FIT1045: Algorithms and Programming Fundamentals in Python

Lecture 4

Loops and Euclid's Algorithm

Recap

https://flux.qa

Clayton: AXXULH
Malaysia: LWERDE

Boolean expressions:

- Can you translate the following sentence to a Boolean expression?

A good fruit salad contains one main fruit which can be either oranges or melons and a second fruit that can be either strawberries or pineapple, but it should never contain avocado.

Boolean operators have precedence:

- Which parentheses can be avoided?

((fruit1=='orange') and (fruit2=='orange')) or ((fruit1=='apple') and (fruit2=='apple'))

This lecture

Learn about loops to implement our first textbook algorithm in Python

Learning outcomes

- 2 (choose and implement appropriate problem solving strategies in Python)
- 5 (determine limitations of algorithms)

Concrete goal: An efficient algorithm for computing the greatest common divisor

Where am I?

1. Greatest Common Divisor
2. While loops
3. Euclid's Algorithm

Motivation: simplifying fractions

$$\frac{18}{24} \xrightarrow{/\mathrm{gcd}(18,24)} \frac{3}{4}$$

$$\frac{18}{24} = \frac{3}{4}$$

$$\frac{18480831109}{9231418071} = ?$$

<u>Greatest Common Divisor Problem</u>

Input: two positive integers m and n

Output: greatest common divisor, $\gcd(m, n)$

https://flux.qa

Clayton: AXXULH
Malaysia: LWERDE

Let's find an algorithm

Greatest Common Divisor Problem

Input: two positive integers *m* and *n*

Output: greatest common divisor, gcd(*m*, *n*)

Observations

- the greatest possible common divisor is the smaller of the two numbers, e.g. gcd(178, 89) = 89
- the smallest possible divisor is 1, e.g. gcd(97, 53) = 1
- we are after the ***greatest*** divisor, e.g. gcd(24, 18) = 6, not 1, 2, or 3

"Brute force" Algorithm

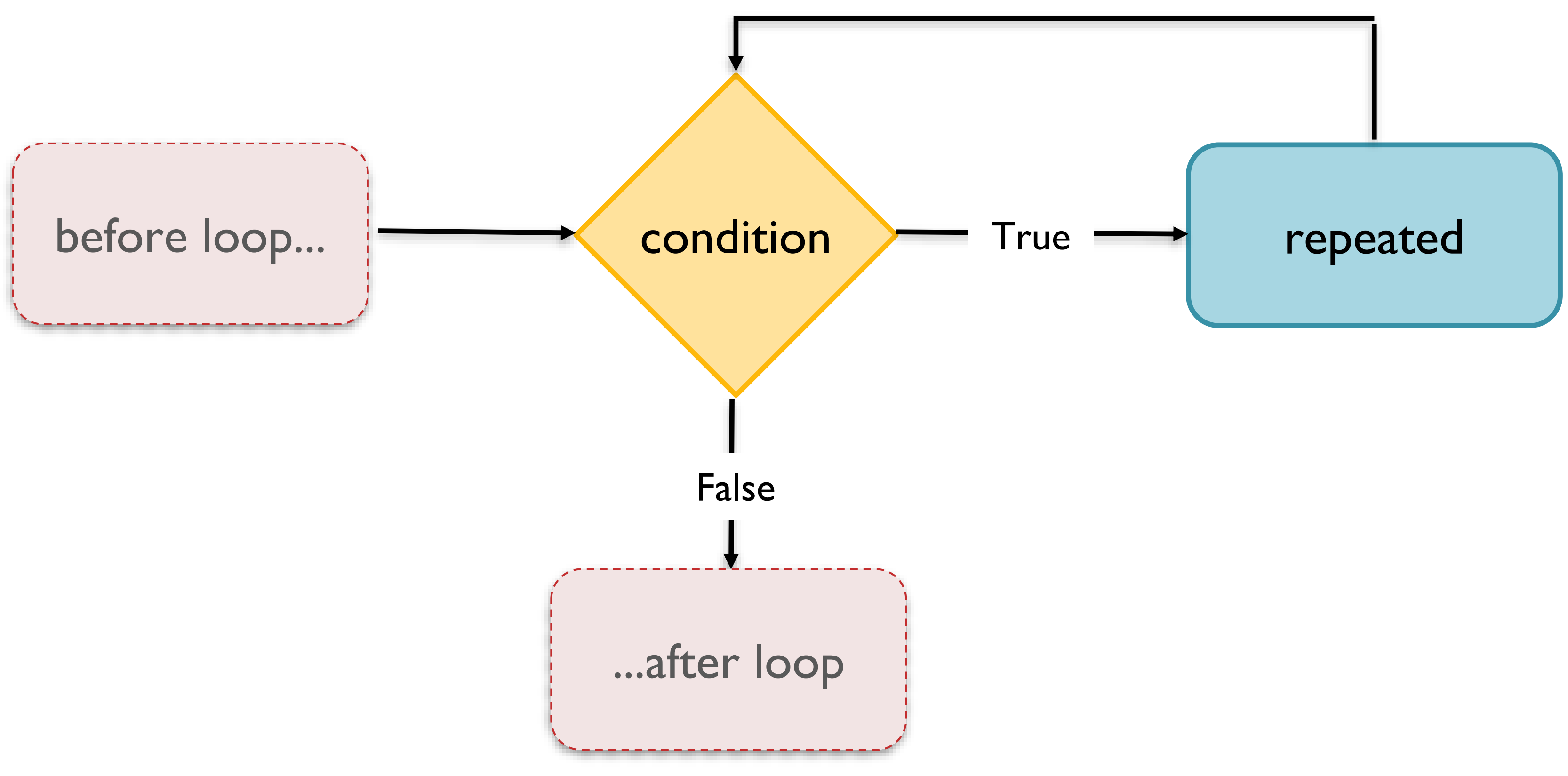
check all integers between $\min(m, n)$ and 1 (from big to small), output first common divisor encountered

How to “check every integer”?

