Computing (ES 112)

Yogesh K. Meena
Shouvick Mondal

August 2024

Computer Science & Engineering
IIT Gandhinagar

Lecture 7 (contd.)

Iteration and Loops

Indefinite Loops

- While loops are called "indefinite loops" because they keep going until a logical condition becomes False
- The loops we have seen so far are pretty easy to examine to see if they will terminate or if they will be "infinite loops"
- Sometimes it is a little harder to be sure if a loop will terminate (until program is executed)

```
while True:
  n = int(input())
  if n == 0:
    break
```

What will be the output of the program, if you enter ints/floats?

Definite Loops: Iterating over a set of items…

- Quite often we have a set of items in a particular order, and we definitely know the number of the items a priori.
- We can write a loop to run (execute) the loop body once for each of the items in a set using the Python for constructs.
- These loops are called "definite loops" because they execute an exact number of times
- We say that "definite loops iterate through the members of a set"

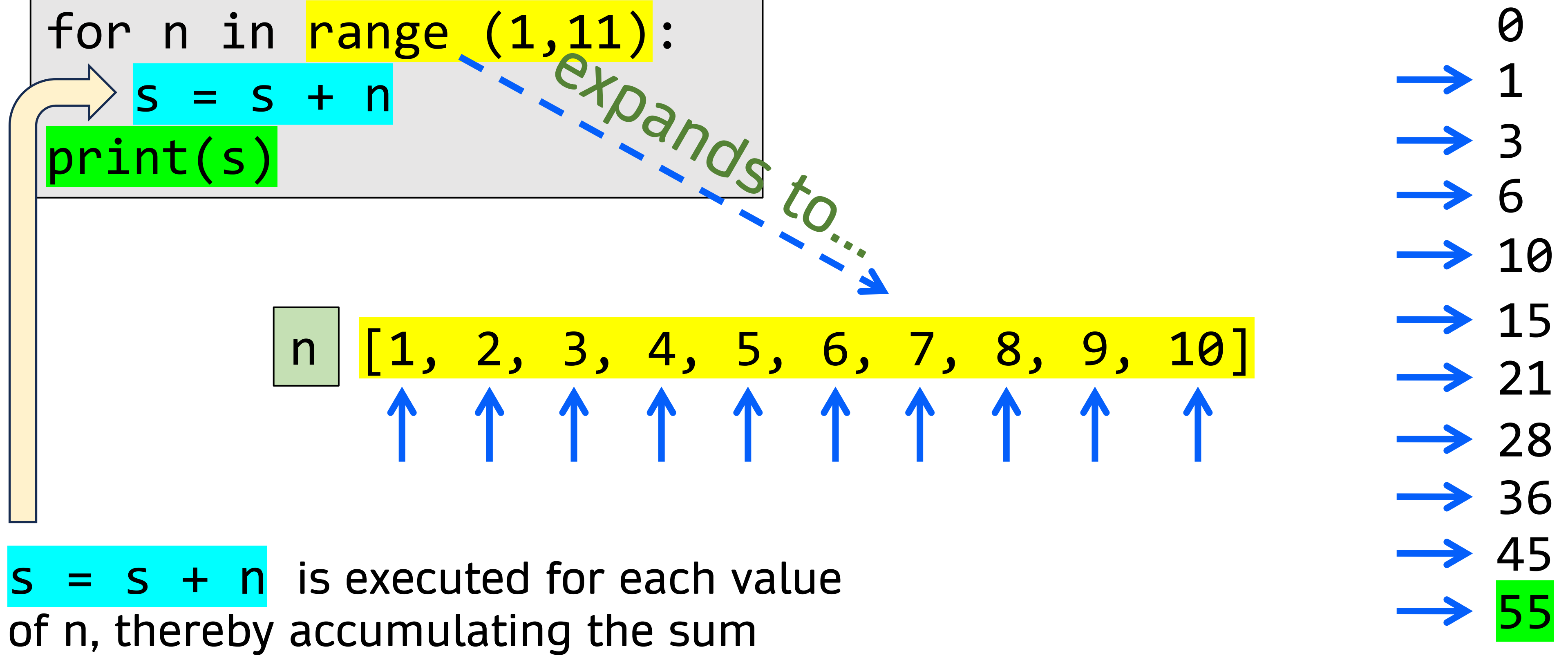
```
n = 1
s = 0
while n <= 10:
    s = s + n
    n = n + 1
print(s)
```

```
s = 0
for n in range (1,11):
   s += n
print(s)
```

