LIFE IS BETTER PAINTED BLACK

ŁUKASZ LANGA

Łukasz Langa
ambv on GitHub

fb.me/ambv
@llanga
lukasz@langa.pl

Problem Statement

Syntax and Trees

Design and Implementation

Fun Problems

Problem Statement

Syntax and Trees

Design and Implementation

Fun Problems

CODE IS THE ENEMY

CODE STYLE


2017
2018
. . .

PEP 8

FROM STYLE GUIDE TO CODE STYLE

LINTERS ARE SO 2003

AUTO-FORMATTING IS THE NEW BLACK

UNIFORMITY TRUMPS PERFECTION

OPINIONATED FORMATTERS ON THE RISE

- `gofmt` for Go
- `Prettier` for JavaScript
- `Black` for Python

Problem Statement

Syntax and Trees

Design and Implementation

Fun Problems

PYTHON GRAMMAR

```
file_input: (NEWLINE | stmt)* ENDMARKER
```

PYTHON GRAMMAR

```
file_input: (NEWLINE | stmt)* ENDMARKER
stmt: simple_stmt | compound_stmt
```

PYTHON GRAMMAR

```
file_input: (NEWLINE | stmt)* ENDMARKER
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
```

PYTHON GRAMMAR

```
file_input: (NEWLINE | stmt)* ENDMARKER
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt |
             flow_stmt | import_stmt | global_stmt |
             nonlocal_stmt | assert_stmt)
```

PYTHON GRAMMAR

```
file_input: (NEWLINE | stmt)* ENDMARKER
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt |
             flow_stmt | import_stmt | global_stmt |
             nonlocal_stmt | assert_stmt)
```

PYTHON GRAMMAR

```
expr_stmt: testlist_star_expr (
  annassign | augassign (yield_expr|testlist) |
  ('=' (yield_expr|testlist_star_expr))*)
```

PYTHON GRAMMAR

```
expr_stmt: testlist_star_expr (
  annassign | augassign (yield_expr|testlist) |
  ('=' (yield_expr|testlist_star_expr))*)
annassign: ':' test ['=' test]
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' |
  '&=' | '|=' | '^=' | '<<=' | '>>=' | '**=' | '//=')
```

PYTHON GRAMMAR

```
expr_stmt: testlist_star_expr (
  annassign | augassign (yield_expr|testlist) |
  ('=' (yield_expr|testlist_star_expr))*)
annassign: ':' test ['=' test]
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' |
  '&=' | '|=' | '^=' | '<<=' | '>>=' | '**=' | '//=')
testlist_star_expr: (test|star_expr) (
  ',' (test|star_expr))* [',']
```

PYTHON GRAMMAR

```
expr_stmt: testlist_star_expr (
  annassign | augassign (yield_expr|testlist) |
  ('=' (yield_expr|testlist_star_expr))*)
annassign: ':' test ['=' test]
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' |
  '&=' | '|=' | '^=' | '<<=' | '>>=' | '**=' | '//=')
testlist_star_expr: (test|star_expr) (
  ',' (test|star_expr))* [',']
```

PYTHON GRAMMAR

```
test: or_test ['if' or_test 'else' test] | lambdef
```

PYTHON GRAMMAR

```
test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
```

PYTHON GRAMMAR

```
test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
```

PYTHON GRAMMAR

```
test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
```

PYTHON GRAMMAR

```
test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: (
   '<'|'>'|'=='|'>='|'<='|'!='|'in'|'not' 'in'|
   'is'|'is' 'not')
```

PYTHON GRAMMAR

```
test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: (
   '<'|'>'|'=='|'>='|'<='|'!='|'in'|'not' 'in'|
   'is'|'is' 'not')
```

PYTHON GRAMMAR

```
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'@'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom_expr ['**' factor]
```

PYTHON GRAMMAR

```
atom_expr: ['await'] atom trailer*
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [testlist_comp] ']' |
       '{' [dictorsetmaker] '}' |
       NAME | NUMBER | STRING+ |
       '...' | 'None' | 'True' | 'False')
```

