SI100B Introduction to Information Science and Technology
Python Programming

张海鹏 Haipeng Zhang
School of Information Science and Technology
ShanghaiTech University

Tutorials

- Python programming tutorials
  - 8:00 pm, Friday evenings in TC201
  - Provides additional knowledge as well as information about the assignments
  - Will be video-recorded, but we encourage you to attend

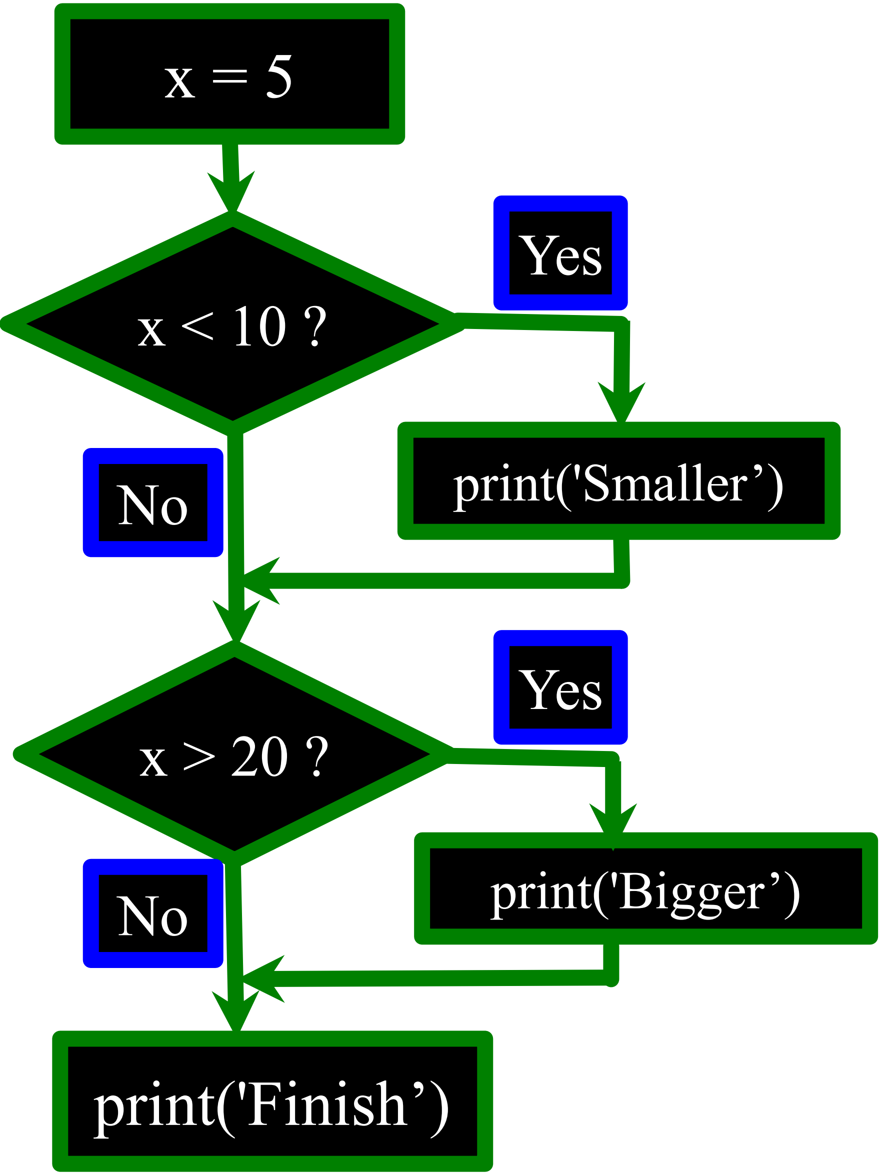
Assignment 1

- Announced on Piazza, Sept 15 evening
- Due 23:59:59 Friday, Oct 2
- Involves control flow, string operations, basic data structures, etc.

Learning Objectives

- Condition
- Python Program Structure

Conditional execution



x = 5

x < 10 ?

Yes

No

print('Smaller’)

x > 20 ?

Yes

No

print('Bigger’)

print('Finish’)

Program:

```
x = 5
if x < 10:
    print('Smaller’)

if x > 20:
    print('Bigger’)

print('Finish’)
```

Output:

```
Smaller
Finish
```

If statement

- Syntax:

```
if expression:
          statement 1
          statement 2
          …
          statement N
outside statement
…
```

if statement

indention

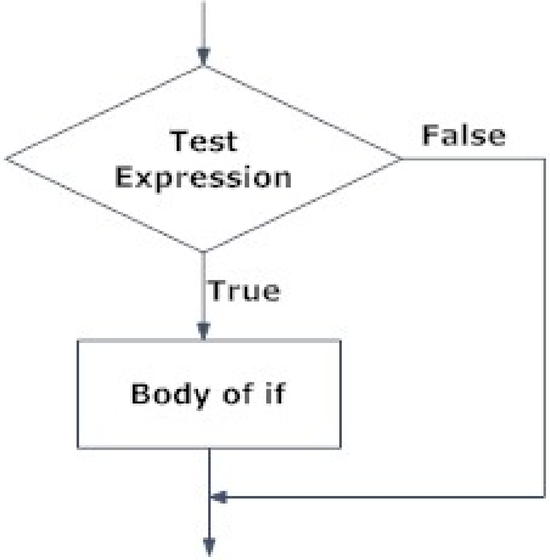


Fig: Operation of if statement

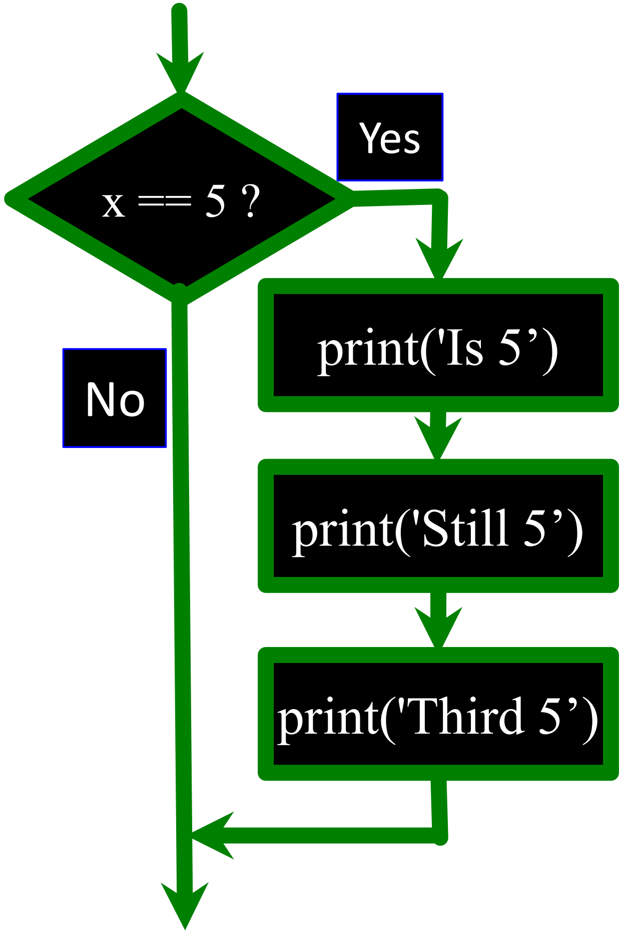
- If the expression is true, statements within the if statement body will be executed, otherwise the entire "if statement" will be ignored

```
>>> x = 5
>>> y = 7
>>> if x+y > 10:
...    print("sum > 10")
sum > 10
```

If statement

```
x = 5
print('Before 5’)
if  x == 5 :
    print('Is 5’)
    print('Is Still 5’)
    print('Third 5’)
print('Afterwards 5’)
print('Before 6’)
if x == 6 :
    print('Is 6’)
    print('Is Still 6’)
    print('Third 6’)
print('Afterwards 6’)
```

```
Before 5
Is 5
Is Still 5
Third 5
Afterwards 5
Before 6
Afterwards 6
```

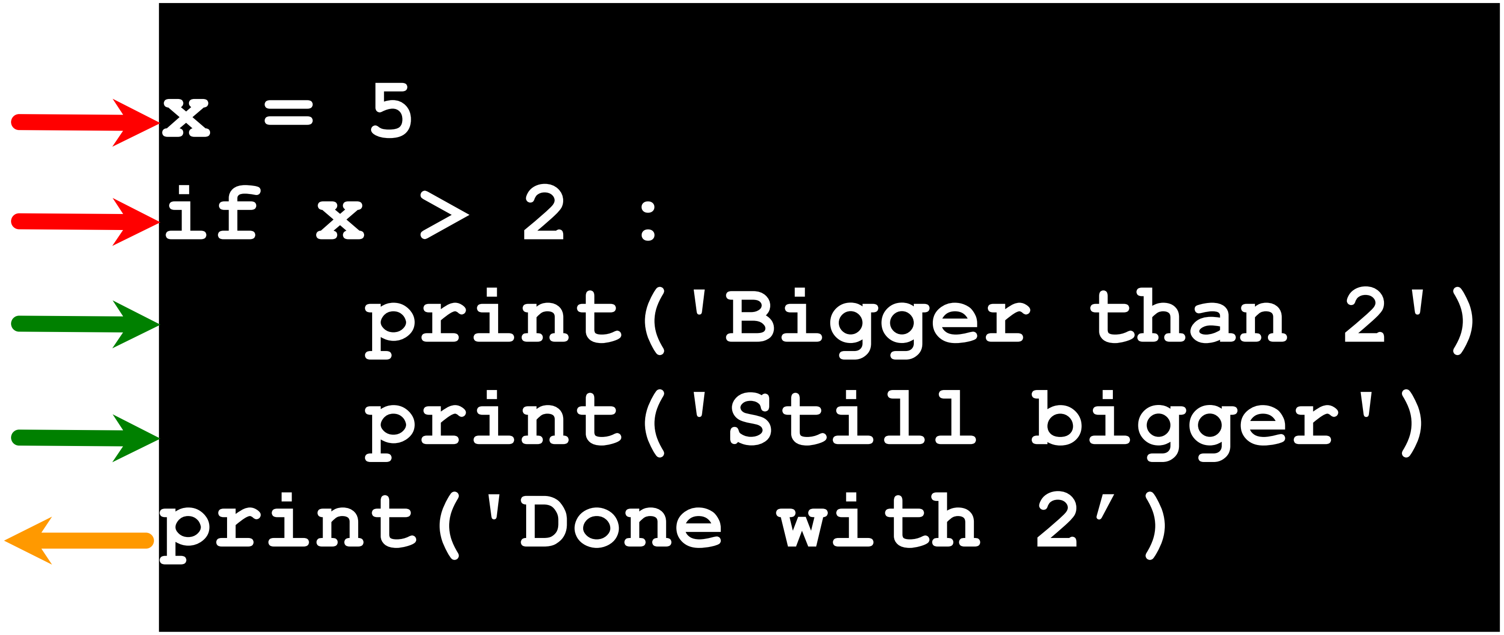


Indention

- Spaces or tabs
- Increase indent indent after an if statement (after :)
- Maintain indent to indicate the scope of the block (which lines are affected by the if)
- Reduce indent *back to* the level of the if statement to indicate the end of the block
- Blank lines are ignored - they do not affect indention

