Evaluation

1. Global Definitions and Regular Expressions

```
KEYWORDS = {'int', 'float', 'if', 'else', 'while', 'return'}
OPERATORS = {'+', '-', '*', '/', '=', '==', '!=', '&&', '||'}
SEPARATORS = {';', ',', '(', ')', '{', '}'}
identifier_pattern = re.compile(r'^[a-zA-Z_]\w*$')
integer_literal_pattern = re.compile(r'^\d+$')
float_literal_pattern = re.compile(r'^\d+\.\d+$')
string_literal_pattern = re.compile(r'^".*"$')
```

Purpose: These definitions establish categories (keywords, operators, separators) and patterns for token recognition.
Effectiveness:
- The sets for `KEYWORDS`, `OPERATORS`, and `SEPARATORS` make checking a token's type fast and straightforward.
- Regular expressions allow for flexible pattern matching for identifiers, literals, and strings:
  - `identifier_pattern` matches valid variable or function names.
  - `integer_literal_pattern` and `float_literal_pattern` differentiate between integer and floating-point literals.
  - `string_literal_pattern` recognizes string literals but only handles single-line strings.

2. `categorize_token` Function

```
def categorize_token(token):
    if token in KEYWORDS:
        return f"{token} (keyword)"
    elif token in OPERATORS:
        return f"{token} (operator)"
    elif token in SEPARATORS:
        return f"{token} (separator)"
    elif identifier_pattern.match(token):
        return f"{token} (identifier)"
    elif integer_literal_pattern.match(token):
        return f"{token} (integer literal)"
    elif float_literal_pattern.match(token):
        return f"{token} (floating-point literal)"
    elif string_literal_pattern.match(token):
        return f"{token} (string literal)"
    return f"{token} (unknown)"
```

Purpose: This function receives a token and matches it against predefined types, returning the token along with its category.
Effectiveness:

- The function is effective and follows a clear decision-making flow to classify each token, from specific (keywords, operators) to general (identifiers, literals).
- Returns an `"unknown"` category if the token doesn’t match any predefined type, providing feedback when a token does not fall into expected categories.

3. `tokenize_and_categorize` Function

```
def tokenize_and_categorize(input_string):
    tokens = re.findall(r'\w+|==|!=|&&|\|\||[^\s\w]', input_string)
    categorized_tokens = [categorize_token(token) for token in tokens]
    return categorized_tokens
```

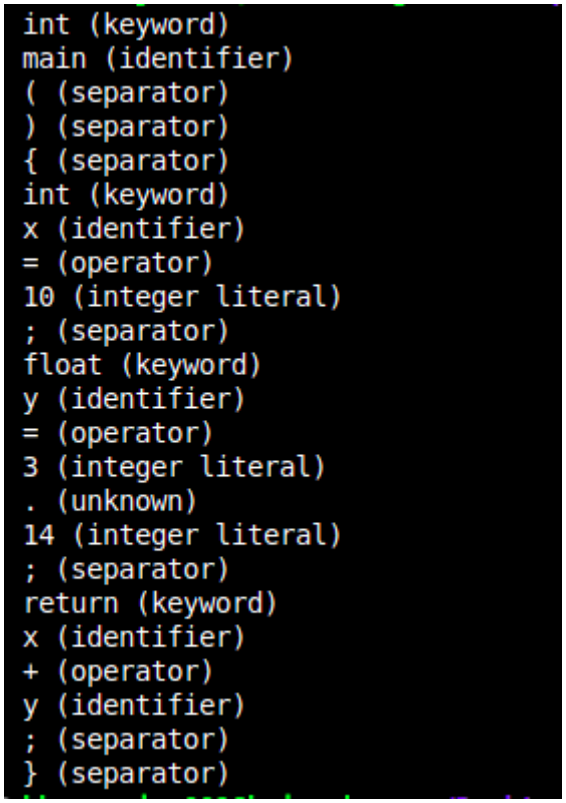
Purpose: This function tokenizes the input string and then categorizes each token.
Effectiveness:
- The regular expression `\w+|==|!=|&&|\|\||[^\s\w]` accurately splits the input into words, operators, and individual symbols.
- Tokenization and categorization are handled in one pass, making it efficient for small to medium-sized code snippets.

4. `process_file` Function

```
def process_file(filename):
    with open(filename, 'r') as file:
        input_code = file.read()
    categorized_tokens = tokenize_and_categorize(input_code)
    for token in categorized_tokens:
        print(token)
```

Purpose: This function opens the specified file, reads its contents, and processes each line to extract and print categorized tokens.
Effectiveness:
- Simple and effective for reading a single code snippet, tokenizing, and printing each token with its type.
- The use of `with open` ensures that the file is closed automatically after reading, following best practices.

Output:


int (keyword)
main (identifier)
((separator)
) (separator)
{ (separator)
int (keyword)
x (identifier)
= (operator)
10 (integer literal)
; (separator)
float (keyword)
y (identifier)
= (operator)
3 (integer literal)
. (unknown)
14 (integer literal)
; (separator)
return (keyword)
x (identifier)
+ (operator)
y (identifier)
; (separator)
} (separator)