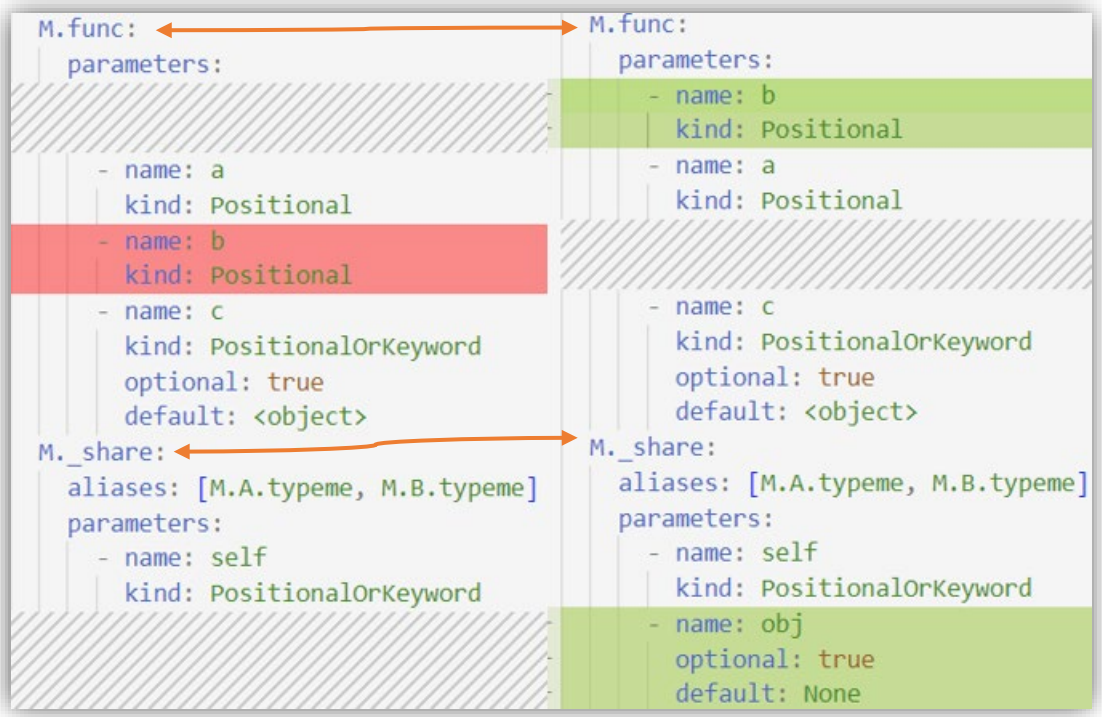
Simulate name resolution and argument passing

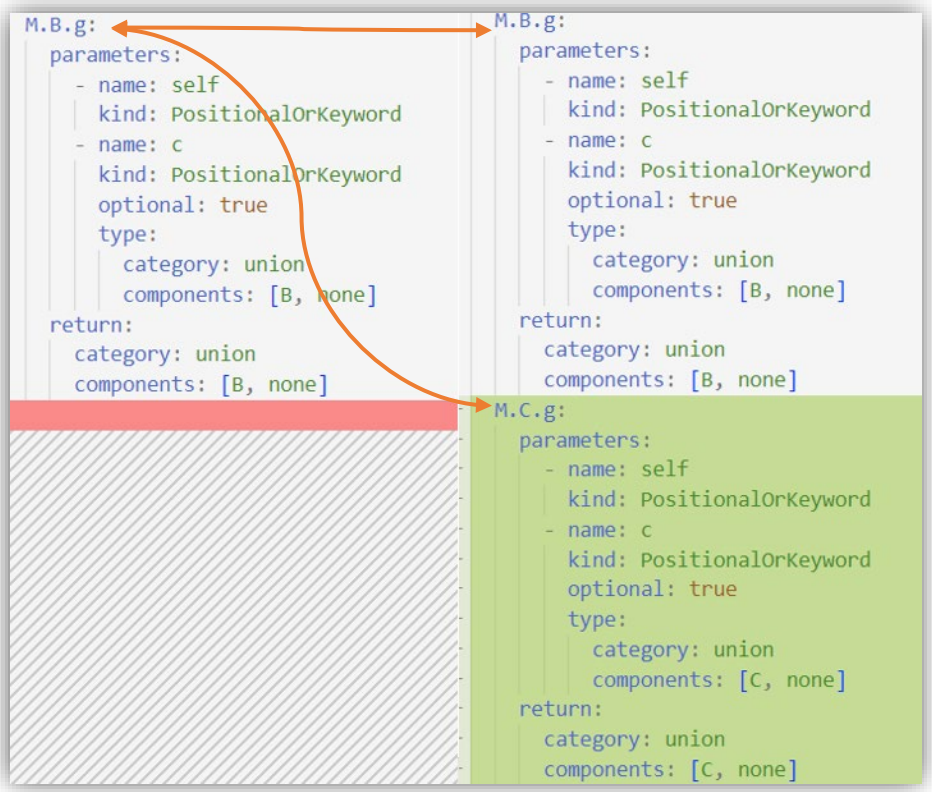
```python
def func(a, b, /, c = []): pass
def _share(self): print(type(self))
```

```python
def func(b, a, /, c = []): pass
def _share(self, obj = None): print(type(obj or self))
```

```python
class C(list, B):
    pass
```

```python
class C(B):
    def g(self, c: "opt[C]" = None) -> "C | None": return c
```





AexPy's Approach
Detection – Algorithm

Paring → Comparing

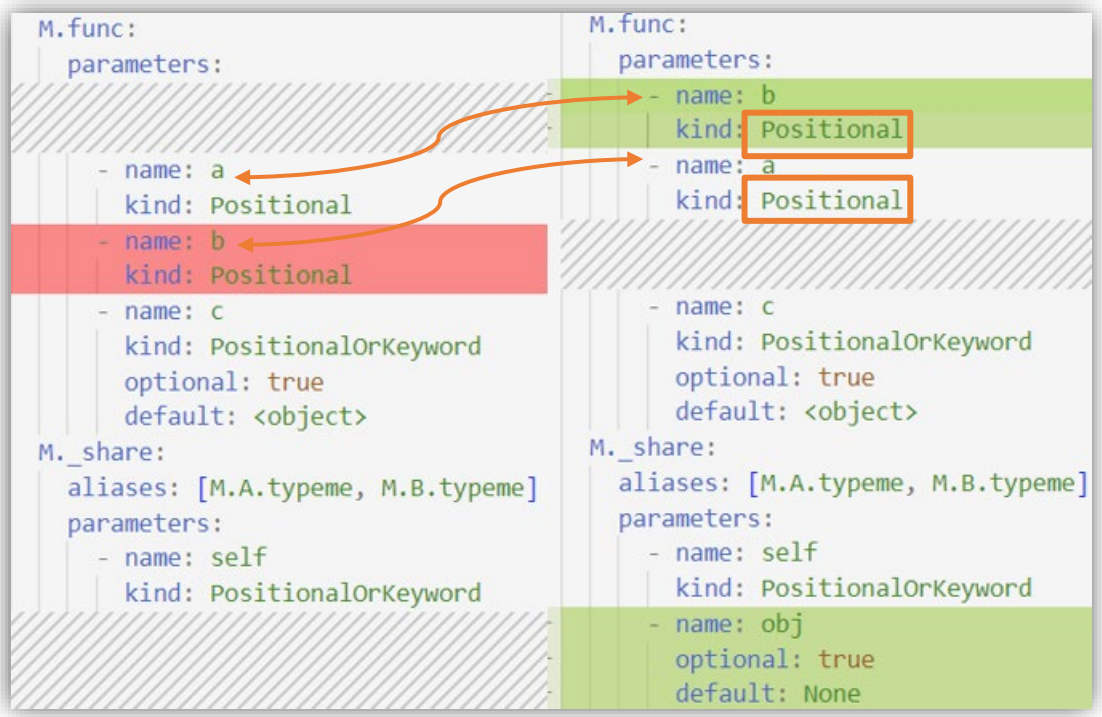
Simulate name resolution and argument passing

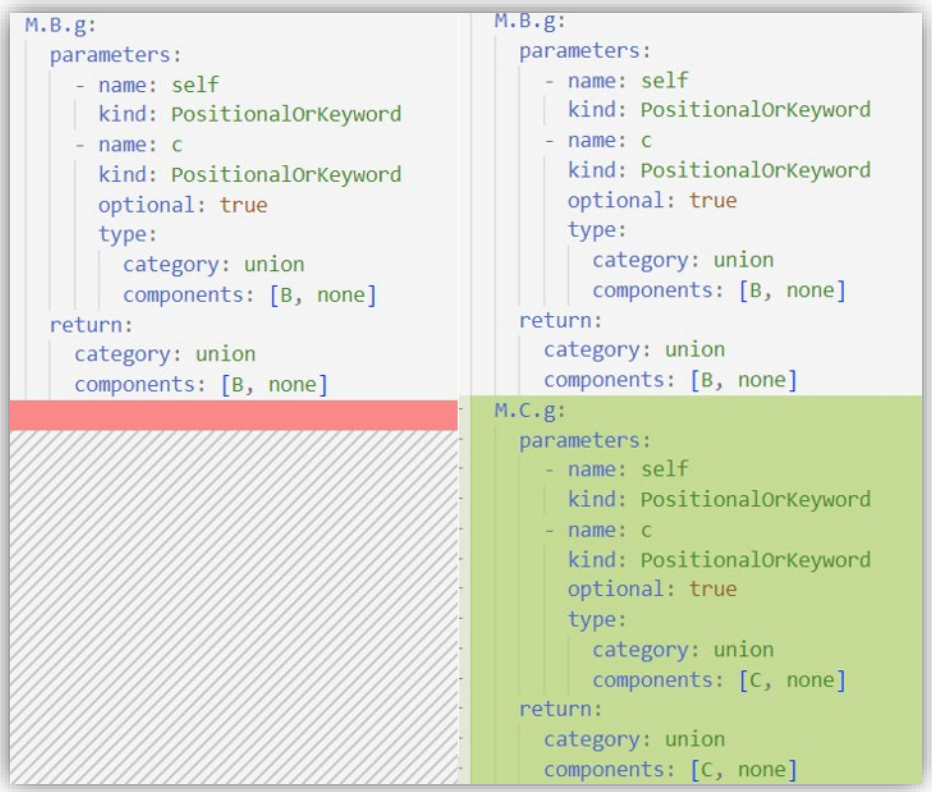
```python
def func(a, b, /, c = []): pass
def _share(self): print(type(self))
```

```python
def func(b, a, /, c = []): pass
def _share(self, obj = None): print(type(obj or self))
```

```python
class C(list, B):
    pass
```

```python
class C(B):
    def g(self, c: "opt[C]" = None) -> "C | None": return c
```





AexPy's Approach

Detection – Algorithm

Paring → Comparing

Define and judge constraints on API pairs

42 change patterns → **42 constraints**

Pattern	Constraint
AddModule	$e = \bot \wedge e' \in M$
RemoveFunction	$e \in F \wedge e' = \bot \wedge \textbf{scope}(e) = \text{Static}$
RemoveBaseClass	$e, e' \in C \wedge \textbf{bases}(e) \not\subseteq \textbf{bases}(e')$
ChangeReturnType	$e, e' \in F \wedge \textbf{return}(e) \neq \textbf{return}(e')$
AddRequiredParameter	$p = \bot \wedge p' \neq \bot \wedge \neg\textbf{optional}(p')$
MoveParameter	$p \neq \bot \wedge p' \neq \bot \wedge \textbf{position}(p) \neq \textbf{position}(p')$
RemoveVarKeywordCandidate	$p \neq \bot \wedge p' = \bot \wedge \textbf{kind}(p) = \text{VarKeywordCandidate}$
RemoveAlias	$e, e' \in M \cup C \wedge (\exists (n, t), t \in E \wedge n \in \textbf{aliases}(t) \wedge (n, t) \in (\textbf{members}(e) - \textbf{members}(e')))$
RemoveExternalAlias	$e, e' \in M \cup C \wedge \exists n, (n, \bot) \in (\textbf{members}(e) - \textbf{members}(e'))$

Full list is at Specification of Changes – AexPy (https://aexpy.netlify.app/change-spec).

