The Program Which Generates This Book

Martin O'Leary

Chapter 1

About this book

This book describes a computer program which, when executed, generates this book. The program is described in three ways:

First, a source code listing is given, in the Python programming language. This is the form of the program which was typed by the author, in text form.

Second, an *abstract syntax tree* is described, which is the computer's interpretation of the textual source code in terms of the language constructs available in the Python programming language.

Finally, the program is described in terms of *bytecode*, the computer's internal representation of the source code, a sequence of unambiguous instructions which can be executed to perform the computation described by the program.

The descriptions given in this book are generated by the program it describes, in conjunction with a Python interpreter, starting from the source code form. Both the abstract syntax tree and the bytecode representation are somewhat unstable. Different versions of the Python interpreter may yield different abstract syntax trees and different bytecode representations of the same program. This book was generated using `Python 3.6.1 (default, Apr 4 2017, 09:40:21) [GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.38)]`.

License

Chapter 2

Source code

```
import ast
import dis
import re
import sys
import types

title = "The Program Which Generates This Book"
author = "Martin O'Leary"

preface = """
This book describes a computer program which, when executed, generates this
book. The program is described in three ways:

First, a source code listing is given, in the Python programming language. This
is the form of the program which was typed by the author, in text form.

Second, an *abstract syntax tree* is described, which is the computer's
interpretation of the textual source code in terms of the language constructs
available in the Python programming language.

Finally, the program is described in terms of *bytecode*, the computer's internal
representation of the source code, a sequence of unambiguous instructions which
can be executed to perform the computation described by the program.

The descriptions given in this book are generated by the program it describes, in
conjunction with a Python interpreter, starting from the source code form. Both
the abstract syntax tree and the bytecode representation are somewhat unstable.
Different versions of the Python interpreter may yield different abstract syntax
trees and different bytecode representations of the same program. This book was
generated using `Python {}`.
""".format(sys.version)


def title_block():
    return "% {}\n% {}\n".format(title, author)


def describe_op(op, codes):
    f = descriptors.get(op.opname, None)
    if f:
        s = f(op, codes)
    else:
        s = ''
    if op.is_jump_target:
```

```
        s = "\n\n### Offset {}\n\n".format(op.offset) + s
    return s


def describe_file(filename):
    codetxt = open(filename).read()
    txt = title_block()
    txt += "# About this book\n\n"
    txt += preface
    txt += "\n\n## License\n\n"
    txt += open("LICENSE.md").read()
    txt += '\n\n# Source code\n\n'
    txt += '```\n' + codetxt + '\n```\n\n'
    txt += '# Abstract syntax tree\n\n'
    txt += describe_node(ast.parse(codetxt))
    txt += '\n\n# Bytecode\n\n'
    codes = [(filename, compile(codetxt, filename, 'exec', optimize=1))]
    while codes:
        name, code = codes.pop(0)
        txt += '## {}'.format(name)
        for op in dis.get_instructions(code):
            desc = describe_op(op, codes)
            if not desc: continue
            if op.starts_line:
                txt += '\n\n'
            txt += desc + ' '
        txt += '\n\n'
    return txt


def describe_number(num):
    words = [
        "zero", "one", "two", "three", "four", "five", "six", "seven", "eight",
        "nine", "ten"
    ]
    if 0 <= num <= 10:
        return words[num]
    elif num >= -10:
        return "minus " + words[-num]
    return str(num)


def as_list(items):
    items = list(items)
    if len(items) == 1:
        return items[0]
    else:
        return ', '.join(items[:-1]) + ", and " + items[-1]


def escape_string(s):
    s = re.sub(r'([_`\*\\#])', r'\\\1', s)
    s = re.sub(r'\n', r'\\\n', s)
    return s


def describe_value(value, codes):
    if isinstance(value, types.CodeType):
        # print(dir(value))
        name = value.co_name
```

```
        if name.startswith('<'):
            name = value.co_name[1:-1] + ':' + str(value.co_firstlineno)
        codes.append((name, value))
        return "the code object described under {}".format(name)
    elif isinstance(value, str):
        return "the literal string *'{}'*".format(escape_string(value))
    elif isinstance(value, int):
        return "the integer constant {}".format(describe_number(value))
    elif value is None:
        return "the constant None"
    elif isinstance(value, tuple):
        return "the tuple consisting of " + as_list(
            describe_value(x, codes) for x in value)
    else:
        print("Uninterpretable constant:", value)
    return repr(value)


def describe_node(node):
    f = descriptors.get(node.__class__.__name__, None)
    if f:
        return f(node)
    else:
        print(node, node._fields)
        return str(node)


descriptors = {}


def descriptor(f):
    descriptors[f.__name__] = f
    return f


@descriptor
def Module(node):
    return "A module, containing the following code:\n\n" + '\n\n'.join(
        describe_node(n) for n in node.body)


@descriptor
def Import(node):
    return "An import statement for a module named `{}`.".format(
        node.names[0].name)


@descriptor
def Assign(node):
    s = "An assignment to {}, of the value of {}.".format(
        describe_node(node.targets[0]), describe_node(node.value))
    return s


@descriptor
def AugAssign(node):
    s = "A modifying assignment to {}, using {}, of the value of {}.".format(
        describe_node(node.target),
        describe_node(node.op), describe_node(node.value))
    return s
```

```
@descriptor
def Add(node):
    return "the addition (or concatenation) operator"


@descriptor
def Mult(node):
    return "the multiplication operator"


@descriptor
def BitAnd(node):
    return "the bitwise 'AND' operator"


@descriptor
def Subscript(node):
    return "{}, subscripted by {}".format(
        describe_node(node.value), describe_node(node.slice))


@descriptor
def Index(node):
    return describe_node(node.value)


@descriptor
def Slice(node):
    if node.lower:
        return "a slice from {} to {}".format(
            describe_node(node.lower), describe_node(node.upper))
    else:
        return "a slice up to {}".format(describe_node(node.upper))


@descriptor
def For(node):
    s = "A for loop, where {} iterates over {}." \
        "The body of the loop is as follows:\n\n".format(
        describe_node(node.target), describe_node(node.iter))
    for nod in node.body:
        s += describe_node(nod) + "\n\n"
    s += "The for loop ends here."
    return s


@descriptor
def While(node):
    s = "A while loop, testing {}." \
        "The body of the loop is as follows:\n\n".format(
        describe_node(node.test))
    for nod in node.body:
        s += describe_node(nod) + "\n\n"
    s += "The while loop ends here."
    return s


@descriptor
```

```
def Continue(node):
    return "A 'continue' statement."


@descriptor
def Name(node):
    return "the name `{}`".format(node.id)


@descriptor
def NameConstant(node):
    return "the constant `{}`".format(node.value)


@descriptor
def List(node):
    if not node.elts:
        return "an empty list"
    else:
        return "a list containing " + as_list(
            describe_node(elt) for elt in node.elts)


@descriptor
def Tuple(node):
    if not node.elts:
        return "an empty tuple"
    else:
        return "a tuple containing " + as_list(
            describe_node(elt) for elt in node.elts)


@descriptor
def Dict(node):
    return "an empty dictionary"


@descriptor
def FunctionDef(node):
    s = "## {node.name}\n\n" \
        "A definition of a function named `{node.name}`".format(
        node=node)
    args = node.args
    if len(args.args) == 1:
        s += ", with argument `{}`.".format(args.args[0].arg)
    elif args.args:
        s += ", with positional arguments {args}.".format(args=as_list(
            ['`{}`'.format(a.arg) for a in args.args]))
    if node.decorator_list:
        s += " The definition is decorated with the function `{}`.".format(
            node.decorator_list[0].id)
    s += " The body of the function is as follows:\n\n"
    for nod in node.body:
        s += describe_node(nod) + '\n\n'

    s += "The function {} ends here.\n\n".format(node.name)
    return s


@descriptor
```

```
def Call(node):
    s = 'a function call, calling the value of {f}'.format(
        f=describe_node(node.func))
    if len(node.args) == 1:
        s += ', with argument {}'.format(describe_node(node.args[0]))
    elif node.args:
        s += ', with positional arguments {args}'.format(args=as_list(
            describe_node(a) for a in node.args))
    else:
        s += ' with no positional arguments'
    if node.keywords:
        if len(node.keywords) == 1:
            s += ', and keyword argument'
        else:
            s += ', and keyword arguments'
        for kw in node.keywords:
            s += ', assigning {} as `{}`'.format(
                describe_node(kw.value), kw.arg)
    return s


@descriptor
def Return(node):
    return "A return statement, returning the value of {}.".format(
        describe_node(node.value))


@descriptor
def Str(node):
    return "the literal string *'{}'*".format(escape_string(node.s))


@descriptor
def Attribute(node):
    return "an attribute lookup of `{}` on {}".format(
        node.attr, describe_node(node.value))


@descriptor
def Expr(node):
    return "A bare expression with value {}.".format(describe_node(node.value))


@descriptor
def BinOp(node):
    return "{}, with left hand side {}, and right hand side {}".format(
        describe_node(node.op),
        describe_node(node.left), describe_node(node.right))


@descriptor
def If(node):
    s = "An `if` statement, testing {}. " \
        "The body of the main branch is as follows:\n\n".format(
        describe_node(node.test))
    for nod in node.body:
        s += describe_node(nod) + "\n\n"
    if node.orelse:
        s += "The other ('else') branch of the `if` statement is as follows:\n\n
        for nod in node.orelse:
```

```
            s += describe_node(nod) + "\n\n"
    s += "The `if` statement ends here.\n\n"
    return s


@descriptor
def Num(node):
    return "a numeric constant with value {}".format(node.n)


@descriptor
def Compare(node):
    if len(node.ops) == 1:
        return "a comparison (using {}) of {} and {}".format(
            describe_node(node.ops[0]),
            describe_node(node.left), describe_node(node.comparators[0]))
    else:
        lefts = [node.left] + node.comparators[:-1]
        rights = node.comparators
        s = "a compound comparison, comparing "
        s += as_list("{} and {} using {}".format(
            describe_node(left), describe_node(right), describe_node(op))
                     for left, op, right in zip(lefts, node.ops, rights))
        return s


@descriptor
def Eq(node):
    return "the equality operator"


@descriptor
def GtE(node):
    return "the 'greater than or equal to' operator"

@descriptor
def LtE(node):
    return "the 'less than or equal to' operator"

@descriptor
def Gt(node):
    return "the 'greater than' operator"


@descriptor
def Is(node):
    return "the identity operator"


@descriptor
def UnaryOp(node):
    return "{} applied to {}".format(
        describe_node(node.op), describe_node(node.operand))


@descriptor
def Not(node):
    return "the unary 'not' operator"
```

```
@descriptor
def USub(node):
    return "the unary negation operator"


@descriptor
def GeneratorExp(node):
    gen = node.generators[0]
    return "a generator expression, taking the value of {}, " \
        "as {} ranges over {}".format(
        describe_node(node.elt),
        describe_node(gen.target), describe_node(gen.iter))


@descriptor
def ListComp(node):
    gen = node.generators[0]
    return "a list comprehension, taking the value of {}, " \
        "as {} ranges over {}".format(
        describe_node(node.elt),
        describe_node(gen.target), describe_node(gen.iter))


@descriptor
def Assert(node):
    return ""


@descriptor
def LOAD_CONST(op, codes):
    return "The computer places {} on top of the stack.".format(
        describe_value(op.argval, codes))


@descriptor
def LOAD_NAME(op, codes):
    return "The computer places the value associated with the name `{}` " \
        "on top of the stack.".format(
        op.argval)


@descriptor
def CALL_FUNCTION(op, codes):
    if op.argval == 0:
        return "The computer takes the top value from the stack " \
            "and calls it as a function (with no arguments), " \
            "placing the return value on top of the stack."
    elif op.argval == 1:
        return "The computer takes the top value from the stack, " \
            "along with another value which it calls as a function, " \
            "using the original value as an argument, " \
            "placing the return value on the stack.".format(
            op.argval)
    else:
        return "The computer takes {} values from the stack, " \
            "along with another value which it calls as a function, " \
            "using the original values as arguments, " \
            "placing the return value on the stack.".format(
            describe_number(op.argval))
```

```
@descriptor
def POP_TOP(op, codes):
    return "The computer discards the top value from the stack."


@descriptor
def RETURN_VALUE(op, codes):
    return "The computer exits the current function, " \
        "returning the top value on the stack."


@descriptor
def STORE_NAME(op, codes):
    return "The computer takes the top value from the stack, " \
        "and stores it under the name `{}`.".format(
        op.argval)


@descriptor
def BINARY_SUBSCR(op, codes):
    return "The computer takes the top two values from the stack " \
        "and retrieves the value of the second item, " \
        "subscripted by the value of the first item."


@descriptor
def LOAD_ATTR(op, codes):
    return "The computer takes the top value from the stack " \
        "and retrieves its attribute named `{}`, " \
        "placing it on the stack.".format(
        op.argval)


@descriptor
def POP_JUMP_IF_FALSE(op, codes):
    return "The computer takes the top value from the stack, " \
        "and if it is false-like (e.g. False, None or zero), " \
        "jumps to offset {}.".format(
        op.argval)


@descriptor
def POP_JUMP_IF_TRUE(op, codes):
    return "The computer takes the top value from the stack, " \
        "and if it is true-like (e.g. True, non-empty or non-zero), " \
        "jumps to offset {}.".format(
        op.argval)


@descriptor
def IMPORT_NAME(op, codes):
    return "The computer takes the top two values from the stack " \
        "and uses them as the 'fromlist' and 'level' of an import " \
        "for the module `{}`, which is placed on the stack.".format(
        op.argval)


@descriptor
def MAKE_FUNCTION(op, codes):
```

```
    txt = "The computer takes the top two values from the stack " \
        "and uses them as the qualified name and code of a new function, " \
        "which is placed on the stack."
    if op.argval & 8:
        txt += ' It also takes the next value as a tuple of cells ' \
            'for free variables, creating a closure.'
    if op.argval & 4:
        txt += ' It also takes the next value as a dictionary ' \
            'of function annotations.'
    if op.argval & 2:
        txt += ' It also takes the next value as a dictionary ' \
            'of keyword arguments.'
    if op.argval & 1:
        txt += ' It also takes the next value as a tuple of default arguments.'
    return txt


@descriptor
def COMPARE_OP(op, codes):
    if op.argval == '==':
        return "The computer takes the top two values from the stack " \
            "and compares them for equality, " \
            "placing the result on top of the stack."
    elif op.argval == 'is':
        return "The computer takes the top two values from the stack " \
            "and compares them for identity, " \
            "placing the result on top of the stack."
    return "The computer takes the top two values from the stack " \
        "and compares them using the operator `{}`, " \
        "placing the result on top of the stack.".format(
        op.argval)


@descriptor
def BUILD_MAP(op, codes):
    if op.argval == 0:
        return "The computer places an empty dictionary on top of the stack."
    return "The computer takes the top {} values from the stack, " \
        "and uses them as key-value pairs in a new dictionary, " \
        "which is placed on top of the stack.".format(
        describe_number(2 * op.argval))


@descriptor
def EXTENDED_ARG(op, codes):
    return ""


@descriptor
def BINARY_ADD(op, codes):
    return "The computer takes the top two values from the stack, " \
        "adds them together, and places the result on top of the stack."


@descriptor
def BINARY_MULTIPLY(op, codes):
    return "The computer takes the top two values from the stack, " \
        "multiplies them together, and places the result on top of the stack."
```

```
@descriptor
def BINARY_AND(op, codes):
    return "The computer takes the top two values from the stack, " \
        "applies a bitwise `AND` operator to them, " \
        "and places the result on top of the stack."


@descriptor
def BUILD_LIST(op, codes):
    if op.argval == 0:
        return "The computer places a new empty list on top of the stack."
    elif op.argval == 1:
        return "The computer takes the top value from the stack, " \
            "puts it in a list, and places it on top of the stack."
    else:
        return "The computer takes the top {} values from the stack, " \
            "puts them in a list, and places it on top of the stack.".format(
            describe_number(op.argval))


@descriptor
def BUILD_SLICE(op, codes):
    return "The computer takes the top two values from the stack, " \
        "creates a slice object from them, and places it on top of the stack."


@descriptor
def BUILD_TUPLE(op, codes):
    if op.argval == 1:
        return "The computer takes the top value from the stack, " \
            "creates a tuple from it, and places it on top of the stack."
    return "The computer takes the top {} values from the stack, " \
        "creates a tuple from them, and places it on top of the stack.".format(
        describe_number(op.argval))


@descriptor
def FOR_ITER(op, codes):
    return "The computer looks at the top value on the stack and " \
        "calls its `next()` method. If it returns a value, " \
        "it places it on top of the stack. If not, it removes " \
        "the top value from the stack and jumps to offset {}.".format(
        op.argval)


@descriptor
def GET_ITER(op, codes):
    return "The computer takes the top value from the stack, " \
        "turns it into an iterator (using `iter()`), " \
        "and places the result on top of the stack."


@descriptor
def INPLACE_ADD(op, codes):
    return "The computer takes the top value from the stack and (in place)" \
        "adds the second from top value from the stack to it, " \
        "placing the result on top of the stack."


@descriptor
```

```
def JUMP_ABSOLUTE(op, codes):
    return "The computer jumps to offset {}.".format(op.argval)


@descriptor
def JUMP_FORWARD(op, codes):
    return "The computer jumps forward to offset {}.".format(op.argval)


@descriptor
def LIST_APPEND(op, codes):
    return "The computer takes the top value from the stack and appends it " \
        "to the list stored {} places from the top of the stack.".format(
        describe_number(op.argval))


@descriptor
def LOAD_CLOSURE(op, codes):
    return "The computer loads a reference to the free variable named `{}` " \
        "and places it on top of the stack.".format(
        op.argval)


@descriptor
def LOAD_DEREF(op, codes):
    return "The computer loads the contents of the free variable named `{}` " \
        "and places it on top of the stack.".format(
        op.argval)


@descriptor
def LOAD_FAST(op, codes):
    return "The computer loads a reference to the local variable named `{}` " \
        "and places it on top of the stack.".format(
        op.argval)


@descriptor
def LOAD_GLOBAL(op, codes):
    return "The computer loads a reference to the global variable named `{}` " \
        "and places it on top of the stack.".format(
        op.argval)


@descriptor
def POP_BLOCK(op, codes):
    return "The computer removes one block from the block stack."


@descriptor
def SETUP_LOOP(op, codes):
    return "The computer places a new block for a loop on top of " \
        "the block stack, extending until offset {}.".format(
        op.argval)


@descriptor
def STORE_DEREF(op, codes):
    return "The computer takes the top value from the stack and stores " \
        "it in the free variable named `{}`.".format(
```

```
        op.argval)


@descriptor
def STORE_FAST(op, codes):
    return "The computer takes the top value from the stack and stores " \
        "it in the local variable named `{}`.".format(
        op.argval)


@descriptor
def STORE_SUBSCR(op, codes):
    return "The computer takes the top value from the stack, " \
        "uses it to index into the next-from-top value, " \
        "and stores the value below that in that location."


@descriptor
def UNPACK_SEQUENCE(op, codes):
    return "The computer takes the top value from the stack, " \
        "unpacks it into {} values, " \
        "then places them each on top of the stack.".format(
        describe_number(op.argval))


@descriptor
def YIELD_VALUE(op, codes):
    return "The computer takes the top value from the stack " \
        "and yields it from the current generator."


@descriptor
def CALL_FUNCTION_KW(op, codes):
    return "The computer takes the top value from the stack " \
        "and interprets it as a tuple of keyword names. " \
        "It then takes values from the top of the stack as " \
        "corresponding values, followed by positional arguments " \
        "up to a total of {} values (both keyword and positional). " \
        "Then it takes the next value from the top of the stack and " \
        "calls it as a function with these arguments, " \
        "placing the return value on top of the stack.".format(
        op.argval)


@descriptor
def DUP_TOP(op, codes):
    return "The computer duplicates the top value on the stack, " \
        "placing the new copy on top of the stack."


@descriptor
def ROT_TWO(op, codes):
    return "The computer takes the top two values from the stack, " \
        "swaps them, and replaces them on top of the stack."


@descriptor
def ROT_THREE(op, codes):
    return "The computer takes the top three values from the stack, " \
        "rotates them so that the top value is now on the bottom, " \
```

```
        "and replaces them on top of the stack."


@descriptor
def UNARY_NEGATIVE(op, codes):
    return "The computer takes the top value from the stack, negates it, " \
        "and places the result on top of the stack."


@descriptor
def JUMP_IF_FALSE_OR_POP(op, codes):
    return "The computer looks at the top value on the stack. " \
        "If it is false-like (e.g. False, None or zero), it jumps " \
        "to offset {}. Otherwise it removes the top value from the stack."


if __name__ == '__main__':
    outfile = sys.argv[1]
    filename = __file__
    if len(sys.argv) > 2:
        filename = sys.argv[2]
    f = open(outfile, "w")
    f.write(describe_file(filename))
    f.close()
```