Indention

- Increase / maintain after if
- Decrease to indicate end of block



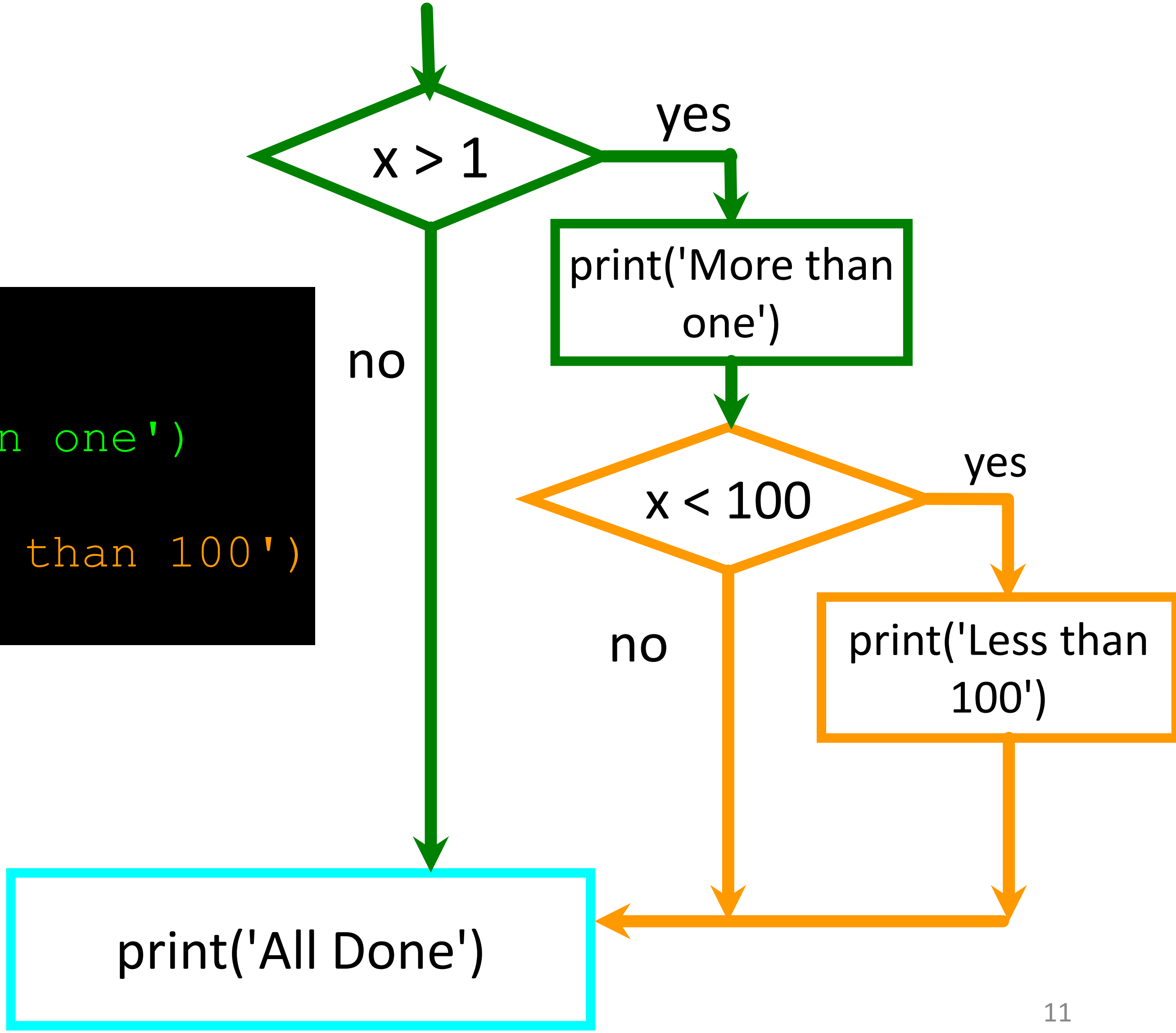
```
x = 5
if x > 2 :
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2’)
```

If statement and indention

```
x = 5
if x > 2 :
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')
```

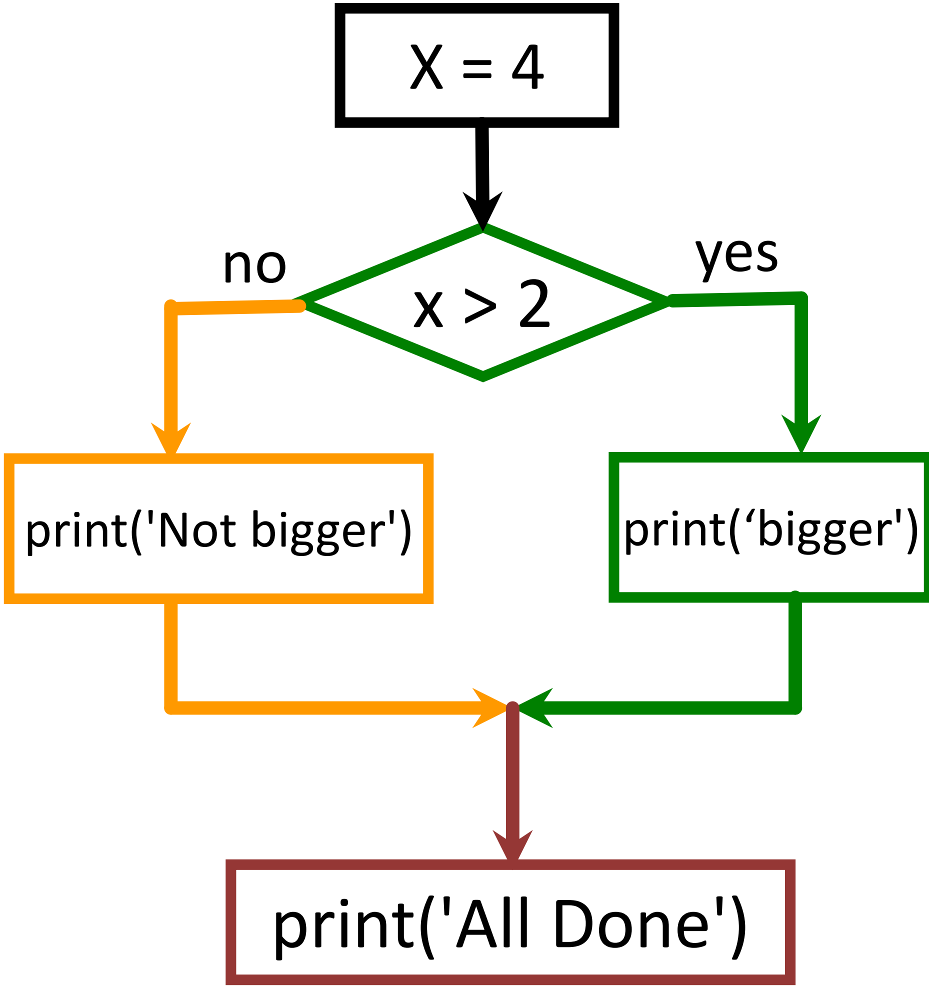
Nested if-statements

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



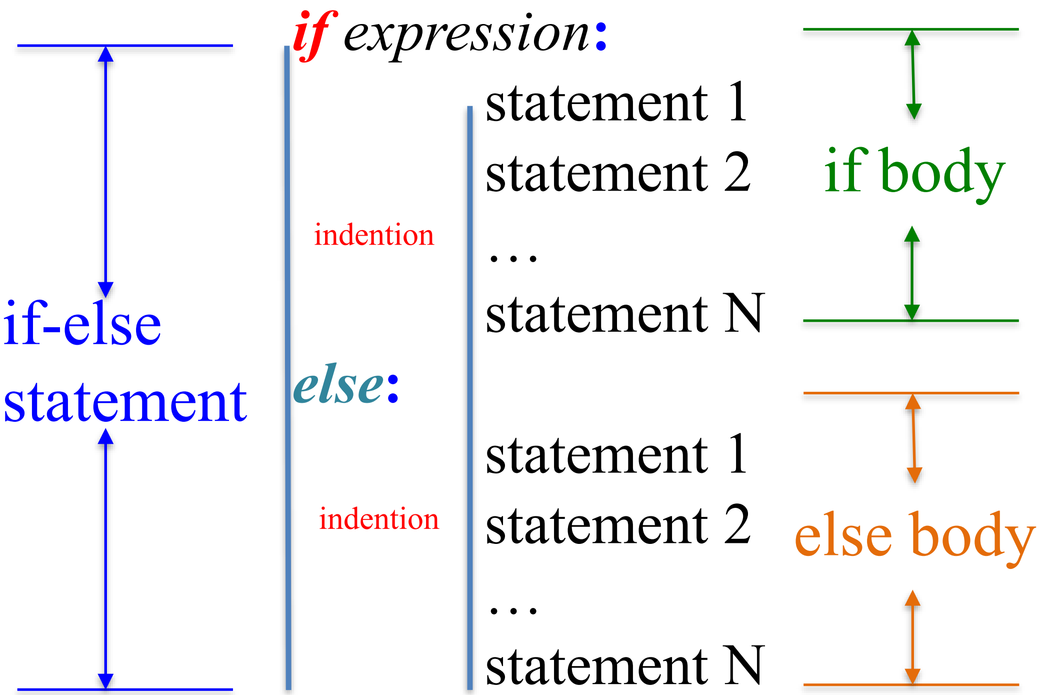
if-else statement

- Sometimes we want to do one thing if a logical expression is true and something else if the expression is false.
- It is like a fork in the road – we must choose one or the other path but not both



if-else statement

- Syntax:



```
if expression:
        statement 1
        statement 2
        …
        statement N
else:
        statement 1
        statement 2
        …
        statement N
```

if-else statement

indention

if body

indention

else body

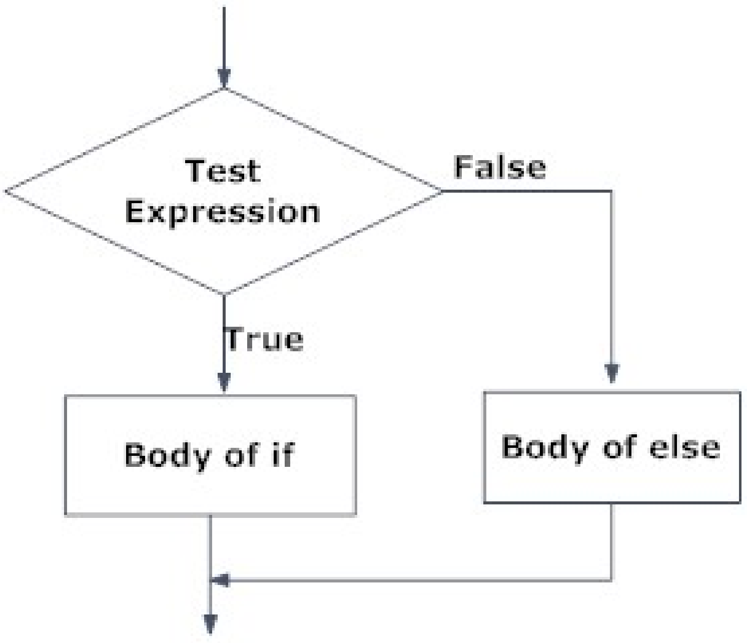
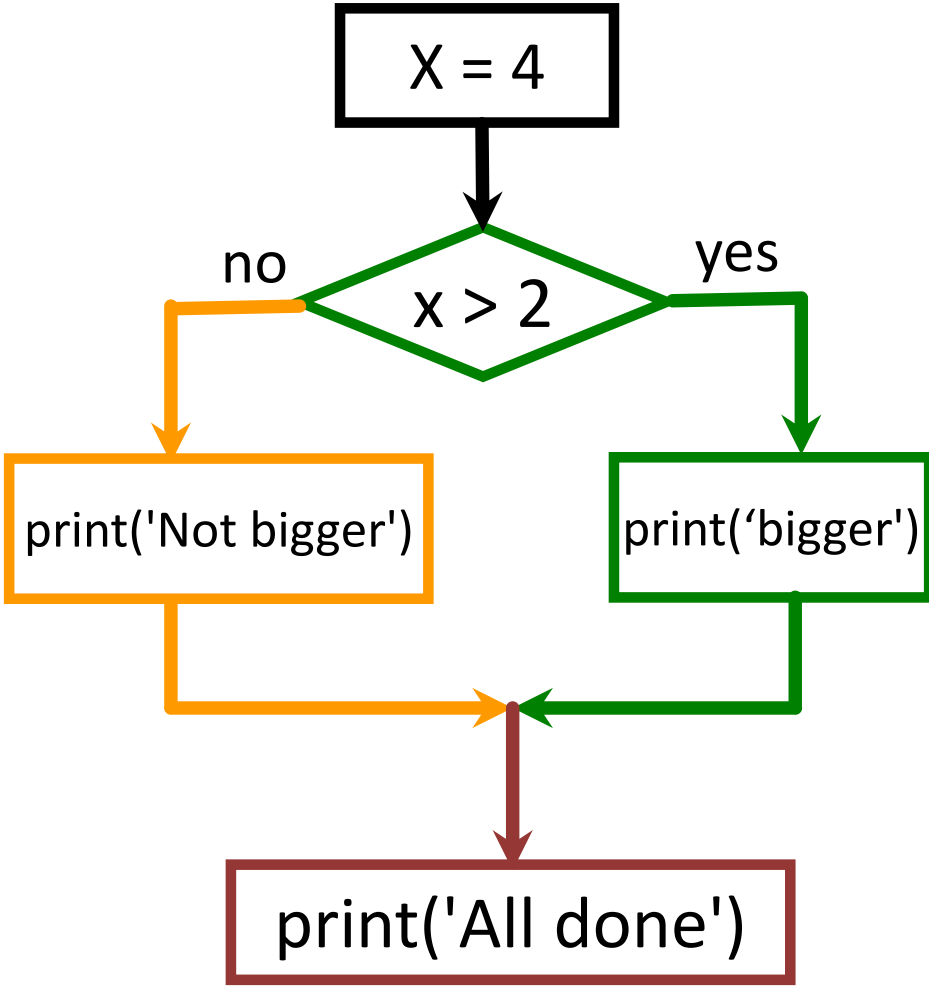


Fig: Operation of if...else statement

- If the expression is true, statements within the if statement body will be executed, otherwise statements within else body will be executed.