HOW DOES PYTHON PARSE SOURCE FILES?

```
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```python
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```
Module
  body=
    Import
      names=
        alias
          asname=None
          name='ast'
        /alias
    /Import
    ImportFrom
      level=0
      module='simple_visitor'
      names=
        alias
          asname=None
          name='visit'
        /alias
    /ImportFrom
```

```
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```
FunctionDef
  args=arguments
    args=
      arg
        annotation='str'
        arg='path'
      /arg
    kwarg=None
    vararg=None
  /arguments
  body=
    With
      body=
        Assign
          targets=
            'code_str'
          value=Call
            func=Attribute
              attr='read'
              value='this_file'
            /Attribute
          /Call
        /Assign
```

```python
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```
Assign
  targets=
    'tree'
  value=Call
    args=
      'code_str'
    func=Attribute
      attr='parse'
      value='ast'
    /Attribute
  /Call
/Assign
```

```
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```
For
  body=
    Expr
      value=Call
        args=
          'line'
        func='print'
        keywords=
          keyword
            arg='end'
            value=Str
              s=''
            /Str
          /keyword
      /Call
    /Expr
  iter=Call
    args=
      'tree'
    func='visit'
  /Call
  target='line'
/For
```

```python
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```
    returns=NameConstant
      value=None
    /NameConstant
  /FunctionDef
  If
    body=
      Expr
        value=Call
          args=
            '__file__'
          func='main'
        /Call
      /Expr
    test=Compare
      comparators=
        Str
          s='__main__'
        /Str
      left='__name__'
      ops=
        Eq
        /Eq
    /Compare
  /If
/Module
```

ABSTRACT VS CONCRETE SYNTAX TREES

```
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```python
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```
file_input
  simple_stmt
    import_name
      NAME 'import'
      NAME ' ' 'ast'
    /import_name
    NEWLINE '\n'
  /simple_stmt
  simple_stmt
    import_from
      NAME '\n' 'from'
      NAME ' ' 'simple_visitor'
      NAME ' ' 'import'
      NAME ' ' 'visit'
    /import_from
    NEWLINE '\n'
  /simple_stmt
```

```python
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```
funcdef
  NAME '\n\n' 'def'
  NAME ' ' 'main'
  parameters
    LPAR '('
    tname
      NAME 'path'
      COLON ':'
      NAME ' ' 'str'
    /tname
    RPAR ')'
  /parameters
  RARROW ' ' '->'
  NAME ' ' 'None'
  COLON ':'
```

```
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```
suite
  NEWLINE '\n'
  INDENT ''
  with_stmt
    NAME '    ' 'with'
    with_item
      power
        NAME ' ' 'open'
        trailer
          LPAR '('
          NAME 'path'
          RPAR ')'
        /trailer
      /power
      NAME ' ' 'as'
      NAME ' ' 'this_file'
    /with_item
    COLON ':'
```

```
import ast

from simple_visitor import visit


def main(path: str) -> None:
    with open(path) as this_file:
        code_str = this_file.read()

    tree = ast.parse(code_str)
    for line in visit(tree):
        print(line, end="")


if __name__ == "__main__":
    main(__file__)
```

```
suite
  NEWLINE '\n'
  INDENT ''
  simple_stmt
    expr_stmt
      NAME '          ' 'code_str'
      EQUAL ' ' '='
      power
        NAME ' ' 'this_file'
        trailer
          DOT '.'
          NAME 'read'
        /trailer
        trailer
          LPAR '('
          RPAR ')'
        /trailer
      /power
    /expr_stmt
    NEWLINE '\n'
  /simple_stmt
  DEDENT ''
/suite
/with_stmt
```

Problem Statement

Syntax and Trees

Design and Implementation

Fun Problems


Black

pip install black

DESIGN GOALS

HORIZONTAL WHITESPACE

VERTICAL WHITESPACE

LINE LENGTH

CONSISTENCY

HORIZONTAL WHITESPACE

```
self.comments[i] = (comma_index - 1, comment)
```

```
self.comments[i] = (comma_index - 1, comment)
def process(node=ROOT): ...
```

```
self.comments[i] = (comma_index - 1, comment)
def process(node=ROOT): ...
def process(node: Node = ROOT) -> bool: ...
```

```
self.comments[i] = (comma_index - 1, comment)
def process(node=ROOT): ...
def process(node: Node = ROOT) -> bool: ...
slice[d :: d + 1]
```

VERTICAL WHITESPACE

```
if TEST:↵
    ...

while TEST:↵
    ...
```

```
def f():
    ...
↵
↵
def g():
    ...
```

```
class C:
    def f():
        ...
    ↵
    def g():
        ...
```

```
def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    debug: bool = False,↵
):
    """Applies `variables` to the `template`
    and writes to `file`.
    """
    with open(file, "w") as f:
        ...
```

LINE LENGTH

80

120

“90-ish”

CONSISTENCY

```
from typing import (
    Any,
    Dict,
    Iterator,
    List,
    Optional,
    Type,
    TypeVar,
    Union,
)
```

STRINGS

f“yeah”

“don’t”

„„„

Do what I mean.