AexPy's Approach

Detection – Algorithm

Paring → Comparing

Define and judge constraints on API pairs

RemoveBaseClass

$$e, e' \in C \wedge \mathbf{bases}(e) \not\subseteq \mathbf{bases}(e')$$

```
M.C:
  bases: [list, M.B]
  abcs: [Sequence, Iterable]
```

```
M.C:
  bases: [M.B]
  abcs: []
```

AexPy's Approach

Detection – Algorithm

Paring → **Comparing**

Define and judge constraints on API pairs

MoveParameter

$$p \neq \bot \wedge p' \neq \bot \wedge \mathbf{position}(p) \neq \mathbf{position}(p')$$

position(a) = **0**
position(b) = **1**

```
M.func:
  parameters:
    - name: a
      kind: Positional
    - name: b
      kind: Positional
    - name: c
      kind: PositionalOrKeyword
      optional: true
      default: <object>
```

```
M.func:
  parameters:
    - name: b
      kind: Positional
    - name: a
      kind: Positional
    - name: c
      kind: PositionalOrKeyword
      optional: true
      default: <object>
```

position(a) = **1**
position(b) = **0**

AexPy's Approach
Grading

Weak constraints on Python APIs

+

Lack of compilation checking

→

Difficulty on restricting client usage

New matplotlib breaking change #1172

Closed mmourafiq opened this issue on 20 Nov 2020 · 0 comments

mmourafiq commented on 20 Nov 2020 • edited — Contributor

Describe the bug

```
Traceback (most recent call last):
  File "/usr/local/lib/python3.6/dist-packages/plotly/matplotlylib/mplexporter/exporter.py", line 118, in crawl_fig
    self.crawl_ax(ax)
  File "/usr/local/lib/python3.6/dist-packages/plotly/matplotlylib/mplexporter/exporter.py", line 123, in crawl_ax
    props=utils.get_axes_properties(ax)):
  File "/usr/local/lib/python3.6/dist-packages/plotly/matplotlylib/mplexporter/utils.py", line 272, in get_axes_properti
    'axes': [get_axis_properties(ax.xaxis),
  File "/usr/local/lib/python3.6/dist-packages/plotly/matplotlylib/mplexporter/utils.py", line 236, in get_axis_properti
    props['grid'] = get_grid_style(axis)
  File "/usr/local/lib/python3.6/dist-packages/plotly/matplotlylib/mplexporter/utils.py", line 246, in get_grid_style
    if axis._gridOnMajor and len(gridlines) > 0:
AttributeError: 'XAxis' object has no attribute '_gridOnMajor'
```

related: plotly/plotly.py#2913
related fix: mpld3/mplexporter@ 2f766e4...be8e3da

New matplotlib breaking change · Issue #1172 · polyaxon/polyaxon (github.com)

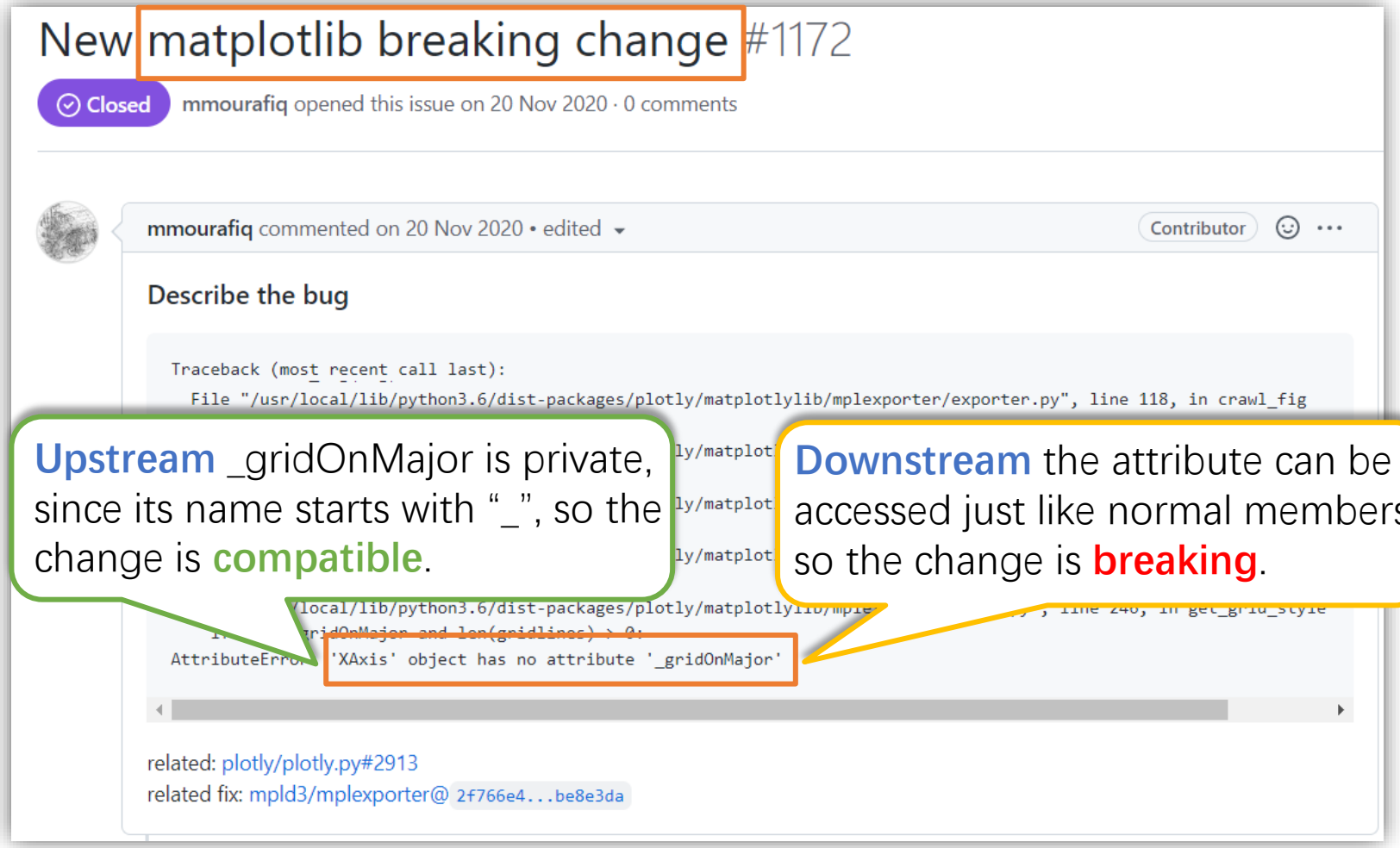
AexPy's Approach
Grading

Weak constraints on Python APIs

+

Lack of compilation checking

→ Difficulty on restricting client usage



New matplotlib breaking change · Issue #1172 · polyaxon/polyaxon (github.com)

AexPy's Approach
Grading

Weak constraints on Python APIs

+

Lack of compilation checking

→ Difficulty on restricting client usage

Backward Compatible / **Breaking**?

New matplotlib breaking change · Issue #1172 · polyaxon/polyaxon (github.com)

AexPy's Approach

Grading

Weak constraints on Python APIs

Lack of compilation checking

Difficulty on restricting client usage

Backward Compatible / **Breaking**?

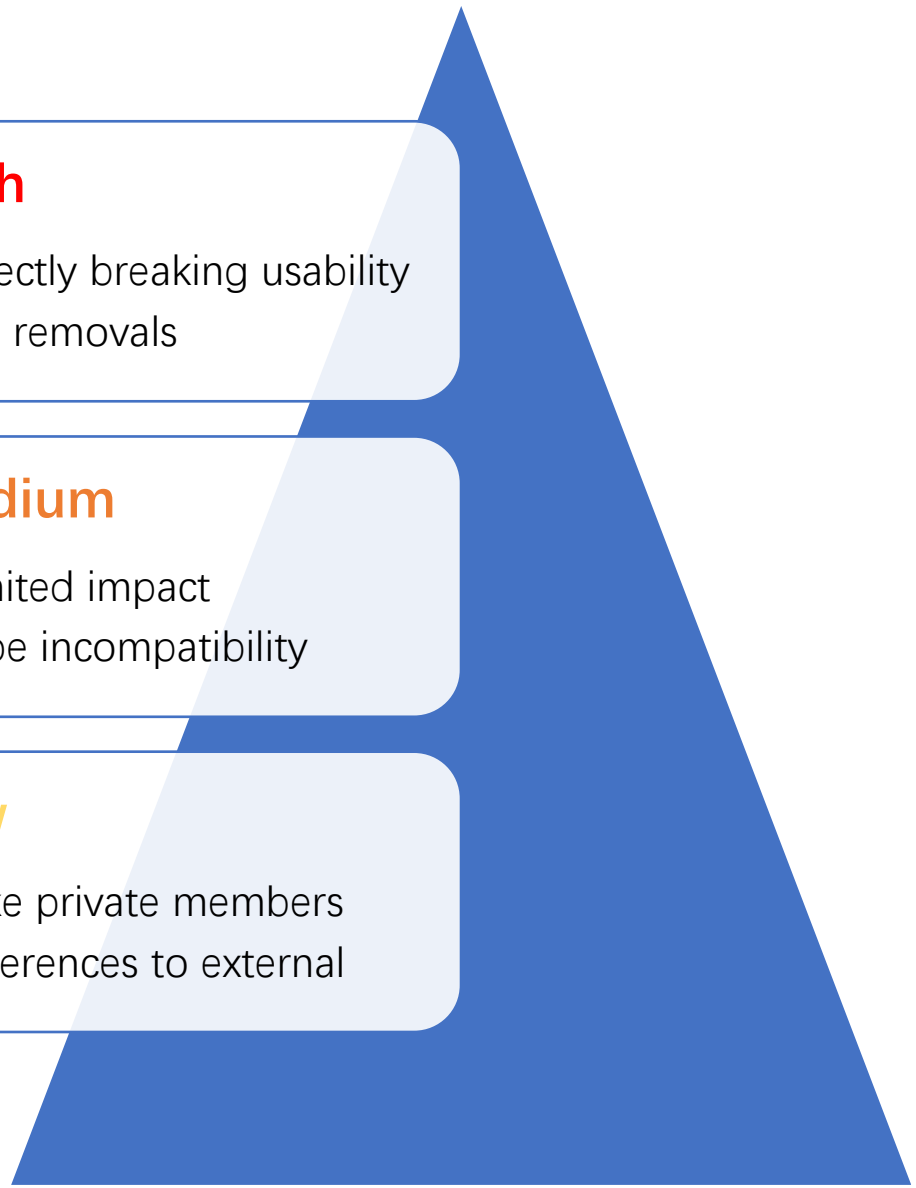
AexPy's Approach

Grading

High Remove base class "list" of class "C".

High Reorder parameter "a" and "b" of "func".

Backward Compatible / **Breaking**?



