Detailed Explanation of PEP 701 Implementation in Black PR #3822

Overview

The implementation adds proper grammar-level support for f-strings as defined by PEP 701, transforming them from opaque STRING tokens into structured token sequences. This fundamental change ripples through Black's tokenization, parsing, and formatting pipeline, requiring coordinated modifications across multiple modules.

Core Tokenization Changes (`src/blib2to3/pgen2/tokenize.py`)

The tokenizer undergoes the most substantial transformation. Previously, f-strings like `f"hello {world}"` were emitted as a single STRING token containing the entire literal. Now, the tokenizer breaks them into components.

New Token Types

The module introduces three new token type constants (likely in the 59-61 range based on Python's token numbering):

- `FSTRING_START`: Captures the f-string prefix and opening quote(s) - e.g., `f"`, `F'`, `rf"""`, `fr'`
- `FSTRING_MIDDLE`: The literal text portions between expressions - e.g., "hello " and "" in `f"hello {world}"`
- `FSTRING_END`: The closing quote(s) matching the opener - e.g., `"`, `'`, `"""`

State Management

The tokenizer maintains several state variables to track f-string parsing context:

- `fstring_level`: An integer counter tracking nesting depth (0 means not in f-string, 1+ means nested level)
- `fstring_stack`: A stack storing the quote style for each nesting level (so it knows when to close)
- Position tracking variables to remember where f-string components begin and end

Detection Logic

A new function `is_fstring_start()` examines tokens to determine if they begin an f-string. It checks for:

- The 'f' or 'F' prefix (possibly combined with 'r'/'R' for raw strings)
- Immediately followed by a quote character (', ", ''', or """)
- The function extracts and returns the quote style for later matching

Tokenization Algorithm

When `generate_tokens()` encounters an f-string start:

1. **Emit FSTRING_START**: Yields a token containing the prefix and opening quote
2. **Enter f-string mode**: Sets state variables indicating we're inside an f-string
3. **Scan for content**: Character-by-character scanning begins:
   - **Regular characters**: Accumulated into a buffer for FSTRING_MIDDLE
   - **Double braces `{{` or `}}`**: Collapsed to single braces in the buffer (escape sequence)
   - **Single `{`**:
     - Emit any buffered FSTRING_MIDDLE content
     - Emit LBRACE token
     - **Recursively call** a sub-tokenizer for the expression inside
     - Continue until matching `}` is found
   - **Single `}` (unmatched)**: If not preceded by `{`, this closes an expression
   - **Closing quote**: If it matches the opening quote style, emit final FSTRING_MIDDLE (if any) and FSTRING_END
4. **Expression tokenization**: When inside `{...}`:
   - Call `generate_tokens()` recursively on the expression text
   - Track brace depth to handle nested structures like `{func({x})}`
   - Recognize `:` as potentially starting a format spec (not a slice operator)

- Recognize `!` as a format conversion specifier (`!r`, `!s`, `!a`)
- Continue yielding expression tokens (NAME, NUMBER, OP, etc.)
- Stop when matching `}` is found at the correct nesting level

Recursive Handling

The key innovation is using the call stack for nesting. When an f-string contains another f-string:

```python
f"outer {f"inner {x}"} text"
```

The tokenizer:

1. Emits FSTRING_START for outer f-string
2. Scans "outer " → FSTRING_MIDDLE
3. Sees `{` → begins expression mode
4. Sees `f"` → recognizes nested f-string start
5. Recursively processes inner f-string with incremented `fstring_level`
6. Returns from recursion when inner f-string closes
7. Continues outer expression until `}`
8. Resumes outer f-string text scanning

Helper Functions

- `extract_fstring_quotes()`: Parses the quote portion from the f-string start to determine single/double/triple quote style
- `match_fstring_end()`: Checks if current position has the closing quote that matches the f-string start
- `handle_double_braces()`: Detects and collapses `{{` or `}}` into single braces
- `tokenize_format_spec()`: Special handling for the format specification after `:` in `{expr:format}`

Token Type Definitions (`src/blib2to3/pgen2/token.py`)

This module receives additions to define the new token types. The constants are added to the existing token enumeration:

```python
FSTRING_START = 59  # or next available number
FSTRING_MIDDLE = 60
FSTRING_END = 61
```

Additionally, the `tok_name` dictionary is updated to map these integers to human-readable names for debugging output. The `EXACT_TOKEN_TYPES` dictionary might be updated if there are special characters that need exact matching in f-string contexts.

Grammar Modifications (`src/blib2to3/Grammar.txt`)

The grammar file receives fundamental updates to formalize f-string structure. Previously, f-strings were handled outside the grammar as STRING tokens.