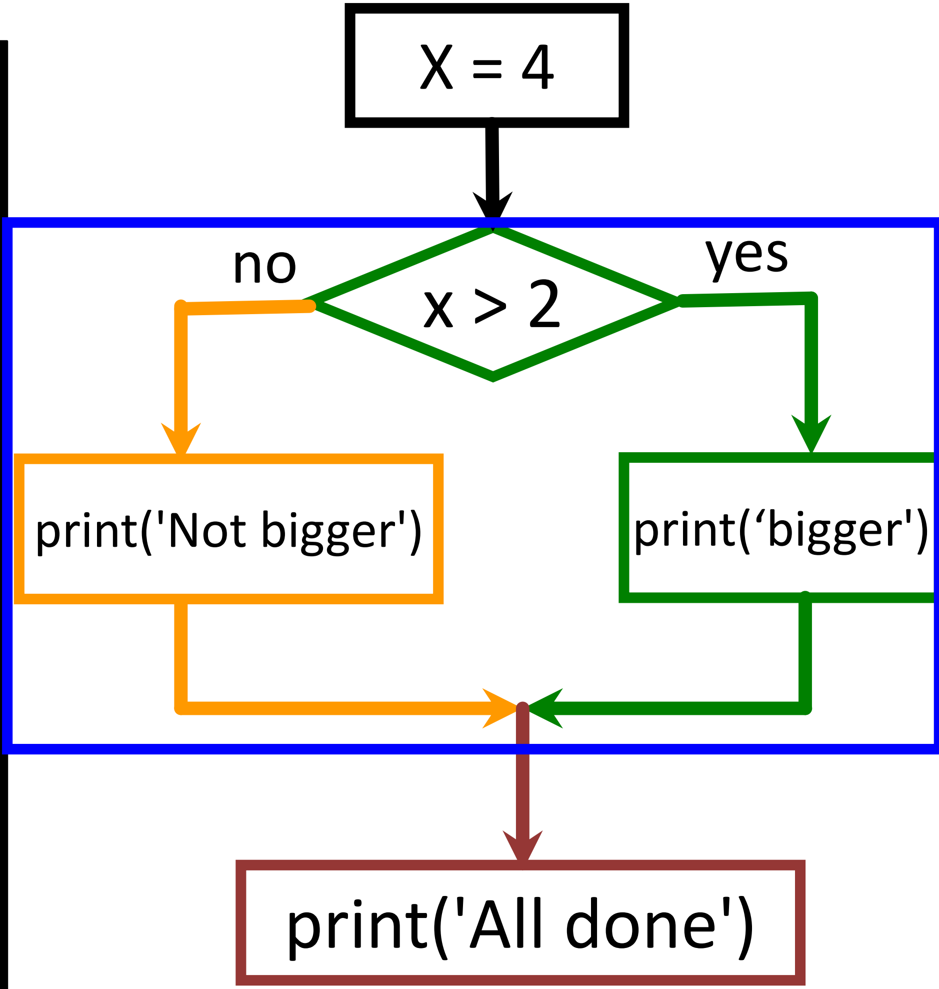
if-else statement

```
x = 4

if x > 2 :
    print('Bigger')
else :
    print('Smaller')

print('All done')
```

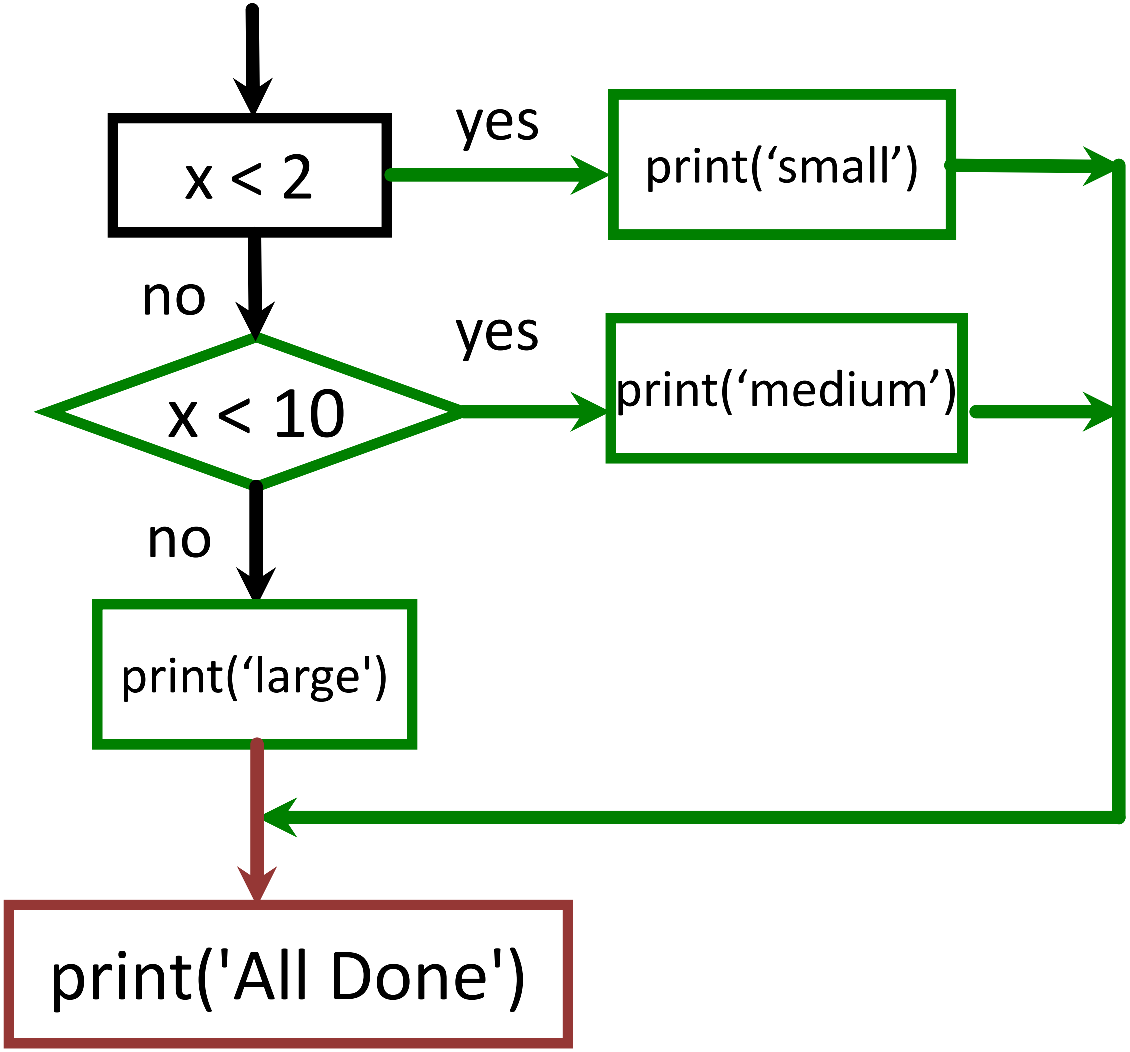


if-else statement

```
x = 4

if x > 2 :
    print('Bigger')
else :
    print('Smaller')

print('All done')
```



Multi-way


x < 2

yes

print('small')

no

x < 10

yes

print('medium')

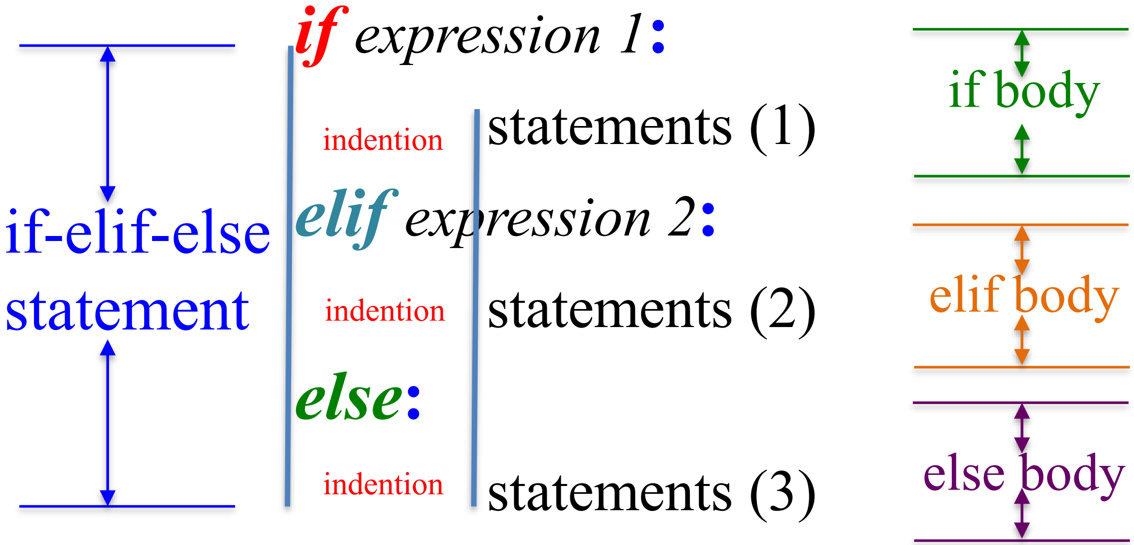
no

print('large')

print('All Done')

if-elif-else statement

- Syntax:



if-elif-else statement

```
if expression 1:
    statements (1)
elif expression 2:
    statements (2)
else:
    statements (3)
```

indention

indention

indention

if body

elif body

else body

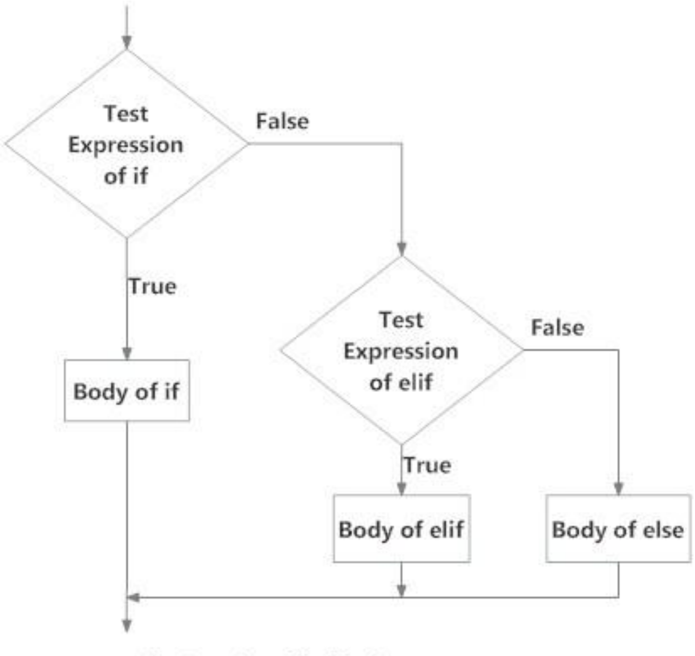


Fig: Operation of if...elif...else statement

- If the expression 1 is true, statements (1) within the if statement body will be executed, otherwise if expression 2 is true, statements (2) within elif body will be executed, otherwise, statements (3) within else body will be executed.

if-elif-else statement

```
x = 5
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
else:
    print(Large')
print('All done')
```

Variations (1)

- Syntax:

  if expression 1:

  statements (1)

  elif expression 2:

  statements (2)

  statements after if-elif

```
# No Else
x = 5
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')

print('All done')
```

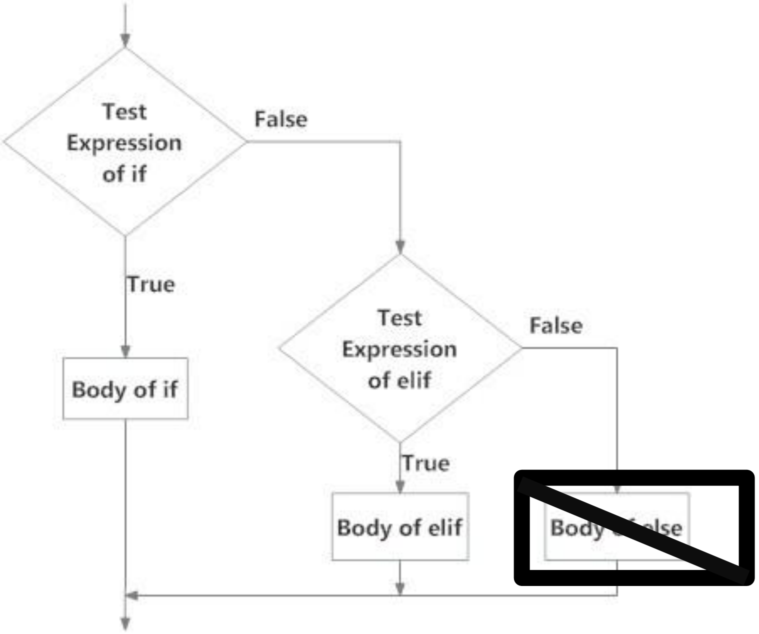


Fig: Operation of if...elif...else statement

Variations (2)

