CS177 Python Programming

Recitation 7
Loops, Debugging

Agenda

- Revisit Loops.
- What are computer bugs?
- When are computer bugs discovered?
  - Compile Time and Runtime
- What kind of bugs are discovered?
  - Syntax, Arithmetic and Logic Errors
- Are there tools to help us find bugs?
  - Print statements and Python debugger

For Loop

- Definite Loops are implemented with for loops
- *For* loops are traditionally used when you have a piece of code which you want to repeat *a fixed* number of times.
- The general form of for loop:

```
For <var> in <sequence>:
    <body>
```

- *var* is called the loop index, it takes consecutive values listed in *sequence*.
- There are two forms of for loop
  - **for i in range(INTEGER)**
  - **for item in (LIST/STRING)**

For Loop

```
myList = range(3)
for i in myList:
    print(i)

>>
0
1
2
```

```
for i in range(3):
    print(i)


>>
0
1
2
```

```
str = ‘Hello’
for c in str:
    print(c)

>>
H
e
l
l
o
```

```
str = ‘Hello’
For i in
range(len(Str)):
    print(str(i))
>>
H
e
l
l
o
```

While Loop

- The general form of a `while` loop:

```
while <condition>:
    <body>
```

- The *`condition`* is a Boolean expression.
- The `while` will keep looping executing the *`body`* as long as the *`condition`* is `True`.

While Loop

- To prevent an infinite loop, the `condition` of the loop must depend on the body, so that after looping couple of times the condition will be evaluated to `False` and the loop would terminate.

- Infinite loop

```
    i = 0
    while True:
        i = i + 1
        print (i)
>>
1
2
3
.
.
```

- Not infinite loop

```
   i = 0
    while i<5:
        i = i + 1
        print (i)
>>
1
2
3
4
5
```

Break & Continue

- `Break` terminates the loop
- `Continue` terminates the current iteration

```
xlist = [2,4,-1,8]
for num in xlist:
    if (num <0):
        break
    print (num)


>>
2
4
```

```
xlist = [2,4,-1,8]
for num in xlist:
    if (num <0):
        continue
    print (num)


>>
2
4
8
```

Nested Loops

- We can define a loop within another loop.
- Each single iteration for the outer loop, all the iterations of the inner loop will be executed.

```
for i in range(4):        #The outer loop
    for j in range(2):    #The inner loop
            print (i,j)
```

```
>>
0 0
0 1
1 0
1 1
2 0
2 1
3 0
3 1
```

- The number of times the print statement is executed = len(range(4))*len(range(2))

Nested Loops

Nested loops are suitable when working with nested lists.
E.g. Given a nested list, print the sum of each inner list.

```
def main():
    myList = [[1,2,3],[10,-5,20],[40]]
    for lst in myList:
        sum = 0
        for number in lst:
            sum = sum + number
        print (sum)
main()
```

```
>>
6
30
40
```

Nested Loops

Given a nested list, add all positive numbers per inner list and print the sum.

```
def main():
    myList = [[1,2,3],[10,-5,20],[40]]
    for lst in myList:
        sum = 0
        for number in lst:
            if(number < 0):
                continue
            sum = sum + number
        print (sum)

main()
```

CORRECT

```
>>
6
30
40
```

```
def main():
    myList = [[1,2,3],[10,-5,20],[40]]
    for lst in myList:
        sum = 0
        for number in lst:
            if(number < 0):
                break
            sum = sum + number
        print (sum)

main()
```

WRONG

```
>>
6
10
40
```

Debugging

Early computers used vacuum tubes. The tubes would get hot and attracted moths. A moth was zapped and interfered with the circuitry. The bug had to be removed to fix the computer. Some say this is how the word “debugging” came into use.

Debugging

- What is a computer bug?
  - A computer bug is a problem that causes a computer to produce an incorrect or unexpected result.

Debugging

- Computer bugs can manifest themselves at different phases of the program execution including:
  - Compile Time (Easy to catch)
  - Runtime (Harder to catch)

When Are Bugs Discovered?

Compile Time, Load Time, & Runtime Bugs

Compile Time Bug

- Compile time bugs show up when the source code is converted into computer code
- A common compile time error is:
  - Syntax Error
- A syntax error means that the source code does not meet the source code specification. For example:
  - Missing a ‘:’ at the end of you def statement

Compile Time Bug Example

```
>>> def t2()
SyntaxError: invalid syntax
```

- Notice the missing ‘:’
- When you run this statement, Python immediately knows that it is invalid code.

Load Time Bug

- Load time bugs, in the context of Python, often have to do with the libraries that are imported
  - The permissions may not be set correctly on an imported library file
  - The library file may not be found

Load Time Bug Example

```
>>> import foo
ImportError: No module named foo
```

- In this case a library named foo does not exist

Runtime Bug

- Runtime bugs show up when the code is executed
- A common runtime error is:
  - NameError
- A name error means that a function or variable was used that wasn't defined

Runtime Bug Example

```
def t1():
   print(a)


>>> t1()
```

- Notice that the variable 'a' is not defined
- When you save the file, Python does not report an error, but when you call the function an error pops up.

Runtime Bug Example

```
def t1():
  print(a)


>>> t1()
NameError: global name 'a' is
not defined
```

- The NameError is produced when the t1 function is called

What are Some Common Bugs?

Syntax, Arithmetic, and Logic Errors

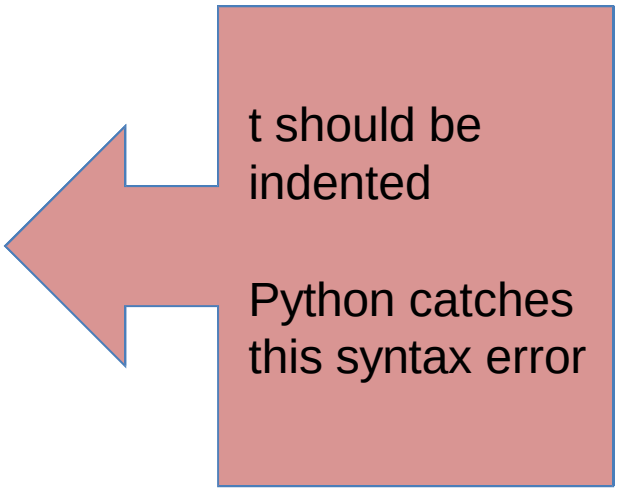
Syntax Bugs

- Syntax errors are often discovered by Python at compile time but not always
- Likely you have encountered many of these:
  - Incorrect indentation
  - Missing elements (like ':')

Syntax Bug

- Incorrect indentation:

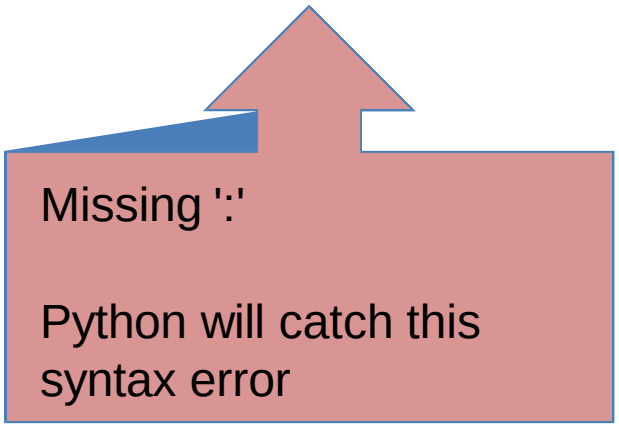
```
def t1():
t = 1


Invalid syntax
```



Syntax Bug

- Missing colon:

```
>>> def t1()
SyntaxError: invalid syntax
```



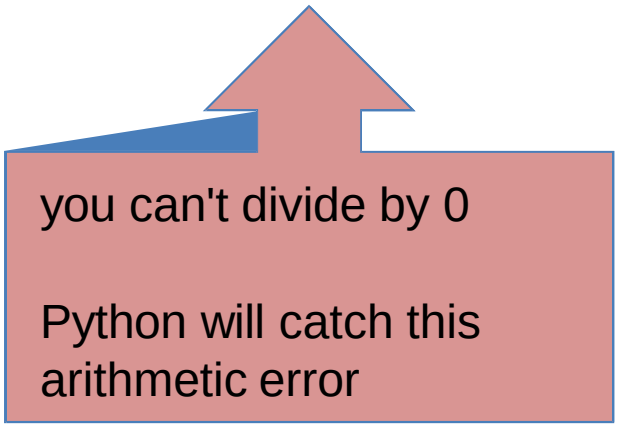
Arithmetic Bugs

- We will only focus on one, but a few more exist.
- One important arithmetic bug is a divide-by-zero error
  - By definition of division, you can't divide by zero