Chapter 3

Abstract syntax tree

A module, containing the following code:
    An import statement for a module named `ast`.
    An import statement for a module named `dis`.
    An import statement for a module named `re`.
    An import statement for a module named `sys`.
    An import statement for a module named `types`.
    An assignment to the name `title`, of the value of the literal string *'The Program Which Generates This Book'*.
    An assignment to the name `author`, of the value of the literal string *'Martin O'Leary'*.
    An assignment to the name `preface`, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'\nThis book describes a computer program which, when executed, generates this\nbook. The program is described in three ways:\n\nFirst, a source code listing is given, in the Python programming language. This\nis the form of the program which was typed by the author, in text form.\n\nSecond, an *abstract syntax tree* is described, which is the computer's\ninterpretation of the textual source code in terms of the language constructs\navailable in the Python programming language.\n\nFinally, the program is described in terms of *bytecode*, the computer's internal\nrepresentation of the source code, a sequence of unambiguous instructions which\ncan be executed to perform the computation described by the program.\n\nThe descriptions given in this book are generated by the program it describes, in\nconjunction with a Python interpreter, starting from the source code form. Both \nthe abstract syntax tree and the bytecode representation are somewhat unstable.\nDifferent versions of the Python interpreter may yield different abstract syntax\ntrees and different bytecode representations of the same program. This book was\ngenerated using 'Python {}'.\n'*, with argument an attribute lookup of `version` on the name `sys`.

title_block

A definition of a function named `title_block` The body of the function is as follows:
    A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'% {}\n% {}\n'*, with positional arguments the name `title`, and the name `author`.
    The function title_block ends here.

describe_op

A definition of a function named `describe_op`, with positional arguments `op`, and `codes`. The body of the function is as follows:

An assignment to the name `f`, of the value of a function call, calling the value of an attribute lookup of `get` on the name `descriptors`, with positional arguments an attribute lookup of `opname` on the name `op`, and the constant `None`.

An `if` statement, testing the name `f`. The body of the main branch is as follows:

An assignment to the name `s`, of the value of a function call, calling the value of the name `f`, with positional arguments the name `op`, and the name `codes`.

The other ('else') branch of the `if` statement is as follows:

An assignment to the name `s`, of the value of the literal string ''.

The `if` statement ends here.

An `if` statement, testing an attribute lookup of `is_jump_target` on the name `op`. The body of the main branch is as follows:

An assignment to the name `s`, of the value of the addition (or concatenation) operator, with left hand side a function call, calling the value of an attribute lookup of `format` on the literal string *'\n\n### Offset {}\n\n'*, with argument an attribute lookup of `offset` on the name `op`, and right hand side the name `s`.

The `if` statement ends here.

A return statement, returning the value of the name `s`.

The function describe_op ends here.

describe_file

A definition of a function named `describe_file`, with argument `filename`. The body of the function is as follows:

An assignment to the name `codetxt`, of the value of a function call, calling the value of an attribute lookup of `read` on a function call, calling the value of the name `open`, with argument the name `filename` with no positional arguments.

An assignment to the name `txt`, of the value of a function call, calling the value of the name `title_block` with no positional arguments.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'# About this book\n\n'*.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the name `preface`.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'\n\n## License\n\n'*.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of a function call, calling the value of an attribute lookup of `read` on a function call, calling the value of the name `open`, with argument the literal string *'LICENSE.md'* with no positional arguments.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'\n\n# Source code\n\n'*.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the addition (or concatenation) operator, with left hand side the addition (or concatenation) operator, with left hand side the literal string *""\n'*, and right hand side the name `codetxt`, and right hand side the literal string *'\n"'\n\n'*.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'# Abstract syntax tree\n\n'*.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of a function call, calling the value of the name `describe_node`, with argument a function call, calling the value of an attribute lookup of `parse` on the name `ast`, with argument the name `codetxt`.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'\n\n# Bytecode\n\n'*.

An assignment to the name `codes`, of the value of a list containing a tuple containing the name `filename`, and a function call, calling the value of the name `compile`, with positional arguments the name `codetxt`, the name `filename`,

and the literal string *'exec'*, and keyword argument, assigning a numeric constant with value 1 as `optimize`.

A while loop, testing the name `codes`.The body of the loop is as follows:

An assignment to a tuple containing the name `name`, and the name `code`, of the value of a function call, calling the value of an attribute lookup of `pop` on the name `codes`, with argument a numeric constant with value 0.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'## {}'*, with argument the name `name`.

A for loop, where the name `op` iterates over a function call, calling the value of an attribute lookup of `get_instructions` on the name `dis`, with argument the name `code`.The body of the loop is as follows:

An assignment to the name `desc`, of the value of a function call, calling the value of the name `describe_op`, with positional arguments the name `op`, and the name `codes`.

An `if` statement, testing the unary 'not' operator applied to the name `desc`. The body of the main branch is as follows:

A 'continue' statement.

The `if` statement ends here.

An `if` statement, testing an attribute lookup of `starts_line` on the name `op`. The body of the main branch is as follows:

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'\n\n'*.

The `if` statement ends here.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the addition (or concatenation) operator, with left hand side the name `desc`, and right hand side the literal string *''*.

The for loop ends here.

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'\n\n'*.

The while loop ends here.

A return statement, returning the value of the name `txt`.

The function describe_file ends here.

describe_number

A definition of a function named `describe_number`, with argument `num`. The body of the function is as follows:

An assignment to the name `words`, of the value of a list containing the literal string *'zero'*, the literal string *'one'*, the literal string *'two'*, the literal string *'three'*, the literal string *'four'*, the literal string *'five'*, the literal string *'six'*, the literal string *'seven'*, the literal string *'eight'*, the literal string *'nine'*, and the literal string *'ten'*.

An `if` statement, testing a compound comparison, comparing a numeric constant with value 0 and the name `num` using the 'less than or equal to' operator, and the name `num` and a numeric constant with value 10 using the 'less than or equal to' operator. The body of the main branch is as follows:

A return statement, returning the value of the name `words`, subscripted by the name `num`.

The other ('else') branch of the `if` statement is as follows:

An `if` statement, testing a comparison (using the 'greater than or equal to' operator) of the name `num` and the unary negation operator applied to a numeric constant with value 10. The body of the main branch is as follows:

A return statement, returning the value of the addition (or concatenation) operator, with left hand side the literal string *'minus'*, and right hand side the name `words`, subscripted by the unary negation operator applied to the name `num`.

The `if` statement ends here.

The `if` statement ends here.

A return statement, returning the value of a function call, calling the value of the name `str`, with argument the name `num`.

The function describe_number ends here.

as_list

A definition of a function named `as_list`, with argument `items`. The body of the function is as follows:

An assignment to the name `items`, of the value of a function call, calling the value of the name `list`, with argument the name `items`.

An `if` statement, testing a comparison (using the equality operator) of a function call, calling the value of the name `len`, with argument the name `items` and a numeric constant with value 1. The body of the main branch is as follows:

A return statement, returning the value of the name `items`, subscripted by a numeric constant with value 0.