New Grammar Rules

The `atom` production rule (which defines what can be a basic expression) is extended:

```
atom: ('(' [yield_expr|testlist_gexp] ')' |
       '[' [listmaker] ']' |
       '{' [dictorsetmaker] '}' |
       '`' testlist1 '`' |
       NAME | NUMBER | STRING+ | fstring+)  # Added fstring+
```

New production rules define f-string structure:

```
fstring: FSTRING_START fstring_middle* FSTRING_END

fstring_middle: FSTRING_MIDDLE | fstring_replacement_field

fstring_replacement_field:
    LBRACE
    (yield_expr | testlist_star_expr)
    ['=' fstring_debug_equal]  # For f"{x=}" syntax
    ['!' fstring_conversion]    # For f"{x!r}" syntax
    [':' fstring_format_spec]   # For f"{x:format}" syntax
    RBRACE

fstring_debug_equal: EQUAL

fstring_conversion: NAME  # 'r', 's', or 'a'

fstring_format_spec: (FSTRING_MIDDLE | fstring_replacement_field)*
```

The recursive nature of `fstring_format_spec` allowing `fstring_replacement_field` enables nested expressions in format specs: `f"{x:{y}}"`.

Impact

This grammar formalization means the parser now builds proper AST nodes for f-strings with structural components, rather than treating them as opaque strings. The parser can validate f-string syntax during parsing rather than requiring later analysis.

Parser Updates (`src/blib2to3/pgen2/parse.py`)

The parser requires minimal changes because it's largely driven by the grammar. However, there may be adjustments to:

Token Stream Handling

- Recognize the new token types in the token stream
- Build AST nodes for `fstring`, `fstring_middle`, and `fstring_replacement_field` grammar rules
- Handle the increased complexity of nested structures

Error Reporting

- Enhanced error messages for malformed f-strings
- Better location tracking since f-strings now span multiple tokens

AST Node Definitions (`src/blib2to3/pytree.py`)

New node types are implicitly created for the new grammar rules. The AST structure changes from:

```
STRING("f'hello {world}'")
```

To:

```
fstring
├── FSTRING_START("f'")
├── fstring_middle
│   └── FSTRING_MIDDLE("hello ")
├── fstring_replacement_field
│   ├── LBRACE("{")
│   ├── NAME("world")
│   └── RBRACE("}")
└── FSTRING_END("'")
```

Black's Core Formatting Logic

Feature Detection (`src/black/__init__.py`)

The f-string detection logic migrates from string-based inspection to AST-based detection:

Old approach (removed):

```
python
```

```
# Scanned STRING tokens looking for:
# - 'f' or 'F' prefix
# - Debug syntax '=' before '}'
# - Manually parsed string content
```

New approach:

```python
# Traverse AST looking for:
# - Nodes with type == token.FSTRING_START
# - Children with type == fstring_replacement_field
# - Debug syntax detected by presence of EQUAL token
```

This is more robust because the parser has already validated the f-string structure.

Line Generation (`src/black/linegen.py`)

The `Visitor` class receives a new method `visit_fstring()` to handle f-string nodes.

Current Implementation (Temporary): The PR includes a conservative approach that converts f-string AST nodes back to STRING tokens to preserve existing behavior:

```python
def visit_fstring(self, node: Node) -> Iterator[Line]:
    # Reconstruct string representation from AST
    string_value = _fstring_to_string(node)
    # Yield as if it were a STRING token
    yield from self.visit_default(string_value)
```

The `_fstring_to_string()` helper function:

- Traverses the f-string AST
- Concatenates FSTRING_START, FSTRING_MIDDLE, expressions, and FSTRING_END
- Returns a string representation matching the original source

Future Implementation (Commented Out): The PR includes commented code showing the intended full implementation:

```python
def visit_fstring(self, node: Node) -> Iterator[Line]:
    # Process each component individually
    for child in node.children:
        if child.type == token.FSTRING_START:
            # Normalize quote style
            ...
        elif child.type == token.FSTRING_MIDDLE:
            # Escape sequences, quote normalization
            ...
        elif child.type == fstring_replacement_field:
            # Format the expression inside
            # Apply Black's rules to the expression
            ...
```

This future implementation would enable:

- Normalizing quote styles in f-strings
- Formatting expressions inside `{...}`
- Consistent spacing around format specifiers
- Proper handling of line breaks in f-string expressions

Line Structure (`src/black/lines.py`)

Adjustments ensure f-string components are handled correctly:

Empty Token Handling:

```python
# Prevent FSTRING_MIDDLE from being truncated
# Even empty FSTRING_MIDDLE is significant:
# f"{x}{y}" has an empty FSTRING_MIDDLE between expressions
```

Line Break Logic:

