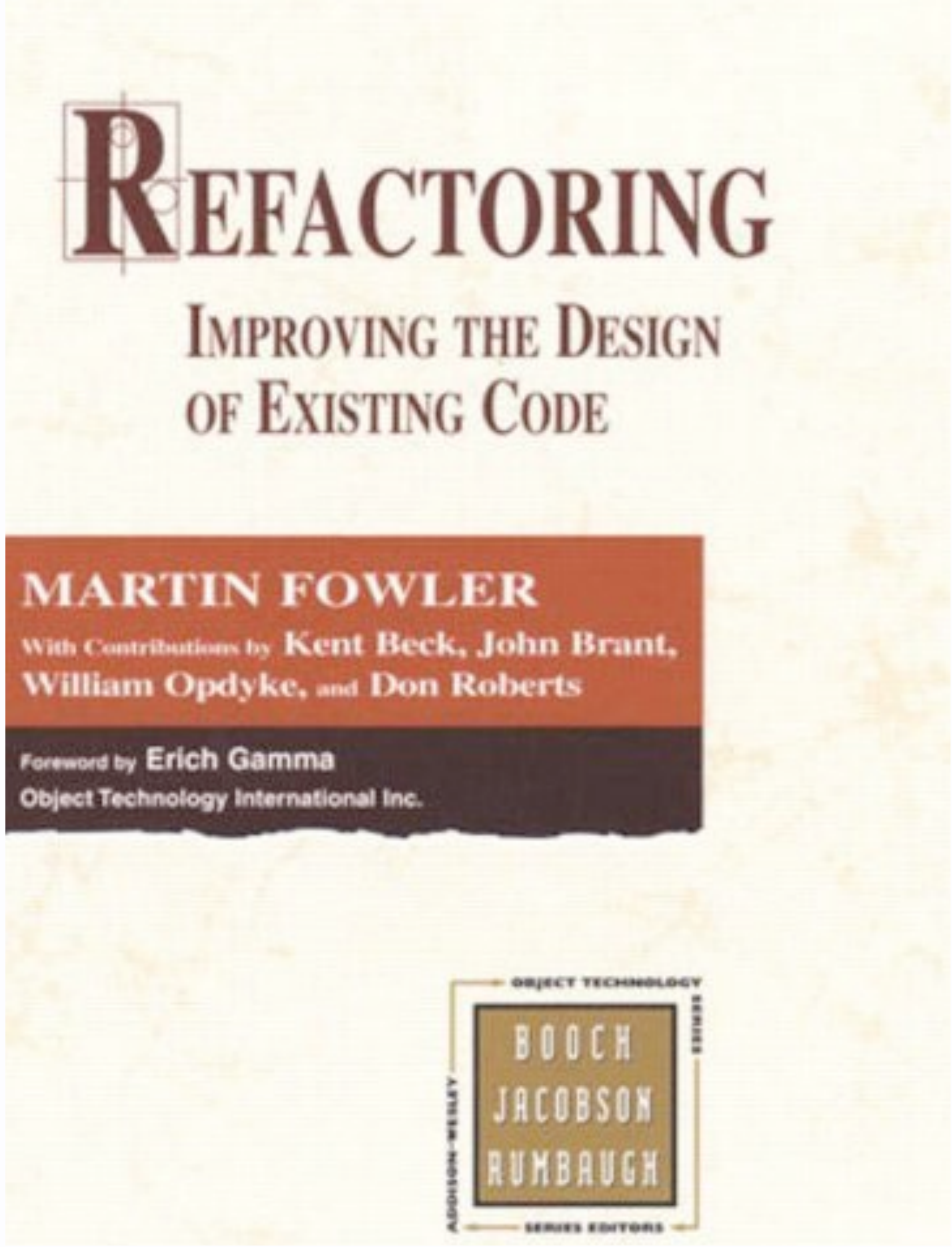
Clone Detection in Python

Valerio Maggio (valerio.maggio@unina.it)

Duplicated Code

*Number one in the stink parade is **duplicated code**.*

If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.



The Python Way

```
In [1]: import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

The Python Way

```
In [1]: import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
```

```
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
```

```
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

NLTK (tree.py)

```
class ImmutableProbabilisticTree(ImmutableTree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ImmutableProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        ImmutableTree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s [%s]' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s [%s]' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1)
        else:
            return val
    convert = classmethod(convert)
```

```
class ProbabilisticTree(Tree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        Tree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s (p=%s)' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s (p=%s)' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1.0)
        else:
            return val
    convert = classmethod(convert)
```

NLTK (tree.py)

```python
class ImmutableProbabilisticTree(ImmutableTree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ImmutableProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        ImmutableTree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s [%s]' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s [%s]' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1)
        else:
            return val
    convert = classmethod(convert)
```

```python
class ProbabilisticTree(Tree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        Tree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s (p=%s)' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s (p=%s)' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1.0)
        else:
            return val
    convert = classmethod(convert)
```

NLTK (tree.py)

```python
class ImmutableProbabilisticTree(ImmutableTree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ImmutableProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        ImmutableTree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s [%s]' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s [%s]' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1)
        else:
            return val
    convert = classmethod(convert)
```

```python
class ProbabilisticTree(Tree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        Tree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s (p=%s)' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s (p=%s)' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1.0)
        else:
            return val
    convert = classmethod(convert)
```

NLTK (tree.py)

```python
class ImmutableProbabilisticTree(ImmutableTree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ImmutableProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        ImmutableTree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s [%s]' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s [%s]' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1)
        else:
            return val
    convert = classmethod(convert)
```

```python
class ProbabilisticTree(Tree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        Tree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s (p=%s)' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s (p=%s)' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1.0)
        else:
            return val
    convert = classmethod(convert)
```

NLTK (tree.py)

```python
class ImmutableProbabilisticTree(ImmutableTree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ImmutableProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        ImmutableTree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s [%s]' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s [%s]' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1)
        else:
            return val
    convert = classmethod(convert)
```

```python
class ProbabilisticTree(Tree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        Tree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s (p=%s)' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s (p=%s)' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1.0)
        else:
            return val
    convert = classmethod(convert)
```

NLTK (tree.py)

```python
class ImmutableProbabilisticTree(ImmutableTree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ImmutableProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        ImmutableTree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s [%s]' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s [%s]' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1)
        else:
            return val
    convert = classmethod(convert)
```

```python
class ProbabilisticTree(Tree, ProbabilisticMixIn):
    def __new__(cls, node_or_str, children=None, **prob_kwargs):
        return super(ProbabilisticTree, cls).__new__(
            cls, node_or_str, children)
    def __init__(self, node_or_str, children=None, **prob_kwargs):
        if children is None: return # see note in Tree.__init__()
        Tree.__init__(self, node_or_str, children)
        ProbabilisticMixIn.__init__(self, **prob_kwargs)

    # We have to patch up these methods to make them work right:
    def _frozen_class(self): return ImmutableProbabilisticTree
    def __repr__(self):
        return '%s (p=%s)' % (Tree.__repr__(self), self.prob())
    def __str__(self):
        return '%s (p=%s)' % (self.pprint(margin=60), self.prob())
    def __cmp__(self, other):
        c = Tree.__cmp__(self, other)
        if c != 0: return c
        return cmp(self.prob(), other.prob())
    def __eq__(self, other):
        if not isinstance(other, Tree): return False
        return Tree.__eq__(self, other) and self.prob()==other.prob()
    def __ne__(self, other):
        return not (self == other)
    def copy(self, deep=False):
        if not deep: return self.__class__(self.node, self, prob=self.prob())
        else: return self.__class__.convert(self)
    def convert(cls, val):
        if isinstance(val, Tree):
            children = [cls.convert(child) for child in val]
            if isinstance(val, ProbabilisticMixIn):
                return cls(val.node, children, prob=val.prob())
            else:
                return cls(val.node, children, prob=1.0)
        else:
            return val
    convert = classmethod(convert)
```

Duplicated Code

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system

Duplicated Code

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system
- Is often created during development

Duplicated Code

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system
- Is often created during development
  - due to time pressure for an upcoming deadline

Duplicated Code

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system
- Is often created during development
  - due to time pressure for an upcoming deadline
  - to overcome limitations of the programming language

Duplicated Code

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system
- Is often created during development
  - due to time pressure for an upcoming deadline
  - to overcome limitations of the programming language
- Three Public Enemies:

Duplicated Code

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system

- Is often created during development
  - due to time pressure for an upcoming deadline
  - to overcome limitations of the programming language