The other ('else') branch of the `if` statement is as follows:

A return statement, returning the value of the addition (or concatenation) operator, with left hand side the addition (or concatenation) operator, with left hand side a function call, calling the value of an attribute lookup of `join` on the literal string *',*', with argument the name `items`, subscripted by a slice up to the unary negation operator applied to a numeric constant with value 1, and right hand side the literal string *', and'*, and right hand side the name `items`, subscripted by the unary negation operator applied to a numeric constant with value 1.

The `if` statement ends here.

The function as_list ends here.

escape_string

A definition of a function named `escape_string`, with argument `s`. The body of the function is as follows:

An assignment to the name `s`, of the value of a function call, calling the value of an attribute lookup of `sub` on the name `re`, with positional arguments the literal string *'([_'*\\#])'*, the literal string *'\\\1'*, and the name `s`.

An assignment to the name `s`, of the value of a function call, calling the value of an attribute lookup of `sub` on the name `re`, with positional arguments the literal string *'\n'*, the literal string *'\\\\n'*, and the name `s`.

A return statement, returning the value of the name `s`.

The function escape_string ends here.

describe_value

A definition of a function named `describe_value`, with positional arguments `value`, and `codes`. The body of the function is as follows:

An `if` statement, testing a function call, calling the value of the name `isinstance`, with positional arguments the name `value`, and an attribute lookup of `CodeType` on the name `types`. The body of the main branch is as follows:

An assignment to the name `name`, of the value of an attribute lookup of `co_name` on the name `value`.

An `if` statement, testing a function call, calling the value of an attribute lookup of `startswith` on the name `name`, with argument the literal string '<'. The body of the main branch is as follows:

An assignment to the name `name`, of the value of the addition (or concatenation) operator, with left hand side the addition (or concatenation) operator, with left hand side an attribute lookup of `co_name` on the name `value`, subscripted by a slice from a numeric constant with value 1 to the unary negation operator applied to a numeric constant with value 1, and right hand side the literal string

':', and right hand side a function call, calling the value of the name `str`, with argument an attribute lookup of `co_firstlineno` on the name `value`.

The `if` statement ends here.

A bare expression with value a function call, calling the value of an attribute lookup of `append` on the name `codes`, with argument a tuple containing the name `name`, and the name `value`.

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'the code object described under {}'*, with argument the name `name`.

The other ('else') branch of the `if` statement is as follows:

An `if` statement, testing a function call, calling the value of the name `isinstance`, with positional arguments the name `value`, and the name `str`. The body of the main branch is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'the literal string *'{}'*'*, with argument a function call, calling the value of the name `escape_string`, with argument the name `value`.

The other ('else') branch of the `if` statement is as follows:

An `if` statement, testing a function call, calling the value of the name `isinstance`, with positional arguments the name `value`, and the name `int`. The body of the main branch is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'the integer constant {}'*, with argument a function call, calling the value of the name `describe_number`, with argument the name `value`.

The other ('else') branch of the `if` statement is as follows:

An `if` statement, testing a comparison (using the identity operator) of the name `value` and the constant `None`. The body of the main branch is as follows:

A return statement, returning the value of the literal string *'the constant None'*.

The other ('else') branch of the `if` statement is as follows:

An `if` statement, testing a function call, calling the value of the name `isinstance`, with positional arguments the name `value`, and the name `tuple`. The body of the main branch is as follows:

A return statement, returning the value of the addition (or concatenation) operator, with left hand side the literal string *'the tuple consisting of'*, and right hand side a function call, calling the value of the name `as_list`, with argument a generator expression, taking the value of a function call, calling the value of the name `describe_value`, with positional arguments the name `x`, and the name `codes`, as the name `x` ranges over the name `value`.

The other ('else') branch of the `if` statement is as follows:

A bare expression with value a function call, calling the value of the name `print`, with positional arguments the literal string *'Uninterpretable constant:'*, and the name `value`.

The `if` statement ends here.

The `if` statement ends here.

The `if` statement ends here.

The `if` statement ends here.

The `if` statement ends here.

A return statement, returning the value of a function call, calling the value of the name `repr`, with argument the name `value`.

The function describe_value ends here.

describe_node

A definition of a function named `describe_node`, with argument `node`. The body of the function is as follows:

An assignment to the name `f`, of the value of a function call, calling the value of an attribute lookup of `get` on the name `descriptors`, with positional argu-

ments an attribute lookup of __name__ on an attribute lookup of __class__ on the name node, and the constant None.

An if statement, testing the name f. The body of the main branch is as follows:

A return statement, returning the value of a function call, calling the value of the name f, with argument the name node.

The other ('else') branch of the if statement is as follows:

A bare expression with value a function call, calling the value of the name print, with positional arguments the name node, and an attribute lookup of _fields on the name node.

A return statement, returning the value of a function call, calling the value of the name str, with argument the name node.

The if statement ends here.

The function describe_node ends here.

An assignment to the name descriptors, of the value of an empty dictionary.

descriptor

A definition of a function named descriptor, with argument f. The body of the function is as follows:

An assignment to the name descriptors, subscripted by an attribute lookup of __name__ on the name f, of the value of the name f.

A return statement, returning the value of the name f.

The function descriptor ends here.

Module

A definition of a function named Module, with argument node. The definition is decorated with the function descriptor. The body of the function is as follows:

A return statement, returning the value of the addition (or concatenation) operator, with left hand side the literal string *'A module, containing the following code:\n\n'*, and right hand side a function call, calling the value of an attribute lookup of join on the literal string '\n\n', with argument a generator expression, taking the value of a function call, calling the value of the name describe_node, with argument the name n, as the name n ranges over an attribute lookup of body on the name node.

The function Module ends here.

Import

A definition of a function named Import, with argument node. The definition is decorated with the function descriptor. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of format on the literal string *'An import statement for a module named '{}'.'*, with argument an attribute lookup of name on an attribute lookup of names on the name node, subscripted by a numeric constant with value 0.

The function Import ends here.

Assign

A definition of a function named Assign, with argument node. The definition is decorated with the function descriptor. The body of the function is as follows:

An assignment to the name `s`, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'An assignment to {}, of the value of {}.'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `targets` on the name `node`, subscripted by a numeric constant with value 0, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `value` on the name `node`.

A return statement, returning the value of the name `s`.

The function Assign ends here.

AugAssign

A definition of a function named `AugAssign`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An assignment to the name `s`, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'A modifying assignment to {}, using {}, of the value of {}.'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `target` on the name `node`, a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `op` on the name `node`, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `value` on the name `node`.

A return statement, returning the value of the name `s`.

The function AugAssign ends here.

Add

A definition of a function named `Add`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'the addition (or concatenation) operator'*.

The function Add ends here.

Mult

A definition of a function named `Mult`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'the multiplication operator'*.

The function Mult ends here.

BitAnd

A definition of a function named `BitAnd`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'the bitwise 'AND' operator'*.

The function BitAnd ends here.

Subscript

A definition of a function named `Subscript`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'{}, subscripted by {}'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `value` on the name `node`, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `slice` on the name `node`.

The function Subscript ends here.

Index

A definition of a function named `Index`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `value` on the name `node`.

The function Index ends here.

Slice

A definition of a function named `Slice`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An `if` statement, testing an attribute lookup of `lower` on the name `node`. The body of the main branch is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'a slice from {} to {}'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `lower` on the name `node`, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `upper` on the name `node`.

The other ('else') branch of the `if` statement is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'a slice up to {}'*, with argument a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `upper` on the name `node`.

The `if` statement ends here.

The function Slice ends here.

For

A definition of a function named `For`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An assignment to the name `s`, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'A for loop, where {} iterates over {}.The body of the loop is as follows:\n\n'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `target` on the name `node`, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `iter` on the name `node`.

A for loop, where the name `nod` iterates over an attribute lookup of `body` on the name `node`.The body of the loop is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the addition (or concatenation) operator, with left hand side a function call, calling the value of the name `describe_node`, with argument the name `nod`, and right hand side the literal string '\n\n'.

The for loop ends here.

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the literal string *'The for loop ends here.'*.

A return statement, returning the value of the name `s`.

The function For ends here.

While

A definition of a function named `While`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An assignment to the name `s`, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'A while loop, testing {}.The body of the loop is as follows:\n\n'*, with argument a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `test` on the name `node`.

A for loop, where the name `nod` iterates over an attribute lookup of `body` on the name `node`.The body of the loop is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the addition (or concatenation) operator, with left hand side a function call, calling the value of the name `describe_node`, with argument the name `nod`, and right hand side the literal string *'\n\n'*.

The for loop ends here.

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the literal string *'The while loop ends here.'*.

A return statement, returning the value of the name `s`.

The function While ends here.

Continue

A definition of a function named `Continue`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'A 'continue' statement.'*.

The function Continue ends here.

Name

A definition of a function named `Name`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'the name '{}''*, with argument an attribute lookup of `id` on the name `node`.

The function Name ends here.

NameConstant

A definition of a function named `NameConstant`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'the constant '{}''*, with argument an attribute lookup of `value` on the name `node`.

The function NameConstant ends here.

List

A definition of a function named `List`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An `if` statement, testing the unary 'not' operator applied to an attribute lookup of `elts` on the name `node`. The body of the main branch is as follows:

A return statement, returning the value of the literal string *'an empty list'*.

The other ('else') branch of the `if` statement is as follows:

A return statement, returning the value of the addition (or concatenation) operator, with left hand side the literal string *'a list containing'*, and right hand side a function call, calling the value of the name `as_list`, with argument a generator expression, taking the value of a function call, calling the value of the name `describe_node`, with argument the name `elt`, as the name `elt` ranges over an attribute lookup of `elts` on the name `node`.

The `if` statement ends here.

The function List ends here.

Tuple

A definition of a function named `Tuple`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An `if` statement, testing the unary 'not' operator applied to an attribute lookup of `elts` on the name `node`. The body of the main branch is as follows:

A return statement, returning the value of the literal string *'an empty tuple'*.

The other ('else') branch of the `if` statement is as follows:

A return statement, returning the value of the addition (or concatenation) operator, with left hand side the literal string *'a tuple containing'*, and right hand side a function call, calling the value of the name `as_list`, with argument a generator expression, taking the value of a function call, calling the value of the name `describe_node`, with argument the name `elt`, as the name `elt` ranges over an attribute lookup of `elts` on the name `node`.

The `if` statement ends here.

The function Tuple ends here.

Dict

A definition of a function named `Dict`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'an empty dictionary'*.

The function Dict ends here.

FunctionDef

A definition of a function named `FunctionDef`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An assignment to the name `s`, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'## {node.name}\n\nA definition of a function named '{node.name}''* with no positional arguments, and keyword argument, assigning the name `node` as `node`.

An assignment to the name `args`, of the value of an attribute lookup of `args` on the name `node`.

An `if` statement, testing a comparison (using the equality operator) of a function call, calling the value of the name `len`, with argument an attribute

lookup of `args` on the name `args` and a numeric constant with value 1. The body of the main branch is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *', with argument '{}'.'*, with argument an attribute lookup of `arg` on an attribute lookup of `args` on the name `args`, subscripted by a numeric constant with value 0.

The other ('else') branch of the `if` statement is as follows:

An `if` statement, testing an attribute lookup of `args` on the name `args`. The body of the main branch is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *', with positional arguments {args}.'* with no positional arguments, and keyword argument, assigning a function call, calling the value of the name `as_list`, with argument a list comprehension, taking the value of a function call, calling the value of an attribute lookup of `format` on the literal string *"{}"*, with argument an attribute lookup of `arg` on the name `a`, as the name `a` ranges over an attribute lookup of `args` on the name `args` as `args`.

The `if` statement ends here.

The `if` statement ends here.

An `if` statement, testing an attribute lookup of `decorator_list` on the name `node`. The body of the main branch is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The definition is decorated with the function '{}'.'*, with argument an attribute lookup of `id` on an attribute lookup of `decorator_list` on the name `node`, subscripted by a numeric constant with value 0.

The `if` statement ends here.

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the literal string *'The body of the function is as follows:\n\n'*.

A for loop, where the name `nod` iterates over an attribute lookup of `body` on the name `node`.The body of the loop is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the addition (or concatenation) operator, with left hand side a function call, calling the value of the name `describe_node`, with argument the name `nod`, and right hand side the literal string *'\n\n'*.

The for loop ends here.

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The function {} ends here.\n\n'*, with argument an attribute lookup of `name` on the name `node`.

A return statement, returning the value of the name `s`.

The function FunctionDef ends here.

Call

A definition of a function named `Call`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An assignment to the name `s`, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'a function call, calling the value of {f}'* with no positional arguments, and keyword argument, assigning a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `func` on the name `node` as `f`.

An `if` statement, testing a comparison (using the equality operator) of a function call, calling the value of the name `len`, with argument an attribute lookup of `args` on the name `node` and a numeric constant with value 1. The body of the main branch is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *', with argument {}'*, with argument a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `args` on the name `node`, subscripted by a numeric constant with value 0.

The other ('else') branch of the `if` statement is as follows:

An `if` statement, testing an attribute lookup of `args` on the name `node`. The body of the main branch is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *', with positional arguments {args}'* with no positional arguments, and keyword argument, assigning a function call, calling the value of the name `as_list`, with argument a generator expression, taking the value of a function call, calling the value of the name `describe_node`, with argument the name `a`, as the name `a` ranges over an attribute lookup of `args` on the name `node` as `args`.

The other ('else') branch of the `if` statement is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the literal string *'with no positional arguments'*.

The `if` statement ends here.

The `if` statement ends here.

An `if` statement, testing an attribute lookup of `keywords` on the name `node`. The body of the main branch is as follows:

An `if` statement, testing a comparison (using the equality operator) of a function call, calling the value of the name `len`, with argument an attribute lookup of `keywords` on the name `node` and a numeric constant with value 1. The body of the main branch is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the literal string *', and keyword argument'*.

