Project 01 - scanner

● Graded

20 Hours, 48 Minutes Late

Student

Ishan Mukherjee

Total Points

100 / 100 pts

Autograder Score

80.0 / 80.0

Passed Tests

Test 1: test01.py
Test 2: test02.py
Test 3: test03.py
Test 4: test04.py
Test 5: test05.py
Test 6: test06.py
Test 7: test07.py

Question 2

Manual review

20 / 20 pts

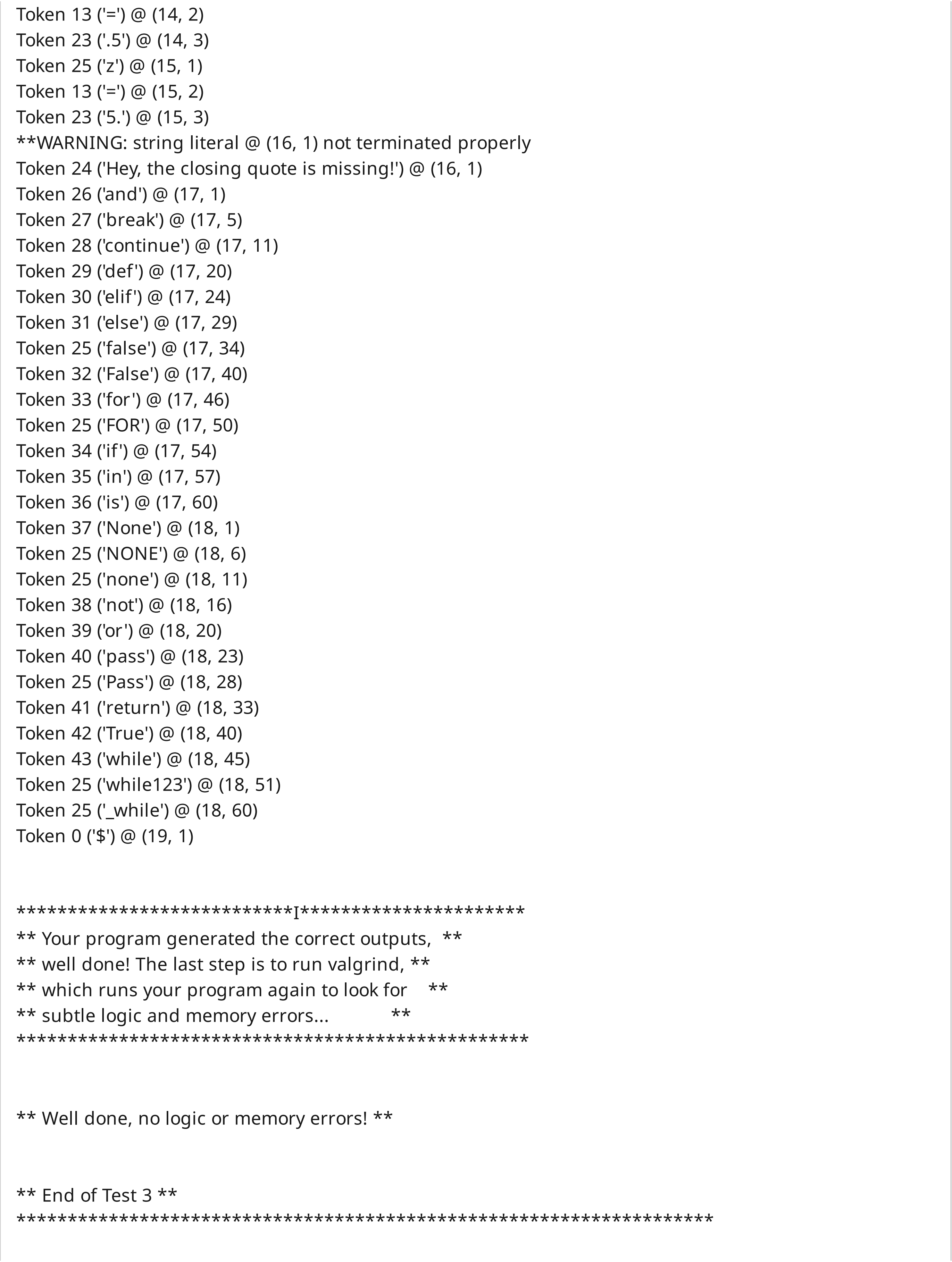
- ✓ **+ 2 pts** "scanner.c" header comment has description
- ✓ **+ 2 pts** "scanner.c" header comment has name
- ✓ **+ 1 pt** "scanner.c" header comment school, course, etc.
- ✓ **+ 4 pts** Definition and use of function for int literals
- ✓ **+ 1 pt** has header comment
- ✓ **+ 4 pts** Definition and use of function for real literals (can be the same function with int literals)
- ✓ **+ 1 pt** has header comment
- ✓ **+ 4 pts** Definition and use of function for str literals
- ✓ **+ 1 pt** has header comment