AexPy's Approach

Grading

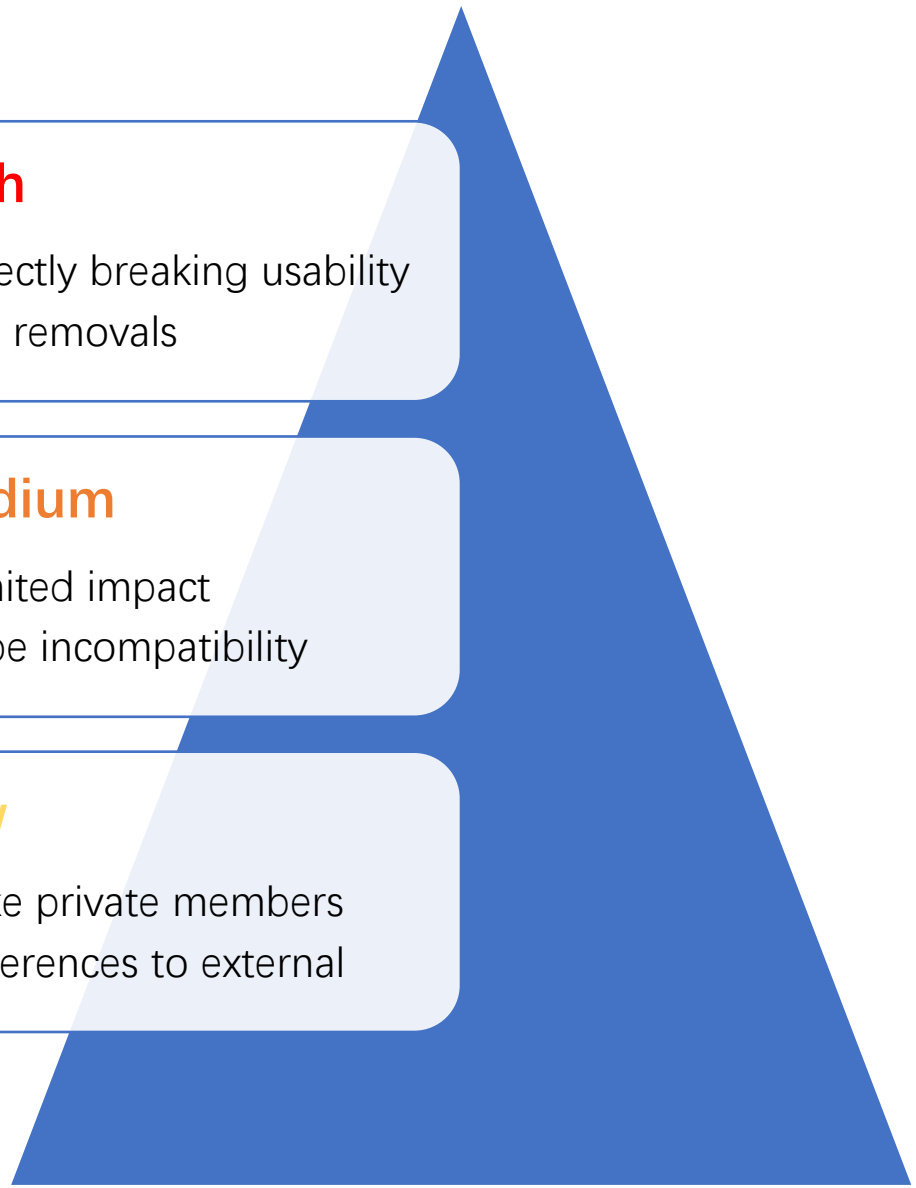
Backward Compatible / **Breaking**?

High Remove base class "list" of class "C".

High Reorder parameter "a" and "b" of "func".

Change type of parameter "c" of function "C.g", which no longer accepts "B".

Change return type of "C.g", which returns "C", a subclass of old return type "B".



AexPy's Approach

Grading

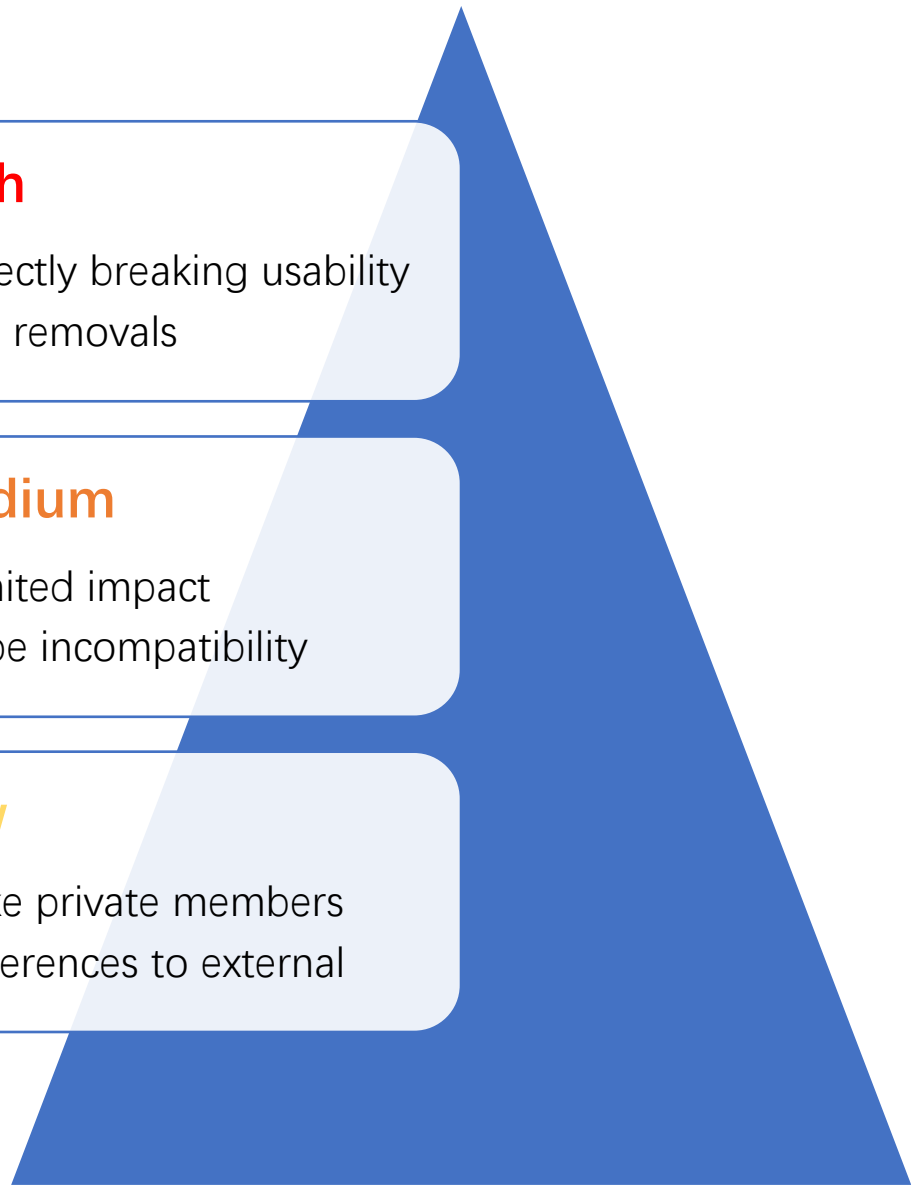
Backward Compatible / **Breaking**?

High Remove base class "list" of class "C".

High Reorder parameter "a" and "b" of "func".

Change **type** of parameter "c" of function "C.g", which no longer accepts "B".

Change **return type** of "C.g", which returns "C", a subclass of old return type "B".



AexPy's Approach

Grading

Backward Compatible / **Breaking**?

High Remove base class "list" of class "C".

High Reorder parameter "a" and "b" of "func".

$$\textsc{Callable:}\ \frac{T_{args} \subseteq S_{args} \qquad S_{ret} \subseteq T_{ret}}{T_{args} \to T_{ret} \subseteq S_{args} \to S_{ret}}$$

Change type of parameter "c" of function "C.g", which no longer accepts "B".

Change return type of "C.g", which returns "C", a subclass of old return type "B".

High

- Directly breaking usability
- API removals

Medium

- Limited impact
- Type incompatibility

Low

- Fake private members
- References to external

AexPy's Approach

Grading

Backward Compatible / **Breaking**?

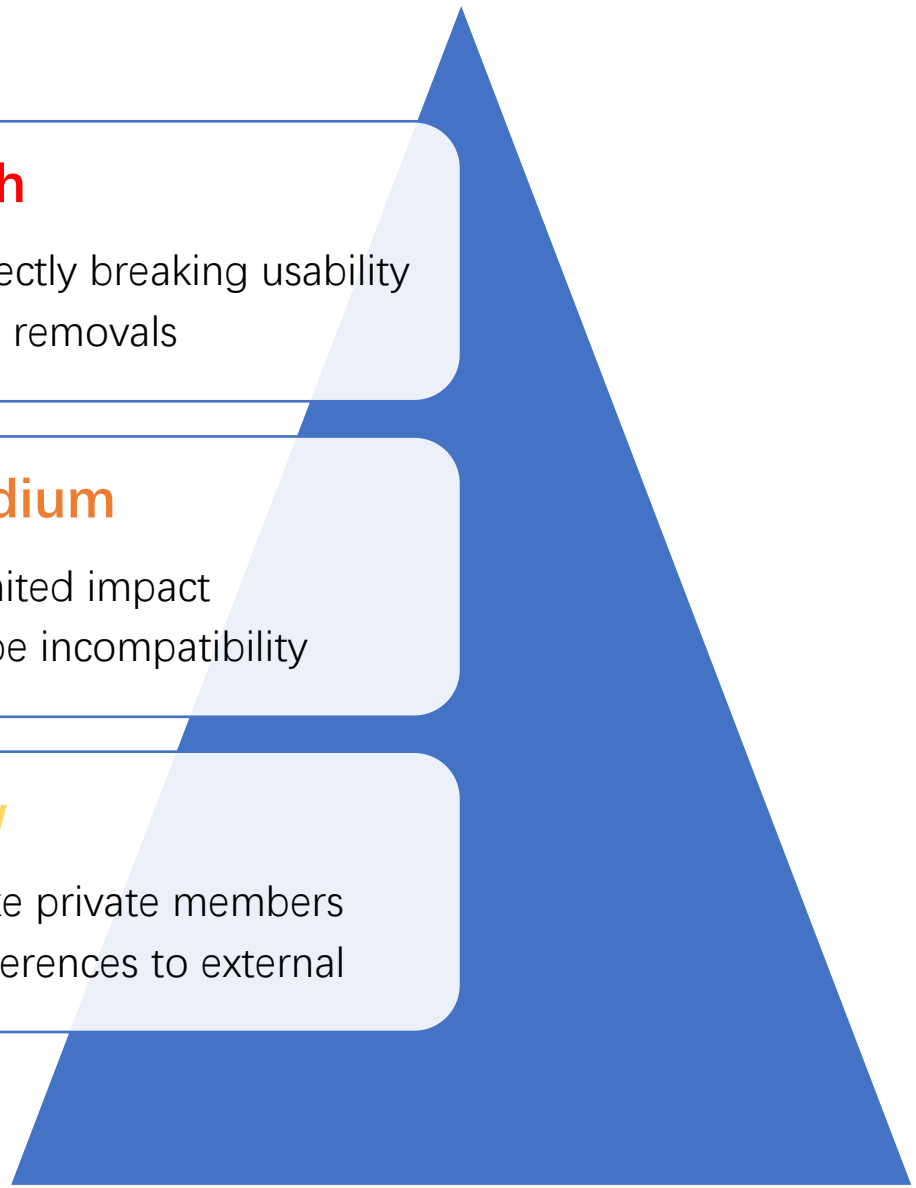
High Remove base class "list" of class "C".

High Reorder parameter "a" and "b" of "func".

$$\textsc{Callable:}\ \frac{T_{args} \subseteq S_{args} \qquad S_{ret} \subseteq T_{ret}}{T_{args} \to T_{ret} \subseteq S_{args} \to S_{ret}}$$

Medium Change type of parameter "c" of function "C.g", which **no longer accepts** "B".

Compatible Change return type of "C.g", which returns "C", **a subclass of old return type** "B".



AexPy's Approach

Grading

Backward Compatible / **Breaking**?

High Remove base class "list" of class "C".

