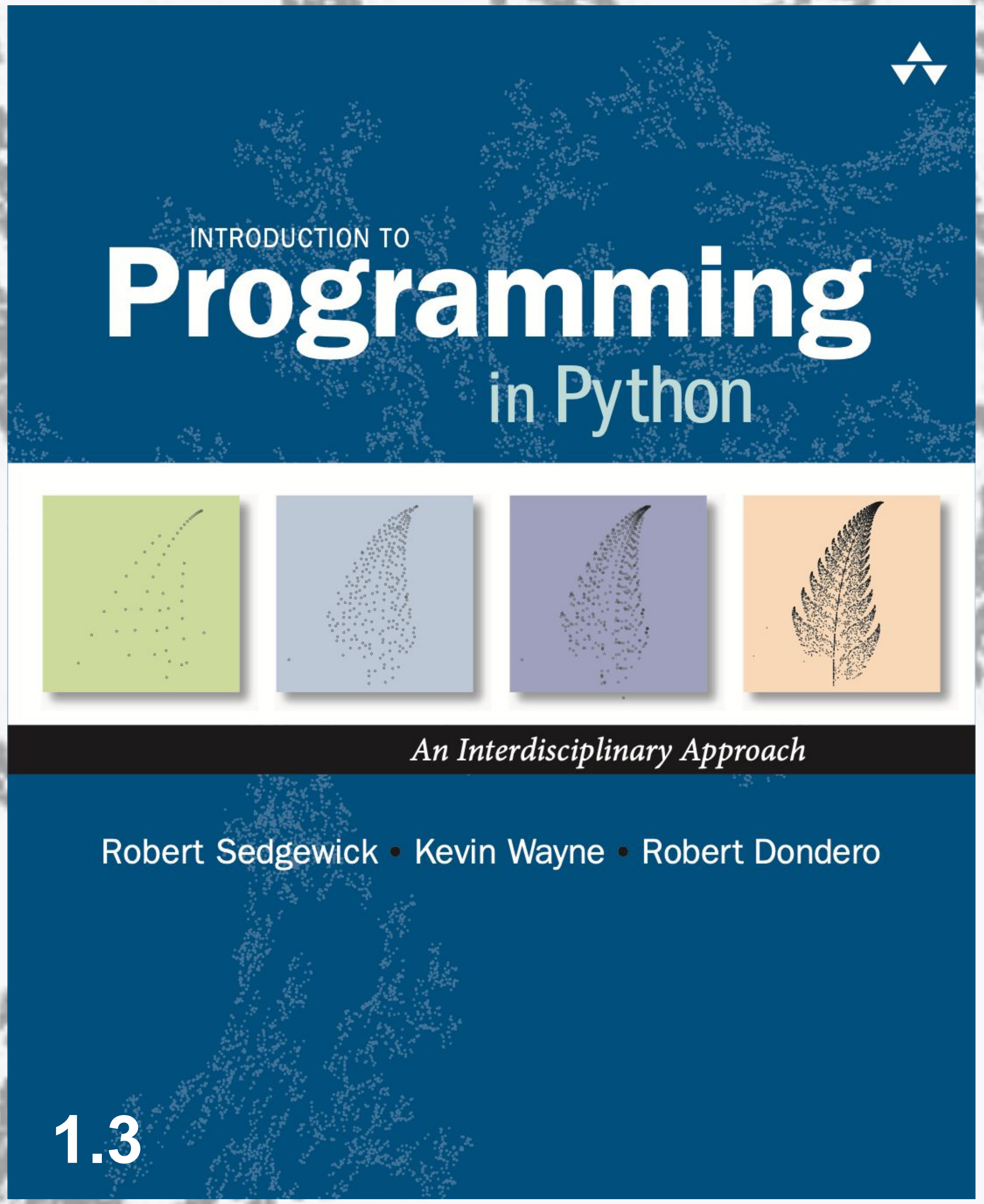
INTRO TO PROGRAMMING
IN PYTHON

SEDGEWICK · WAYNE · DONDERO



INTRODUCTION TO
Programming
in Python

An Interdisciplinary Approach

Robert Sedgewick · Kevin Wayne · Robert Dondero

1.3

https://introcs.cs.princeton.edu/python

2. Conditionals & Loops

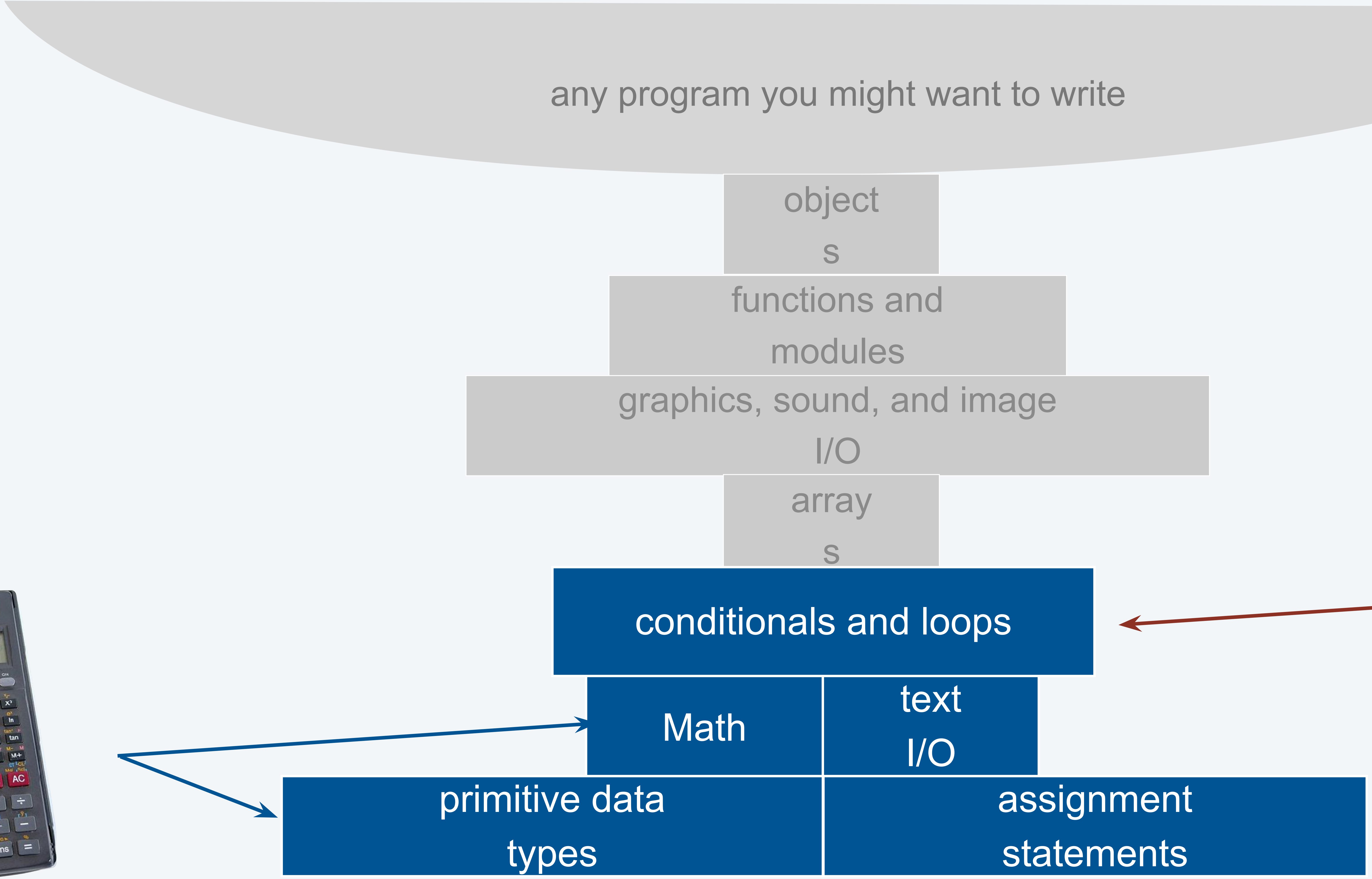
INTRO TO PROGRAMMING
IN PYTHON

SEDGEWICK·WAYNE·DONDERO

2. Conditionals & Loops

- Conditionals: the `if` statement
- Loops: the `while` statement
- An alternative: the `for` loop
- `Do-While` loop
- Nesting
- Debugging

Context: basic building blocks for programming



any program you might want to write

objects

functions and modules

graphics, sound, and image I/O

arrays

conditionals and loops

Math

text I/O

primitive data types

assignment statements

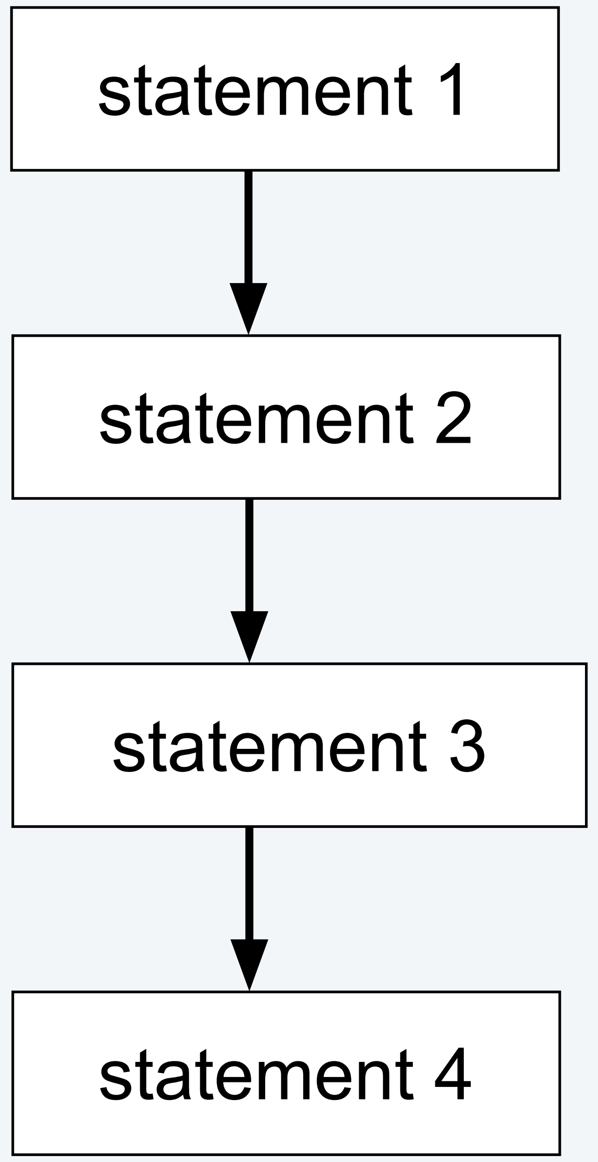
Previous lecture:
equivalent to a calculator

This lecture:
to infinity and beyond!

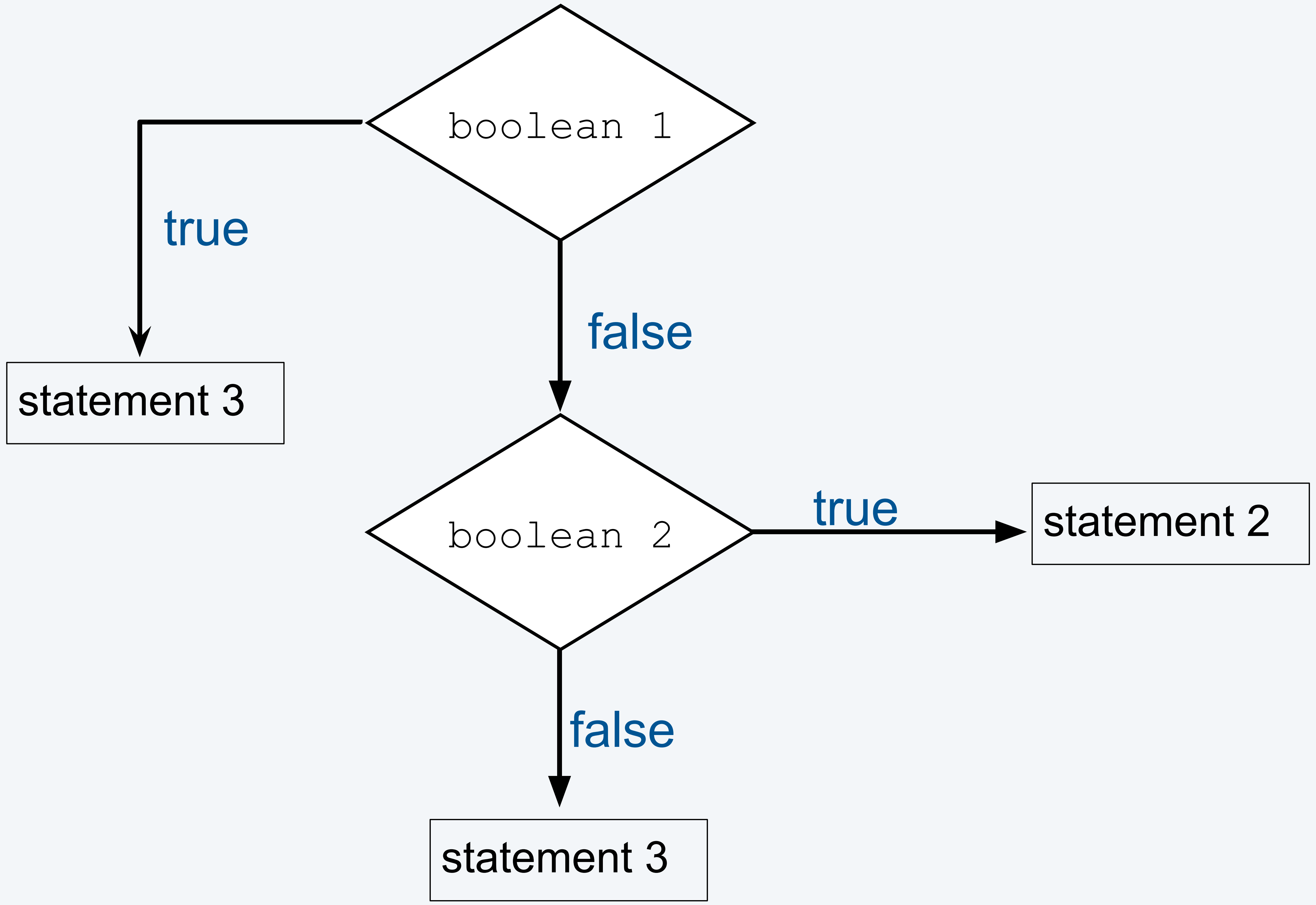
Conditionals and Loops

Control flow

- The sequence of statements that are actually executed in a program.
- Conditionals and loops enable us to choreograph control flow.



straight-line control flow



control flow with conditionals and loops

Conditionals

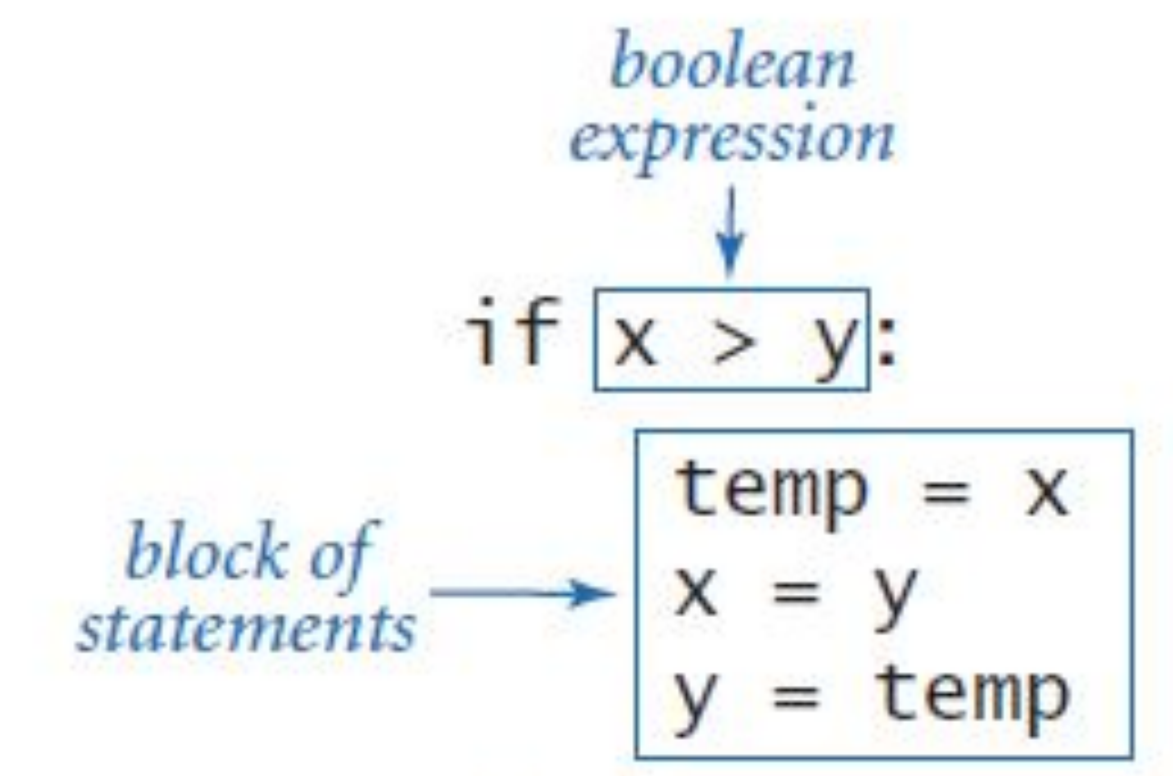
The if statement

The if statement. A common branching structure.

- Evaluate a `boolean` expression.
- If True, execute a statement.
- The `else` option: If False, execute a different statement.

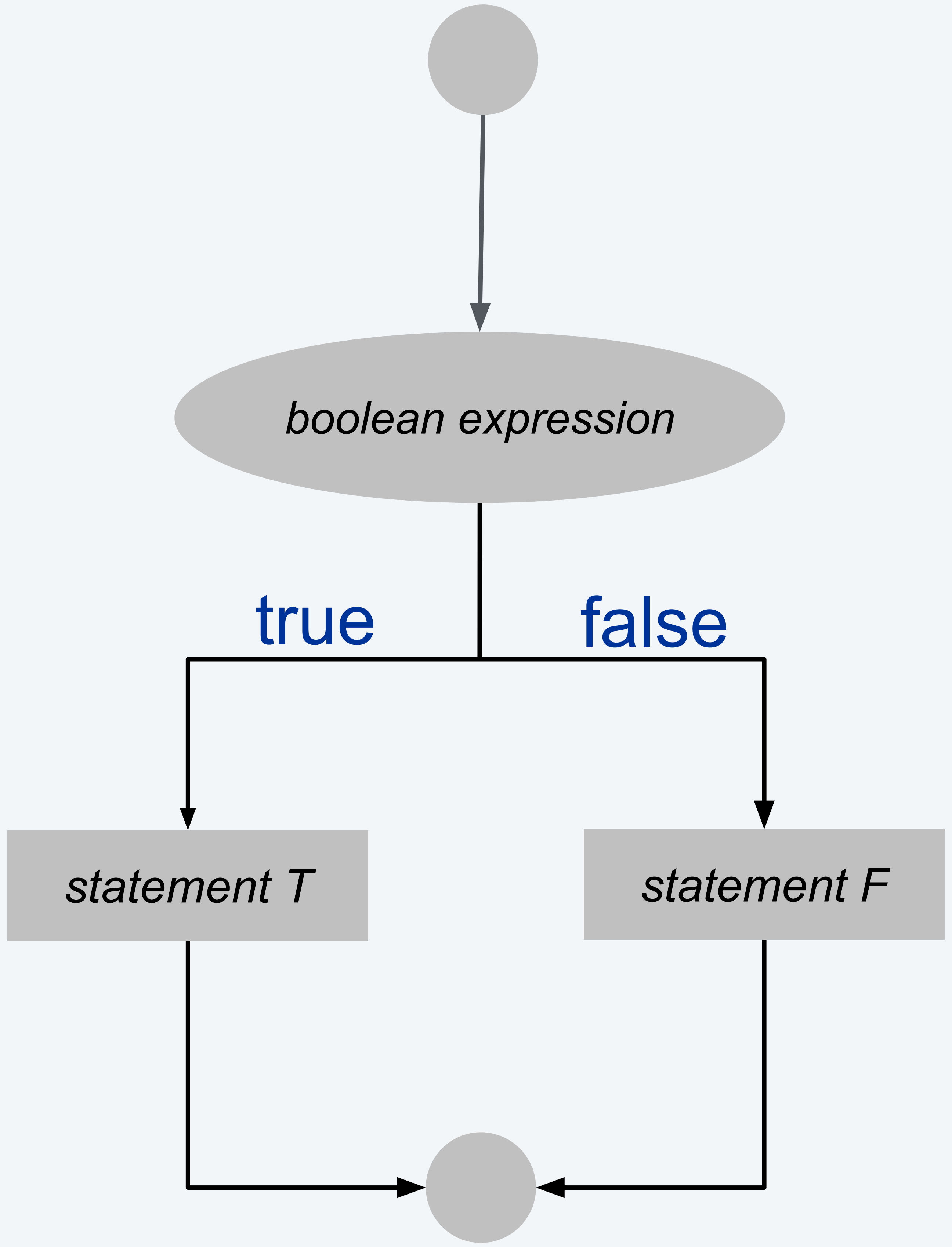
```
if (boolean expression):
    statement T
else:
    statement F
```

can be any sequence of statements

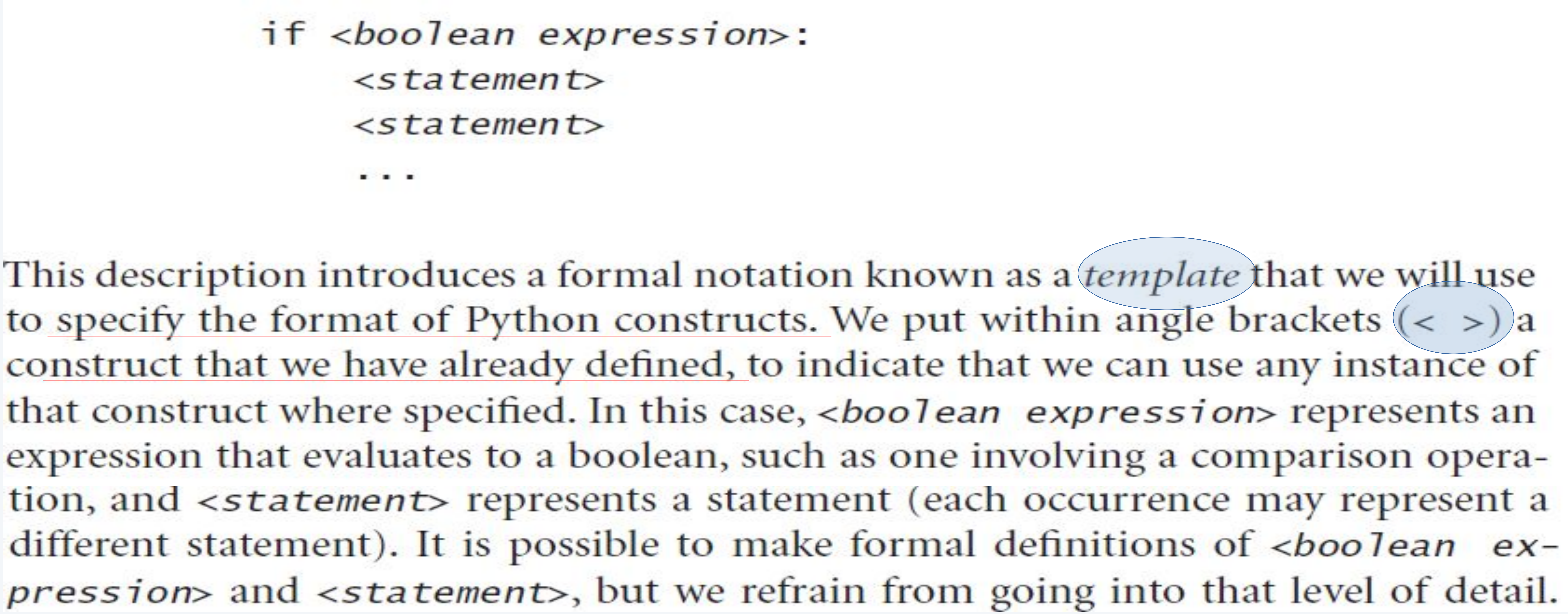


```
if x > y:
    temp = x
    x = y
    y = temp
```

Anatomy of an if statement



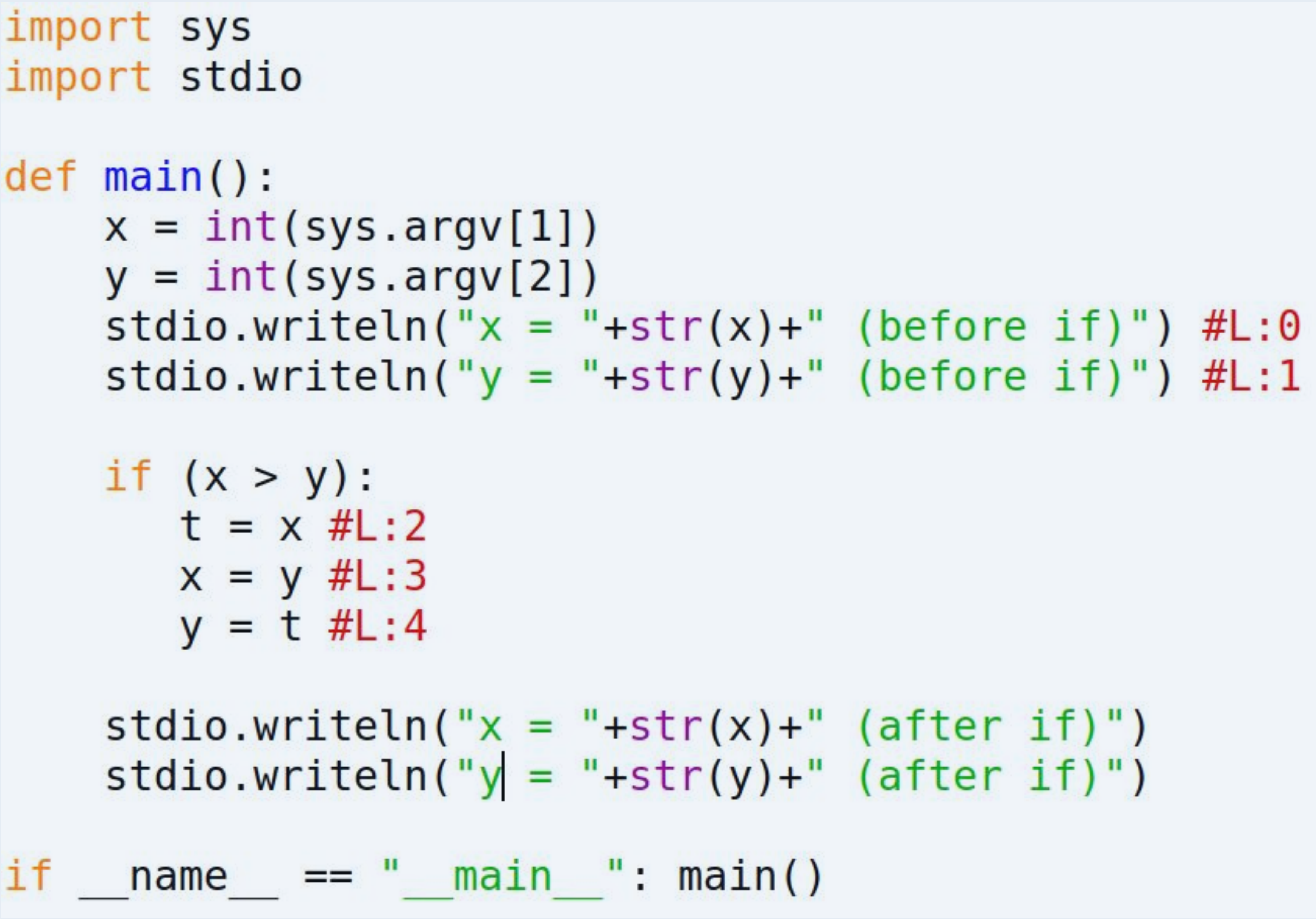
Template



```
if <boolean expression>:
    <statement>
    <statement>
    ...
```

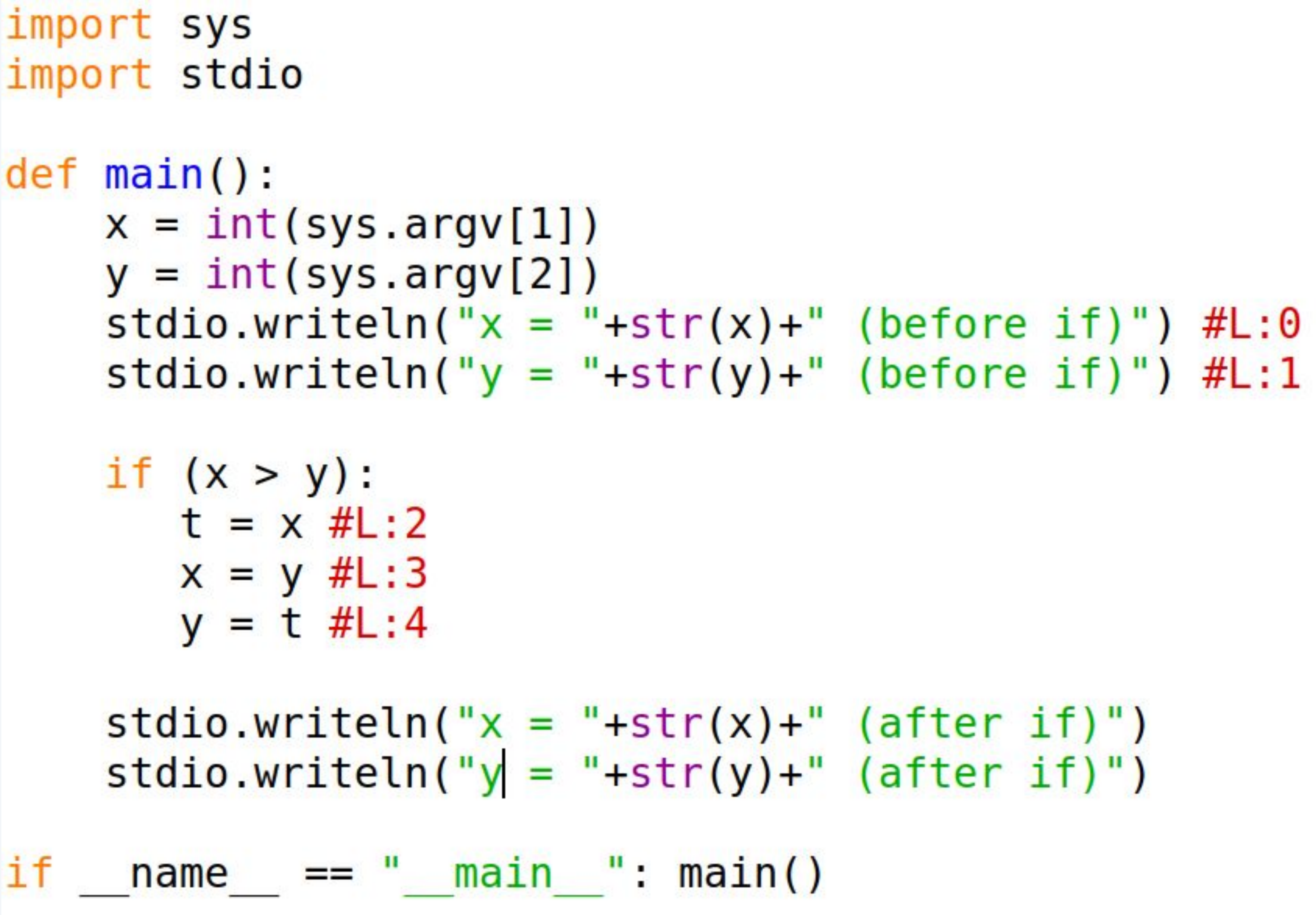
This description introduces a formal notation known as a *template* that we will use to specify the format of Python constructs. We put within angle brackets (< >) a construct that we have already defined, to indicate that we can use any instance of that construct where specified. In this case, `<boolean expression>` represents an expression that evaluates to a boolean, such as one involving a comparison operation, and `<statement>` represents a statement (each occurrence may represent a different statement). It is possible to make formal definitions of `<boolean expression>` and `<statement>`, but we refrain from going into that level of detail.