Autograder Results

Autograder Output

```
*********************************************************
This is submission #3
Submitted @ 20:47 on 2024-1-13 (Chicago time)

Submission history:
 Submission #2: score=80, submitted @ 20:30 on 2024-1-13 (Chicago time)
 Submission #1: score=30, submitted @ 20:17 on 2024-1-13 (Chicago time)

Total # of valid submissions so far: 2
# of valid submissions since midnight: 2
# of minutes since last valid submission: 17
*********************************************************
You have 2 submissions this 24-hr period.


******************************************************************
** Number of Submissions This Time Period                       **
******************************************************************

This is submission #3 in current time period

You are allowed a total of 7 submissions per 24-hr time period.

******************************************************************



*******************************************************************
** Test Number: 1 **


** Test Input:
print(x)
{
  a = b
}


** Your output (first 600 lines) **
Token 25 ('print') @ (1, 1)
Token 1 ('(') @ (1, 6)
Token 25 ('x') @ (1, 7)
Token 2 (')') @ (1, 8)
Token 5 ('{') @ (2, 1)
Token 25 ('a') @ (3, 3)
Token 13 ('=') @ (3, 5)
Token 25 ('b') @ (3, 7)
Token 6 ('}') @ (4, 1)
Token 0 ('$') @ (5, 1)
```

```
**************************I*********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 1 **
***********************************************************************


***********************************************************************
** Test Number: 2 **


** Test Input:
print(count)

if x<y:
{
  z = 123
  _var = 0.5
}
else:
{
  s1 = 'this is a string'
  s2 = "and this has 123"
  s3 = "and I quote 'hi'"
  s4 = 'and "quote" within'
}

print("The loop yields:")

while x<y:
{
  print(x)
  print(y)
}


** Your output (first 600 lines) **
Token 25 ('print') @ (1, 1)
Token 1 ('(') @ (1, 6)
Token 25 ('count') @ (1, 7)
Token 2 (')') @ (1, 12)
Token 34 ('if') @ (3, 1)
Token 25 ('x') @ (3, 4)
Token 16 ('<') @ (3, 5)
Token 25 ('y') @ (3, 6)
Token 21 (':') @ (3, 7)
Token 5 ('{') @ (4, 1)
Token 25 ('z') @ (5, 3)
Token 13 ('=') @ (5, 5)
Token 22 ('123') @ (5, 7)
Token 25 ('_var') @ (6, 3)
Token 13 ('=') @ (6, 8)
Token 23 ('0.5') @ (6, 10)
Token 6 ('}') @ (7, 1)
Token 31 ('else') @ (8, 1)
Token 21 (':') @ (8, 5)
Token 5 ('{') @ (9, 1)
Token 25 ('s1') @ (10, 3)
Token 13 ('=') @ (10, 6)
Token 24 ('this is a string') @ (10, 8)
Token 25 ('s2') @ (11, 3)
Token 13 ('=') @ (11, 6)
Token 24 ('and this has 123') @ (11, 8)
Token 25 ('s3') @ (12, 3)
Token 13 ('=') @ (12, 6)
Token 24 ('and I quote 'hi'') @ (12, 8)
Token 25 ('s4') @ (13, 3)
Token 13 ('=') @ (13, 6)
Token 24 ('and "quote" within') @ (13, 8)
Token 6 ('}') @ (14, 1)
Token 25 ('print') @ (16, 1)
Token 1 ('(') @ (16, 6)
Token 24 ('The loop yields:') @ (16, 7)
Token 2 (')') @ (16, 25)
Token 43 ('while') @ (18, 1)
Token 25 ('x') @ (18, 7)
Token 16 ('<') @ (18, 8)
Token 25 ('y') @ (18, 9)
Token 21 (':') @ (18, 10)
Token 5 ('{') @ (19, 1)
Token 25 ('print') @ (20, 3)
Token 1 ('(') @ (20, 8)
Token 25 ('x') @ (20, 9)
Token 2 (')') @ (20, 10)
Token 25 ('print') @ (21, 3)
Token 1 ('(') @ (21, 8)
Token 25 ('y') @ (21, 9)
Token 2 (')') @ (21, 10)
Token 6 ('}') @ (22, 1)
Token 0 ('$') @ (23, 1)


**************************************************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
*************************************************


** Well done, no logic or memory errors! **


** End of Test 2 **
********************************************************************


********************************************************************
** Test Number: 3 **


** Test Input:
#
# this is a comment
#
print(_count)