The other ('else') branch of the `if` statement is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the literal string *', and keyword arguments'*.

The `if` statement ends here.

A for loop, where the name `kw` iterates over an attribute lookup of `keywords` on the name `node`.The body of the loop is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *', assigning {} as '{}''*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `value` on the name `kw`, and an attribute lookup of `arg` on the name `kw`.

The for loop ends here.

The `if` statement ends here.

A return statement, returning the value of the name `s`.

The function Call ends here.

Return

A definition of a function named `Return`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'A return statement, returning the value of {}.'*, with argument a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `value` on the name `node`.

The function Return ends here.

Str

A definition of a function named `Str`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'the literal string *'{}'*'*, with argument a function call, calling the value of the name `escape_string`, with argument an attribute lookup of `s` on the name `node`.

The function Str ends here.

Attribute

A definition of a function named `Attribute`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'an attribute lookup of '{}' on {}'*, with positional arguments an attribute lookup of `attr` on the name `node`, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `value` on the name `node`.

The function Attribute ends here.

Expr

A definition of a function named `Expr`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'A bare expression with value {}.'*, with argument a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `value` on the name `node`.

The function Expr ends here.

BinOp

A definition of a function named `BinOp`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'{}, with left hand side {}, and right hand side {}'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `op` on the name `node`, a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `left` on the name `node`, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `right` on the name `node`.

The function BinOp ends here.

If

A definition of a function named `If`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An assignment to the name `s`, of the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'An 'if' statement, testing {}. The body of the main branch is as follows:\n\n'*, with argument a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `test` on the name `node`.

A for loop, where the name `nod` iterates over an attribute lookup of `body` on the name `node`.The body of the loop is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the addition (or concatenation) operator, with left hand side a function call, calling the value of the name `describe_node`, with argument the name `nod`, and right hand side the literal string '\n\n'.

The for loop ends here.

An `if` statement, testing an attribute lookup of `orelse` on the name `node`. The body of the main branch is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the literal string *'The other ('else') branch of the 'if' statement is as follows:*\n\n'.

A for loop, where the name `nod` iterates over an attribute lookup of `orelse` on the name `node`.The body of the loop is as follows:

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the addition (or concatenation) operator, with left hand side a function call, calling the value of the name `describe_node`, with argument the name `nod`, and right hand side the literal string '\n\n'.

The for loop ends here.

The `if` statement ends here.

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of the literal string *'The 'if' statement ends here.*\n\n'.

A return statement, returning the value of the name `s`.

The function If ends here.

Num

A definition of a function named `Num`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'a numeric constant with value {}'*, with argument an attribute lookup of `n` on the name `node`.

The function Num ends here.

Compare

A definition of a function named `Compare`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An `if` statement, testing a comparison (using the equality operator) of a function call, calling the value of the name `len`, with argument an attribute lookup of `ops` on the name `node` and a numeric constant with value 1. The body of the main branch is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'a comparison (using {}) of {} and {}'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `ops` on the name `node`, subscripted by a numeric constant with value 0, a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `left` on the name `node`, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `comparators` on the name `node`, subscripted by a numeric constant with value 0.

The other ('else') branch of the `if` statement is as follows:

An assignment to the name `lefts`, of the value of the addition (or concatenation) operator, with left hand side a list containing an attribute lookup of `left` on the name `node`, and right hand side an attribute lookup of `comparators` on the name `node`, subscripted by a slice up to the unary negation operator applied to a numeric constant with value 1.

An assignment to the name `rights`, of the value of an attribute lookup of `comparators` on the name `node`.

An assignment to the name `s`, of the value of the literal string *'a compound comparison, comparing'*.

A modifying assignment to the name `s`, using the addition (or concatenation) operator, of the value of a function call, calling the value of the name `as_list`, with argument a generator expression, taking the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'{} and {} using {}'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument the name `left`, a function call, calling the value of the name `describe_node`, with argument the name `right`, and a function call, calling the value of the name `describe_node`, with argument the name `op`, as a tuple containing the name `left`, the name `op`, and the name `right` ranges over a function call, calling the value of the name `zip`, with positional arguments the name `lefts`, an attribute lookup of `ops` on the name `node`, and the name `rights`.

A return statement, returning the value of the name `s`.

The `if` statement ends here.

The function Compare ends here.

Eq

A definition of a function named `Eq`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'the equality operator'*.

The function Eq ends here.

GtE

A definition of a function named `GtE`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'the 'greater than or equal to' operator'*.

The function GtE ends here.

LtE

A definition of a function named `LtE`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'the 'less than or equal to' operator'*.

The function LtE ends here.

Gt

A definition of a function named `Gt`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'the 'greater than' operator'*.

The function Gt ends here.

Is

A definition of a function named `Is`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'the identity operator'*.

The function Is ends here.

UnaryOp

A definition of a function named `UnaryOp`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'{} applied to {}'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `op` on the name `node`, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `operand` on the name `node`.

The function UnaryOp ends here.

Not

A definition of a function named `Not`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'the unary 'not' operator'*.

The function Not ends here.

USub

A definition of a function named `USub`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'the unary negation operator'*.

The function USub ends here.

GeneratorExp

A definition of a function named `GeneratorExp`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An assignment to the name `gen`, of the value of an attribute lookup of `generators` on the name `node`, subscripted by a numeric constant with value 0.

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'a generator expression, taking the value of {}, as {} ranges over {}'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `elt` on the name `node`, a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `target` on the name `gen`, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `iter` on the name `gen`.

The function GeneratorExp ends here.

ListComp

A definition of a function named `ListComp`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An assignment to the name `gen`, of the value of an attribute lookup of `generators` on the name `node`, subscripted by a numeric constant with value 0.

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'a list comprehension, taking the value of {}, as {} ranges over {}'*, with positional arguments a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `elt` on the name `node`, a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `target` on the name `gen`, and a function call, calling the value of the name `describe_node`, with argument an attribute lookup of `iter` on the name `gen`.

The function ListComp ends here.

Assert

A definition of a function named `Assert`, with argument `node`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string ''.

The function Assert ends here.

LOAD_CONST

A definition of a function named `LOAD_CONST`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer places {} on top of the stack.'*, with argument a function call, calling the value of the name `describe_value`, with positional arguments an attribute lookup of `argval` on the name `op`, and the name `codes`.

The function LOAD_CONST ends here.

LOAD_NAME

A definition of a function named `LOAD_NAME`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer places the value associated with the name '{}' on top of the stack.'*, with argument an attribute lookup of `argval` on the name `op`.

The function LOAD_NAME ends here.

CALL_FUNCTION

A definition of a function named `CALL_FUNCTION`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An `if` statement, testing a comparison (using the equality operator) of an attribute lookup of `argval` on the name `op` and a numeric constant with value 0. The body of the main branch is as follows:

A return statement, returning the value of the literal string *'The computer takes the top value from the stack and calls it as a function (with no arguments), placing the return value on top of the stack.'*.

The other ('else') branch of the `if` statement is as follows:

An `if` statement, testing a comparison (using the equality operator) of an attribute lookup of `argval` on the name `op` and a numeric constant with value 1. The body of the main branch is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack.'*, with argument an attribute lookup of `argval` on the name `op`.

The other ('else') branch of the `if` statement is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes {} values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack.'*, with argument a function call, calling the value of the name `describe_number`, with argument an attribute lookup of `argval` on the name `op`.

The `if` statement ends here.

The `if` statement ends here.

The function CALL_FUNCTION ends here.

POP_TOP

A definition of a function named `POP_TOP`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer discards the top value from the stack.'*.

The function POP_TOP ends here.

RETURN_VALUE

A definition of a function named `RETURN_VALUE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer exits the current function, returning the top value on the stack.'*.

The function RETURN_VALUE ends here.

STORE_NAME

A definition of a function named `STORE_NAME`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top value from the stack, and stores it under the name '{}'.'*, with argument an attribute lookup of `argval` on the name `op`.

The function STORE_NAME ends here.

BINARY_SUBSCR

A definition of a function named `BINARY_SUBSCR`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item.'*.

The function BINARY_SUBSCR ends here.

LOAD_ATTR

A definition of a function named `LOAD_ATTR`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top value from the stack and retrieves its attribute named '{}', placing it on the stack.'*, with argument an attribute lookup of `argval` on the name `op`.

The function LOAD_ATTR ends here.

POP_JUMP_IF_FALSE

A definition of a function named `POP_JUMP_IF_FALSE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset {}.'*, with argument an attribute lookup of `argval` on the name `op`.

The function POP_JUMP_IF_FALSE ends here.

POP_JUMP_IF_TRUE

A definition of a function named `POP_JUMP_IF_TRUE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top value from the stack, and if it is true-like (e.g. True, non-empty or non-zero), jumps to offset {}.'*, with argument an attribute lookup of `argval` on the name `op`.

The function POP_JUMP_IF_TRUE ends here.

IMPORT_NAME

A definition of a function named `IMPORT_NAME`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top two values from the stack and uses them as the 'fromlist' and 'level' of an import for the module '{}', which is placed on the stack.'*, with argument an attribute lookup of `argval` on the name `op`.

The function IMPORT_NAME ends here.

MAKE_FUNCTION

A definition of a function named `MAKE_FUNCTION`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An assignment to the name `txt`, of the value of the literal string *'The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack.'*.

An `if` statement, testing the bitwise 'AND' operator, with left hand side an attribute lookup of `argval` on the name `op`, and right hand side a numeric constant with value 8. The body of the main branch is as follows:

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'It also takes the next value as a tuple of cells for free variables, creating a closure.'*.

The `if` statement ends here.

An `if` statement, testing the bitwise 'AND' operator, with left hand side an attribute lookup of `argval` on the name `op`, and right hand side a numeric constant with value 4. The body of the main branch is as follows:

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'It also takes the next value as a dictionary of function annotations.'*.

The `if` statement ends here.

An `if` statement, testing the bitwise 'AND' operator, with left hand side an attribute lookup of `argval` on the name `op`, and right hand side a numeric constant with value 2. The body of the main branch is as follows:

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'It also takes the next value as a dictionary of keyword arguments.'*.

The `if` statement ends here.

An `if` statement, testing the bitwise 'AND' operator, with left hand side an attribute lookup of `argval` on the name `op`, and right hand side a numeric constant with value 1. The body of the main branch is as follows:

A modifying assignment to the name `txt`, using the addition (or concatenation) operator, of the value of the literal string *'It also takes the next value as a tuple of default arguments.'*.

The `if` statement ends here.

A return statement, returning the value of the name `txt`.

The function MAKE_FUNCTION ends here.

COMPARE_OP

A definition of a function named `COMPARE_OP`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An `if` statement, testing a comparison (using the equality operator) of an attribute lookup of `argval` on the name `op` and the literal string *'=='*. The body of the main branch is as follows:

A return statement, returning the value of the literal string *'The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack.'*.

The other ('else') branch of the `if` statement is as follows:

An `if` statement, testing a comparison (using the equality operator) of an attribute lookup of `argval` on the name `op` and the literal string *'is'*. The body of the main branch is as follows:

A return statement, returning the value of the literal string *'The computer takes the top two values from the stack and compares them for identity, placing the result on top of the stack.'*.

The `if` statement ends here.

The `if` statement ends here.

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top two values from the stack and compares them using the operator '{}', placing the result on top of the stack.'*, with argument an attribute lookup of `argval` on the name `op`.

The function COMPARE_OP ends here.

BUILD_MAP

A definition of a function named `BUILD_MAP`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The

body of the function is as follows:

An `if` statement, testing a comparison (using the equality operator) of an attribute lookup of `argval` on the name `op` and a numeric constant with value 0. The body of the main branch is as follows:

A return statement, returning the value of the literal string *'The computer places an empty dictionary on top of the stack.'*.

The `if` statement ends here.

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top {} values from the stack, and uses them as key-value pairs in a new dictionary, which is placed on top of the stack.'*, with argument a function call, calling the value of the name `describe_number`, with argument the multiplication operator, with left hand side a numeric constant with value 2, and right hand side an attribute lookup of `argval` on the name `op`.

The function BUILD_MAP ends here.

EXTENDED_ARG

A definition of a function named `EXTENDED_ARG`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string ''.

The function EXTENDED_ARG ends here.

BINARY_ADD

A definition of a function named `BINARY_ADD`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top two values from the stack, adds them together, and places the result on top of the stack.'*.

The function BINARY_ADD ends here.

BINARY_MULTIPLY

A definition of a function named `BINARY_MULTIPLY`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top two values from the stack, multiplies them together, and places the result on top of the stack.'*.

The function BINARY_MULTIPLY ends here.

BINARY_AND

A definition of a function named `BINARY_AND`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top two values from the stack, applies a bitwise 'AND' operator to them, and places the result on top of the stack.'*.

The function BINARY_AND ends here.

BUILD_LIST

A definition of a function named `BUILD_LIST`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An `if` statement, testing a comparison (using the equality operator) of an attribute lookup of `argval` on the name `op` and a numeric constant with value 0. The body of the main branch is as follows:

A return statement, returning the value of the literal string *'The computer places a new empty list on top of the stack.'*.

The other ('else') branch of the `if` statement is as follows:

An `if` statement, testing a comparison (using the equality operator) of an attribute lookup of `argval` on the name `op` and a numeric constant with value 1. The body of the main branch is as follows:

A return statement, returning the value of the literal string *'The computer takes the top value from the stack, puts it in a list, and places it on top of the stack.'*.

The other ('else') branch of the `if` statement is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top {} values from the stack, puts them in a list, and places it on top of the stack.'*, with argument a function call, calling the value of the name `describe_number`, with argument an attribute lookup of `argval` on the name `op`.

The `if` statement ends here.

The `if` statement ends here.

The function BUILD_LIST ends here.

BUILD_SLICE

A definition of a function named `BUILD_SLICE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top two values from the stack, creates a slice object from them, and places it on top of the stack.'*.

The function BUILD_SLICE ends here.