If Statement Example



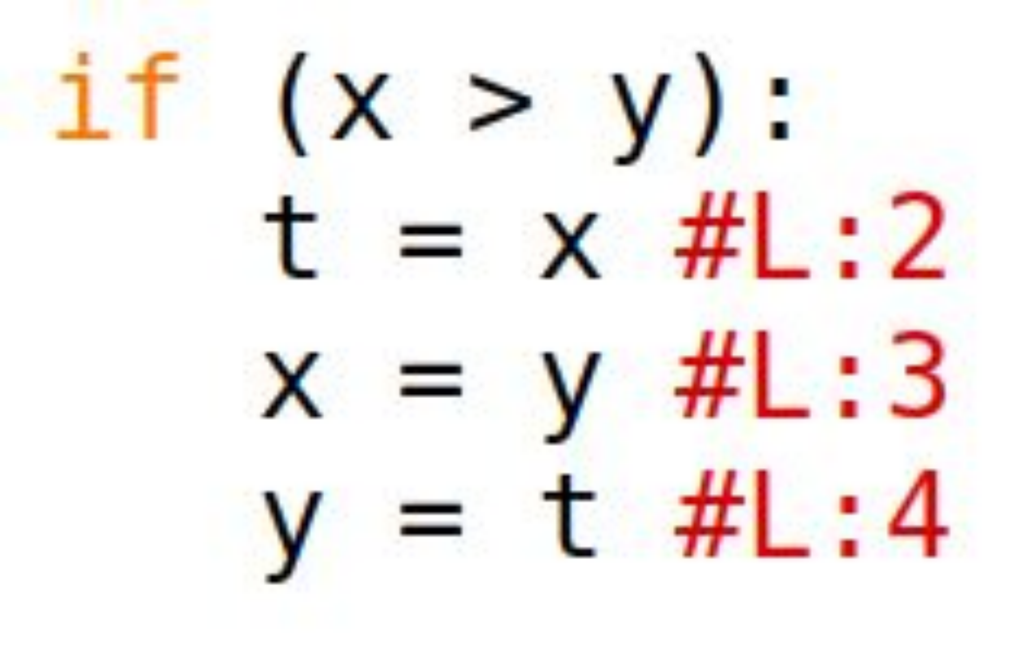
```
import sys
import stdio

def main():
    x = int(sys.argv[1])
    y = int(sys.argv[2])
    stdio.writeln("x = "+str(x)+" (before if)") #L:0
    stdio.writeln("y = "+str(y)+" (before if)") #L:1

    if (x > y):
       t = x #L:2
       x = y #L:3
       y = t #L:4

    stdio.writeln("x = "+str(x)+" (after if)")
    stdio.writeln("y = "+str(y)+" (after if)")

if __name__ == "__main__": main()
```

CASE 1: Ascending Order

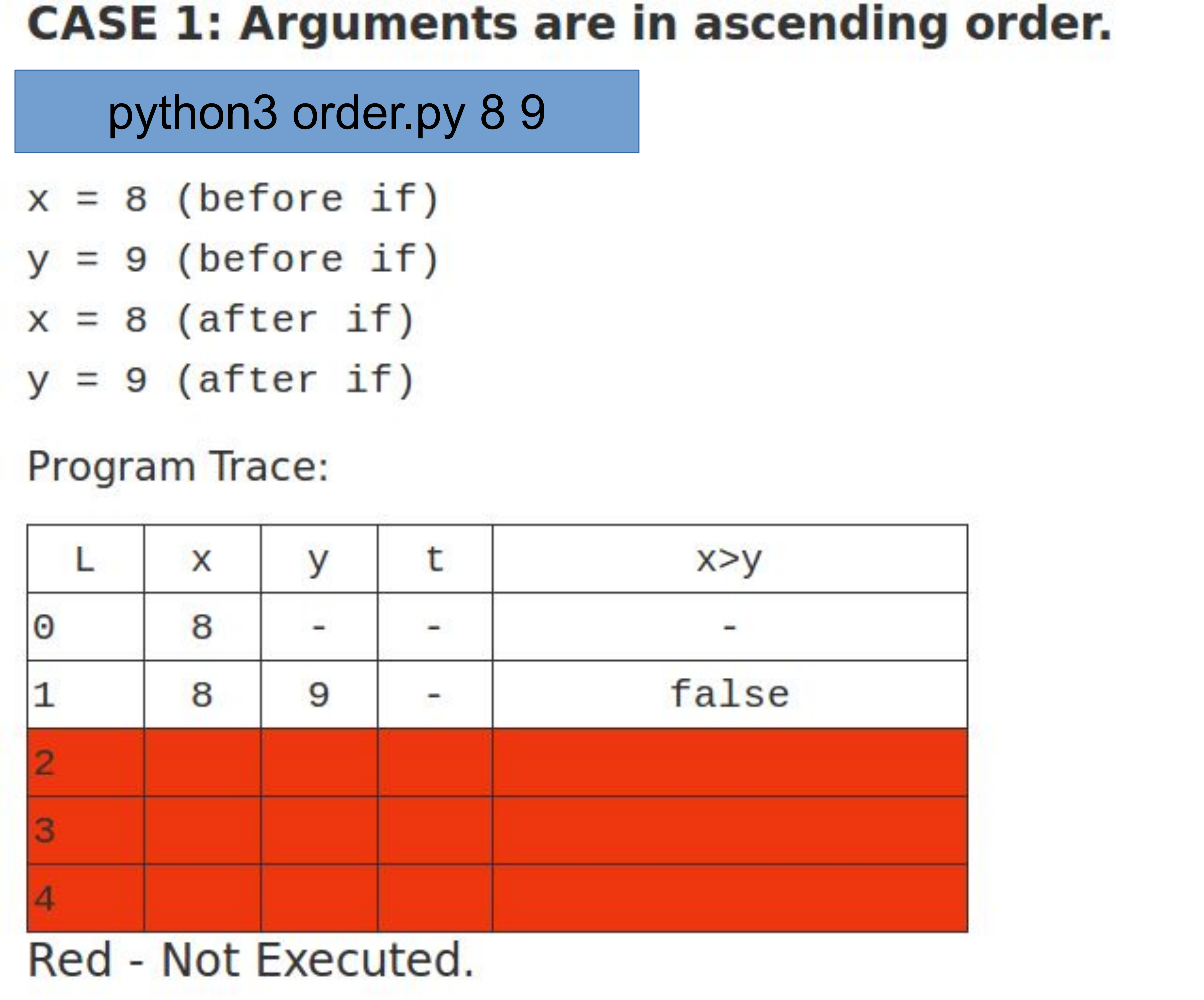


```
import sys
import stdio

def main():
    x = int(sys.argv[1])
    y = int(sys.argv[2])
    stdio.writeln("x = "+str(x)+" (before if)") #L:0
    stdio.writeln("y = "+str(y)+" (before if)") #L:1

    if (x > y):
       t = x #L:2
       x = y #L:3
       y = t #L:4

    stdio.writeln("x = "+str(x)+" (after if)")
    stdio.writeln("y = "+str(y)+" (after if)")

if __name__ == "__main__": main()
```

8 > 9 is *false*



```
if (x > y):
    t = x #L:2
    x = y #L:3
    y = t #L:4
```



CASE 1: Arguments are in ascending order.

```
python3 order.py 8 9
```

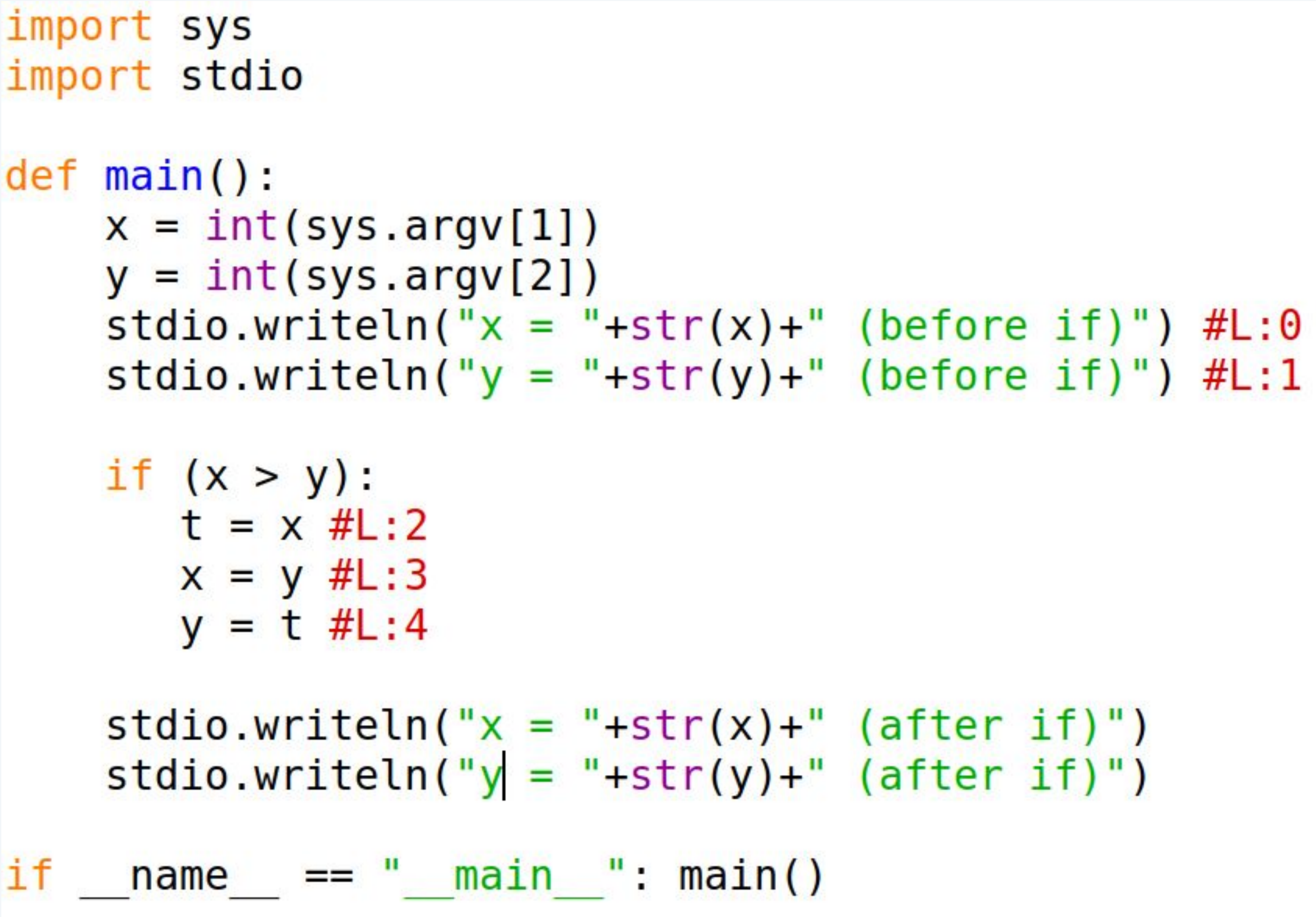
```
x = 8 (before if)
y = 9 (before if)
x = 8 (after if)
y = 9 (after if)
```

Program Trace:

L	x	y	t	x>y
0	8	-	-	-
1	8	9	-	false
2 🟥	🟥	🟥	🟥	🟥
3 🟥	🟥	🟥	🟥	🟥
4 🟥	🟥	🟥	🟥	🟥

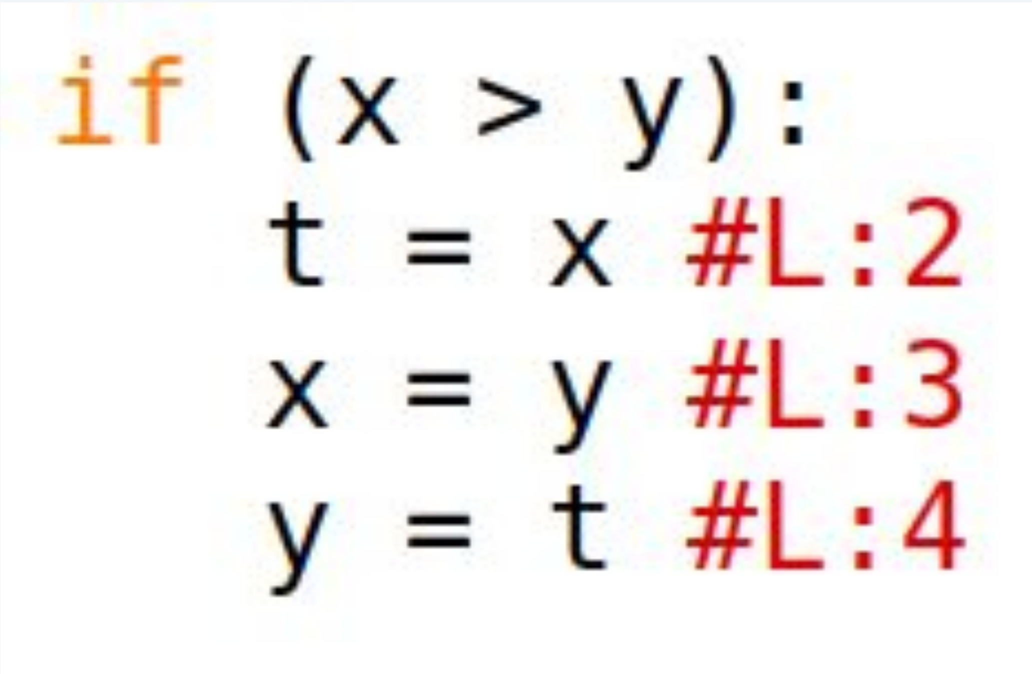
Red - Not Executed.

CASE 2: Descending Order

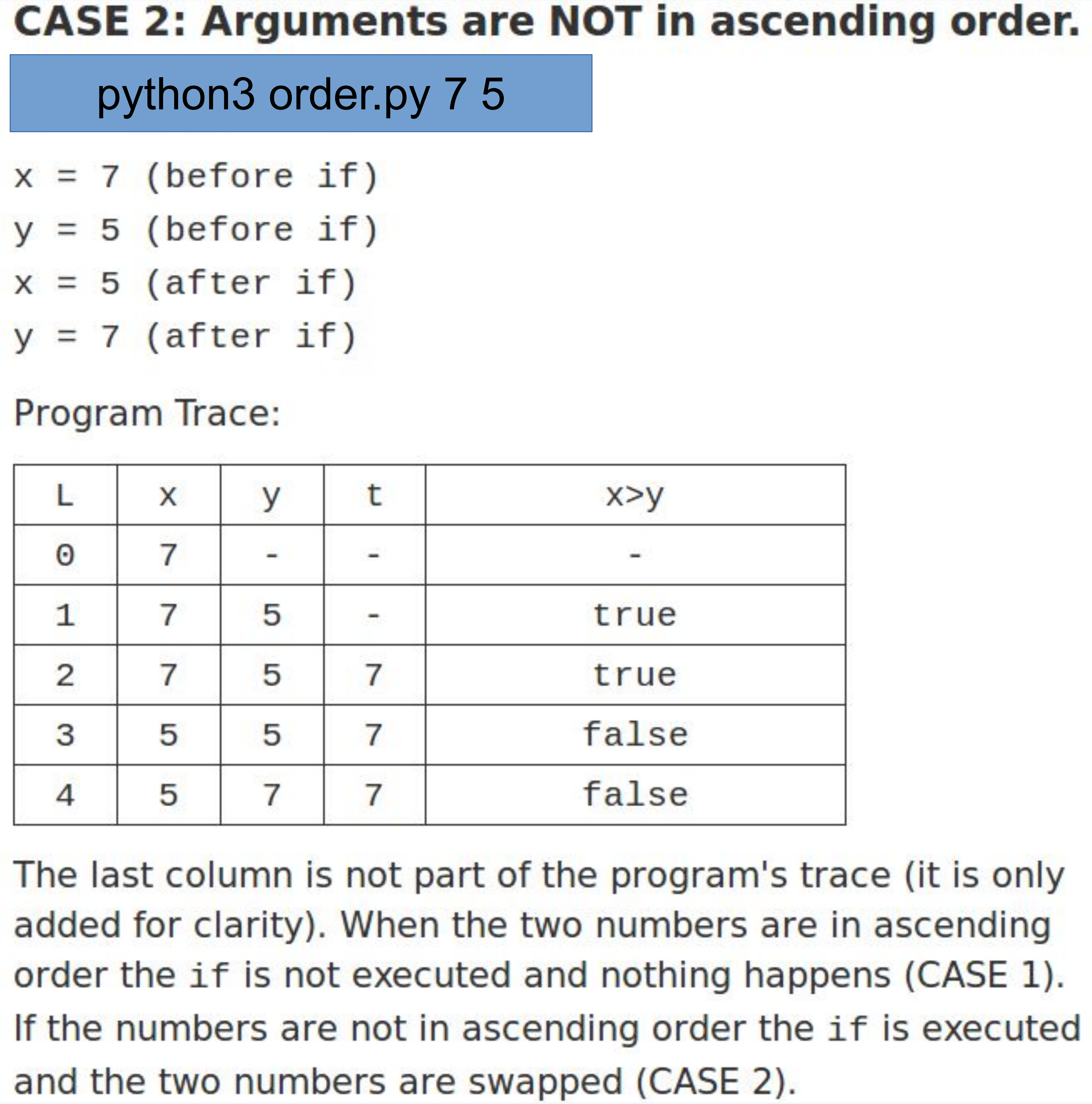


```
import sys
import stdio

def main():
    x = int(sys.argv[1])
    y = int(sys.argv[2])
    stdio.writeln("x = "+str(x)+" (before if)") #L:0
    stdio.writeln("y = "+str(y)+" (before if)") #L:1

    if (x > y):
       t = x #L:2
       x = y #L:3
       y = t #L:4

    stdio.writeln("x = "+str(x)+" (after if)")
    stdio.writeln("y = "+str(y)+" (after if)")

if __name__ == "__main__": main()
```

7 > 5 is *true*



```
if (x > y):
   t = x #L:2
   x = y #L:3
   y = t #L:4
```



CASE 2: Arguments are NOT in ascending order.

python3 order.py 7 5

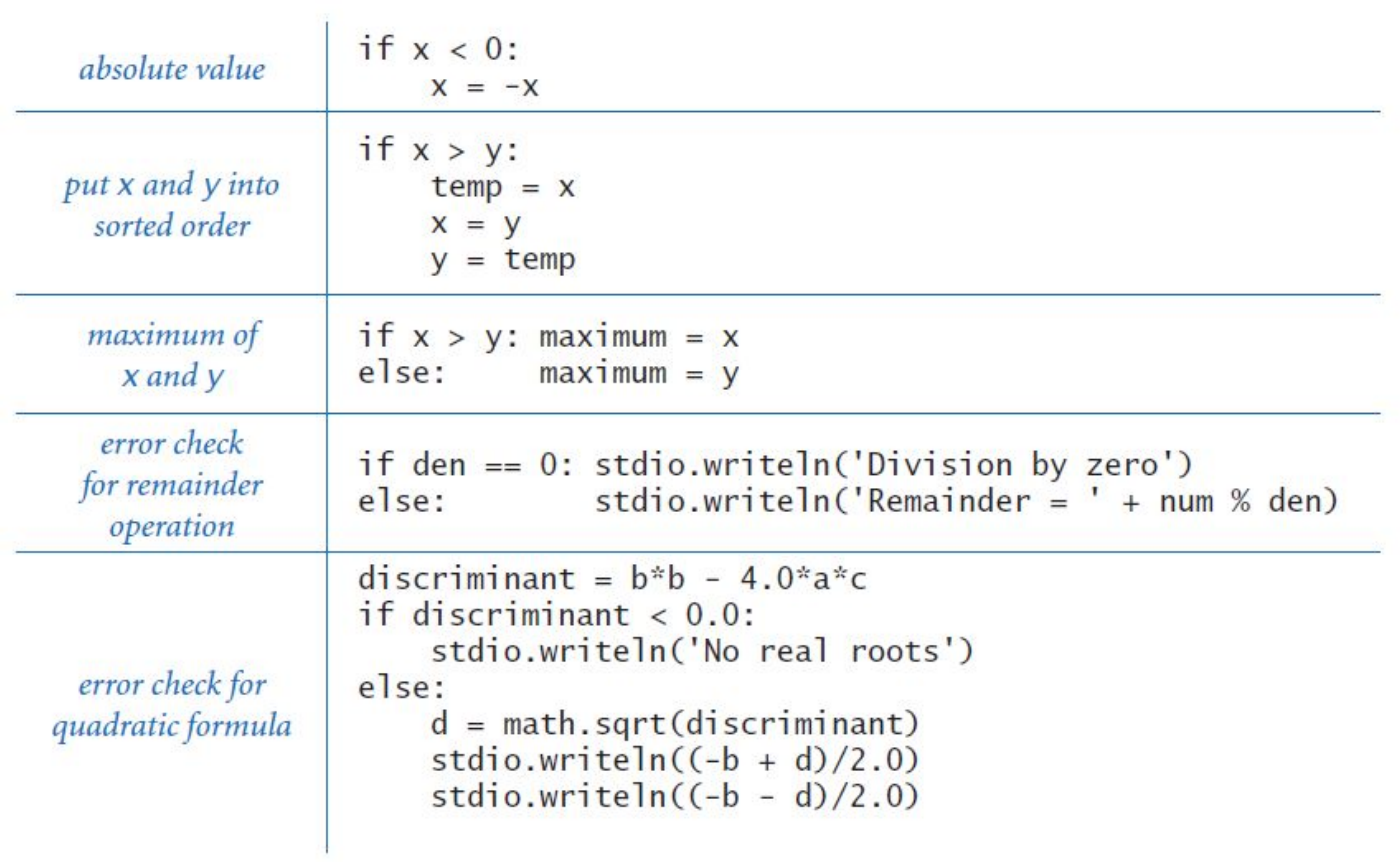
```
x = 7 (before if)
y = 5 (before if)
x = 5 (after if)
y = 7 (after if)
```

Program Trace:

L	x	y	t	x>y
0	7	-	-	-
1	7	5	-	true
2	7	5	7	true
3	5	5	7	false
4	5	7	7	false

The last column is not part of the program's trace (it is only added for clarity). When the two numbers are in ascending order the `if` is not executed and nothing happens (CASE 1). If the numbers are not in ascending order the `if` is executed and the two numbers are swapped (CASE 2).

If Statement Examples



absolute value	`if x < 0:`
	`    x = -x`

put x and y into sorted order	`if x > y:`
	`    temp = x`
	`    x = y`
	`    y = temp`

maximum of x and y	`if x > y: maximum = x`
	`else:      maximum = y`

error check for remainder operation	`if den == 0: stdio.writeln('Division by zero')`
	`else:        stdio.writeln('Remainder = ' + num % den)`

error check for quadratic formula	`discriminant = b*b - 4.0*a*c`
	`if discriminant < 0.0:`
	`    stdio.writeln('No real roots')`
	`else:`
	`    d = math.sqrt(discriminant)`
	`    stdio.writeln((-b + d)/2.0)`
	`    stdio.writeln((-b - d)/2.0)`

Typical examples of using `if` *statements*

Example of if statement use: simulate a coin flip

Take different action depending on value of variable.

flip.py

```
import stdrandom


def main():
    if stdrandom.uniform() < 0.5:
        stdio.writeln('Heads')
    else:
        stdio.writeln('Tails')

if __name__ == '__main__': main()
```

```
% python flip.py
Heads

% python flip.py
Heads

% python flip.py
Tails

% python flip.py
Heads
```

Example of `if` statement use

2-sort

Q. What does this program do?

TwoSort.py

```
def main():
    a = int(sys.argv[1])
    b = int(sys.argv[2])
    if (b < a):
        t = a
        a = b
        b = t
    stdio.writeln(a)
    stdio.writeln(b)
```

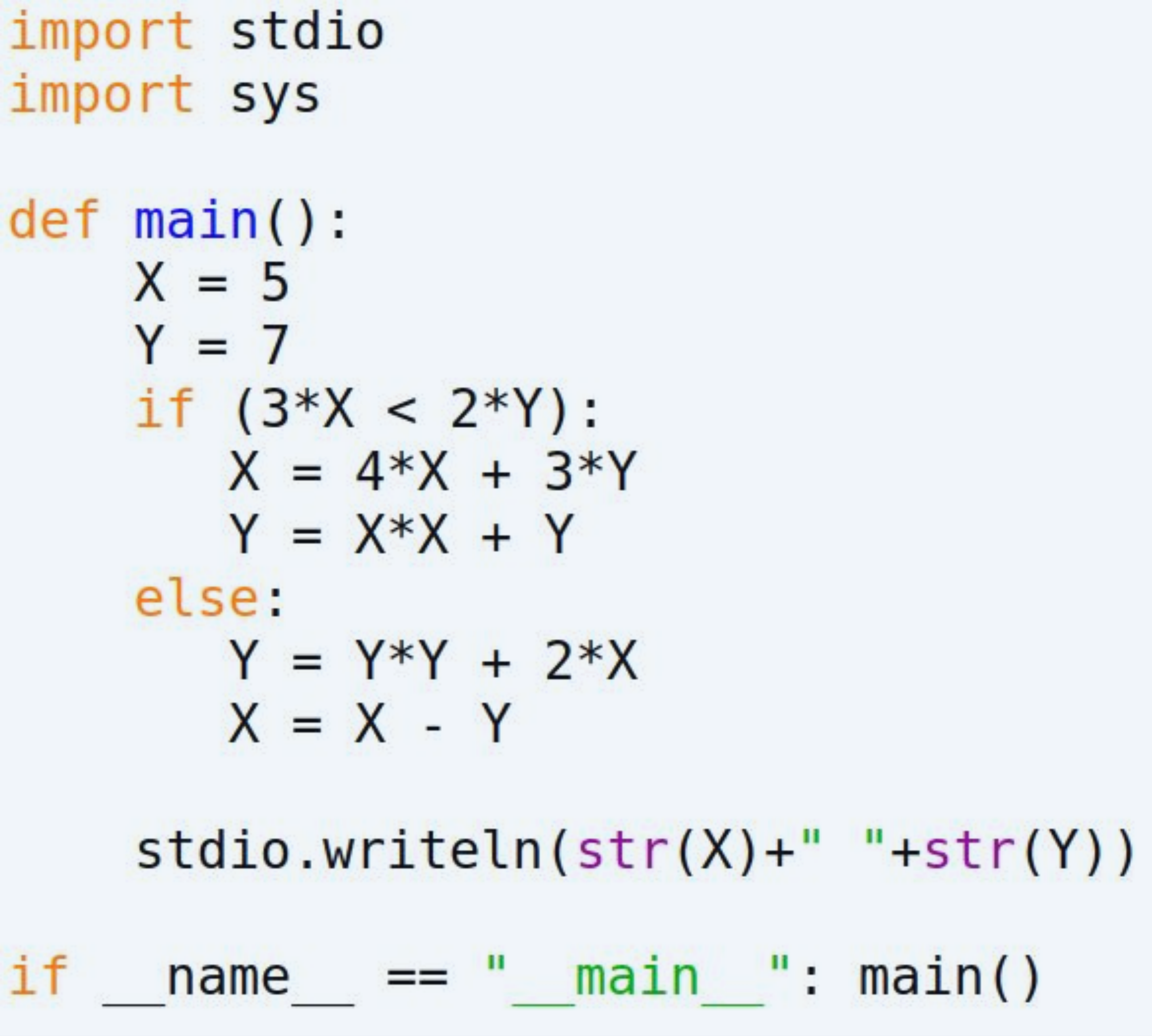
alternatives for if and else can be a *sequence* of statements

```
% python TwoSort.py 1234 99
99
1234

% python TwoSort.py 99 1234
99
1234
```

A. Reads two integers from the command line, then prints them out in numerical order.

Exercise: If Example

What does the following program print to the terminal?



```
import stdio
import sys

def main():
    X = 5
    Y = 7
    if (3*X < 2*Y):
       X = 4*X + 3*Y
       Y = X*X + Y
    else:
       Y = Y*Y + 2*X
       X = X - Y

    stdio.writeln(str(X)+" "+str(Y))

if __name__ == "__main__": main()
```

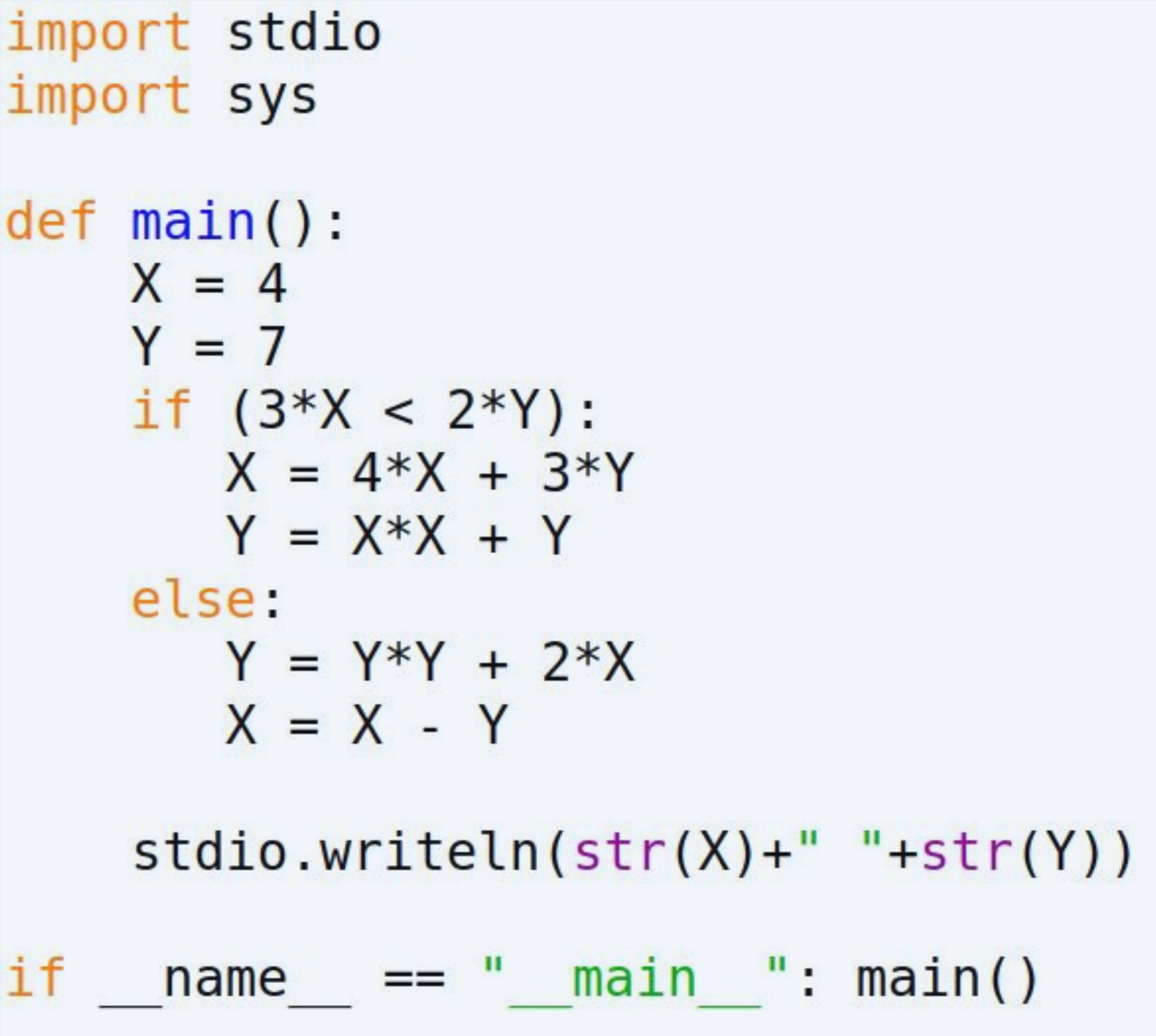
Answer

What does the following program print to the terminal?

```
X = 5
Y = 7
(3*5)<(2*7) => false
Y = 7*7+10 = 59
X = 5 - 59 = -54
-54 59
```

Exercise: If Example 2

What does the following program print to the terminal?



```
import stdio
import sys

def main():
    X = 4
    Y = 7
    if (3*X < 2*Y):
       X = 4*X + 3*Y
       Y = X*X + Y
    else:
       Y = Y*Y + 2*X
       X = X - Y

    stdio.writeln(str(X)+" "+str(Y))

if __name__ == "__main__": main()
```

Answer

What does the following program print to the terminal?

```
X = 4
Y = 7
(3*4)<(2*7) => true
X = 4*4 + 3*7 = 37
Y = 37*37+7 = 1376
37 1376
```

Example of `if` statement use: error checks

```
% python intops.py 5 2
5 + 2 = 7
5 * 2 = 10
5 / 2 = 2
5 % 2 = 1

% python IntOps.py 5 0
5 + 0 = 5
5 * 0 = 0
Division by zero
Division by zero
```

```
import stdio
def main():
    a = int(sys.argv[1])
    b = int(sys.argv[2])
    summ = a + b
    prod = a * b
    stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(summ) )
    stdio.writeln(str(a) + ' * ' + str(b) + ' = ' + str(prod) )
    if (b == 0): stdio.writeln('Division by zero');
    else:        stdio.writeln(str(a) + ' / ' + str(b) + ' = ' + str(a / b) )
    if (b == 0): stdio.writeln("Division by zero");
    else:        stdio.writeln(str(a) + ' % ' + str(b) + ' = ' + str(a % b) )
```

Good programming practice. Use conditionals to check for *and avoid* runtime errors.