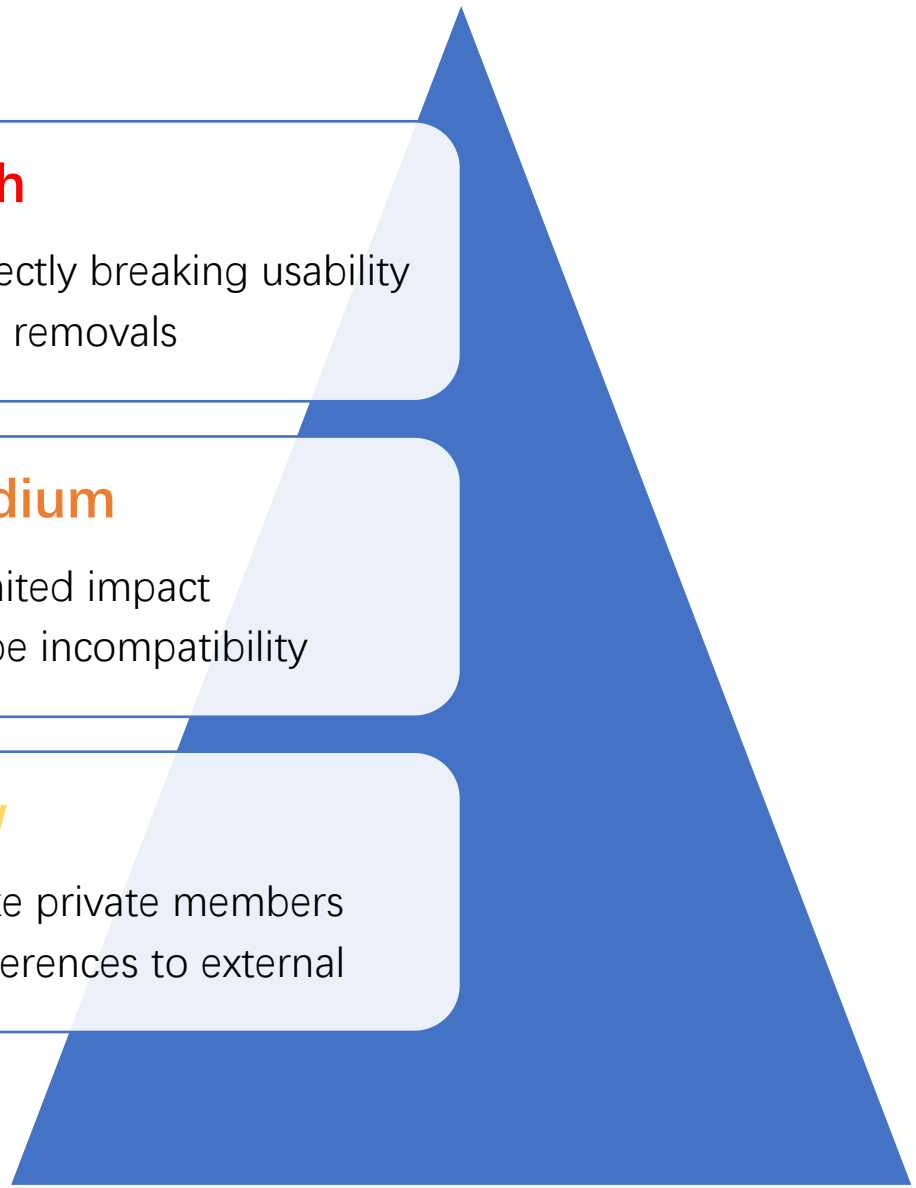
High Reorder parameter "a" and "b" of "func".

$$\textsc{Callable:}\ \frac{T_{args} \subseteq S_{args} \qquad S_{ret} \subseteq T_{ret}}{T_{args} \to T_{ret} \subseteq S_{args} \to S_{ret}}$$

Medium Change type of parameter "c" of function "C.g", which no longer accepts "B".

Compatible Change return type of "C.g", which returns "C", a subclass of old return type "B".

Low Add optional parameter "obj" to "_share".



AexPy's Approach
Summary

Dynamic Language Features

- Dynamic reflection
- Static analysis

Complex API References

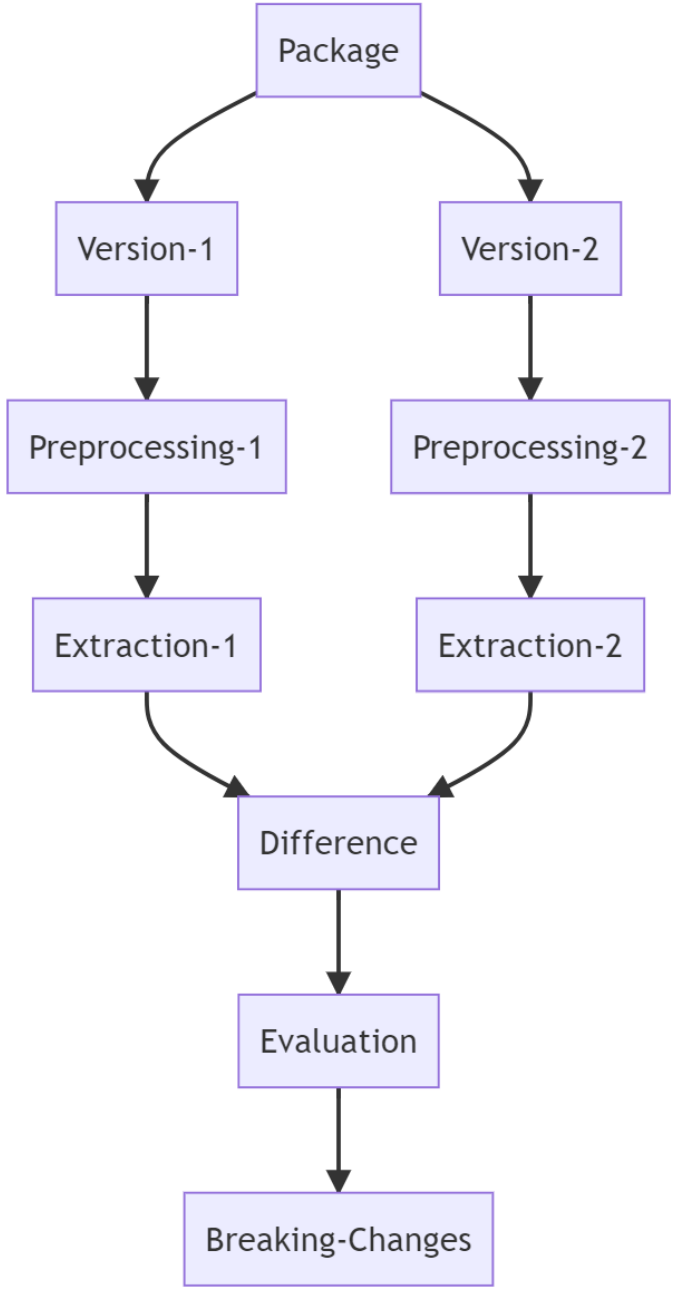
- Import and inspect
- Identify aliases by runtime objects

Fake Private Members

- Breadth-first search for accessible APIs
- Identify by API aliases and grade to low level

Flexible Argument Passing

- Model parameter kinds
- Match parameters in diff algorithm



AexPy's Approach
Summary

Dynamic Language Features

- Dynamic reflection
- Static analysis

Complex API References

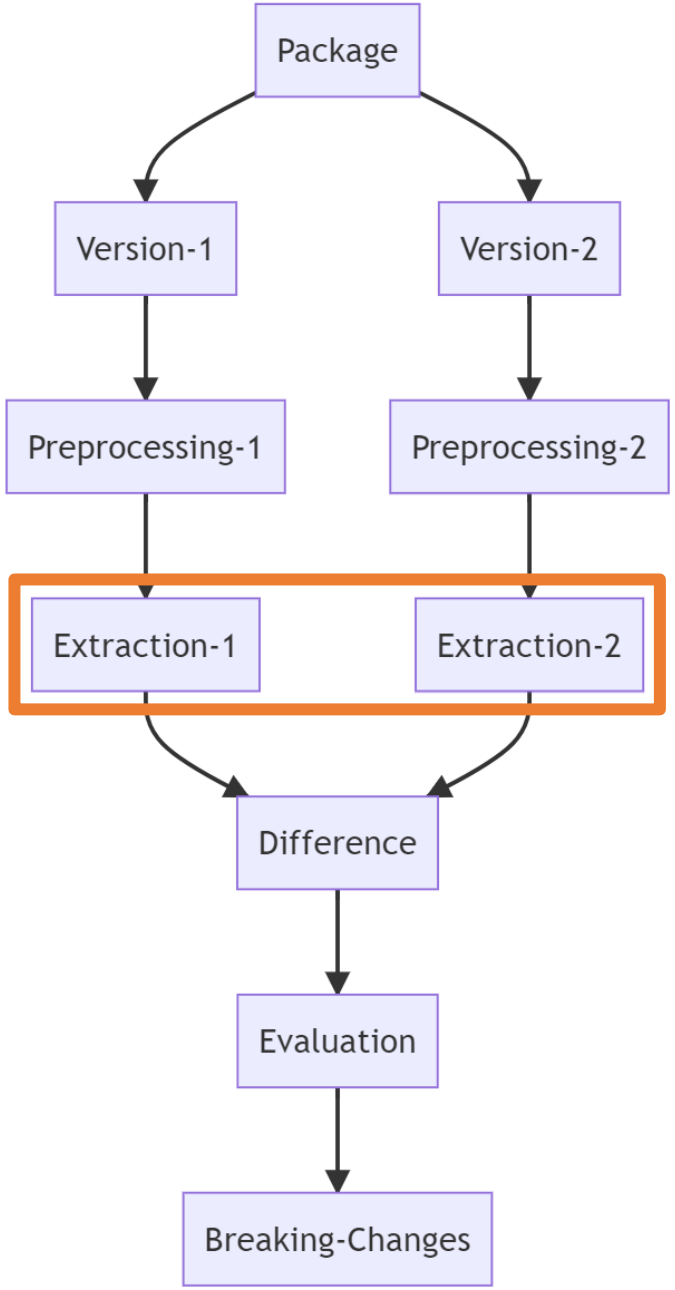
- Import and inspect
- Identify aliases by runtime objects

Fake Private Members

- Breadth-first search for accessible APIs
- Identify by API aliases and grade to low level

Flexible Argument Passing

- Model parameter kinds
- Match parameters in diff algorithm



AexPy's Approach
Summary

Dynamic Language Features

- Dynamic reflection
- Static analysis

Complex API References

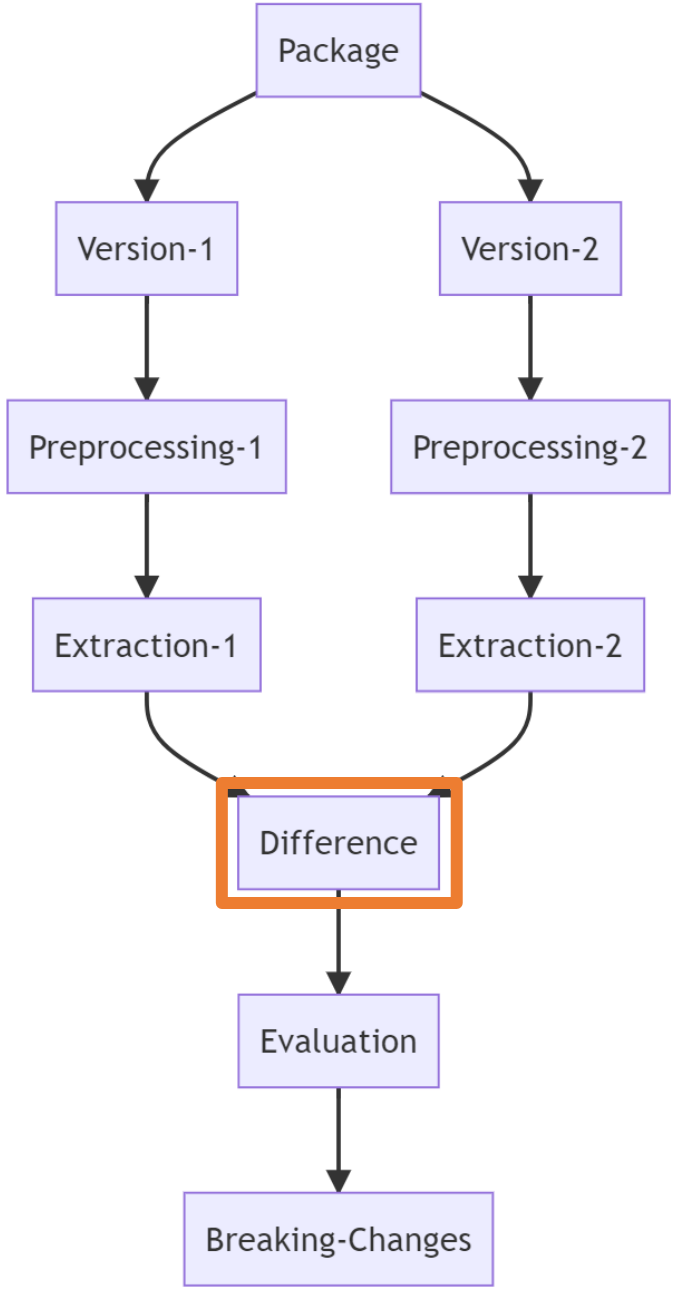
- Import and inspect
- Identify aliases by runtime objects

Fake Private Members

- Breadth-first search for accessible APIs
- Identify by API aliases and grade to low level