BUILD_TUPLE

A definition of a function named `BUILD_TUPLE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

An `if` statement, testing a comparison (using the equality operator) of an attribute lookup of `argval` on the name `op` and a numeric constant with value 1. The body of the main branch is as follows:

A return statement, returning the value of the literal string *'The computer takes the top value from the stack, creates a tuple from it, and places it on top of the stack.'*.

The `if` statement ends here.

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top {} values from the stack, creates a tuple from them, and places it on top of the stack.'*, with argument a function call, calling the value of the name `describe_number`, with argument an attribute lookup of `argval` on the name `op`.

The function BUILD_TUPLE ends here.

FOR_ITER

A definition of a function named `FOR_ITER`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer looks at the top value on the stack and calls its 'next()' method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset {}.'*, with argument an attribute lookup of `argval` on the name `op`.

The function FOR_ITER ends here.

GET_ITER

A definition of a function named `GET_ITER`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top value from the stack, turns it into an iterator (using 'iter()'), and places the result on top of the stack.'*.

The function GET_ITER ends here.

INPLACE_ADD

A definition of a function named `INPLACE_ADD`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack.'*.

The function INPLACE_ADD ends here.

JUMP_ABSOLUTE

A definition of a function named `JUMP_ABSOLUTE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer jumps to offset {}.'*, with argument an attribute lookup of `argval` on the name `op`.

The function JUMP_ABSOLUTE ends here.

JUMP_FORWARD

A definition of a function named `JUMP_FORWARD`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer jumps forward to offset {}.'*, with argument an attribute lookup of `argval` on the name `op`.

The function JUMP_FORWARD ends here.

LIST_APPEND

A definition of a function named `LIST_APPEND`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top value from the stack and appends it to the list stored {} places from the top of the stack.'*, with argument a function call, calling the value of the name `describe_number`, with argument an attribute lookup of `argval` on the name `op`.

The function LIST_APPEND ends here.

LOAD_CLOSURE

A definition of a function named `LOAD_CLOSURE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer loads a reference to the free variable named '{}' and places it on top of the stack.'*, with argument an attribute lookup of `argval` on the name `op`.

The function LOAD_CLOSURE ends here.

LOAD_DEREF

A definition of a function named `LOAD_DEREF`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer loads the contents of the free variable named '{}' and places it on top of the stack.'*, with argument an attribute lookup of `argval` on the name `op`.

The function LOAD_DEREF ends here.

LOAD_FAST

A definition of a function named `LOAD_FAST`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer loads a reference to the local variable named '{}' and places it on top of the stack.'*, with argument an attribute lookup of `argval` on the name `op`.

The function LOAD_FAST ends here.

LOAD_GLOBAL

A definition of a function named `LOAD_GLOBAL`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer loads a reference to the global variable named '{}' and places it on top of the stack.'*, with argument an attribute lookup of `argval` on the name `op`.

The function LOAD_GLOBAL ends here.

POP_BLOCK

A definition of a function named `POP_BLOCK`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer removes one block from the block stack.'*.

The function POP_BLOCK ends here.

SETUP_LOOP

A definition of a function named `SETUP_LOOP`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer places a new block for a loop on top of the block stack, extending until offset {}.'*, with argument an attribute lookup of `argval` on the name `op`.

The function SETUP_LOOP ends here.

STORE_DEREF

A definition of a function named `STORE_DEREF`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top value from the stack and stores it in the free variable named '{}'.'*, with argument an attribute lookup of `argval` on the name `op`.

The function STORE_DEREF ends here.

STORE_FAST

A definition of a function named `STORE_FAST`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top value from the stack and stores it in the local variable named '{}'.'*, with argument an attribute lookup of `argval` on the name `op`.

The function STORE_FAST ends here.

STORE_SUBSCR

A definition of a function named `STORE_SUBSCR`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top value from the stack, uses it to index into the next-from-top value, and stores the value below that in that location.'*.

The function STORE_SUBSCR ends here.

UNPACK_SEQUENCE

A definition of a function named `UNPACK_SEQUENCE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top value from the stack, unpacks it into {} values, then places them each on top of the stack.'*, with argument a function call, calling the value of the name `describe_number`, with argument an attribute lookup of `argval` on the name `op`.

The function UNPACK_SEQUENCE ends here.

YIELD_VALUE

A definition of a function named `YIELD_VALUE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top value from the stack and yields it from the current generator.'*.

The function YIELD_VALUE ends here.

CALL_FUNCTION_KW

A definition of a function named `CALL_FUNCTION_KW`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of a function call, calling the value of an attribute lookup of `format` on the literal string *'The computer takes the top value from the stack and interprets it as a tuple of keyword names. It then takes values from the top of the stack as corresponding values, followed by positional arguments up to a total of {} values (both keyword and positional). Then it takes the next value from the top of the stack and calls it as a function with these arguments, placing the return value on top of the stack.'*, with argument an attribute lookup of `argval` on the name `op`.

The function CALL_FUNCTION_KW ends here.

DUP_TOP

A definition of a function named `DUP_TOP`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer duplicates the top value on the stack, placing the new copy on top of the stack.'*.

The function DUP_TOP ends here.

ROT_TWO

A definition of a function named `ROT_TWO`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top two values from the stack, swaps them, and replaces them on top of the stack.'*.

The function ROT_TWO ends here.

ROT_THREE

A definition of a function named `ROT_THREE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top three values from the stack, rotates them so that the top value is now on the bottom, and replaces them on top of the stack.'*.

The function ROT_THREE ends here.

UNARY_NEGATIVE

A definition of a function named `UNARY_NEGATIVE`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer takes the top value from the stack, negates it, and places the result on top of the stack.'*.

The function UNARY_NEGATIVE ends here.

JUMP_IF_FALSE_OR_POP

A definition of a function named `JUMP_IF_FALSE_OR_POP`, with positional arguments `op`, and `codes`. The definition is decorated with the function `descriptor`. The body of the function is as follows:

A return statement, returning the value of the literal string *'The computer looks at the top value on the stack. If it is false-like (e.g. False, None or zero), it jumps to offset {}. Otherwise it removes the top value from the stack.'*.

The function JUMP_IF_FALSE_OR_POP ends here.

An `if` statement, testing a comparison (using the equality operator) of the name `__name__` and the literal string *'__main__'*. The body of the main branch is as follows:

An assignment to the name `outfile`, of the value of an attribute lookup of `argv` on the name `sys`, subscripted by a numeric constant with value 1.

An assignment to the name `filename`, of the value of the name `__file__`.

An `if` statement, testing a comparison (using the 'greater than' operator) of a function call, calling the value of the name `len`, with argument an attribute lookup of `argv` on the name `sys` and a numeric constant with value 2. The body of the main branch is as follows:

An assignment to the name `filename`, of the value of an attribute lookup of `argv` on the name `sys`, subscripted by a numeric constant with value 2.

The `if` statement ends here.

An assignment to the name `f`, of the value of a function call, calling the value of the name `open`, with positional arguments the name `outfile`, and the literal string *'w'*.

A bare expression with value a function call, calling the value of an attribute lookup of `write` on the name `f`, with argument a function call, calling the value of the name `describe_file`, with argument the name `filename`.

A bare expression with value a function call, calling the value of an attribute lookup of `close` on the name `f` with no positional arguments.

The `if` statement ends here.

Chapter 4

Bytecode

describe.py

The computer places the integer constant zero on top of the stack. The computer places the constant None on top of the stack. The computer takes the top two values from the stack and uses them as the 'fromlist' and 'level' of an import for the module `ast`, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `ast`.

The computer places the integer constant zero on top of the stack. The computer places the constant None on top of the stack. The computer takes the top two values from the stack and uses them as the 'fromlist' and 'level' of an import for the module `dis`, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `dis`.

The computer places the integer constant zero on top of the stack. The computer places the constant None on top of the stack. The computer takes the top two values from the stack and uses them as the 'fromlist' and 'level' of an import for the module `re`, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `re`.

The computer places the integer constant zero on top of the stack. The computer places the constant None on top of the stack. The computer takes the top two values from the stack and uses them as the 'fromlist' and 'level' of an import for the module `sys`, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `sys`.

The computer places the integer constant zero on top of the stack. The computer places the constant None on top of the stack. The computer takes the top two values from the stack and uses them as the 'fromlist' and 'level' of an import for the module `types`, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `types`.

The computer places the literal string *'The Program Which Generates This Book'* on top of the stack. The computer takes the top value from the stack, and stores it under the name `title`.

The computer places the literal string *'Martin O'Leary'* on top of the stack. The computer takes the top value from the stack, and stores it under the name `author`.

The computer places the literal string *'\nThis book describes a computer program which, when executed, generates this\nbook. The program is described in three ways:\n\nFirst, a source code listing is given, in the Python programming language. This\nis the form of the program which was typed by the author, in text form.\n\nSecond, an *abstract syntax tree* is described, which is the computer's\ninterpretation of the textual source code in terms of the language constructs\navailable in the Python programming language.\n\nFinally, the program is described in terms of *bytecode*, the computer's internal\nrepresentation of the source code, a sequence of unambiguous instructions which\ncan be executed to perform the computation described by the program.\n\nThe descriptions given in this book are*

generated by the program it describes, in\nconjunction with a Python interpreter, starting from the source code form. Both \nthe abstract syntax tree and the bytecode representation are somewhat unstable.\nDifferent versions of the Python interpreter may yield different abstract syntax\ntrees and different bytecode representations of the same program. This book was\ngenerated using 'Python {}'.\n' on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer places the value associated with the name `sys` on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `version`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `preface`.

The computer places the code object described under title_block on top of the stack. The computer places the literal string *'title_block'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `title_block`.

The computer places the code object described under describe_op on top of the stack. The computer places the literal string *'describe_op'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `describe_op`.

The computer places the code object described under describe_file on top of the stack. The computer places the literal string *'describe_file'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `describe_file`.

The computer places the code object described under describe_number on top of the stack. The computer places the literal string *'describe_number'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `describe_number`.

The computer places the code object described under as_list on top of the stack. The computer places the literal string *'as_list'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `as_list`.

The computer places the code object described under escape_string on top of the stack. The computer places the literal string *'escape_string'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `escape_string`.

The computer places the code object described under describe_value on top of the stack. The computer places the literal string *'describe_value'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `describe_value`.

The computer places the code object described under describe_node on top of the stack. The computer places the literal string *'describe_node'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `describe_node`.

The computer places an empty dictionary on top of the stack. The computer takes the top value from the stack, and stores it under the name `descriptors`.

The computer places the code object described under descriptor on top of the stack. The computer places the literal string *'descriptor'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, and stores it under the name `descriptor`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Module on top of the stack. The computer places the literal string *'Module'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Module`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Import on top of the stack. The computer places the literal string *'Import'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Import`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Assign on top of the stack. The computer places the literal string *'Assign'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Assign`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under AugAssign on top of the stack. The computer places the literal string *'AugAssign'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `AugAssign`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Add on top of the stack. The computer places the literal string *'Add'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Add`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Mult on top of the stack. The computer places the literal string *'Mult'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on

the stack. The computer takes the top value from the stack, and stores it under the name `Mult`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under BitAnd on top of the stack. The computer places the literal string *'BitAnd'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `BitAnd`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Subscript on top of the stack. The computer places the literal string *'Subscript'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Subscript`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Index on top of the stack. The computer places the literal string *'Index'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Index`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Slice on top of the stack. The computer places the literal string *'Slice'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Slice`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under For on top of the stack. The computer places the literal string *'For'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `For`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under While on top of the stack. The computer places the literal string *'While'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `While`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Continue on top of the stack. The computer places the literal string *'Continue'* on top of the stack. The computer takes the top two values from the stack and uses them as

the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Continue`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Name on top of the stack. The computer places the literal string *'Name'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Name`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under NameConstant on top of the stack. The computer places the literal string *'NameConstant'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `NameConstant`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under List on top of the stack. The computer places the literal string *'List'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `List`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Tuple on top of the stack. The computer places the literal string *'Tuple'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Tuple`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Dict on top of the stack. The computer places the literal string *'Dict'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Dict`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under FunctionDef on top of the stack. The computer places the literal string *'FunctionDef'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `FunctionDef`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Call on top of the stack. The computer places the literal string *'Call'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Call`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Return on top of the stack. The computer places the literal string *'Return'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Return`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Str on top of the stack. The computer places the literal string *'Str'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Str`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Attribute on top of the stack. The computer places the literal string *'Attribute'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Attribute`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Expr on top of the stack. The computer places the literal string *'Expr'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Expr`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under BinOp on top of the stack. The computer places the literal string *'BinOp'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `BinOp`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under If on top of the stack. The computer places the literal string *'If'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a

function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `If`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Num on top of the stack. The computer places the literal string *'Num'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Num`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Compare on top of the stack. The computer places the literal string *'Compare'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Compare`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Eq on top of the stack. The computer places the literal string *'Eq'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Eq`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under GtE on top of the stack. The computer places the literal string *'GtE'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `GtE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under LtE on top of the stack. The computer places the literal string *'LtE'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `LtE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Gt on top of the stack. The computer places the literal string *'Gt'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Gt`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Is on top of the stack. The computer places the literal string *'Is'* on top of the stack. The

computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Is`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under UnaryOp on top of the stack. The computer places the literal string *'UnaryOp'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `UnaryOp`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Not on top of the stack. The computer places the literal string *'Not'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Not`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under USub on top of the stack. The computer places the literal string *'USub'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `USub`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under GeneratorExp on top of the stack. The computer places the literal string *'GeneratorExp'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `GeneratorExp`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under ListComp on top of the stack. The computer places the literal string *'ListComp'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `ListComp`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under Assert on top of the stack. The computer places the literal string *'Assert'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `Assert`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under LOAD_CONST on top of the stack. The computer places the literal string *'LOAD_CONST'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `LOAD_CONST`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under LOAD_NAME on top of the stack. The computer places the literal string *'LOAD_NAME'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `LOAD_NAME`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under CALL_FUNCTION on top of the stack. The computer places the literal string *'CALL_FUNCTION'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `CALL_FUNCTION`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under POP_TOP on top of the stack. The computer places the literal string *'POP_TOP'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `POP_TOP`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under RETURN_VALUE on top of the stack. The computer places the literal string *'RETURN_VALUE'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `RETURN_VALUE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under STORE_NAME on top of the stack. The computer places the literal string *'STORE_NAME'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `STORE_NAME`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under BINARY_SUBSCR on top of the stack. The computer places the literal string *'BINARY_SUBSCR'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack,