INTRO TO PROGRAMMING

IN PYTHON

SEDGEWICK · WAYNE · DONDERO

2. Conditionals & Loops

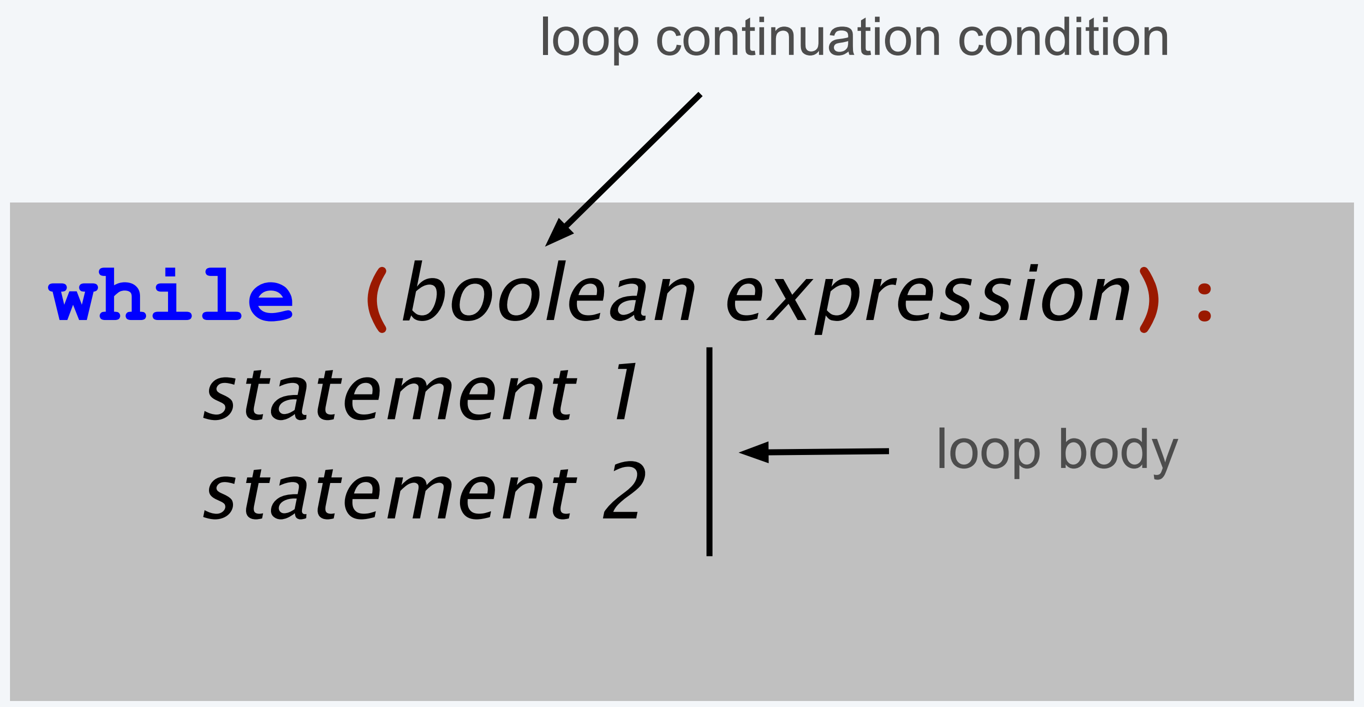
- Conditionals: the `if` statement
- Loops: the `while` statement
- An alternative: the `for` loop
- `Do-While` loop
- Nesting
- Debugging

The while Loop

The while loop

The while loop. Execute certain statements repeatedly until certain conditions are met.

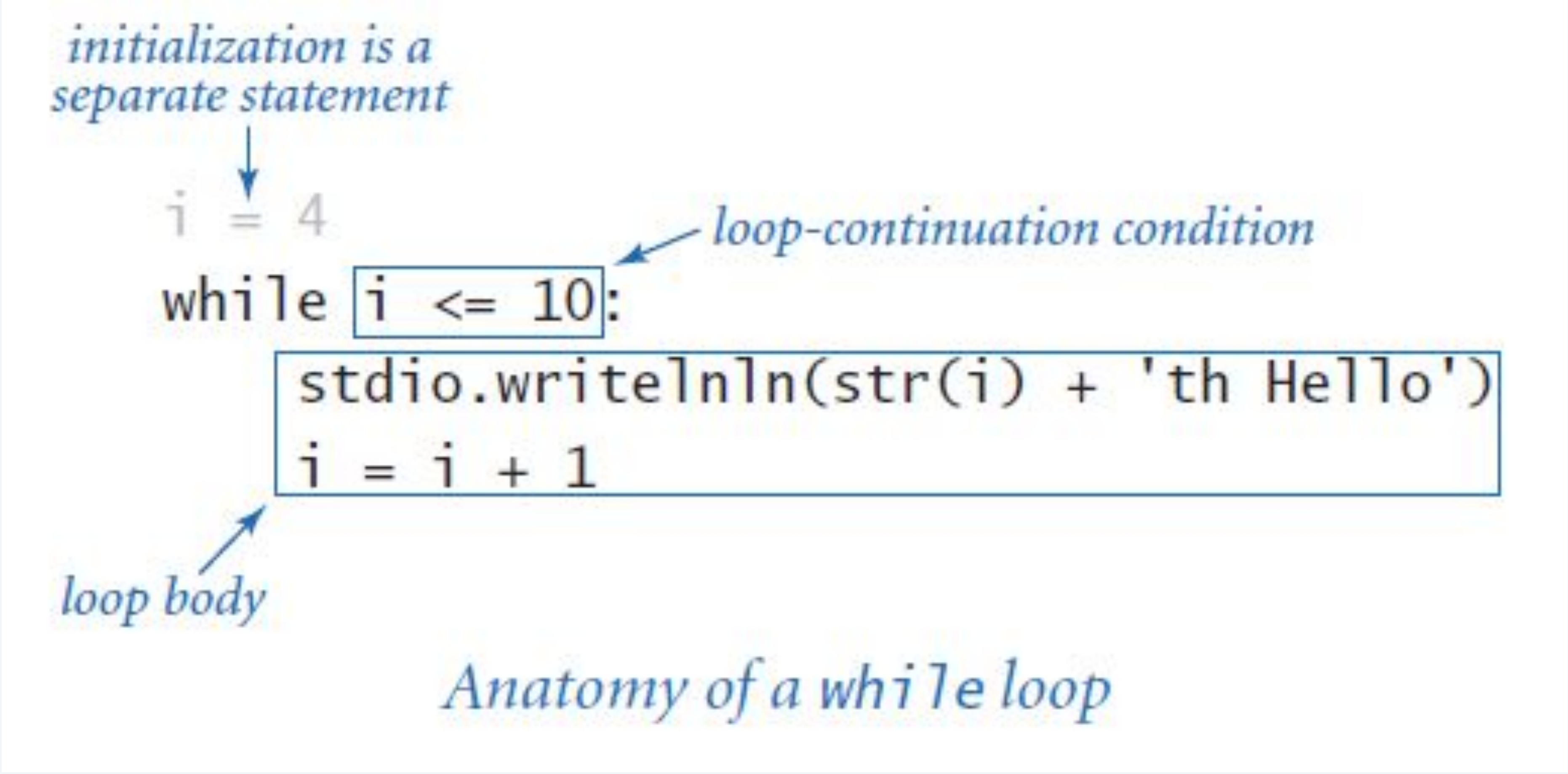
- Evaluate a `boolean` expression.
- If `true`, execute a sequence of statements.
- Repeat.



loop continuation condition

```
while (boolean expression):
    statement 1
    statement 2
```

loop body



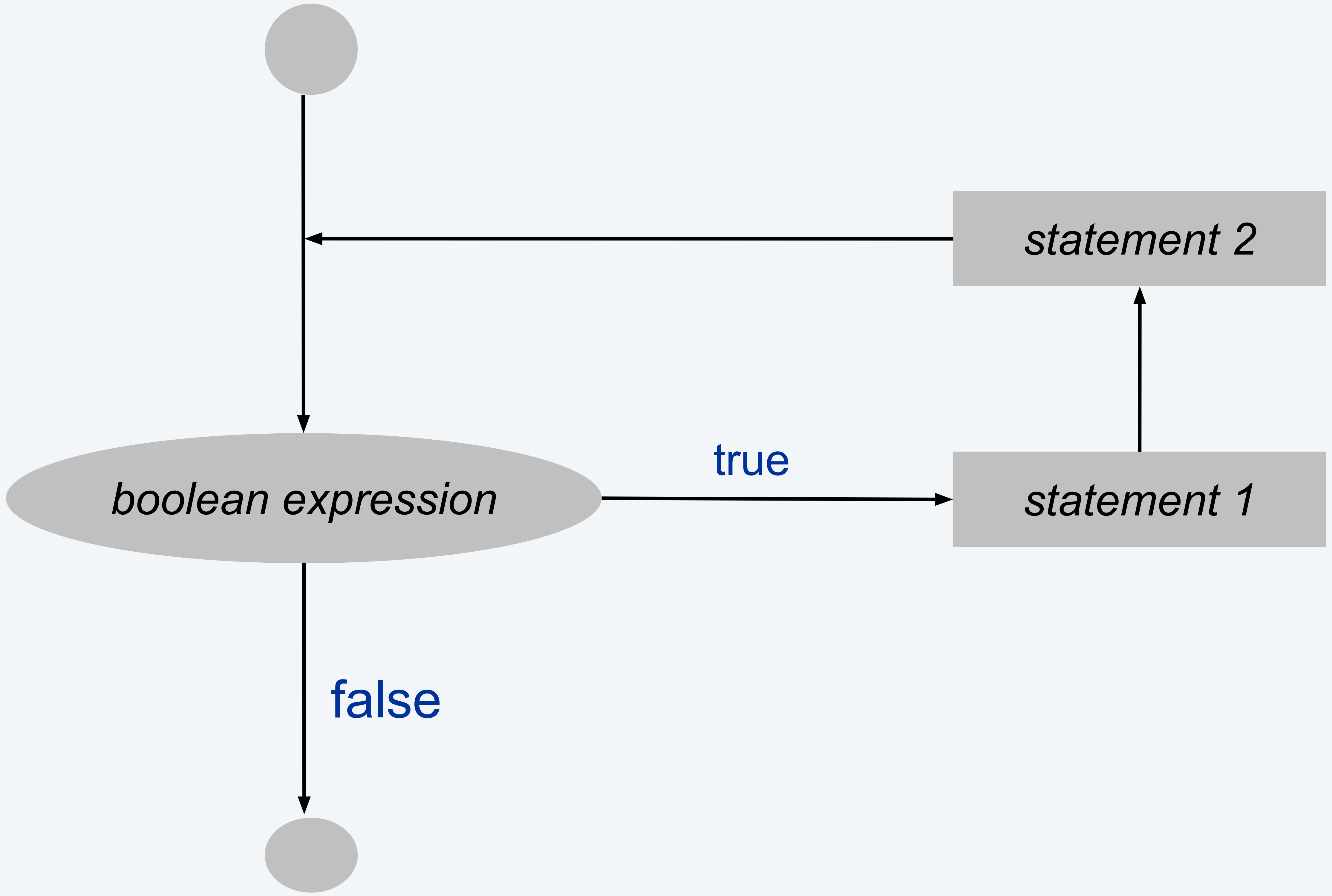
initialization is a separate statement

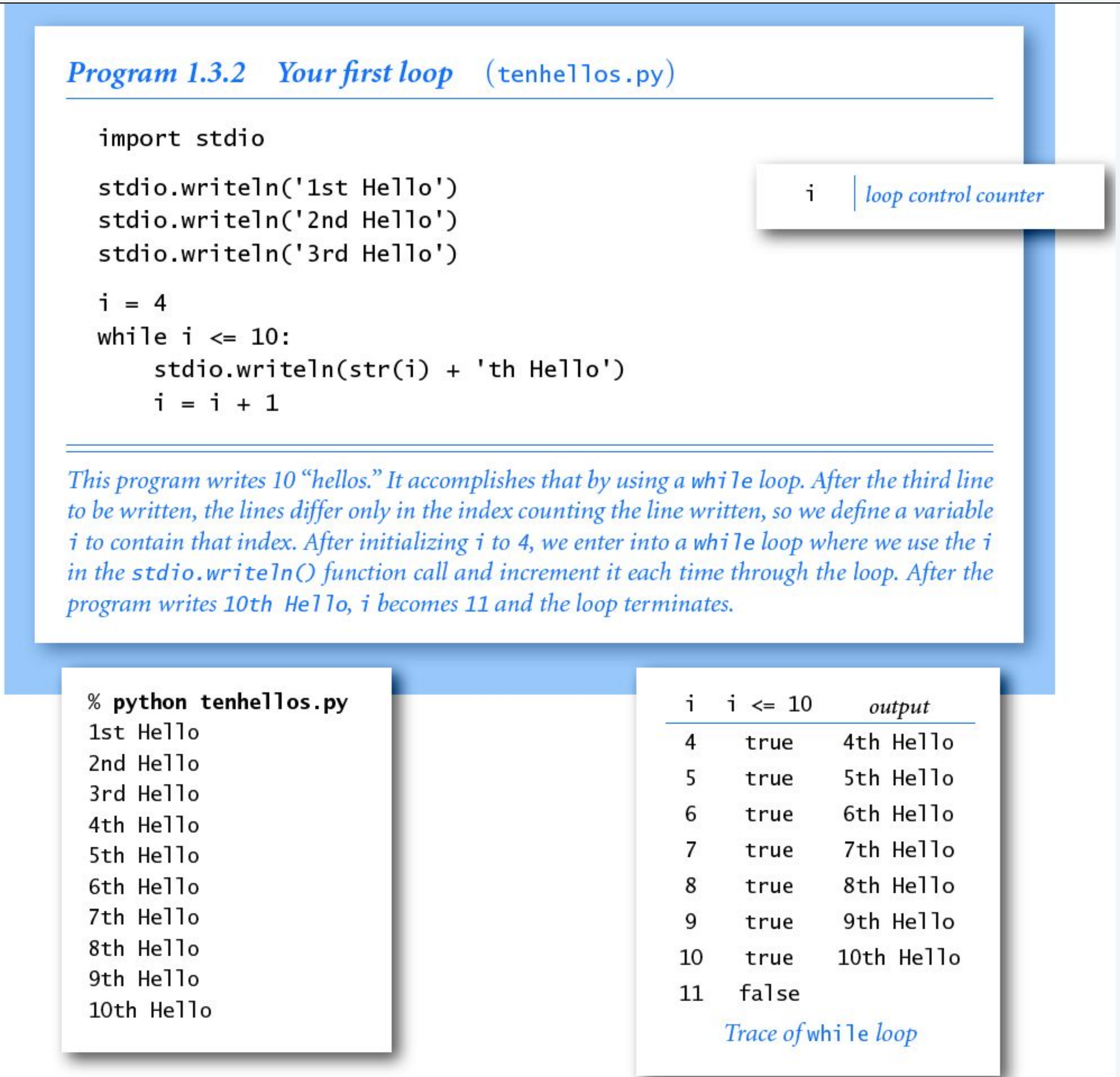
```
i = 4
while i <= 10:
    stdio.writeln(str(i) + 'th Hello')
    i = i + 1
```

loop-continuation condition

loop body

Anatomy of a while loop





Program 1.3.2 Your first loop (tenhellos.py)

```
import stdio

stdio.writeln('1st Hello')
stdio.writeln('2nd Hello')
stdio.writeln('3rd Hello')

i = 4
while i <= 10:
    stdio.writeln(str(i) + 'th Hello')
    i = i + 1
```

i	*loop control counter*

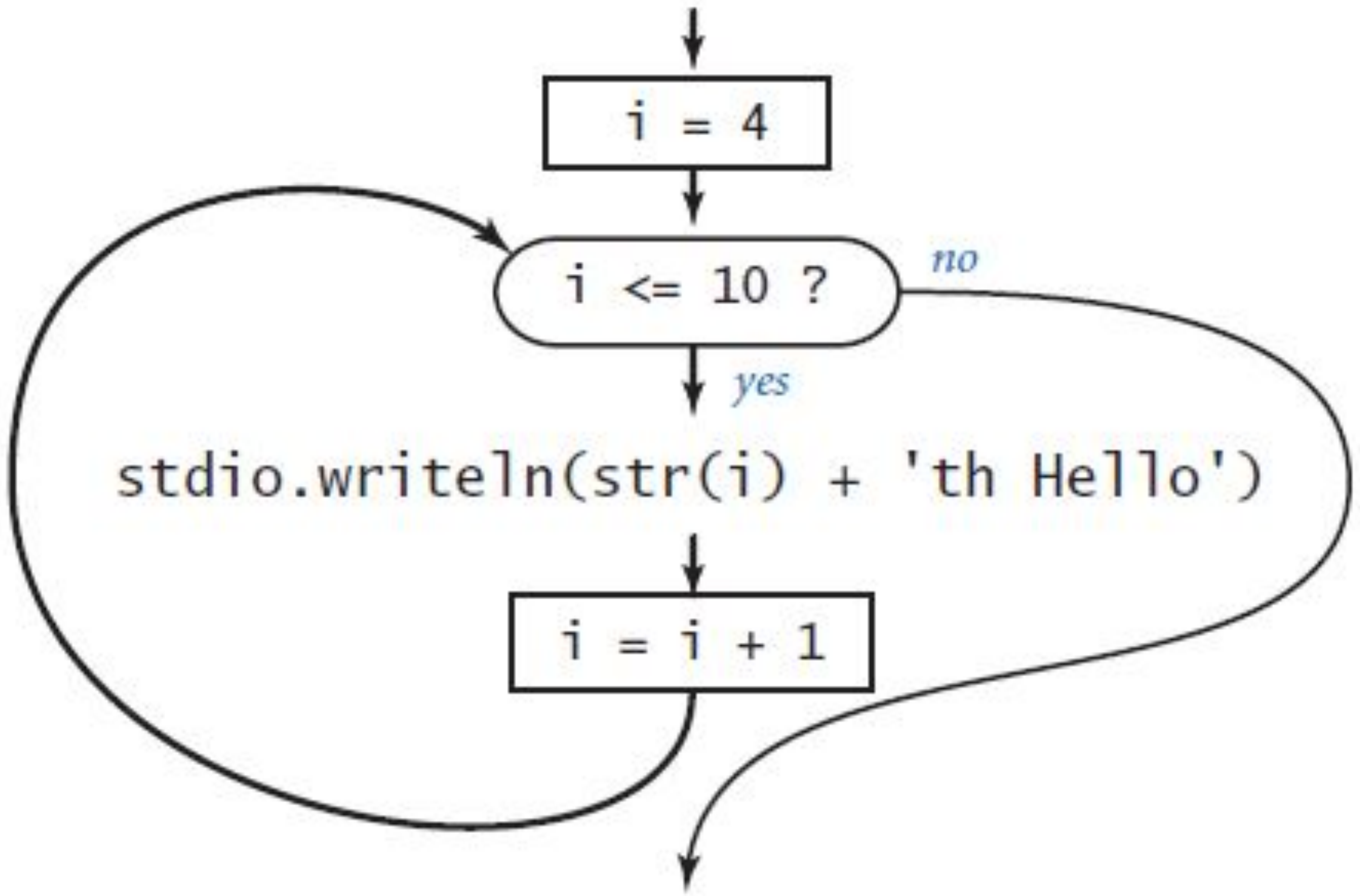
This program writes 10 "hellos." It accomplishes that by using a `while` *loop. After the third line to be written, the lines differ only in the index counting the line written, so we define a variable* `i` *to contain that index. After initializing* `i` *to 4, we enter into a* `while` *loop where we use the* `i` *in the* `stdio.writeln()` *function call and increment it each time through the loop. After the program writes* `10th Hello`*,* `i` *becomes* `11` *and the loop terminates.*

```
% python tenhellos.py
1st Hello
2nd Hello
3rd Hello
4th Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
```

i	i <= 10	*output*
4	true	4th Hello
5	true	5th Hello
6	true	6th Hello
7	true	7th Hello
8	true	8th Hello
9	true	9th Hello
10	true	10th Hello
11	false	

Trace of `while` *loop*

```
i = 4
while i <= 10:
    stdio.writelnln(str(i) + 'th Hello')
    i = i + 1
```



Flowchart example (`while` statement)

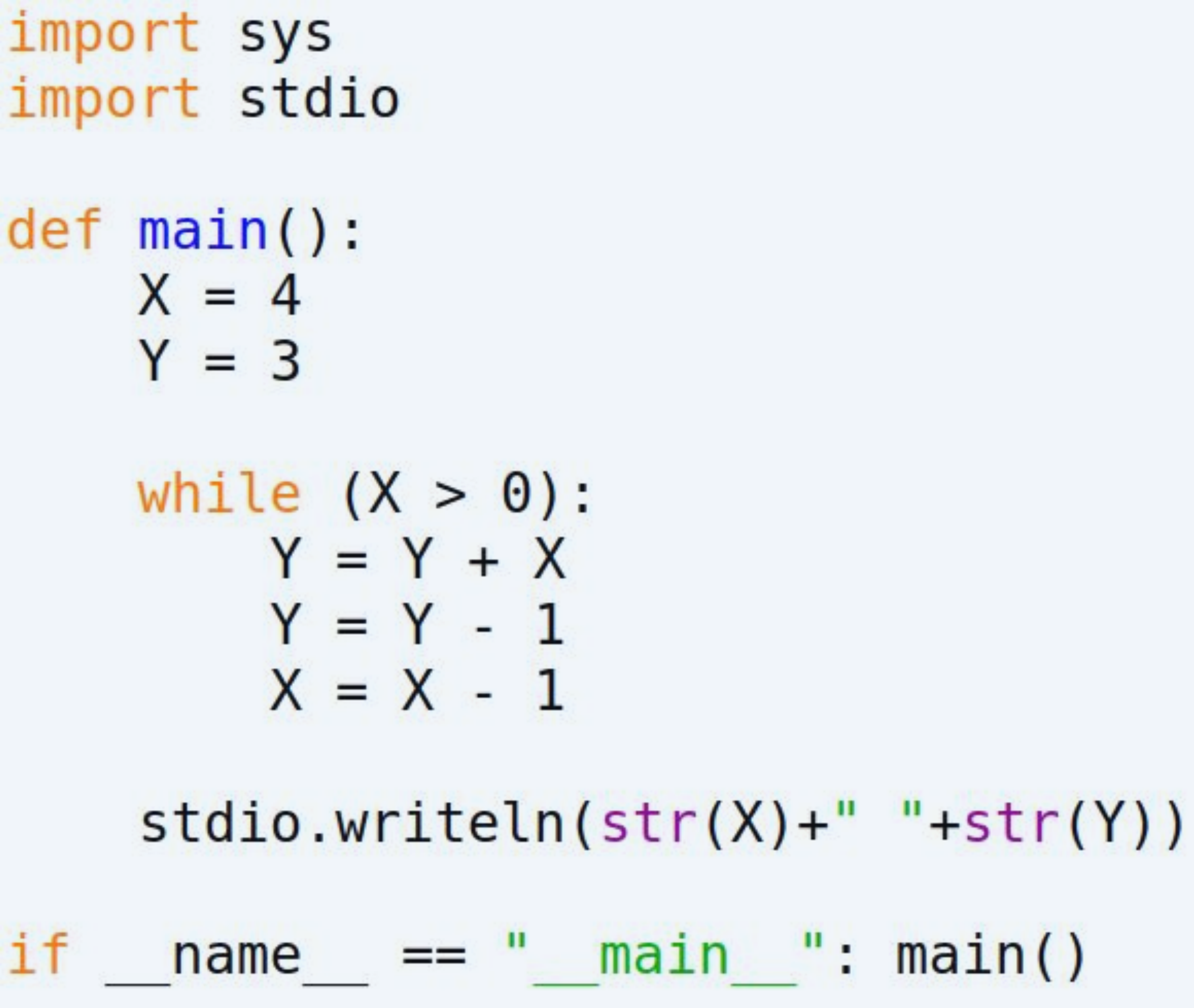
i	i <= 10	*output*
4	true	4th Hello
5	true	5th Hello
6	true	6th Hello
7	true	7th Hello
8	true	8th Hello
9	true	9th Hello
10	true	10th Hello
11	false	

Trace of `while` loop

```
% python tenhellos.py
1st Hello
2nd Hello
3rd Hello
4th Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
```

Exercise: While Example

What does the following program print to the terminal?



```python
import sys
import stdio

def main():
    X = 4
    Y = 3

    while (X > 0):
        Y = Y + X
        Y = Y - 1
        X = X - 1

    stdio.writeln(str(X)+" "+str(Y))

if __name__ == "__main__": main()
```

Answer

What does the following program print to the terminal?

```
X = 4, Y = 3
************
 Y = 3+4-1 = 6
 X = 3
************
X = 3, Y = 6
************
 Y = 6+3-1 = 8
 X = 2
************
X  = 2, Y = 8
************
 Y = 8+2-1 = 9
 X = 1
************
 X = 1, Y = 9
************
  Y = 9+1-1 = 9
  X = 0
************
```

0 9

Example of while loop use: print powers of two

Ex. Print powers of 2 that are ≤ 2N.

- Increment i from 0 to N.
- Double v each time.

```
def main():
    n = int(sys.argv[1])
    i = 0
    v = 1
    while (i <= n):
        stdio.writeln(v)
        i = i + 1
        v = 2 * v
```

i	v	i <= n
0	1	true
1	2	true
2	4	true
3	8	true
4	16	true
5	32	true
6	64	true
7	128	false

```
% python PowersOfTwo.py 6
1
2
4
8
16
32
64
```

Pop quiz on while loops

Q. Anything wrong with the following code?

```
import stdio
import sys

def main():
    n = int(sys.argv[1])
    i = 0
    v = 1
    while (i <= n):
        stdio.writeln(v)
    i = i + 1
    v = 2 * v

if __name__ == '__main__': main()
```

Pop quiz on `while` loops

Q. Anything wrong with the following code?

```
import stdio
import sys

def main():
    n = int(sys.argv[1])
    i = 0
    v = 1
    while (i <= n):
        stdio.writeln(v)
    ....i = i + 1
    ....v = 2 * v

if __name__ == '__main__': main()
```

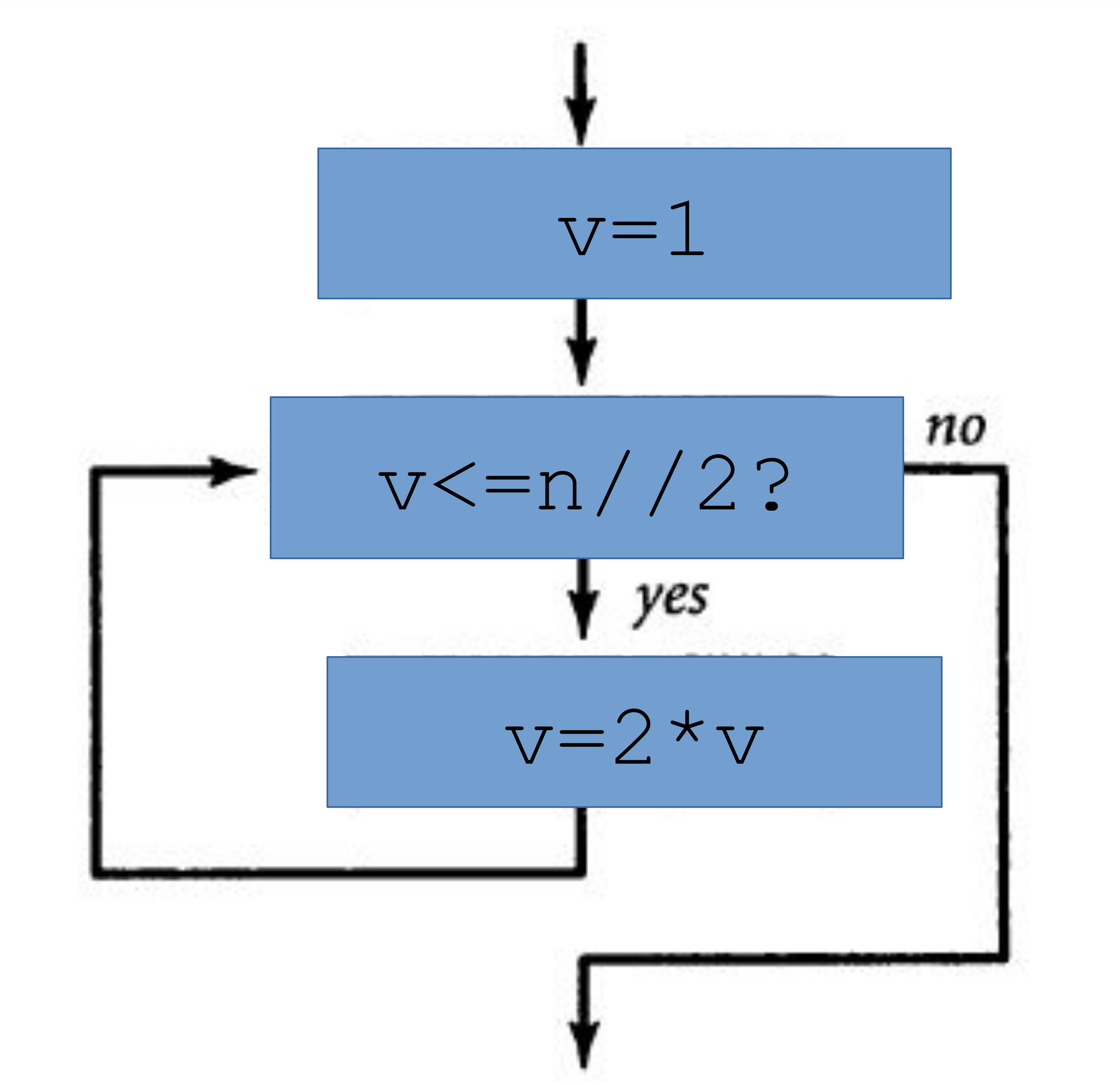
A. Yes! Needs proper indentation.

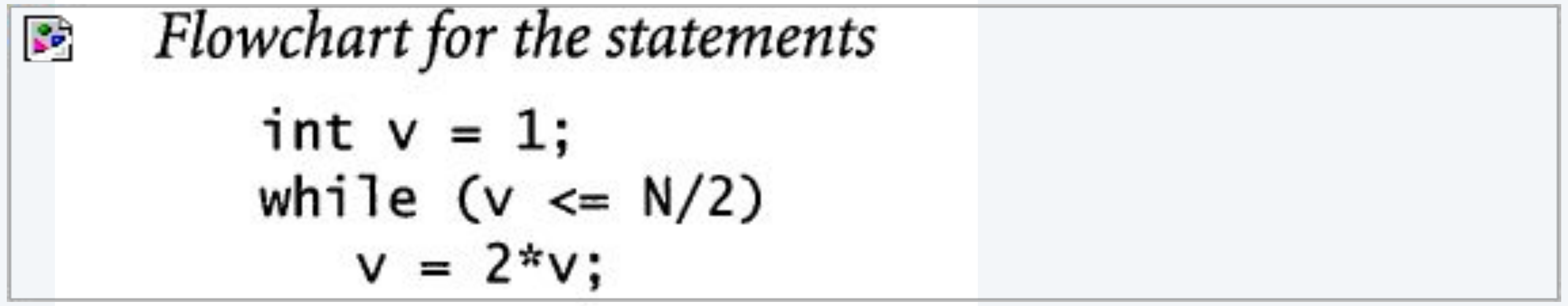
Q. What does it do without the increment?