- Three Public Enemies:
  - **Copy**, **Paste** and **Modify**

Clone Detection in Python

Clone Detection in Python

Code Clones

(Def.) **“***Software Clones are segments of code that are similar according to some definition of similarity***”** (I.D. Baxter, 1998)

- There can be different definitions of similarity, based on:
  - **Program Text** (text, syntax)
  - **Semantics**

Code Clones

(Def.) **"***Software Clones are segments of code that are similar according to some definition of similarity***"** (I.D. Baxter, 1998)

- There can be different definitions of similarity, based on:
  - **Program Text** (text, syntax)
  - **Semantics**
- **Four Different** Types of Clones

```python
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end---'):
    lines = list()
    with open(filepath) as report:
        for l in report:
            if l.endswith(marker):
                lines.append(l) # Stores only lines that ends with "marker"
    return lines #Return the list of different lines
```

The original one

Type 1: Exact Copy

- Identical code segments except for differences in layout, whitespace, and comments

Type 1: Exact Copy

- Identical code segments except for differences in layout, whitespace, and comments

```
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end---'):
    lines = list()
    with open(filepath) as report:
        for l in report:
            if l.endswith(marker):
                lines.append(l) # Stores only lines that ends with "marker"
    return lines #Return the list of different lines
```

```
def do_something_cool_in_Python (filepath, marker='---end---'):
    lines = list() # This list is initially empty

     with open(filepath) as report:
        for l in report: # It goes through the lines of the file
            if l.endswith(marker):
                lines.append(l)
    return lines
```

Type 2: Parameter Substituted Clones

- Structurally identical segments except for differences in identifiers, literals, layout, whitespace, and comments

Type 2: Parameter Substituted Clones

- Structurally identical segments except for differences in identifiers, literals, layout, whitespace, and comments

```python
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end---'):
    lines = list()
    with open(filepath) as report:
        for l in report:
            if l.endswith(marker):
                lines.append(l) # Stores only lines that ends with "marker"
    return lines #Return the list of different lines
```

```python
# Type 2 Clone
def do_something_cool_in_Python(path, end='---end---'):
    targets = list()
    with open(path) as data_file:
        for t in data_file:
            if l.endswith(end):
                targets.append(t) # Stores only lines that ends with "marker"
    #Return the list of different lines
    return targets
```

Type 3: Structure Substituted Clones

- Similar segments with further modifications such as changed, added (or deleted) statements, in additions to variations in identifiers, literals, layout and comments

Type 3: Structure Substituted Clones

- Similar segments with further modifications such as changed, added (or deleted) statements, in additions to variations in identifiers, literals, layout and comments

```
import os
def do_something_with(path, marker='---end---'):
    # Check if the input path corresponds to a file
    if not os.path.isfile(path):
        return None

    bad_ones = list()
    good_ones = list()
    with open(path) as report:
        for line in report:
            line = line.strip()
            if line.endswith(marker):
                good_ones.append(line)
            else:
                bad_ones.append(line)
     #Return the lists of different lines
    return good_ones, bad_ones
```

Type 3: Structure Substituted Clones

- Similar segments with further modifications such as changed, added (or deleted) statements, in additions to variations in identifiers, literals, layout and comments

```
import os
def do_something_with(path, marker='---end---'):
   # Check if the input path corresponds to a file
   if not os.path.isfile(path):
      return None

   bad_ones = list()
   good_ones = list()
   with open(path) as report:
      for line in report:
         line = line.strip()
         if line.endswith(marker):
            good_ones.append(line)
         else:
            bad_ones.append(line)
    #Return the lists of different lines
   return good_ones, bad_ones
```

Type 3: Structure Substituted Clones

- Similar segments with further modifications such as changed, added (or deleted) statements, in additions to variations in identifiers, literals, layout and comments

```
import os
def do_something_with(path, marker='---end---'):
    # Check if the input path corresponds to a file
    if not os.path.isfile(path):
        return None

    bad_ones = list()
    good_ones = list()
    with open(path) as report:
        for line in report:
            line = line.strip()
            if line.endswith(marker):
                good_ones.append(line)
            else:
                bad_ones.append(line)
     #Return the lists of different lines
    return good_ones, bad_ones
```

Type 3: Structure Substituted Clones

- Similar segments with further modifications such as changed, added (or deleted) statements, in additions to variations in identifiers, literals, layout and comments

```python
import os
def do_something_with(path, marker='---end---'):
    # Check if the input path corresponds to a file
    if not os.path.isfile(path):
        return None

    bad_ones = list()
    good_ones = list()
    with open(path) as report:
        for line in report:
            line = line.strip()
            if line.endswith(marker):
                good_ones.append(line)
            else:
                bad_ones.append(line)
     #Return the lists of different lines
    return good_ones, bad_ones
```

Type 4: “Semantic” Clones

- Semantically equivalent segments that perform the same computation but are implemented by different syntactic variants

Type 4: "Semantic" Clones

- Semantically equivalent segments that perform the same computation but are implemented by different syntactic variants

```python
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end---'):
    lines = list()
    with open(filepath) as report:
        for l in report:
            if l.endswith(marker):
                lines.append(l) # Stores only lines that ends with "marker"
    return lines #Return the list of different lines
```

```python
def do_always_the_same_stuff(filepath, marker='---end---'):
    report = open(filepath)
    file_lines = report.readlines()
    report.close()
    #Filters only the lines ending with marker
    return filter(lambda l: len(l) and l.endswith(marker), file_lines)
```

What are the consequences?

- Do clones increase the maintenance effort?
- *Hypothesis*:
  - Cloned code increases code size
  - A fix to a clone must be applied to all similar fragments
  - Bugs are duplicated together with their clones

- However: it is not always possible to <u>remove</u> clones
  - Removal of Clones is harder if variations exist.