Arithmetic Bug

- Division by zero:

```
>>> 4/0
ZeroDivisionError: int division
or modulo by zero
```



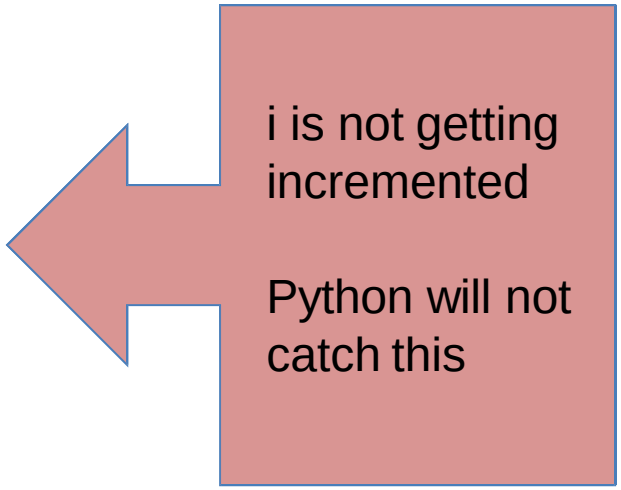
Logic Bugs

- Logic bugs are usually not caught automatically by the computer like Syntax Errors or Name Errors.
- The bug may be subtle and manifest itself in peculiar ways.
- Usually takes human source code analysis to track down the bug

Logic Bug

- Infinite loop:

```
i = 0
while(i<5):
    i = 1
```



Logic Bug

- Off-by-one error

```
x = [21,22,23,24]
i = 0
while i <= len(x):
    s = s + x[i]
    i = i + 1
IndexError: list index out of range
```

x[4] does not exist

Python catches this logic error

Find the bug ?!

```
a = 3
if(a=2):
    print(a)
```

a=2
is a syntax error
should be:
a==2

```
>>Traceback (most recent call last):
  File "python", line 2
    if(a=2):
```

Find the bug ?!

This program should add the numbers in a list.

```
def add(a,b):
      a=a+b
def main():
      myList = [5,4,3]
      sum = 0
      for i in myList:
            add(sum,myList[i])
      print(sum)
main()
```

Should be add(sum,i)
since i takes values: 5,4,3
there is no myList[5]

```
>>IndexError: list index out of
range
```

Find the bug ?!

```
def add(a,b):
    a=a+b
def main():
    myList = [5,4,3]
    sum = 0
    for i in myList:
        add(sum,i)
    print(sum)
main()
```

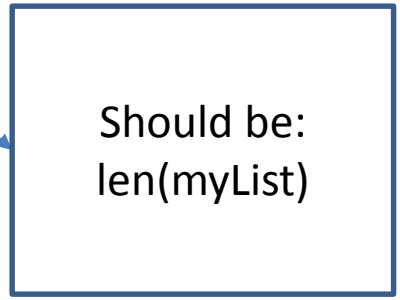
sum and i are immutables, so the value of sum will not change after calling add.

```
>>0
```

Find the bug ?!

This program should find the greatest value in a list:

```
def getMax(myList):
     max = 0
    for i in range(myList):
         if(myList[i]>myList[i+1]):
              return myList[i]

def main():
    myList = [1,5,3,4,7,2]
    print(getMax(myList))
main()
```




```
>>TypeError: 'list' object cannot be interpreted as an integer
```

Find the bug ?!

This program should find the greatest value in a list:

```
def getMax(myList):
     max = 0
    for i in range(len(myList)):
         if(myList[i]>myList[i+1]):
             return myList[i]

def main():
    myList = [1,5,3,4,7,2]
    print(getMax(myList))
main()
```

At i=5(last iteration)
myList[i+1] does not exist

```
>>IndexError: list index out of range
```

Find the bug ?!

This program should find the greatest value in a list:

```
def getMax(myList):
     max=0
    for i in range(len(myList)):
         if(myList[i]>max):
              max = myList[i]
     return max

def main():
    myList = [1,5,3,4,7,2]
    print(getMax(myList))
main()
```

NO BUG, for positive numbers !

>>7

Are there tools to help us find bugs?

Print Statements and Python Debugger

Print Statements

- Strategically places print() statements can be placed in the source code to verify values
- Advantage: Using print statements (or equivalents) to debug works in every language, no language specific tool must be learned
- Disadvantage: Not everything is printable

Using Print Statements

- Verfiy input and output

```
def sort3(x, y, z):
    print("Input: x=",x,"y=",y,"z=",z)
    r = sorted([x,y,z])
    print("Output:", r)

>>> sort3(8, 11, 3)
Input: x= 8 y= 11 z= 3
Output: [3, 8, 11]
```

Using Print Statements

- Print intermediate live values

```
def t():
    s = 0
    for i in range(3):
        ns = s + i
        print(ns, "=", s, "+", i)
        s = ns

>>> t()
0 = 0 + 0
1 = 0 + 1
3 = 1 + 2
```

Python Debugger

- Many programming languages have debuggers available
- A debugger lets you analyze the program state after each statement
  - Called stepping

Python Debugger

- To launch the Python debugger from IDLE:
  - From the IDLE command window choose the menu: Debug->Debugger
  - Your command window will show [DEBUG ON]
  - Then run commands as normal and you should see the debugger window...

Python Debugger

- Options
  - Stack: Current running function
  - Source: Show me in the source what statement is currently running

  Locals: What are the values of the local variables
  - Global: What are the values of global variables