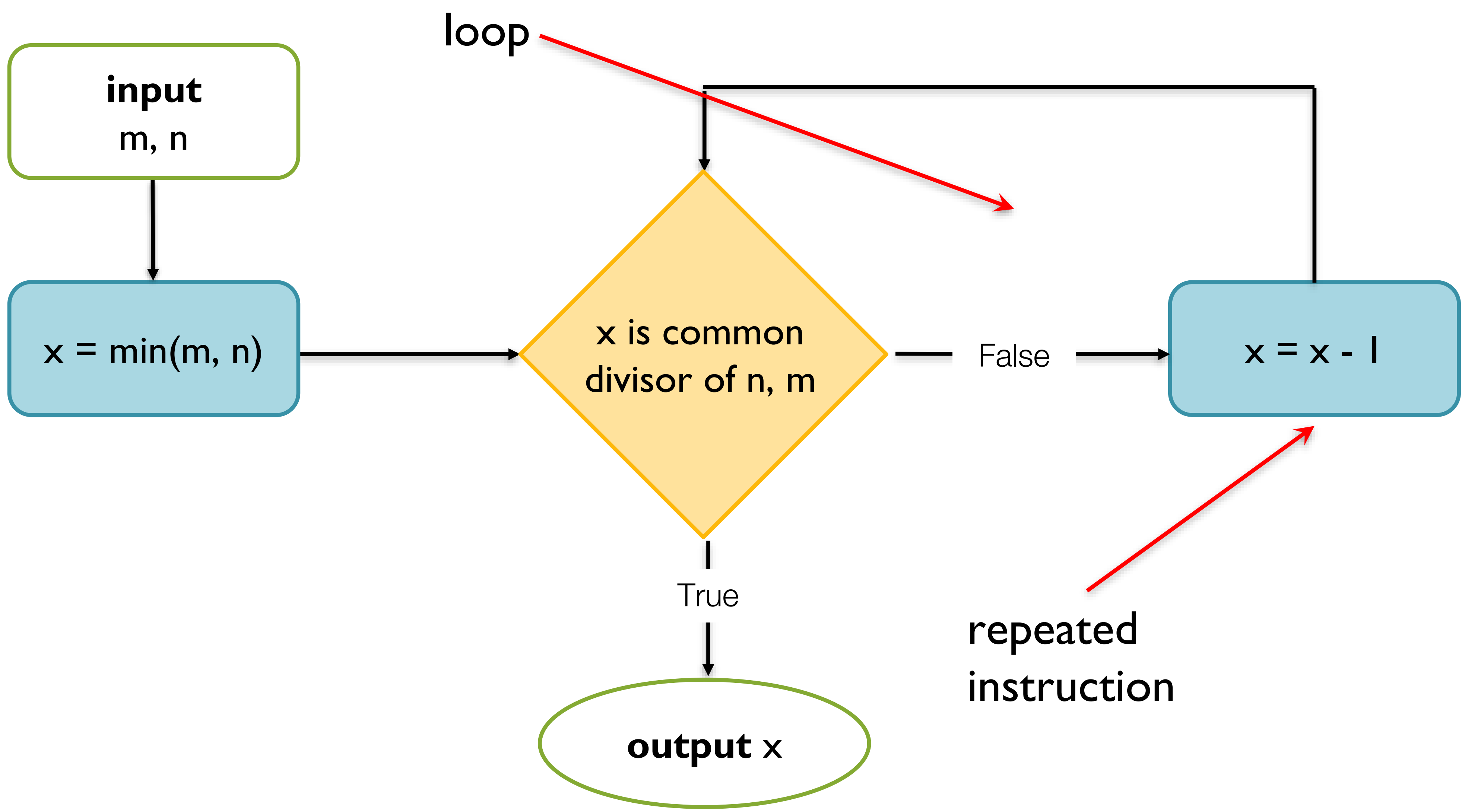
Observations

- Depending on input there can be an arbitrary number of integers to check
- Program will always have only a fixed number of instructions

Need to repeat some instructions many times in a loop.



How to "check every integer"?



"Brute force" Algorithm

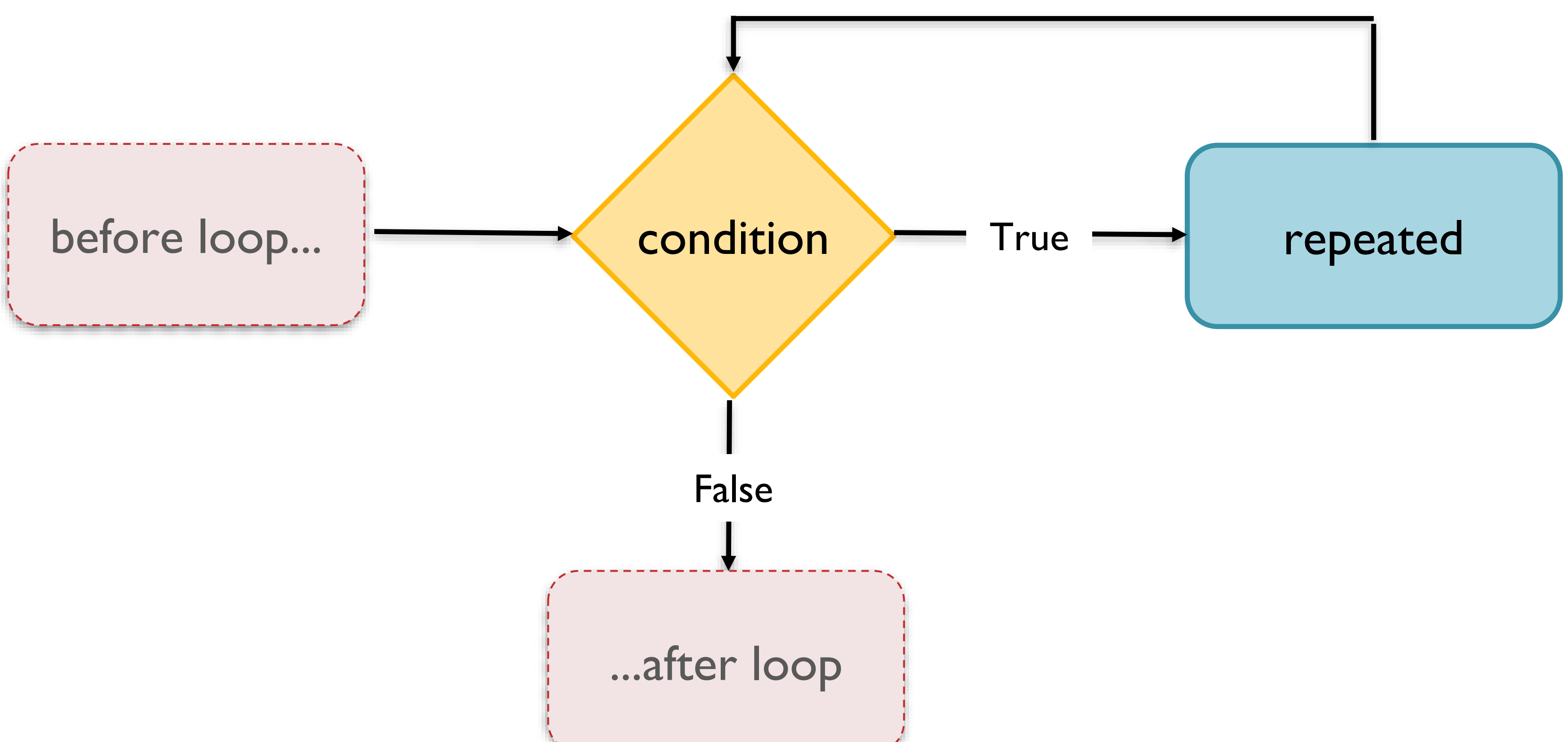
check all integers between min(m, n) and 1 (from big to small), output first common divisor encountered