Clone Detection Tools

Duploc

SDD

Dup

CPD

NiCAD

Duplix

Gemini

Dude

Simian

CCFinder

Shinobi

CLICS

Clone Detective

iClones

Scorpio

ConQAT

Clone Digger

Deckard

Duplix

PMD

JCCD

CloneDr

KClone

SimScan

Clone Detection Tools

Duploc

SDD

NiCAD

Dude

Simian

CLICS

Dup

CPD

- **Text Based Tools**:
  - Lines are compared to other lines

Scorpio

ConQAT

Clone Digger

Duplix

PMD

Deckard

JCCD

CloneDr

KClone

SimScan

Clone Detection Tools

Duploc

SDD

NiCAD

Dude

Simian

CLICS

Scorpio

Duplix

PMD

Dup

CPD

Duplix

Gemini

CCFinder

Shinobi

Clone Detective

iClones

- **Token Based Tools**:
  - Token sequences are compared to sequences

Clone Detection Tools

Duploc

SDD

NiCAD

Dude

Simian

CLICS

Dup

CPD

Duplix

Gemini

CCFinder

Shinobi

Clone Detective

iClones

onQAT

Clone Digger

Deckard

JCCD

oneDr

KClone

SimScan

- **Syntax Based Tools**:
  - Syntax subtrees are compared to each other

Clone Detection Tools

Duploc

SDD

NiCAD

Dude

Simian

CLICS

Dup

CPD

Duplix

Gemini

CCFinder

Shinobi

Clone Detective

iClones

Scorpio

Duplix

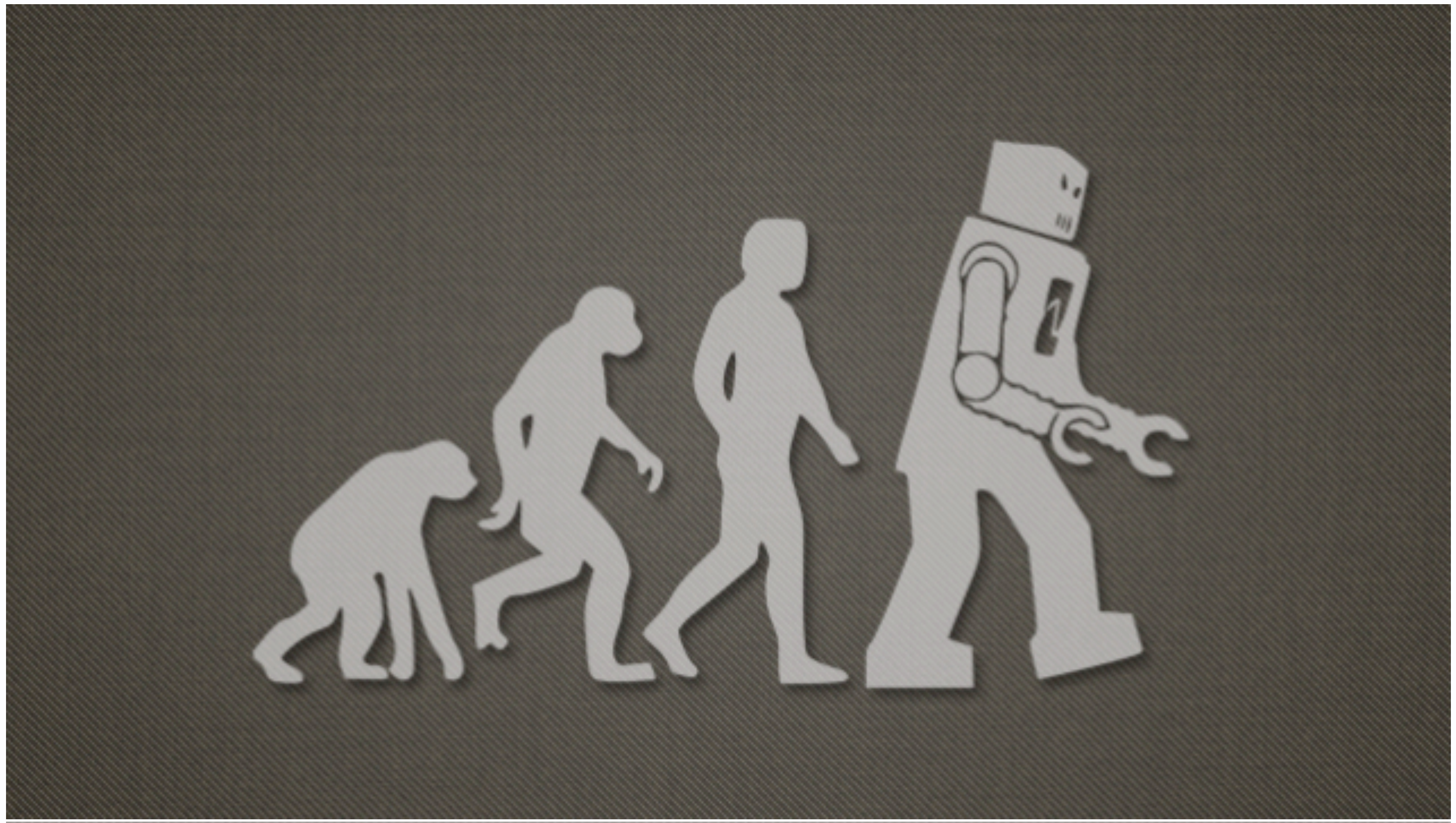
PMD

- **Graph Based Tools**:
  - (sub) graphs are compared to each other

Clone Detection Techniques

- **String/Token** based Techiniques:
  - Pros: Run very fast
  - Cons: Too many false clones
- **Syntax** based (AST) Techniques:
  - Pros: Well suited to detect structural similarities
  - Cons: Not Properly suited to detect **Type 3 Clones**
- **Graph** based Techniques:
  - Pros: The only one able to deal with **Type 4 Clones**
  - Cons: Performance Issues

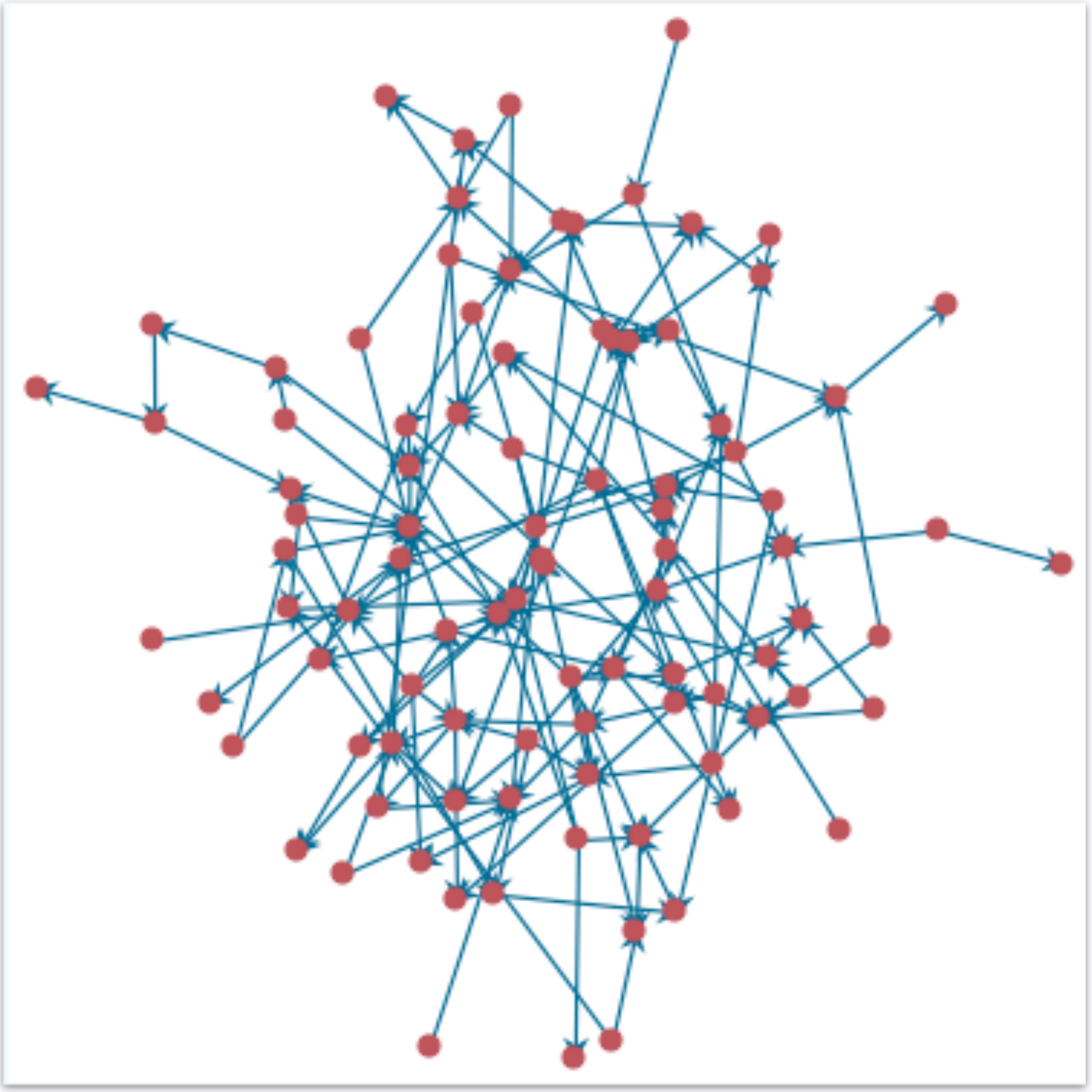
The idea: Use Machine Learning, Luke



- Use **Machine Learning** Techniques to compute similarity of fragments by exploiting specific *features* of the code.
- Combine different sources of Information
  - Structural Information: **ASTs**, **PDGs**
  - Lexical Information: **Program Text**

Kernel Methods for Structured Data

- Well-grounded on solid and awful Math
- Based on the idea that objects can be described in terms of their constituent Parts
- Can be easily tailored to specific domains
  - **Tree Kernels**
  - **Graph Kernels**
  - ....