„„„

```
#include ”Python.h”
```

WHAT IS A "LINE", REALLY?

SUPER-SIMPLIFIED ALGORITHM

- Create a CST of the input file.
- Visit the tree, generating theoretical Line objects as you go.
- When a new Line is generated and its length requires splitting, split.
- For each split part of the Line, immediately recursively split again, if needed.
- Track blank lines separately outside of the line generation logic and apply a uniform number of them between classes, functions, etc.
- A Line once formatted and emitted externally is not touched again.

SUPER-SIMPLIFIED ALGORITHM

- Create a CST of the input file.
- Visit the tree, generating theoretical Line objects as you go.
- When a new Line is generated and its length requires splitting, split.
- For each split part of the Line, immediately recursively split again, if needed.
- Track blank lines separately outside of the line generation logic and apply a uniform number of them between classes, functions, etc.
- A Line once formatted and emitted externally is not touched again.

WHAT KINDS OF SPLITS ARE THERE?

BRACKET SPLITS

LEFT-HAND SIDE

- Take the left-most opening bracket.
- Emit all content up to and including that opening bracket as a new Line.
- Emit all content up to the matching closing bracket as a new Line.
- Emit all content from and including the matching closing bracket as a new Line.

RIGHT-HAND SIDE

- Take the right-most closing bracket.
- Find the matching opening bracket.
- Emit all content up to and including that opening bracket as a new Line.
- Emit all content up to the selected closing bracket as a new Line.
- Emit all content from and including the matching closing bracket as a new Line.

DELIMITER SPLITS

- If the Line currently received is within brackets, look for all **delimiters**.
- **Delimiters** are operators, commas, spaces between consecutive string literals, standalone comments, and so on.
- Look for the highest priority **delimiter**. If there are at least two of those, split the line on those delimiters.
- Split lines are still valid Python because this Line is **within a pair of brackets** so significant indentation does not apply.

SIMPLIFIED ALGORITHM

- Create a CST of the input file.
- Visit the tree, generating theoretical Line objects as you go.
  - We still put at most one statement per line.
  - We still split statements that have "suites" (e.g. an indented block after a colon).
- When a new Line is generated and its length requires splitting, attempt splits:
  - if this Line is not within a bracket pair, attempt a bracket split
  - otherwise, split by delimiters first, and if that doesn't work, attempt a bracket split
- For each splitted Line, immediately recursively split again.
- Blank lines are tracked separately outside of the line generation logic.
- A Line once formatted and emitted externally is not touched again.

SIMPLIFIED ALGORITHM

- Create a CST of the input file.
- Visit the tree, generating theoretical Line objects as you go.
  - We still put at most one statement per line.
  - We still split statements that have "suites" (e.g. an indented block after a colon).
- When a new Line is generated and its length requires splitting, attempt splits:
  - if this Line is not within a bracket pair, attempt a bracket split
  - otherwise, split by delimiters first, and if that doesn't work, attempt a bracket split
- For each splitted Line, immediately recursively split again.
- Blank lines are tracked separately outside of the line generation logic.
- A Line once formatted and emitted externally is not touched again.

```
class Visitor(Generic[T]):
    def visit(self, node: LN) -> Iterator[T]:
        if node.type < 256:
            name = token.tok_name[node.type]
        else:
            name = type_repr(node.type)
        yield from getattr(
            self, f"visit_{name}", self.visit_default
        )(node)

    def visit_default(self, node: LN) -> Iterator[T]:
        if isinstance(node, Node):
            for child in node.children:
                yield from self.visit(child)
```

```
@dataclass
class DebugVisitor(Visitor[T]):
    tree_depth: int = 0

    def visit_default(self, node: LN) -> Iterator[T]:
        indent = " " * (2 * self.tree_depth)
        if isinstance(node, Node):
            _type = type_repr(node.type)
            out(f"{indent}{_type}", fg="yellow")
            self.tree_depth += 1
            for child in node.children:
                yield from self.visit(child)
            self.tree_depth -= 1
            out(f"{indent}/{_type}", fg="yellow", bold=False)
        else:
            _type = token.tok_name.get(node.type, str(node.type))
            out(f"{indent}{_type}", fg="blue", nl=False)
            if node.prefix:
                # We don't have to handle prefixes for `Node` objects since
                # that delegates to the first child anyway.
                out(f" {node.prefix!r}", fg="green", bold=False, nl=False)
            out(f" {node.value!r}", fg="blue", bold=False)
```