Flexible Argument Passing

- Model parameter kinds
- Match parameters in diff algorithm



AexPy's Approach
Summary

Dynamic Language Features

- Dynamic reflection
- Static analysis

Complex API References

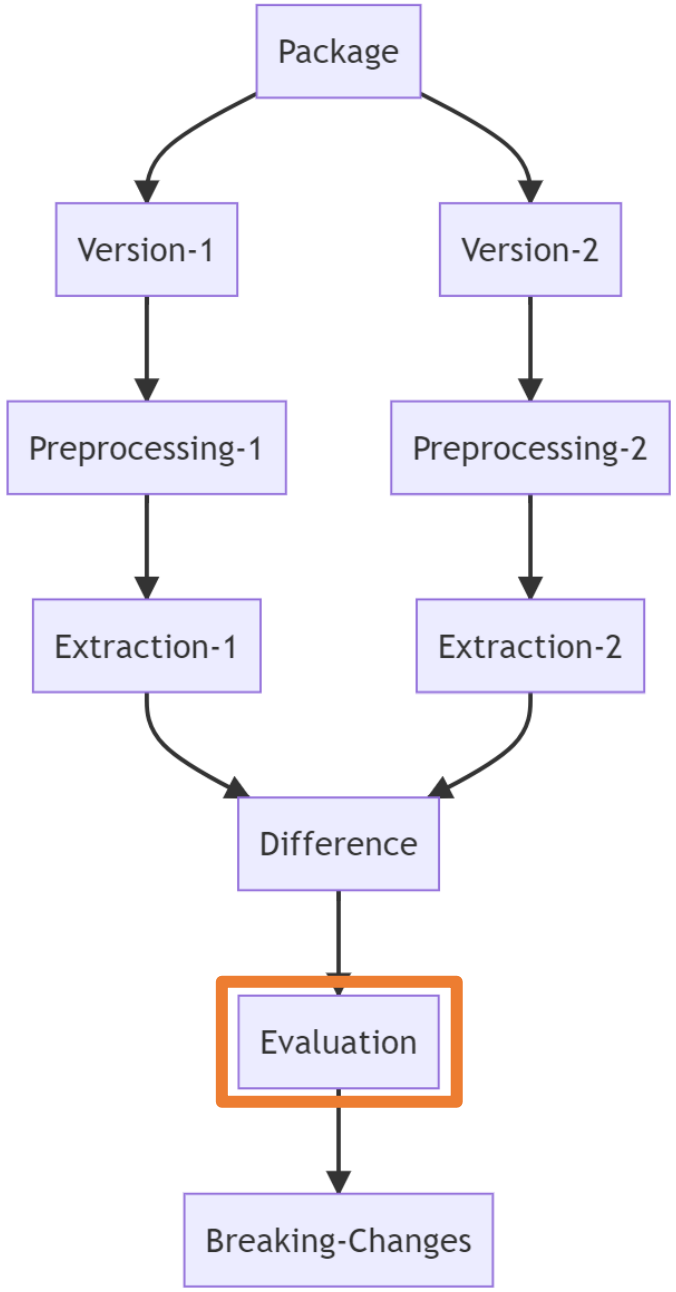
- Import and inspect
- Identify aliases by runtime objects

Fake Private Members

- Breadth-first search for accessible APIs
- Identify by API aliases and grade to low level

Flexible Argument Passing

- Model parameter kinds
- Match parameters in diff algorithm



AexPy's Approach
Summary

Dynamic Language Features

- Dynamic reflection
- Static analysis

Complex API References

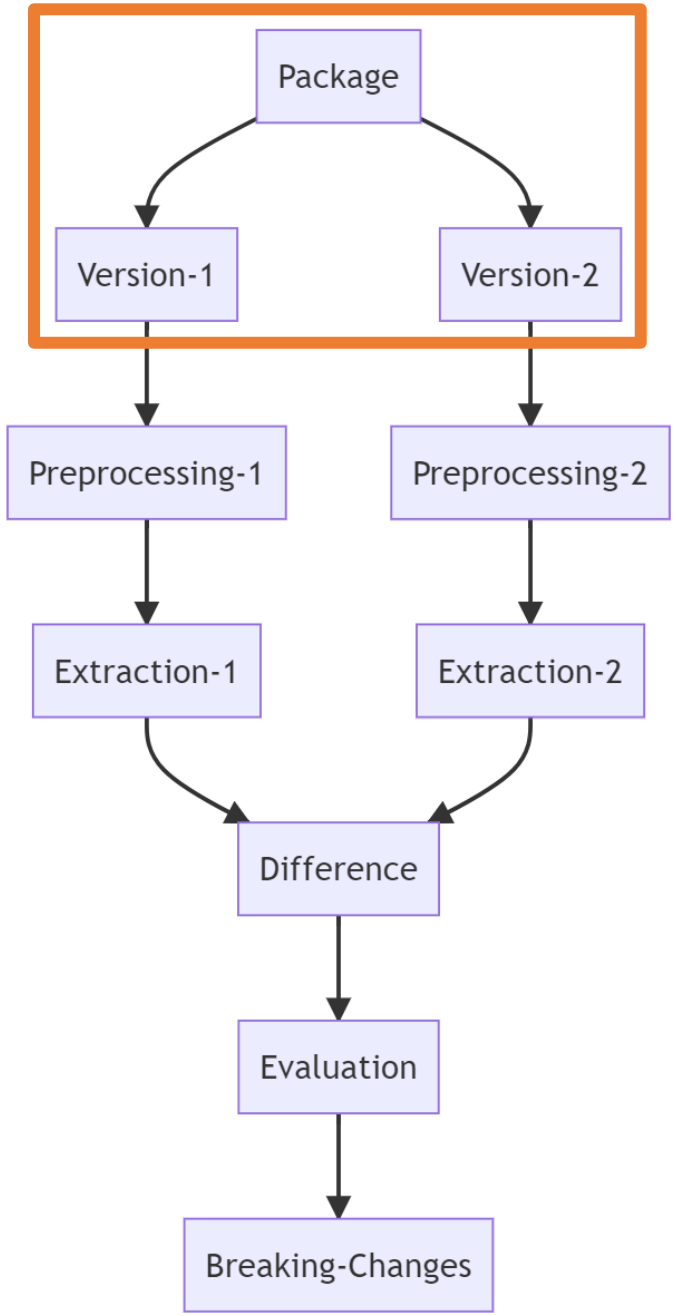
- Import and inspect
- Identify aliases by runtime objects

Fake Private Members

- Breadth-first search for accessible APIs
- Identify by API aliases and grade to low level

Flexible Argument Passing

- Model parameter kinds
- Match parameters in diff algorithm



AexPy's Approach
Summary

Dynamic Language Features

- Dynamic reflection
- Static analysis

Complex API References

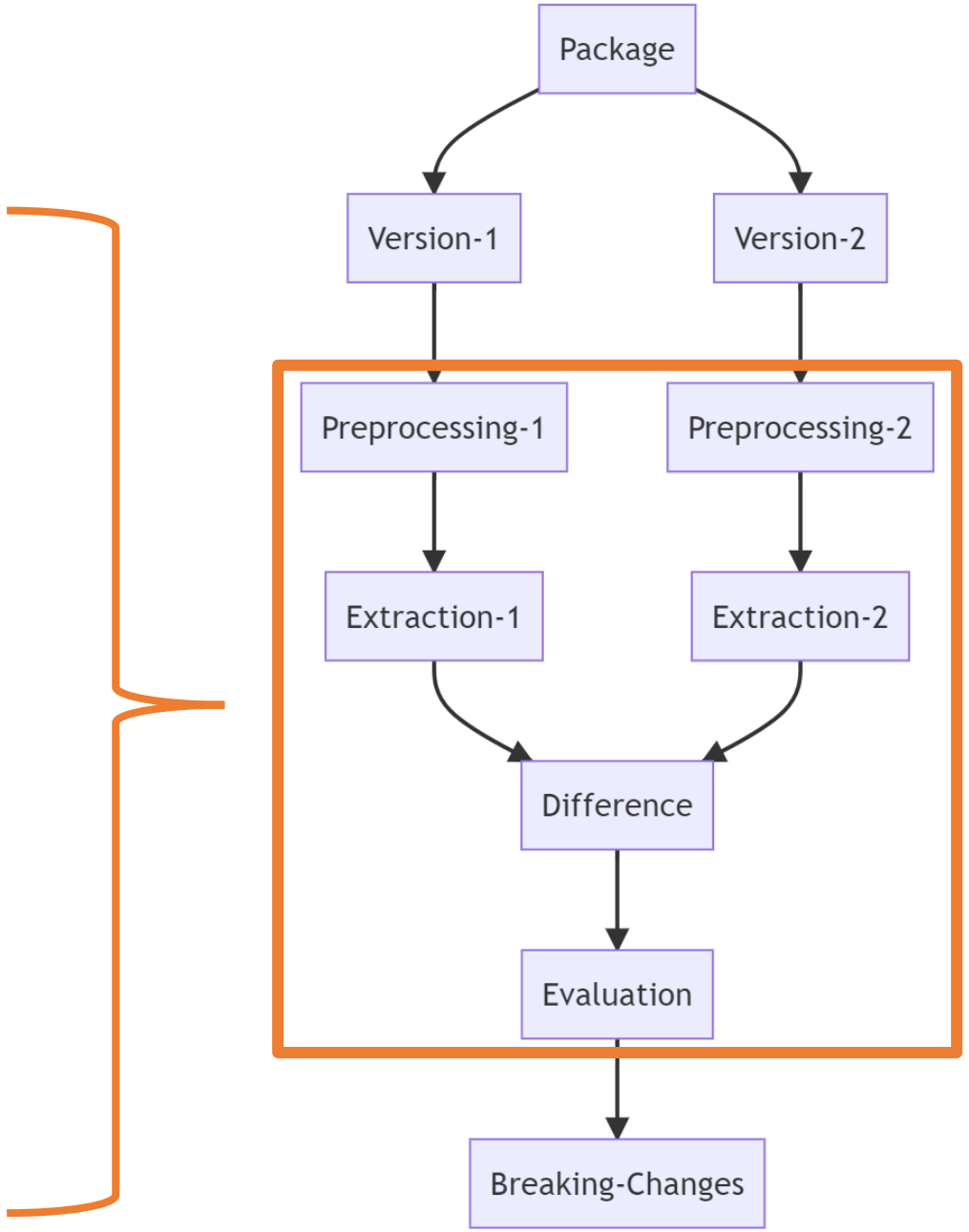
- Import and inspect
- Identify aliases by runtime objects

Fake Private Members

- Breadth-first search for accessible APIs
- Identify by API aliases and grade to low level

Flexible Argument Passing

- Model parameter kinds
- Match parameters in diff algorithm



AexPy's Approach
Summary

Dynamic Language Features

- Dynamic reflection
- Static analysis

Complex API References

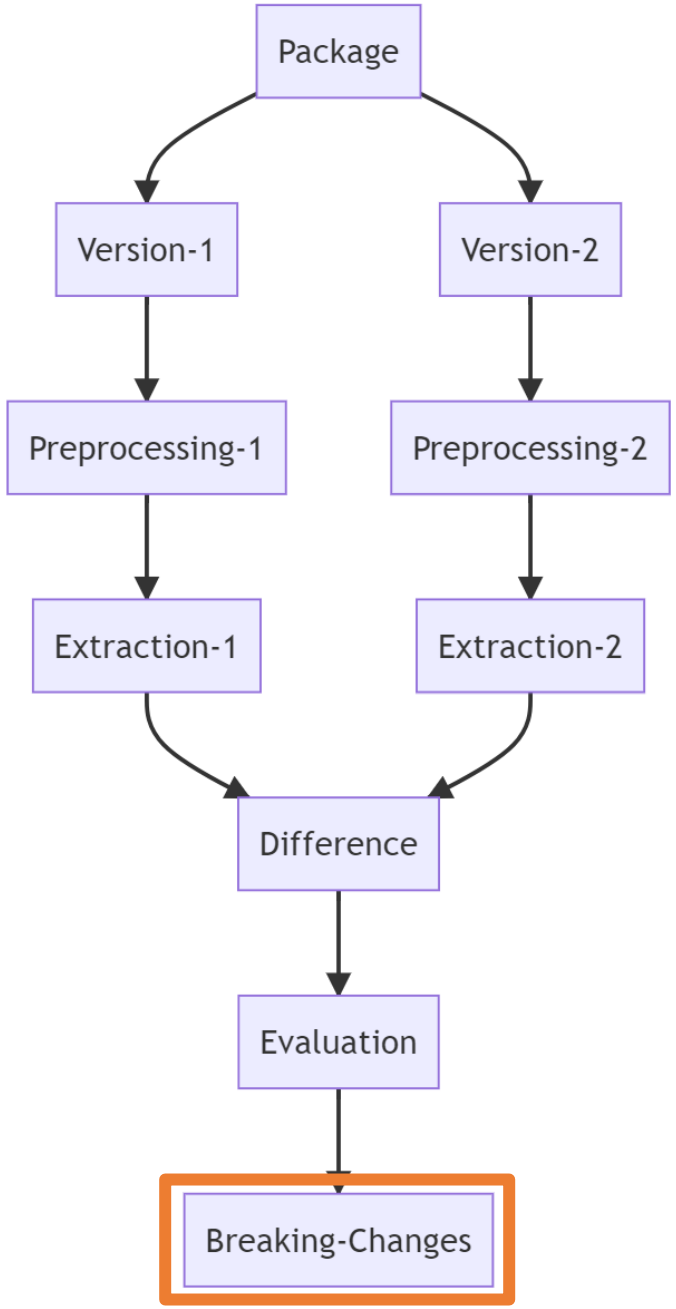
- Import and inspect
- Identify aliases by runtime objects