Defining a Kernel for Structured Data

Defining a Kernel for Structured Data

The definition of a new Kernel for a **Structured Object** requires the definition of:

Defining a Kernel for Structured Data

The definition of a new Kernel for a **Structured Object** requires the definition of:

- Set of **features** to annotate each part of the object

Defining a Kernel for Structured Data

The definition of a new Kernel for a **Structured Object** requires the definition of:

- Set of **features** to annotate each part of the object

- A **Kernel function** to measure the similarity on the smallest part of the object

Defining a Kernel for Structured Data

The definition of a new Kernel for a **Structured Object** requires the definition of:

- Set of **features** to annotate each part of the object
- A **Kernel function** to measure the similarity on the smallest part of the object
  - e.g., Nodes for AST and Graphs

Defining a Kernel for Structured Data

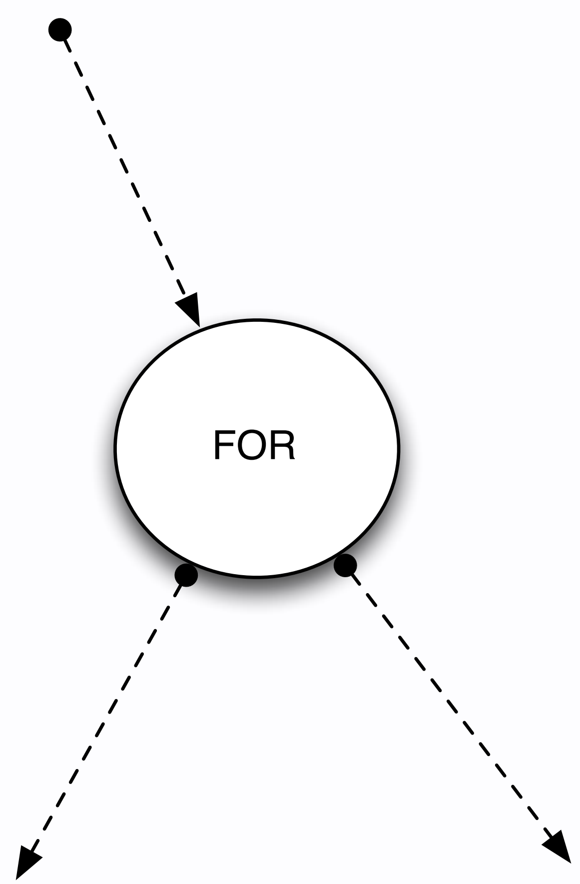
The definition of a new Kernel for a **Structured Object** requires the definition of:

- Set of **features** to annotate each part of the object

- A **Kernel function** to measure the similarity on the smallest part of the object
  - e.g., Nodes for AST and Graphs

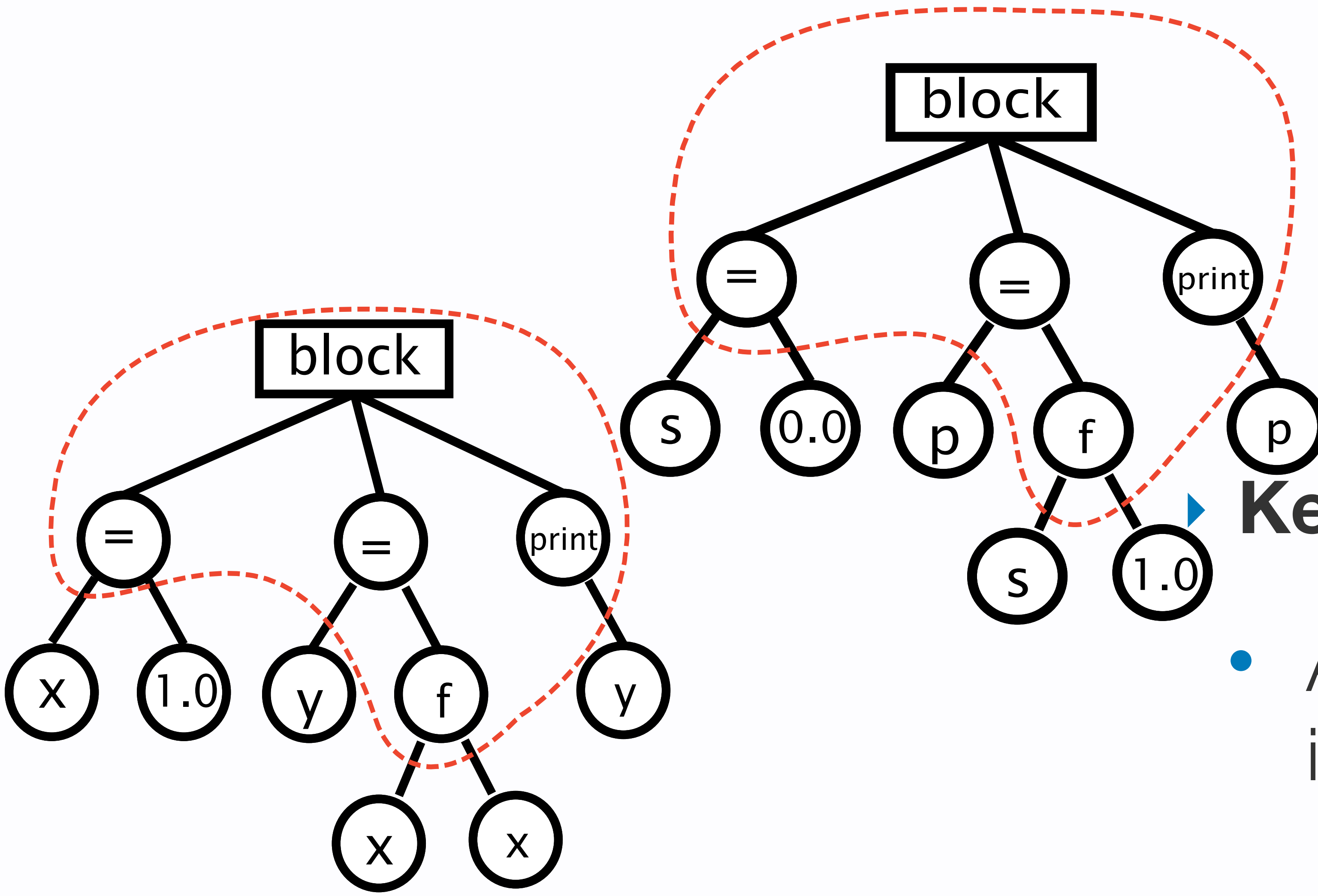
- A **Kernel function** to apply the computation on the different (sub)parts of the structured object

Kernel Methods for Clones: Tree Kernels Example on AST



- **Features**: We annotate each node by a set of 4 *features*
  - **Instruction Class**
    - i.e., LOOP, CONDITIONAL_STATEMENT, CALL
  - **Instruction**
    - i.e., FOR, IF, WHILE, RETURN
  - **Context**
    - i.e. Instruction Class of the closer statement node
  - **Lexemes**
    - Lexical information gathered (recursively) from leaves
    - i.e., Lexical Information

Kernel Methods for Clones: Tree Kernels Example on AST



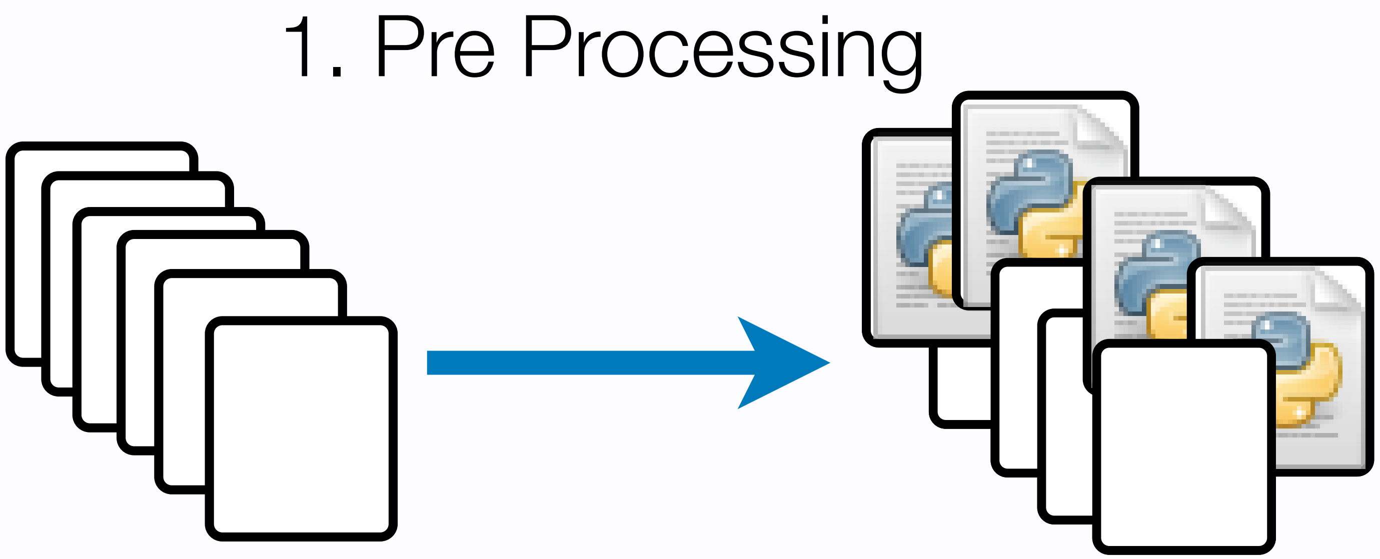
- **Kernel Function**:
  - Aims at identify the maximum isomorphic Tree/Subtree

$$K(T_1, T_2) = \sum_{n \in T_1} \sum_{n' \in T_2} \sigma(n, n') \cdot K_{subt}(n, n')$$