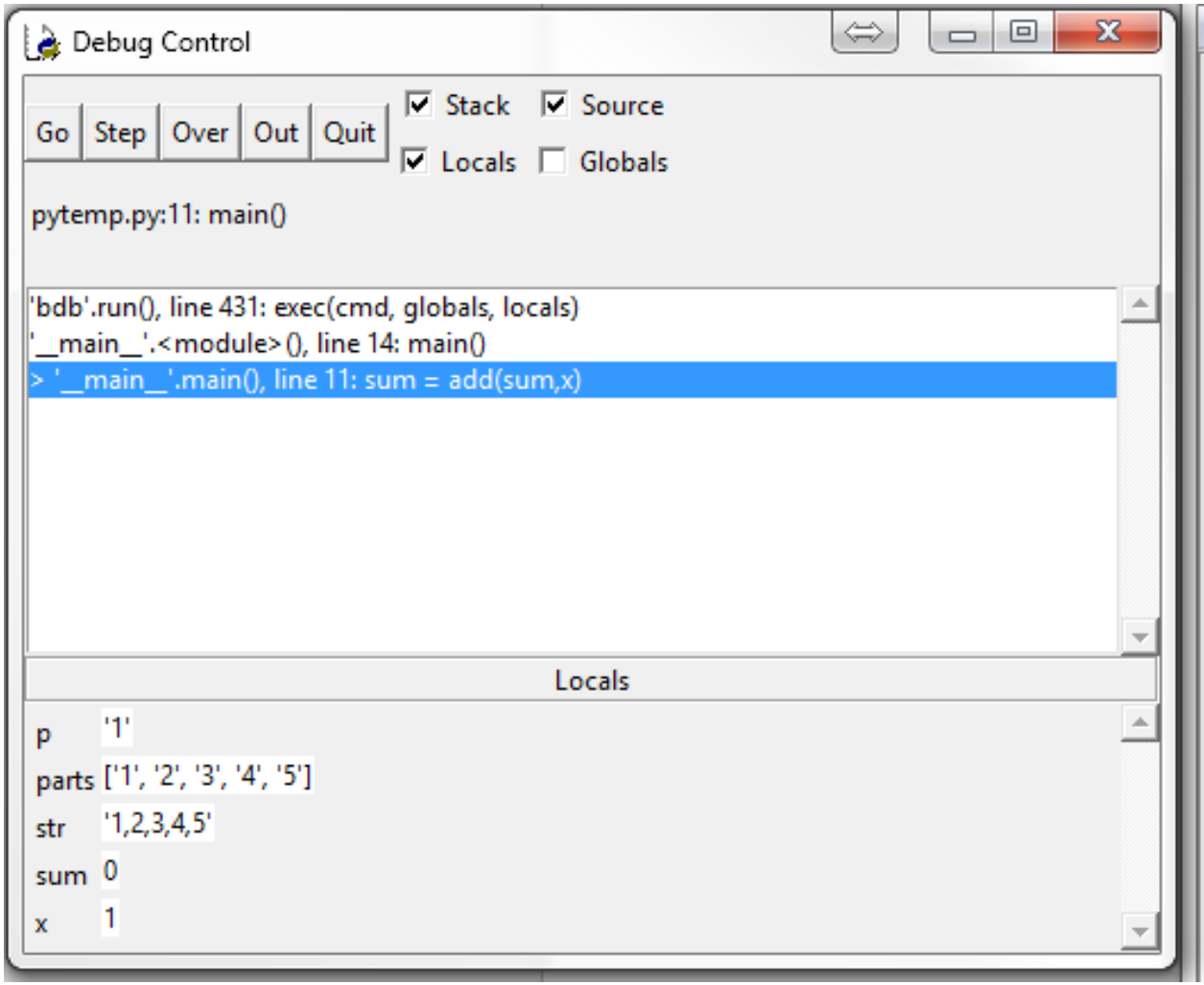
Python Debugger

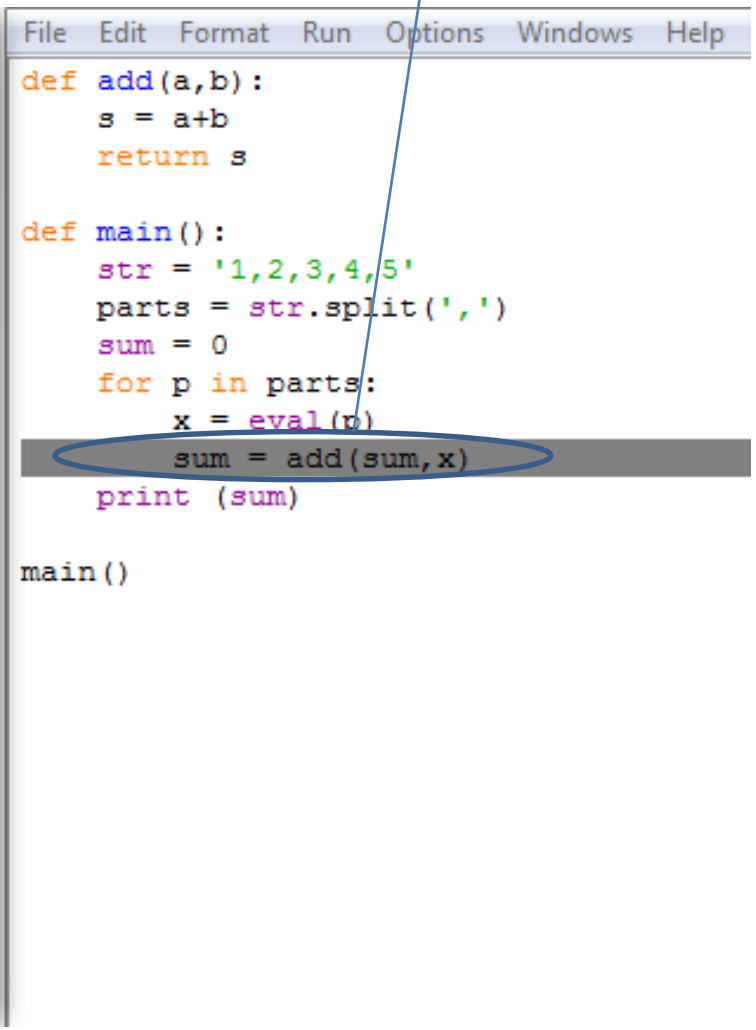
Debugging this code

```
def add(a,b):
    s = a+b
    return s

def main():
    str = '1,2,3,4,5'
    parts = str.split(',')
    sum = 0
    for p in parts:
        x = eval(p)
        sum = add(sum,x)
    print (sum)

main()
```