- Syntax:

```
if expression 1:
        statements (1)
elif expression 2:
        statements (2)
elif expression 3:
        statements (3)
…
else:
        statements (N)
```

```
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
elif x < 20 :
    print('Big')
elif x < 40 :
    print('Large')
elif x < 100:
    print('Huge')
else :
    print('Ginormous')
```

Puzzles

- Which will never print?

```
if x < 2 :
    print('Below 2')
elif x >= 2 :
    print('Two or more')
else :
    print('Something
else')
```

```
if x < 2 :
    print('Below 2')
elif x < 20 :
    print('Below 20')
elif x < 10 :
    print('Below 10')
else :
    print('Something
else')
```

Nested cases

- All statements in the body of ***if***, ***elif***, and ***else*** can also be conditional statements.

```
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
elif x < 20 :
    print('Big')
elif x < 100:
    print('Huge')
else :
    print('Ginormous')
```

```
if x < 100 :
   if x < 20 :
     if x < 10 :
        if x < 2:
          print('Small')
        else:
          print('Medium’)
     else:
        print('Big')
   else:
     print('Huge')
else :
    print('Ginormous')
```

Nested cases

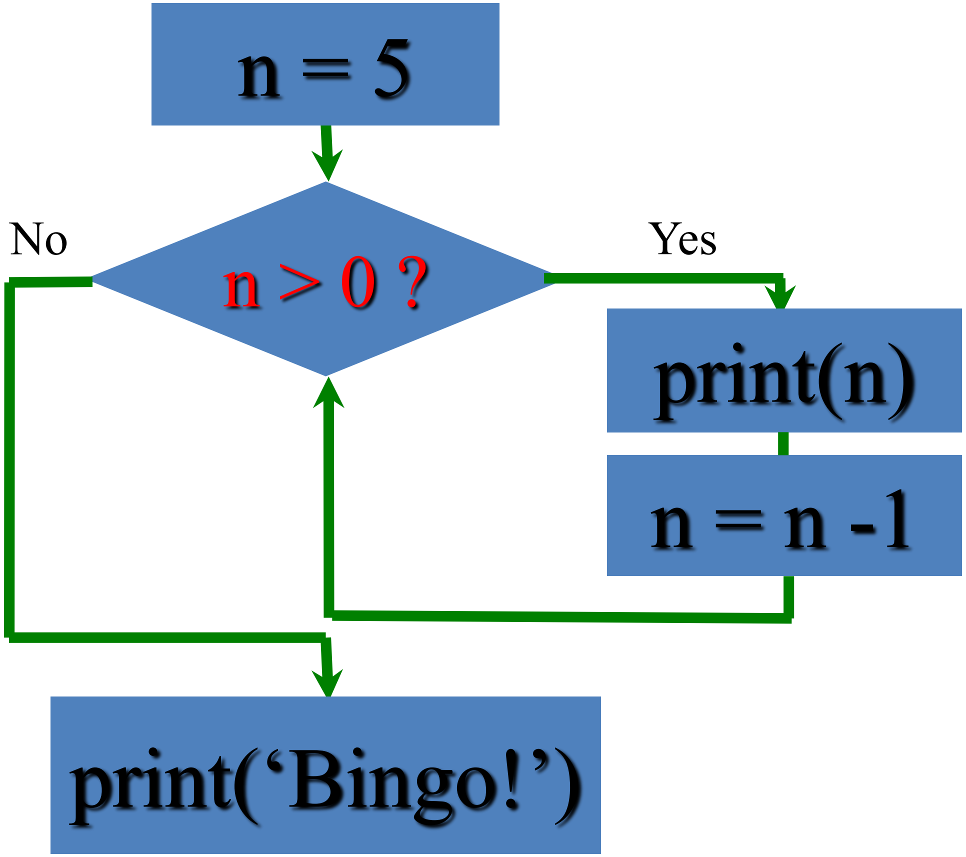
- All statements in the body of ***if***, ***elif***, and ***else*** can also be conditional statements.

```
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
elif x < 20 :
    print('Big')
elif x < 100:
    print('Huge')
else :
    print('Ginormous')
```

```
if x < 2 :
    print('Small')
else:
   if x < 10 :
      print('Medium')
   elif x < 20 :
      print('Big')
   elif x < 100:
      print('Huge')
   else :
      print('Ginormous')
```

Repeated steps

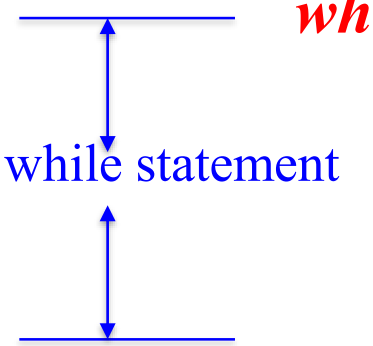
- Computers are often used to automate repetitive tasks (**loop**)



```
Output:

5
4
3
2
1
Bingo!
```

The *while* statement

- Syntax:



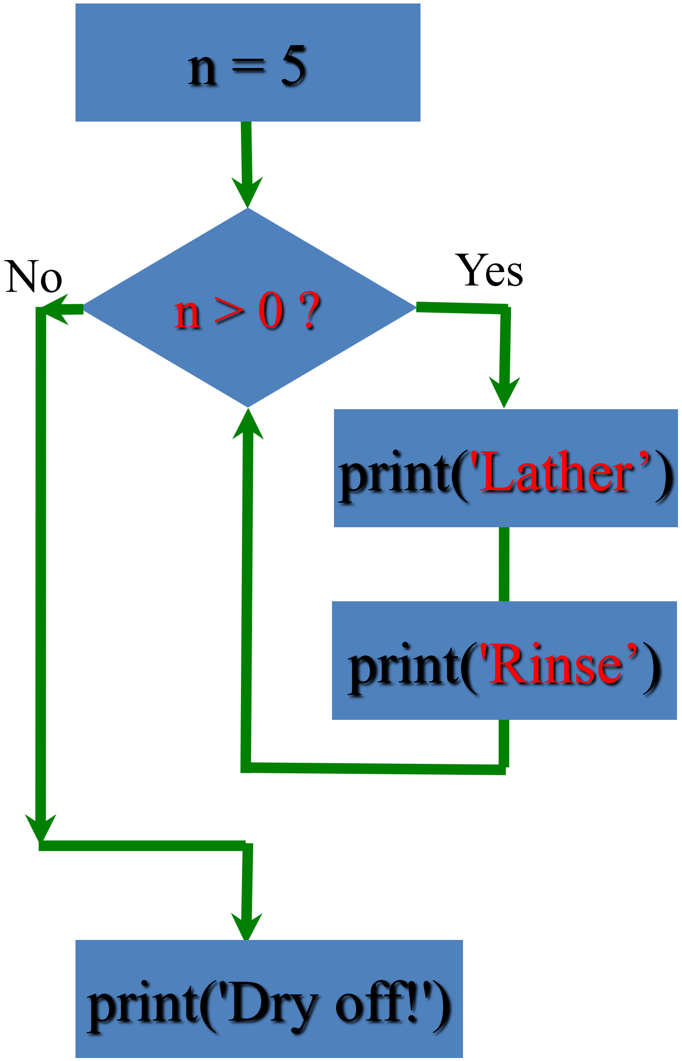
```
while expression:
        statement 1
        statement 2
        …
        statement N
next statement
```

- The flow of execution
  a) Evaluate the expression, yielding True or False
  b) If the expression is False, exit the entire while statement and continue execution at the next statement
  c) If the expression is True, execute each of the statements in the body and then go back to step (a)

Example

```
test1.py

x = 5
while x > 3:
   print(x)
   x = x - 1
print(x+1)

python test1.py
5
4
4
```

An infinite loop



n = 5

No

n > 0 ?

Yes

print('Lather’)

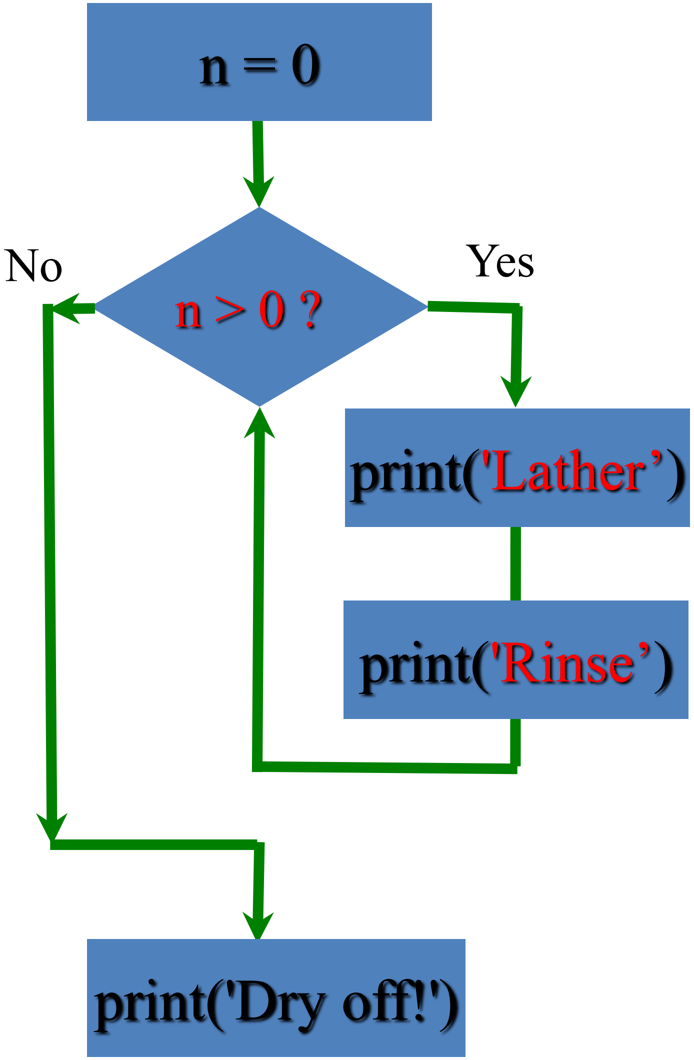
print('Rinse’)

print('Dry off!')

```
n = 5
while n > 0 :
    print('Lather’)
    print('Rinse')
print('Dry off!')
```

What is wrong with this loop?

Another loop



```
n = 0
while n > 0 :
    print('Lather’)
    print('Rinse')
print('Dry off!')
```

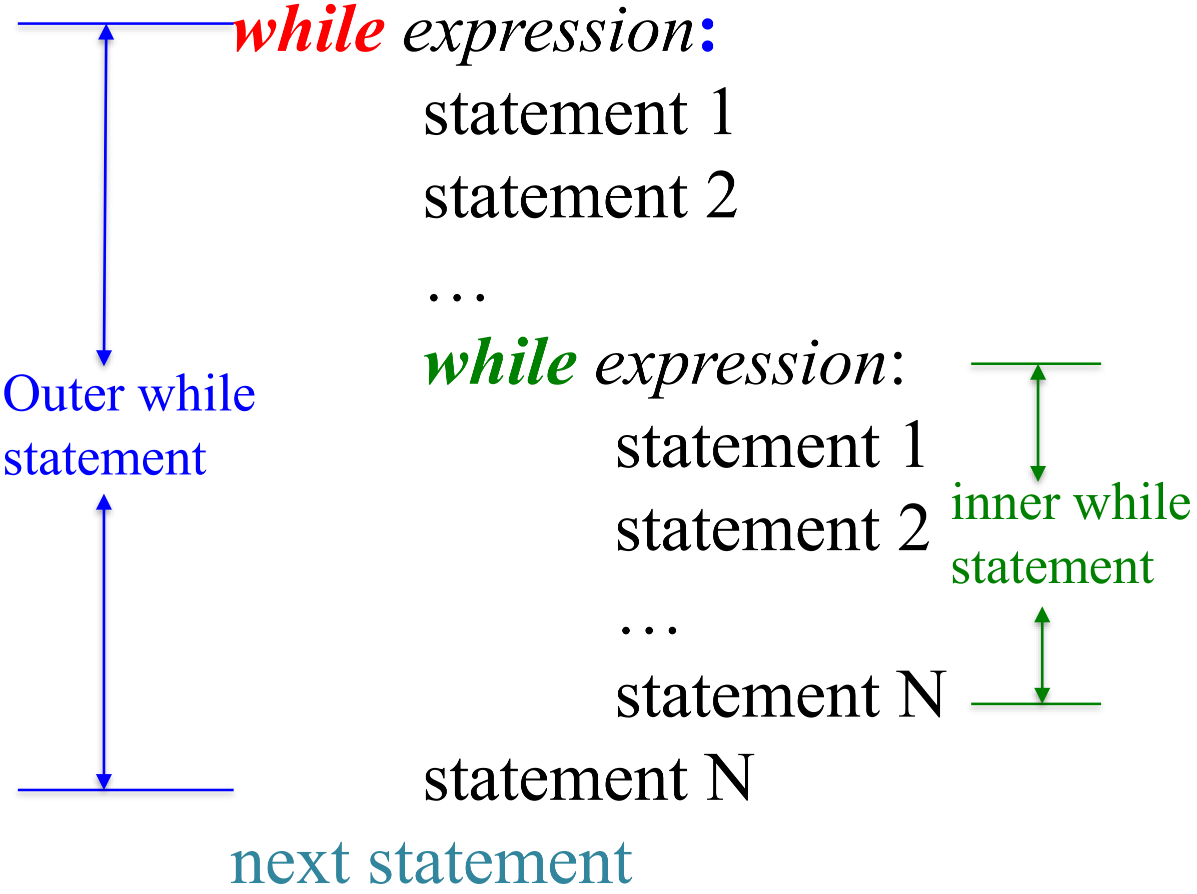
What does this loop do?