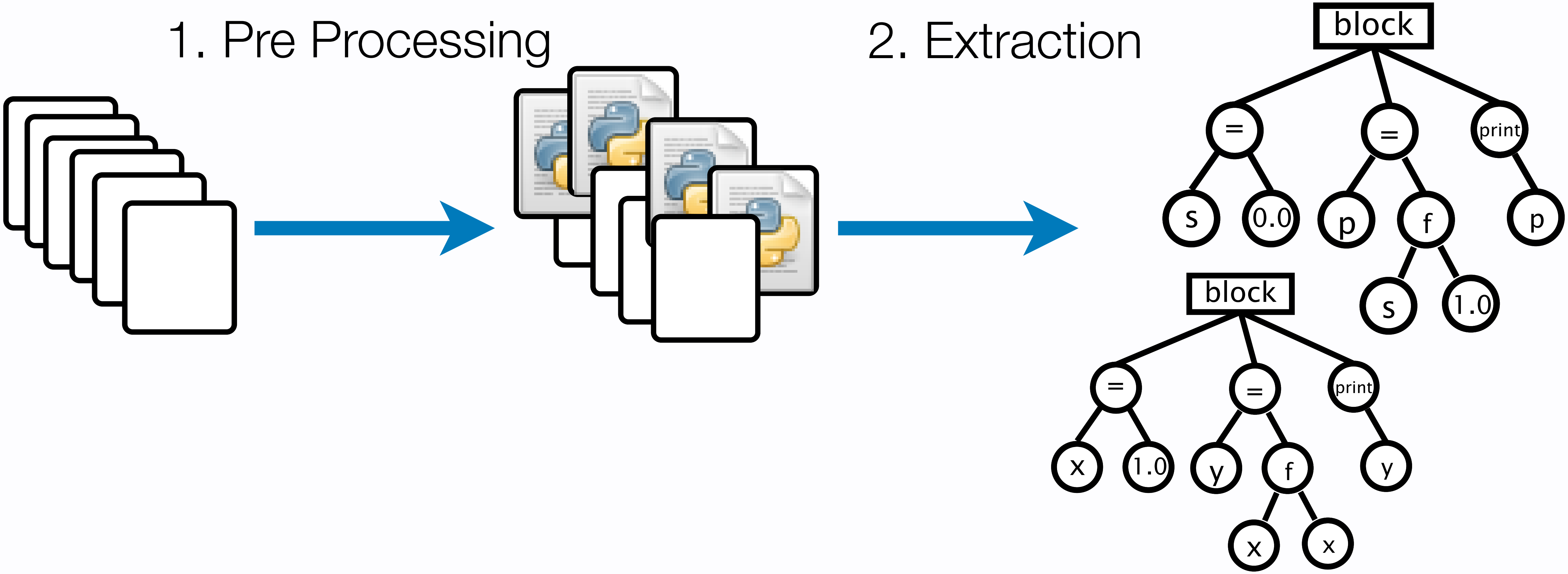
$$K_{subt}(n, n') = \lambda sim(n, n') + (1 - \lambda) \sum_{(n_1, n_2) \in Ch(n, n')} k(n_1, n_2)$$