IDLE Debugger

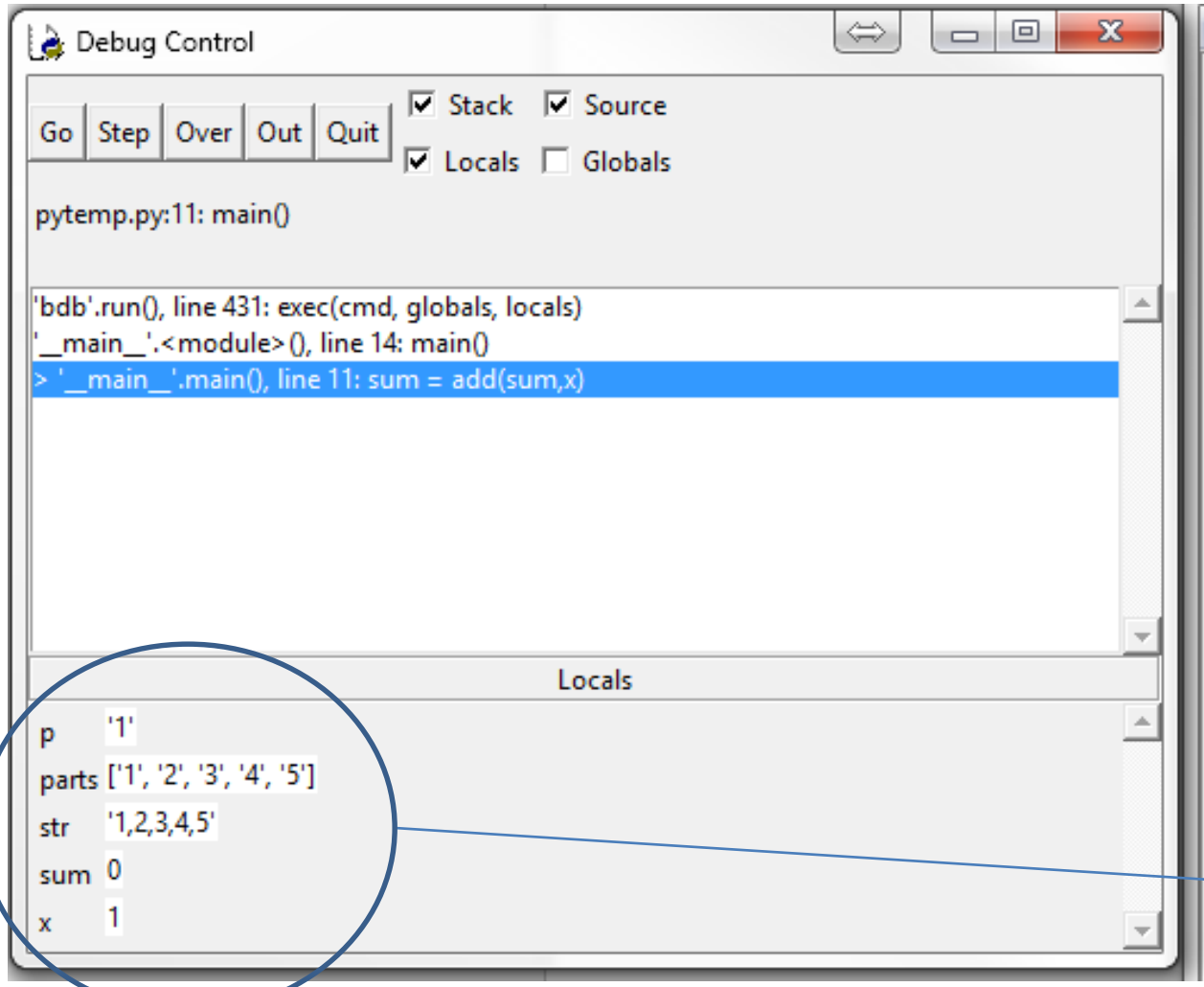
The next line to be executed

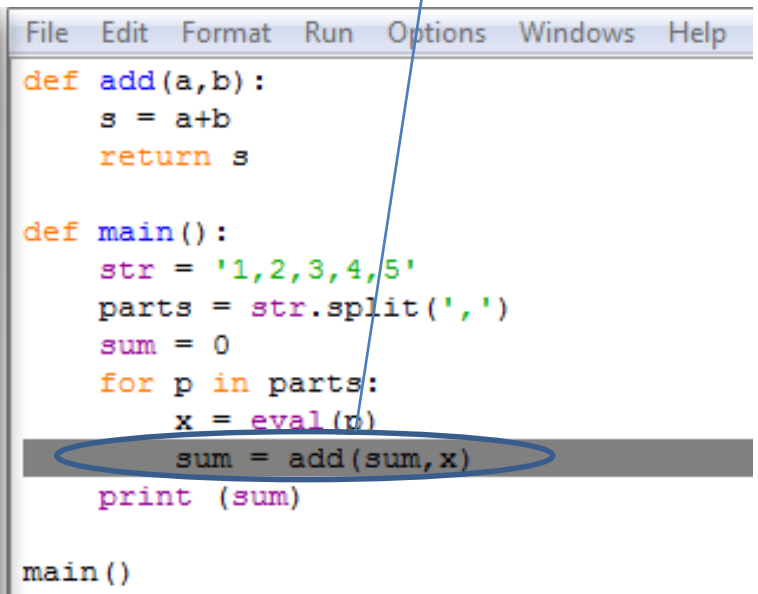




IDLE Debugger

The next line to be executed

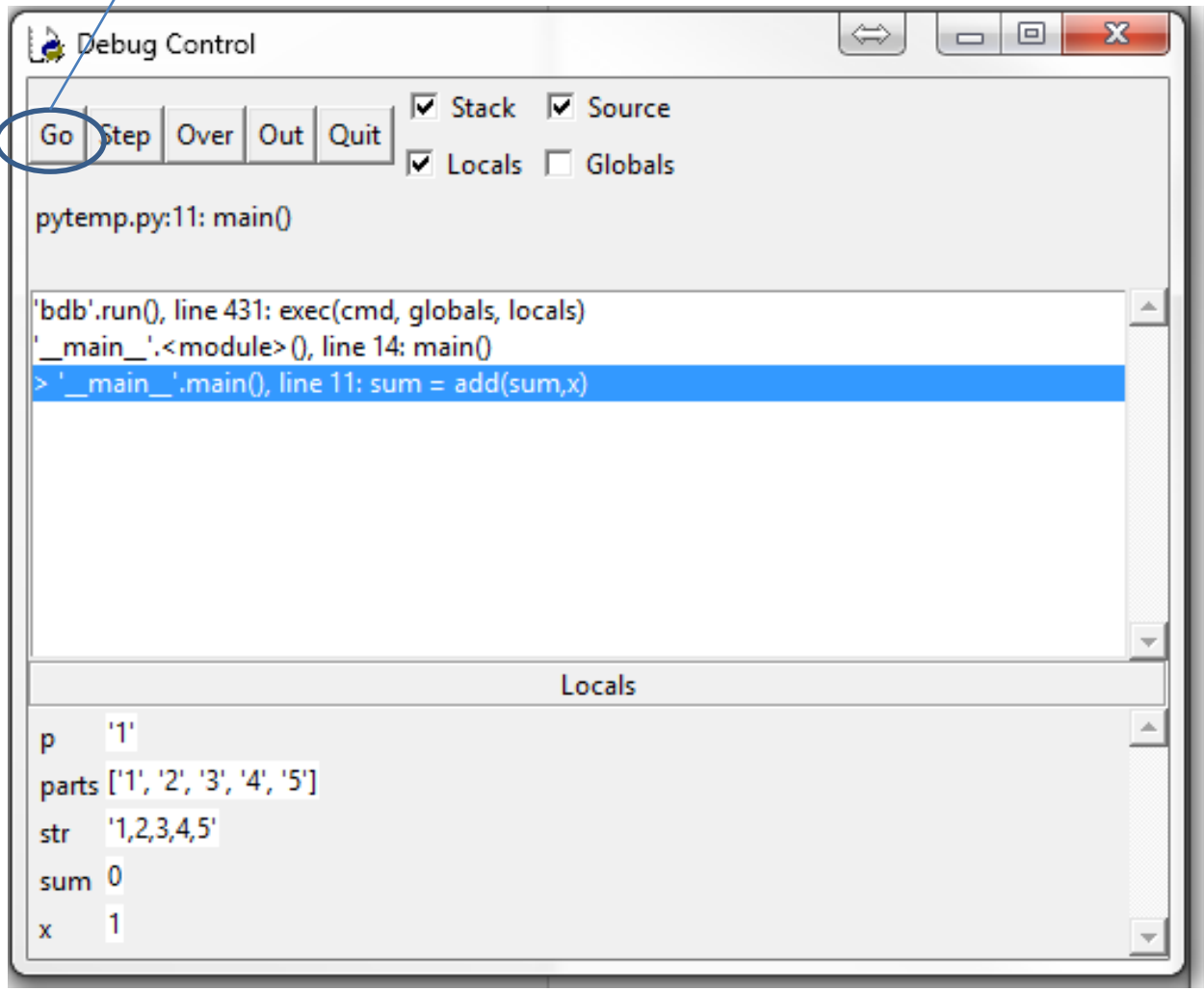


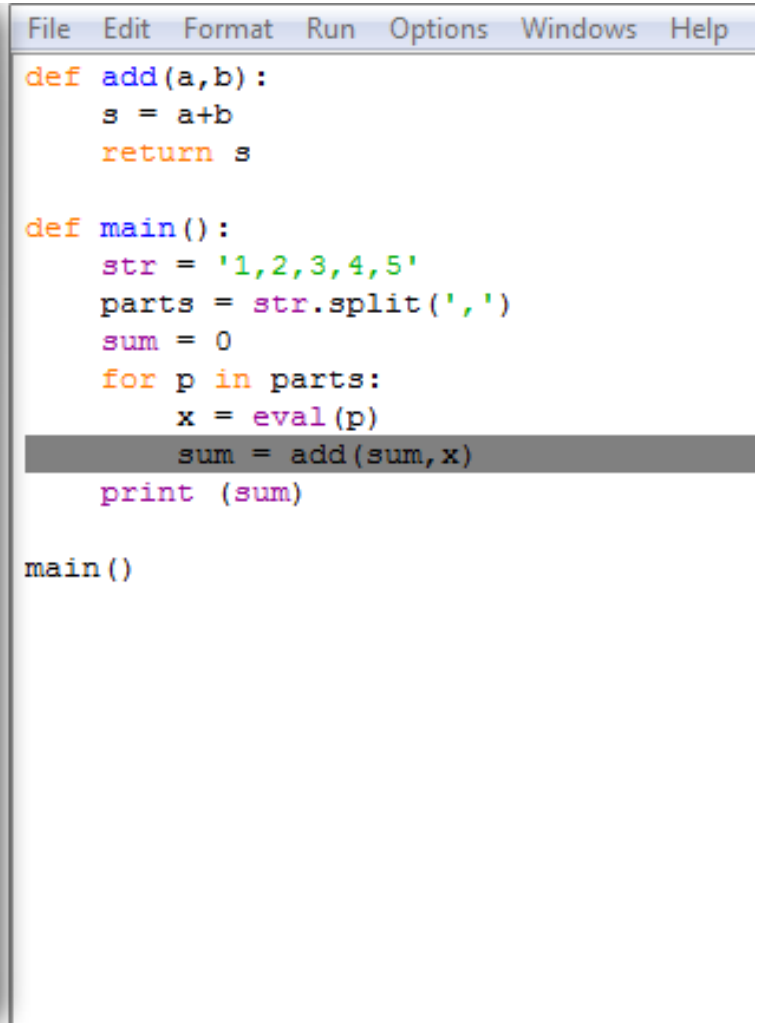


Current values of local variables before the execution of the gray line.