- F-string components shouldn't be split across lines inappropriately
- FSTRING_START and FSTRING_END must stay with their content
- Multi-line f-strings (triple-quoted) need special handling

Node Whitespace Rules (`src/black/nodes.py`)

New rules govern spacing around f-string tokens:

```python
# No space before:
# - FSTRING_MIDDLE: f"hello{world}" not f"hello {world}"
# - FSTRING_END: proper quote closure
# - BANG: f"{x!r}" not f"{x !r}"

# No space after:
# - BANG: f"{x!r}" not f"{x! r}"
# - LBRACE when starting replacement field: f"{x}" not f"{ x}"

# Space rules for replacement fields:
# - Expressions inside follow normal rules
# - Format specs may need special handling
```

Backwards Compatibility

The implementation maintains compatibility with pre-3.12 Python versions:

Permissive Parsing

- Black accepts PEP 701 syntax even when formatting for older Python targets
- The tokenizer doesn't enforce Python version restrictions
- Users are responsible for target compatibility

Rationale

Black's philosophy is to format code, not validate runtime compatibility. If code parses correctly, Black will format it, regardless of whether it's valid for a specific Python version.

Edge Cases Handled

Nested F-Strings

```python
f"outer {f"inner {f"deepest {x}"}"}"
```

The recursive tokenization handles arbitrary nesting depth (though the spec doesn't mandate unlimited depth).

Escaped Braces

```python
f"{{literal braces}}"  # → {literal braces}
```

Double braces are collapsed in FSTRING_MIDDLE tokens.

Backslashes

```python
f"path: {path}\nline2"  # Backslash outside expression
f"{s.replace('\\', '/')}"  # Backslash in expression (now allowed)
```

Format Specifiers

```python
f"{x:05d}"  # Format spec
f"{x!r:>10}"  # Conversion then format
f"{x:{width}.{precision}}"  # Nested expressions in format spec
```

Debug Syntax

```python
f"{x=}"  # Expands to "x=<value>"
f"{x + y=}"  # Works with expressions
```

Quote Reuse

```python
f"She said {greet("hello")}"  # Same quotes nested
```

This is the key PEP 701 feature - previously impossible, now allowed.

Multiline F-Strings

```python
f"""text
{expr
 spanning
 lines}
more"""
```

Raw F-Strings

```python
rf"regex: {pattern}"  # Both r and f prefixes
```

Testing Strategy

The PR includes comprehensive test files (covered in next phase):

- `tests/data/cases/pep_701.py`: Input test cases
- Various edge cases for all features mentioned above

- Regression tests for bugs found during development

Performance Considerations

Tokenization Overhead

Breaking f-strings into multiple tokens increases the token count, but the impact is minimal because:

- The tokenizer is already scanning character-by-character
- Recursive calls reuse existing logic
- The added complexity is proportional to f-string nesting (typically shallow)

Parsing Complexity

The grammar additions don't significantly increase parser complexity because:

- The grammar rules are straightforward
- Nesting is handled through recursion (natural for parsers)
- No ambiguity introduced

Future Work

The PR lays groundwork for future enhancements:

1. **Full F-String Formatting:** The commented-out code in `visit_fstring()` would enable Black to reformat f-string contents
2. **Quote Normalization:** Consistently using double quotes (or user preference) in f-strings
3. **Expression Formatting:** Applying Black's formatting rules to expressions inside `{...}`
4. **Multiline Expression Handling:** Properly indenting complex expressions that span lines within f-strings

Summary of Changes by File

- **`tokenize.py`**: ~300-400 lines added/modified - core tokenization logic
- **`token.py`**: ~10 lines added - token type definitions

- `Grammar.txt`: ~30 lines added - grammar formalization
- `parse.py`: ~50 lines modified - parser adjustments
- `__init__.py`: ~100 lines modified - feature detection migration
- `linegen.py`: ~50 lines added - visit_fstring() implementation
- `lines.py`: ~20 lines modified - empty token handling
- `nodes.py`: ~30 lines modified - whitespace rules
- **Test files**: ~500 lines added - comprehensive test coverage

Implementation Philosophy

The implementation is conservative by design: it enables Black to **parse** PEP 701 f-strings without crashing, while deferring actual **formatting** improvements to future work. This allows Black to work with Python 3.12+ code immediately while the team refines the formatting behavior.

The key insight is that by properly tokenizing and parsing f-strings at the grammar level, Black gains the infrastructure needed for future enhancements. The current "pass-through" approach (converting back to STRING tokens) is a temporary measure that allows the core infrastructure to be tested and validated before adding more complex formatting logic.

Document Version: 1.0
PR Reference: psf/black#3822
PEP Reference: PEP 701 - Syntactic formalization of f-strings
Author: Tushar Sadhwani
Merged: April 22, 2024