along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `BINARY_SUBSCR`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under LOAD_ATTR on top of the stack. The computer places the literal string *'LOAD_ATTR'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `LOAD_ATTR`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under POP_JUMP_IF_FALSE on top of the stack. The computer places the literal string *'POP_JUMP_IF_FALSE'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `POP_JUMP_IF_FALSE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under POP_JUMP_IF_TRUE on top of the stack. The computer places the literal string *'POP_JUMP_IF_TRUE'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `POP_JUMP_IF_TRUE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under IMPORT_NAME on top of the stack. The computer places the literal string *'IMPORT_NAME'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `IMPORT_NAME`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under MAKE_FUNCTION on top of the stack. The computer places the literal string *'MAKE_FUNCTION'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `MAKE_FUNCTION`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under COMPARE_OP on top of the stack. The computer places the literal string *'COMPARE_OP'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `COMPARE_OP`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under BUILD_MAP

on top of the stack. The computer places the literal string *'BUILD_MAP'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `BUILD_MAP`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under EXTENDED_ARG on top of the stack. The computer places the literal string *'EXTENDED_ARG'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `EXTENDED_ARG`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under BINARY_ADD on top of the stack. The computer places the literal string *'BINARY_ADD'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `BINARY_ADD`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under BINARY_MULTIPLY on top of the stack. The computer places the literal string *'BINARY_MULTIPLY'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `BINARY_MULTIPLY`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under BINARY_AND on top of the stack. The computer places the literal string *'BINARY_AND'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `BINARY_AND`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under BUILD_LIST on top of the stack. The computer places the literal string *'BUILD_LIST'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `BUILD_LIST`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under BUILD_SLICE on top of the stack. The computer places the literal string *'BUILD_SLICE'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the

return value on the stack. The computer takes the top value from the stack, and stores it under the name `BUILD_SLICE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under BUILD_TUPLE on top of the stack. The computer places the literal string *'BUILD_TUPLE'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `BUILD_TUPLE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under FOR_ITER on top of the stack. The computer places the literal string *'FOR_ITER'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `FOR_ITER`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under GET_ITER on top of the stack. The computer places the literal string *'GET_ITER'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `GET_ITER`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under INPLACE_ADD on top of the stack. The computer places the literal string *'INPLACE_ADD'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `INPLACE_ADD`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under JUMP_ABSOLUTE on top of the stack. The computer places the literal string *'JUMP_ABSOLUTE'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `JUMP_ABSOLUTE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under JUMP_FORWARD on top of the stack. The computer places the literal string *'JUMP_FORWARD'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `JUMP_FORWARD`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under LIST_APPEND on top of the stack. The computer places the literal string *'LIST_APPEND'* on top of the stack. The computer takes the top two values from the stack and uses

them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `LIST_APPEND`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under LOAD_CLOSURE on top of the stack. The computer places the literal string '*LOAD_CLOSURE*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `LOAD_CLOSURE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under LOAD_DEREF on top of the stack. The computer places the literal string '*LOAD_DEREF*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `LOAD_DEREF`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under LOAD_FAST on top of the stack. The computer places the literal string '*LOAD_FAST*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `LOAD_FAST`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under LOAD_GLOBAL on top of the stack. The computer places the literal string '*LOAD_GLOBAL*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `LOAD_GLOBAL`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under POP_BLOCK on top of the stack. The computer places the literal string '*POP_BLOCK*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `POP_BLOCK`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under SETUP_LOOP on top of the stack. The computer places the literal string '*SETUP_LOOP*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `SETUP_LOOP`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under STORE_DEREF on top of the stack. The computer places the literal string '*STORE_DEREF*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `STORE_DEREF`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under STORE_FAST on top of the stack. The computer places the literal string '*STORE_FAST*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `STORE_FAST`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under STORE_SUBSCR on top of the stack. The computer places the literal string '*STORE_SUBSCR*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `STORE_SUBSCR`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under UNPACK_SEQUENCE on top of the stack. The computer places the literal string '*UNPACK_SEQUENCE*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `UNPACK_SEQUENCE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under YIELD_VALUE on top of the stack. The computer places the literal string '*YIELD_VALUE*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `YIELD_VALUE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under CALL_FUNCTION_KW on top of the stack. The computer places the literal string '*CALL_FUNCTION_KW*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `CALL_FUNCTION_KW`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under DUP_TOP on top of the stack. The computer places the literal string '*DUP_TOP*' on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The

computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `DUP_TOP`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under ROT_TWO on top of the stack. The computer places the literal string *'ROT_TWO'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `ROT_TWO`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under ROT_THREE on top of the stack. The computer places the literal string *'ROT_THREE'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `ROT_THREE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under UNARY_NEGATIVE on top of the stack. The computer places the literal string *'UNARY_NEGATIVE'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `UNARY_NEGATIVE`.

The computer places the value associated with the name `descriptor` on top of the stack. The computer places the code object described under JUMP_IF_FALSE_OR_POP on top of the stack. The computer places the literal string *'JUMP_IF_FALSE_OR_POP'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `JUMP_IF_FALSE_OR_POP`.

The computer places the value associated with the name `__name__` on top of the stack. The computer places the literal string *'__main__'* on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 1214.

The computer places the value associated with the name `sys` on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argv`, placing it on the stack. The computer places the integer constant one on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack, and stores it under the name `outfile`.

The computer places the value associated with the name `__file__` on top of the stack. The computer takes the top value from the stack, and stores it under the name `filename`.

The computer places the value associated with the name `len` on top of the stack. The computer places the value associated with the name `sys` on top of the stack. The computer takes the top value from the stack and retrieves its attribute

named `argv`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the integer constant two on top of the stack. The computer takes the top two values from the stack and compares them using the operator `>`, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 1182.

The computer places the value associated with the name `sys` on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argv`, placing it on the stack. The computer places the integer constant two on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack, and stores it under the name `filename`.

Offset 1182

The computer places the value associated with the name `open` on top of the stack. The computer places the value associated with the name `outfile` on top of the stack. The computer places the literal string *'w'* on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack, and stores it under the name `f`.

The computer places the value associated with the name `f` on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `write`, placing it on the stack. The computer places the value associated with the name `describe_file` on top of the stack. The computer places the value associated with the name `filename` on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer discards the top value from the stack.

The computer places the value associated with the name `f` on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `close`, placing it on the stack. The computer takes the top value from the stack and calls it as a function (with no arguments), placing the return value on top of the stack. The computer discards the top value from the stack.

Offset 1214

The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

title_block

The computer places the literal string *'% {}\n% {}\n'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the global variable named `title` and places it on top of the stack. The computer loads a reference to the global variable named `author` and places it on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

describe_op

The computer loads a reference to the global variable named `descriptors` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `get`, placing it on the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `opname`, placing it on the stack. The computer places the constant None on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `f`.

The computer loads a reference to the local variable named `f` and places it on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 30.

The computer loads a reference to the local variable named `f` and places it on top of the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer loads a reference to the local variable named `codes` and places it on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps forward to offset 34.

Offset 30

The computer places the literal string ″ on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

Offset 34

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `is_jump_target`, placing it on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 56.

The computer places the literal string *'\n\n### Offset {}\n\n'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `offset`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

Offset 56

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

describe_file

The computer loads a reference to the global variable named `open` and places it on top of the stack. The computer loads a reference to the local variable named

`filename` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and retrieves its attribute named `read`, placing it on the stack. The computer takes the top value from the stack and calls it as a function (with no arguments), placing the return value on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `codetxt`.

The computer loads a reference to the global variable named `title_block` and places it on top of the stack. The computer takes the top value from the stack and calls it as a function (with no arguments), placing the return value on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string *'# About this book\n\n'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer loads a reference to the global variable named `preface` and places it on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string *'\n\n## License\n\n'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer loads a reference to the global variable named `open` and places it on top of the stack. The computer places the literal string *'LICENSE.md'* on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and retrieves its attribute named `read`, placing it on the stack. The computer takes the top value from the stack and calls it as a function (with no arguments), placing the return value on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string *'\n\n# Source code\n\n'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string *"'\n"* on top of the stack. The computer loads a reference to the local variable named `codetxt` and places it on top of the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer places the literal string *'\n"'\n\n'* on top of the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string *'# Abstract syntax tree\n\n'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the global variable named `ast` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `parse`, placing it on the stack. The computer loads a reference to the local variable named `codetxt` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string *'\n\n# Bytecode\n\n'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `filename` and places it on top of the stack. The computer loads a reference to the global variable named `compile` and places it on top of the stack. The computer loads a reference to the local variable named `codetxt` and places it on top of the stack. The computer loads a reference to the local variable named `filename` and places it on top of the stack. The computer places the literal string *'exec'* on top of the stack. The computer places the integer constant one on top of the stack. The computer places the tuple consisting of the literal string *'optimize'* on top of the stack. The computer takes the top value from the stack and interprets it as a tuple of keyword names. It then takes values from the top of the stack as corresponding values, followed by positional arguments up to a total of 4 values (both keyword and positional). Then it takes the next value from the top of the stack and calls it as a function with these arguments, placing the return value on top of the stack. The computer takes the top two values from the stack, creates a tuple from them, and places it on top of the stack. The computer takes the top value from the stack, puts it in a list, and places it on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `codes`.

The computer places a new block for a loop on top of the block stack, extending until offset 246.

Offset 140

The computer loads a reference to the local variable named `codes` and places it on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 244.

The computer loads a reference to the local variable named `codes` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `pop`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, unpacks it into two values, then

places them each on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `name`. The computer takes the top value from the stack and stores it in the local variable named `code`.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string '## {}' on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `name` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer places a new block for a loop on top of the block stack, extending until offset 234. The computer loads a reference to the global variable named `dis` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `get_instructions`, placing it on the stack. The computer loads a reference to the local variable named `code` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack.

Offset 184

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 232. The computer takes the top value from the stack and stores it in the local variable named `op`.

The computer loads a reference to the global variable named `describe_op` and places it on top of the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer loads a reference to the local variable named `codes` and places it on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `desc`.

The computer loads a reference to the local variable named `desc` and places it on top of the stack. The computer takes the top value from the stack, and if it is true-like (e.g. True, non-empty or non-zero), jumps to offset 204. The computer jumps to offset 184.

Offset 204

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `starts_line`, placing it on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 218.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string '\n\n' on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

Offset 218

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer loads a reference to the local variable named `desc` and places it on top of the stack. The computer places the literal string '' on top of the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`. The computer jumps to offset 184.

Offset 232

The computer removes one block from the block stack.

Offset 234

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string '\n\n' on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`. The computer jumps to offset 140.

Offset 244

The computer removes one block from the block stack.

Offset 246

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

describe_number

The computer places the literal string *'zero'* on top of the stack. The computer places the literal string *'one'* on top of the stack. The computer places the literal string *'two'* on top of the stack. The computer places the literal string *'three'* on top of the stack. The computer places the literal string *'four'* on top of the stack. The computer places the literal string *'five'* on top of the stack. The computer places the literal string *'six'* on top of the stack. The computer places the literal string *'seven'* on top of the stack. The computer places the literal string *'eight'* on top of the stack.

The computer places the literal string *'nine'* on top of the stack. The computer places the literal string *'ten'* on top of the stack. The computer takes the top minus zero values from the stack, puts them in a list, and places it on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `words`.

The computer places the integer constant zero on top of the stack. The computer loads a reference to the local variable named `num` and places it on top of the stack. The computer duplicates the top value on the stack, placing the new copy on top of the stack. The computer takes the top three values from the stack, rotates them so that the top value is now on the bottom, and replaces them on top of the stack. The computer takes the top two values from the stack and compares them using the operator <=, placing the result on top of the stack. The computer looks at the top value on the stack. If it is false-like (e.g. False, None or zero), it jumps to offset {}. Otherwise it removes the top value from the stack.

The computer places the integer constant ten on top of the stack. The computer takes the top two values from the stack and compares them using the operator `<=`, placing the result on top of the stack. The computer jumps forward to offset 48.

Offset 44

The computer takes the top two values from the stack, swaps them, and replaces them on top of the stack. The computer discards the top value from the stack.

Offset 48

The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 58.

The computer loads a reference to the local variable named `words` and places it on top of the stack. The computer loads a reference to the local variable named `num` and places it on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer exits the current function, returning the top value on the stack.

Offset 58

The computer loads a reference to the local variable named `num` and places it on top of the stack. The computer places the integer constant minus ten on top of the stack. The computer takes the top two values from the stack and compares them using the operator `>=`, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 80.

The computer places the literal string *'minus'* on top of the stack. The computer loads a reference to the local variable named `words` and places it on top of the stack. The computer loads a reference to the local variable named `num` and places it on top of the stack. The computer takes the top value from the stack, negates it, and places the result on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 80

The computer loads a reference to the global variable named `str` and places it on top of the stack. The computer loads a reference to the local variable named `num` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

as_list

The computer loads a reference to the global variable named `list` and places it on top of the stack. The computer loads a reference to the local variable named `items` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `items`.

The computer loads a reference to the global variable named `len` and places it on top of the stack. The computer loads a reference to the local variable named `items` and places it on top of the stack. The computer takes the top value from

the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the integer constant one on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 28.

The computer loads a reference to the local variable named `items` and places it on top of the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer exits the current function, returning the top value on the stack.