Here we don't need to explicitly update the iteration variable n

Definite Loops: Iterating over a sequence of ints

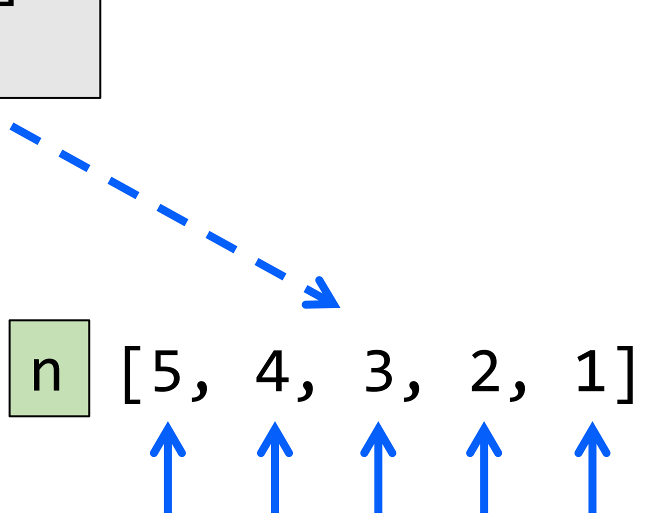
```
s = 0
for n in range (1,11):
    s = s + n
print(s)
```



`s = s + n` is executed for each value of n, thereby accumulating the sum

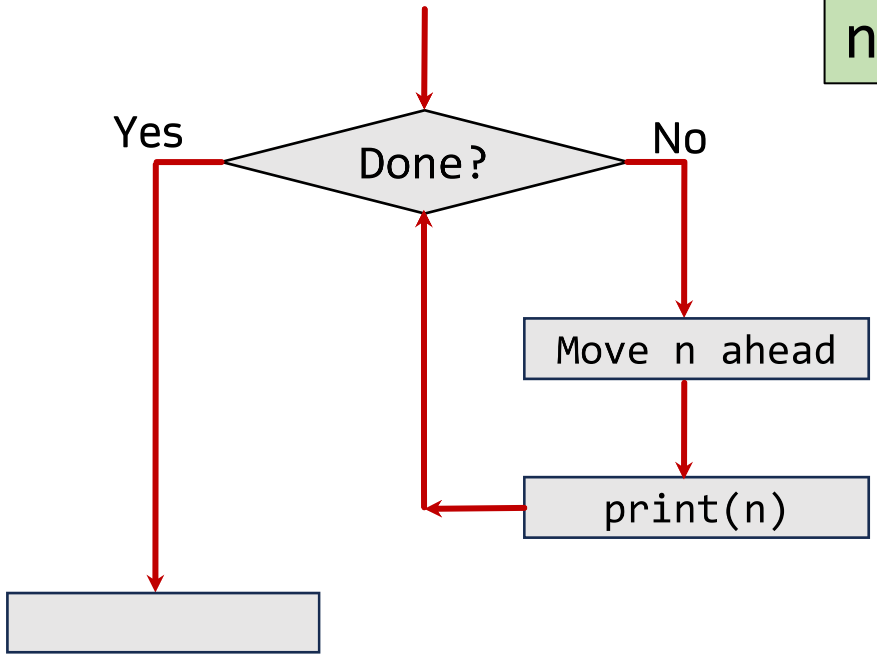
Definite Loops: Iterating over a list of ints

```
for n in [5, 4, 3, 2, 1]:
    print(n)
```



Output

```
5
4
3
2
1
```



Definite loops (for loops) have explicit iteration variables that change each time through a loop. These iteration variables move through the collection of items in order.

Definite Loops: Iterating over characters in strings

```
for i in "IIT GANDHINAGAR":
    print(i)
```

```
I
I
T
 
G
A
N
D
H
I
N
A
G
A
R
```

A string is a sequence of individual characters.

How can we print the characters on the same line with for loop?

```
for i in "IIT GANDHINAGAR":
    print(i)
else:
    print("GUJARAT, INDIA")
```

```
I
I
T
 
G
A
N
D
H
I
N
A
G
A
R
GUJARAT, INDIA
```

"else:" can also be associated with a for loop.