+-*/%**=!===
<=<>>=&:#123
123<123.456
(+123)
if(123==_123count456):{hi}#random (123) comment
{-9}
[ +0.5 123 .5 . ]  # the .5 is a real number, the . by itself is unknown
-12.
x=123456.7890012435
y=.5
z=5.
"Hey, the closing quote is missing!
and break continue def elif else false False for FOR if in is
None NONE none not or pass Pass return True while while123 _while


** Your output (first 600 lines) **
Token 25 ('print') @ (4, 1)
Token 1 ('(') @ (4, 6)
Token 25 ('_count') @ (4, 7)
Token 2 (')') @ (4, 13)
Token 7 ('+') @ (5, 1)
Token 8 ('-') @ (5, 2)
Token 9 ('*') @ (5, 3)
Token 12 ('/') @ (5, 4)
Token 11 ('%') @ (5, 5)
Token 10 ('**') @ (5, 6)
Token 13 ('=') @ (5, 8)
Token 15 ('!=') @ (5, 9)
Token 14 ('==') @ (5, 11)
Token 17 ('<=') @ (6, 1)
Token 16 ('<') @ (6, 3)
Token 18 ('>') @ (6, 4)
Token 19 ('>=') @ (6, 5)
Token 20 ('&') @ (6, 7)
Token 21 (':') @ (6, 8)
Token 22 ('123') @ (7, 1)
Token 16 ('<') @ (7, 4)
Token 23 ('123.456') @ (7, 5)
Token 1 ('(') @ (8, 1)
Token 7 ('+') @ (8, 2)
Token 22 ('123') @ (8, 3)
Token 2 (')') @ (8, 6)
Token 34 ('if') @ (9, 1)
Token 1 ('(') @ (9, 3)
Token 22 ('123') @ (9, 4)
Token 14 ('==') @ (9, 7)
Token 25 ('_123count456') @ (9, 9)
Token 2 (')') @ (9, 21)
Token 21 (':') @ (9, 22)
Token 5 ('{') @ (9, 23)
Token 25 ('hi') @ (9, 24)
Token 6 ('}') @ (9, 26)
Token 5 ('{') @ (10, 1)
Token 8 ('-') @ (10, 2)
Token 22 ('9') @ (10, 3)
Token 6 ('}') @ (10, 4)
Token 3 ('[') @ (11, 1)
Token 7 ('+') @ (11, 3)
Token 23 ('0.5') @ (11, 4)
Token 22 ('123') @ (11, 8)
Token 23 ('.5') @ (11, 12)
Token -1 ('.') @ (11, 15)
Token 4 (']') @ (11, 17)
Token 8 ('-') @ (12, 1)
Token 23 ('12.') @ (12, 2)
Token 25 ('x') @ (13, 1)
Token 13 ('=') @ (13, 2)
Token 23 ('123456.7890012435') @ (13, 3)
Token 25 ('y') @ (14, 1)
```



```
Token 13 ('=') @ (14, 2)
Token 23 ('.5') @ (14, 3)
Token 25 ('z') @ (15, 1)
Token 13 ('=') @ (15, 2)
Token 23 ('5.') @ (15, 3)
**WARNING: string literal @ (16, 1) not terminated properly
Token 24 ('Hey, the closing quote is missing!') @ (16, 1)
Token 26 ('and') @ (17, 1)
Token 27 ('break') @ (17, 5)
Token 28 ('continue') @ (17, 11)
Token 29 ('def') @ (17, 20)
Token 30 ('elif') @ (17, 24)
Token 31 ('else') @ (17, 29)
Token 25 ('false') @ (17, 34)
Token 32 ('False') @ (17, 40)
Token 33 ('for') @ (17, 46)
Token 25 ('FOR') @ (17, 50)
Token 34 ('if') @ (17, 54)
Token 35 ('in') @ (17, 57)
Token 36 ('is') @ (17, 60)
Token 37 ('None') @ (18, 1)
Token 25 ('NONE') @ (18, 6)
Token 25 ('none') @ (18, 11)
Token 38 ('not') @ (18, 16)
Token 39 ('or') @ (18, 20)
Token 40 ('pass') @ (18, 23)
Token 25 ('Pass') @ (18, 28)
Token 41 ('return') @ (18, 33)
Token 42 ('True') @ (18, 40)
Token 43 ('while') @ (18, 45)
Token 25 ('while123') @ (18, 51)
Token 25 ('_while') @ (18, 60)
Token 0 ('$') @ (19, 1)


**********************************************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
***********************************************


** Well done, no logic or memory errors! **


** End of Test 3 **
***************************************************************
```

```
***************************************************************************
** Test Number: 4 **


** Test Input:
print(x)
[
  a + b * c / d % f
]
if (x):
  pass
elif (x==y) and (y<z):
  return
else:
{
  while True:
    true = False
}

!

x---y
x++++y




** Your output (first 600 lines) **
Token 25 ('print') @ (1, 1)
Token 1 ('(') @ (1, 6)
Token 25 ('x') @ (1, 7)
Token 2 (')') @ (1, 8)
Token 3 ('[') @ (2, 1)