A. Goes into an *infinite loop*.

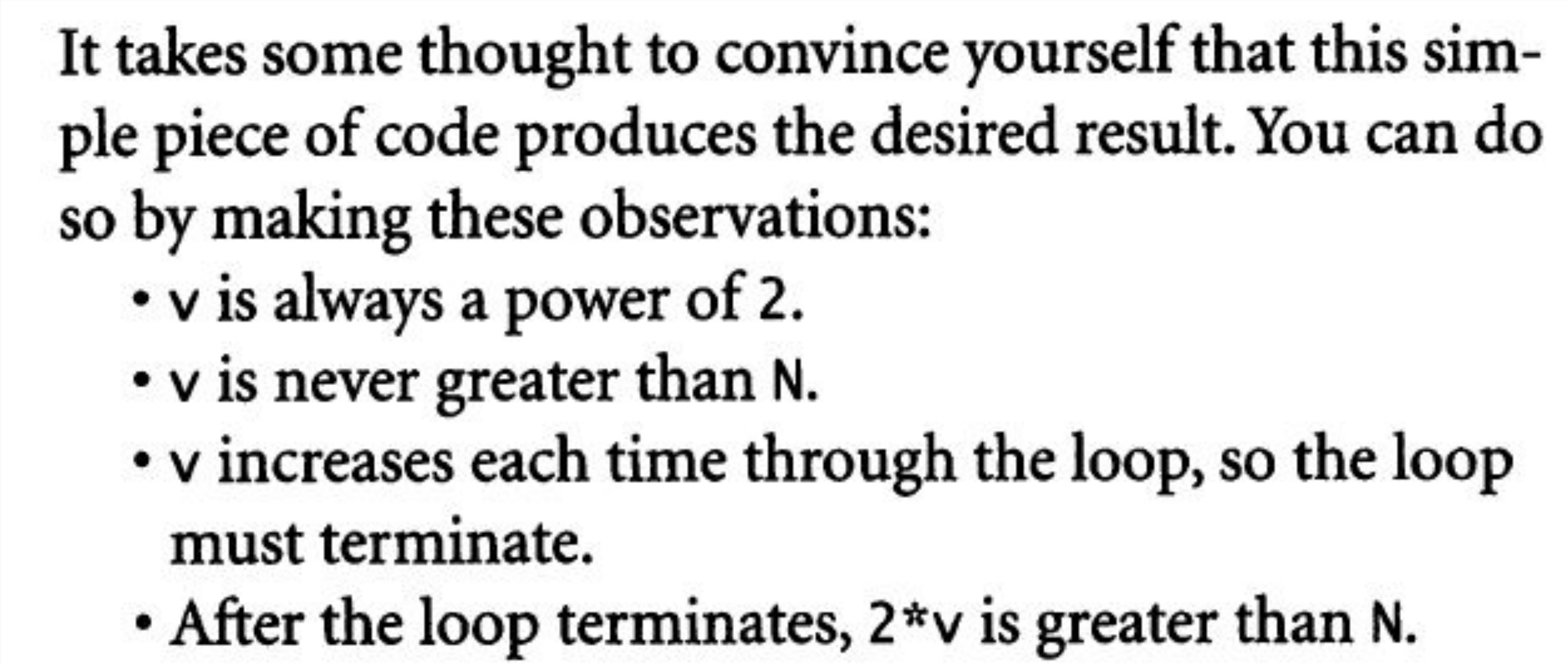
```
% python pqwhile.py 6
1
1
1
1
1
...
```

Another While Loop




Flowchart for the statements

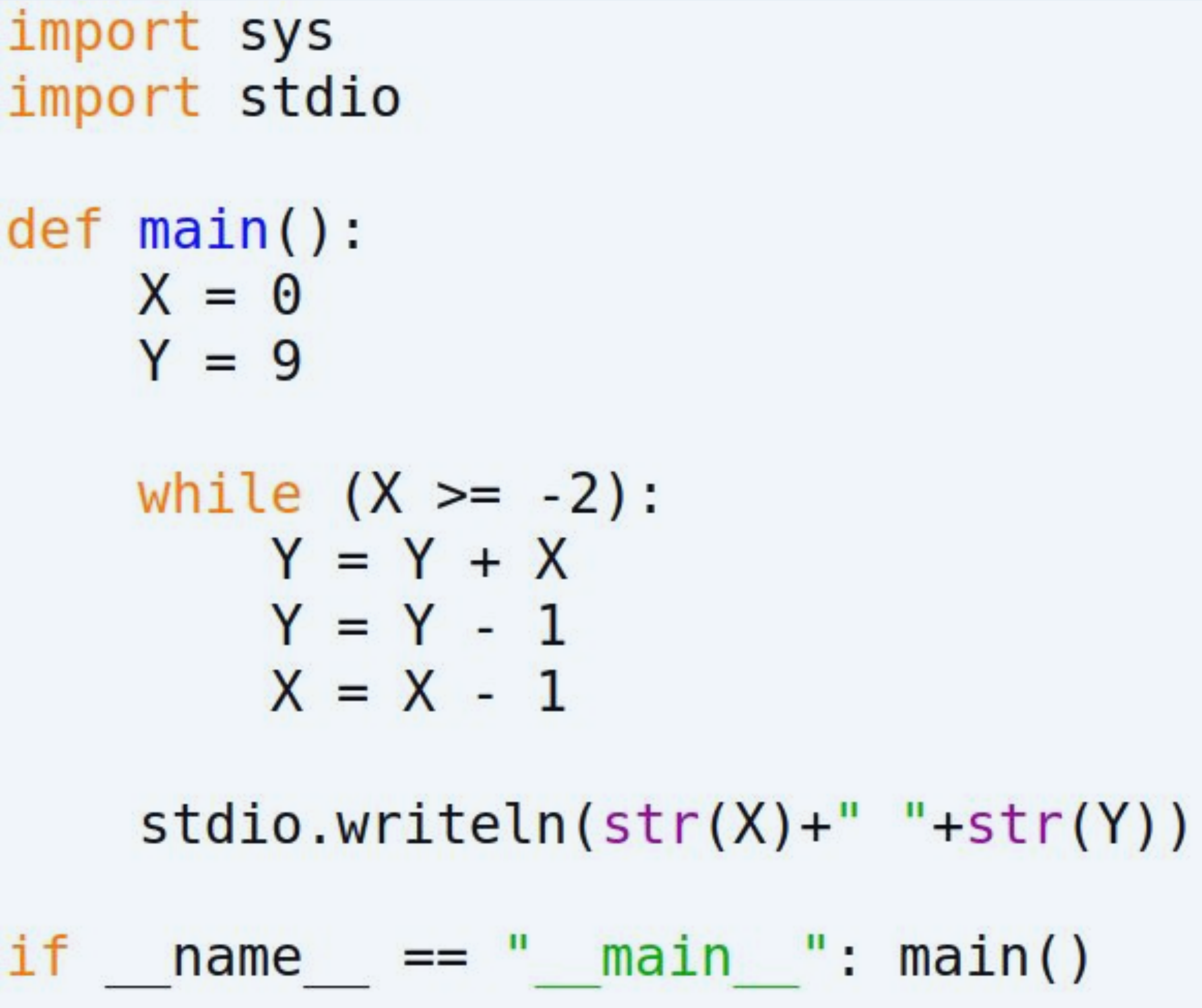
```
int v = 1;
while (v <= N/2)
   v = 2*v;
```


It takes some thought to convince yourself that this simple piece of code produces the desired result. You can do so by making these observations:

- v is always a power of 2.
- v is never greater than N.
- v increases each time through the loop, so the loop must terminate.
- After the loop terminates, 2*v is greater than N.

Exercise: While Example 2

What does the following program print to the terminal?



```
import sys
import stdio

def main():
    X = 0
    Y = 9

    while (X >= -2):
        Y = Y + X
        Y = Y - 1
        X = X - 1

    stdio.writeln(str(X)+" "+str(Y))

if __name__ == "__main__": main()
```

Answer

What does the following program print to the terminal?

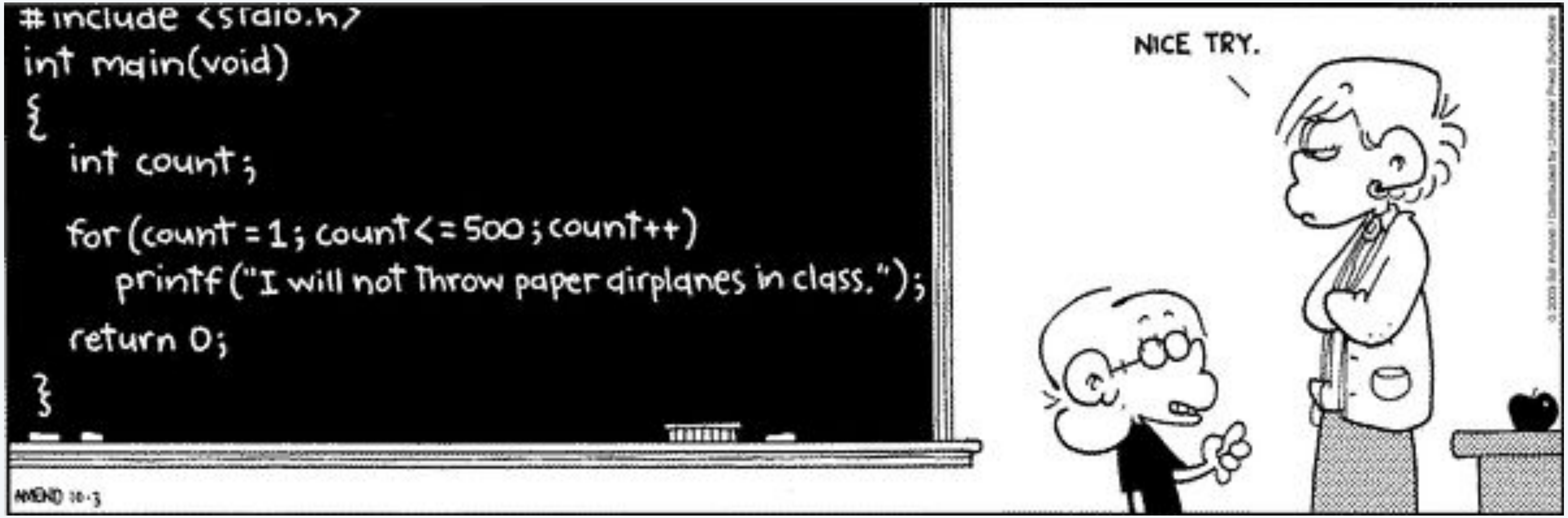
```
X = 0, Y = 9
************
 Y = 9+0-1 = 8
 X = -1
************
X = -1, Y = 8
************
 Y = 8-1-1 = 6
 X = -2
************
X  = -2, Y = 6
************
 Y = 6-2-1 = 3
 X = -3
************
```

-3 3

2. Conditionals & Loops

- Conditionals: the `if` statement
- Loops: the `while` statement
- An alternative: the `for` loop
- `Do-While` loop
- Nesting
- Debugging

The For Loop



```
#include <stdio.h>
int main(void)
{
  int count;

  for(count=1;count<=500;count++)
    printf("I will not Throw paper airplanes in class.");
  return 0;
}
```

Copyright 2004, FoxTrot by Bill Amend

www.ucomics.com/foxtrot/2003/10/03

The for loop

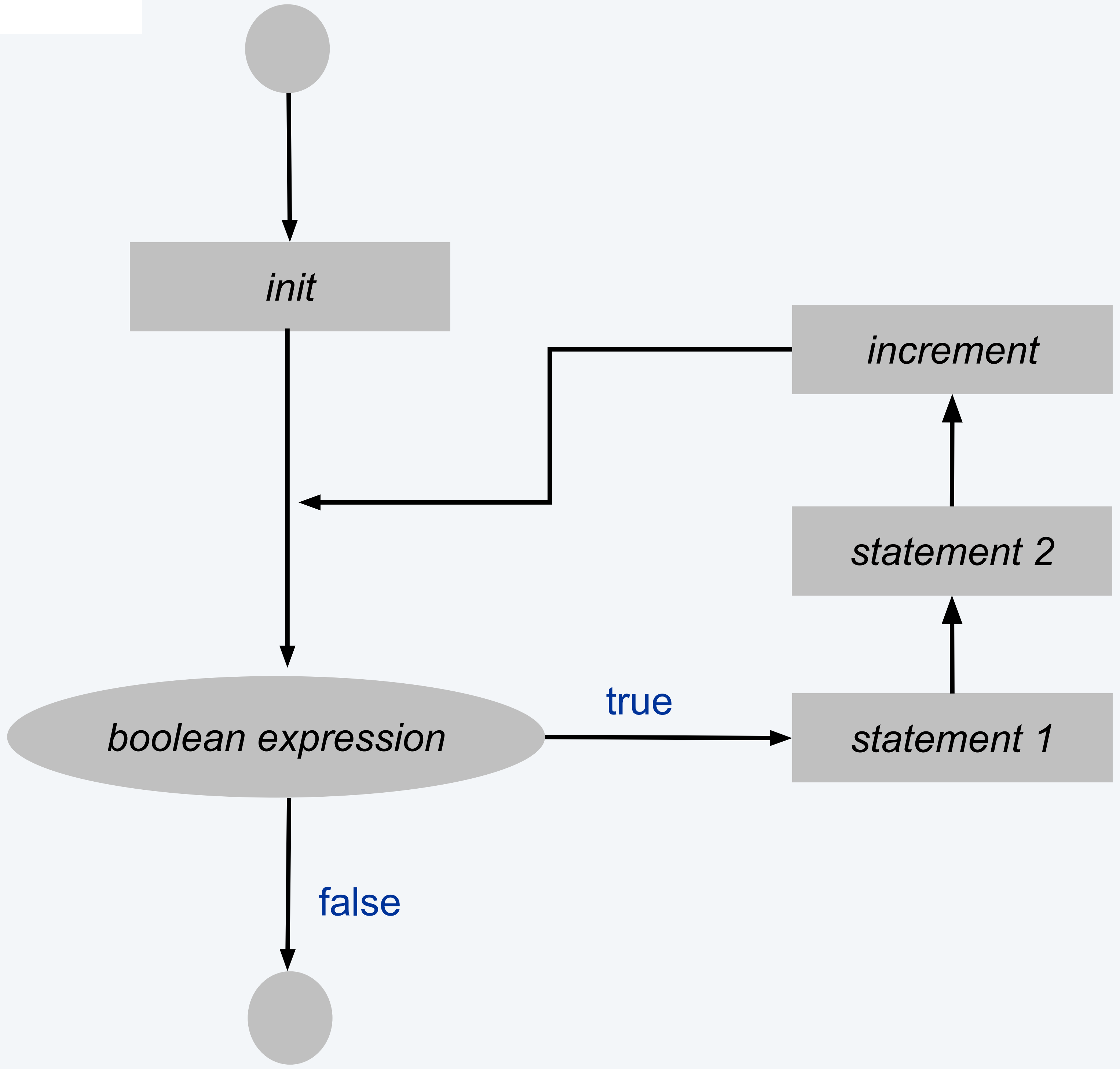
The for loop. An alternative repetition structure.

- Execute an *initialization statement*.
- Evaluate a `boolean` *expression*.
- If true, execute a *sequence of statements*,
- then execute an increment statement.
- Repeat.

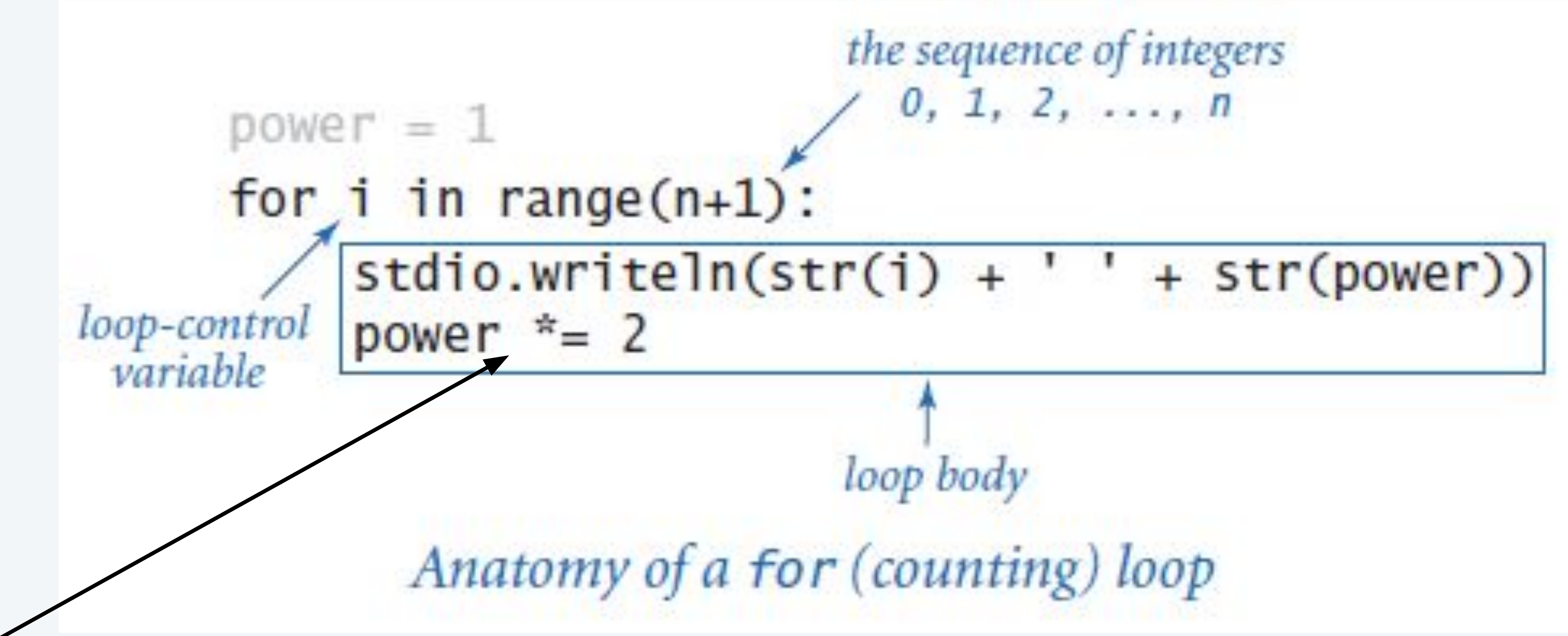
loop continuation condition

```
for <variable> in range(<start>,<stop>):
   <statement>
   <statement>
```

body



Anatomy of a For Loop



```
power = 1
for i in range(n+1):
    stdio.writeln(str(i) + ' ' + str(power))
    power *= 2
```

Anatomy of a `for` *(counting) loop*

compund assignment idiom
power *=2 => power = power *2

Q. What does it print?

A.

```
0 1
1 2
2 4
3 8
...
```

For Loops: Subdivisions of a Ruler

Create subdivisions of a ruler to 1/*N* inches.

- Initialize `ruler` to one space.
- For each value `i` from `1` to `N`:
  sandwich `i` between two copies of `ruler`.

```
import stdio

def main():
    N = int(sys.argv[1])
    ruler = ' ';
    for i in range(1, N+1):
        ruler = ruler + str(i) + ruler
    stdio.writeln(ruler)

if __name__ == '__main__': main()
```

Note: Small program can produce huge amount of output.

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

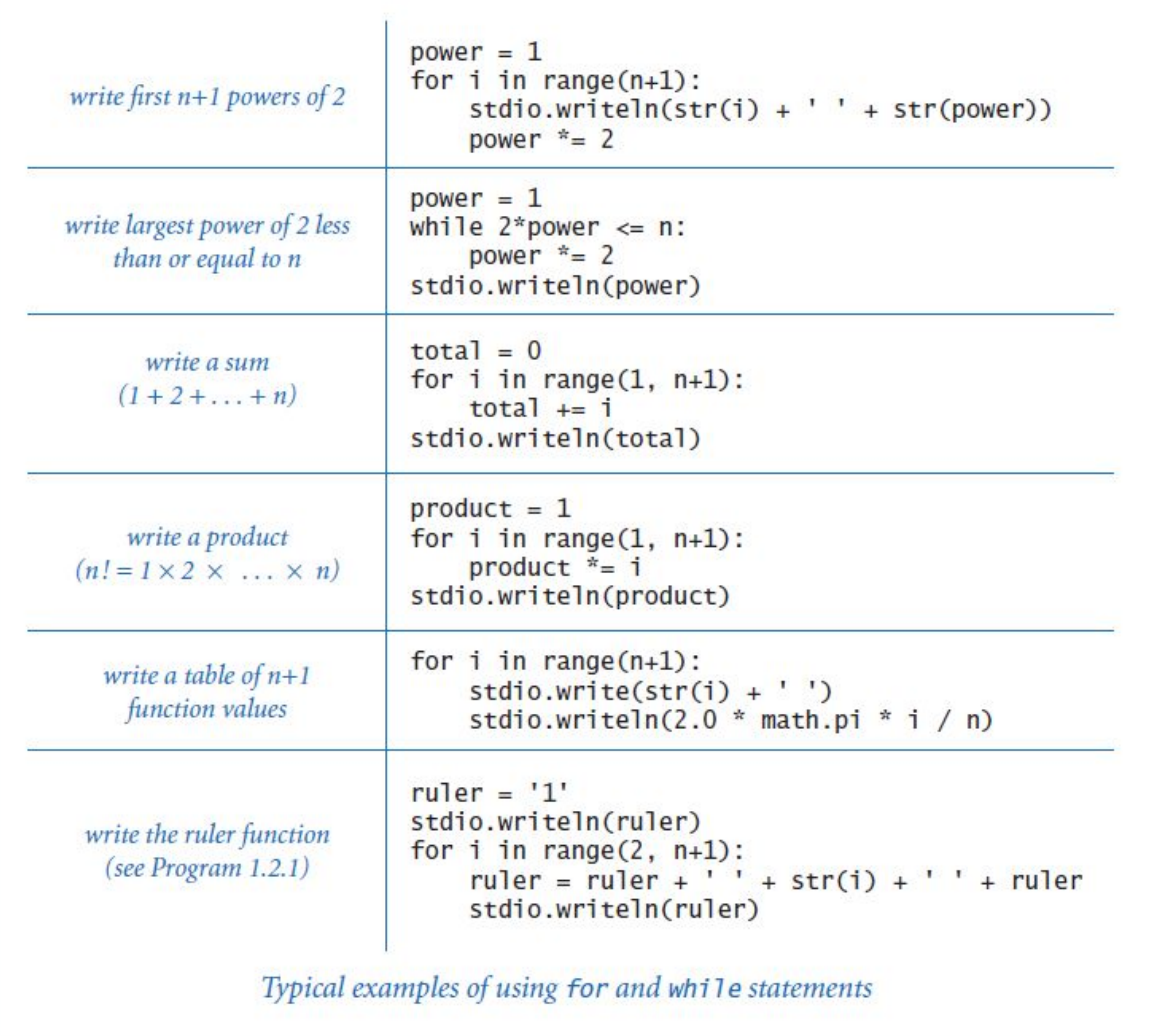
i	ruler
1	" 1 "
2	" 1 2 1 "
3	" 1 2 1 3 1 2 1 "
4	" 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 "

End-of-loop trace

```
python Ruler.py 4
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

```
% DO NOT RUN
% python Ruler.py 100
(will never finish)
```

Loop Examples



write first n+1 powers of 2	`power = 1`
	`for i in range(n+1):`
	`    stdio.writeln(str(i) + ' ' + str(power))`
	`    power *= 2`

write largest power of 2 less than or equal to n	`power = 1`
	`while 2*power <= n:`
	`    power *= 2`
	`stdio.writeln(power)`

write a sum
(1 + 2 + . . . + n)
	`total = 0`
	`for i in range(1, n+1):`
	`    total += i`
	`stdio.writeln(total)`

write a product
(n! = 1 × 2 × . . . × n)
	`product = 1`
	`for i in range(1, n+1):`
	`    product *= i`
	`stdio.writeln(product)`

write a table of n+1 function values	`for i in range(n+1):`
	`    stdio.write(str(i) + ' ')`
	`    stdio.writeln(2.0 * math.pi * i / n)`

write the ruler function
(see Program 1.2.1)
	`ruler = '1'`
	`stdio.writeln(ruler)`
	`for i in range(2, n+1):`
	`    ruler = ruler + ' ' + str(i) + ' ' + ruler`
	`    stdio.writeln(ruler)`

Typical examples of using `for` *and* `while` *statements*

Exercise: for loop

Use a for loop to compute 1 + 3 + 5+ … + *N.* Assume N is odd and has already been defined.

Pop quiz on `for` loops

Q. What does the following program print?

```
def main():
    f = 0
    g = 1
    for i in range(0, 11):
        stdio.writeln(f)
        f = f + g
        g = f - g
```

Pop quiz on `for` loops

Q. What does the following program print?

```
def main():
    f = 0
    g = 1
    for i in range(0, 11):
        stdio.writeln(f)
        f = f + g
        g = f - g
```

A.

Beginning-of-loop trace

i	f	g
0	0	1
1	1	0
2	1	1
3	2	1
4	3	2
5	5	3
6	8	5
7	13	8
8	21	13
9	34	21
10	55	34

values printed

INTRO TO PROGRAMMING

IN PYTHON

SEDGEWICK·WAYNE·DONDERO

2. Conditionals & Loops

- Conditionals: the `if` statement
- Loops: the `while` statement
- An alternative: `Do-While` loop
- `Do-While` loop
- Nesting
- Debugging

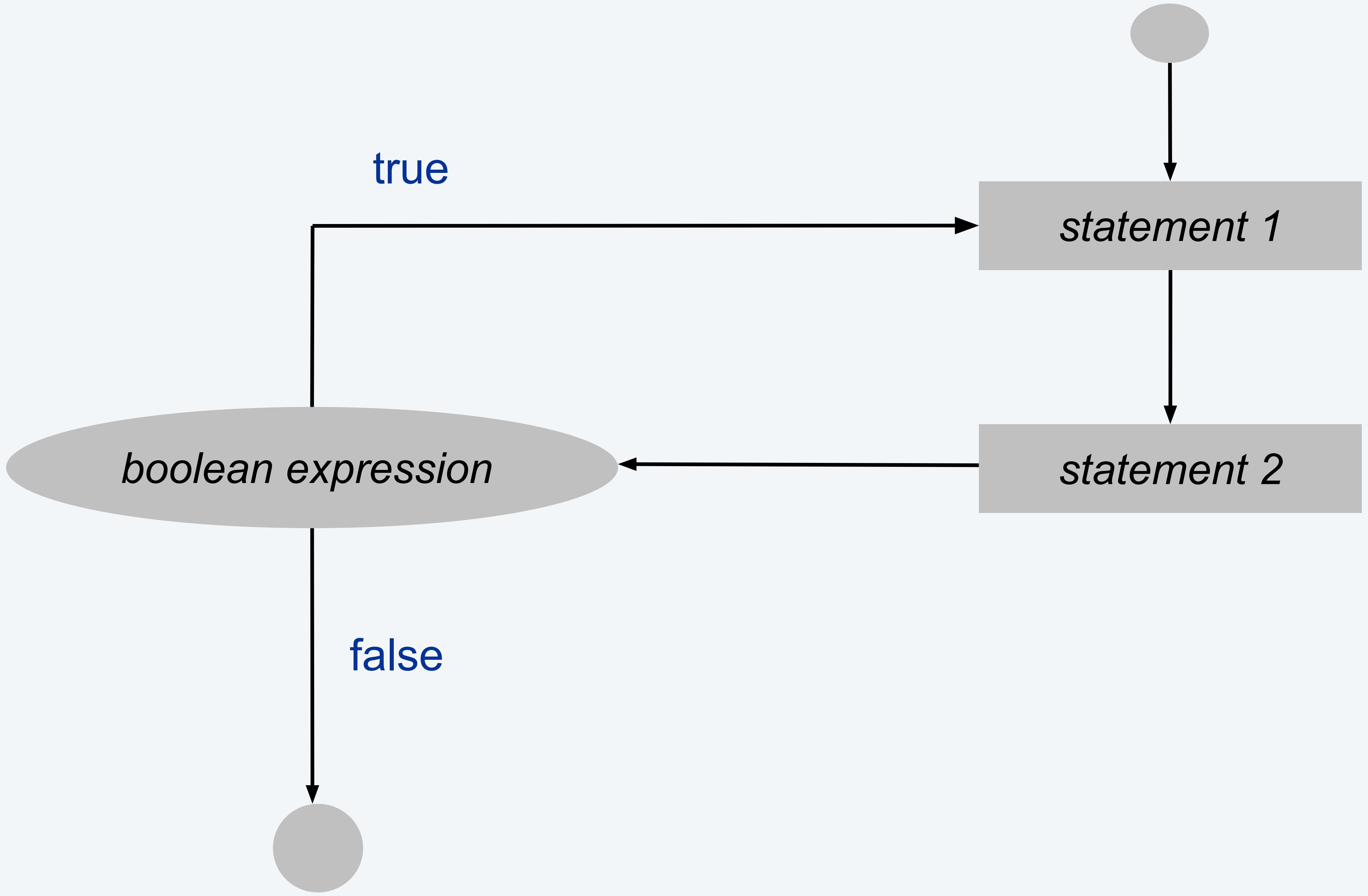
Do-While Loop (loop and a half)

The **do-while** loop. A less common repetition structure.

- Execute sequence of statements.
- Check loop-continuation condition.
- Repeat.