Example

```
test2.py

x = 5
while x != 1:
   print(x)
   if x%2 == 0:
      x = x / 2
   else:
      x = x*3 + 1

python test2.py
5
16
8
4
2
```

The nested *while* statement

- Syntax:



```
while expression:
          statement 1
          statement 2
          …
          while expression:
                    statement 1
                    statement 2
                    …
                    statement N
          statement N
next statement
```

Outer while statement

inner while statement

Example

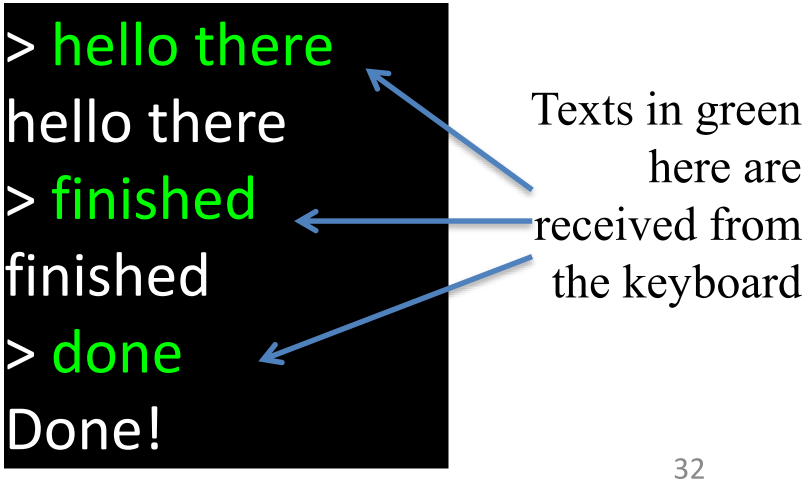
```
test3.py

x = 5
while x != 1:
    print(x)
    while x > 3:
        print(‘x>3’)
        x = x - 1
    x = x - 1

python test3.py
```

```
Output:
5
x>3
x>3
2
```

Breaking out of a loop

- The break statement ends the **current innermost** loop and jumps to the statement immediately following the loop.
- It can happen anywhere in the body of the loop, depending on your needs.

```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line)
print('Done!')
```

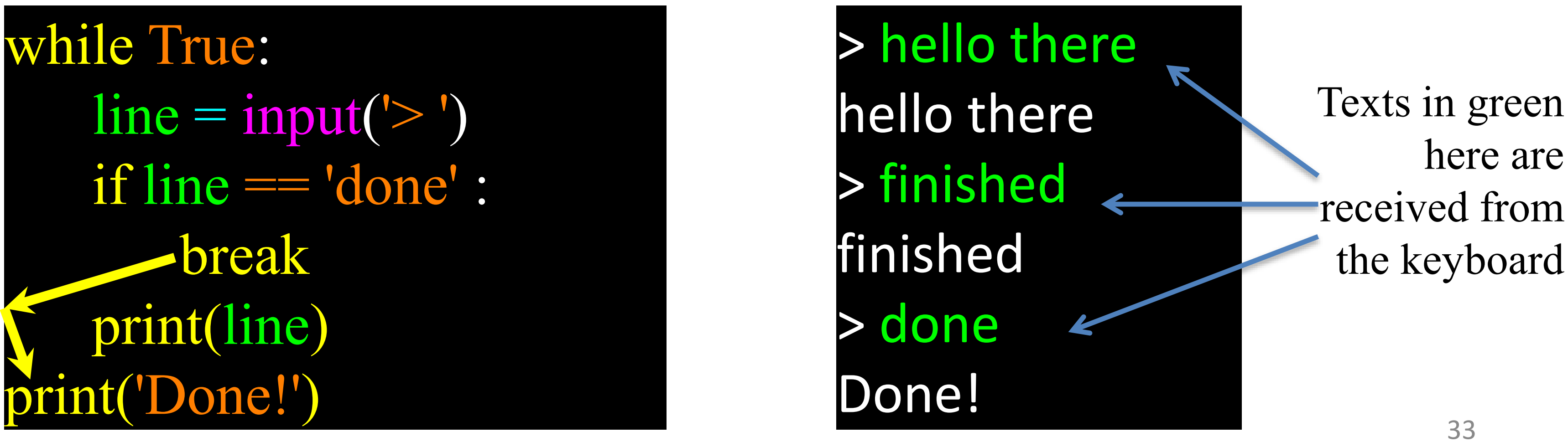


```
> hello there
hello there
> finished
finished
> done
Done!
```

Texts in green here are received from the keyboard

Breaking out of a loop

- The break statement **ends the current innermost** loop and jumps to the statement immediately following the loop.
- It can happen anywhere in the body of the loop, depending on your needs.



```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> finished
finished
> done
Done!
```

Texts in green here are received from the keyboard

Breaking out of a loop

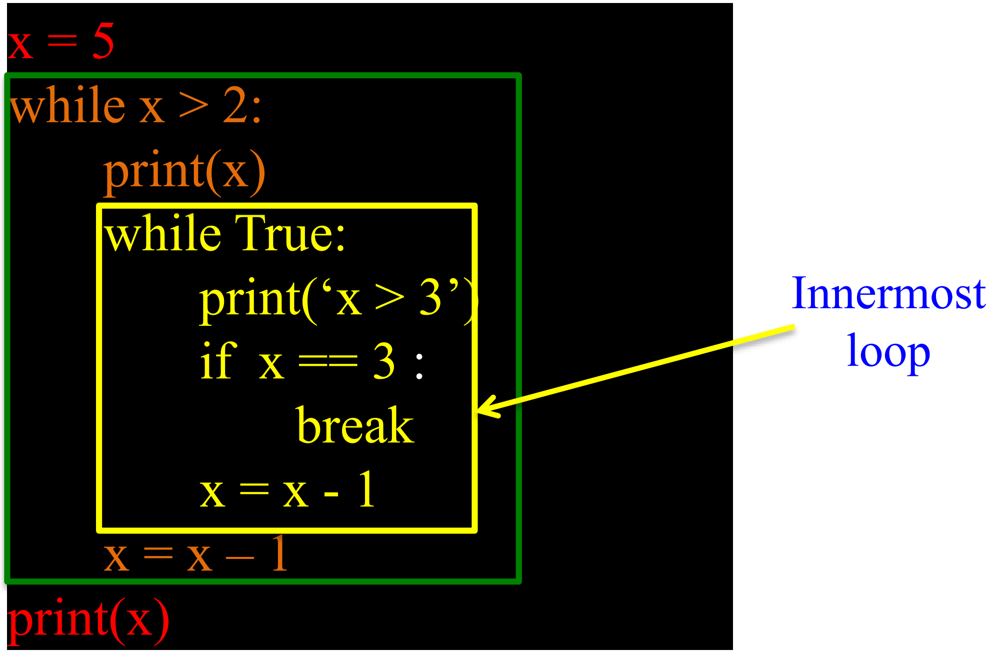
- All statements in the loop body and after break will NOT be executed if break happens.

```
x = 5
while x > 0:
    print(x)
    if x == 3:
        break
    x = x – 1
print(x)
```

```
Output:

5
4
3
3
```

Breaking out of a loop

- The break statement ends the current loop and jumps to the statement immediately following the loop.
- All statements in the loop body and after break will NOT be executed if break happens.



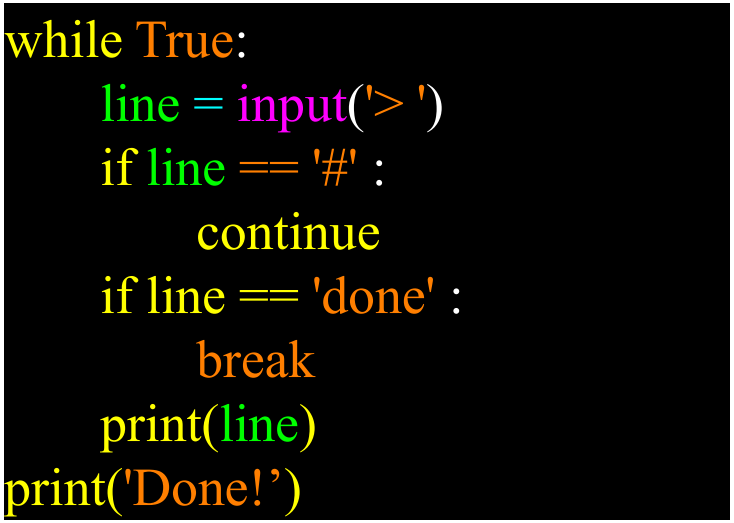
```
x = 5
while x > 2:
      print(x)
      while True:
            print(‘x > 3’)
            if  x == 3 :
                  break
            x = x - 1
      x = x – 1
print(x)
```

Innermost loop

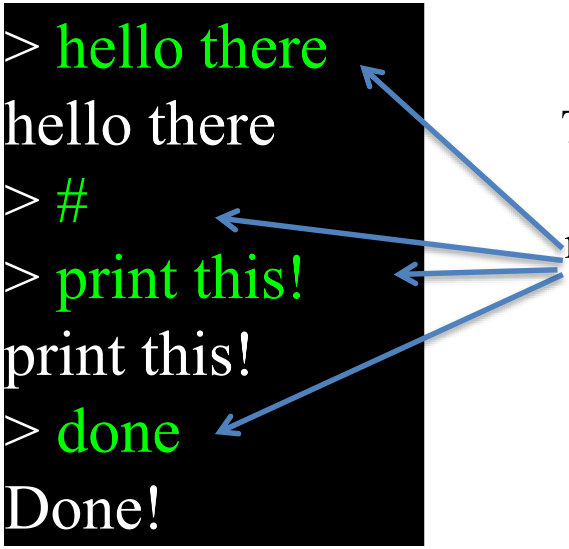
```
Output:

5
x>3
x>3
x>3
2
```

The *continue* statement

- The continue statement ends the current iteration of the innermost loop and jumps to the top of the loop and starts the next iteration.
- It can happen anywhere in the body of the loop, depending on your needs.



```
while True:
    line = input('> ')
    if line == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!’)
```

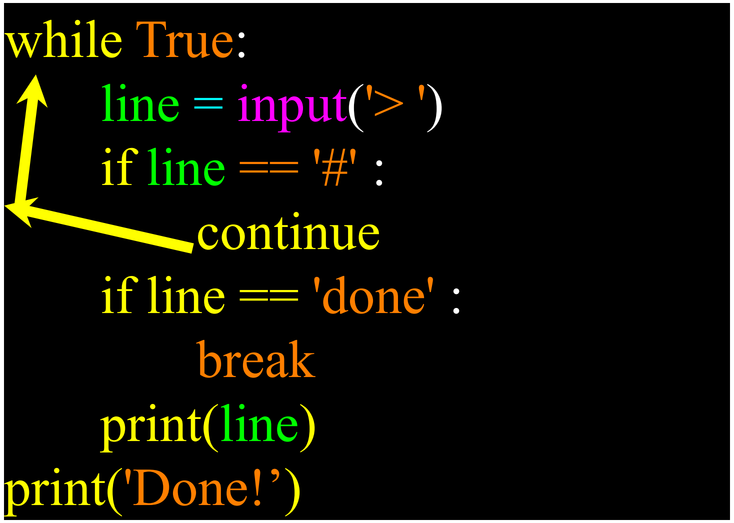


```
> hello there
hello there
> #
> print this!
print this!
> done
Done!
```

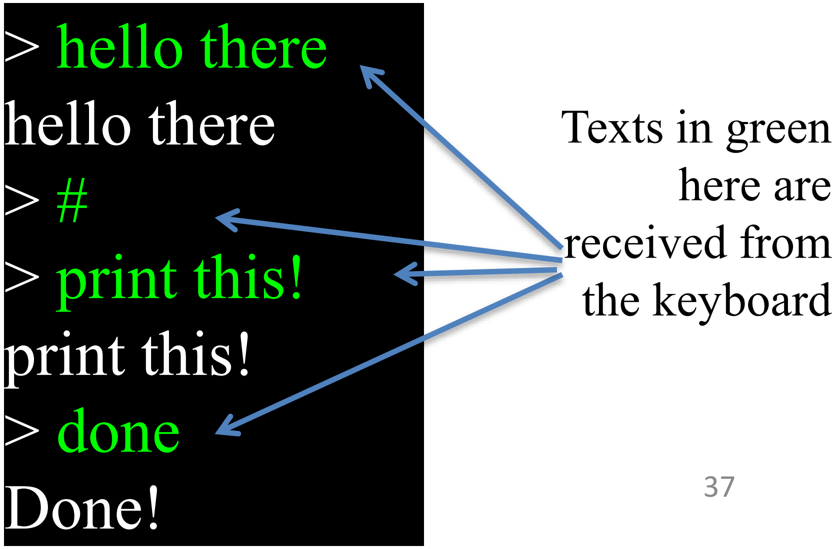
Texts in green here are received from the keyboard