Fake Private Members

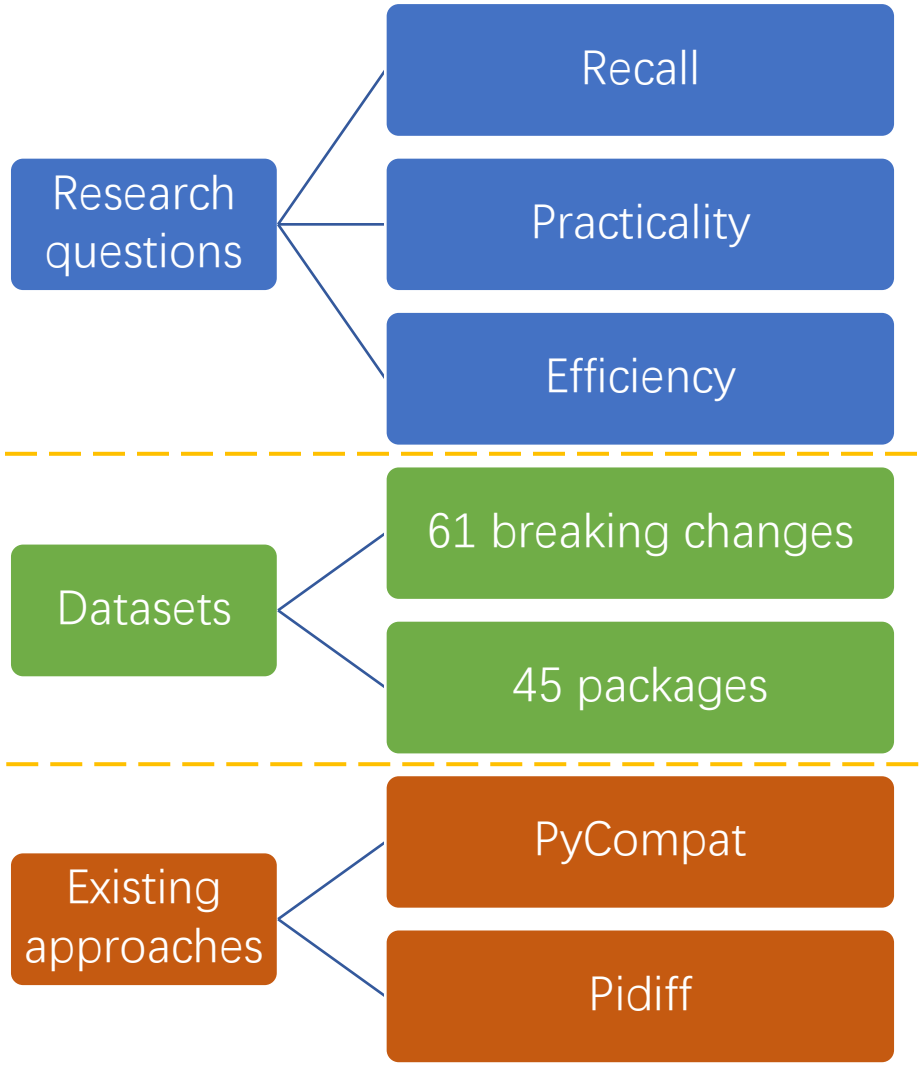
- Breadth-first search for accessible APIs
- Identify by API aliases and grade to low level

Flexible Argument Passing

- Model parameter kinds
- Match parameters in diff algorithm



Evaluation
Setup



Does AexPy detect more **known** breaking changes than existing approaches?

Can AexPy find potential **unknown** breaking changes?

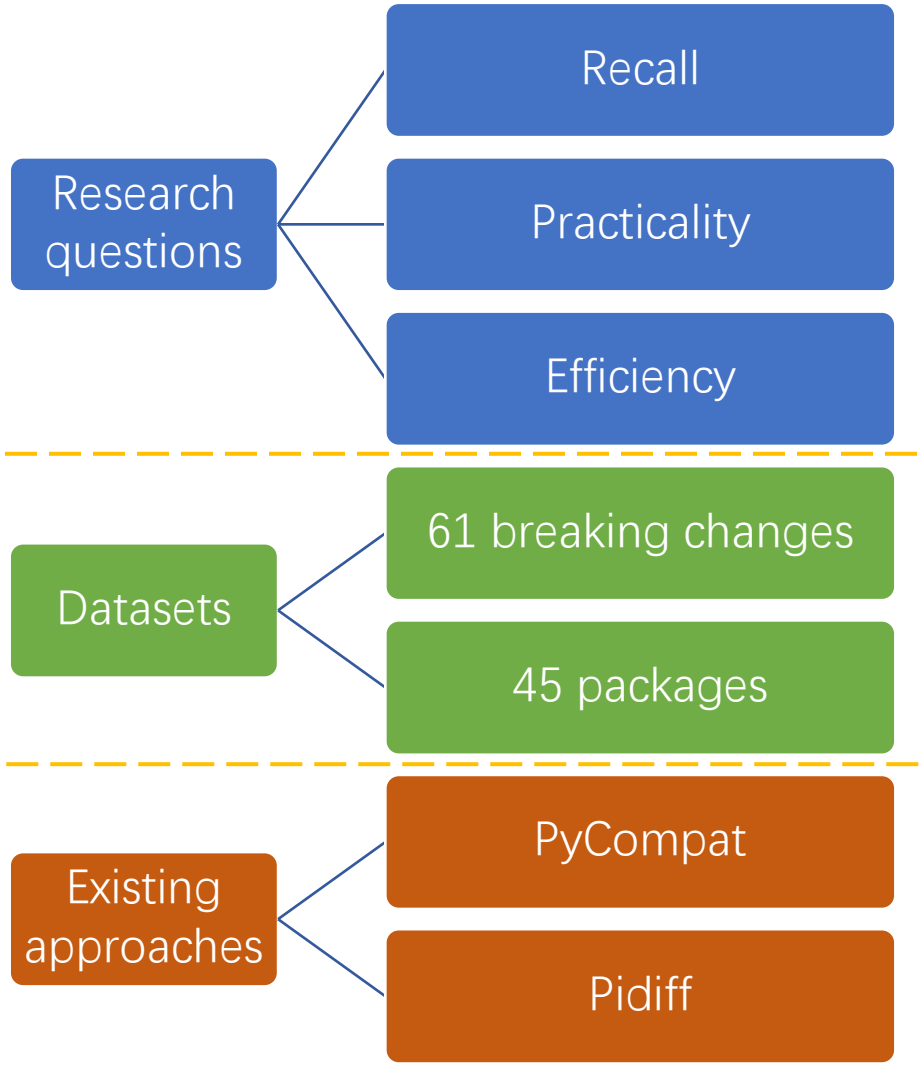
What is the **time** performance of AexPy?

From resolved GitHub issues and 10 popular packages' changelogs

An API misuse detector with its own change detector

A semantic versioning checker for Python packages

Evaluation
Setup



Does AexPy detect more **known** breaking changes than existing approaches?

Can AexPy find potential **unknown** breaking changes?

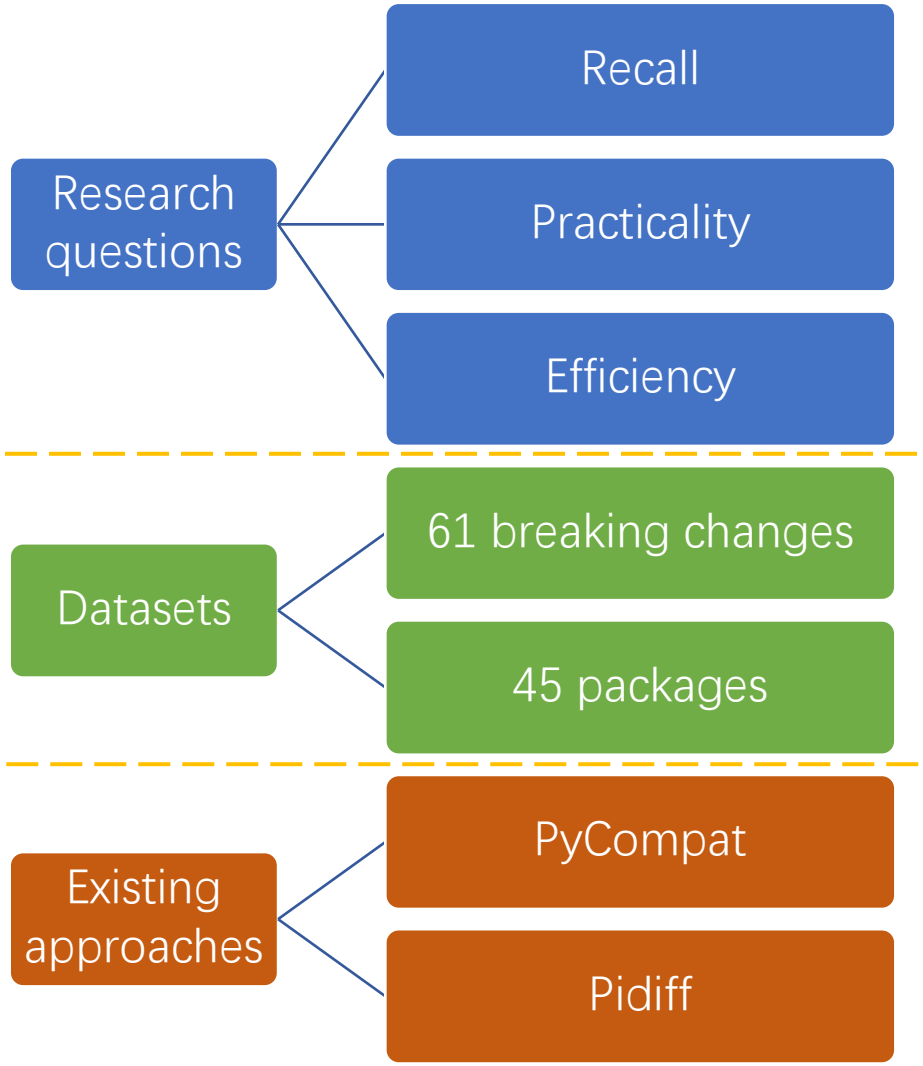
What is the **time** performance of AexPy?

From resolved GitHub **issues** and 10 popular packages' **changelogs**

An API misuse detector with its own change detector

A semantic versioning checker for Python packages

Evaluation
Setup



Does AexPy detect more **known** breaking changes than existing approaches?

Can AexPy find potential **unknown** breaking changes?

What is the **time** performance of AexPy?