Where am I?

1. Greatest Common Divisor
2. While loops
3. Euclid's Algorithm

While statement in Python for loopy control flows

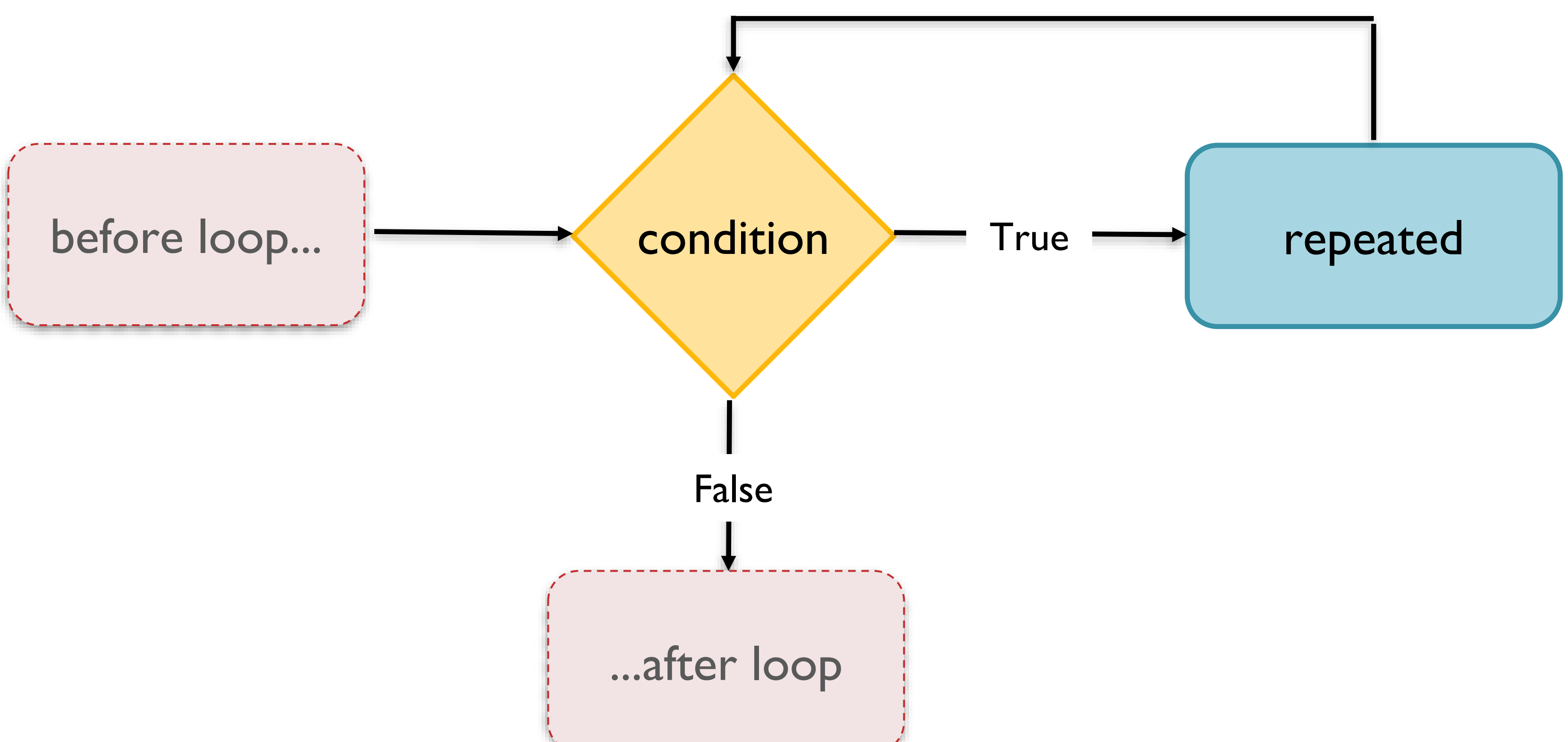
```
# before loop
# ...
#
while condition:
    # repeated
    # ...
    #
# after loop
# ...
#
```



Example: summing first n integers

```
def sum_of_first_n_ints(n):
    """
    Input : positive integer n
    Output: sum of pos. integers up to n"""
```