Token 25 ('a') @ (3, 3)
Token 7 ('+') @ (3, 5)
Token 25 ('b') @ (3, 7)
Token 9 ('*') @ (3, 9)
Token 25 ('c') @ (3, 11)
Token 12 ('/') @ (3, 13)
Token 25 ('d') @ (3, 15)
Token 11 ('%') @ (3, 17)
Token 25 ('f') @ (3, 19)
Token 4 (']') @ (4, 1)
Token 34 ('if') @ (5, 1)
Token 1 ('(') @ (5, 4)
Token 25 ('x') @ (5, 5)
Token 2 (')') @ (5, 6)
Token 21 (':') @ (5, 7)
Token 40 ('pass') @ (6, 3)
Token 30 ('elif') @ (7, 1)
Token 1 ('(') @ (7, 6)
Token 25 ('x') @ (7, 7)
Token 14 ('==') @ (7, 8)
Token 25 ('y') @ (7, 10)
Token 2 (')') @ (7, 11)
Token 26 ('and') @ (7, 13)
Token 1 ('(') @ (7, 17)
Token 25 ('y') @ (7, 18)
Token 16 ('<') @ (7, 19)
Token 25 ('z') @ (7, 20)
Token 2 (')') @ (7, 21)
Token 21 (':') @ (7, 22)
Token 41 ('return') @ (8, 3)
Token 31 ('else') @ (9, 1)
Token 21 (':') @ (9, 5)
Token 5 ('{') @ (10, 1)
Token 43 ('while') @ (11, 3)
Token 42 ('True') @ (11, 9)
Token 21 (':') @ (11, 13)
Token 25 ('true') @ (12, 5)
Token 13 ('=') @ (12, 10)
Token 32 ('False') @ (12, 12)
Token 6 ('}') @ (13, 1)
Token -1 ('!') @ (15, 1)
Token 25 ('x') @ (17, 1)
Token 8 ('-') @ (17, 2)
Token 8 ('-') @ (17, 3)
Token 8 ('-') @ (17, 4)
Token 25 ('y') @ (17, 5)
Token 25 ('x') @ (18, 1)
Token 7 ('+') @ (18, 2)
Token 7 ('+') @ (18, 3)
Token 7 ('+') @ (18, 4)
Token 7 ('+') @ (18, 5)
Token 25 ('y') @ (18, 6)
Token 0 ('$') @ (21, 1)


**********************************************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
**********************************************


** Well done, no logic or memory errors! **
** End of Test 4 **
********************************************************************


********************************************************************
** Test Number: 5 **


** Test Input:
x="hi"
"Hello there"'and hi there'
print('oops, forgot closing quote)

print("output is")
print('output is')

print("output 'is'")
print('output "quoted by..." I'm sure it's okay?')

print("oops, forgot closing quote)
x = count_123




** Your output (first 600 lines) **
Token 25 ('x') @ (1, 1)
Token 13 ('=') @ (1, 2)
Token 24 ('hi') @ (1, 3)
Token 24 ('Hello there') @ (2, 1)
Token 24 ('and hi there') @ (2, 14)
Token 25 ('print') @ (3, 1)
Token 1 ('(') @ (3, 6)
**WARNING: string literal @ (3, 7) not terminated properly
Token 24 ('oops, forgot closing quote)') @ (3, 7)
Token 25 ('print') @ (5, 1)
Token 1 ('(') @ (5, 6)
Token 24 ('output is') @ (5, 7)
Token 2 (')') @ (5, 18)
Token 25 ('print') @ (6, 1)
Token 1 ('(') @ (6, 6)
Token 24 ('output is') @ (6, 7)
Token 2 (')') @ (6, 18)
Token 25 ('print') @ (8, 1)
Token 1 ('(') @ (8, 6)
Token 24 ('output 'is'') @ (8, 7)
Token 2 (')') @ (8, 20)
Token 25 ('print') @ (9, 1)
Token 1 ('(') @ (9, 6)
Token 24 ('output "quoted by..." I') @ (9, 7)
Token 25 ('m') @ (9, 32)
Token 25 ('sure') @ (9, 34)
Token 25 ('it') @ (9, 39)
Token 24 ('s okay?') @ (9, 41)
Token 2 (')') @ (9, 50)
Token 25 ('print') @ (11, 1)
Token 1 ('(') @ (11, 6)
**WARNING: string literal @ (11, 7) not terminated properly
Token 24 ('oops, forgot closing quote)') @ (11, 7)
Token 25 ('x') @ (12, 1)
Token 13 ('=') @ (12, 3)
Token 25 ('count_123') @ (12, 5)
Token 0 ('$') @ (15, 1)


**************************************************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
**************************************************


** Well done, no logic or memory errors! **


** End of Test 5 **
********************************************************************


********************************************************************
** Test Number: 6 **


** Test Input:
123
123.456
12.
.1
0.
.0
000000.0000
0000000
0
x=123.45+7.0
y=123 + +9912345448383 - -456 + 12 - 45