The *continue* statement

- The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration.
- It can happen anywhere in the body of the loop, depending on your needs.



```
while True:
    line = input('> ')
    if line == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!’)
```



```
> hello there
hello there
> #
> print this!
print this!
> done
Done!
```

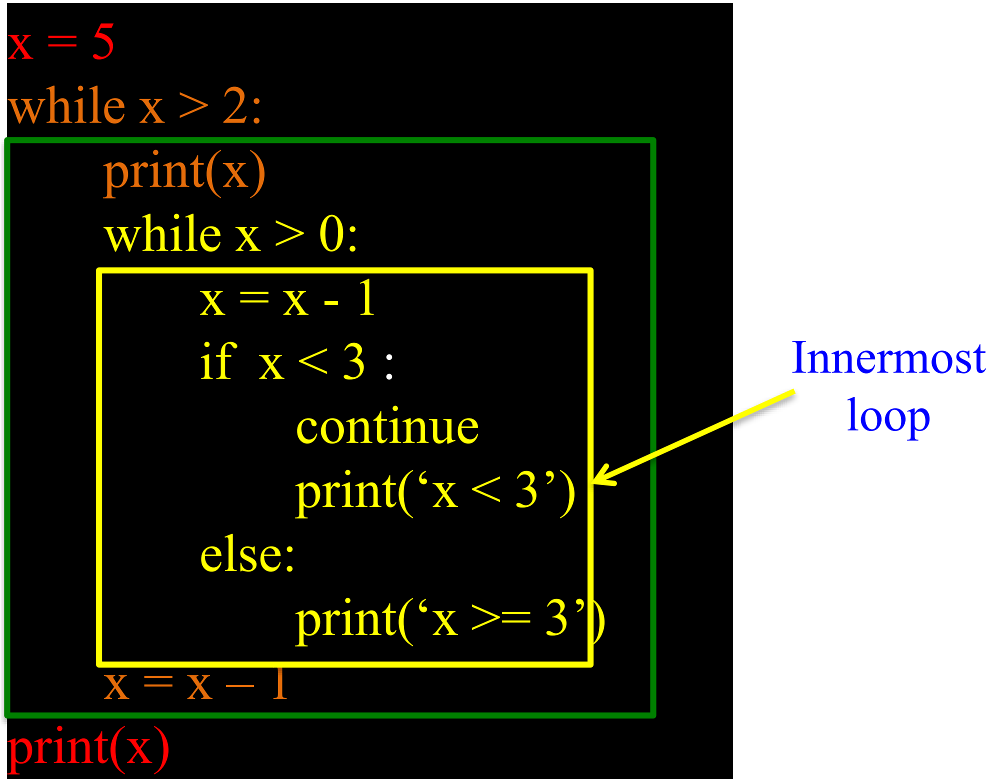
Texts in green here are received from the keyboard

Example

```
x = 5
while x > 0:
    x = x – 1
    if x == 3:
        continue
    print(x)
print(x)
```

```
Output:

4
2
1
0
0
```

Example



```
x = 5
while x > 2:
        print(x)
        while x > 0:
                x = x - 1
                if  x < 3 :
                        continue
                        print(‘x < 3’)
                else:
                        print(‘x >= 3’)
        x = x – 1
print(x)
```

Innermost loop

```
Output:

5
x>=3
x>=3
-1
```

Indefinite loop

- While loops are called “indefinite loops” because they keep going until a logical expression becomes False
- The loops we have seen so far are easy to examine to see if they will terminate or if they are “infinite loops”
- Sometimes it is harder to be sure if a loop will terminate

Definite loop

- Quite often we have a list of items – effectively a finite set of things
- We can write a loop to run the loop once for each of the items in a set using the Python for construct
- 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"

The *for* statement

- Syntax:

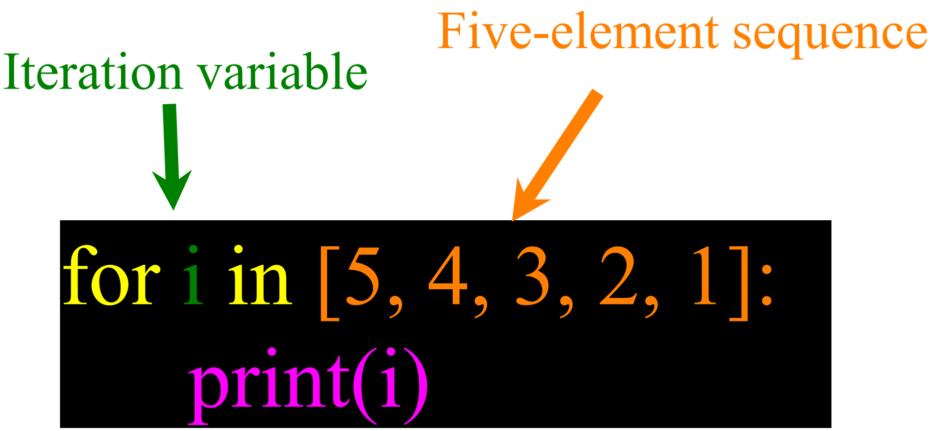
```
for iterator in expression_list:
            statement 1
            statement 2
            …
            statement N
```

for statement

- The flow of execution
  - ❑The expression list is evaluated once; it should yield an iterable object (e.g., list, tuple, etc.)
  - ❑For each member in the expression_list, execute all statements in the for body.

The *for* statement

- The iteration variable "iterates" though the sequence (ordered set)
- The block (body) of code is executed once for each element in the sequence
- The iteration variable moves through all of the values in the sequence



Iteration variable

Five-element sequence

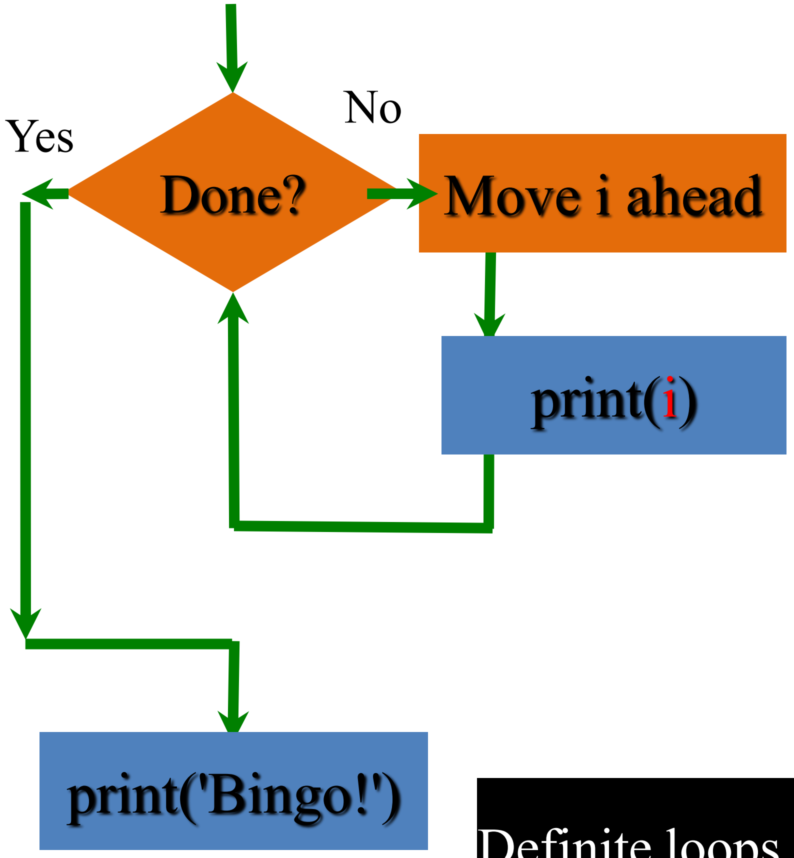
```
for i in [5, 4, 3, 2, 1]:
    print(i)
```

Example (1)

```
for i in [5, 4, 3, 2, 1]:
    print(i)
print('Bingo!')
```

```
Output
5
4
3
2
1
Bingo!
```

The *for* statement



```
for i in [5, 4, 3, 2, 1]:
    print(i)
print('Bingo!')
```

```
5
4
3
2
1
Bingo!
```

Definite loops (for loops) have explicit iteration variables that change each time through a loop. These iteration variables move through the sequence or set.

Example (2)

```
for i in [5, 4, 3, 2, 1]:
    if i % 2 == 0:
        print(i, “: even”)
    else:
        print(i, “: odd”)
print('Bingo!')
```

```
Output
5: odd
4: even
3: odd
2: even
1: odd
Bingo!
```

Nested *for* statement

- Syntax:

```
for iterator in expression_list:
        statement 1
        statement 2
        …
        for iterator in expression_list:
                statement 1
                …
                statement N
        statement (outer for)
statements (after outer for)
```

Outer for statement

Inner for statement

Example (1)

```
for i in [1, 2, 3] :
    for j in [1, 2, 3]:
        print(i*j)
print('Bingo!')
```

```
Output
1
2
3
2
4
6
3
6
9
Bingo!
```

Example (2)

```
for i in [1, 2, 3]:
    j = 1
    while j<=i:
        print(i)
        j = j+1
print('Bingo!')
```

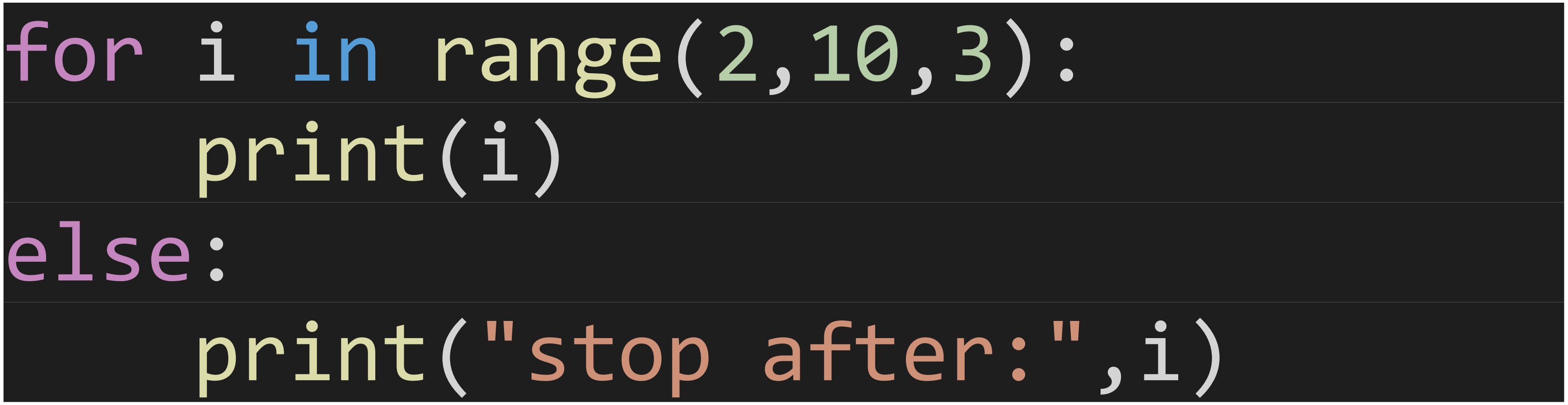
```
Output
1
2
2
3
3
3
Bingo!
```

range()

- The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops
- range(m) range from zero to m-1
- range(x, y) range from x to y-1
- range(x, y, step_size)
- If x>y, empty range object

```
>>> list(range(4))
[0,1,2,3]

>>> list(range(3,9))
[3,4,5,6,7,8]

>>> list(range(3,9,2))
[3,5,7]

>>> list(range(4,1))
[]
```