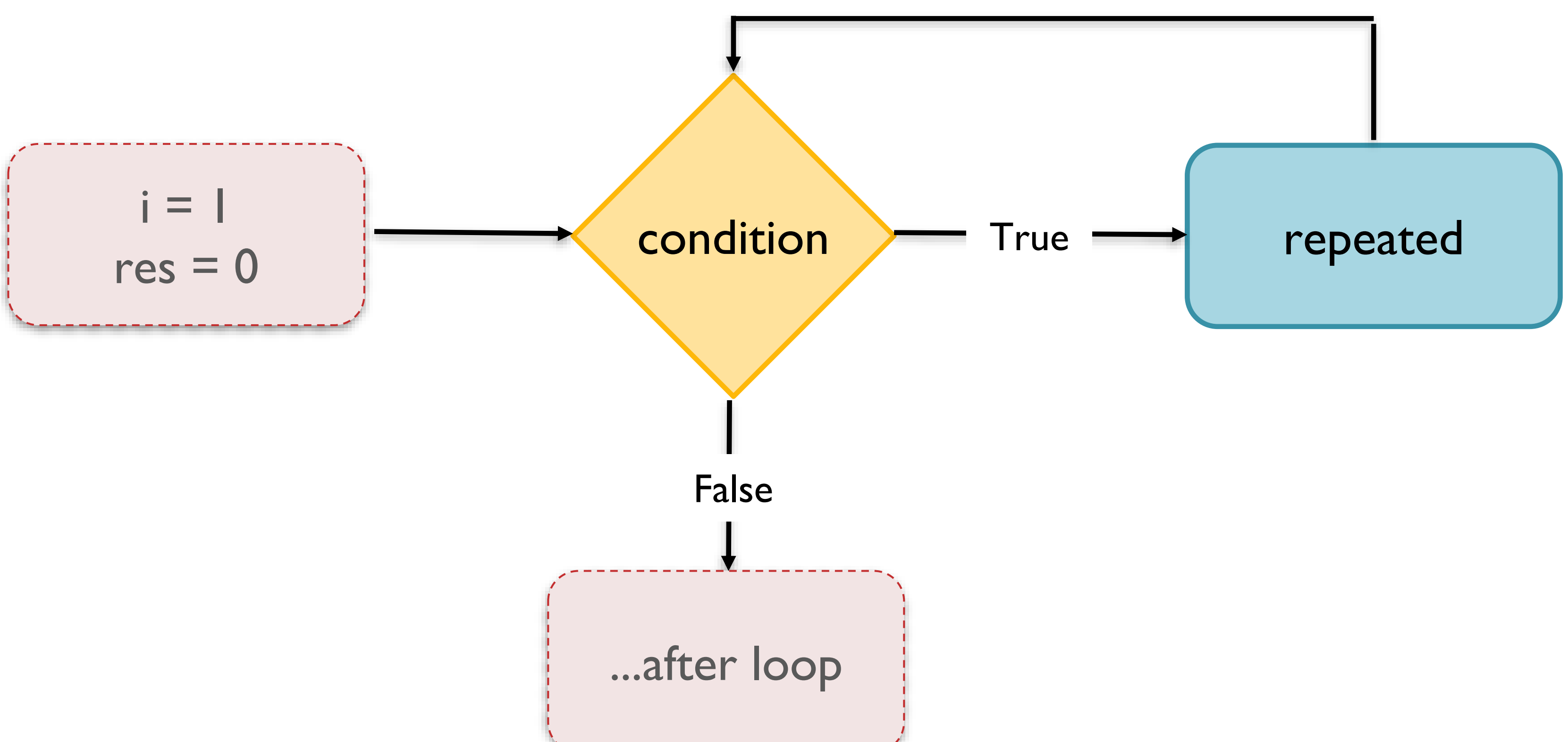
$$1 + 2 + \cdots + n$$
$$= ?$$



Example: summing first n integers

```
def sum_of_first_n_ints(n):
    """
    Input : positive integer n
    Output: sum of pos. integers up to n"""
    i = 1   #iteration variable
    res = 0 #accumulation variable
```