```

```
42.41.40
39..23
39.5..23.5
x=++39*-38/2
y=++39.5*-38.2/99.1234567890

y=123. + +9912345448383.23 - -456.0 + 12.123 - 45.9 * .8 / 1.;
```

```
** Your output (first 600 lines) **
Token 22 ('123') @ (1, 1)
Token 23 ('123.456') @ (2, 1)
Token 23 ('12.') @ (3, 1)
Token 23 ('.1') @ (4, 1)
Token 23 ('0.') @ (5, 1)
Token 23 ('.0') @ (6, 1)
Token 23 ('000000.0000') @ (7, 1)
Token 22 ('0000000') @ (8, 1)
Token 22 ('0') @ (9, 1)
Token 25 ('x') @ (10, 1)
Token 13 ('=') @ (10, 2)
Token 23 ('123.45') @ (10, 3)
Token 7 ('+') @ (10, 9)
Token 23 ('7.0') @ (10, 10)
Token 25 ('y') @ (11, 1)
Token 13 ('=') @ (11, 2)
Token 22 ('123') @ (11, 3)
Token 7 ('+') @ (11, 7)
Token 7 ('+') @ (11, 9)
Token 22 ('9912345448383') @ (11, 10)
Token 8 ('-') @ (11, 24)
Token 8 ('-') @ (11, 26)
Token 22 ('456') @ (11, 27)
Token 7 ('+') @ (11, 31)
Token 22 ('12') @ (11, 33)
Token 8 ('-') @ (11, 36)
Token 22 ('45') @ (11, 38)
Token 23 ('42.41') @ (12, 1)
Token 23 ('.40') @ (12, 6)
Token 23 ('39.') @ (13, 1)
Token 23 ('.23') @ (13, 4)
Token 23 ('39.5') @ (14, 1)
Token -1 ('.') @ (14, 5)
Token 23 ('.23') @ (14, 6)
Token 23 ('.5') @ (14, 9)
Token 25 ('x') @ (15, 1)
Token 13 ('=') @ (15, 2)
Token 7 ('+') @ (15, 3)
Token 7 ('+') @ (15, 4)
Token 22 ('39') @ (15, 5)
Token 9 ('*') @ (15, 7)
Token 8 ('-') @ (15, 8)
Token 22 ('38') @ (15, 9)
Token 12 ('/') @ (15, 11)
Token 22 ('2') @ (15, 12)
Token 25 ('y') @ (16, 1)
Token 13 ('=') @ (16, 2)
Token 7 ('+') @ (16, 3)
Token 7 ('+') @ (16, 4)
Token 23 ('39.5') @ (16, 5)
Token 9 ('*') @ (16, 9)
Token 8 ('-') @ (16, 10)
Token 23 ('38.2') @ (16, 11)
Token 12 ('/') @ (16, 15)
Token 23 ('99.1234567890') @ (16, 16)
Token 25 ('y') @ (18, 1)
Token 13 ('=') @ (18, 2)
Token 23 ('123.') @ (18, 3)
Token 7 ('+') @ (18, 8)
Token 7 ('+') @ (18, 10)
Token 23 ('9912345448383.23') @ (18, 11)
Token 8 ('-') @ (18, 28)
Token 8 ('-') @ (18, 30)
Token 23 ('456.0') @ (18, 31)
Token 7 ('+') @ (18, 37)
Token 23 ('12.123') @ (18, 39)
Token 8 ('-') @ (18, 46)
Token 23 ('45.9') @ (18, 48)
Token 9 ('*') @ (18, 53)
Token 23 ('.8') @ (18, 55)
Token 12 ('/') @ (18, 58)
Token 23 ('1.') @ (18, 60)
Token -1 (';') @ (18, 62)
Token 0 ('$') @ (25, 1)


**************************************************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
**************************************************
```

```
** Well done, no logic or memory errors! **


** End of Test 6 **
*****************************************************************


*****************************************************************
** Test Number: 7 **


** Test Input:
#
# this is a comment
#
print(_count)
# and another # comment
+ -   *       /       % ** = != ==
< = <> >= &:#123+456
123 < 123.456
(+123)
if(123==_123count456):[hi]][[][#random (123) comment
{-9}()(()))({)_a b c c_123+456ab
[+0.5123.5--123.45]# +0.5123 . 5 - .893 -123.45***5=====x 91.
+0.5123 . 5 - .893 -123.45***5=====x 91.
-12 -12...5 12 -12.45 12.45
x<==123456.7890012435
and '' #empty literal?
break ""
continue"abc 123"def elif defelif
else false False for FOR if in is inis isin is_in
None NONE none not or pass Pass return True while
123while"a string literal"
while123 _while !True




** Your output (first 600 lines) **
Token 25 ('print') @ (4, 1)
Token 1 ('(') @ (4, 6)
Token 25 ('_count') @ (4, 7)
Token 2 (')') @ (4, 13)
Token 7 ('+') @ (6, 1)
Token 8 ('-') @ (6, 3)
Token 9 ('*') @ (6, 7)
Token 12 ('/') @ (6, 9)
Token 11 ('%') @ (6, 11)
Token 10 ('**') @ (6, 13)
Token 13 ('=') @ (6, 16)
Token 15 ('!=') @ (6, 18)
Token 14 ('==') @ (6, 21)
Token 16 ('<') @ (7, 1)
Token 13 ('=') @ (7, 3)
Token 16 ('<') @ (7, 5)
Token 18 ('>') @ (7, 6)