Definite Loops: Iterating over characters in strings

```
for i in "IIT GANDHINAGAR":
    print(i,end="")
```

```
IIT GANDHINAGAR
```

A string is a sequence of individual characters.

```
for i in "IIT GANDHINAGAR":
    print(i)
else:
    print("GUJARAT, INDIA")
```

```
I
I
T
 
G
A
N
D
H
I
N
A
G
A
R
GUJARAT, INDIA
```

"else:" can also be associated with a for loop.

The is and is not Operators

- Python has an **is** operator that can be used in logical expressions
- Implies "**is the same as**"
- Similar to, but stronger than **==**
- **is not** also is a logical operator

```
smallest = None
print('Before')
for value in [3, 41, 12, 9, 74, 15]:
    if smallest is None:
        smallest = value
    elif value < smallest:
        smallest = value
    print(smallest, value)

print('After', smallest)
```

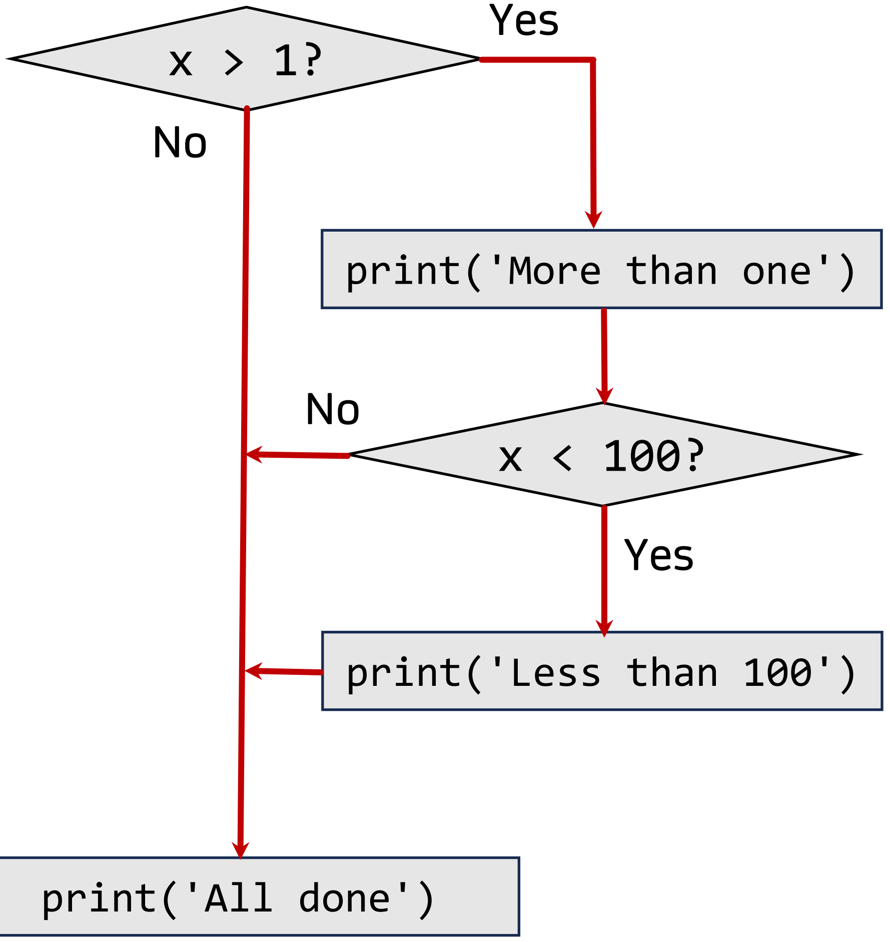
Incremental/iterative approach to find the smallest number. At the end of each iteration, we have the so far seen smallest number. In the end, we find the actual one

Nested ("one within other"): Recap

```
x = 42
if x > 1:
  print('More than one')
  if x < 100:
    print('Less than 100')
print('All done')
```

Here the inner if statement is nested within the outer if statement

Note how the corresponding blocks become nested too because of the if nesting



Nested Loops

Program

```
for i in range(1,6):
    for j in range(1,i+1):
        print(j,end=" ")
    print("")
```