THE ACTUAL VISITOR IN USE

```
lines = LineGenerator(
    remove_u_prefix=py36 or "unicode_literals" in future_imports,
    is_pyi=is_pyi,
    normalize_strings=normalize_strings,
    allow_underscores=py36
    and not bool(mode & FileMode.NO_NUMERIC_UNDERSCORE_NORMALIZATION),
)
elt = EmptyLineTracker(is_pyi=is_pyi)
empty_line = Line()
after = 0
for current_line in lines.visit(src_node):
    for _ in range(after):
        dst_contents += str(empty_line)
    before, after = elt.maybe_empty_lines(current_line)
    for _ in range(before):
        dst_contents += str(empty_line)
    for line in split_line(current_line, line_length=line_length, py36=py36):
        dst_contents += str(line)
```

OTHER POSSIBLE ALGORITHMS

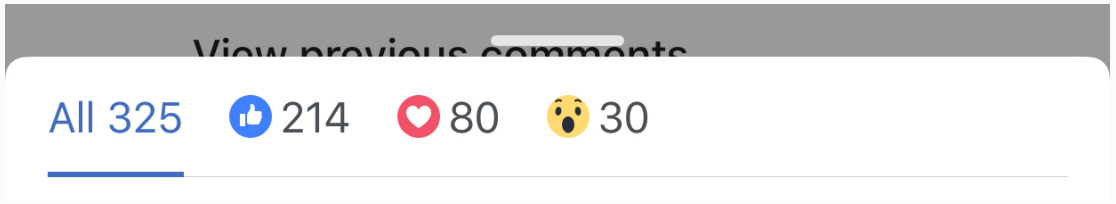
Problem Statement

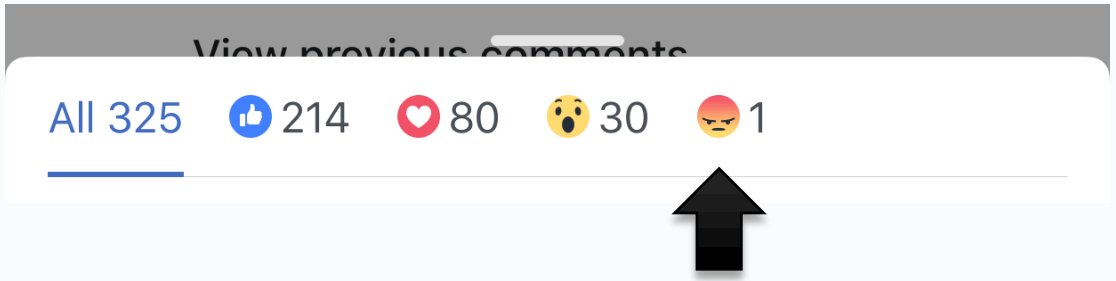
Syntax and Trees

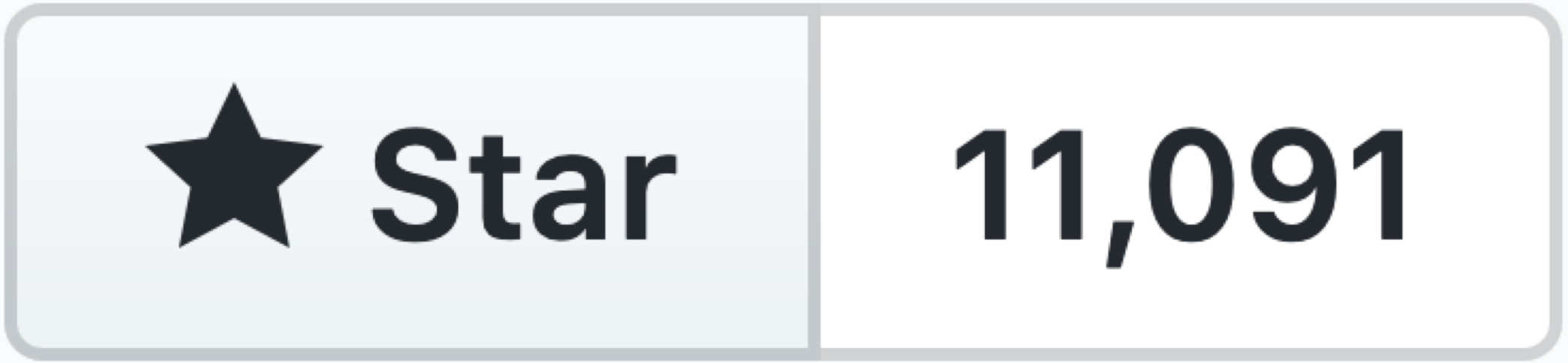
Design and Implementation

Fun Problems

USERS


View previous comments
All 325
214
80
30


View previous comments
All 325
214
80
30
1


Star
11,091


downloads
3M

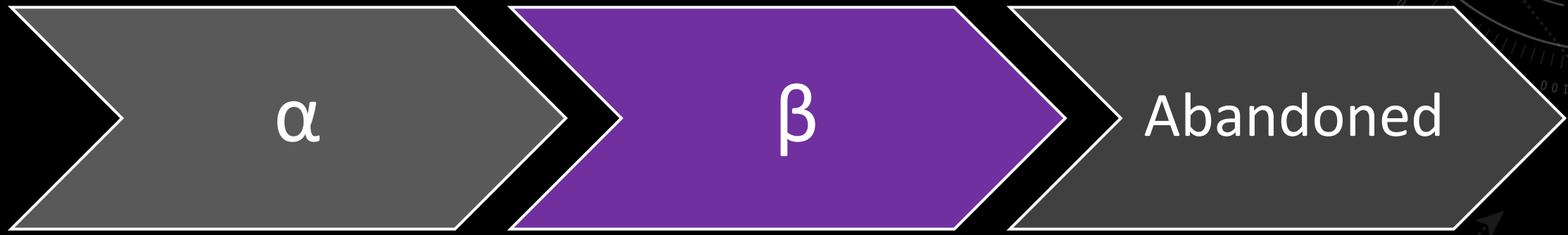
COMMENTS

OPTIONAL PARENTHESES

fmt: off

DETERMINISM

ACCIDENTAL QUADRATIC BEHAVIOR


α
β
Abandoned


α
β
Stable


α
β
Stable

YES

TIPS & TRICKS FOR ADOPTION

git-hyper-blame(1) Manual Page

NAME