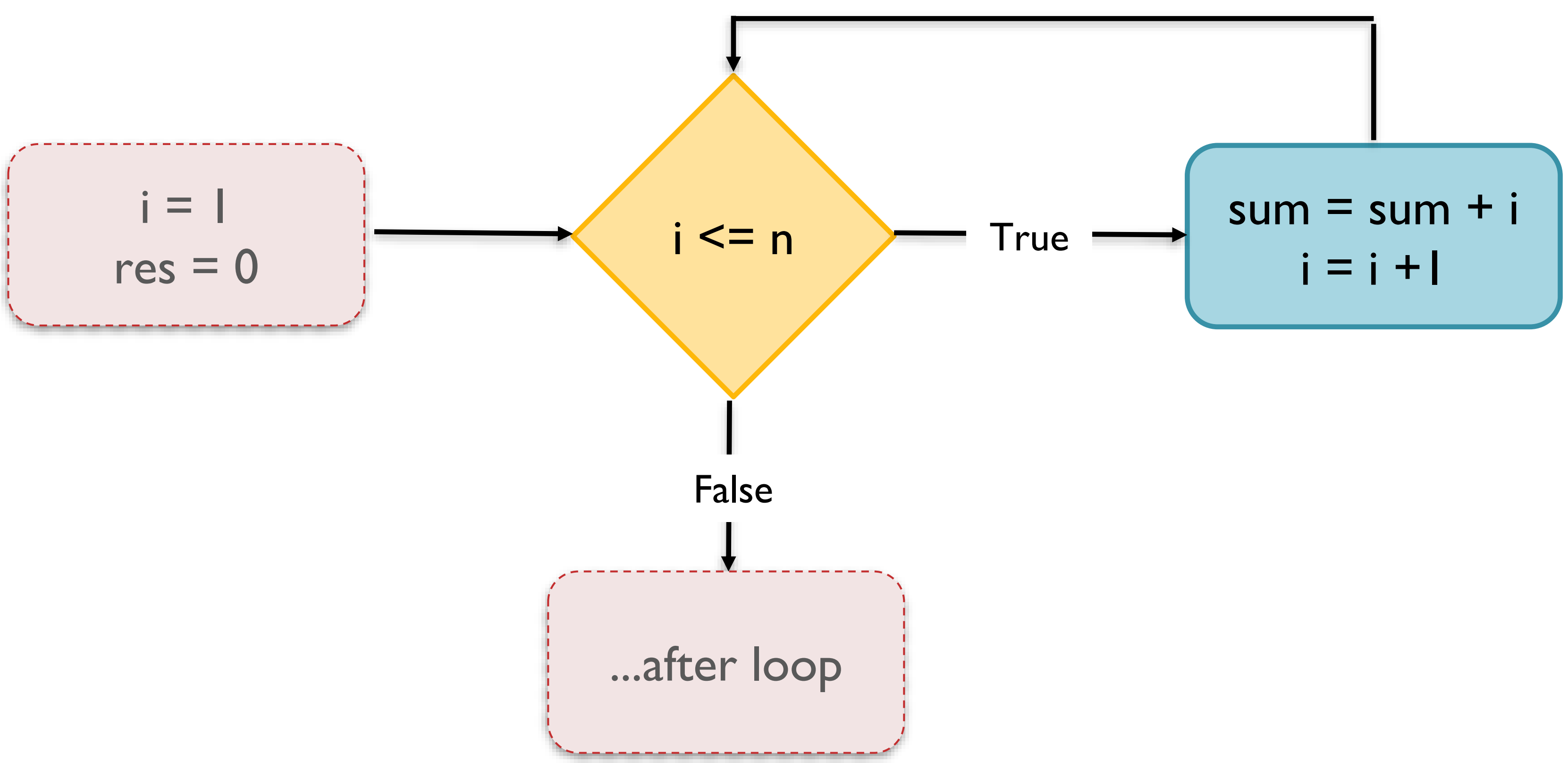
$$1 + 2 + \cdots + n$$
$$= ?$$



Example: summing first n integers

```python
def sum_of_first_n_ints(n):
    """
    Input : positive integer n
    Output: sum of pos. integers up to n"""
    i = 1  #iteration variable
    res = 0 #accumulation variable
    while i <= n:
        res = res + i
        i = i + 1
```

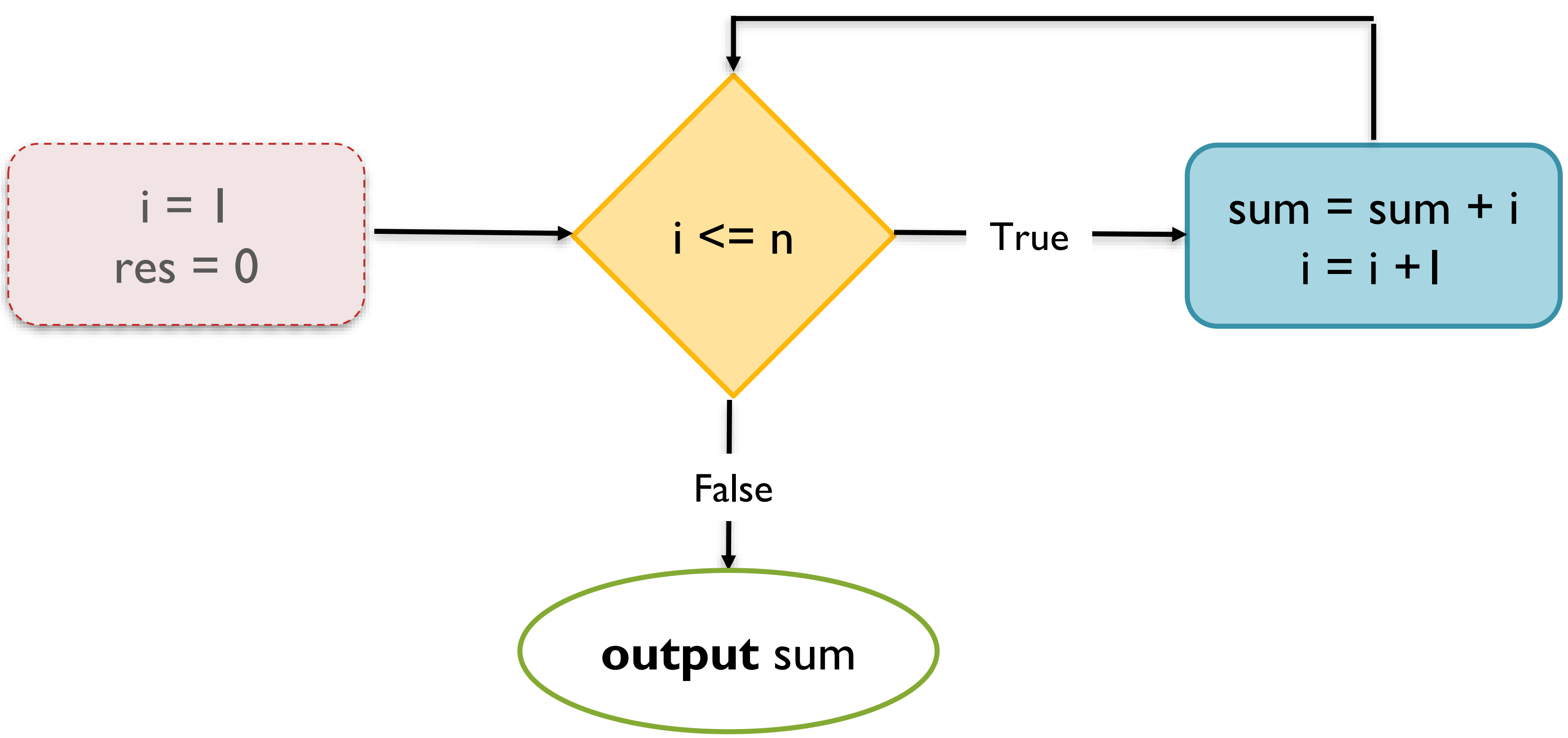
$$1 + 2 + \cdots + n$$
$$= ?$$



Example: summing first n integers

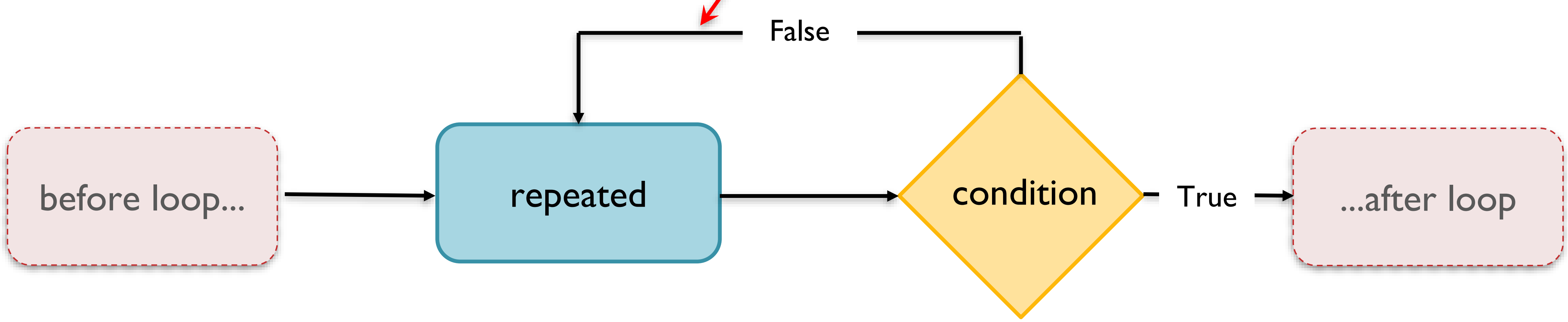
```python
def sum_of_first_n_ints(n):
    """
    Input : positive integer n
    Output: sum of pos. integers up to n"""
    i = 1  #iteration variable
    res = 0 #accumulation variable
    while i <= n:
        res = res + i
        i = i + 1
    return res
```