Output

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
for i in range(1,6):
    for j in range(1,i+1):
        print(i,end=" ")
    print("")
```

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

Nested Loops

Program

```
for i in range(1,6):
    for j in range(1,i+1):
        print("",end=" ")
    print(i)
```

Output

```
1
  2
    3
      4
        5
```

Here the for loop's nesting depth is two. Too high nesting can be complex and difficult to manage

```
for i in range(1,6):
    for j in range(1,i+1):
        print("*",end="")
    print(i)
```

```
*1
**2
***3
****4
*****5
```

Programs with loops: Average of numbers

Input

```
1
100
```

Program

```
#Average of numbers a..b (incl.)
a=int(input()) #lower boundary
b=int(input()) #upper boundary

#check what is now stored in a,b
print(a)
print(b)
sum=0.0
#initialize to 0.0 (float)
for i in range (a,b+1):
  sum = sum + i #(b-a+1) times
sum = sum/(b-a+1)
print(sum)
```

Output

```
1
100
50.5
```

Intermediate outputs: 1, 100
Final output: 50.5

print() is used here to check intermediate results (knows as program states). How is this useful for larger programs?

Programs with loops: Largest number in a list

Program

```
#Largest number L in a list A
A = [2,5,-1,-5,1]
L = -99999 #this is necessary
print(L)
#initialize
for i in A:
  if i >= L:
    L = i #change largest
    print(i,L) #so far largest

print(L) #final largest
```

Output

```
-99999
2  2
15  15
-1  15
-5  15
1  15
15
```

L changed

L changed

How will you find the second largest number? There are many ways to do it...

`if i >= L`: 5 times
`L = i`: 2 times
`print(i,L)`: 5 times

What will happen if L=99999 initially and we change >= to <=?

Programs with loops: 2nd Largest number in a list

Program

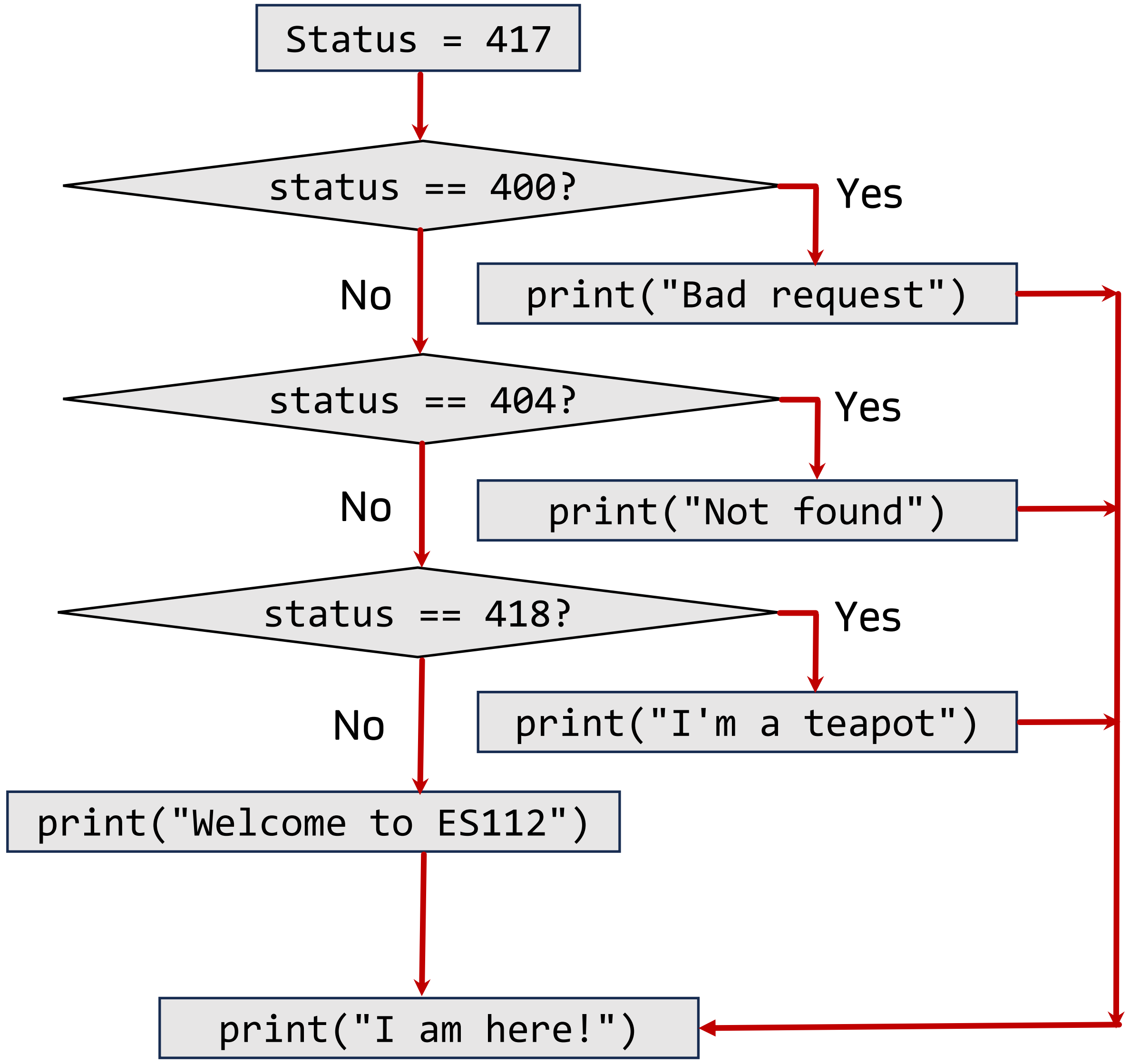
```python
#Second Largest number L2 in a list A
A = [831,88366666,-1,-5,4666,1778]
L1 = -9999999 #this is necessary
L2 = -9999999 #this is also necessary
for i in A:
   print("before: ",i,L1,L2)
   if i >= L1:
      L2=L1
      L1=i
   if i >= L2 and i < L1:
      L2=i
   print("after: ",i,L1,L2)
print("Second largest: "+str(L2))
```

Output