For else

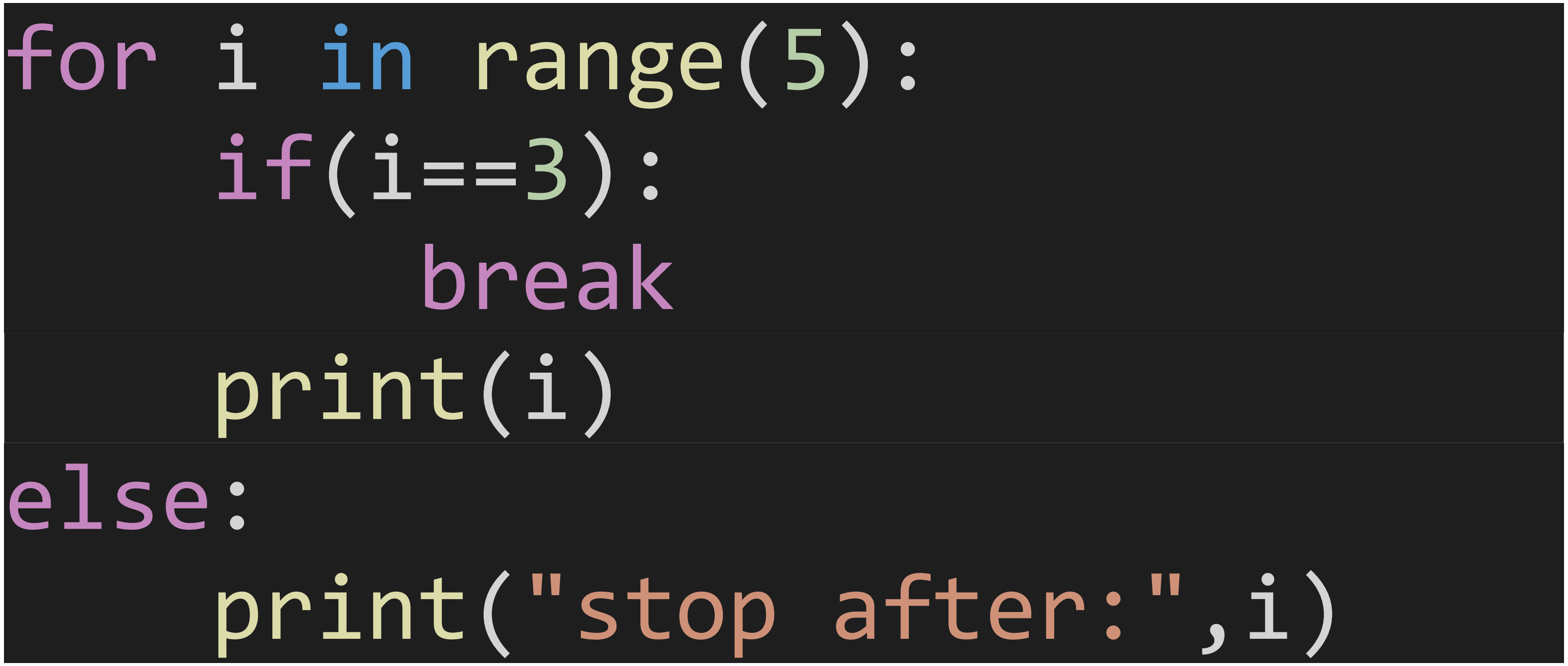


```python
for i in range(2,10,3):
    print(i)
else:
    print("stop after:",i)
```

Output

```
2
5
8
stop after: 8
>>>
```

For else



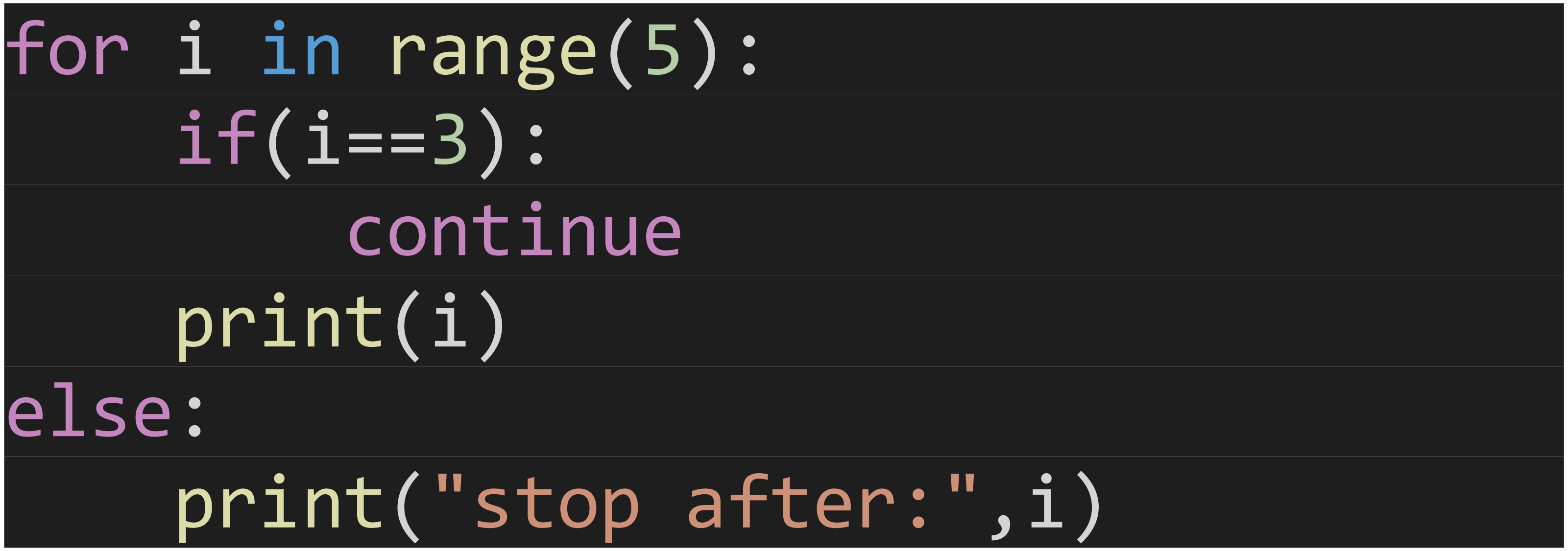
```python
for i in range(5):
    if(i==3):
        break
    print(i)
else:
    print("stop after:",i)
```

Output

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

“else” is **not** executed after **break**

For else



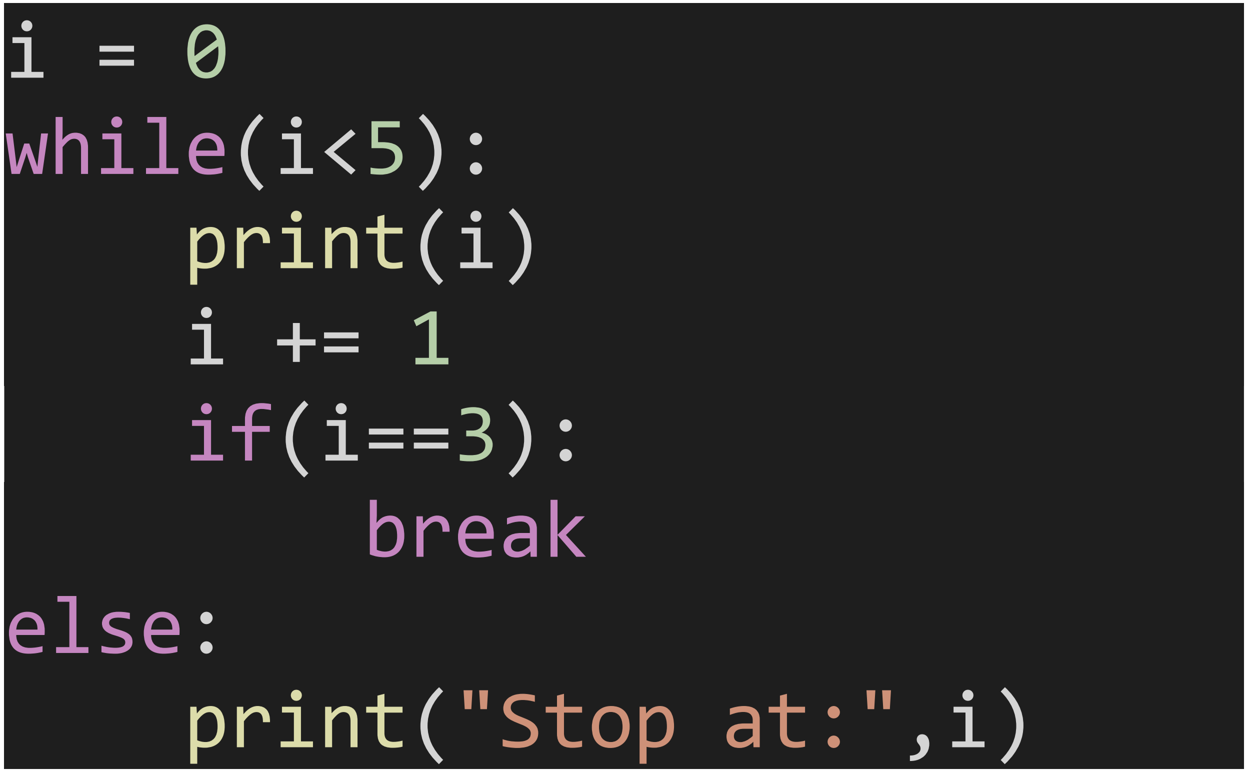
```
for i in range(5):
    if(i==3):
        continue
    print(i)
else:
    print("stop after:",i)
```

Output

```
0
1
2
4
stop after: 4
>>>
```

“else” is executed after **continue**

While else



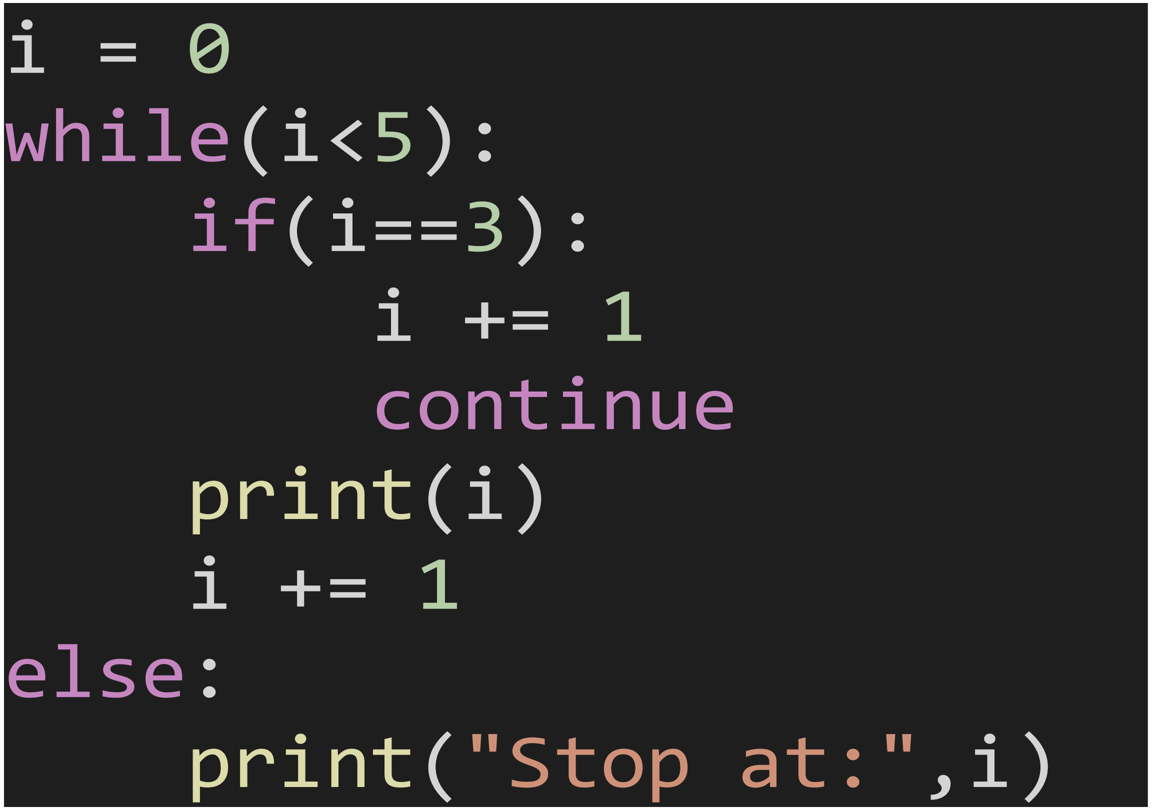
```python
i = 0
while(i<5):
    print(i)
    i += 1
    if(i==3):
        break
else:
    print("Stop at:",i)
```

Output

```
0
1
2
```

“else” is **not** executed after **break**

While else



```
i = 0
while(i<5):
    if(i==3):
        i += 1
        continue
    print(i)
    i += 1
else:
    print("Stop at:",i)
```

Output

```
0
1
2
4
Stop at: 5
>>>
```

"else" is executed after **continue**

While Loop vs. For Loop

- For loop knows the number of times of the loop
  - ✓ based on a generator or a sequence of items
  - ✓ always terminate
- While loop does not know the number of times
  - ✓ based on a condition (True or False)
  - ✓ may not terminate (infinite)
- Any for loop can be converts into while loop
- It is better to use for loop when it is possible

What is the largest number

3 41 12 9 74 15

largest_so_far -1 3 41 74

What is the largest number

```
largest_so_far = -1
for current in [3, 41, 12, 9, 74, 15]:
    if current > largest_so_far:
        largest_so_far = current
print(largest_so_far)
```

Counting in a loop

```
i = 0
print('Before', i)
for thing in [9, 41, 12, 3, 74, 15] :
    i= i+ 1
    print(i, thing)
print('After’, i)
```

```
python countloop.py
Before 0
1 9
2 41
3 12
4 3
5 74
6 15
After 6
```

To count how many times we execute a loop we introduce a counter variable that starts at 0 and we add one to it each time through the loop.