```
while (True):
   statement 1
   Statement 2
   if (boolean expression):
      break
```

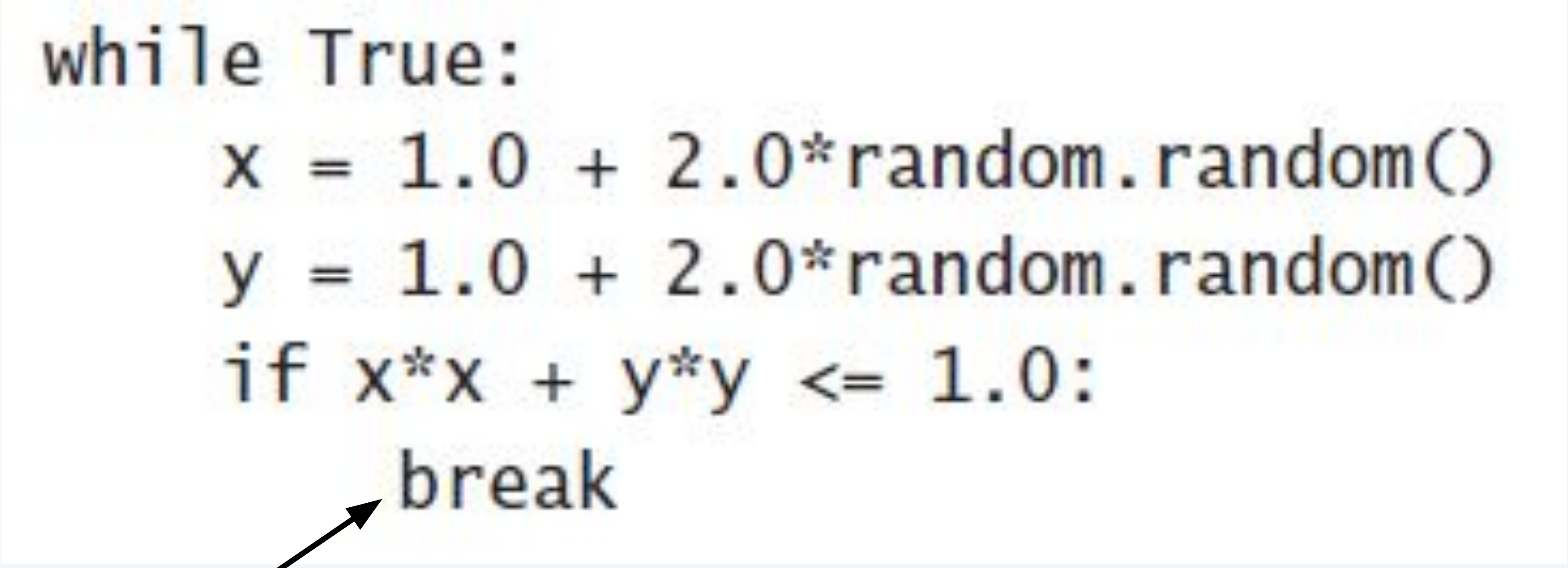
do-while loop syntax



Do-While Loop

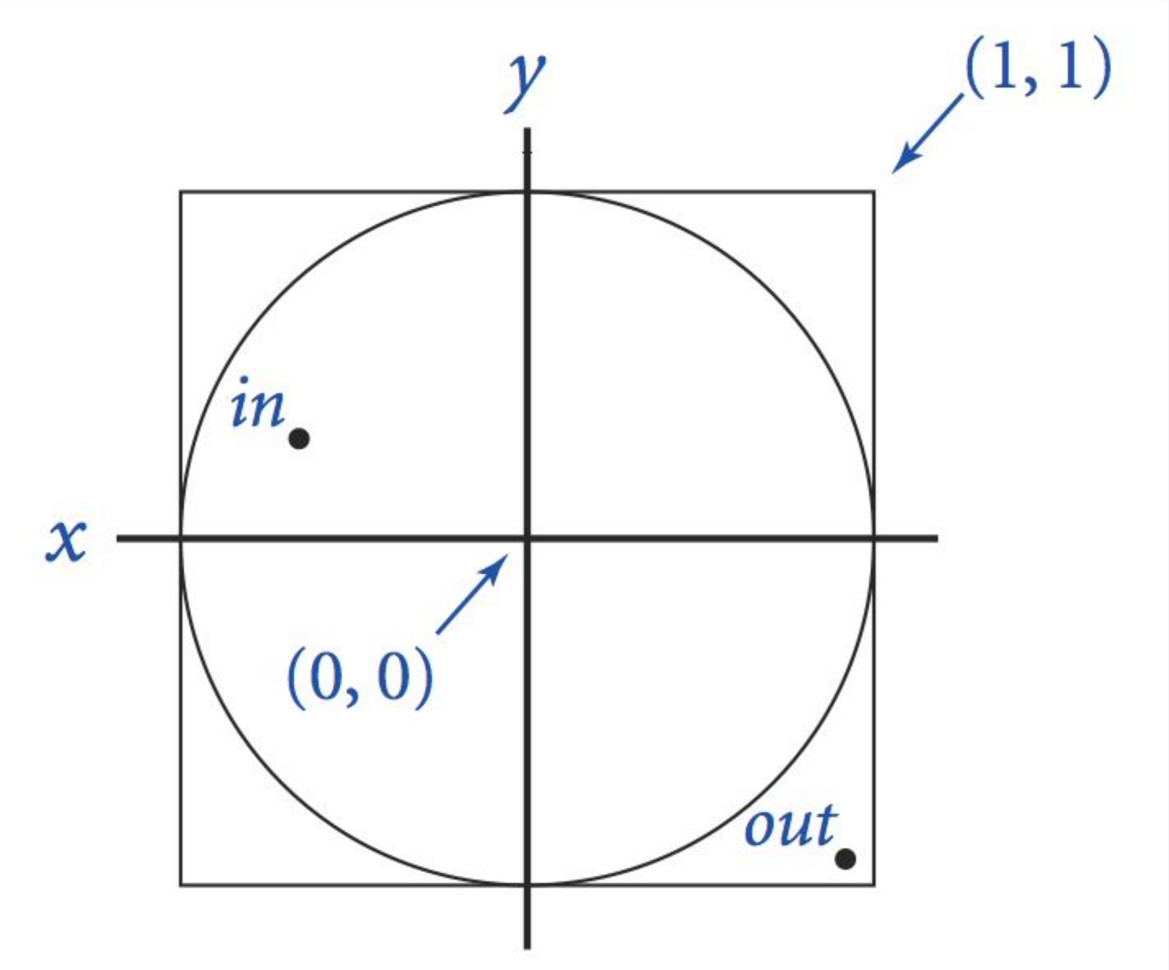
The **do-while** loop. A less common repetition structure.

- Execute sequence of statements.
- Check loop-continuation condition.
- Repeat.

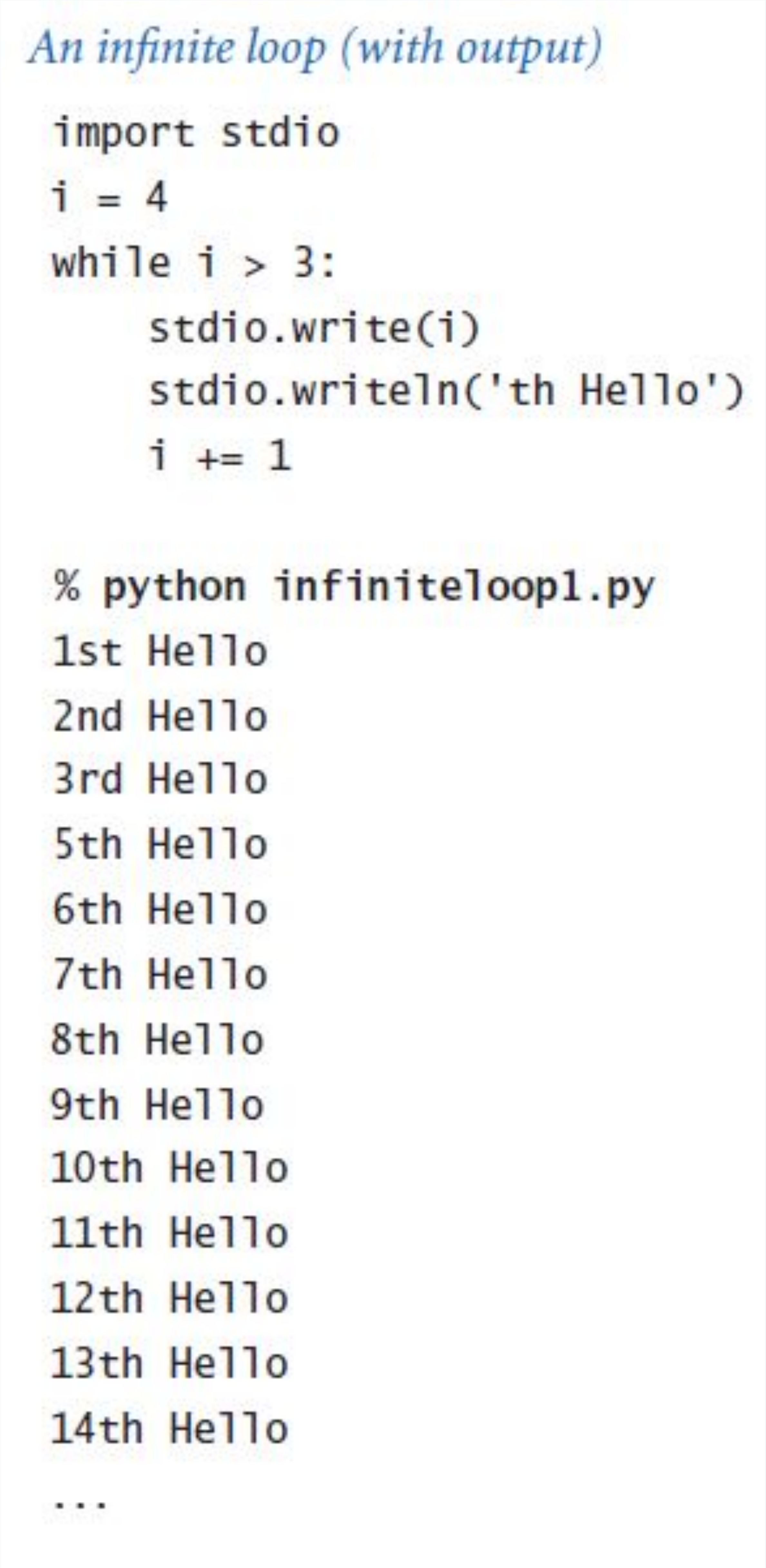


```
while True:
    x = 1.0 + 2.0*random.random()
    y = 1.0 + 2.0*random.random()
    if x*x + y*y <= 1.0:
        break
```

ends loop



Infinite Loop



An infinite loop (with output)

```
import stdio
i = 4
while i > 3:
    stdio.write(i)
    stdio.writeln('th Hello')
    i += 1
```

```
% python infiniteloop1.py
1st Hello
2nd Hello
3rd Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
11th Hello
12th Hello
13th Hello
14th Hello
...
```

<ctrl-c>

2. Conditionals & Loops

- Conditionals: the `if` statement
- Loops: the `while` statement
- An alternative: `Do-While` loop
- `Do-While` loop
- **Nesting**
- Debugging

Nesting

Nesting Conditionals and Loops

Conditionals enable you to do one of 2^n sequences of operations with n lines.

```
if (a0 > 0): stdio.write(str(0))
if (a1 > 0): stdio.write(str(1))
if (a2 > 0): stdio.write(str(2))
if (a3 > 0): stdio.write(str(3))
if (a4 > 0): stdio.write(str(4))
if (a5 > 0): stdio.write(str(5))
if (a6 > 0): stdio.write(str(6))
if (a7 > 0): stdio.write(str(7))
if (a8 > 0): stdio.write(str(8))
if (a9 > 0): stdio.write(str(9))
```

2^{10} = 1024 possible results, depending on input

Loops enable you to do an operation n times

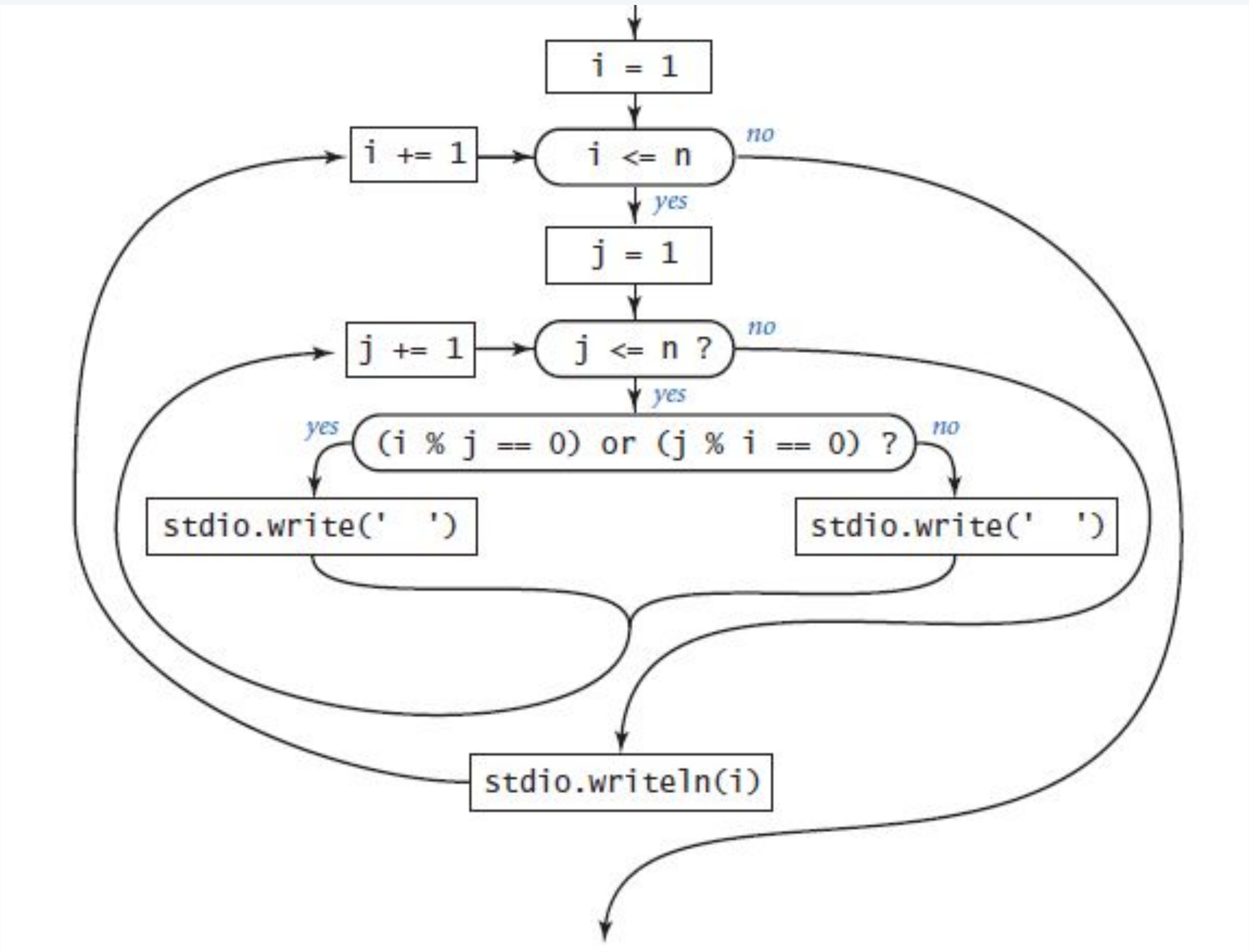
using only 2 lines of code.

```
sum = 0.0
for i in range(1,1025):
    sum = sum + 1.0 / i
```

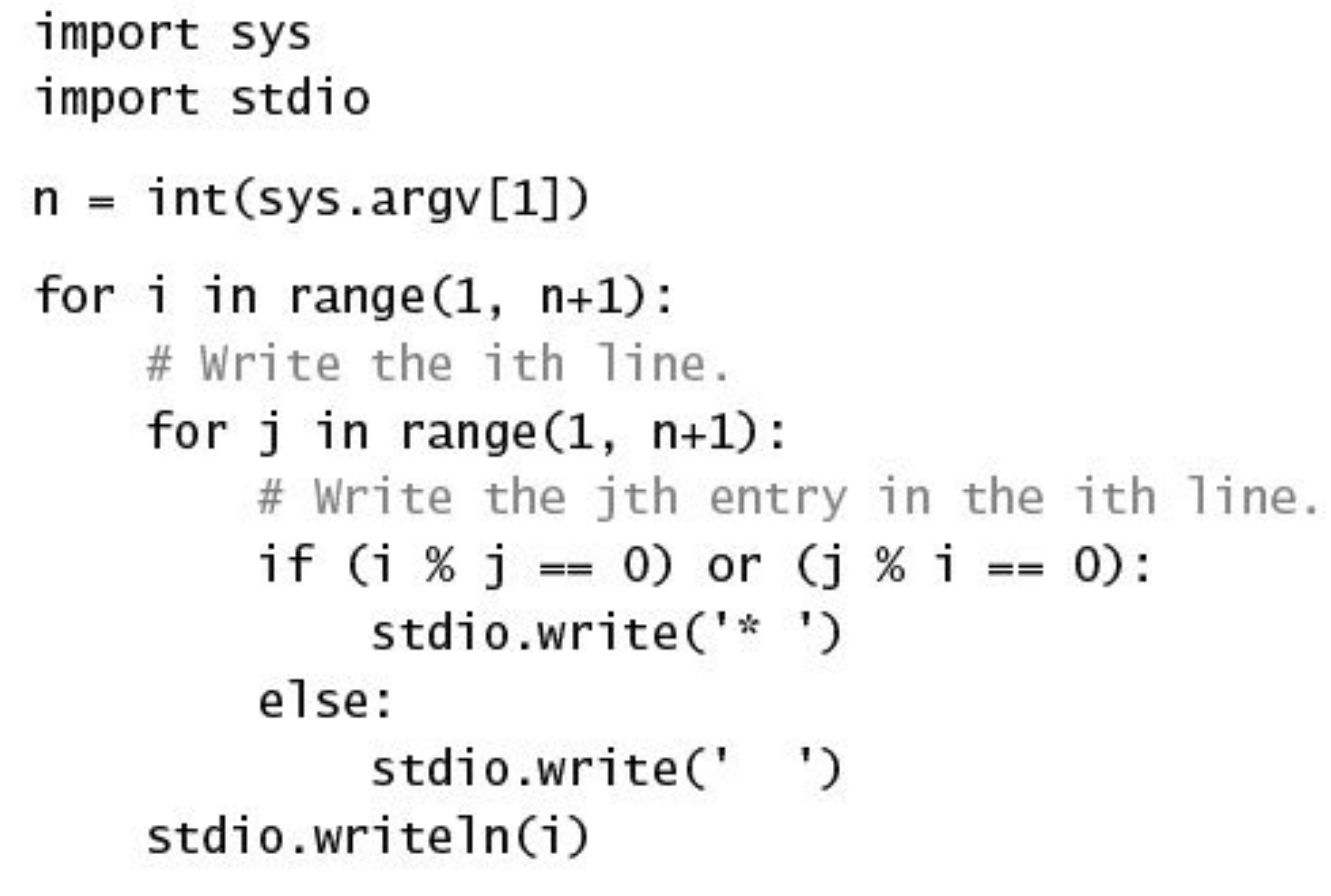
computes 1/1 + 1/2 + ... + 1/1024

More sophisticated programs.

- Nest conditionals within conditionals.
- Nest loops within loops.
- Nest conditionals within loops within loops.

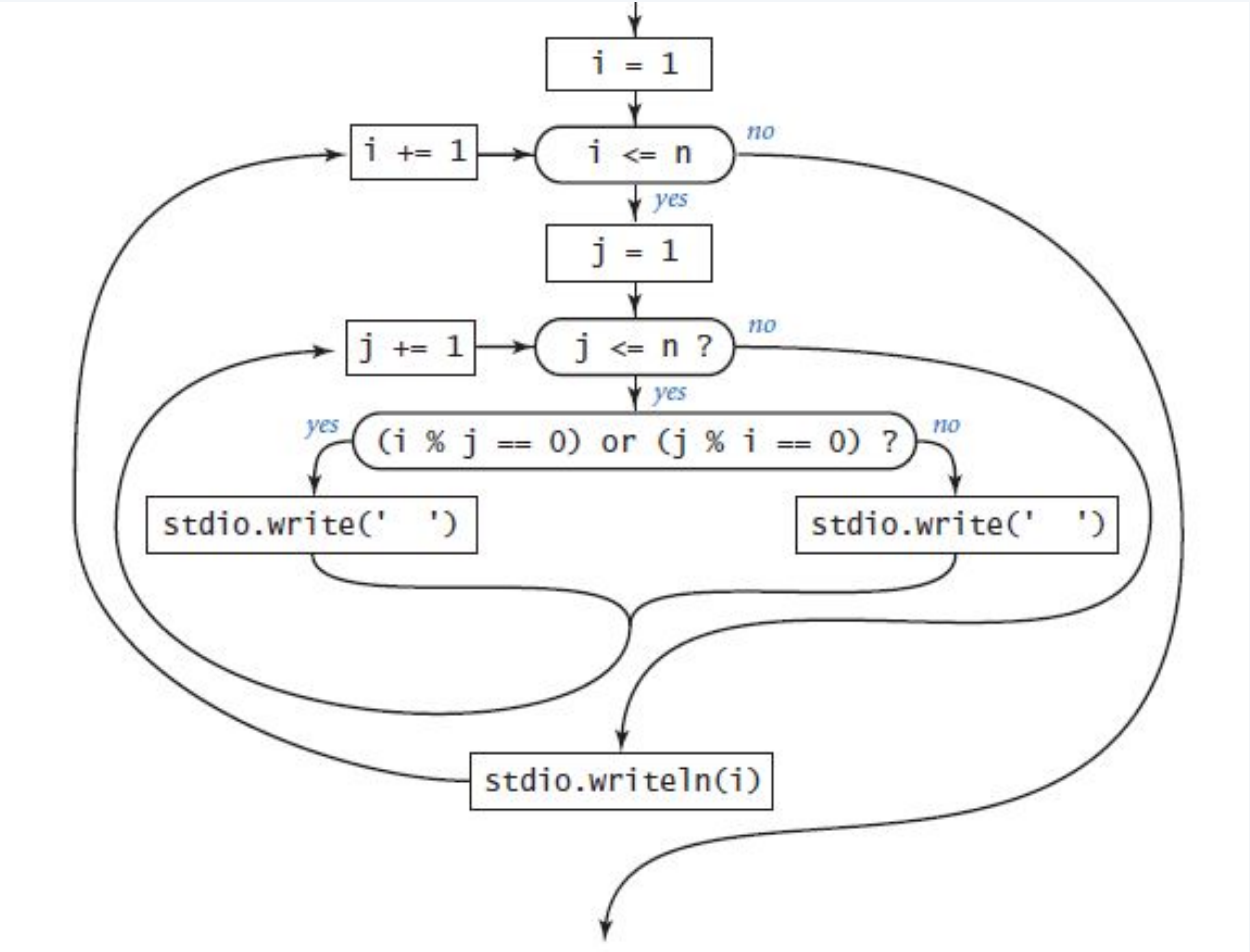


Nested Loop Statements

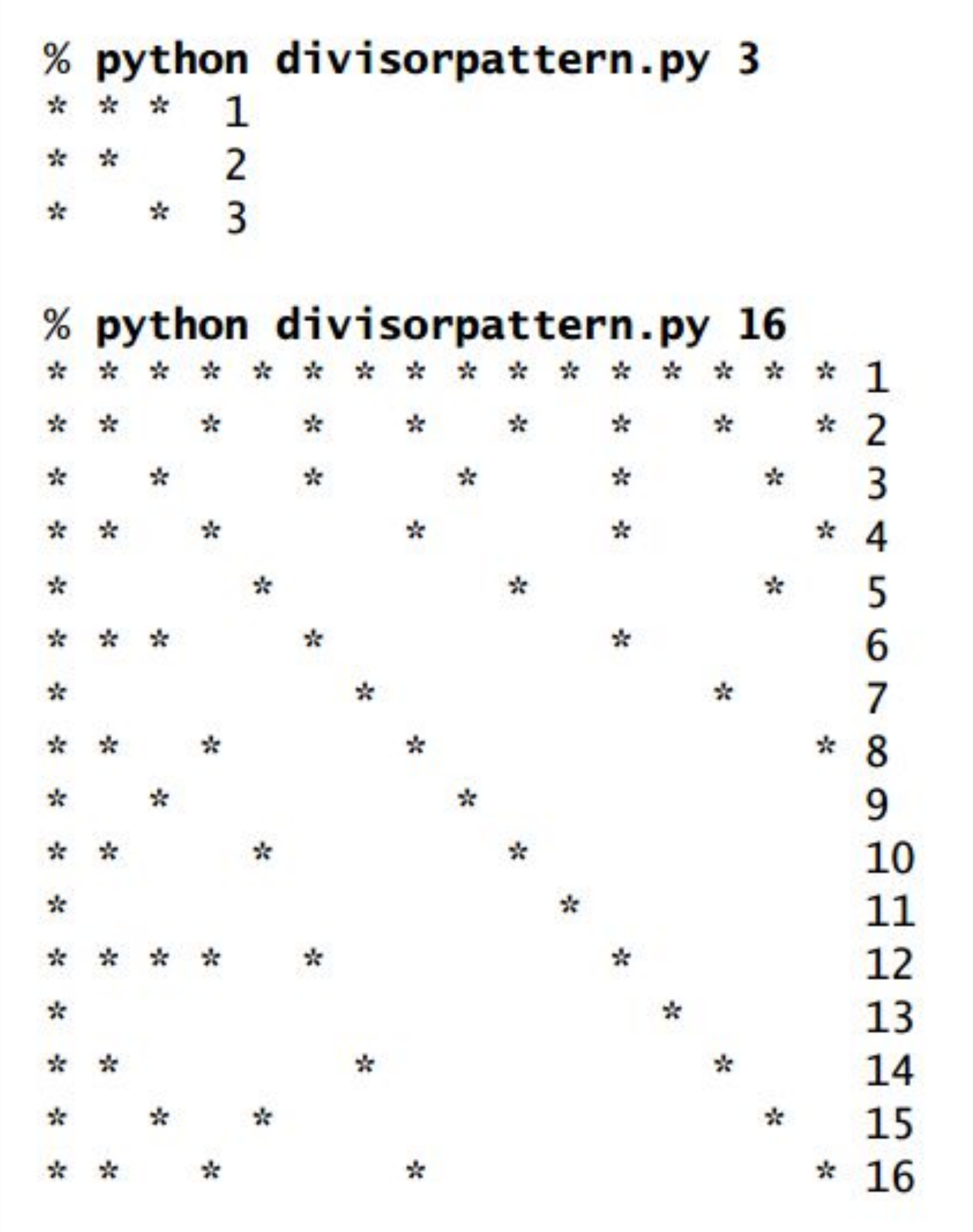


```
import sys
import stdio

n = int(sys.argv[1])

for i in range(1, n+1):
    # Write the ith line.
    for j in range(1, n+1):
        # Write the jth entry in the ith line.
        if (i % j == 0) or (j % i == 0):
            stdio.write('* ')
        else:
            stdio.write('  ')
    stdio.writeln(i)
```

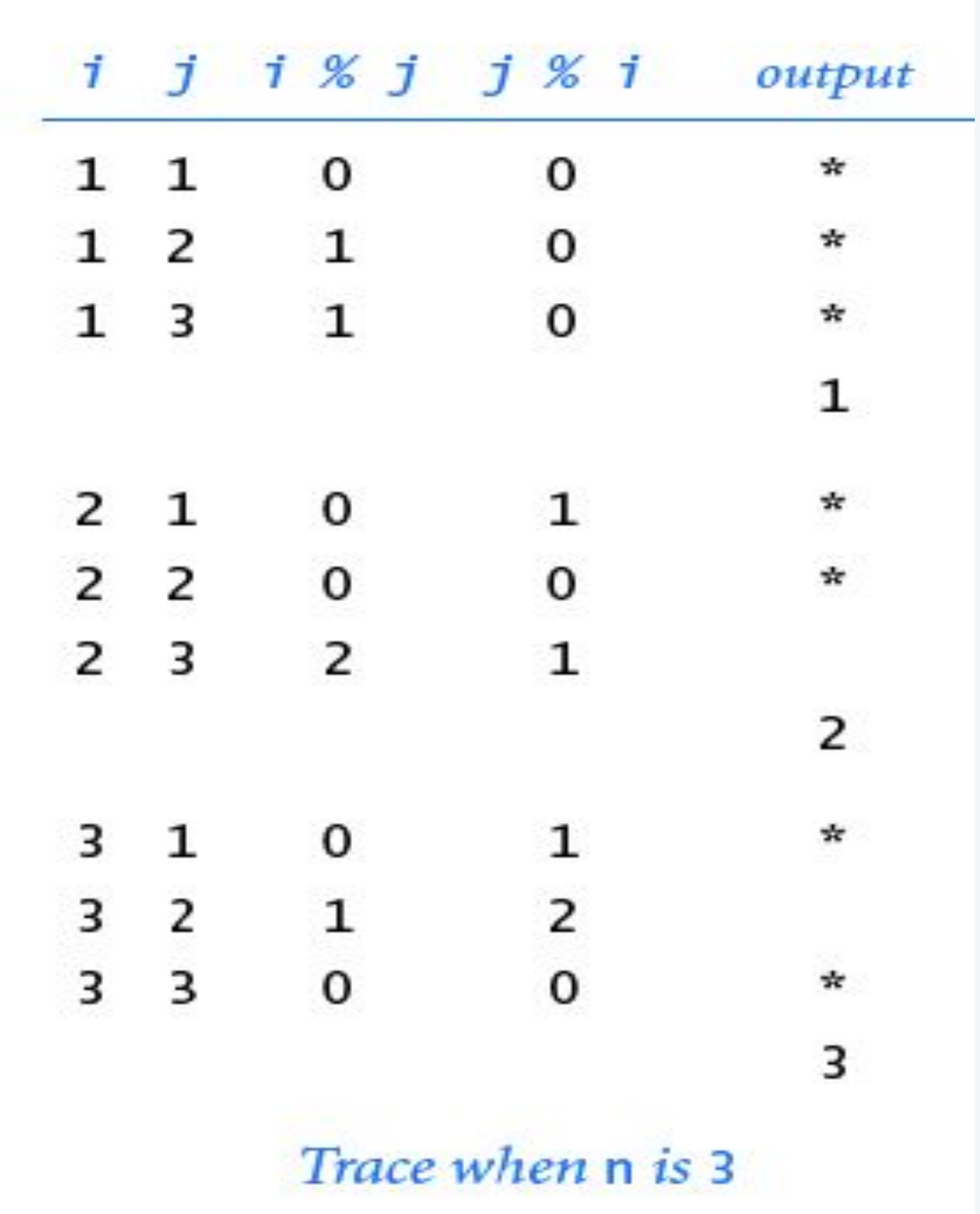
Flow Diagram of Divisor Pattern



Trace of DivisorPattern



```
% python divisorpattern.py 3
* * *  1
* *    2
*   *  3

% python divisorpattern.py 16
* * * * * * * * * * * * * * * * 1
* *   *   *   *   *   *   *   * 2
*   *     *     *     *     *   3
* *   *       *       *       * 4
*       *         *         *   5
* * *     *           *         6
*           *             *     7
* *   *       *               * 8
*   *           *               9
* *     *         *             10
*                   *           11
* * * *   *           *         12
*                       *       13
* *           *             *   14
*   *   *                   *   15
* *   *       *               * 16
```



i	*j*	*i % j*	*j % i*	*output*
1	1	0	0	*
1	2	1	0	*
1	3	1	0	*
				1
2	1	0	1	*
2	2	0	0	*
2	3	2	1	
				2
3	1	0	1	*
3	2	1	2	
3	3	0	0	*
				3

Trace when n *is* 3

Exercise: Modify DivisorPattern

Modify DivisorPattern.py to print the following pattern:

```
 * * * * * * *
* ** ** ** ** *
 * *** *** ***
*** **** **** *
  ** ***** ****
***** ****** **
 * *** *******
* ***** *******
 ** **** ******
********* *****
   * ***** ****
*********** ***
 **** ****** **
* * ********* *
 * *** *******
```

```
not(A or B) = not A and not B
```

TIP:: De Moore’s law

Answer

Modify DivisorPattern.py to print the following pattern:

```
 * * * * * * *
* ** ** ** ** *
 * *** *** ***
*** **** **** *
  ** ***** ****
***** ****** **
 * *** *******
* ***** *******
 ** **** ******
********* *****
   * ***** ****
*********** ***
 **** ****** **
* * ********* *
 * *** *******
```

```
not(A or B) = not A and not B
```

TIP:: De Morgan's law

```
if (i%j==0) or (j%i==0)
```

→

```
if (i%j != 0) and (j%i != 0)
```

Nested If Statements

Ex. Pay a certain tax rate depending on income level.

Income	Rate
0 - 47,450	22%
47,450 - 114,650	25%
114,650 - 174,700	28%
174,700 - 311,950	33%
311,950 -	35%

5 mutually exclusive alternatives

else if

```
rate = 0.0
if    (income <  47450): rate = 0.22
elif (income < 114650): rate = 0.25
elif (income < 174700): rate = 0.28
elif (income < 311950): rate = 0.33
else:                      rate = 0.35
```

graduated income tax calculation

Nested If Statement Challenge

Q. What's wrong with the following for income tax calculation?

Income	Rate
0 - 47,450	22%
47,450 - 114,650	25%
114,650 - 174,700	28%
174,700 - 311,950	33%
311,950 -	35%

```
rate = 0.35
if (income <  47450): rate = 0.22
if (income < 114650): rate = 0.25
if (income < 174700): rate = 0.28
if (income < 311950): rate = 0.33
```

wrong graduated income tax calculation

Nesting conditionals and loops

Nesting

- Any “statement” within a conditional or loop may itself be a conditional or a loop statement.
- Enables complex control flows.
- Adds to challenge of debugging.

Example:

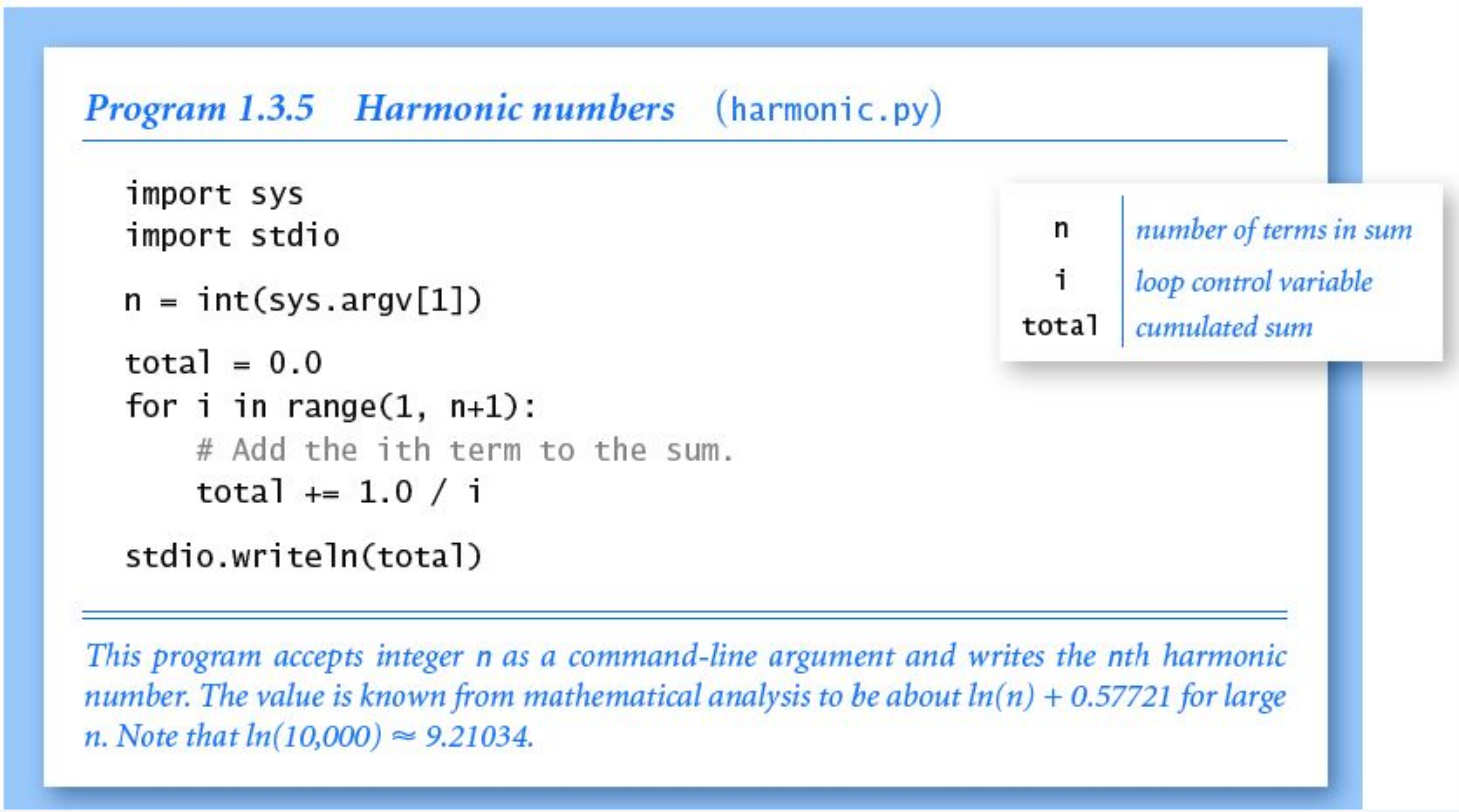
```
for t in range(0,trials):
    cash = stake
    while (cash > 0 and cash < goal):
        if (stdrandom.uniform() < 0.5):
            cash += 1
        else:
            cash -= 1
    if (cash == goal): wins += 1
```

`if-else` statement within a `while` loop within a `for` loop

Gambler's ruin simulation

Math Applications

Harmonic Numbers



Program 1.3.5 Harmonic numbers (harmonic.py)