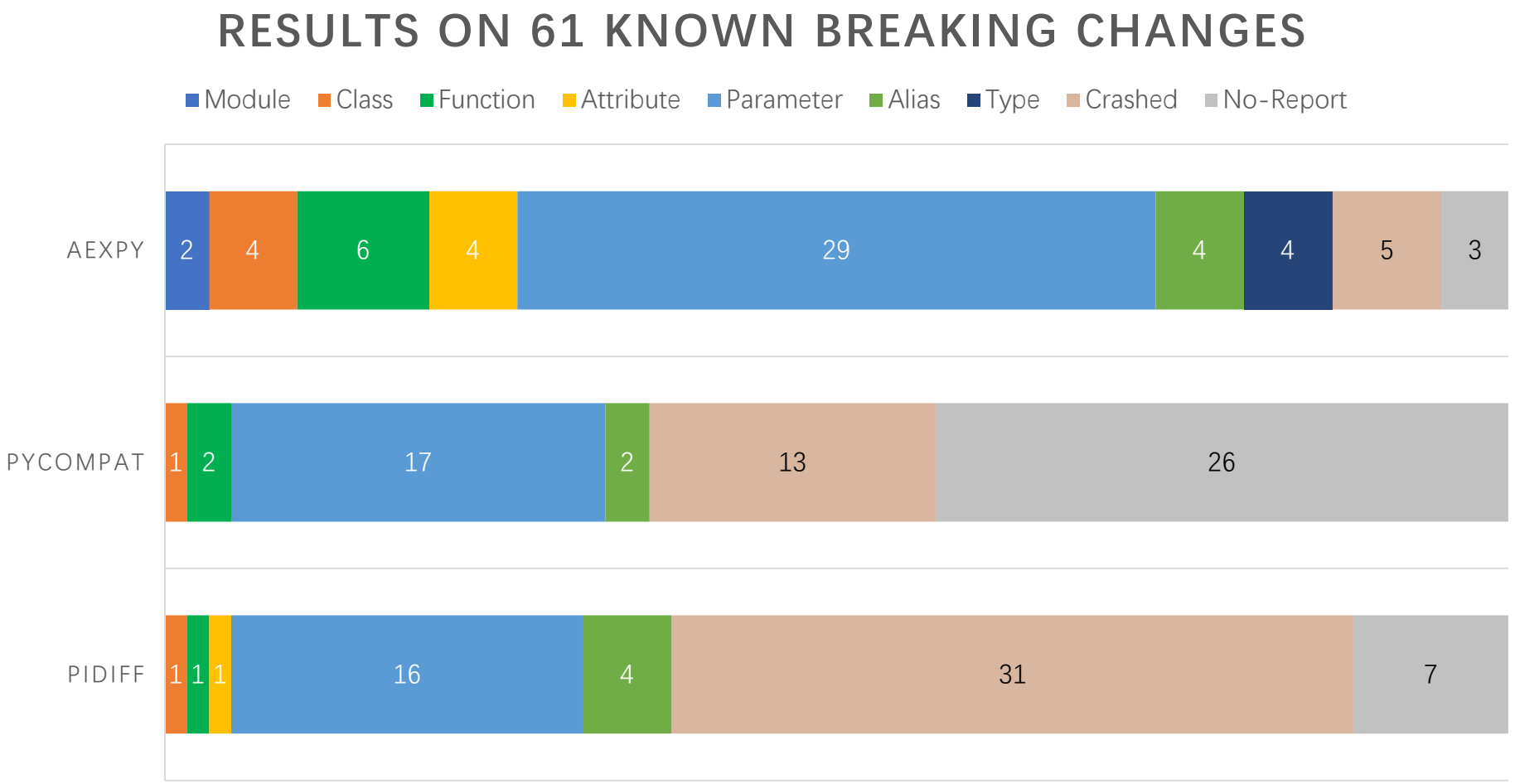
From resolved GitHub **issues** and 10 popular packages' **changelogs**

An API misuse detector with its own **change detector**

A **semantic versioning checker** for Python packages

Evaluation

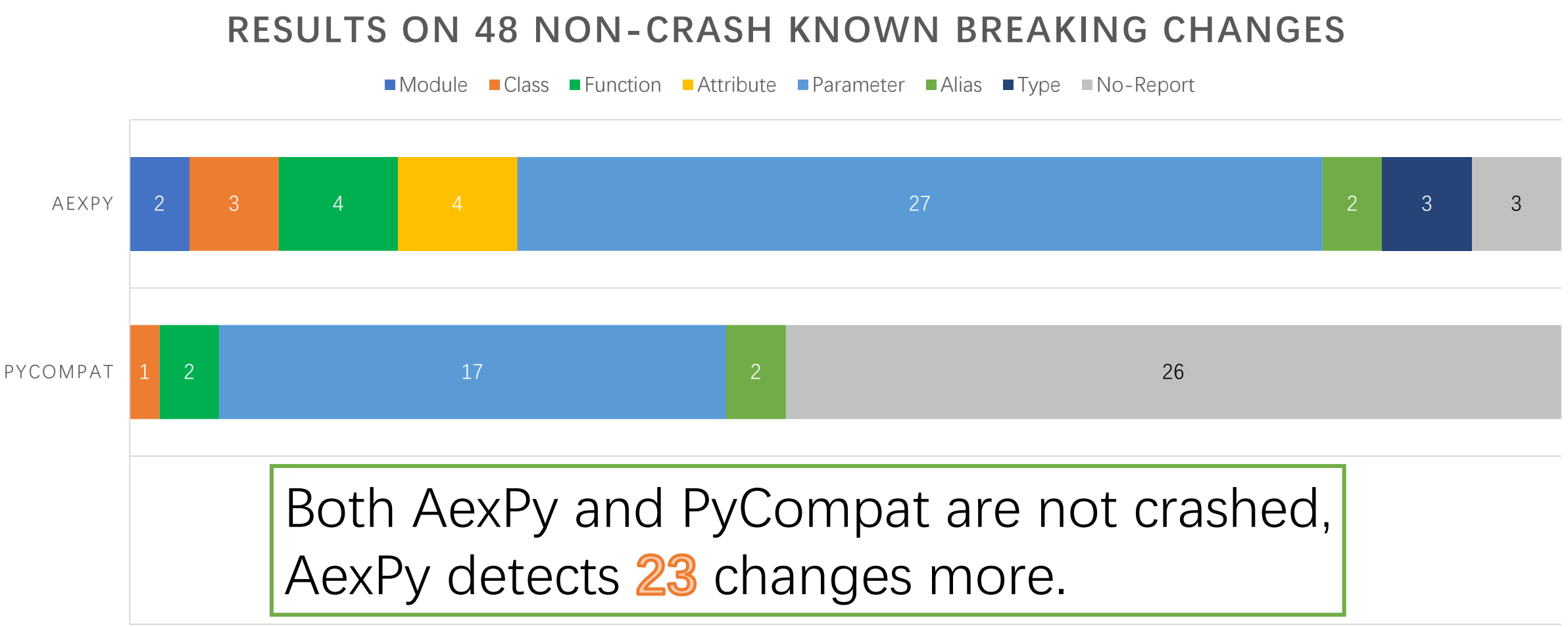
Does AexPy detect more known breaking changes than existing approaches?



AexPy detects **53** changes, increasing by ~**50%** recall.

Evaluation

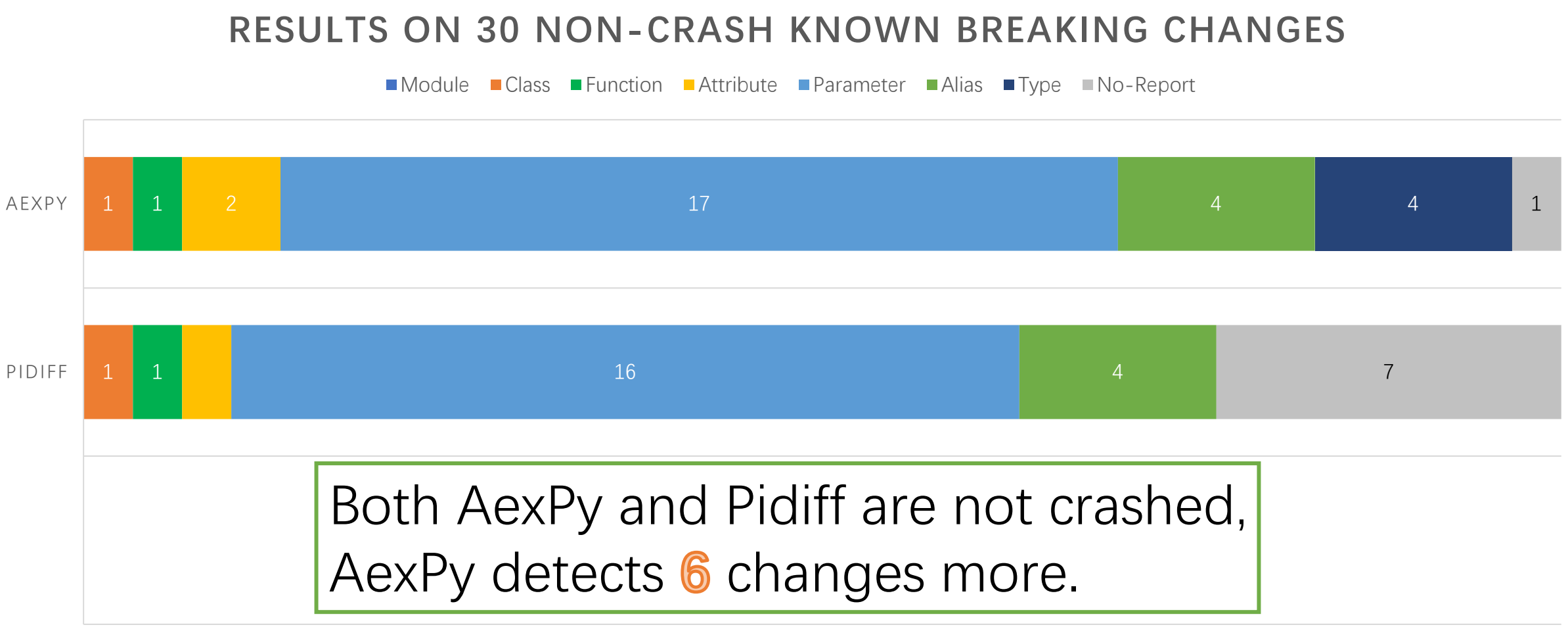
Does AexPy detect more known breaking changes than existing approaches?



Both AexPy and PyCompat are not crashed, AexPy detects 23 changes more.

Evaluation

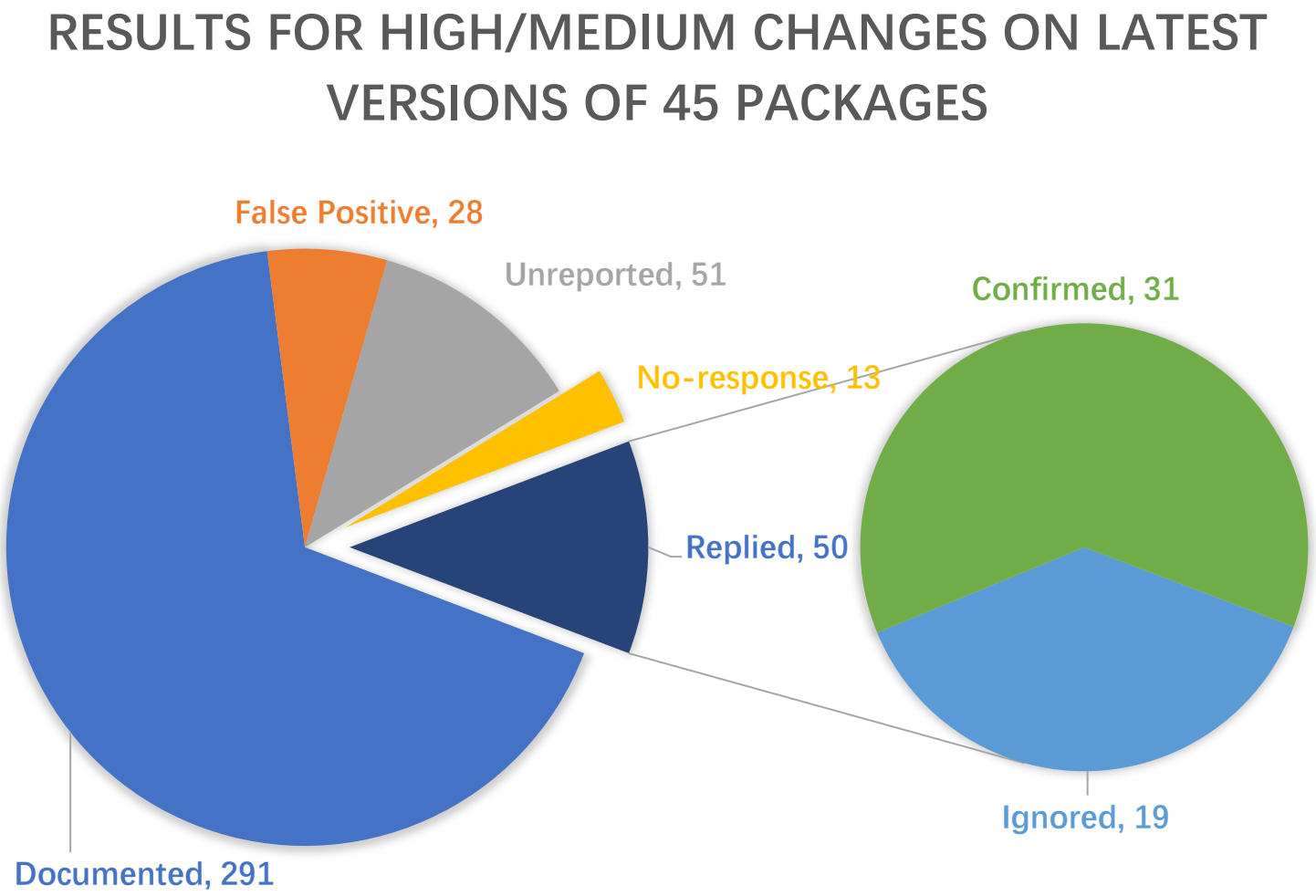
Does AexPy detect more known breaking changes than existing approaches?