Clone Detection in Python

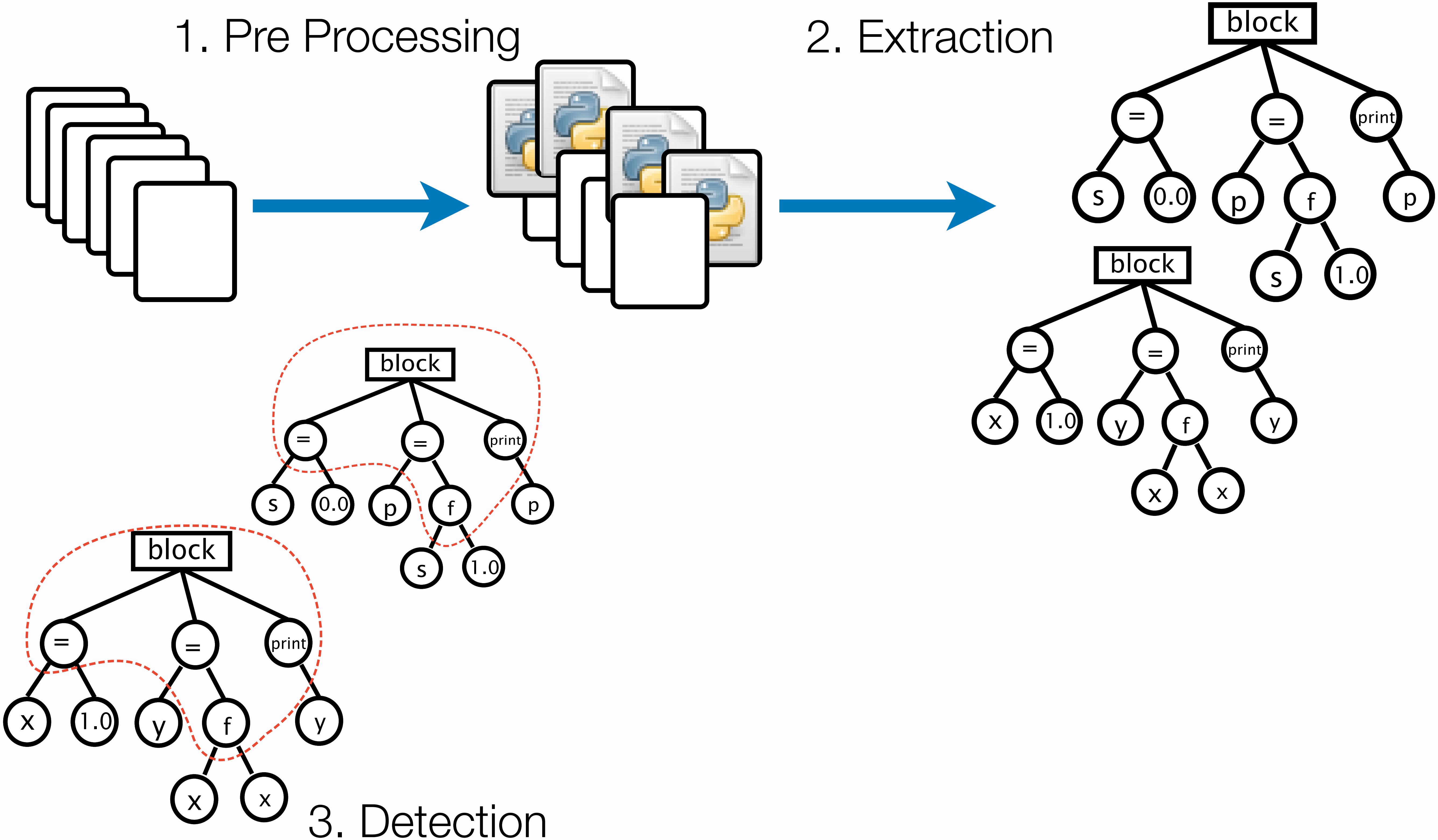
Clone Detection in Python



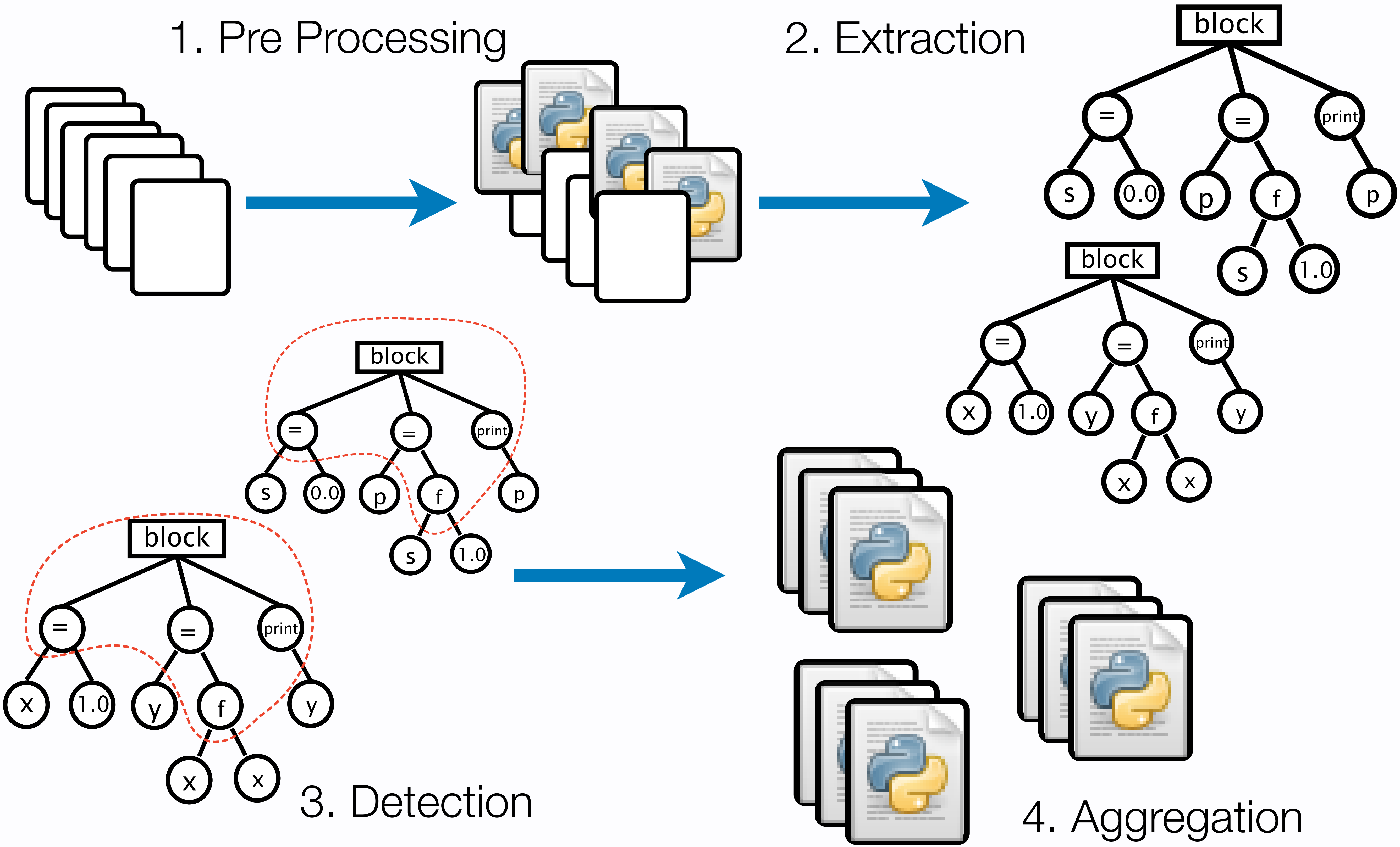
The Overall Process Sketch



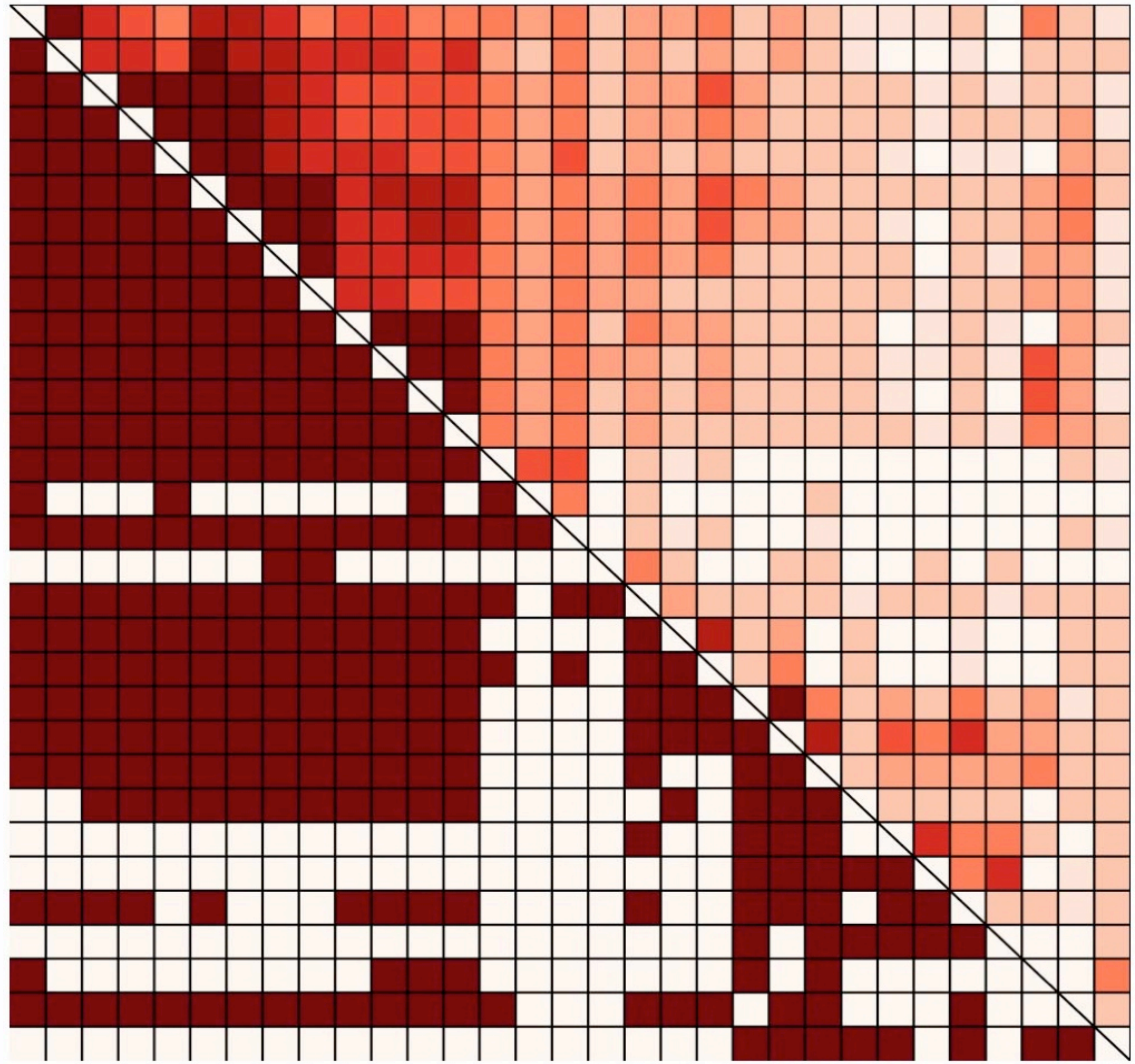
The Overall Process Sketch



The Overall Process Sketch



The Overall Process Sketch

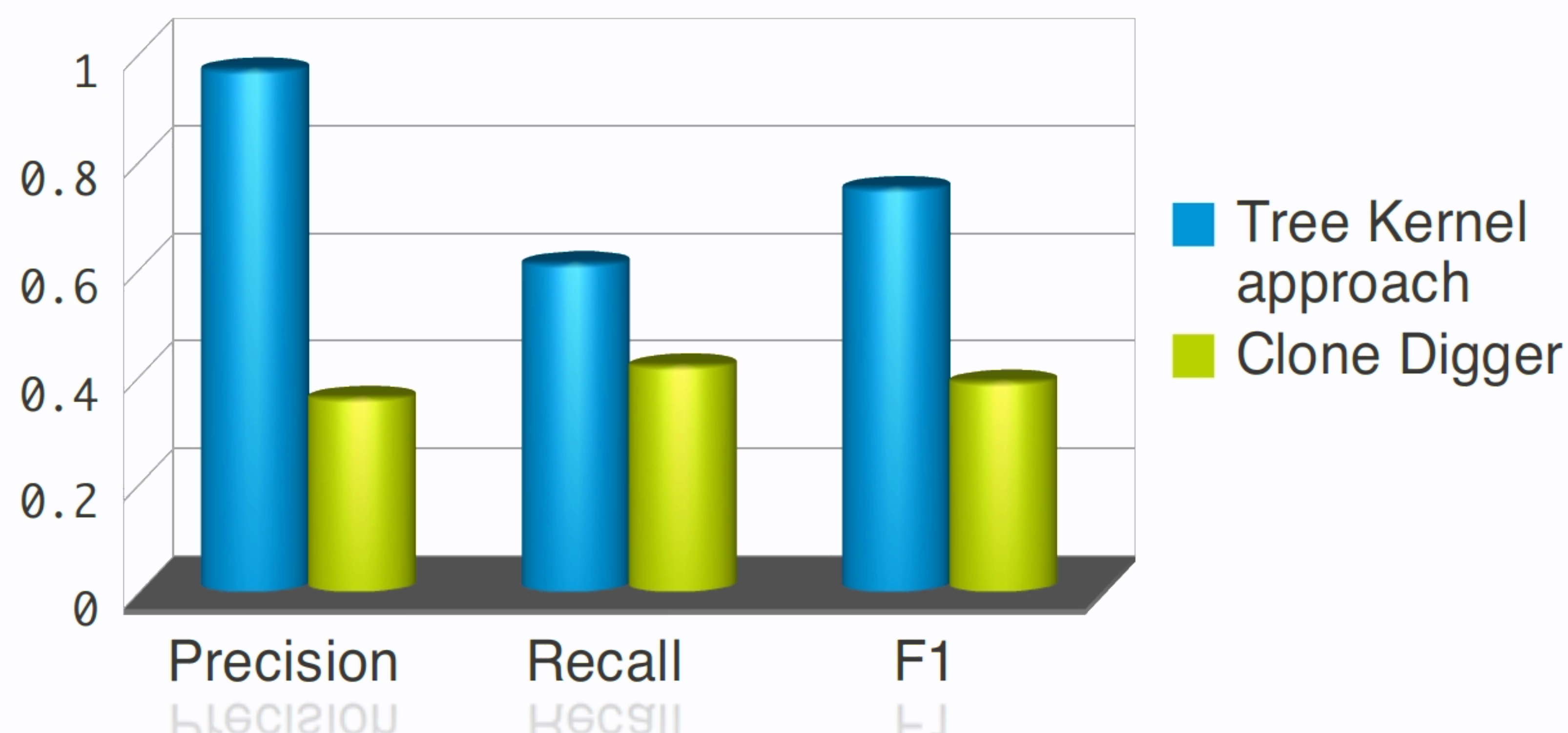


Detection Process

Empirical Evaluation

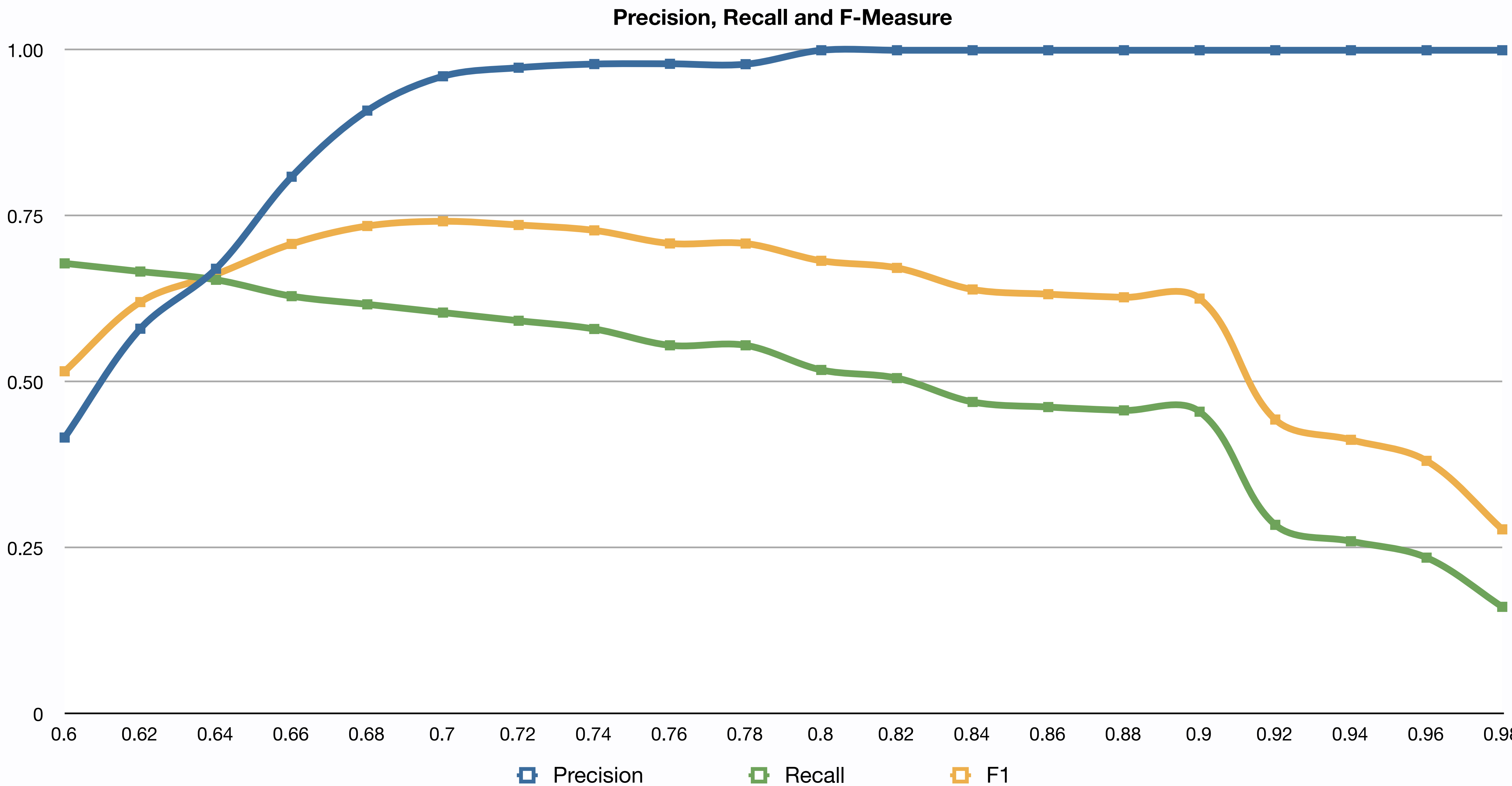