```
import sys
import stdio

n = int(sys.argv[1])

total = 0.0
for i in range(1, n+1):
    # Add the ith term to the sum.
    total += 1.0 / i

stdio.writeln(total)
```

n	*number of terms in sum*
i	*loop control variable*
total	*cumulated sum*

This program accepts integer n as a command-line argument and writes the nth harmonic number. The value is known from mathematical analysis to be about ln(n) + 0.57721 for large n. Note that ln(10,000) ≈ 9.21034.

$$H = \sum_{i=1}^{N} \frac{1}{i}$$

Exercise: Harmonic

What does the following command, python3 Harmonic.py 3, print to the terminal when executed?

Answer

What does the following command, python Harmonic.py 3, print to the terminal when executed?

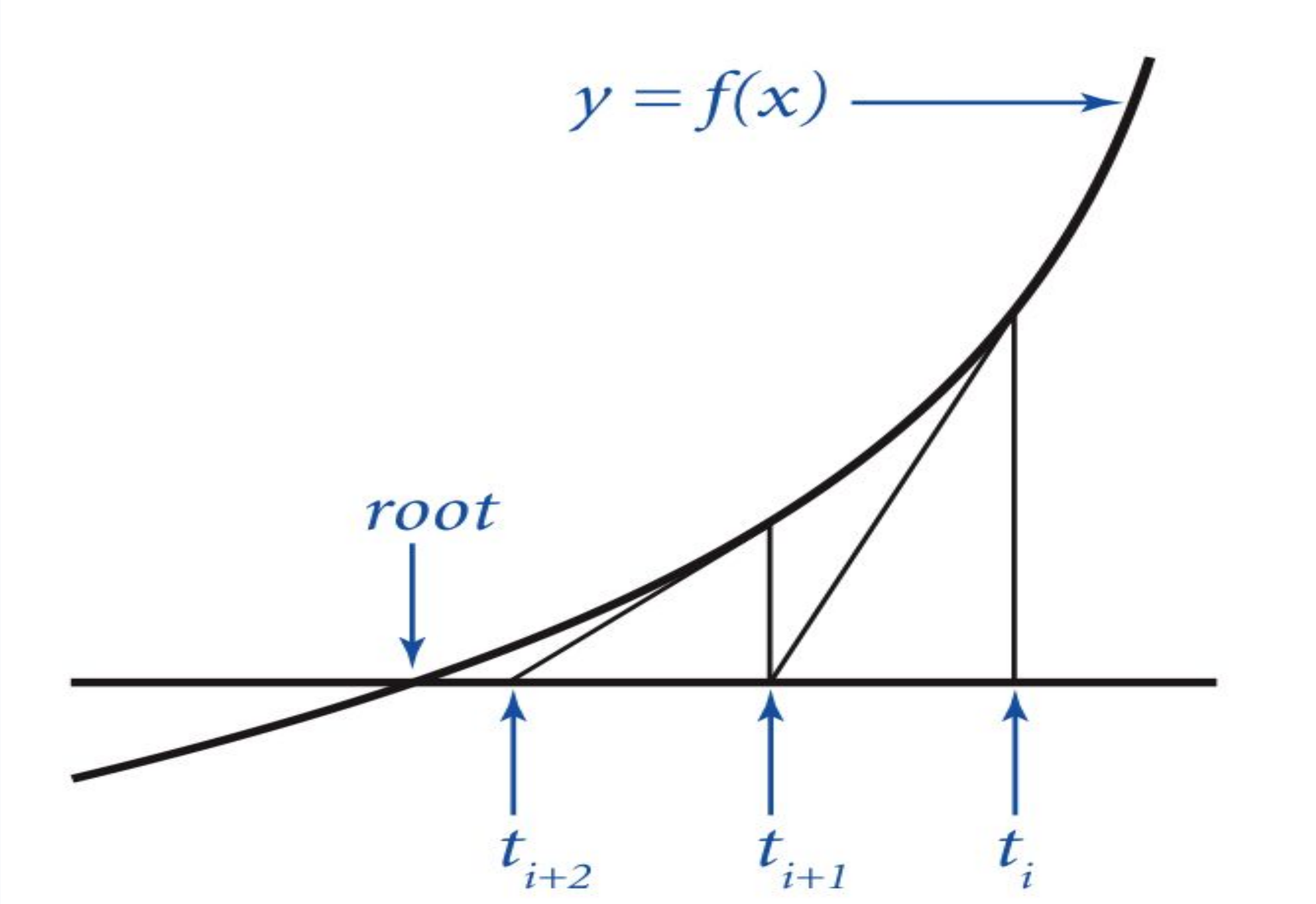
```
1.833333333333
```

Newton-Raphson Method

Square root method explained.

- Goal: find root of any function f(x).
- Start with estimate t_0.
- Draw line tangent to curve at x= t_i.
- Set t_{i+1} to be x-coordinate where the line intersects the x-axis.
- Repeat until desired precision.

$f(x) = x^2 - c$ to compute $\sqrt{c}$



$$t_{i+1} = t_i - \frac{f(t_i)}{f'(t_i)}$$

Technical conditions. f(x) is smooth; t_0 is good estimate.

Math

Computing Constant

$$\begin{aligned} y &= mx + c_0 \\ f(x_t) &= f'(x_t)x_t + c_0 \\ c_0 &= f(x_t) - f'(x_t)x_t \end{aligned}$$

Tangent Line

$$\begin{aligned} y &= f'(x_t)x + f(x_t) - f'(x_t)x_t \\ y &= f'(x_t)[x - x_t] + f(x_t) \\ 0 &= f'(x_t)[x - x_t] + f(x_t) \\ x_{t+1} &= x_t - \frac{f(x_t)}{f'(x_t)} \end{aligned}$$

$f(x) = x^2 - c$ to compute $\sqrt{c}$

$$\begin{aligned} t_{i+1} &= t_i - \frac{t_i^2 - c}{2t_i} \\ t_{i+1} &= \frac{1}{2}\left(t_i + \frac{c}{t_i}\right) \end{aligned}$$

Substitution

While Loops: Square Root

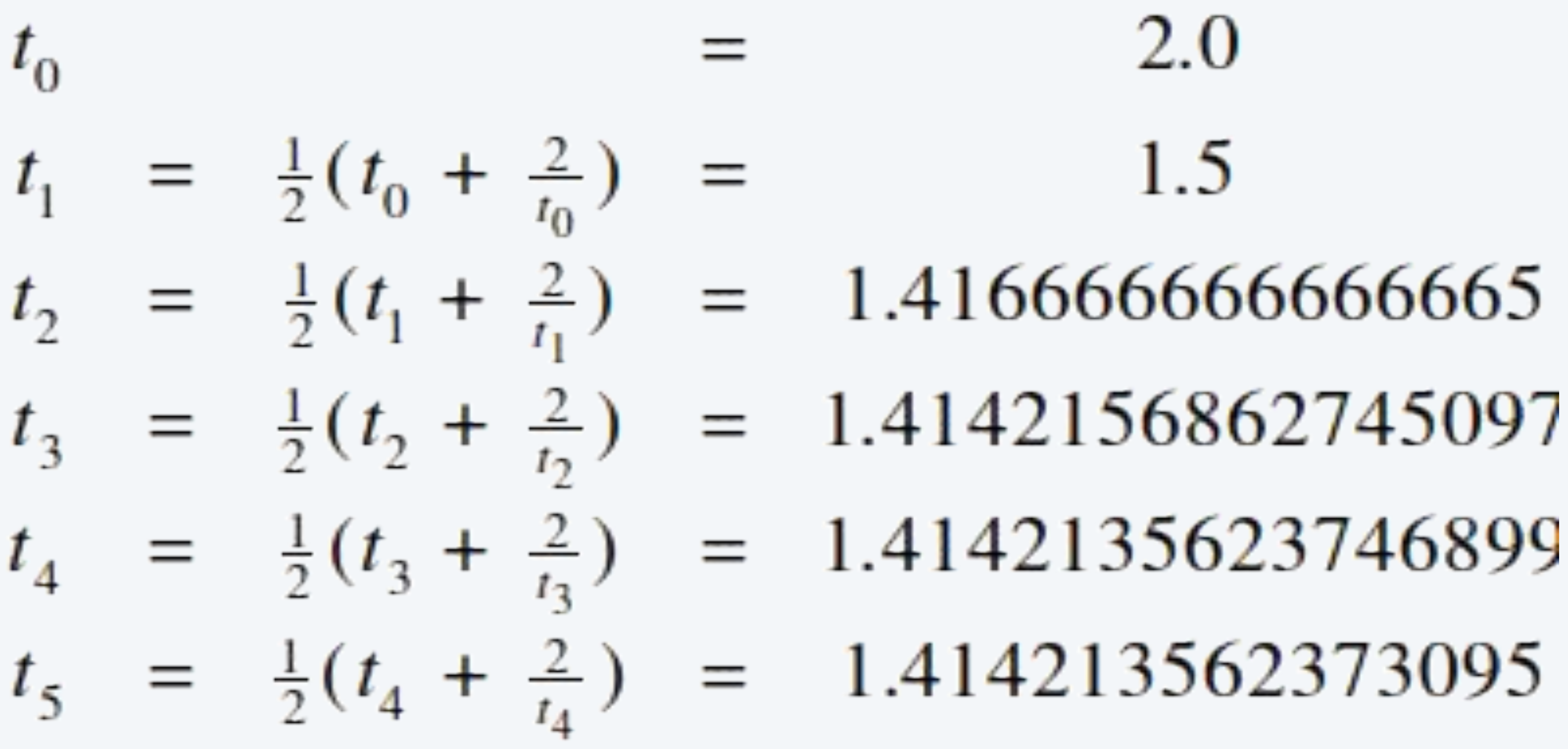
Goal. Implement **math.sqrt()**.

```
% python3 sqrt.py 2.0
1.414213562373095
```

15 decimal digits of accuracy in 5 iterations

Newton-Raphson method to compute the square root of c:

- Initialize t_0 = c.
- Repeat up to desired precision:
  set t_{i+1} to be the average of t_i and c / t_i.



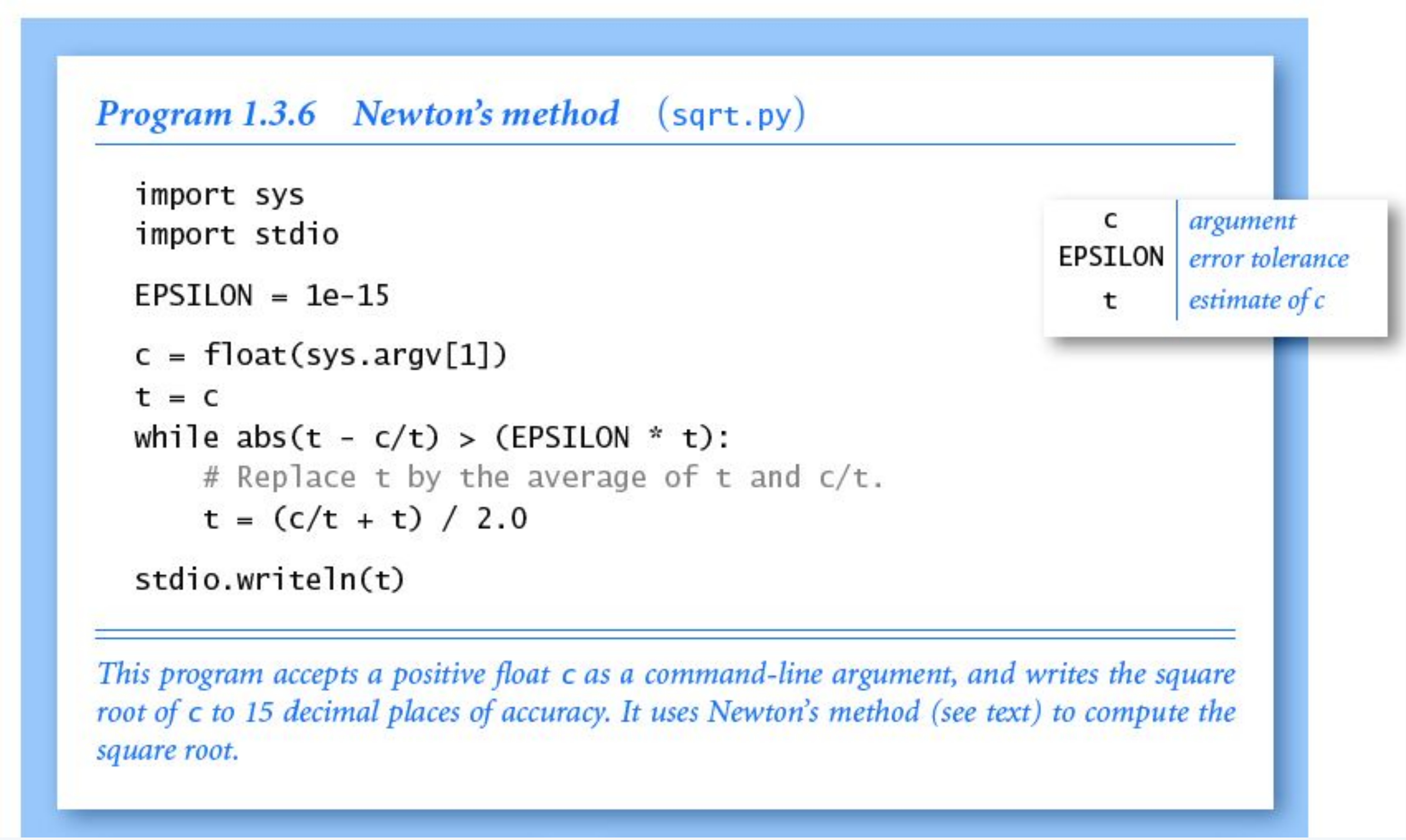
$$
\begin{aligned}
t_0 & & &= 2.0 \\
t_1 &= \tfrac{1}{2}(t_0 + \tfrac{2}{t_0}) &&= 1.5 \\
t_2 &= \tfrac{1}{2}(t_1 + \tfrac{2}{t_1}) &&= 1.4166666666666665 \\
t_3 &= \tfrac{1}{2}(t_2 + \tfrac{2}{t_2}) &&= 1.4142156862745097 \\
t_4 &= \tfrac{1}{2}(t_3 + \tfrac{2}{t_3}) &&= 1.4142135623746899 \\
t_5 &= \tfrac{1}{2}(t_4 + \tfrac{2}{t_4}) &&= 1.414213562373095
\end{aligned}
$$

computing the square root of 2

"A wonderful square root. Let's hope it can be used for the good of mankind."

Copyright 2004, Sidney Harris
www.sciencecartoonsplus.com

While Loops: Square Root

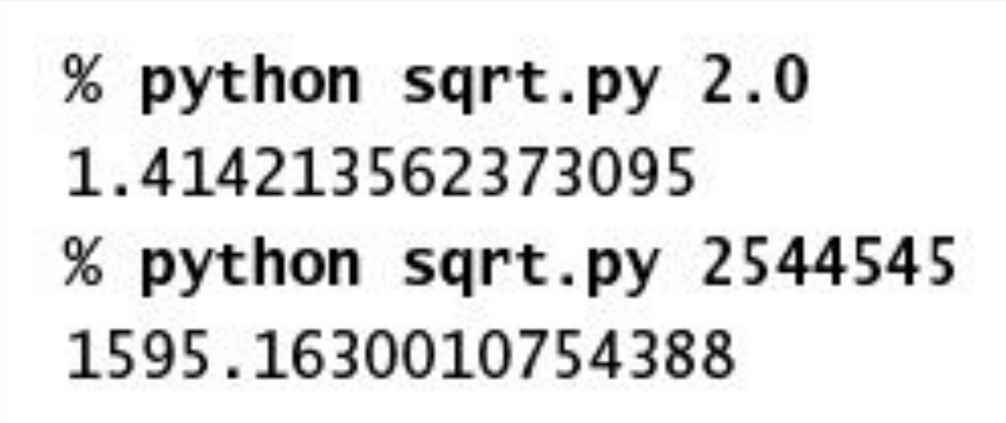


Program 1.3.6 *Newton's method* (sqrt.py)

```
import sys
import stdio

EPSILON = 1e-15

c = float(sys.argv[1])
t = c
while abs(t - c/t) > (EPSILON * t):
    # Replace t by the average of t and c/t.
    t = (c/t + t) / 2.0

stdio.writeln(t)
```

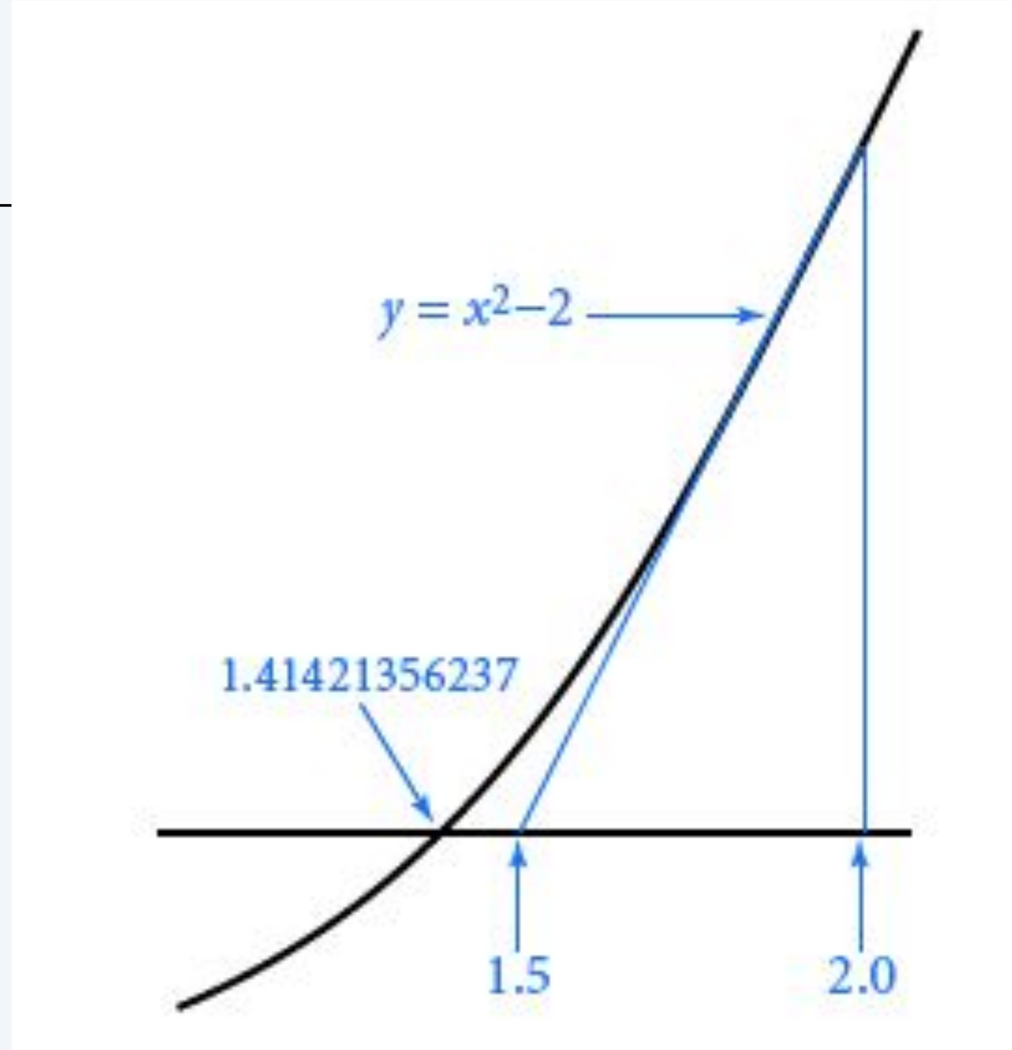
c	*argument*
EPSILON	*error tolerance*
t	*estimate of c*

This program accepts a positive float c *as a command-line argument, and writes the square root of* c *to 15 decimal places of accuracy. It uses Newton's method (see text) to compute the square root.*



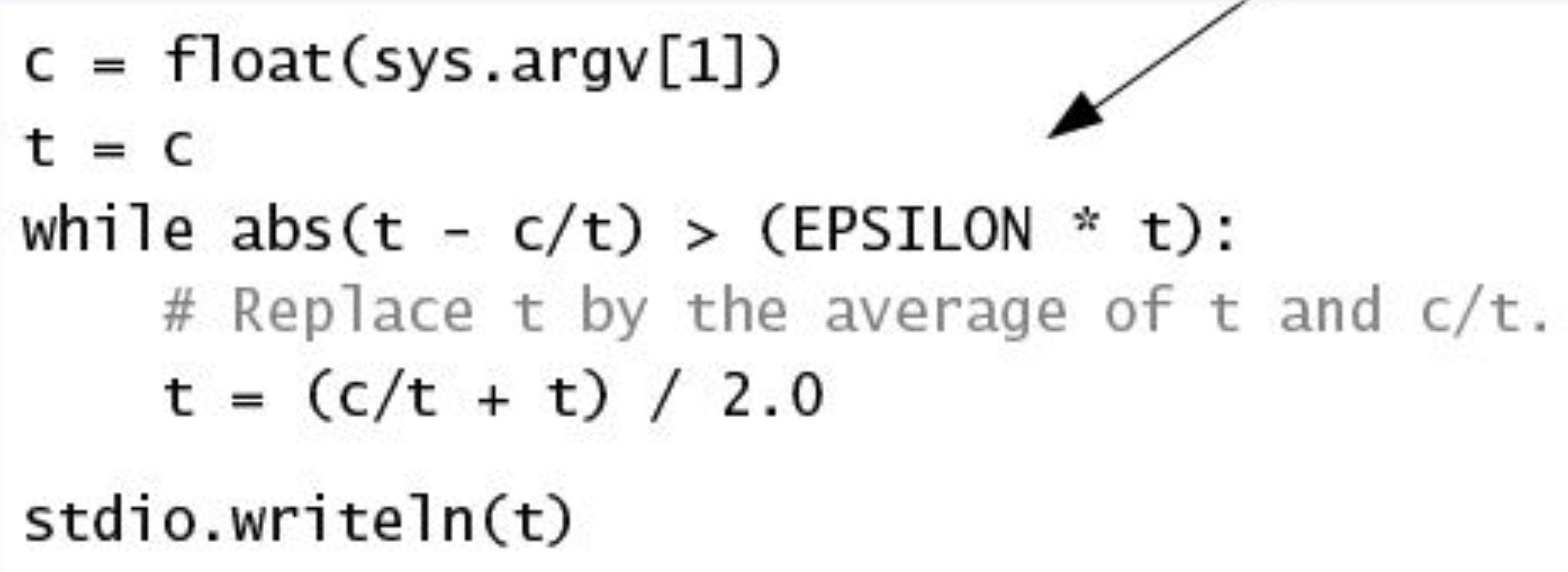
```
% python sqrt.py 2.0
1.414213562373095
% python sqrt.py 2544545
1595.1630010754388
```

While Loops: Square Root

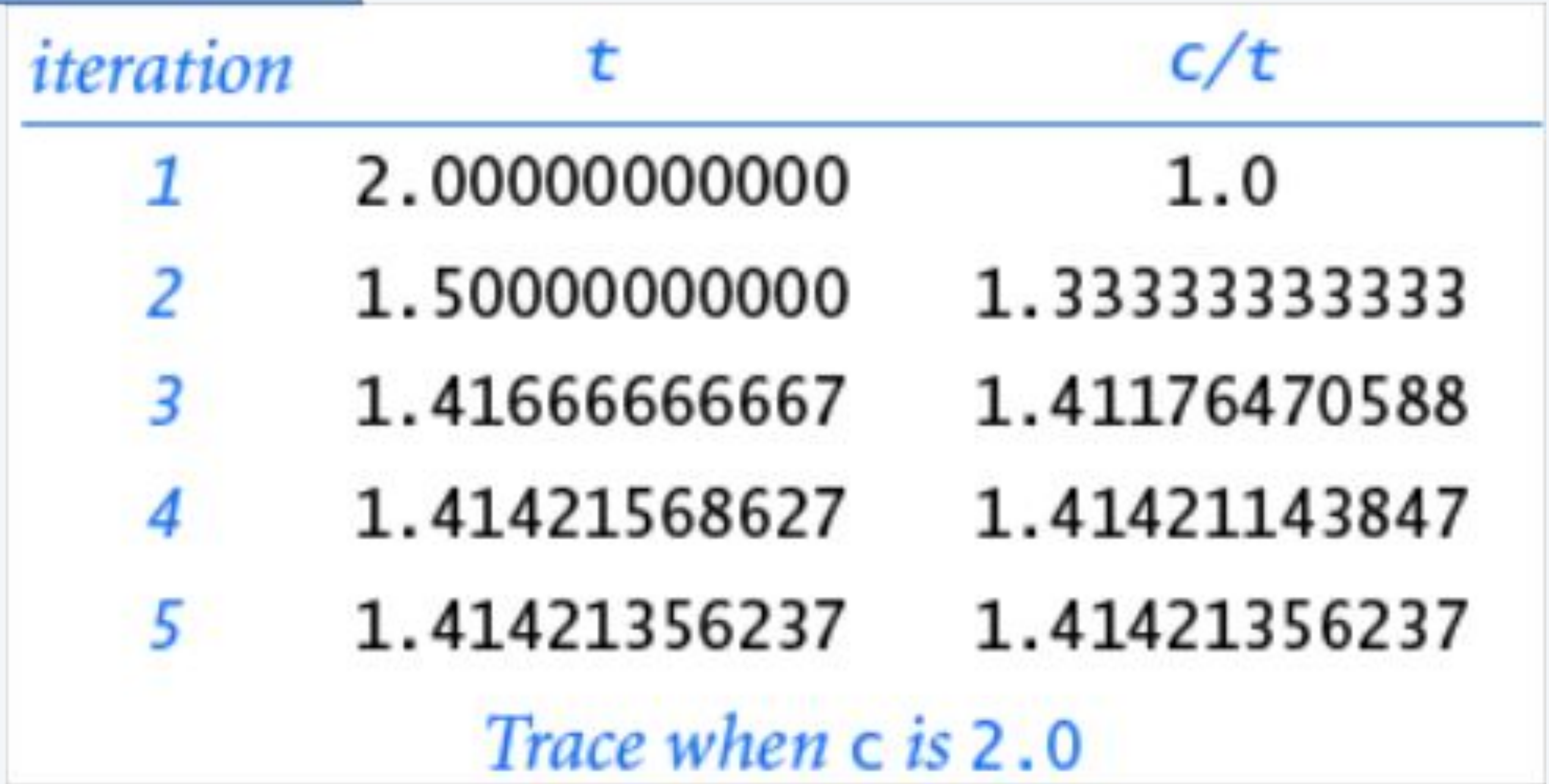


abs => absolute value

ensures error tolerance is of correct magnitude

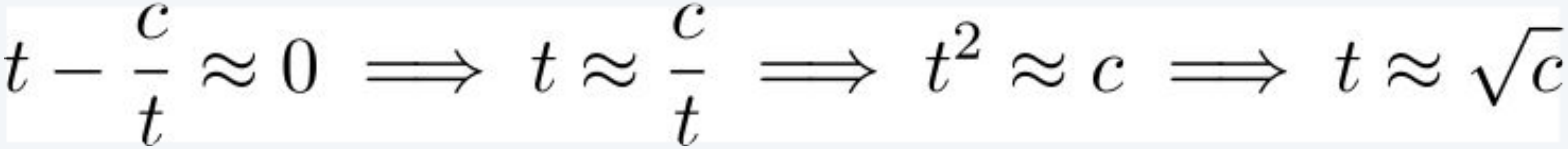


```
c = float(sys.argv[1])
t = c
while abs(t - c/t) > (EPSILON * t):
    # Replace t by the average of t and c/t.
    t = (c/t + t) / 2.0

stdio.writeln(t)
```



iteration	t	c/t
1	2.00000000000	1.0
2	1.50000000000	1.33333333333
3	1.41666666667	1.41176470588
4	1.41421568627	1.41421143847
5	1.41421356237	1.41421356237

Trace when c is 2.0



$$t - \frac{c}{t} \approx 0 \implies t \approx \frac{c}{t} \implies t^2 \approx c \implies t \approx \sqrt{c}$$

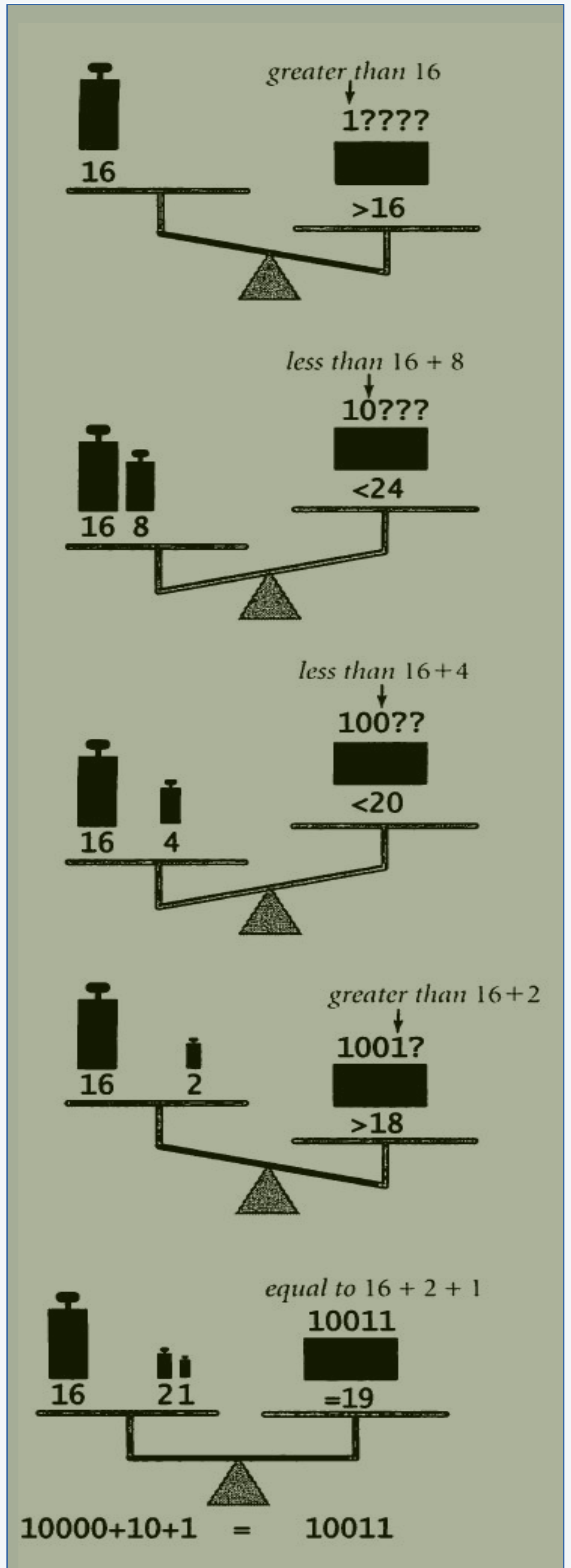
Exercise: Even While Loop

- Write a while loop that prints out all the even numbers between 1 and N.
- Assume N has already been defined. Include N in your calculation.

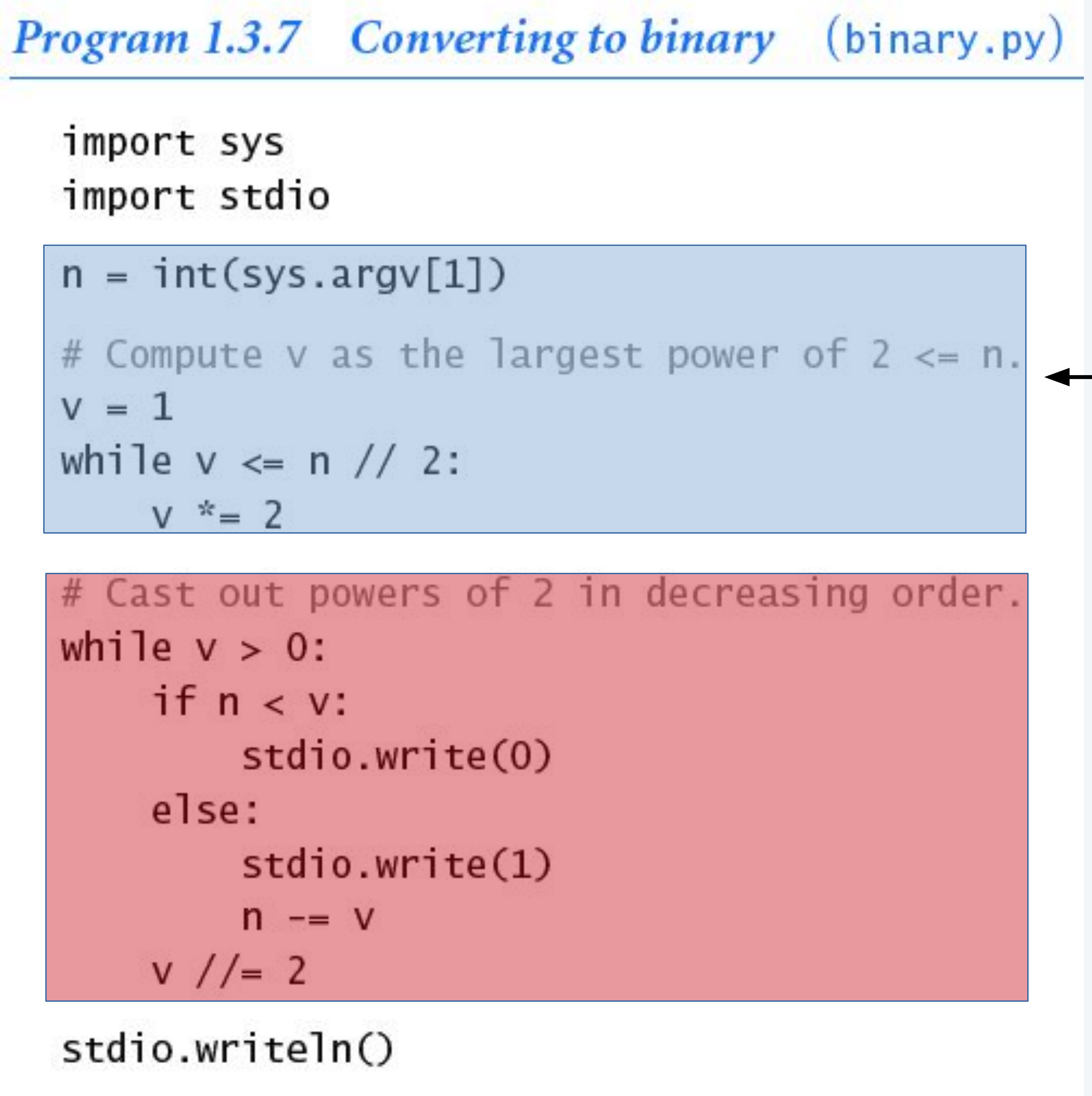
Answer

Write a while loop that prints out all the even numbers between 1 and N.
Assume N has already been defined. Include N in your calculation.