git-hyper-blame - Like git blame, but with the ability to ignore or bypass certain commits.

SYNOPSIS

> *git hyper-blame* [-i <rev> [-i <rev> ...]] [--ignore-file=<file>]
> [--no-default-ignores] [<rev>] [--] <file>

DESCRIPTION

`git hyper-blame` is like `git blame` but it can ignore or "look through" a given set of commits, to find the rea

This is useful if you have a commit that makes sweeping changes that are unlikely to be what you are looking for

Add more commits by pushing to the **`git`** branch on **jgirardet/black**.

All checks have passed

3 successful checks

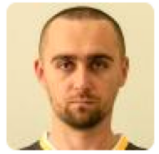
Show all checks

This branch has no conflicts with the base branch

Merging can be performed automatically.

Squash and merge ▾

You can also open this in GitHub Desktop or view command line instructions.



Write Preview

Leave a comment

pre-commit

A framework for managing and maintaining multi-language pre-commit hooks.

Introduction

Git hook scripts are useful for identifying simple issues before submission to code review. We run our hooks on every commit to automatically point out issues in code such as missing semicolons, trailing whitespace, and debug statements. By pointing these issues out before code review, this allows a code


ANY CODE STYLE YOU LIKE.
AS LONG AS IT'S
Black

Łukasz Langa
ambv on GitHub

fb.me/ambv
@llanga
lukasz@langa.pl