Offset 28

The computer places the literal string *','* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `join`, placing it on the stack. The computer loads a reference to the local variable named `items` and places it on top of the stack. The computer places the constant None on top of the stack. The computer places the integer constant minus one on top of the stack. The computer takes the top two values from the stack, creates a slice object from them, and places it on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the literal string *', and'* on top of the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer loads a reference to the local variable named `items` and places it on top of the stack. The computer places the integer constant minus one on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer exits the current function, returning the top value on the stack. The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

escape_string

The computer loads a reference to the global variable named `re` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `sub`, placing it on the stack. The computer places the literal string *'([_'*\\#])'* on top of the stack. The computer places the literal string *'\\\1'* on top of the stack. The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer takes three values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the global variable named `re` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `sub`, placing it on the stack. The computer places the literal string *'\n'* on top of the stack. The computer places the literal string *'\\\\n'* on top of the stack. The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer takes three values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

describe_value

The computer loads a reference to the global variable named `isinstance` and places it on top of the stack. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer loads a reference to the global variable named `types` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `CodeType`, placing it on the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 80.

The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `co_name`, placing it on the stack. The computer takes the top value from the stack and stores it in the local variable named `name`.

The computer loads a reference to the local variable named `name` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `startswith`, placing it on the stack. The computer places the literal string '<' on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 56.

The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `co_name`, placing it on the stack. The computer places the integer constant one on top of the stack. The computer places the integer constant minus one on top of the stack. The computer takes the top two values from the stack, creates a slice object from them, and places it on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer places the literal string ':' on top of the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer loads a reference to the global variable named `str` and places it on top of the stack. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `co_firstlineno`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `name`.

Offset 56

The computer loads the contents of the free variable named `codes` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `append`, placing it on the stack. The computer loads a reference to the local variable named `name` and places it on top of the stack. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer takes the top two values from the stack, creates a tuple from them, and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer discards the top value from the stack.

The computer places the literal string '*the code object described under {}*' on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `name` and places it on top of the stack. The computer

takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Offset 80

The computer loads a reference to the global variable named `isinstance` and places it on top of the stack. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer loads a reference to the global variable named `str` and places it on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 104.

The computer places the literal string *'the literal string *'{}'*'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the global variable named `escape_string` and places it on top of the stack. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Offset 104

The computer loads a reference to the global variable named `isinstance` and places it on top of the stack. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer loads a reference to the global variable named `int` and places it on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 128.

The computer places the literal string *'the integer constant {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the global variable named `describe_number` and places it on top of the stack. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Offset 128

The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer places the constant None on top of the stack. The computer takes the top two values from the stack and compares them for identity, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 140.

The computer places the literal string *'the constant None'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 140

The computer loads a reference to the global variable named `isinstance` and places it on top of the stack. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer loads a reference to the global variable named `tuple` and places it on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 176.

The computer places the literal string *'the tuple consisting of'* on top of the stack. The computer loads a reference to the global variable named `as_list` and places it on top of the stack.

The computer loads a reference to the free variable named `codes` and places it on top of the stack. The computer takes the top value from the stack, creates a tuple from it, and places it on top of the stack. The computer places the code object described under genexpr:117 on top of the stack. The computer places the literal string *'describe_value..'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. It also takes the next value as a tuple of cells for free variables, creating a closure. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 176

The computer loads a reference to the global variable named `print` and places it on top of the stack. The computer places the literal string *'Uninterpretable constant:'* on top of the stack. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer discards the top value from the stack.

The computer loads a reference to the global variable named `repr` and places it on top of the stack. The computer loads a reference to the local variable named `value` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

describe_node

The computer loads a reference to the global variable named `descriptors` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `get`, placing it on the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `__class__`, placing it on the stack. The computer takes the top value from the stack and retrieves its attribute named `__name__`, placing it on the stack. The computer places the constant None on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function,

using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `f`.

The computer loads a reference to the local variable named `f` and places it on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 28.

The computer loads a reference to the local variable named `f` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Offset 28

The computer loads a reference to the global variable named `print` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `_fields`, placing it on the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer discards the top value from the stack.

The computer loads a reference to the global variable named `str` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack. The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

descriptor

The computer loads a reference to the local variable named `f` and places it on top of the stack. The computer loads a reference to the global variable named `descriptors` and places it on top of the stack. The computer loads a reference to the local variable named `f` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `__name__`, placing it on the stack. The computer takes the top value from the stack, uses it to index into the next-from-top value, and stores the value below that in that location.

The computer loads a reference to the local variable named `f` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

Module

The computer places the literal string *'A module, containing the following code:\n\n'* on top of the stack. The computer places the literal string *'\n\n'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `join`, placing it on the stack.

The computer places the code object described under genexpr:143 on top of the stack. The computer places the literal string *'Module..'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer loads a reference to the local variable named `node` and places it on top of the

stack. The computer takes the top value from the stack and retrieves its attribute named `body`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer exits the current function, returning the top value on the stack.

Import

The computer places the literal string *'An import statement for a module named '{}'.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `names`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack and retrieves its attribute named `name`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Assign

The computer places the literal string *'An assignment to {}, of the value of {}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `targets`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `value`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

AugAssign

The computer places the literal string *'A modifying assignment to {}, using {}, of the value of {}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `target`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `op`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `value`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes three values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

Add

The computer places the literal string *'the addition (or concatenation) operator'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Mult

The computer places the literal string *'the multiplication operator'* on top of the stack. The computer exits the current function, returning the top value on the stack.

BitAnd

The computer places the literal string *'the bitwise 'AND' operator'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Subscript

The computer places the literal string *'{}, subscripted by {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `value`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `slice`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Index

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `value`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Slice

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `lower`, placing it on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 30.

The computer places the literal string *'a slice from {} to {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `lower`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `upper`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Offset 30

The computer places the literal string *'a slice up to {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `upper`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack. The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

For

The computer places the literal string *'A for loop, where {} iterates over {}.The body of the loop is as follows:\n\n'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `target`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `iter`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer places a new block for a loop on top of the block stack, extending until offset 56. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `body`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack.

Offset 32

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 54. The computer takes the top value from the stack and stores it in the local variable named `nod`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `nod` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the literal string *'\n\n'* on top of the

stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps to offset 32.

Offset 54

The computer removes one block from the block stack.

Offset 56

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *'The for loop ends here.'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

While

The computer places the literal string *'A while loop, testing {}.The body of the loop is as follows:\n\n'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `test`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer places a new block for a loop on top of the block stack, extending until offset 48. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `body`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack.

Offset 24

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 46. The computer takes the top value from the stack and stores it in the local variable named `nod`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `nod` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the literal string *'\n\n'* on top of the stack. The computer takes the top two values from the stack, adds them together,

and places the result on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps to offset 24.

Offset 46

The computer removes one block from the block stack.

Offset 48

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *'The while loop ends here.'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

Continue

The computer places the literal string *'A 'continue' statement.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Name

The computer places the literal string *'the name '{}''* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `id`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

NameConstant

The computer places the literal string *'the constant '{}''* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `value`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

List

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `elts`, placing it on the stack. The computer takes the top value

from the stack, and if it is true-like (e.g. True, non-empty or non-zero), jumps to offset 10.

The computer places the literal string *'an empty list'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 10

The computer places the literal string *'a list containing'* on top of the stack. The computer loads a reference to the global variable named `as_list` and places it on top of the stack.

The computer places the code object described under genexpr:245 on top of the stack. The computer places the literal string *'List..'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `elts`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer exits the current function, returning the top value on the stack. The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

Tuple

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `elts`, placing it on the stack. The computer takes the top value from the stack, and if it is true-like (e.g. True, non-empty or non-zero), jumps to offset 10.

The computer places the literal string *'an empty tuple'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 10

The computer places the literal string *'a tuple containing'* on top of the stack. The computer loads a reference to the global variable named `as_list` and places it on top of the stack.

The computer places the code object described under genexpr:254 on top of the stack. The computer places the literal string *'Tuple..'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `elts`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer exits the current function, returning the top value on the

stack. The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

Dict

The computer places the literal string *'an empty dictionary'* on top of the stack. The computer exits the current function, returning the top value on the stack.

FunctionDef

The computer places the literal string *'## {node.name}\n\nA definition of a function named '{node.name}''* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer places the tuple consisting of the literal string *'node'* on top of the stack. The computer takes the top value from the stack and interprets it as a tuple of keyword names. It then takes values from the top of the stack as corresponding values, followed by positional arguments up to a total of 1 values (both keyword and positional). Then it takes the next value from the top of the stack and calls it as a function with these arguments, placing the return value on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `args`, placing it on the stack. The computer takes the top value from the stack and stores it in the local variable named `args`.

The computer loads a reference to the global variable named `len` and places it on top of the stack. The computer loads a reference to the local variable named `args` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `args`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the integer constant one on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 56.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *', with argument '{}'.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `args` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `args`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack and retrieves its attribute named `arg`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps forward to offset 94.

Offset 56

The computer loads a reference to the local variable named `args` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `args`, placing it on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 94.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *', with positional arguments {args}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the global variable named `as_list` and places it on top of the stack.

The computer places the code object described under listcomp:272 on top of the stack. The computer places the literal string *'FunctionDef..'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer loads a reference to the local variable named `args` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `args`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the tuple consisting of the literal string *'args'* on top of the stack. The computer takes the top value from the stack and interprets it as a tuple of keyword names. It then takes values from the top of the stack as corresponding values, followed by positional arguments up to a total of 1 values (both keyword and positional). Then it takes the next value from the top of the stack and calls it as a function with these arguments, placing the return value on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

Offset 94

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `decorator_list`, placing it on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 122.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *'The definition is decorated with the function '{}'.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `decorator_list`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack and retrieves its attribute named `id`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The

computer takes the top value from the stack and stores it in the local variable named `s`.

Offset 122

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *'The body of the function is as follows:\n\n'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer places a new block for a loop on top of the block stack, extending until offset 162. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `body`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack.

Offset 138

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 160. The computer takes the top value from the stack and stores it in the local variable named `nod`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `nod` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the literal string *'\n\n'* on top of the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps to offset 138.

Offset 160

The computer removes one block from the block stack.

Offset 162

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *'The function {} ends here.\n\n'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `name`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

Call

The computer places the literal string *'a function call, calling the value of {f}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `func`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the tuple consisting of the literal string *'f'* on top of the stack. The computer takes the top value from the stack and interprets it as a tuple of keyword names. It then takes values from the top of the stack as corresponding values, followed by positional arguments up to a total of 1 values (both keyword and positional). Then it takes the next value from the top of the stack and calls it as a function with these arguments, placing the return value on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the global variable named `len` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `args`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the integer constant one on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 58.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *', with argument {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `args`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps forward to offset 106.

Offset 58

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `args`, placing it on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 98.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *', with positional arguments {args}'* on top of the stack. The computer takes the top value from the stack and

retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the global variable named `as_list` and places it on top of the stack.

The computer places the code object described under genexpr:292 on top of the stack. The computer places the literal string *'Call..'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `args`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the tuple consisting of the literal string *'args'* on top of the stack. The computer takes the top value from the stack and interprets it as a tuple of keyword names. It then takes values from the top of the stack as corresponding values, followed by positional arguments up to a total of 1 values (both keyword and positional). Then it takes the next value from the top of the stack and calls it as a function with these arguments, placing the return value on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps forward to offset 106.

Offset 98

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *'with no positional arguments'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

Offset 106

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `keywords`, placing it on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 184.

The computer loads a reference to the global variable named `len` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `keywords`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the integer constant one on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 136.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *', and keyword argument'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps forward to offset 144.

Offset 136

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *', and keyword arguments'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

Offset 144

The computer places a new block for a loop on top of the block stack, extending until offset 184. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `keywords`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack.

Offset 152

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 182. The computer takes the top value from the stack and stores it in the local variable named `kw`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *', assigning {} as '{}''* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `kw` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `value`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the local variable named `kw` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `arg`, placing it on the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps to offset 152.

Offset 182

The computer removes one block from the block stack.

Offset 184

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

Return

The computer places the literal string *'A return statement, returning the value of {}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `value`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Str

The computer places the literal string *'the literal string *'{}'*'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the global variable named `escape_string` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `s`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Attribute

The computer places the literal string *'an attribute lookup of '{}' on {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `attr`, placing it on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `value`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Expr

The computer places the literal string *'A bare expression with value {}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `value`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return

value on the stack. The computer exits the current function, returning the top value on the stack.

BinOp

The computer places the literal string *'{}, with left hand side {}, and right hand side {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `op`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `left`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `right`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes three values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

If

The computer places the literal string *'An 'if' statement, testing {}. The body of the main branch is as follows:\n\n'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `test`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer places a new block for a loop on top of the block stack, extending until offset 48. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `body`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack.

Offset 24

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 46. The computer takes the top value from the stack and stores it in the local variable named `nod`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `nod` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the literal string *'\n\n'* on top of the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps to offset 24.

Offset 46

The computer removes one block from the block stack.

Offset 48

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `orelse`, placing it on the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 94.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *'The other ('else') branch of the 'if' statement is as follows:\n\n'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer places a new block for a loop on top of the block stack, extending until offset 94. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `orelse`, placing it on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack.

Offset 70

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 92. The computer takes the top value from the stack and stores it in the local variable named `nod`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `nod` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the literal string *'\n\n'* on top of the stack. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it,

placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`. The computer jumps to offset 70.

Offset 92

The computer removes one block from the block stack.

Offset 94

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer places the literal string *'The 'if' statement ends here.\n\n'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

Num

The computer places the literal string *'a numeric constant with value {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `n`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Compare

The computer loads a reference to the global variable named `len` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `ops`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer places the integer constant one on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 54.

The computer places the literal string *'a comparison (using {}) of {} and {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `ops`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local

variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `left`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `comparators`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes three values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Offset 54

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `left`, placing it on the stack. The computer takes the top value from the stack, puts it in a list, and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `comparators`, placing it on the stack. The computer places the constant None on top of the stack. The computer places the integer constant minus one on top of the stack. The computer takes the top two values from the stack, creates a slice object from them, and places it on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top two values from the stack, adds them together, and places the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `lefts`.