```
i = 1
while (i<=N):
  if (i%2 == 0):
      stdio.write(str(i)+” “)
   i += 1;

stdio.writeln()
```



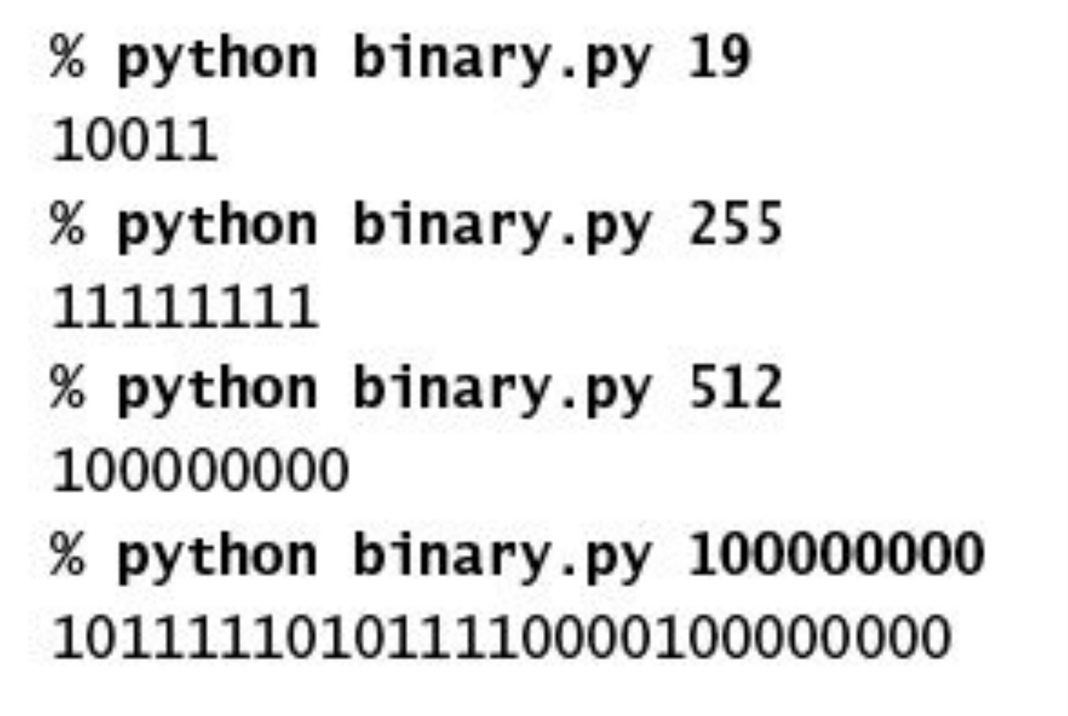
Scale analog to binary conversion



Program 1.3.7 ***Converting to binary*** (binary.py)

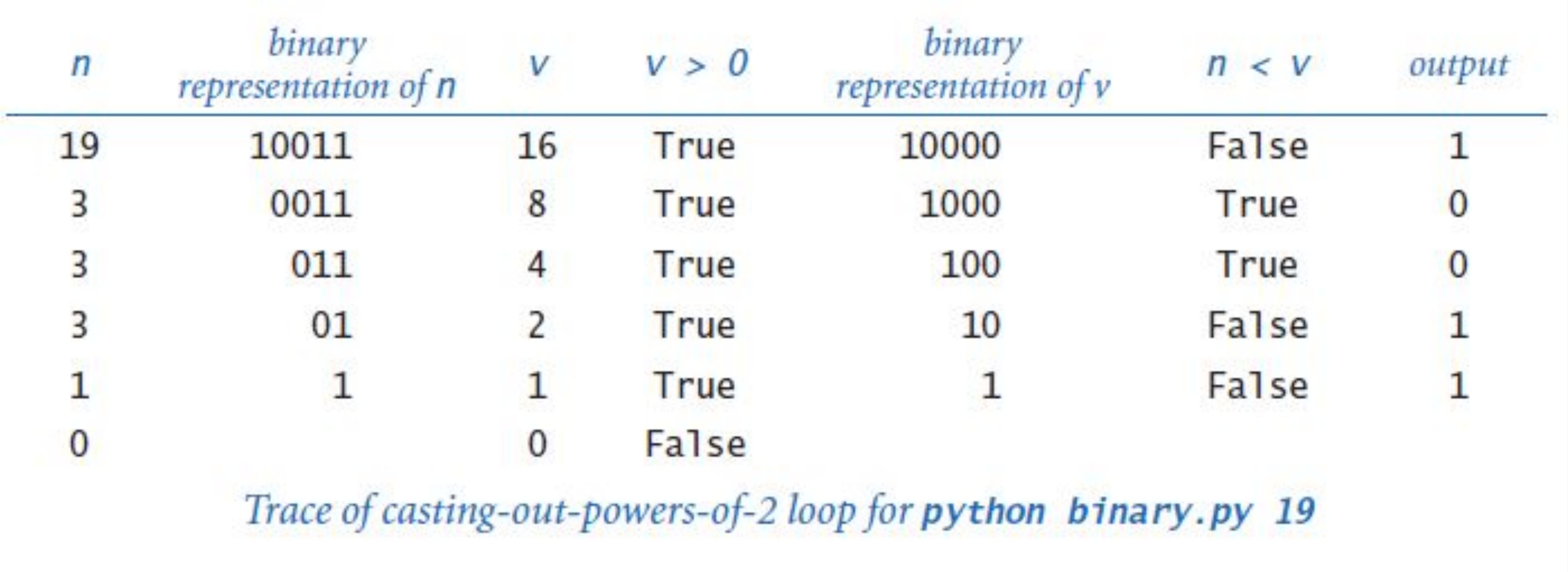
```
import sys
import stdio

n = int(sys.argv[1])

# Compute v as the largest power of 2 <= n.
v = 1
while v <= n // 2:
    v *= 2

# Cast out powers of 2 in decreasing order.
while v > 0:
    if n < v:
        stdio.write(0)
    else:
        stdio.write(1)
        n -= v
    v //= 2

stdio.writeln()
```

Refactoring



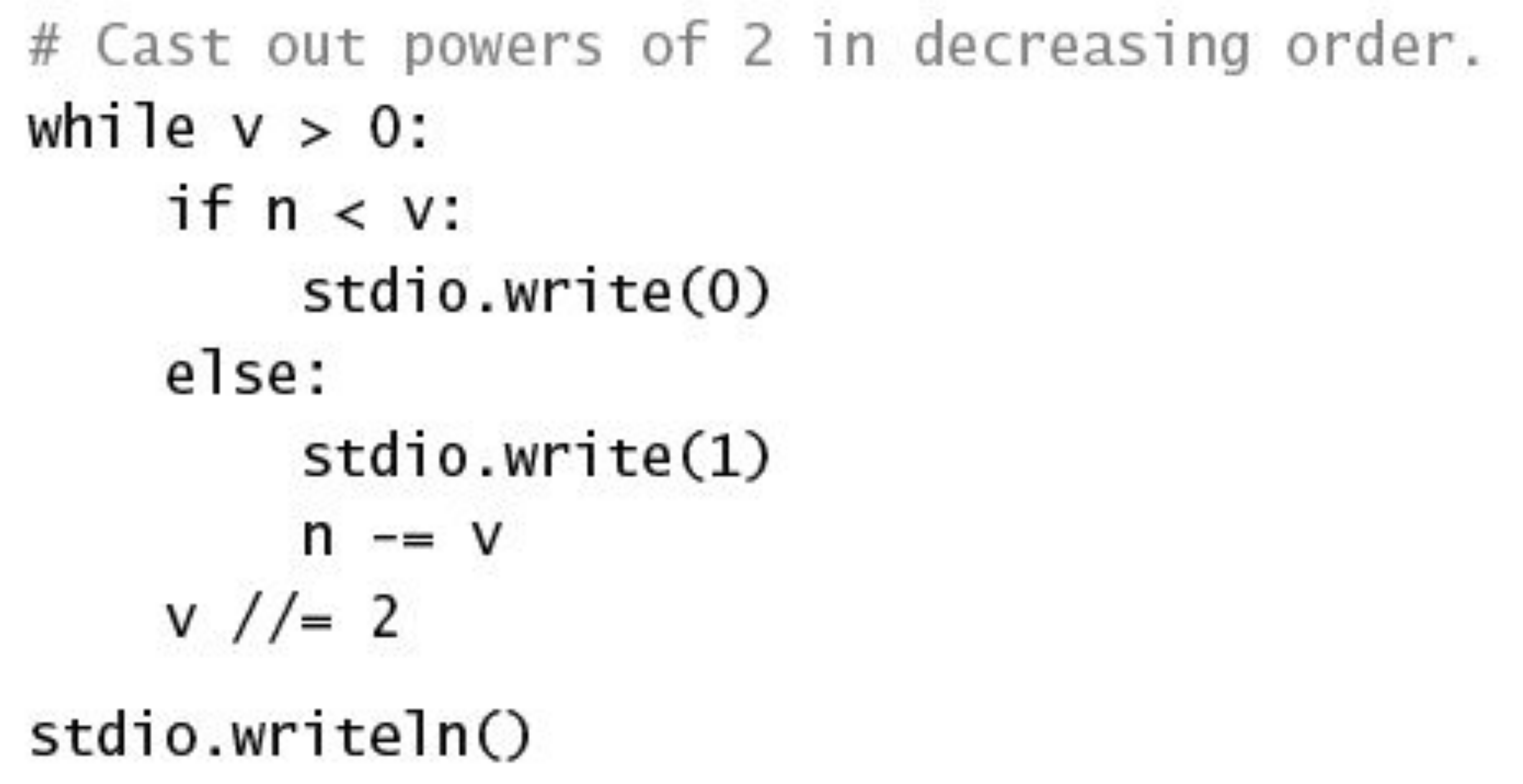
```
% python binary.py 19
10011
% python binary.py 255
11111111
% python binary.py 512
1000000000
% python binary.py 100000000
101111101011110000100000000
```

$$19 = 1.2^0 + 1.2^1 + 0.2^2 + 0.2^3 + 1.2^4$$



n	*binary representation of n*	*v*	*v > 0*	*binary representation of v*	*n < v*	*output*
19	10011	16	True	10000	False	1
3	0011	8	True	1000	True	0
3	011	4	True	100	True	0
3	01	2	True	10	False	1
1	1	1	True	1	False	1
0		0	False			

Trace of casting-out-powers-of-2 loop for **`python binary.py 19`**



```
# Cast out powers of 2 in decreasing order.
while v > 0:
    if n < v:
        stdio.write(0)
    else:
        stdio.write(1)
        n -= v
    v //= 2

stdio.writeln()
```

Exercise: Convert To Binary

Convert 21 to its binary representation.

Answer

Convert 21 to its binary representation:

10101

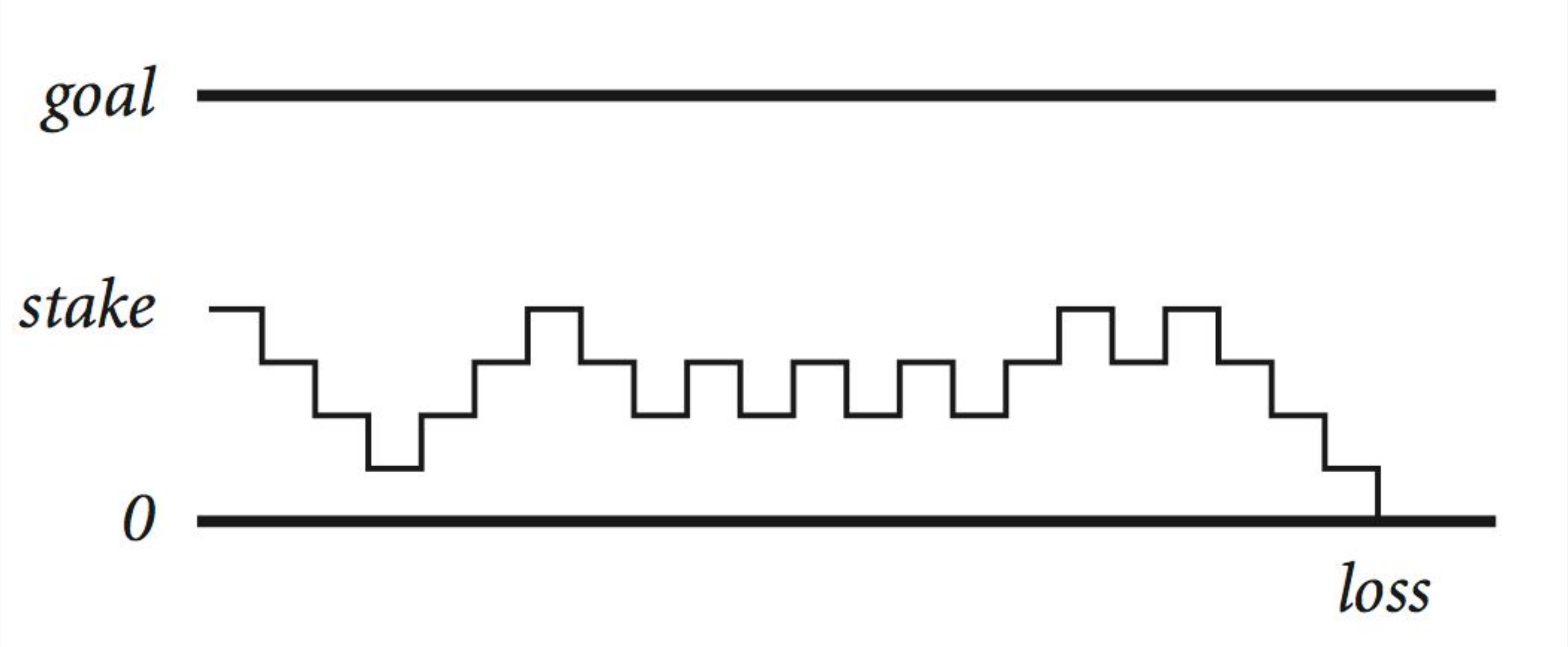
$$21 = 1.2^0 + 0.2^1 + 1.2^2 + 0.2^3 + 1.2^4$$

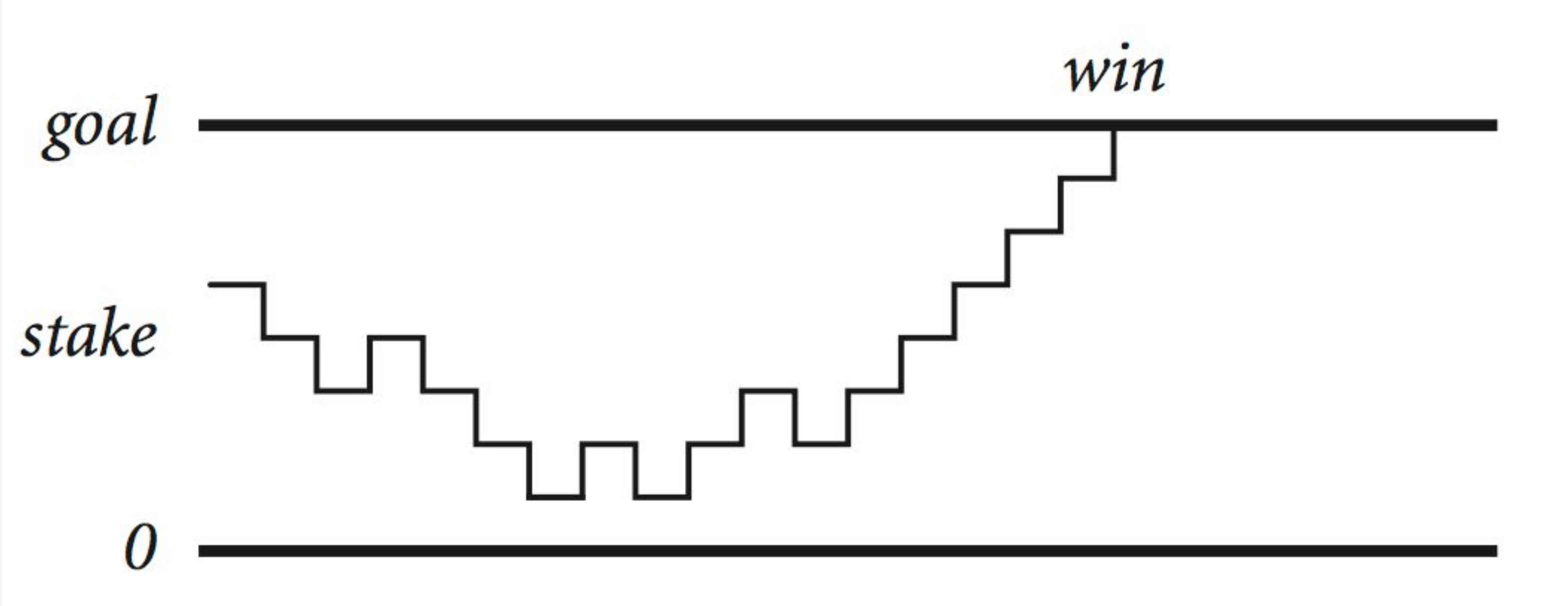
Monte Carlo Simulation

Gambler's ruin problem

A gambler starts with $*stake* and places $1 fair bets.

- Outcome 1 (loss): Gambler goes broke with $0.
- Outcome 2 (win): Gambler reaches $*goal*.



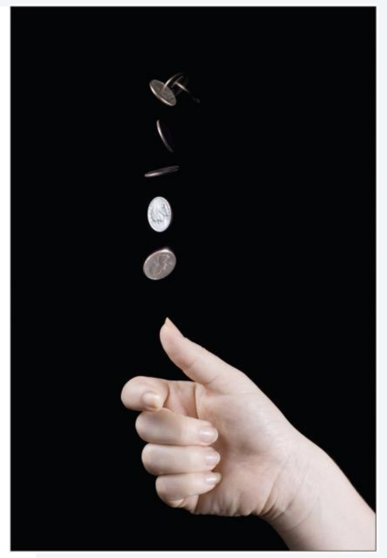


Q. What are the chances of winning?

Q. How many bets until win or loss?

One approach: Monte Carlo simulation.

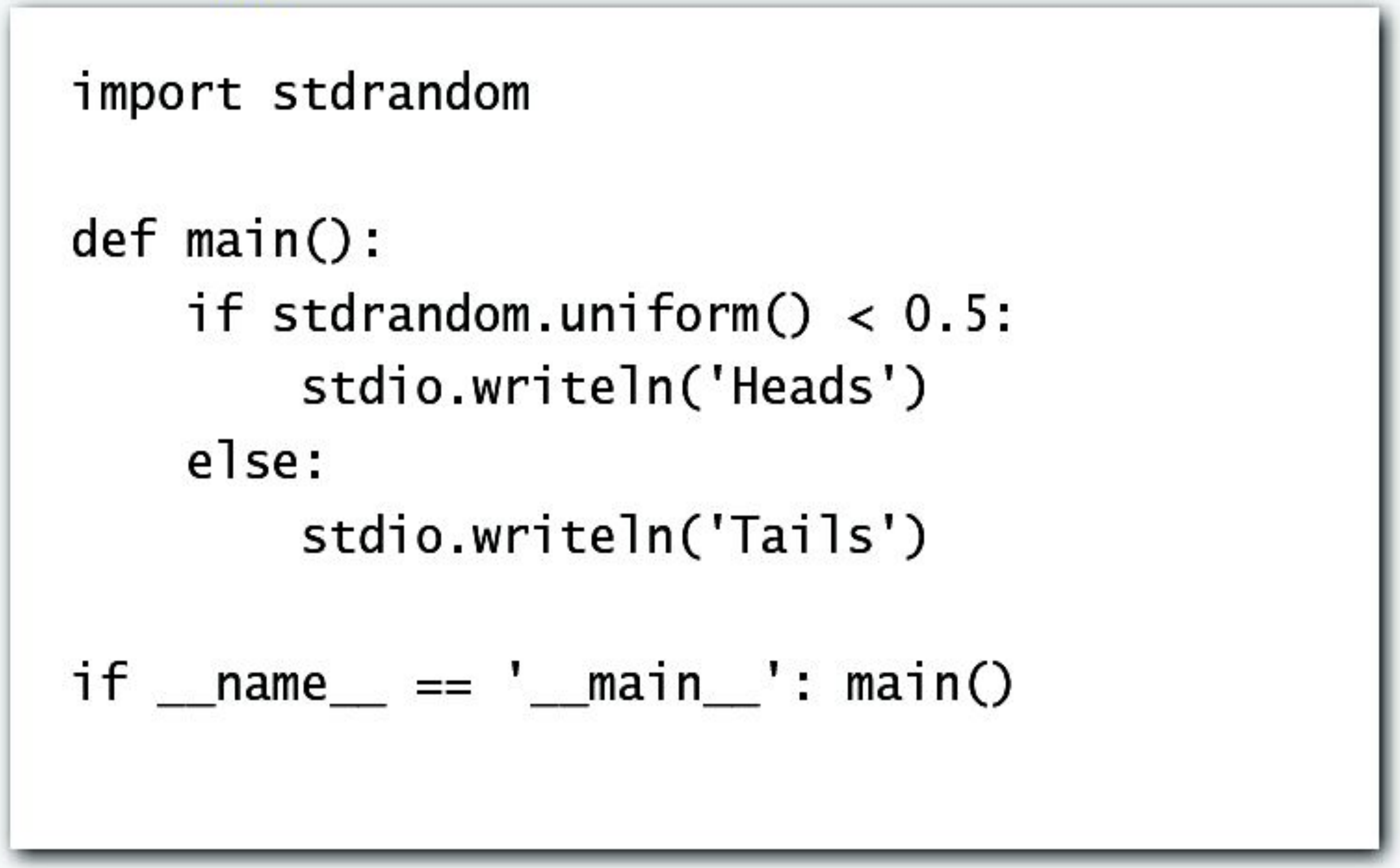
- Use a *simulated coin flip*.
- Repeat and compute statistics.



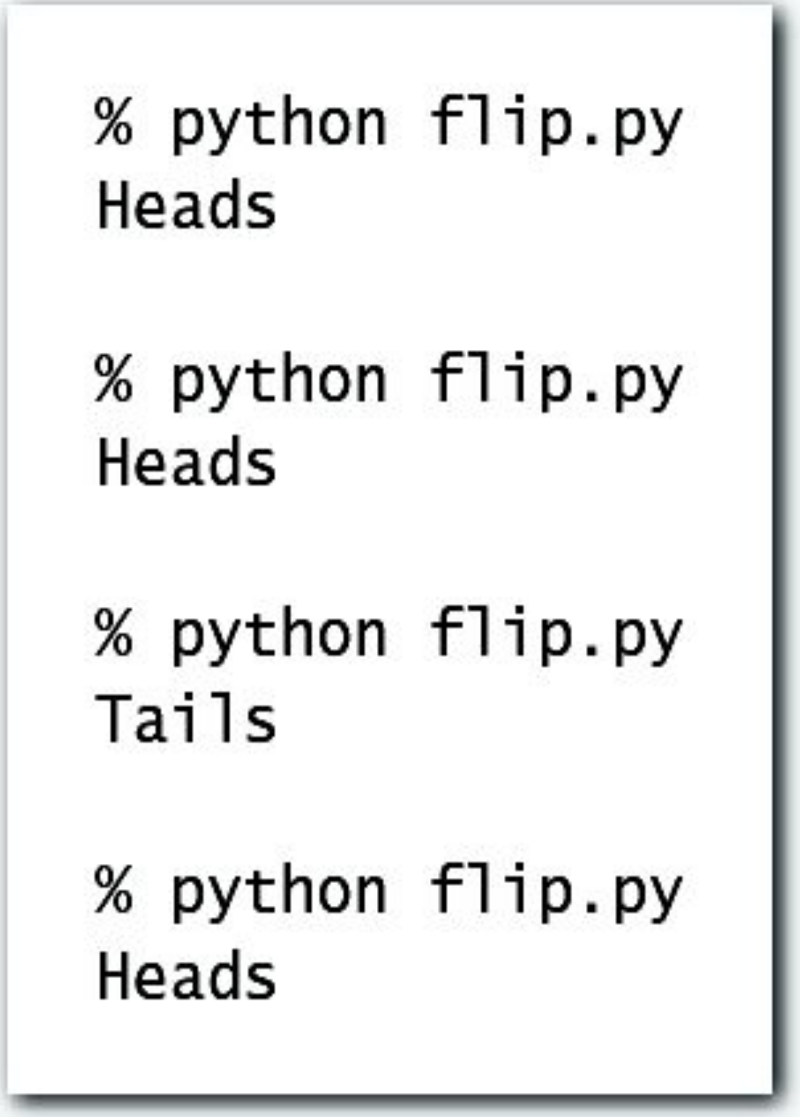
Preamble

Take different action depending on value of variable.

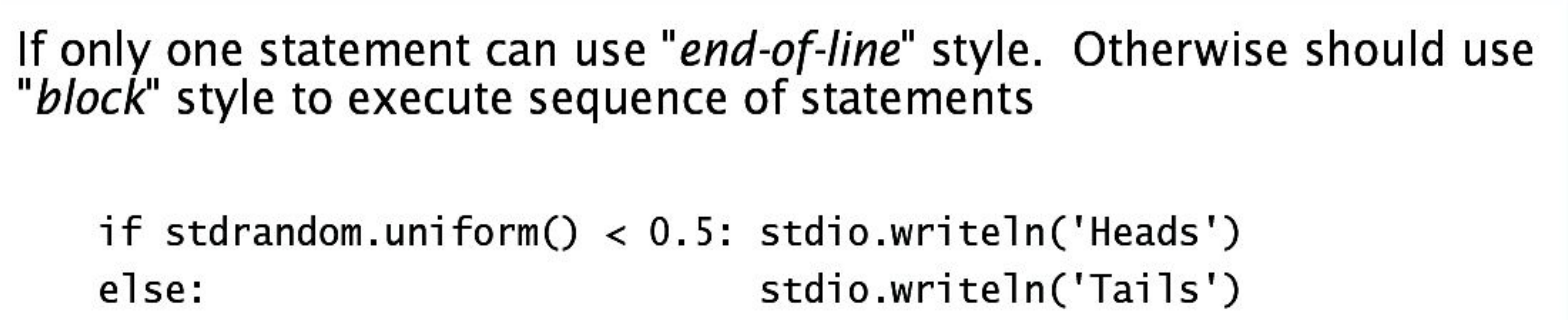
flip.py



```
import stdrandom

def main():
    if stdrandom.uniform() < 0.5:
        stdio.writeln('Heads')
    else:
        stdio.writeln('Tails')

if __name__ == '__main__': main()
```



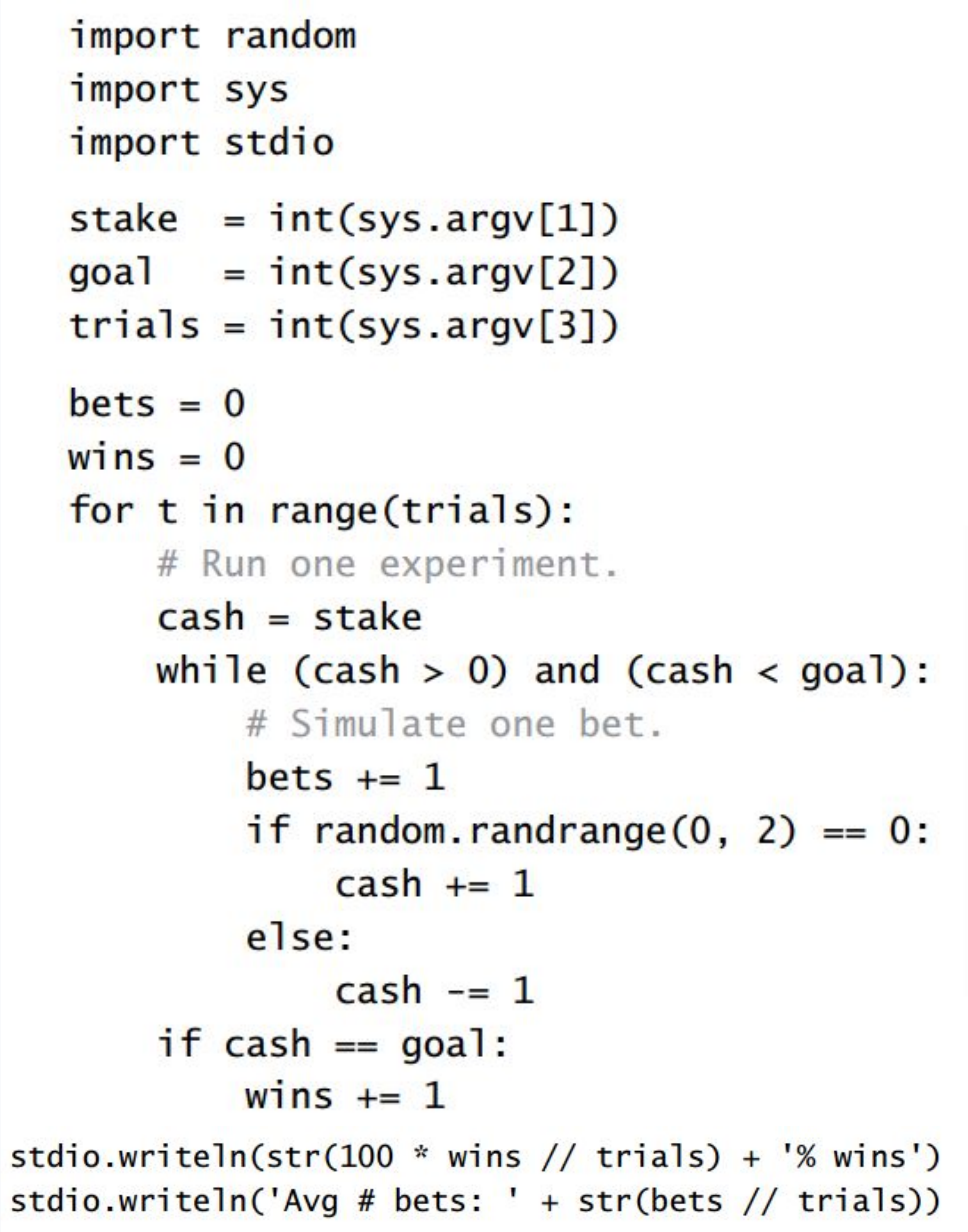
```
% python flip.py
Heads