$$1 + 2 + \cdots + n$$
$$= ?$$

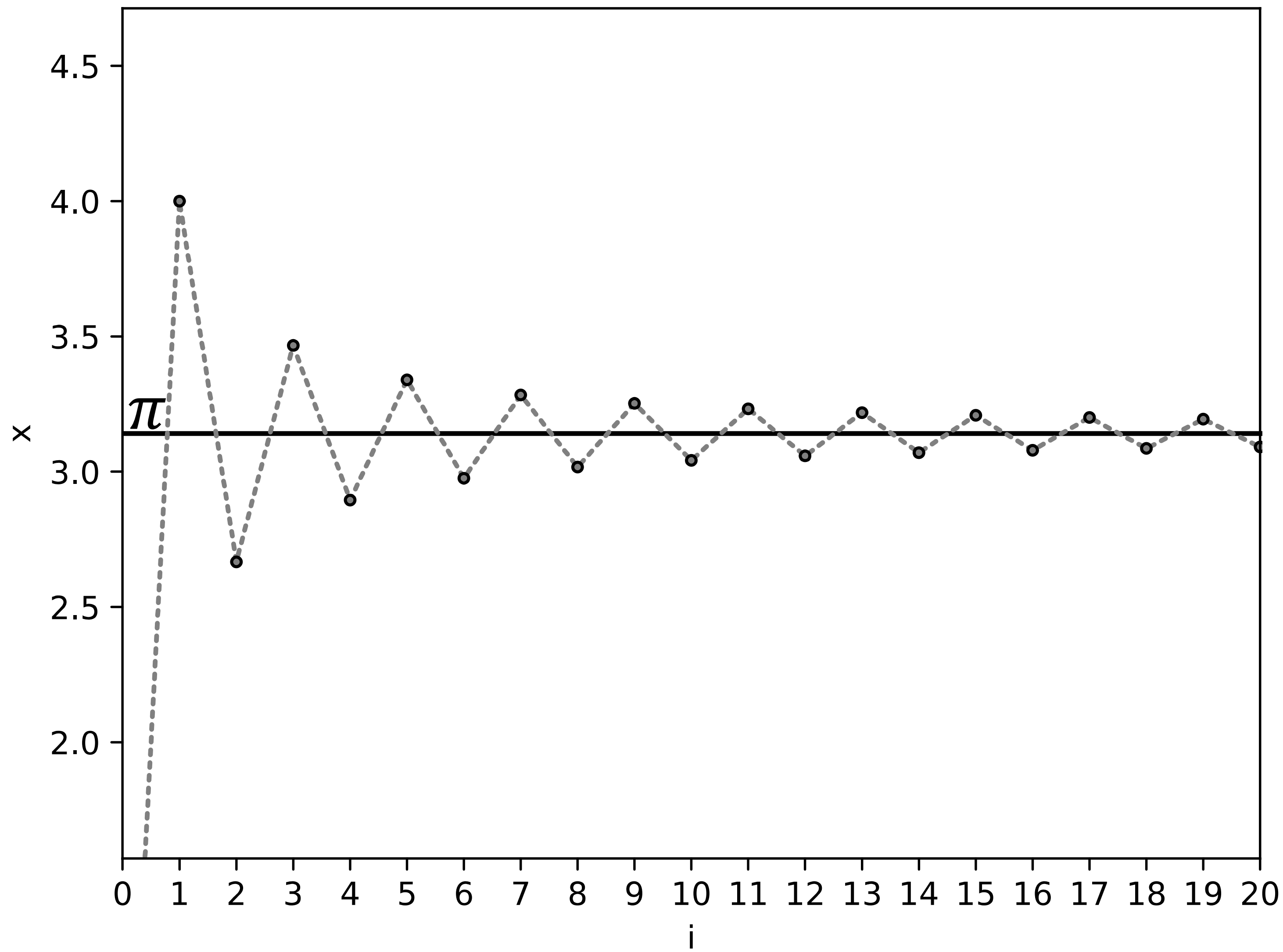


Sometimes we want condition last

instructions will be repeated while the condition is **False**

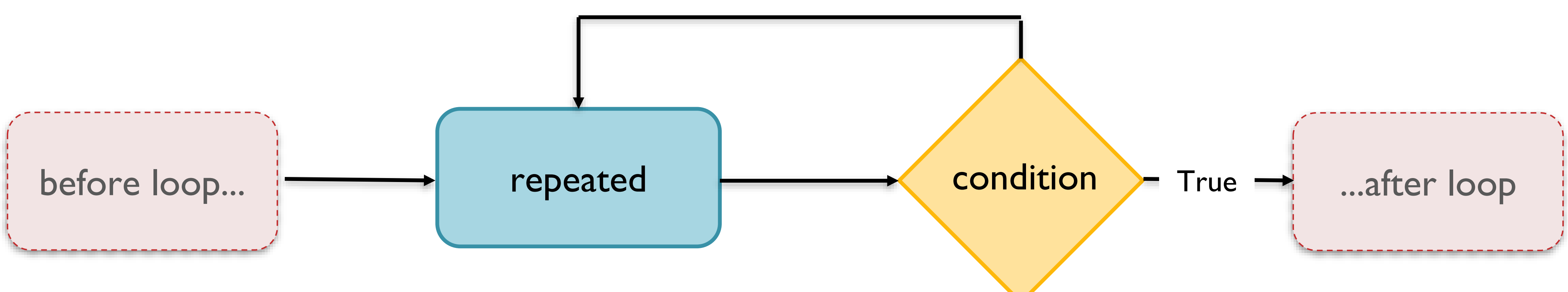


Example: approximating π

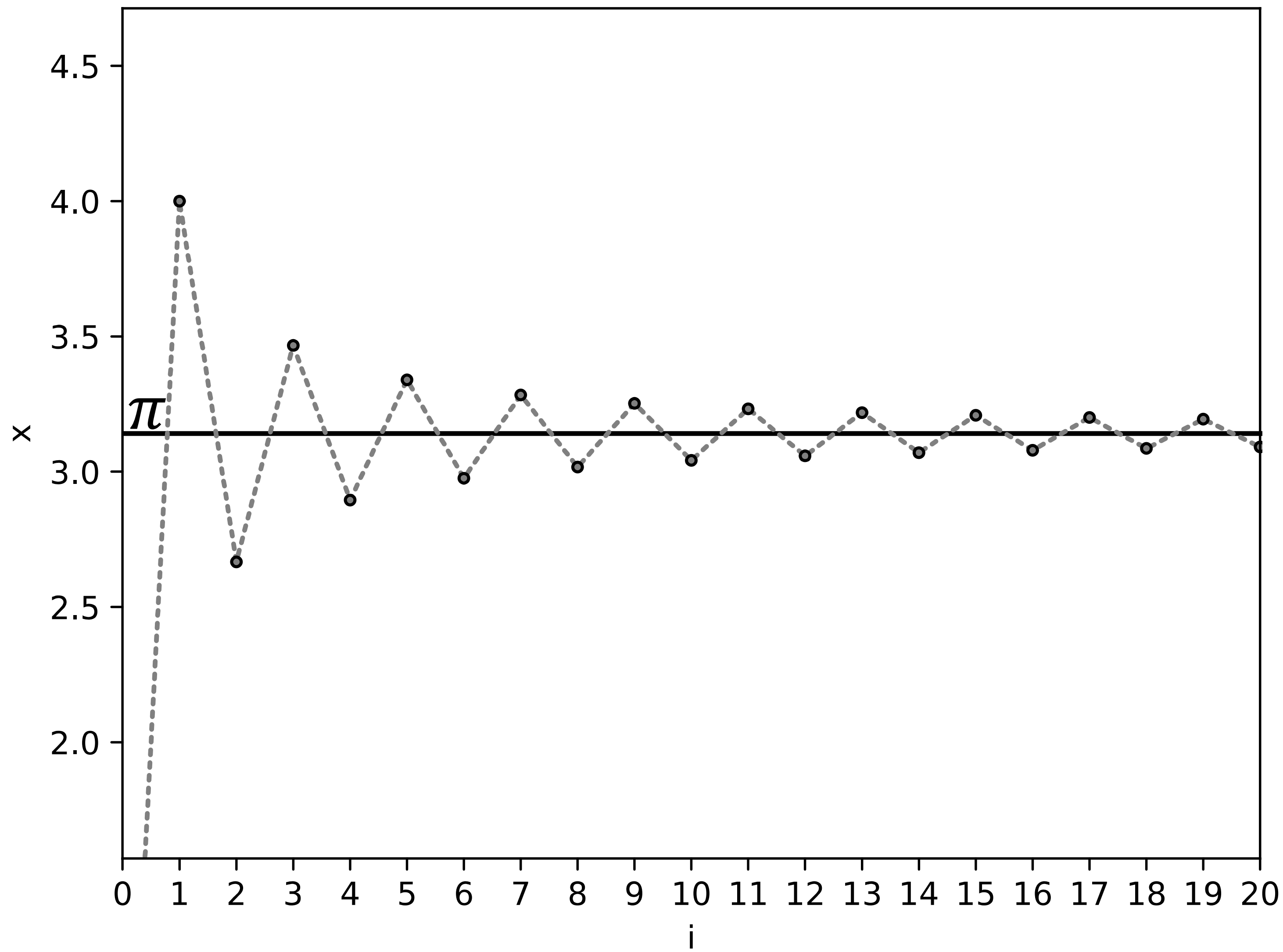


$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} \cdots$$

$$= \sum_{i=1}^{\infty} (-1)^{i+1} \frac{4}{2i-1}$$

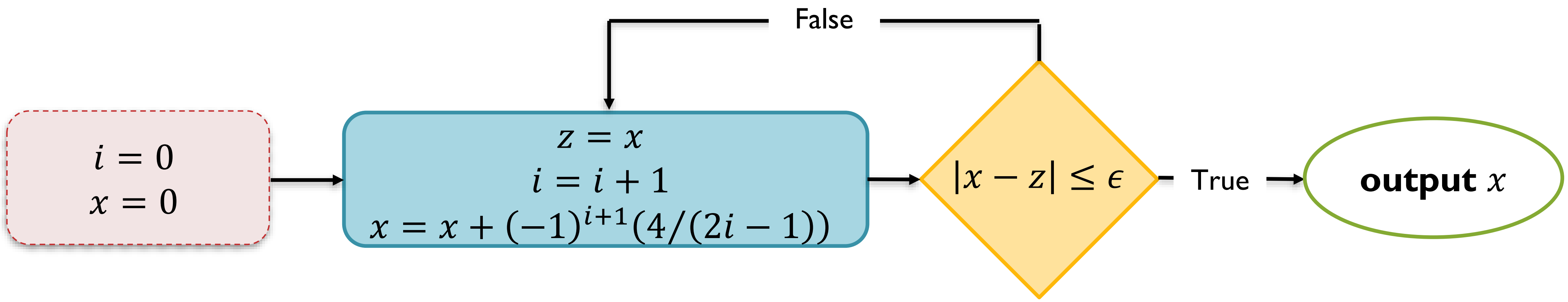


Example: approximating π



$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} \cdots$$

$$= \sum_{i=1}^{\infty} (-1)^{i+1} \frac{4}{2i-1}$$

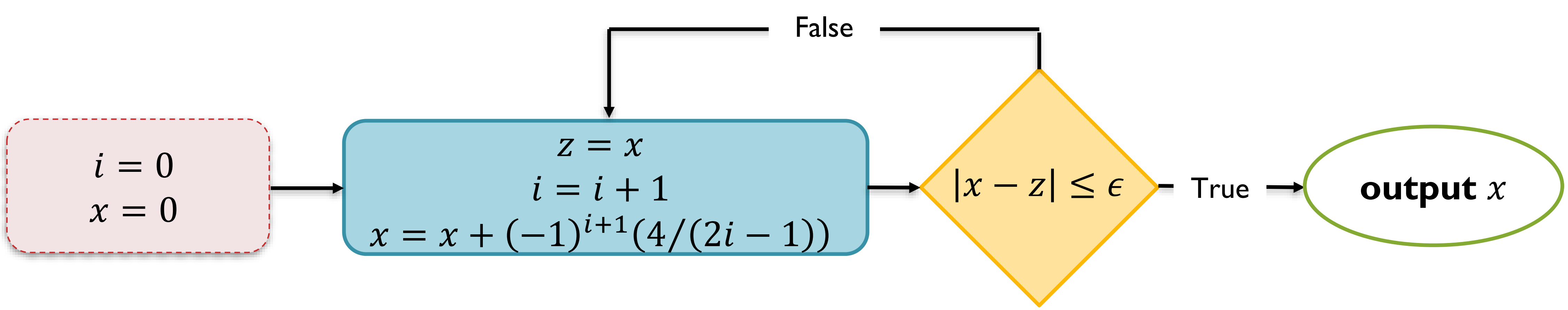


To realise condition at end we can use conditional **break** statement