Token 19 ('>=') @ (7, 8)
Token 20 ('&') @ (7, 11)
Token 21 (':') @ (7, 12)
Token 22 ('123') @ (8, 1)
Token 16 ('<') @ (8, 5)
Token 23 ('123.456') @ (8, 7)
Token 1 ('(') @ (9, 1)
Token 7 ('+') @ (9, 2)
Token 22 ('123') @ (9, 3)
Token 2 (')') @ (9, 6)
Token 34 ('if') @ (10, 1)
Token 1 ('(') @ (10, 3)
Token 22 ('123') @ (10, 4)
Token 14 ('==') @ (10, 7)
Token 25 ('_123count456') @ (10, 9)
Token 2 (')') @ (10, 21)
Token 21 (':') @ (10, 22)
Token 3 ('[') @ (10, 23)
Token 25 ('hi') @ (10, 24)
Token 4 (']') @ (10, 26)
Token 4 (']') @ (10, 27)
Token 3 ('[') @ (10, 28)
Token 3 ('[') @ (10, 29)
Token 4 (']') @ (10, 30)
Token 3 ('[') @ (10, 31)
Token 5 ('{') @ (11, 1)
Token 8 ('-') @ (11, 2)
Token 22 ('9') @ (11, 3)
Token 6 ('}') @ (11, 4)
Token 1 ('(') @ (11, 5)
Token 2 (')') @ (11, 6)
Token 1 ('(') @ (11, 7)
Token 1 ('(') @ (11, 8)
Token 2 (')') @ (11, 9)
Token 2 (')') @ (11, 10)
Token 2 (')') @ (11, 11)
Token 1 ('(') @ (11, 12)
Token 5 ('{') @ (11, 13)
Token 2 (')') @ (11, 14)
Token 25 ('_a') @ (11, 15)
Token 25 ('b') @ (11, 18)
Token 25 ('c') @ (11, 20)
Token 25 ('c_123') @ (11, 22)
Token 7 ('+') @ (11, 27)
Token 22 ('456') @ (11, 28)
Token 25 ('ab') @ (11, 31)
Token 3 ('[') @ (12, 1)
Token 7 ('+') @ (12, 2)
Token 23 ('0.5123') @ (12, 3)
Token 23 ('.5') @ (12, 9)
Token 8 ('-') @ (12, 11)
Token 8 ('-') @ (12, 12)
Token 23 ('123.45') @ (12, 13)
Token 4 (']') @ (12, 19)
Token 7 ('+') @ (13, 1)
Token 23 ('0.5123') @ (13, 2)
Token -1 ('.') @ (13, 9)
Token 22 ('5') @ (13, 11)
Token 8 ('-') @ (13, 13)
Token 23 ('.893') @ (13, 15)
Token 8 ('-') @ (13, 20)
Token 23 ('123.45') @ (13, 21)
Token 10 ('**') @ (13, 27)
Token 9 ('*') @ (13, 29)
Token 22 ('5') @ (13, 30)
Token 14 ('==') @ (13, 31)
Token 14 ('==') @ (13, 33)
Token 13 ('=') @ (13, 35)
Token 25 ('x') @ (13, 36)
Token 23 ('91.') @ (13, 38)
Token 8 ('-') @ (14, 1)
Token 22 ('12') @ (14, 2)
Token 8 ('-') @ (14, 5)
Token 23 ('12.') @ (14, 6)
Token -1 ('.') @ (14, 9)
Token 23 ('.5') @ (14, 10)
Token 22 ('12') @ (14, 13)
Token 8 ('-') @ (14, 16)
Token 23 ('12.45') @ (14, 17)
Token 23 ('12.45') @ (14, 23)
Token 25 ('x') @ (15, 1)
Token 17 ('<=') @ (15, 2)
Token 13 ('=') @ (15, 4)
Token 23 ('123456.7890012435') @ (15, 5)
Token 26 ('and') @ (16, 1)
Token 24 ('') @ (16, 5)
Token 27 ('break') @ (17, 1)
Token 24 ('') @ (17, 7)
Token 28 ('continue') @ (18, 1)
Token 24 ('abc 123') @ (18, 9)
Token 29 ('def') @ (18, 18)
Token 30 ('elif') @ (18, 22)
Token 25 ('defelif') @ (18, 27)
Token 31 ('else') @ (19, 1)
Token 25 ('false') @ (19, 6)
Token 32 ('False') @ (19, 12)
Token 33 ('for') @ (19, 18)
Token 25 ('FOR') @ (19, 22)
Token 34 ('if') @ (19, 26)
Token 35 ('in') @ (19, 29)
Token 36 ('is') @ (19, 32)
Token 25 ('inis') @ (19, 35)
Token 25 ('isin') @ (19, 40)
Token 25 ('is_in') @ (19, 45)
Token 37 ('None') @ (20, 1)
Token 25 ('NONE') @ (20, 6)
Token 25 ('none') @ (20, 11)
Token 38 ('not') @ (20, 16)
Token 39 ('or') @ (20, 20)
Token 40 ('pass') @ (20, 23)
Token 25 ('Pass') @ (20, 28)
Token 41 ('return') @ (20, 33)
Token 42 ('True') @ (20, 40)
Token 43 ('while') @ (20, 45)
Token 22 ('123') @ (21, 1)
Token 43 ('while') @ (21, 4)
Token 24 ('a string literal') @ (21, 9)
Token 25 ('while123') @ (22, 1)
Token 25 ('_while') @ (22, 10)
Token -1 ('!') @ (22, 17)
Token 42 ('True') @ (22, 18)
Token 0 ('$') @ (26, 1)


******************************************************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
*******************************************************



** Well done, no logic or memory errors! **



** End of Test 7 **
********************************************************************************
```