Finish the execution of the program

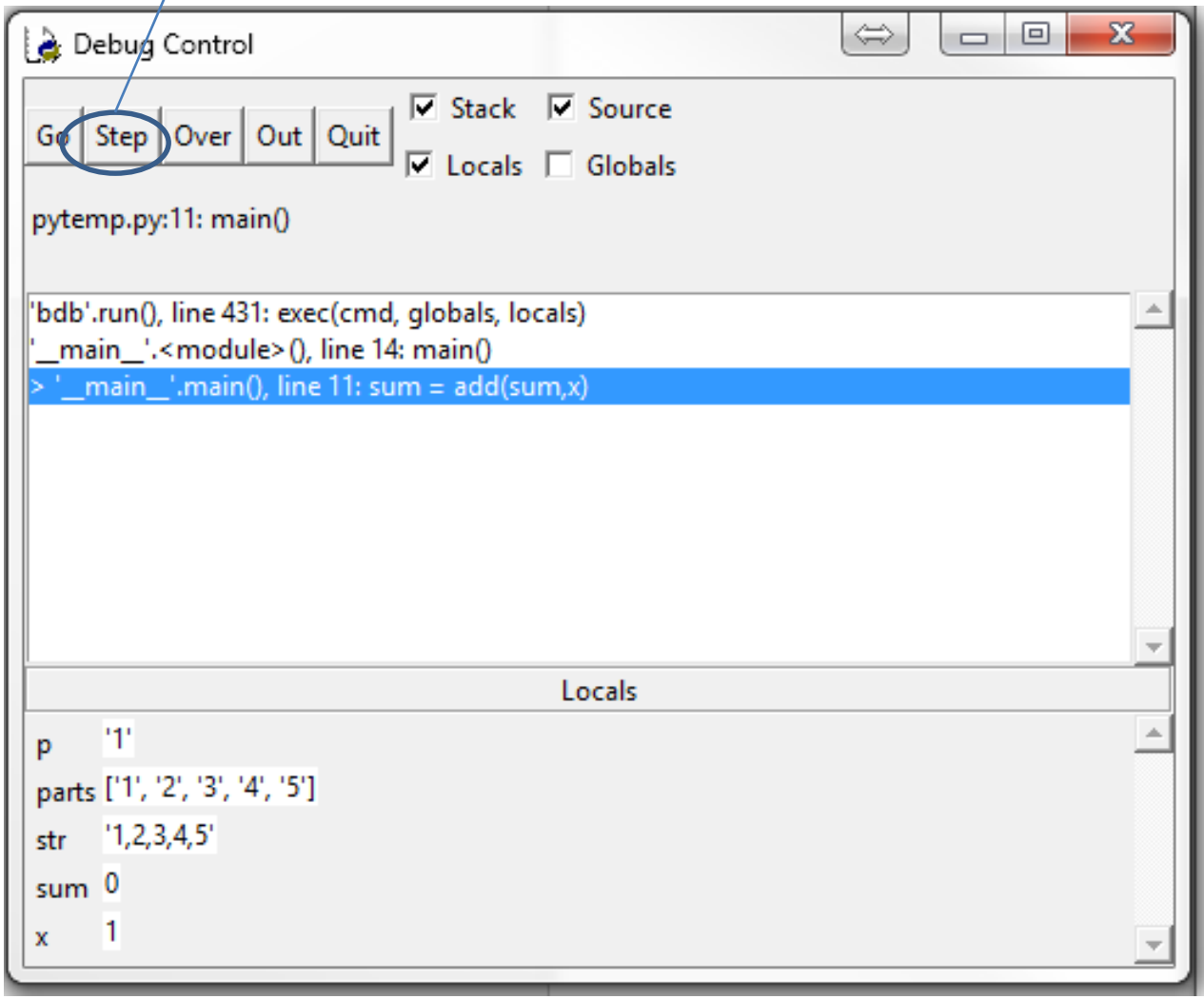
IDLE Debugger

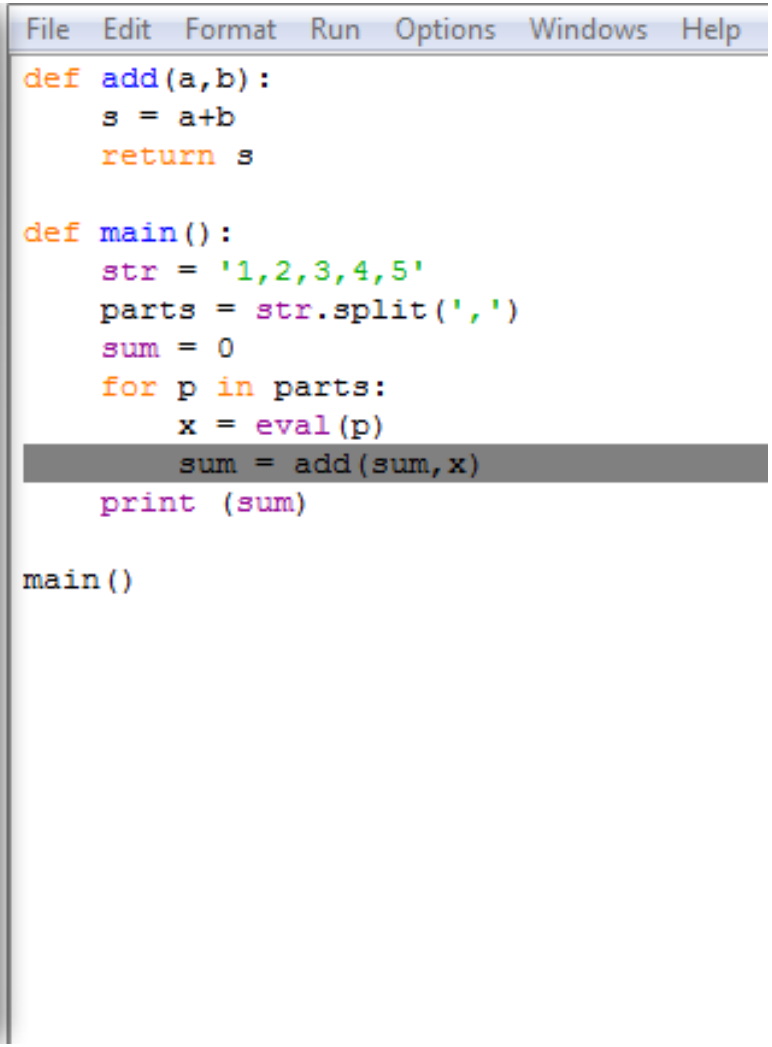




Step into. If the next line is a function call, step into will go to this function and walk through its execution line by line