```
def pi_approximation(eps):
    """
    Input : positive float eps (accuracy)
    Output: approximation x to pi with abs(x-pi)<=eps
    """
    i = 0  #iteration variable
    x = 0  #candidate solution
    while True:
        i += 1
        z = x
        x = x + (-1)**(i+1)*4/(2*i-1)
        # exit loop here if abs(x-z)<=eps

    return x
```

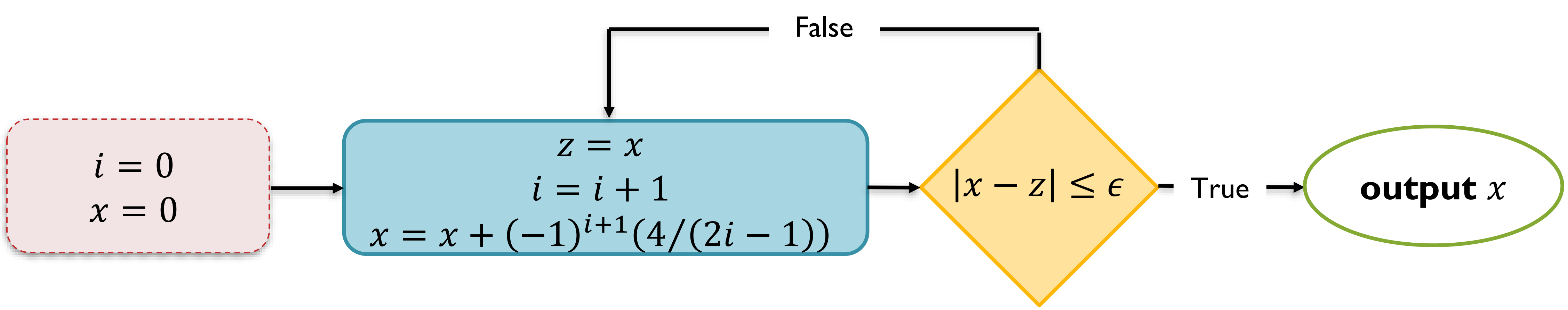
loop never exits here



To realise condition at end we can use conditional **break** statement

```python
def pi_approximation(eps):
    """
    Input : positive float eps (accuracy)
    Output: approximation x to pi with abs(x-pi)<=eps
    """
    i = 0  #iteration variable