Excellent, perfect score!

Test 1: test01.py

Test 1: test01.py from handout (Appendix A) -- yay, output correct!

Test 2: test02.py

Test 2: test02.py from handout (Appendix A) -- yay, output correct!

Test 3: test03.py

Test 3: test03.py from handout (Appendix A) -- yay, output correct!

Test 4: test04.py

Test 4: test04.py, punctuation and keywords -- yay, output correct!

Test 5: test05.py

Test 5: test05.py, string literals -- yay, output correct!

Test 6: test06.py

Test 6: test06.py, integer and real literals -- yay, output correct!

Test 7: test07.py

Test 7: test07.py, all tokens wtih lots of edge cases -- yay, output correct!

Submitted Files

▾ scanner.c

Download

```c
/*scanner.c*/

//
// << WHAT IS THE PURPOSE OF THIS FILE??? >>
// Scan input to classify as tokens, with suitable token ID, line num and col num
// << WHAT IS YOUR NAME >> Ishan Mukherjee
// << WHAT SCHOOL IS THIS >> Northwestern University
// << WHAT COURSE IS THIS >> CS 211
// << WHAT QUARTER IS THIS >> Winter 2024
//
// Starter code: Prof. Joe Hummel
//

#include <stdio.h>
#include <stdbool.h>  // true, false
#include <ctype.h>    // isspace, isdigit, isalpha
#include <string.h>   // strcmp
#include <assert.h>   // assert

#include "scanner.h"


//
// collect_identifier
//
// Given the start of an identifier, collects the rest into value
// while advancing the column number.
//
static void collect_identifier(FILE* input, int c, int* colNumber, char* value)
{
  assert(isalpha(c) || c == '_');  // should be start of an identifier

  int i = 0;

  while (isalnum(c) || c == '_')  // letter, digit, or underscore
  {
    value[i] = (char)c;  // store char
    i++;

    (*colNumber)++;  // advance col # past char

    c = fgetc(input);  // get next char
  }

  // at this point we found the end of the identifer, so put
  // that last char back for processing next:
```

```
  ungetc(c, input);

  // turn the value into a string:
  value[i] = '\0';  // build C-style string:

  return;
}

//
// collect_real_or_integer_literal
//
// Given the start of a real or integer literal, collects the rest into value
// while advancing the column number.
//

static char collect_real_or_integer_literal(FILE* input, int c, int* colNumber, char* value)
{
  assert(isdigit(c) || c == '.');  // should be start of an identifier
  int num_decimals = 0;
  // return value:
  // '.' if a simple dot
  // 'i' if an int literal
  // 'r' if a real literal
  // default assumption is simple dot
  char is_dot_int_or_real = '.';
  int i = 0;

  while (isdigit(c) || (c == '.' && num_decimals < 1))  // digit or decimal point (and number of decimal points is below 1, since real literals can't have multiple decimal points)
  {
    if (c == '.')
    {
      num_decimals++;
    }
    else
    {
      is_dot_int_or_real = 'i';
    }
    value[i] = (char) c;  // store char
    i++;

    (*colNumber)++;  // advance col # past char

    c = fgetc(input);  // get next char
  }

  // at this point we found the end of the real literal, so put
  // that last char back for processing next:
  ungetc(c, input);

  // turn the value into a string:
  value[i] = '\0';  // build C-style string:

  if (num_decimals > 0 && is_dot_int_or_real != '.')
  {
    is_dot_int_or_real = 'r';
  }

  return is_dot_int_or_real;
}

//
// collect_string_literal
//
// Given the start of a string literal, collects the rest into value
// while advancing the column number.
//

static bool collect_string_literal(char c,FILE* input,int* colNumber, int* lineNumber, char* value)
{
  char start_quote_type = c;
  bool string_terminated = true;

  c = fgetc(input); // first character of string literal
  (*colNumber)++;

  int i = 0;

  while (c != start_quote_type)  // termination of string literal
  {
    if (c == '\n')
    {
      string_terminated = false;
      (*lineNumber)++;  // next line, restart column:
      (*colNumber) = 1;
      break;
    }

    value[i] = (char) c;  // store char
    i++;

    (*colNumber)++;  // advance col # past char

    c = fgetc(input);  // get next char
  }
  if (string_terminated)
  {
    (*colNumber)++; // accounting for closing quote
  }

  // turn the value into a string:
  value[i] = '\0';  // build C-style string:

  return string_terminated;
}

//
// collect_single_char
//
// Given a single character-long punctuation, collects it into value
// while advancing the column number, and returns the appropriate Token.
//
struct Token collect_single_char(char c, int token_id, struct Token T, int* lineNumber, int* colNumber, char*value)
{
  T.id = token_id;
  T.line = *lineNumber;
  T.col = *colNumber;

  (*colNumber)++;  // advance col # past char

  value[0] = (char)c;
  value[1] = '\0';

  return T;
}

//
// collect_double_char
//
// Given a potential double character-long punctuation, collects it into value
// while advancing the column number, and returns the appropriate Token.
//
struct Token collect_double_char(char c, char second_char, int single_char_token_id, int double_char_token_id, struct Token T, int* lineNumber, int* colNumber, char*value, FILE* input)
{
  //
  // could be single or double char, let's assume single char for now:
  //
  T.id = single_char_token_id;
  T.line = *lineNumber;
  T.col = *colNumber;

  (*colNumber)++;  // advance col # past char

  value[0] = (char) c;
  value[1] = '\0';

  //
  // now let's read the next char and see what we have:
  //
  c = fgetc(input);

  if (c == second_char)  // it's double char
  {
    T.id = double_char_token_id;

    (*colNumber)++;  // advance col # past char

    value[1] = (char) c;
    value[2] = '\0';

    return T;
  }

  //
  // if we get here, then next char did not
  // form a token, so we need to put the char
  // back to be processed on the next call:
  //
  ungetc(c, input);

  return T;
}


//
// scanner_init
//
// Initializes line number, column number, and value before
// the start of processing the input stream.
//
void scanner_init(int* lineNumber, int* colNumber, char* value)
{
  assert(lineNumber != NULL);
  assert(colNumber != NULL);
  assert(value != NULL);

  *lineNumber = 1;
  *colNumber = 1;
  value[0] = '\0';  // empty string
}


//
// scanner_nextToken
```

```
//
// Returns the next token in the given input stream, advancing the line
// number and column number as appropriate. The token's string-based