Both AexPy and Pidiff are not crashed, AexPy detects 6 changes more.

Evaluation

Can AexPy find potential unknown breaking changes?



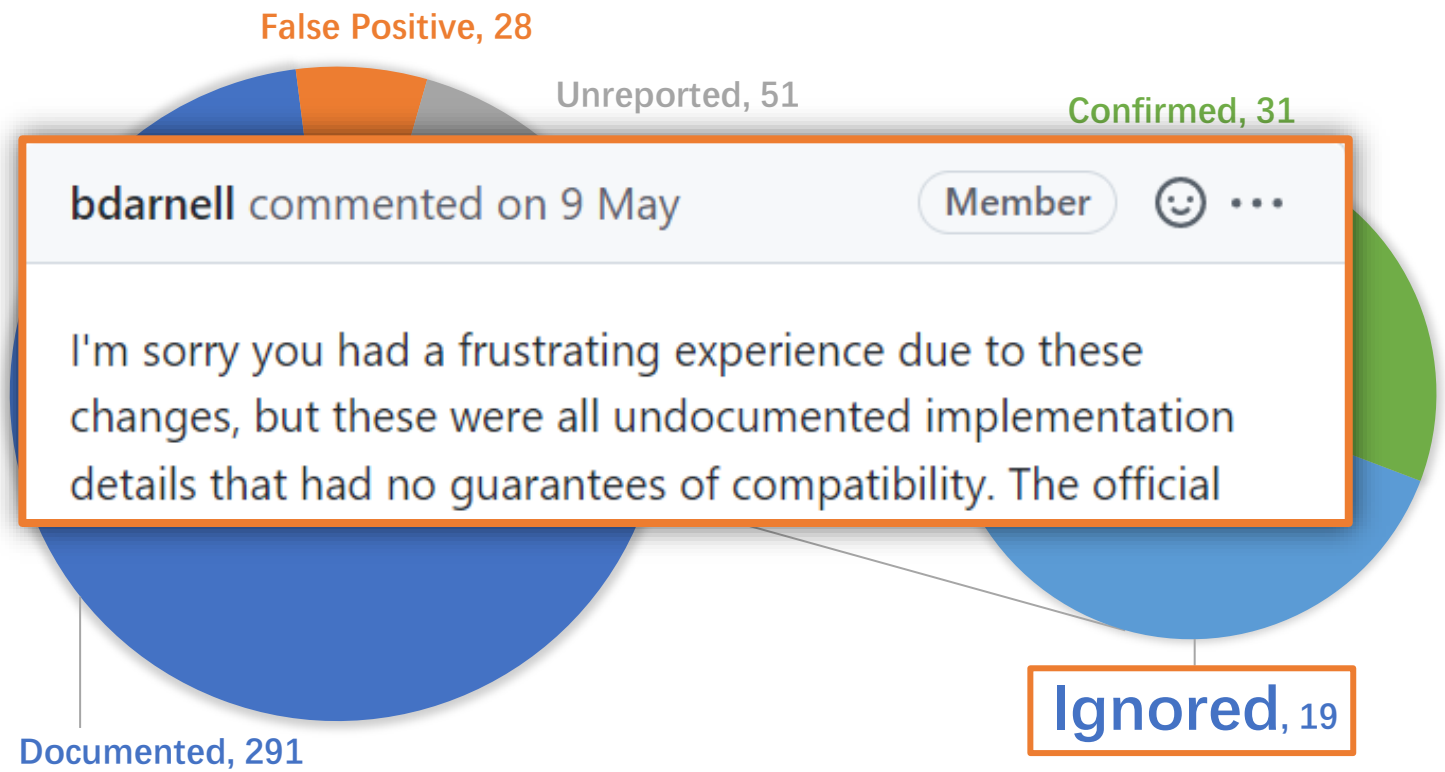
AexPy detects 433 high/medium changes, **405** are true.

Among 63 reported potential changes, **31** are confirmed by developers.

Evaluation

Can AexPy find potential unknown breaking changes?

RESULTS FOR HIGH/MEDIUM CHANGES ON LATEST VERSIONS OF 45 PACKAGES

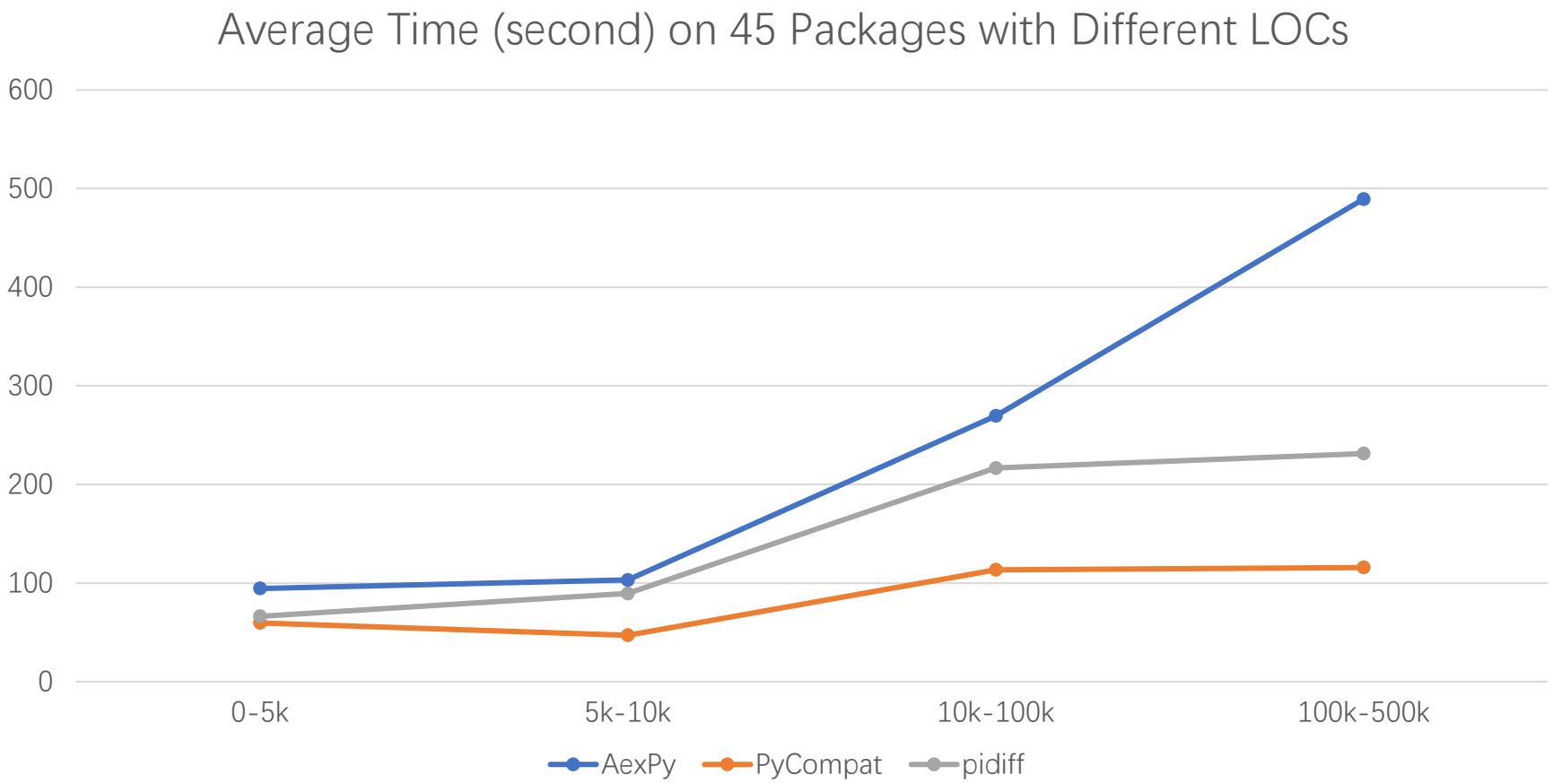


AexPy detects 433 high/medium changes, **405** are true.

Among 63 reported potential changes, **31** are confirmed by developers.

Evaluation

What is the time performance of AexPy?



Experiment Environment
Containers on a Ubuntu 18.04, 12 CPUs of 3.8GHz, 64 GBs RAM
Limited in 50GBs RAM, 1 hour, for each version pair

The time performances of the three tools are **in the same order of magnitude**.

Contributions

AexPy: Detecting API Breaking Changes in Python Packages

An API breaking change detection approach for Python packages

- High recall and strong robustness
- Detecting potential breaking changes
- Comparable time performance

Key ideas

- Detailed model for APIs, changes, and breaking levels
- Hybrid analysis to enhance API extraction
- Constraint-based method to detect and grade API changes

Future works

- Study more applications of the built API and change knowledge base
- Consider more aspects of breaking changes including API semantics

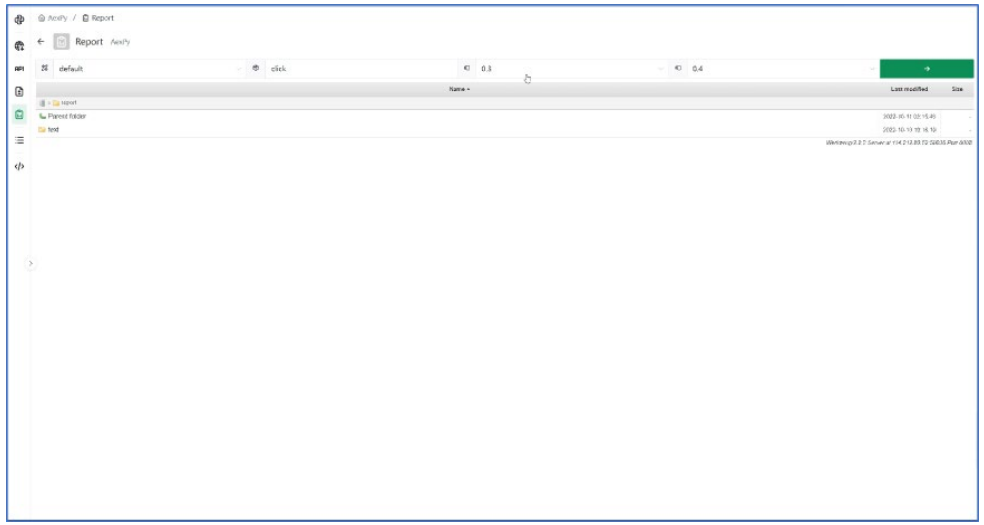
Contributions

AexPy: Detecting API Breaking Changes in Python Packages

Xingliang Du
xingliangdu@smail.nju.edu.cn

Jun Ma
majun@nju.edu.cn

https://github.com/**StardustDL/aexpy**

https://pypi.org/project/**aexpy**/

https://hub.docker.com/r/**stardustdl/aexpy**