- Comparison with another (pure) AST-based: **Clone Digger**
  - It has been the first Clone detector for and in Python :-)
  - Presented at EuroPython 2006
- Comparison on a system with randomly seeded clones

- Results refer only to **Type 3 Clones**
- On **Type 1** and **Type 2** we got the same results

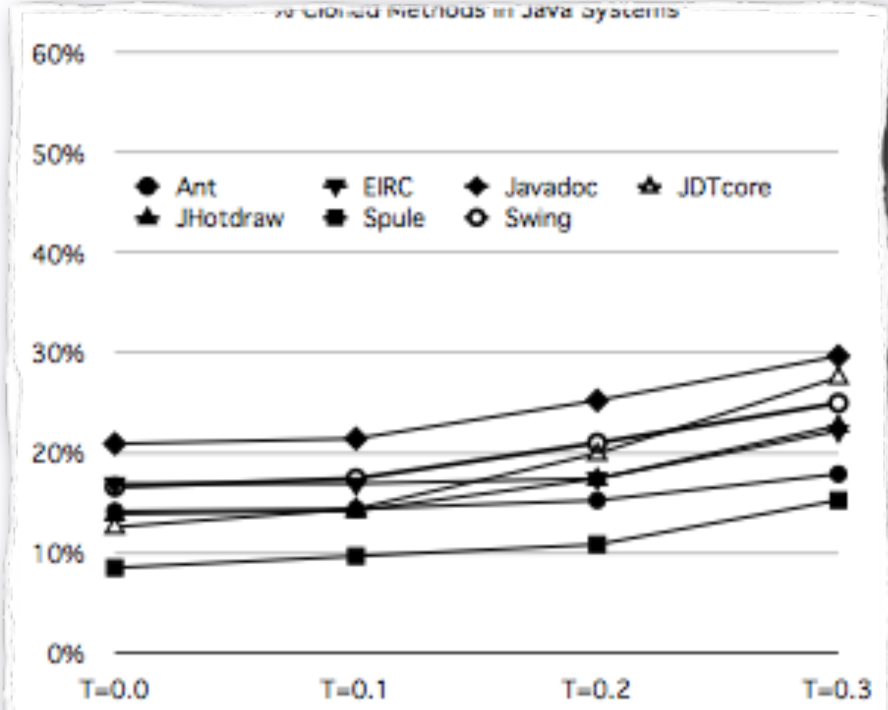


Precision: How **accurate** are the obtained results?
(Altern.) How many errors do they contain?