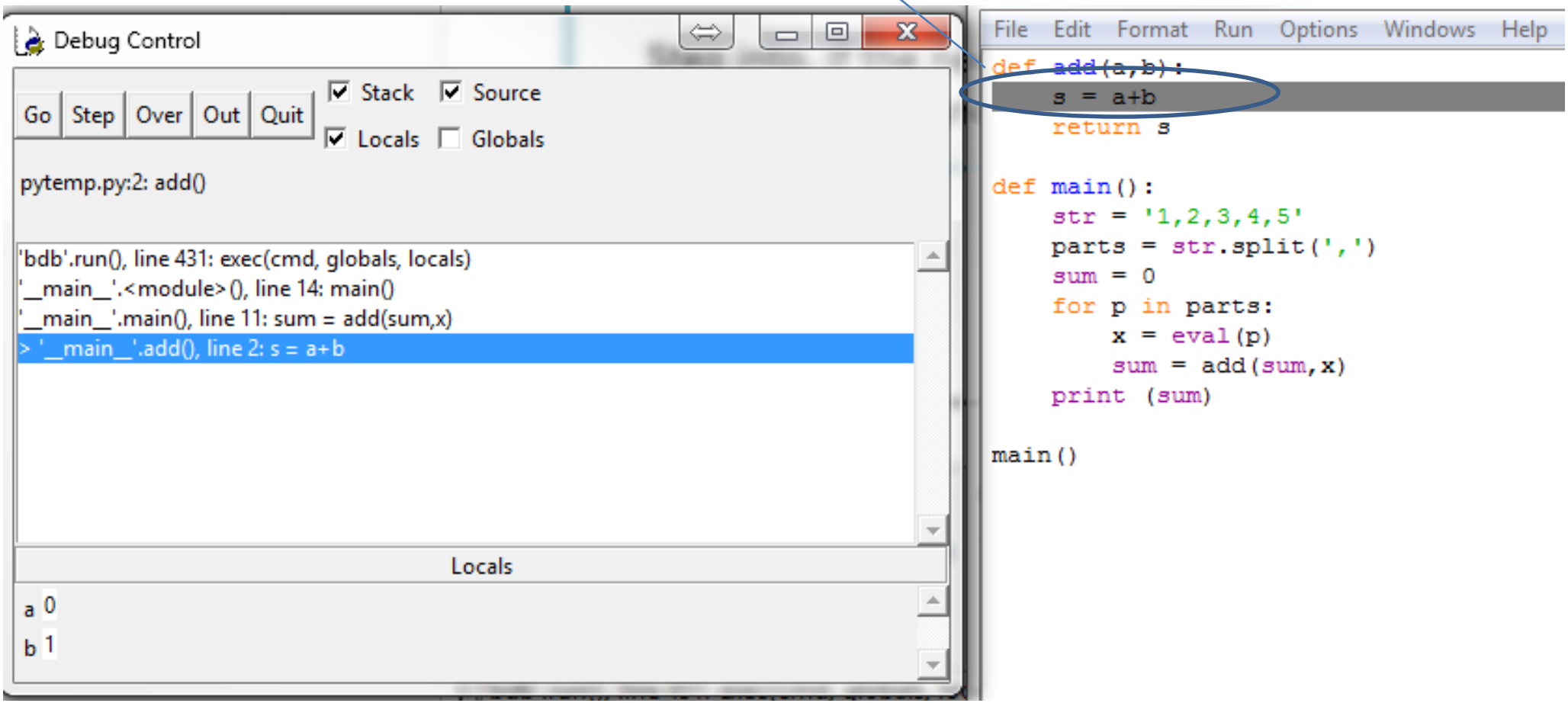
IDLE Debugger





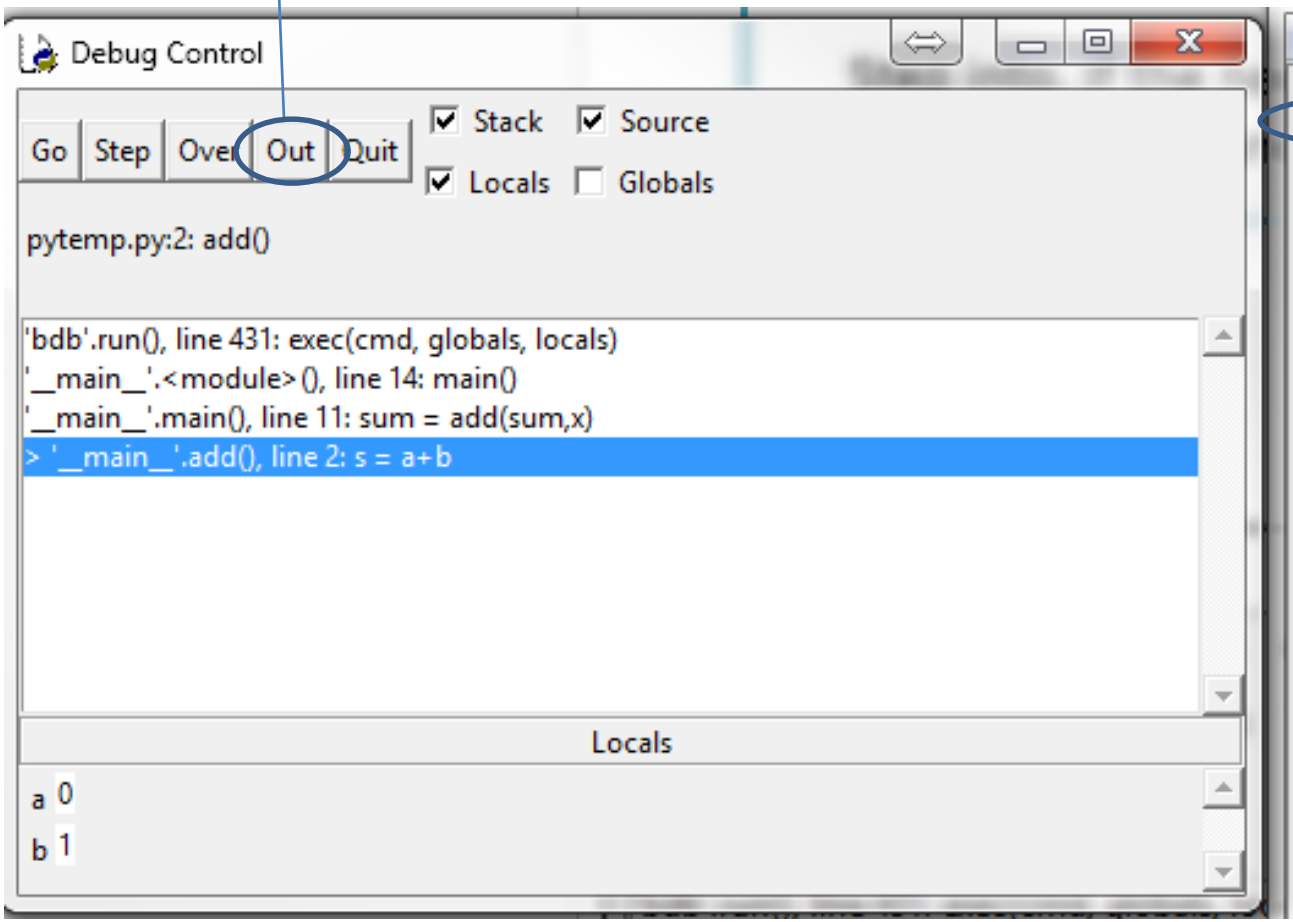
Stepped into the function, to execute it line by line.

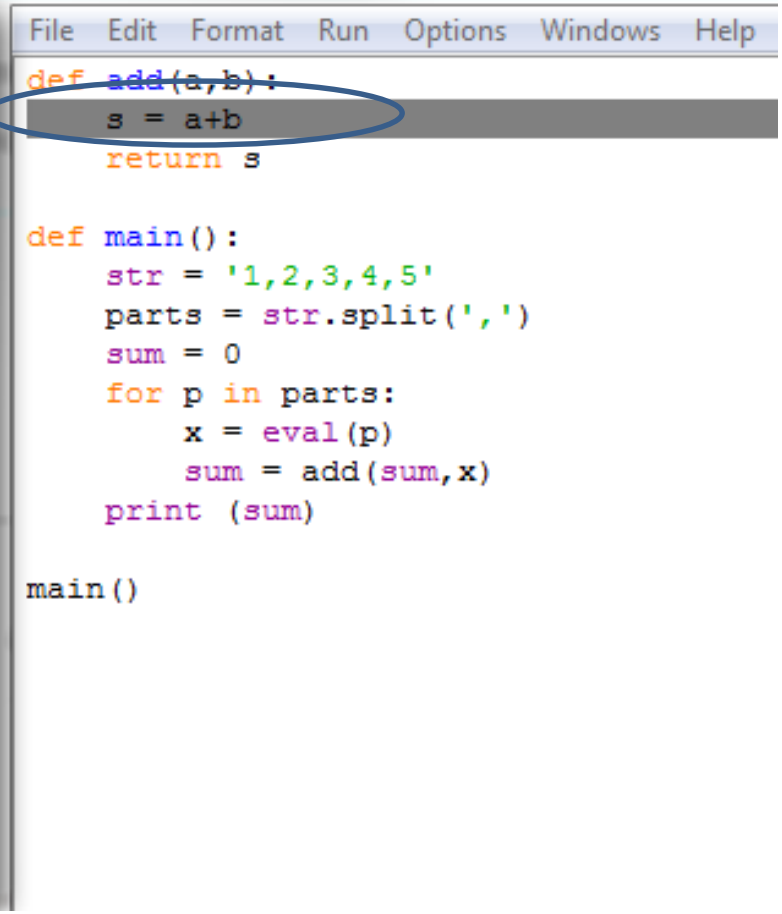
IDLE Debugger



Step out of the current function. Step out will continue the execution of the function and returns to its calling site.