// value is returned via the "value" parameter. For example, if the
// token returned is an integer literal, then the value returned is
// the actual literal in string form, e.g. "456". For an identifer,
// the value is the identifer itself, e.g. "print" or "y". For a
// string literal such as 'hi class', the value is the contents of the
// string literal without the quotes.
//
struct Token scanner_nextToken(FILE* input, int* lineNumber, int* colNumber, char* value)
{
  assert(input != NULL);
  assert(lineNumber != NULL);
  assert(colNumber != NULL);
  assert(value != NULL);

  struct Token T;

  //
  // repeatedly input characters one by one until a token is found:
  //
  while (true)
  {
    //
    // Get the next input character:
    //
    int c = fgetc(input);

    //
    // Let's see what we have...
    //

    if (c == EOF || c == '$')  // no more input, return EOS:
    {
      T.id = nuPy_EOS;
      T.line = *lineNumber;
      T.col = *colNumber;

      value[0] = '$';
      value[1] = '\0';

      return T;
    }
    else if (c == '\n')  // end of line, keep going:
    {
      (*lineNumber)++;  // next line, restart column:
      *colNumber = 1;
      continue;
    }
    else if (isspace(c))  // other form of whitespace, skip:
    {
      (*colNumber)++;  // advance col # past char
      continue;
    }
    else if (c == '(')
    {
      return collect_single_char(c, nuPy_LEFT_PAREN, T, lineNumber, colNumber, value);
    }
    else if (c == ')')
    {
      return collect_single_char(c, nuPy_RIGHT_PAREN, T, lineNumber, colNumber, value);
    }
    // identifier or keyword
    else if (c == '_' || isalpha(c))
    {
      //
      // start of identifier or keyword, let's assume identifier for now:
      //
      T.id = nuPy_IDENTIFIER;
      T.line = *lineNumber;
      T.col = *colNumber;

      collect_identifier(input, c, colNumber, value);

      //
      // TODO: is the identifier a keyword? If so, return that
      // token id instead.
      //
      // array of keywords represented as strings
      char *keywords[] = {"and", "break", "continue", "def", "elif", "else", "False", "for", "if", "in", "is", "None", "not", "or", "pass", "return", "True", "while"};
      // iterate through the keywords array
      for (int i = 0, list_len = sizeof(keywords) / sizeof(keywords[0]); i < list_len; i++)
      {
        if (strcmp(keywords[i], value) == 0)
        {
          // id = (index of "and" keyword in the enum) + (loop iterator)
          T.id = nuPy_KEYW_AND + i;
        }
      }

      return T;
    }
    // * or **
    else if (c == '*')
    {
      return collect_double_char(c, '*', nuPy_ASTERISK, nuPy_POWER, T, lineNumber, colNumber, value,
input);
    }
    //
    //
    // TODO: all the remaining tokens (punctuation, literals), and
    // also need to handle line comments.
    //
    //
    else if (c == '#')
    {
      while (c != '\n')
      {
        c = fgetc(input);
      }
      (*lineNumber)++;
      (*colNumber) = 1;
      continue;
    }
    else if (c == '[')
    {
      return collect_single_char(c, nuPy_LEFT_BRACKET, T, lineNumber, colNumber, value);
    }
    else if (c == ']')
    {
      return collect_single_char(c, nuPy_RIGHT_BRACKET, T, lineNumber, colNumber, value);
    }
    else if (c == '{')
    {
      return collect_single_char(c, nuPy_LEFT_BRACE, T, lineNumber, colNumber, value);
    }
    else if (c == '}')
    {
      return collect_single_char(c, nuPy_RIGHT_BRACE, T, lineNumber, colNumber, value);
    }
    else if (c == '+')
    {
      return collect_single_char(c, nuPy_PLUS, T, lineNumber, colNumber, value);
    }
    else if (c == '-')
    {
      return collect_single_char(c, nuPy_MINUS, T, lineNumber, colNumber, value);
    }
    else if (c == '/')
    {
      return collect_single_char(c, nuPy_SLASH, T, lineNumber, colNumber, value);
    }
    else if (c == '%')
    {
      return collect_single_char(c, nuPy_PERCENT, T, lineNumber, colNumber, value);
    }
    else if (c == '&')
    {
      return collect_single_char(c, nuPy_AMPERSAND, T, lineNumber, colNumber, value);
    }
    else if (c == ':')
    {
      return collect_single_char(c, nuPy_COLON, T, lineNumber, colNumber, value);
    }
    // = or ==
    else if (c == '=')
    {
      return collect_double_char(c, '=', nuPy_EQUAL, nuPy_EQUALEQUAL, T, lineNumber, colNumber, value, input);
    }
    // < or <=
    else if (c == '<')
    {
      return collect_double_char(c, '=', nuPy_LT, nuPy_LTE, T, lineNumber, colNumber, value, input);
    }
    // > or >=
    else if (c == '>')
    {
      return collect_double_char(c, '=', nuPy_GT, nuPy_GTE, T, lineNumber, colNumber, value, input);
    }
    // ! (unknown token) or !=
    else if (c == '!')
    {
      return collect_double_char(c, '=', nuPy_UNKNOWN, nuPy_NOTEQUAL, T, lineNumber, colNumber, value, input);
    }
    // string literals
    else if (c == '"' || c == '\'')
    {
      T.id = nuPy_STR_LITERAL;
      T.line = *lineNumber;
      T.col = *colNumber;

      bool string_terminated = collect_string_literal(c, input, colNumber, lineNumber, value);

      if (!string_terminated)
      {
        printf("**WARNING: string literal @ (%d, %d) not terminated properly\n", T.line, T.col);
      }

      return T;
    }
    // special case of real literals
    else if (c == '.')
    {
      T.id = nuPy_UNKNOWN; // default assumption that "." is unknown token
      T.line = *lineNumber;
      T.col = *colNumber;

      char type_id = collect_real_or_integer_literal(input, c, colNumber, value);
      if (type_id == 'r')
      {
        T.id = nuPy_REAL_LITERAL;
      }

      return T;
    }
    // real or integer literals
    else if (isdigit(c))
    {
      T.id = nuPy_INT_LITERAL; // default assumption
      T.line = *lineNumber;
      T.col = *colNumber;

      char type_id = collect_real_or_integer_literal(input, c, colNumber, value);
      if (type_id == 'r')
      {
        T.id = nuPy_REAL_LITERAL;
      }

      return T;
    }
    // unknown token
    else
    {
      //
      // if we get here, then char denotes an UNKNOWN token:
      //
      return collect_single_char(c, nuPy_UNKNOWN, T, lineNumber, colNumber, value);
    }

  }//while

  //
  // execution should never get here, return occurs
  // from within loop
  //
}

```