% python flip.py
Heads

% python flip.py
Tails

% python flip.py
Heads
```



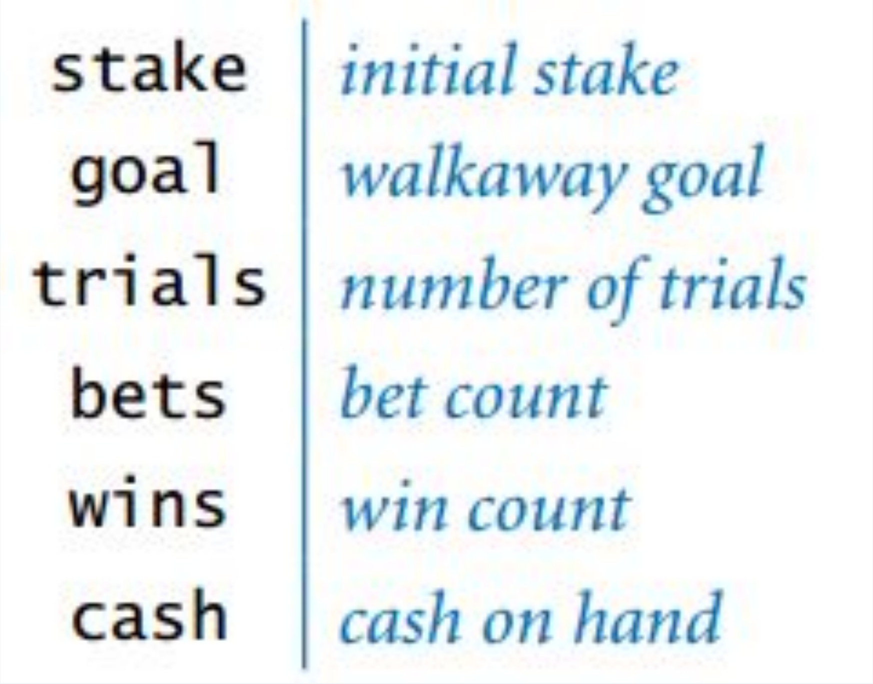
If only one statement can use "*end-of-line*" style. Otherwise should use "*block*" style to execute sequence of statements

```
if stdrandom.uniform() < 0.5: stdio.writeln('Heads')
else:                         stdio.writeln('Tails')
```

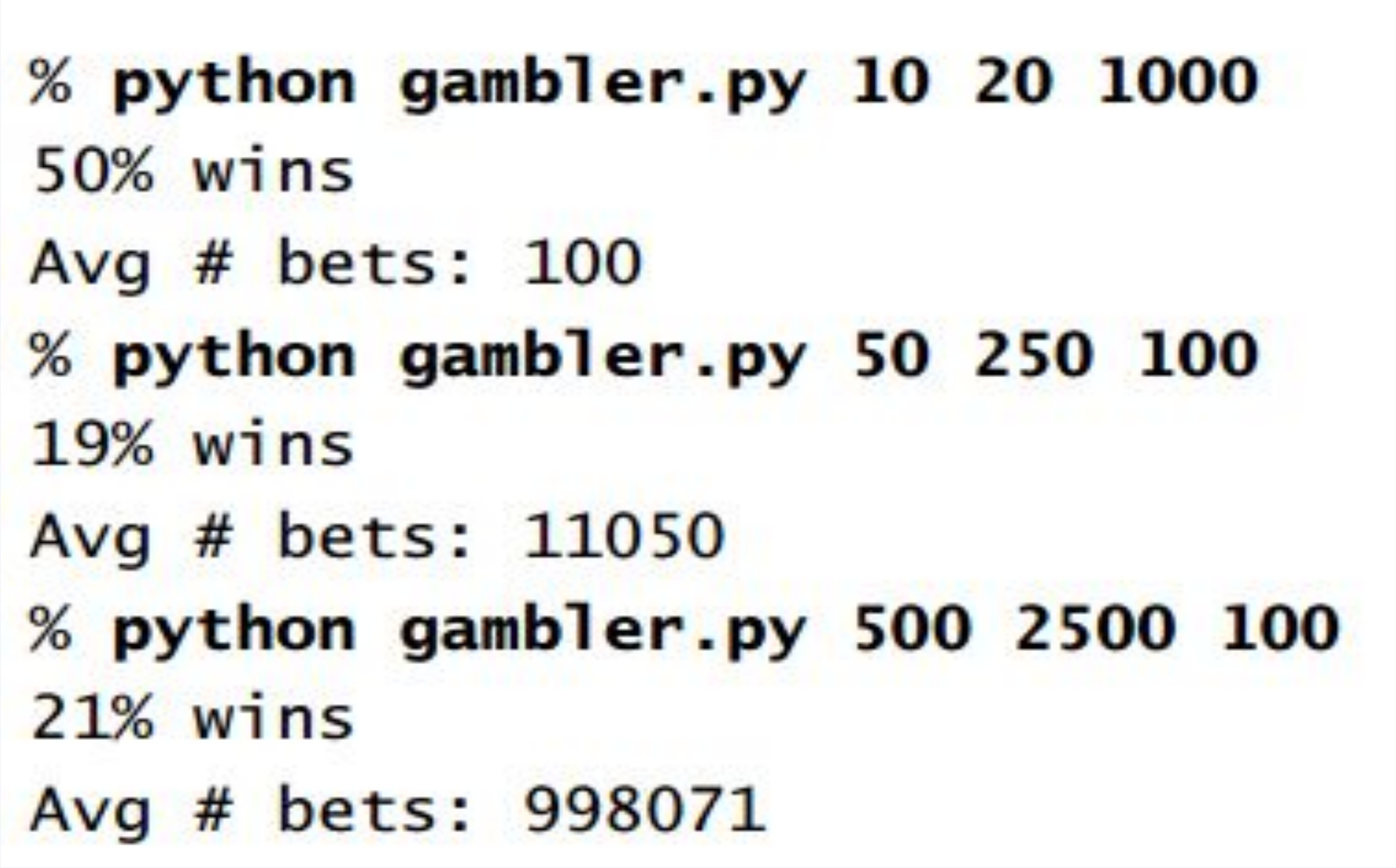
Gambler's Ruin



```
import random
import sys
import stdio

stake  = int(sys.argv[1])
goal   = int(sys.argv[2])
trials = int(sys.argv[3])

bets = 0
wins = 0
for t in range(trials):
    # Run one experiment.
    cash = stake
    while (cash > 0) and (cash < goal):
        # Simulate one bet.
        bets += 1
        if random.randrange(0, 2) == 0:
            cash += 1
        else:
            cash -= 1
    if cash == goal:
        wins += 1
stdio.writeln(str(100 * wins // trials) + '% wins')
stdio.writeln('Avg # bets: ' + str(bets // trials))
```



stake	*initial stake*
goal	*walkaway goal*
trials	*number of trials*
bets	*bet count*
wins	*win count*
cash	*cash on hand*



```
% python gambler.py 10 20 1000
50% wins
Avg # bets: 100
% python gambler.py 50 250 100
19% wins
Avg # bets: 11050
% python gambler.py 500 2500 100
21% wins
Avg # bets: 998071
```

Digression: Simulation and Analysis

stake goal T

```
% python3 gambler.py 5 25 1000
191 wins of 1000

% python3 gambler.py 5 25 1000
203 wins of 1000

% python3 gambler.py 500 2500 1000
197 wins of 1000
```

after a substantial wait….

Fact. Probability of winning = stake ÷ goal.
Fact. Expected number of bets = stake × desired gain.
Ex. 20% chance of turning $500 into $2500,
but expect to make one million $1 bets.

Remark. Both facts can be proved mathematically; for more complex scenarios, computer simulation is often the best (only) plan of attack.

500/2500 = 20%

500 * (2500 - 500) = 1 million

Exercise: Gambler

Estimate the output of the following command, python Gambler.py 400 1600 1000, if it is executed.

Answer

Estimate the output the following command, python Gambler 400 1600 1000, will print to the terminal if it is executed.

(400/1600) x 1000 = 250

Exercise: EvenStripes

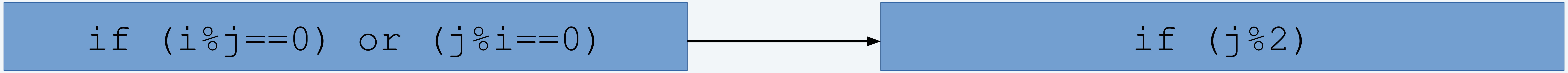
Modify DivisorPattern.py so that it produces the following pattern:

```
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
```

Answer

Modify DivisorPattern.py so that it produces the following pattern:

```
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *
```



```
if (i%j==0) or (j%i==0)
```

```
if (j%2)
```

Control Flow Summary

Control flow.

- Sequence of statements that are actually executed in a program.
- Conditionals and loops: enable us to choreograph the control flow.

Control Flow	Description	Examples
straight-line programs	all statements are executed in the order given	
conditionals	certain statements are executed depending on the values of certain variables	`if` `elif`
loops	certain statements are executed repeatedly until certain conditions are met	`while` `for`

INTRO TO PROGRAMMING
IN PYTHON

SEDGEWICK · WAYNE · DONDERO

2. Conditionals & Loops

- Conditionals: the `if` statement
- Loops: the `while` statement
- An alternative: the `for` loop
- Nesting
- Debugging

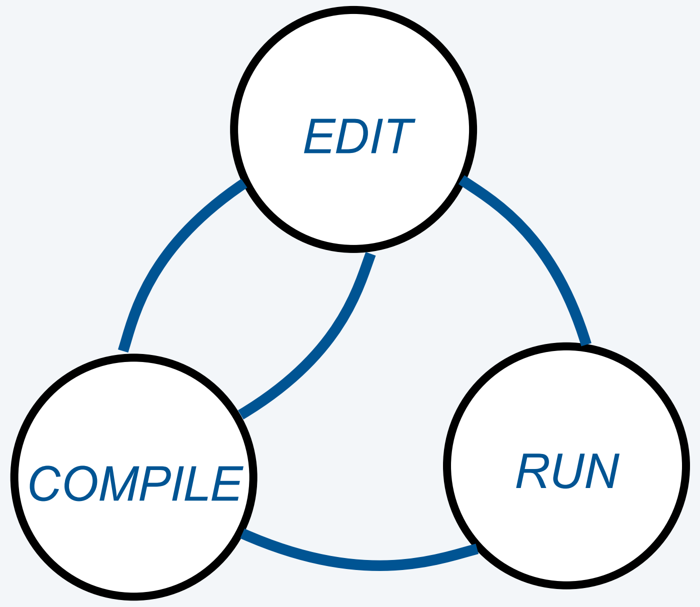
Debugging

is 99% of program development in any programming language, *even for experts.*

Bug: A mistake in a program.

Debugging: The process of eliminating bugs.

You will make many mistakes as you write programs. It's normal.



> *“ As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs. ”*

Impossible ideal: "Please compile, execute, and debug my program."

Why is this impossible? Stay tuned.

Bottom line: Programming is primarily a process of finding and fixing mistakes.

Debugging

is challenging because conditionals and loops *dramatically increase* the number of possible outcomes.

program structure	*no loops*	*n conditionals*	*1 loop*
number of possible execution sequences	1	2^n	no limit

Most programs contain numerous conditionals and loops, with nesting.

Good news. Conditionals and loops provide structure that helps us understand our programs.

Old and low-level languages have a goto statement that provides arbitrary structure. Eliminating gotos was controversial until Edsgar Dijkstra published the famous note "Goto considered harmful" in 1968.

> *“ The quality of programmers is a decreasing function of the number of goto statements in the programs they produce. ”*

– Edsgar Dijkstra

Debugging a program: a running example

Problem: Factor a large integer *n*.

Application: Cryptography.

Surprising fact: Security of internet commerce depends on difficulty of factoring large integers.

Method

- Consider each integer *i* less than *n*
- While *i* divides n evenly

    Print *i* (it is a factor of *n*).

    Replace *n* with *n*/*i* .

Rationale:

1. Any factor of *n*/*i* is a factor of *n*.
2. *i* may be a factor of *n*/i.

3,757,208 = 2 × 2 × 2 × 7 × 13 × 13 × 397

98 = 2 × 7× 7

17 = 17

11,111,111,111,111,111 = 2,071,723 × 5,363,222,357

```
import stdio
import sys
def main():
    n = int(sys.argv[1])

    i = 0
    while i <= n:
        while n % i == 0
            # Cast out and write factor.
            n //= i
            stdio.write(str(i) + ' ')
        i += 1
if __name__ == '__main__': main()
```

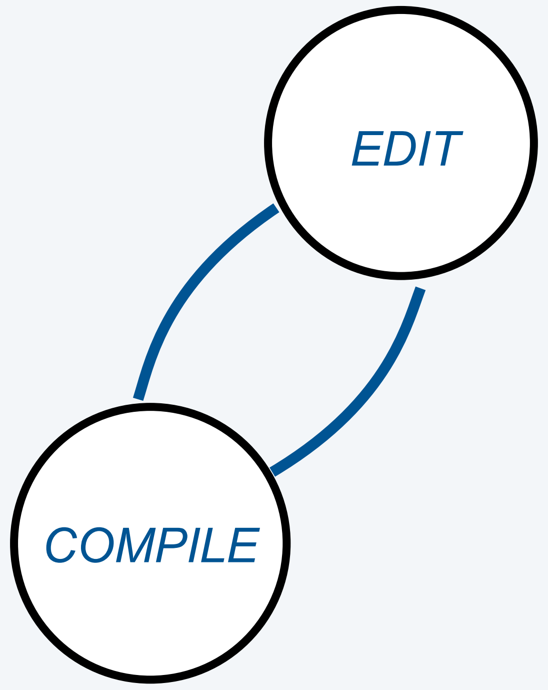
This program has bugs!

Debugging a program: syntax errors

Is your program a legal Python program?

- Python compiler can help you find out.
- Find the *first* compiler error (if any).
- Repeat.
- Result: An executable file

```
% python factors.py 5
File "factors.py", line 13
    while n % i == 0
                        ^
SyntaxError: invalid syntax
```



Trying to tell a computer what to do

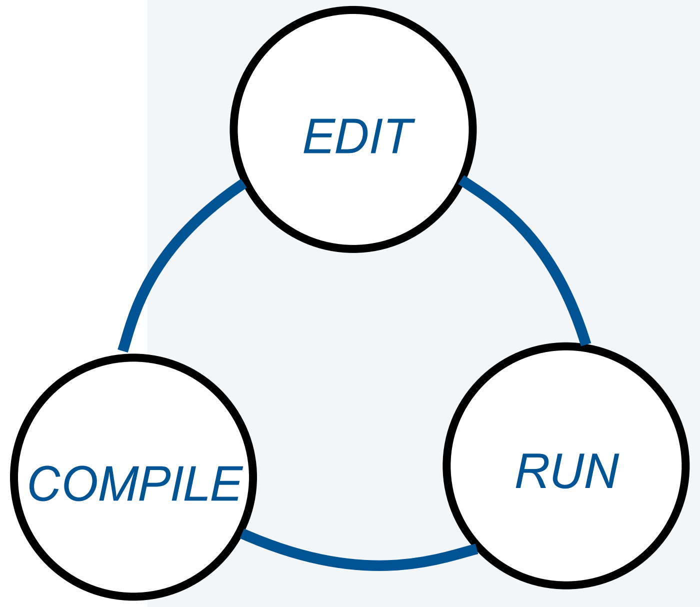
```
import stdio
import sys
def main():
    n = int(sys.argv[1])

    i = 0
    while i <= n:
        while n % i == 0:
            # Cast out and write factor.
            n //= i
            stdio.write(str(i) + ' ')
        i += 1



if __name__ == '__main__': main()
```

This legal program still has bugs!

Debugging a program: runtime and semantic errors

Does your legal Python program do what you want it to do?

- You need to run it to find out.
- Find the *first* runtime error (if any).
- Fix and repeat.



```
% python3 factors.py

Traceback (most recent call last):
  File "factors.py", line 25, in <module>
    if __name__ == '__main__': main()
  File "factors.py", line 9, in main
    n = int(sys.argv[1])
IndexError: list index out of range
```

oops, need argument

you will see this message!

```
% python factors.py 98
2 7 7%
```

98 = 2 × 7× 7 ✓

```
import stdio
import sys
def main():
    n = int(sys.argv[1])

    i = 2
      le i <= n:
        while n % i == 0:
            # Cast out and write factor.
            n //= i
            stdio.write(str(i) + ' ')
        i += 1



if __name__ == '__main__': main()
```

need to start at 2 since 0 and 1 are not factors

This working program still has bugs!

Debugging a program: testing

Does your legal Python program *always* do what you want it to do?

- You need to test on many types of inputs it to find out.
- Add trace code to find the first error.
- Fix the error.
- Repeat.

```
% python factors.py 98
2 7 7%
```

need newline

```
% python factors.py 5
TRACE 2 5
TRACE 3 5
TRACE 4 5
TRACE 5 5
5
```

```
import stdio
import sys
def main():
    n = int(sys.argv[1])

    i = 2
    while i <= n:
        stdio.writeln('TRACE '+str(i)+' '+str(n))
        while n % i == 0:
            # Cast out and write factor.
            n //= i
            stdio.write(str(i) + ' ')
        i += 1
    stdio.writeln()
if __name__ == '__main__': main()
```

Debugging a program: testing

Does your legal Python program *always* do what you want it to do?

- You need to test on many types of inputs it to find out.
- Add trace code to find the first error.
- Fix the error.
- Repeat.

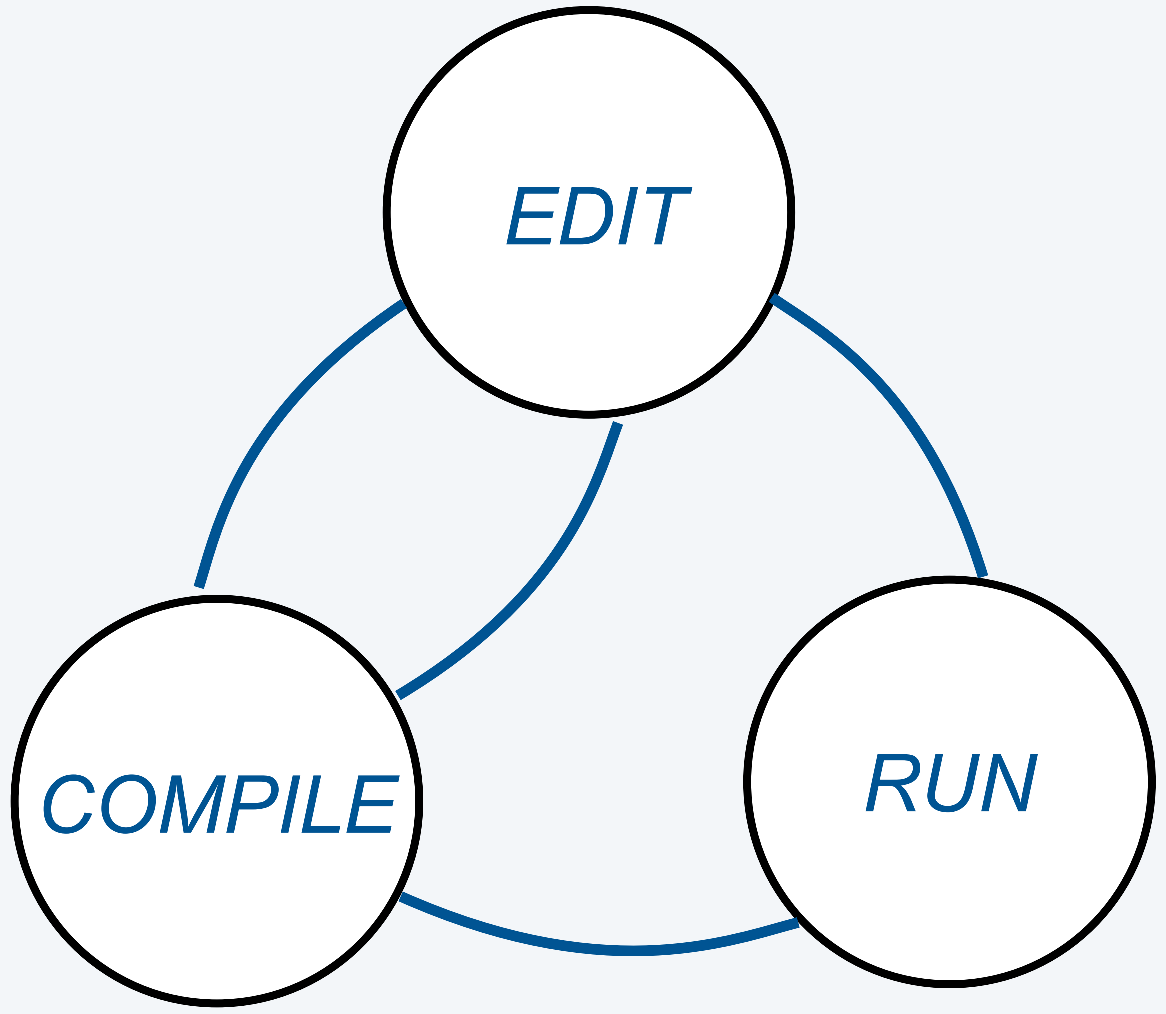
```
% python factors.py 5
TRACE 2 5
TRACE 3 5
TRACE 4 5
TRACE 5 5
% python factors.py 5
5
% python factors.py 6
2 3
% python factors.py 98
2 7 7
% python factors.py 3757208
2 2 2 7 13 13 397
```

```
import stdio
import sys
def main():
    n = int(sys.argv[1])

    i = 2
    while i <= n:
        #stdio.writeln('TRACE '+i+' '+factor)
        while n % i == 0:
            # Cast out and write factor.
            n //= i
            stdio.write(str(i) + ' ')
        i += 1
    stdio.writeln()
if __name__ == '__main__': main()
```

Program development in Python

is a three-step process, *with feedback*

1. EDIT your program
   - Create it by typing on your computer's keyboard.
   - Result: a text file such as `factors.py`.
2. COMPILE it to create an executable file
   - Use the Python compiler
   - Result: Python bytecode file
   - Mistake? Go back to 1. to fix and recompile.
3. RUN your program
   - Use the Python interpreter.
   - Result: your program’s output.
   - Mistake? Go back to 1. to fix, recompile, and execute



Debugging a program: where we left off

Problem: Factor a large integer *N*.

Application: Cryptography.

Method

- Consider each integer *i* less equal than *N*
- While *i* divides N evenly

  print *i* (it is a factor of *N*)

  replace *N* with *N*/*i* .

```
import stdio
import sys
def main():
    n = int(sys.argv[1])

    i = 2
    while i <= n:
        while n % i == 0:
            # Cast out and write factor.
            n //= i
            stdio.write(str(i) + ' ')
        i += 1
    stdio.writeln()
if __name__ == '__main__': main()
```

```
% python factors.py 98
2 7 7
% python factors.py 3757208
2 2 2 7 13 13 397
```

✓

Debugging a program: performance

Is your working Python program fast enough to solve your problem?

- You need to test it on increasing problem sizes to find out.
- May need to change the algorithm to fix it.

change the algorithm: no need to check when i*i>n since all smaller factors already checked

Method

- Consider each integer *i*i* less than *N*
- While *i* divides n evenly

  print *i* (it is a factor of *n*)

  replac

```
% python factors.py 11111111
11 73 101 137
% python factors.py 11111111111
21649 513239
% python factors.py 11111111111111
11 239 4649 909091
% python factors.py 11111111111111111
2071723
```

```
import stdio
import sys
def main():
    n = int(sys.argv[1])

    i = 2
    while i*i <= n:
        while n % i == 0:
            # Cast out and write factor.
            n //= i
            stdio.write(str(i) + ' ')
        i += 1
    stdio.writeln()
if __name__ == '__main__': main()
```

No need to check all i < n

5363222357 ← immediate

Debugging a program: performance analysis

Q. How large an integer can I factor?

```
% python factors.py 9201111169755555703
9201111169755555703
```

digits in largest factor	*i <= N*	*i*i <= N*
3	instant	instant
6	instant	instant
9	77 seconds	instant
12	21 hours†	instant
15	2.4 years†	2.7 seconds
18	2.4 millenia†	92 seconds

† estimated, using analytic number theory

```
import stdio
import sys
def main():
    n = int(sys.argv[1])

    i = 2
    while i*i <= n:
        while n % i == 0:
            # Cast out and write factor.
            n //= i
            stdio.write(str(i) + ' ')
        i += 1
    stdio.writeln()
if __name__ == '__main__': main()
```

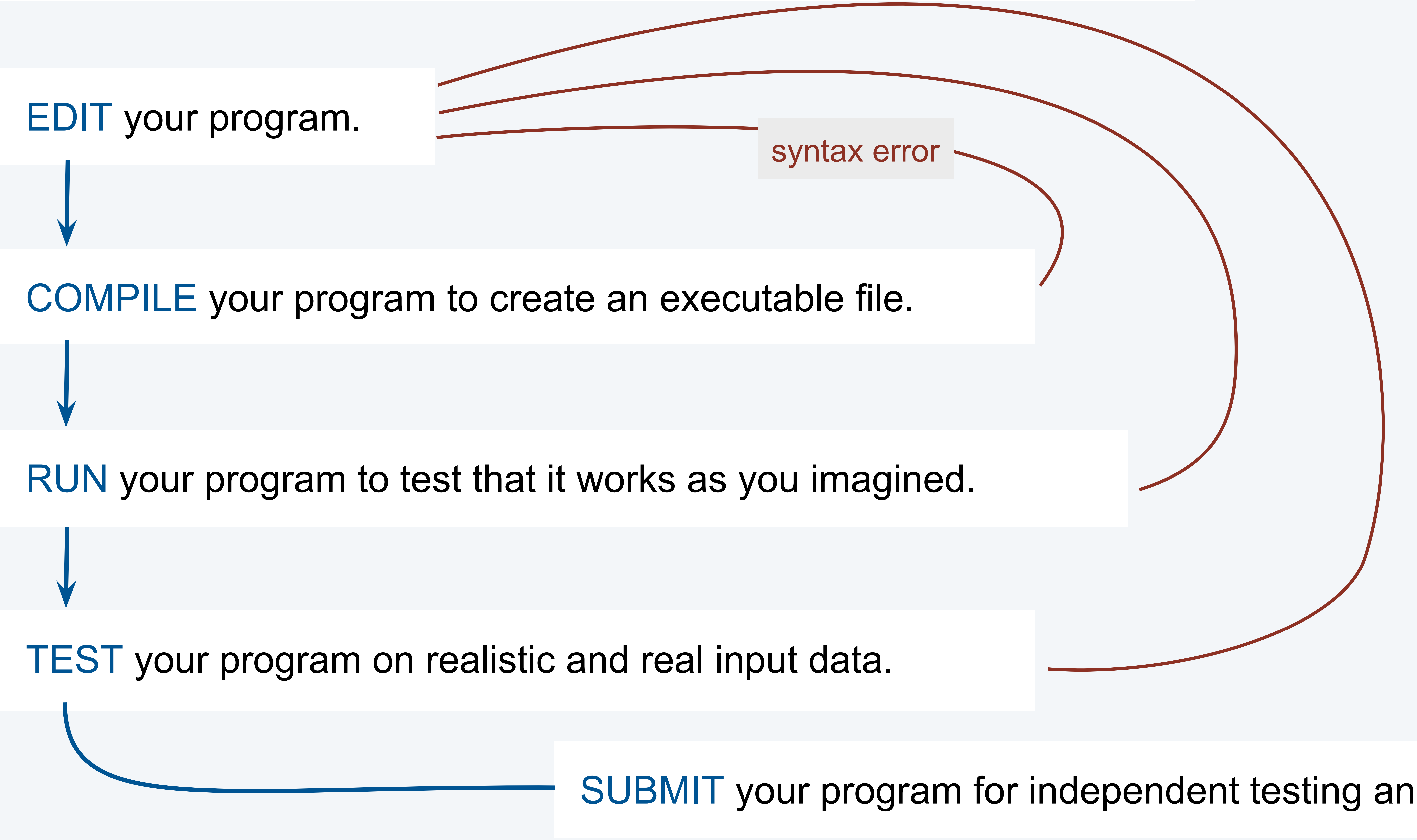
Lesson. Performance matters!

experts are still trying to develop better algorithms for this problem

Note. Internet commerce is still secure: it depends on the difficulty of factoring 200-digit integers.

Debugging your program: summary

Program development is a *four*-step process, with feedback.



EDIT your program.

COMPILE your program to create an executable file.

RUN your program to test that it works as you imagined.

TEST your program on realistic and real input data.

SUBMIT your program for independent testing and approval.

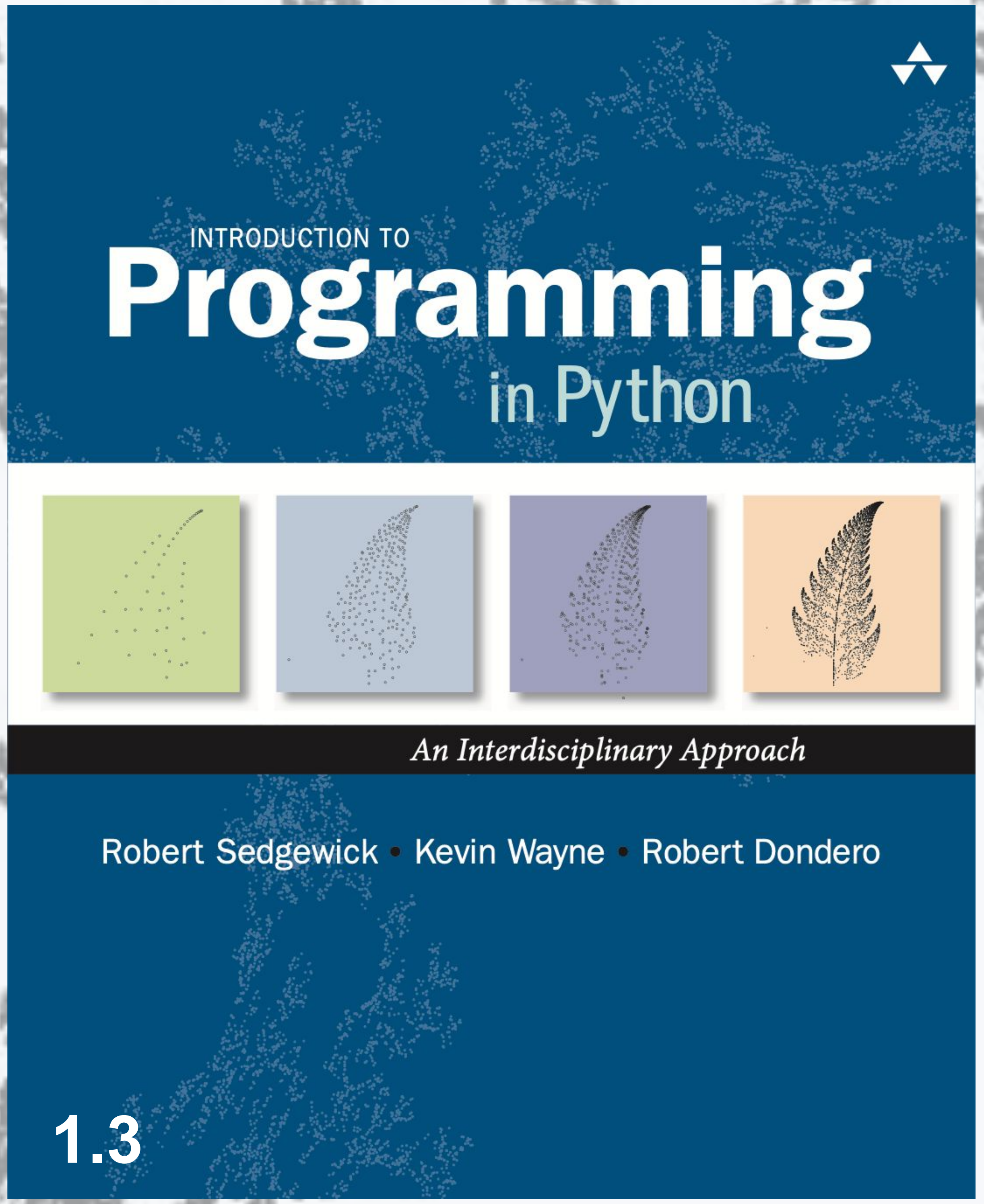
Telling a computer what to do when you know what you're doing

Image sources

```
http://playatlantic.com/sites/default/files/good%20looking%20girl%20on%20laptop_0.jpg

http://en.wikipedia.org/wiki/Edsger_W._Dijkstra#mediaviewer/File:Edsger_Wybe_Dijkstra.jpg

http://pixabay.com/en/hourglass-clock-timer-sand-time-152090/
```

INTRO TO PROGRAMMING
IN PYTHON

SEDGEWICK · WAYNE · DONDERO



INTRODUCTION TO
Programming
in Python

An Interdisciplinary Approach

Robert Sedgewick · Kevin Wayne · Robert Dondero

1.3

https://introcs.cs.princeton.edu/python

Title Text

Slides ported to Python by James Pope
Last updated on 2023/10/20 08:40