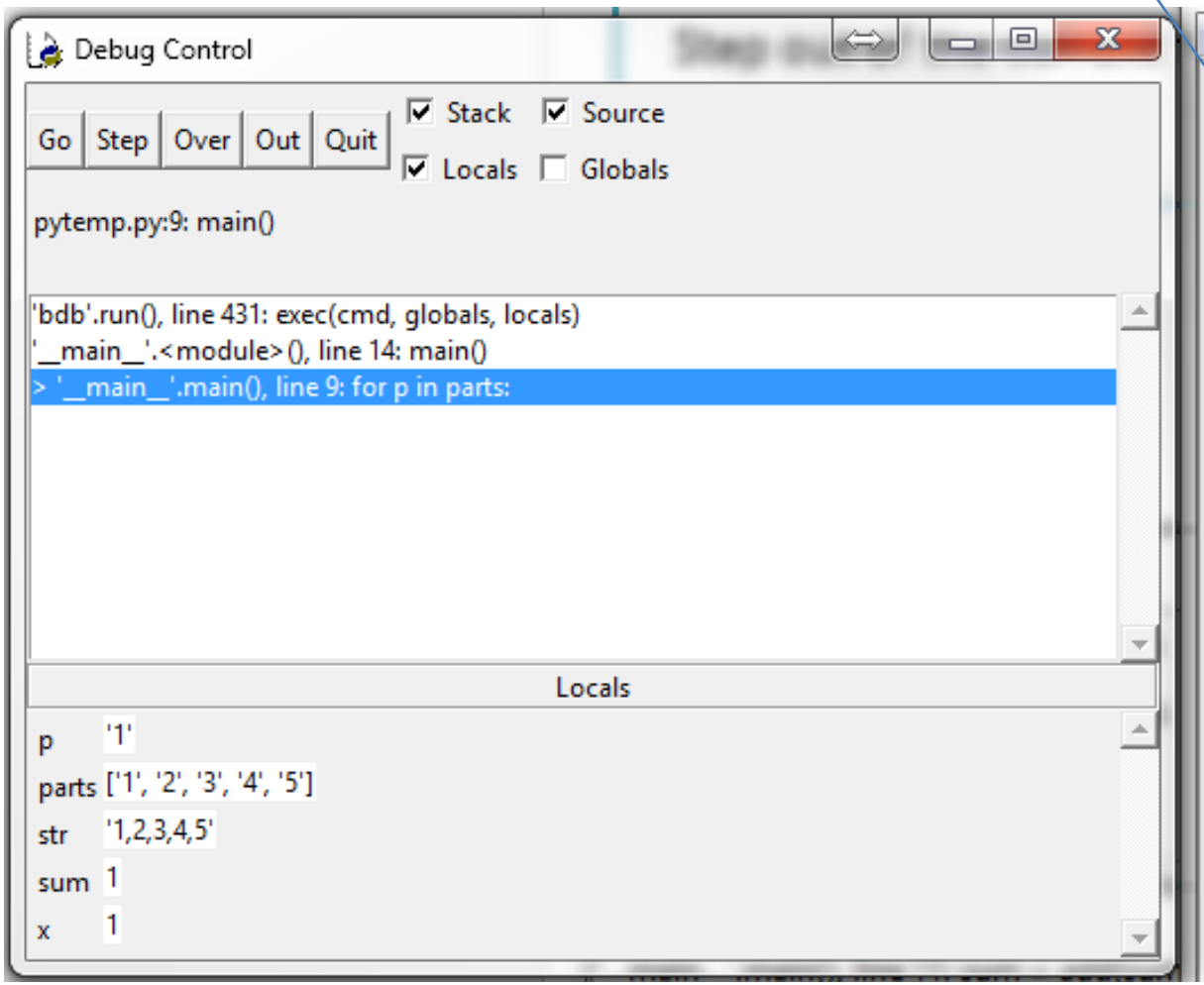
IDLE Debugger

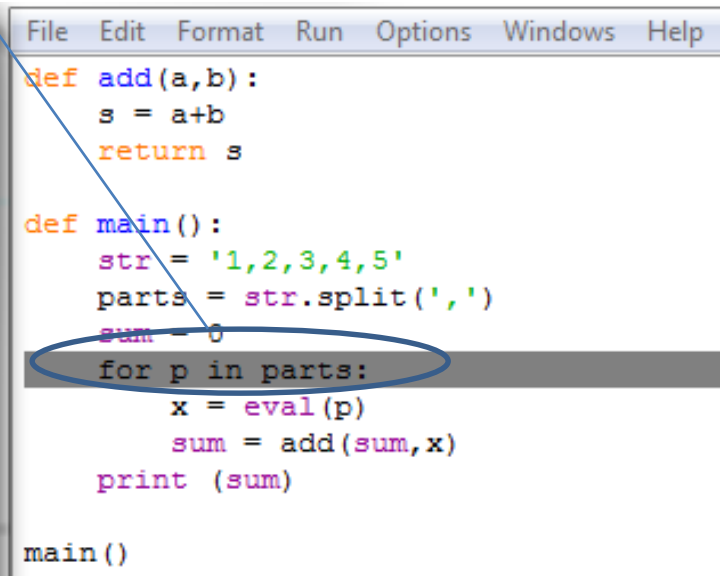




After stepping out, the flow returned to the main, to continue execution what comes after the function call.