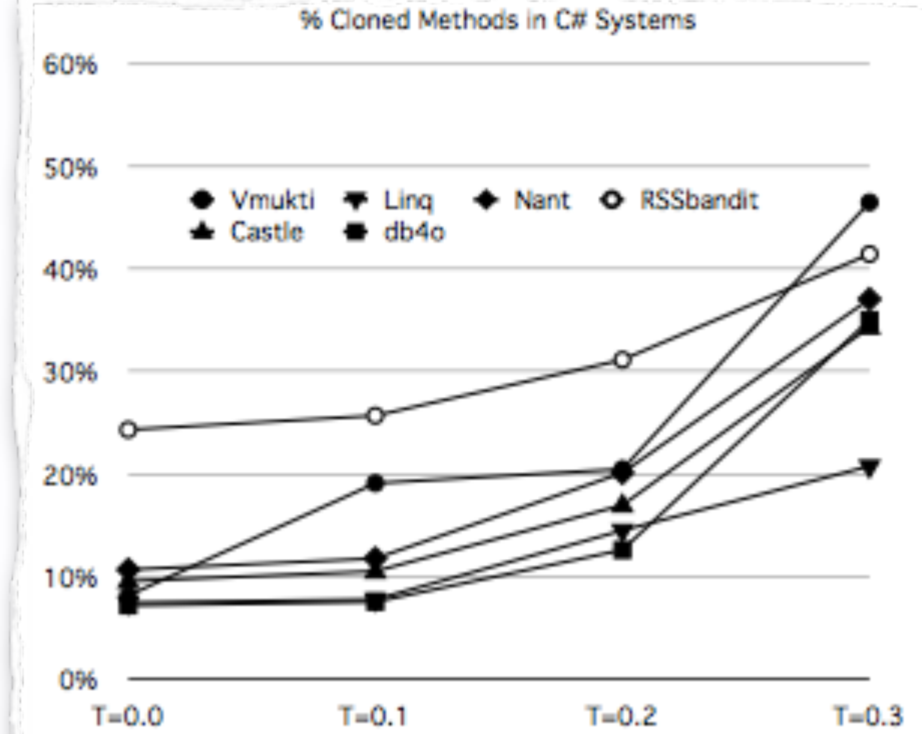
Recall: How **complete** are the obtained results?
(Altern.) How many clones have been retrieved w.r.t. Total Clones?



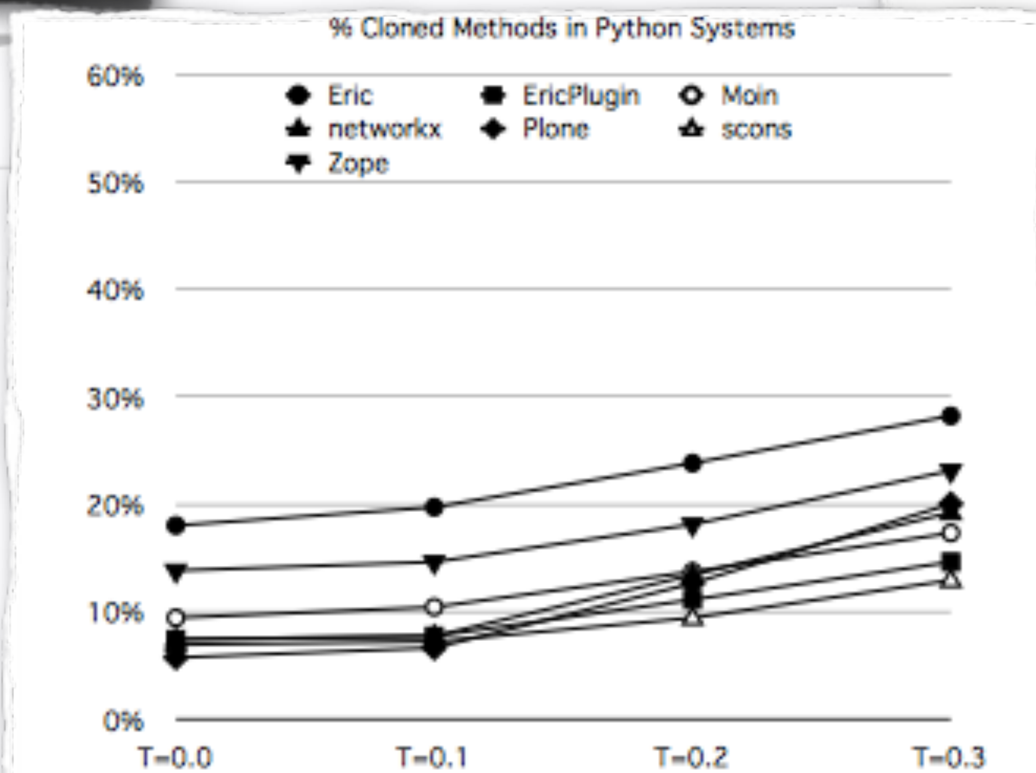
Precision/Recall Plot



(d) Java Systems



(e) C# Systems



(b) Python Systems

Roy et. al., IWSC, 2010

Is Python less *clone prone*?

Clones in CPython 2.5.1

```c
atic PyObject *
etime_new(PyTypeObject *type, PyObject *args, PyObject *kw)

	PyObject *self = NULL;
	PyObject *state;
	int year;
	int month;
	int day;
	int hour = 0;
	int minute = 0;
	int second = 0;
	int usecond = 0;
	PyObject *tzinfo = Py_None;

	/* Check for invocation from pickle with __getstate__ state */
	if (PyTuple_GET_SIZE(args) >= 1 &&
	    PyTuple_GET_SIZE(args) <= 2 &&
	    PyString_Check(state = PyTuple_GET_ITEM(args, 0)) &&
	    PyString_GET_SIZE(state) == _PyDateTime_DATETIME_DATASIZE &&
	    MONTH_IS_SANE(PyString_AS_STRING(state)[2]))
	{
		PyDateTime_DateTime *me;
		char aware;

		if (PyTuple_GET_SIZE(args) == 2) {
			tzinfo = PyTuple_GET_ITEM(args, 1);
			if (check_tzinfo_subclass(tzinfo) < 0) {
				PyErr_SetString(PyExc_TypeError, "bad "
					"tzinfo state arg");
				return NULL;
			}
		}
		aware = (char)(tzinfo != Py_None);
		me = (PyDateTime_DateTime *) (type->tp_alloc(type , aware));
		if (me != NULL) {
			char *pdata = PyString_AS_STRING(state);

			memcpy(me->data, pdata, _PyDateTime_DATETIME_DATASIZE);
			me->hashcode = -1;
			me->hastzinfo = aware;
			if (aware) {
				Py_INCREF(tzinfo);
				me->tzinfo = tzinfo;
			}
```

```c
static PyObject *
time_new(PyTypeObject *type, PyObject *args, PyObject *kw)
{
	PyObject *self = NULL;
	PyObject *state;
	int hour = 0;
	int minute = 0;
	int second = 0;
	int usecond = 0;
	PyObject *tzinfo = Py_None;

	/* Check for invocation from pickle with __getstate__ state */
	if (PyTuple_GET_SIZE(args) >= 1 &&
	    PyTuple_GET_SIZE(args) <= 2 &&
	    PyString_Check(state = PyTuple_GET_ITEM(args, 0)) &&
	    PyString_GET_SIZE(state) == _PyDateTime_TIME_DATASIZE &&
	    ((unsigned char) (PyString_AS_STRING(state)[0])) < 24)
	{
		PyDateTime_Time *me;
		char aware;

		if (PyTuple_GET_SIZE(args) == 2) {
			tzinfo = PyTuple_GET_ITEM(args, 1);
			if (check_tzinfo_subclass(tzinfo) < 0) {
				PyErr_SetString(PyExc_TypeError, "bad "
					"tzinfo state arg");
				return NULL;
			}
		}
		aware = (char)(tzinfo != Py_None);
		me = (PyDateTime_Time *) (type->tp_alloc(type, aware));
		if (me != NULL) {
			char *pdata = PyString_AS_STRING(state);

			memcpy(me->data, pdata, _PyDateTime_TIME_DATASIZE);
			me->hashcode = -1;
			me->hastzinfo = aware;
			if (aware) {
				Py_INCREF(tzinfo);
				me->tzinfo = tzinfo;
			}
		}
		return (PyObject *)me;
	}
```

EUROPYTHON 2012
FLORENCE, JULY 2–8

Thank you!