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `comparators`, placing it on the stack. The computer takes the top value from the stack and stores it in the local variable named `rights`.

The computer places the literal string *'a compound comparison, comparing'* on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer loads a reference to the global variable named `as_list` and places it on top of the stack. The computer places the code object described under genexpr:365 on top of the stack. The computer places the literal string *'Compare..'* on top of the stack. The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack.

The computer loads a reference to the global variable named `zip` and places it on top of the stack. The computer loads a reference to the local variable named `lefts` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `ops`, placing it on the stack. The computer loads a reference to the local variable named `rights` and places it on top of the stack. The computer takes three values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack, turns it into an iterator (using `iter()`), and places the result on top of the stack. The computer takes the top value from the stack,

along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `s`.

The computer loads a reference to the local variable named `s` and places it on top of the stack. The computer exits the current function, returning the top value on the stack. The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

Eq

The computer places the literal string *'the equality operator'* on top of the stack. The computer exits the current function, returning the top value on the stack.

GtE

The computer places the literal string *'the 'greater than or equal to' operator'* on top of the stack. The computer exits the current function, returning the top value on the stack.

LtE

The computer places the literal string *'the 'less than or equal to' operator'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Gt

The computer places the literal string *'the 'greater than' operator'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Is

The computer places the literal string *'the identity operator'* on top of the stack. The computer exits the current function, returning the top value on the stack.

UnaryOp

The computer places the literal string *'{} applied to {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `op`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named

`operand`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Not

The computer places the literal string *'the unary 'not' operator'* on top of the stack. The computer exits the current function, returning the top value on the stack.

USub

The computer places the literal string *'the unary negation operator'* on top of the stack. The computer exits the current function, returning the top value on the stack.

GeneratorExp

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `generators`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack and stores it in the local variable named `gen`.

The computer places the literal string *'a generator expression, taking the value of {}, as {} ranges over {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `elt`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `gen` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `target`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `gen` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `iter`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes three values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

ListComp

The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `generators`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item. The computer takes the top value from the stack and stores it in the local variable named `gen`.

The computer places the literal string *'a list comprehension, taking the value of {}, as {} ranges over {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `node` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `elt`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack.

The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `gen` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `target`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `gen` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `iter`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes three values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Assert

The computer places the literal string '' on top of the stack. The computer exits the current function, returning the top value on the stack.

LOAD_CONST

The computer places the literal string *'The computer places {} on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_value` and places it on top of the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer loads a reference to the local variable named `codes` and places it on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

LOAD_NAME

The computer places the literal string *'The computer places the value associated with the name '{}' on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

CALL_FUNCTION

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 14.

The computer places the literal string *'The computer takes the top value from the stack and calls it as a function (with no arguments), placing the return value on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 14

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the integer constant one on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 36.

The computer places the literal string *'The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

Offset 36

The computer places the literal string *'The computer takes {} values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_number` and places it on top of the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the

stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack. The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

POP_TOP

The computer places the literal string *'The computer discards the top value from the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

RETURN_VALUE

The computer places the literal string *'The computer exits the current function, returning the top value on the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

STORE_NAME

The computer places the literal string *'The computer takes the top value from the stack, and stores it under the name '{}'.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

BINARY_SUBSCR

The computer places the literal string *'The computer takes the top two values from the stack and retrieves the value of the second item, subscripted by the value of the first item.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

LOAD_ATTR

The computer places the literal string *'The computer takes the top value from the stack and retrieves its attribute named '{}', placing it on the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

POP_JUMP_IF_FALSE

The computer places the literal string *'The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset {}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

POP_JUMP_IF_TRUE

The computer places the literal string *'The computer takes the top value from the stack, and if it is true-like (e.g. True, non-empty or non-zero), jumps to offset {}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

IMPORT_NAME

The computer places the literal string *'The computer takes the top two values from the stack and uses them as the 'fromlist' and 'level' of an import for the module '{}', which is placed on the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

MAKE_FUNCTION

The computer places the literal string *'The computer takes the top two values from the stack and uses them as the qualified name and code of a new function, which is placed on the stack.'* on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the integer constant eight on top of the stack. The computer takes the top two values from the stack, applies a bitwise `AND` operator to them, and places the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 22.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string *'It also takes the next value as a tuple of cells for free variables, creating a closure.'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

Offset 22

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the integer constant four on top of the stack. The computer takes the top two values from the stack, applies a bitwise `AND` operator to them, and places the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 40.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string *'It also takes the next value as a dictionary of function annotations.'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

Offset 40

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the integer constant two on top of the stack. The computer takes the top two values from the stack, applies a bitwise `AND` operator to them, and places the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 58.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string *'It also takes the next value as a dictionary of keyword arguments.'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

Offset 58

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the integer constant one on top of the stack. The computer takes the top two values from the stack, applies a bitwise `AND` operator to them, and places the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 76.

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer places the literal string *'It also takes the next value as a tuple of default arguments.'* on top of the stack. The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `txt`.

Offset 76

The computer loads a reference to the local variable named `txt` and places it on top of the stack. The computer exits the current function, returning the top value on the stack.

COMPARE_OP

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the

literal string '==' on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 14.

The computer places the literal string *'The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 14

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the literal string *'is'* on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 28.

The computer places the literal string *'The computer takes the top two values from the stack and compares them for identity, placing the result on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 28

The computer places the literal string *'The computer takes the top two values from the stack and compares them using the operator '{}', placing the result on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

BUILD_MAP

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 14.

The computer places the literal string *'The computer places an empty dictionary on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 14

The computer places the literal string *'The computer takes the top {} values from the stack, and uses them as key-value pairs in a new dictionary, which is placed on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_number` and places it on top of the stack. The computer places the integer constant two on top of the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the

stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top two values from the stack, multiplies them together, and places the result on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

EXTENDED_ARG

The computer places the literal string '' on top of the stack. The computer exits the current function, returning the top value on the stack.

BINARY_ADD

The computer places the literal string *'The computer takes the top two values from the stack, adds them together, and places the result on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

BINARY_MULTIPLY

The computer places the literal string *'The computer takes the top two values from the stack, multiplies them together, and places the result on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

BINARY_AND

The computer places the literal string *'The computer takes the top two values from the stack, applies a bitwise 'AND' operator to them, and places the result on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

BUILD_LIST

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the integer constant zero on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 14.

The computer places the literal string *'The computer places a new empty list on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 14

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the integer constant one on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the

stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 28.

The computer places the literal string *'The computer takes the top value from the stack, puts it in a list, and places it on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 28

The computer places the literal string *'The computer takes the top {} values from the stack, puts them in a list, and places it on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_number` and places it on top of the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack. The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

BUILD_SLICE

The computer places the literal string *'The computer takes the top two values from the stack, creates a slice object from them, and places it on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

BUILD_TUPLE

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer places the integer constant one on top of the stack. The computer takes the top two values from the stack and compares them for equality, placing the result on top of the stack. The computer takes the top value from the stack, and if it is false-like (e.g. False, None or zero), jumps to offset 14.

The computer places the literal string *'The computer takes the top value from the stack, creates a tuple from it, and places it on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

Offset 14

The computer places the literal string *'The computer takes the top {} values from the stack, creates a tuple from them, and places it on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_number` and places it on top of the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack,

along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

FOR_ITER

The computer places the literal string *'The computer looks at the top value on the stack and calls its 'next()' method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset {}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

GET_ITER

The computer places the literal string *'The computer takes the top value from the stack, turns it into an iterator (using 'iter()'), and places the result on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

INPLACE_ADD

The computer places the literal string *'The computer takes the top value from the stack and (in place)adds the second from top value from the stack to it, placing the result on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

JUMP_ABSOLUTE

The computer places the literal string *'The computer jumps to offset {}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

JUMP_FORWARD

The computer places the literal string *'The computer jumps forward to offset {}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

LIST_APPEND

The computer places the literal string *'The computer takes the top value from the stack and appends it to the list stored {} places from the top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_number` and places it on top of the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

LOAD_CLOSURE

The computer places the literal string *'The computer loads a reference to the free variable named '{}' and places it on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

LOAD_DEREF

The computer places the literal string *'The computer loads the contents of the free variable named '{}' and places it on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

LOAD_FAST

The computer places the literal string *'The computer loads a reference to the local variable named '{}' and places it on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

LOAD_GLOBAL

The computer places the literal string *'The computer loads a reference to the global variable named '{}' and places it on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

POP_BLOCK

The computer places the literal string *'The computer removes one block from the block stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

SETUP_LOOP

The computer places the literal string *'The computer places a new block for a loop on top of the block stack, extending until offset {}.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

STORE_DEREF

The computer places the literal string *'The computer takes the top value from the stack and stores it in the free variable named '{}'.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

STORE_FAST

The computer places the literal string *'The computer takes the top value from the stack and stores it in the local variable named '{}'.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

STORE_SUBSCR

The computer places the literal string *'The computer takes the top value from the stack, uses it to index into the next-from-top value, and stores the value below that in that location.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

UNPACK_SEQUENCE

The computer places the literal string *'The computer takes the top value from the stack, unpacks it into {} values, then places them each on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the global variable named `describe_number` and places it on top of the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

YIELD_VALUE

The computer places the literal string *'The computer takes the top value from the stack and yields it from the current generator.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

CALL_FUNCTION_KW

The computer places the literal string *'The computer takes the top value from the stack and interprets it as a tuple of keyword names. It then takes values from the top of the stack as corresponding values, followed by positional arguments up to a total of {} values (both keyword and positional). Then it takes the next value from the top of the stack and calls it as a function with these arguments, placing the return value on top of the stack.'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack.

The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `argval`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer exits the current function, returning the top value on the stack.

DUP_TOP

The computer places the literal string *'The computer duplicates the top value on the stack, placing the new copy on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

ROT_TWO

The computer places the literal string *'The computer takes the top two values from the stack, swaps them, and replaces them on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

ROT_THREE

The computer places the literal string *'The computer takes the top three values from the stack, rotates them so that the top value is now on the bottom, and replaces them on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

UNARY_NEGATIVE

The computer places the literal string *'The computer takes the top value from the stack, negates it, and places the result on top of the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

JUMP_IF_FALSE_OR_POP

The computer places the literal string *'The computer looks at the top value on the stack. If it is false-like (e.g. False, None or zero), it jumps to offset {}. Otherwise it removes the top value from the stack.'* on top of the stack. The computer exits the current function, returning the top value on the stack.

genexpr:117

The computer loads a reference to the local variable named `.0` and places it on top of the stack.

Offset 2

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 20. The computer takes the top value from the stack and stores it in the local variable named `x`. The computer loads a reference to the global variable named `describe_value` and places it on top of the stack. The computer loads a reference to the local variable named `x` and places it on top of the stack. The computer loads the contents of the free variable named `codes` and places it on top of the stack. The computer takes two values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and yields it from the current generator. The computer discards the top value from the stack. The computer jumps to offset 2.

Offset 20

The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

genexpr:143

The computer loads a reference to the local variable named `.0` and places it on top of the stack.

Offset 2

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 18. The computer takes the top value from the stack and stores it in the local variable named `n`. The computer loads a reference to the global variable named `describe_node` and places it on top

of the stack. The computer loads a reference to the local variable named `n` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and yields it from the current generator. The computer discards the top value from the stack. The computer jumps to offset 2.

Offset 18

The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

genexpr:245

The computer loads a reference to the local variable named `.0` and places it on top of the stack.

Offset 2

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 18. The computer takes the top value from the stack and stores it in the local variable named `elt`. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `elt` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and yields it from the current generator. The computer discards the top value from the stack. The computer jumps to offset 2.

Offset 18

The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

genexpr:254

The computer loads a reference to the local variable named `.0` and places it on top of the stack.

Offset 2

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 18. The computer takes the top value from the stack and stores it in the local variable named `elt`. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `elt` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and yields it from the current generator. The computer discards the top value from the stack. The computer jumps to offset 2.

Offset 18

The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

listcomp:272

The computer places a new empty list on top of the stack. The computer loads a reference to the local variable named `.0` and places it on top of the stack.

Offset 4

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 22. The computer takes the top value from the stack and stores it in the local variable named `a`. The computer places the literal string *"{}"* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the local variable named `a` and places it on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `arg`, placing it on the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and appends it to the list stored two places from the top of the stack. The computer jumps to offset 4.

Offset 22

The computer exits the current function, returning the top value on the stack.

genexpr:292

The computer loads a reference to the local variable named `.0` and places it on top of the stack.

Offset 2

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 18. The computer takes the top value from the stack and stores it in the local variable named `a`. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `a` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes the top value from the stack and yields it from the current generator. The computer discards the top value from the stack. The computer jumps to offset 2.

Offset 18

The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.

genexpr:365

The computer loads a reference to the local variable named `.0` and places it on top of the stack.

Offset 2

The computer looks at the top value on the stack and calls its `next()` method. If it returns a value, it places it on top of the stack. If not, it removes the top value from the stack and jumps to offset 42.

The computer takes the top value from the stack, unpacks it into three values, then places them each on top of the stack. The computer takes the top value from the stack and stores it in the local variable named `left`. The computer takes the top value from the stack and stores it in the local variable named `op`. The computer takes the top value from the stack and stores it in the local variable named `right`. The computer places the literal string *'{} and {} using {}'* on top of the stack. The computer takes the top value from the stack and retrieves its attribute named `format`, placing it on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `left` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `right` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer loads a reference to the global variable named `describe_node` and places it on top of the stack. The computer loads a reference to the local variable named `op` and places it on top of the stack. The computer takes the top value from the stack, along with another value which it calls as a function, using the original value as an argument, placing the return value on the stack. The computer takes three values from the stack, along with another value which it calls as a function, using the original values as arguments, placing the return value on the stack. The computer takes the top value from the stack and yields it from the current generator. The computer discards the top value from the stack. The computer jumps to offset 2.

Offset 42

The computer places the constant None on top of the stack. The computer exits the current function, returning the top value on the stack.