```
before: 831 -9999999 -9999999
after: 831 831 -9999999
before: 88366666 831 -9999999
after: 88366666 88366666 831
before: -1 88366666 831
after: -1 88366666 831
before: -5 88366666 831
after: -5 88366666 831
before: 4666 88366666 831
after: 4666 88366666 4666
before: 1778 88366666 4666
after: 1778 88366666 4666
Second largest: 4666
```

Decision Making: match-case (recap)

```python
status = 417
match status:
   case 400:
      print("Bad request")
   case 404:
      print("Not found")
   case 418:
      print("I'm a teapot")
   case _:
      print("Welcome to ES112")
print("I am here!")
```



The **underscore** '_' is a wildcard.
If nothing else matches, this is the **last resort (default)** action to be performed

Decision Making: match-case (with |)

```
status = 417
match status:
  case 400 | 404 | 418:
    print("Bad request")
    print("Not found")
    print("I'm a teapot")
  case _:
    print("Welcome to ES112")
print("I am here!")
```

When used in the `case` clause, the pipe operator '|' (bitwise OR) does not actually perform the bitwise operation, but denotes a mutually exclusive merger of multiple cases

What will happen if we replace '|' (bitwise OR) with logical OR 'or'?

```
File "/home/runner/TestES112/main.py", line 4
  case 400 or 404 or 418:
           ^^
SyntaxError: invalid syntax
exit status 1
```

Decision Making: `match-case` (with ranges) #1

```
#for numbers 1..50, print "Hello"
#for numbers 51..100, print "World"
for i in range(1,101):
   match(i):
      case i if i <=50:
         print("Hello")
      case _:
         print("World")
```

The usage of '`if`' in this case saves space of writing 49 more cases. This increases the readability of code

Can we rewrite the above code without the `if` condition and still get the same output?

Decision Making: match-case (with ranges) #2

```
#for numbers 1..50, print "Hello"
#for numbers 51..100, print "World"
for i in range(1,101):
   match(i <= 50):
      case True:
         print("Hello")
      case False:
         print("World")
```

The usage of Boolean "True" makes this match-case a two-way decision switch

Acknowledgements / Contributions

Initial Development: Charles Severance, University of Michigan School of Information

Contributors 2024 - Yogesh K. Meena and Shouvick Mondal, IIT Gandhinagar