Summing in a loop

```
sum = 0
print('Before', sum)
for thing in [9, 41, 12, 3, 74, 15] :
    sum= sum+ thing
    print(sum, thing)
print('After', sum)
```

```
python sumloop.py
Before 0
9 9
50 41
62 12
65 3
139 74
154 15
After 154
```

To add up a value we encounter in a loop, we introduce a sum variable that starts at 0 and we add the value to sum each time through the loop.

Finding the average in a loop

```
count = 0
sum = 0
print('Before', count, sum)
for value in [9, 41, 12, 3, 74, 15] :
    count = count+1
    sum= sum+ value
    print(count, sum, value)
print('After', count, sum, sum/count)
```

```
python avgloop.py
Before 0 0
1 9 9
2 50 41
3 62 12
4 65 3
5 139 74
6 154 15
After 6 154 25.66
```

An average just combines the counting and sum patterns and divides when the loop is done.

Search in a loop

```
found = False
print('Before', found)
for value in [9, 41, 12, 3, 74, 15] :
    if value == 3:
        found = True
    print(found, value)
print('After', found)
```

```
python searchloop.py
Before False
False 9
False 41
False 12
True 3
True 74
True 15
After True
```

If we just want to search and know if a value was found, we use a variable that start at False and is set to True as soon as we find the value.

Two iteration variables

- We loop through the key-value pairs in a dictionary using two iteration variables
- Each iteration, the first variable is the key and the second variable is the corresponding value for the key

```
students = {‘name’:’alice’, ‘age’:20, ’gender’: ‘f’}

for k,v in students.items():
      print(k,”:”,v)

Outputs:
name: alice
age: 20
gender: f
```

Python Program Structure

- **A Python program is constructed from code blocks**
- **Block is a piece of Python program statements that is executed as a unit, i.e., module, function, class, etc**
- **Interactive session, statements are executed as they are typed in, until the interpreter is terminated**
- **Script file (xx.py), the interpreter reads statements from the file and executes them until end-of-file (EOF) is encountered**

Python Program Structure

- Each statement usually occupies a single line ending with the newline character (Pythonic)

Newline \n

```
print("Hello World 1")
```

- Multiple statements per line separated by semicolon ‘;’

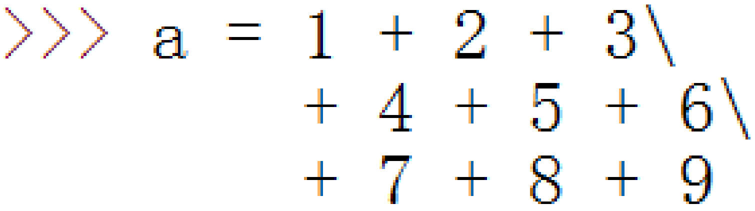
```
print("Hello World 1"); … ; print("Hello World 2")
```

- Pythonic: PEP 8 -- Style Guide for Python Code

  https://www.python.org/dev/peps/pep-0008

One statement in multiple lines

- **Explicit Line Continuation:** in cases where implicit line continuation is not readily available or practicable, you can specify a backslash ‘\’ character

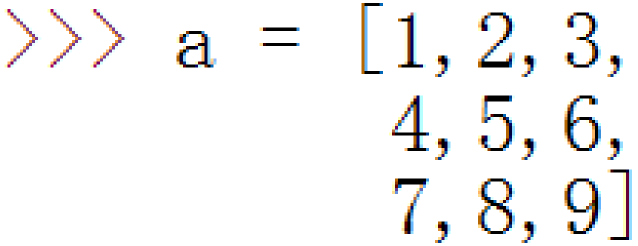


```
>>> a = 1 + 2 + 3\
        + 4 + 5 + 6\
        + 7 + 8 + 9
```

Newline **'\n'** characters after ‘ \ ’

One statement in multiple lines

- **Implicit Line Continuation:** any statement containing '(', '[', or '{' is presumed to be incomplete until all matched



```
>>> a = [1, 2, 3,
         4, 5, 6,
         7, 8, 9]
```

Newline **'\n'** characters after ‘ , ’

Python Program Structure

- **Comments: the hash character (#) signifies a comment. The interpreter will ignore everything from the hash character through the end of that line**

  #This is a comment

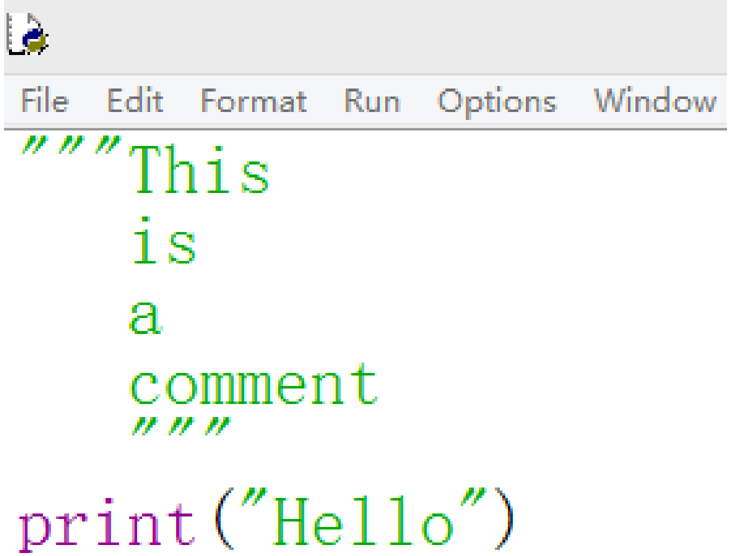
- **But, a hash character inside a string literal is protected, and does not indicate a comment**

  “# This is not a comment, it is a string”

- **Using multiple hash characters (#) for block comments**

Python Program Structure

- **Triple-quoted string:** ''' or """ can span multiple lines, it can effectively function as a multiline comment in script file ***.py, **not in interactive session**
- **But**, this is called **docstring** and used as a special comment at the beginning of a user-defined function that documents the function’s behavior (to be **Pythonic**)



```
"""This
   is
   a
   comment
   """

print("Hello")
```

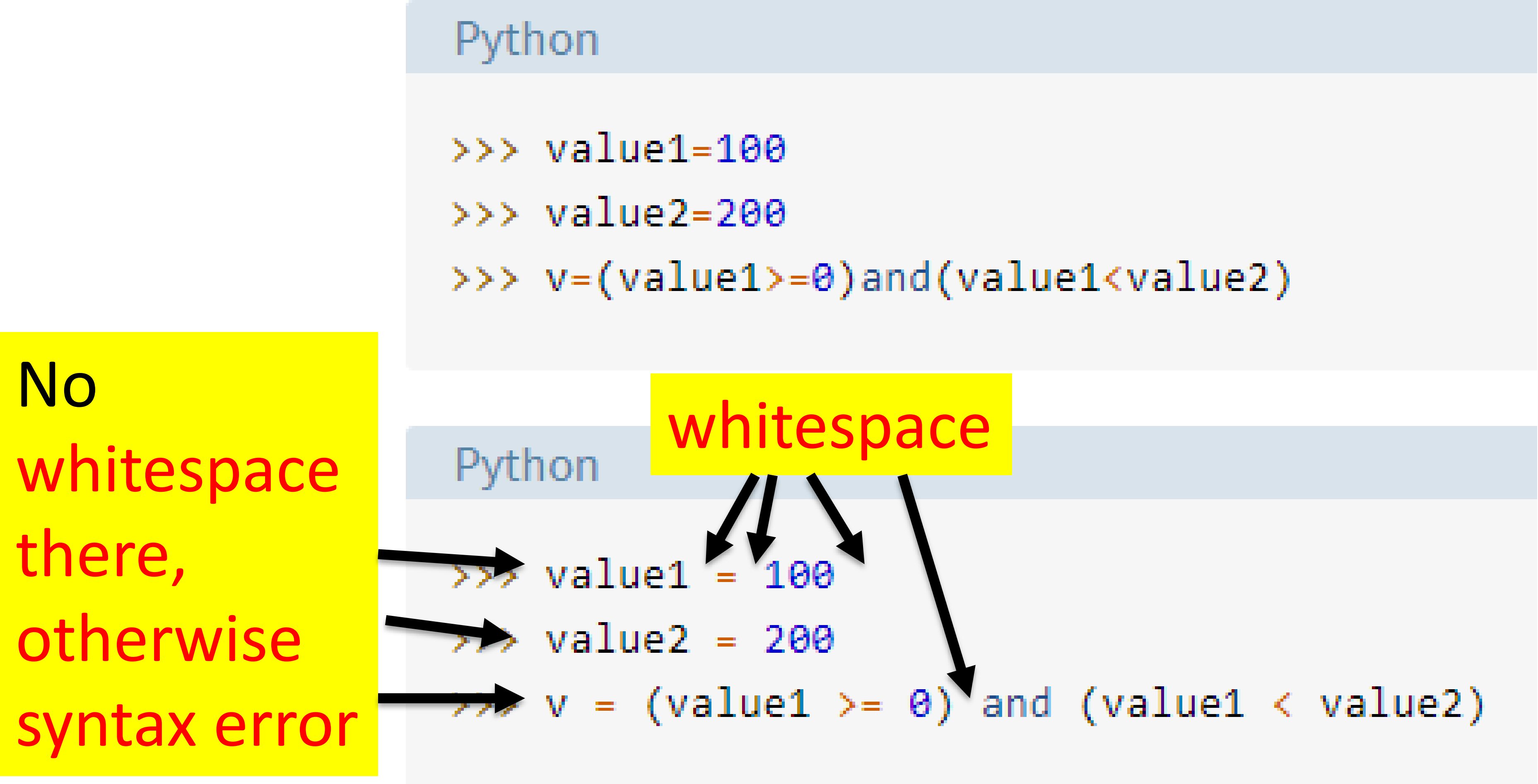
Python Program Structure

- **Whitespace: almost always enhances readability in most programming languages**

Character	Literal Expression
space	' '
tab	'\t'
newline	'\n'

Python Program Structure

- These programs are identical, whitespace are used for **readability**



```
>>> value1=100
>>> value2=200
>>> v=(value1>=0)and(value1<value2)
```

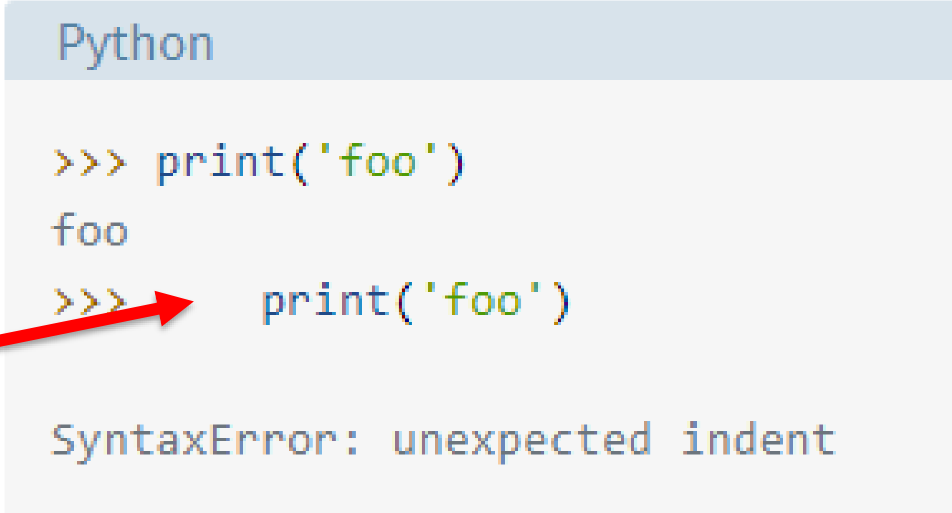
```
>>> value1 = 100
>>> value2 = 200
>>> v = (value1 >= 0) and (value1 < value2)
```

whitespace

No whitespace there, otherwise syntax error

Python Program Structure

- **Whitespace as Indention**
  - **whitespace for Python code Indention**
  - **whitespace that appears to the left of the first token on a line—used to compute a line's indention level, which in turn is used to determine grouping of statements**



Python

```
>>> print('foo')
foo
>>>     print('foo')

SyntaxError: unexpected indent
```

Syntax error

Readings (recommended)

- The Python Tutorial
  - 5. Data Structures

Recap

- Condition
- Python Program Structure