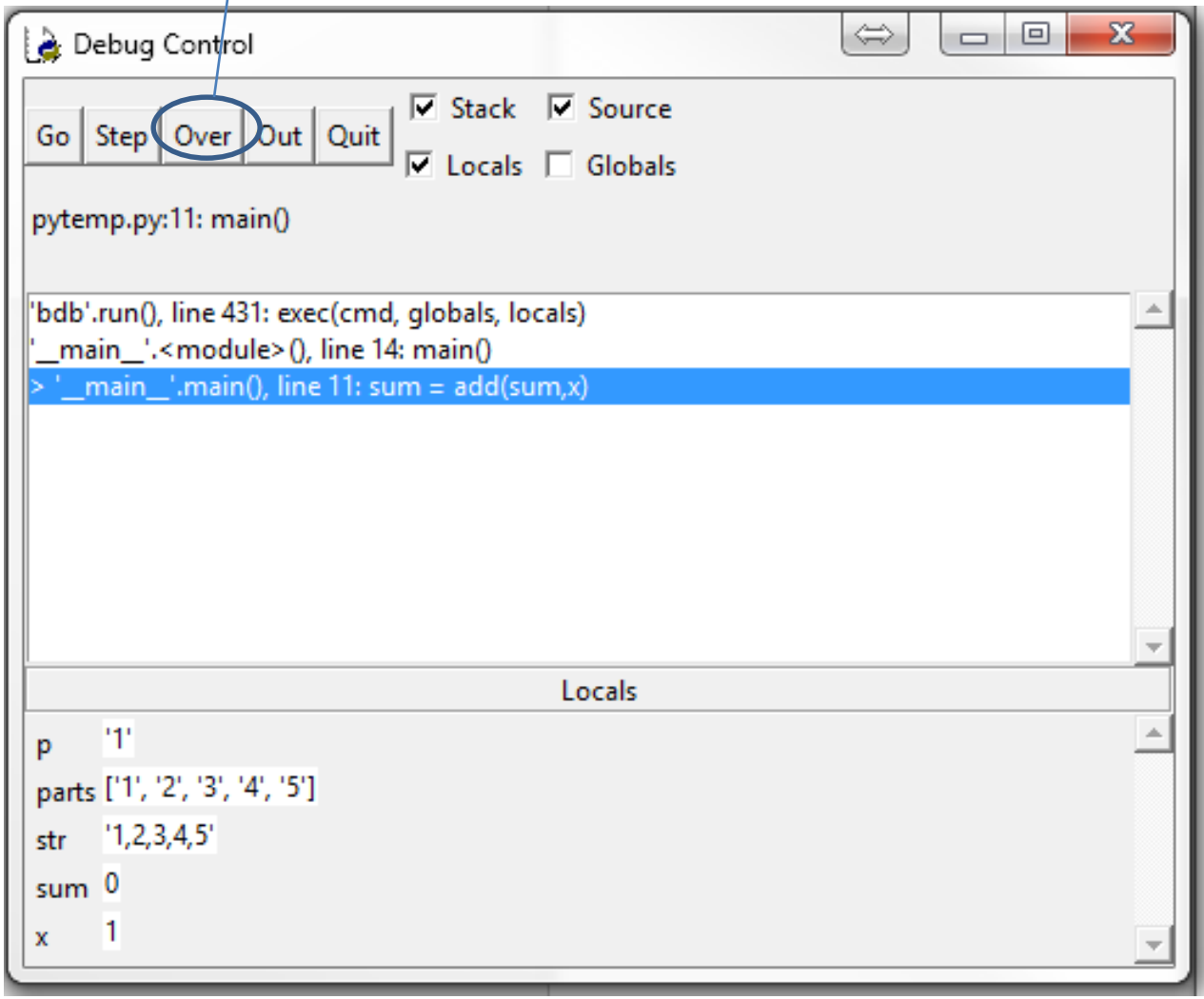
IDLE Debugger

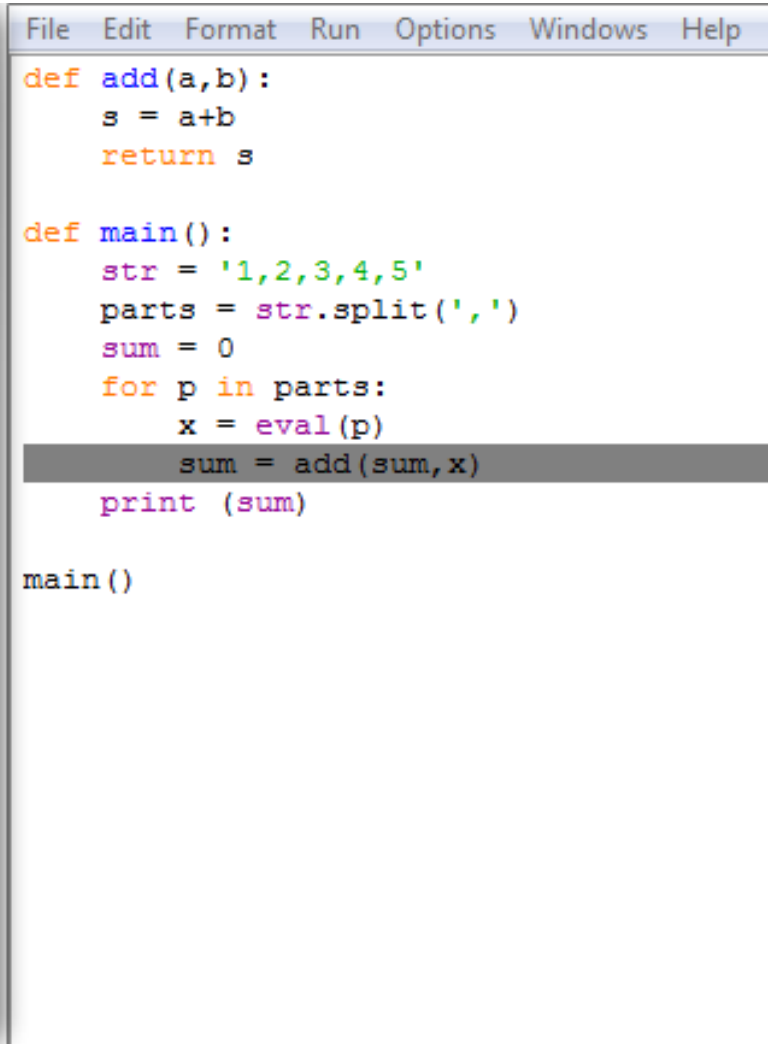




Step over the next line (execute it without going into its details). If the next line is a function call, step over will execute the function without walking through its lines of code.

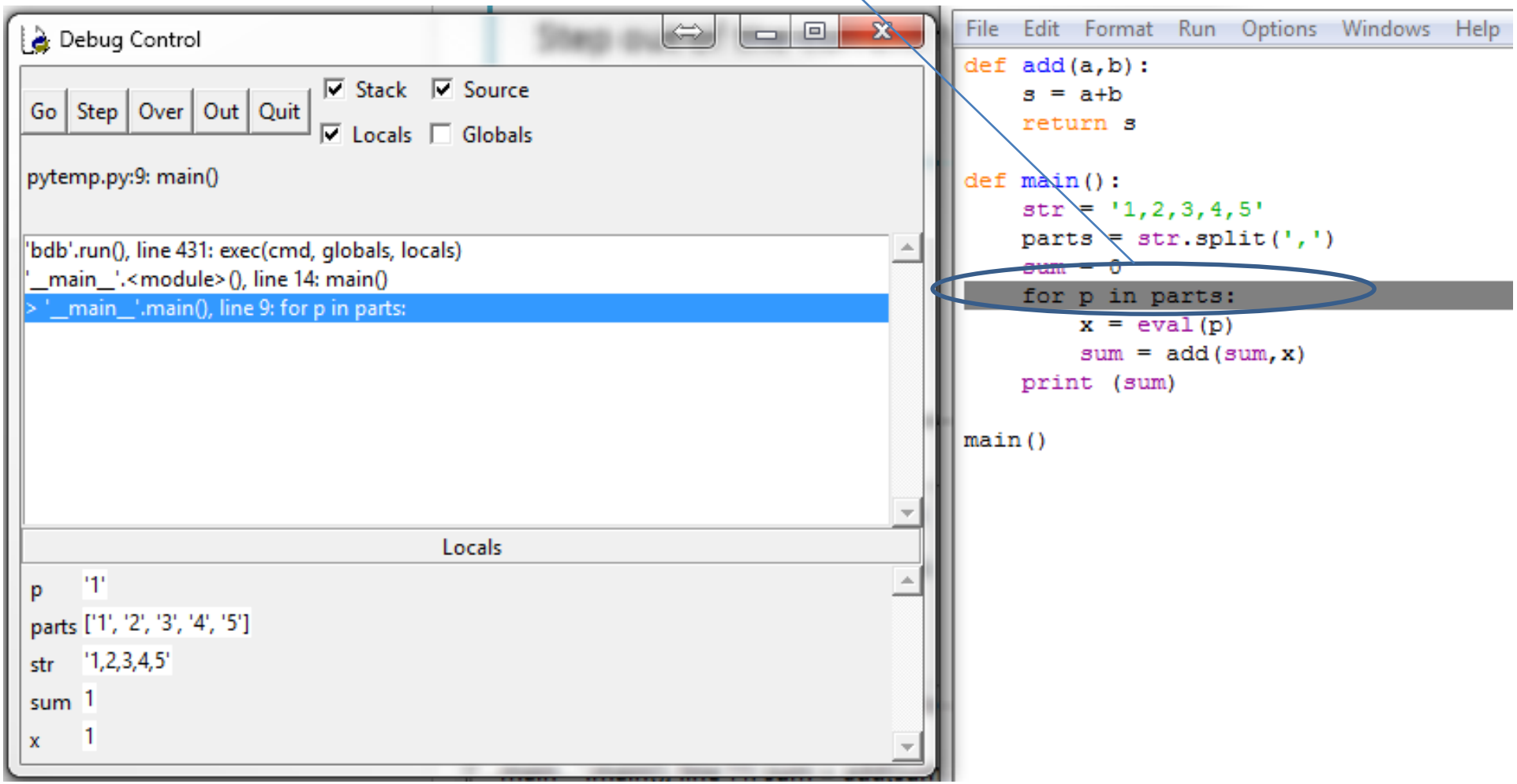
IDLE Debugger





Step over executed the function without walking through its lines, and continue the execution of the program.

IDLE Debugger



ANY QUESTIONS?