    x = 0  #candidate solution
    while True:
        i += 1
        z = x
        x = x + (-1)**(i+1)*4/(2*i-1)
        if abs(x - z) <= eps:
            break
    return x
```

break statement exits surrounding loop



Loops pose a new kind of danger

```
def sum_of_first_n_ints(n):
    """
    Input : positive integer n
    Output: sum of pos. integers up to n"""
    i = 1  #iteration variable
    res = 0
    while i <= n:
        res = res + i
        i = i + 1
    return sum
```

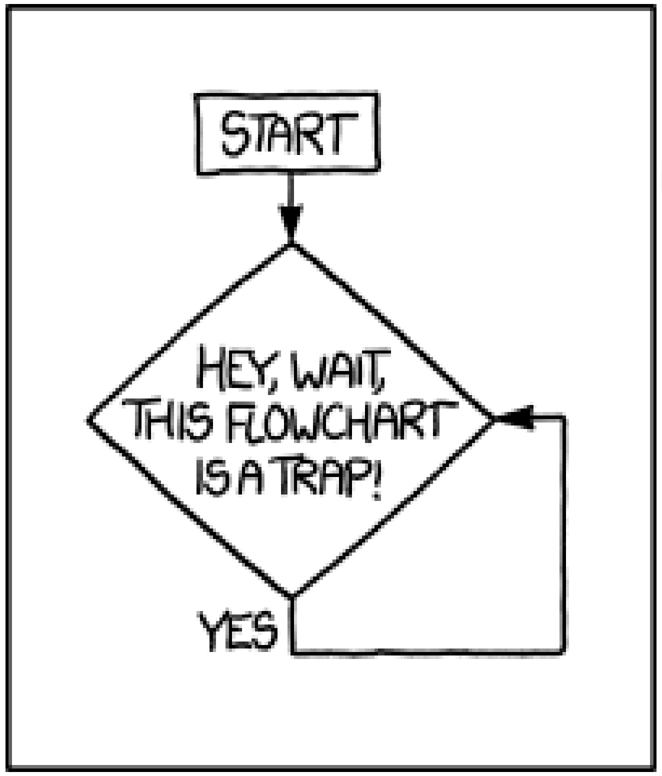
$$1 + 2 + \cdots + n = ?$$

Forgetting just one line results in this flowchart ☺

Everyone in this unit (including staff) will write an infinite loop by accident at some point

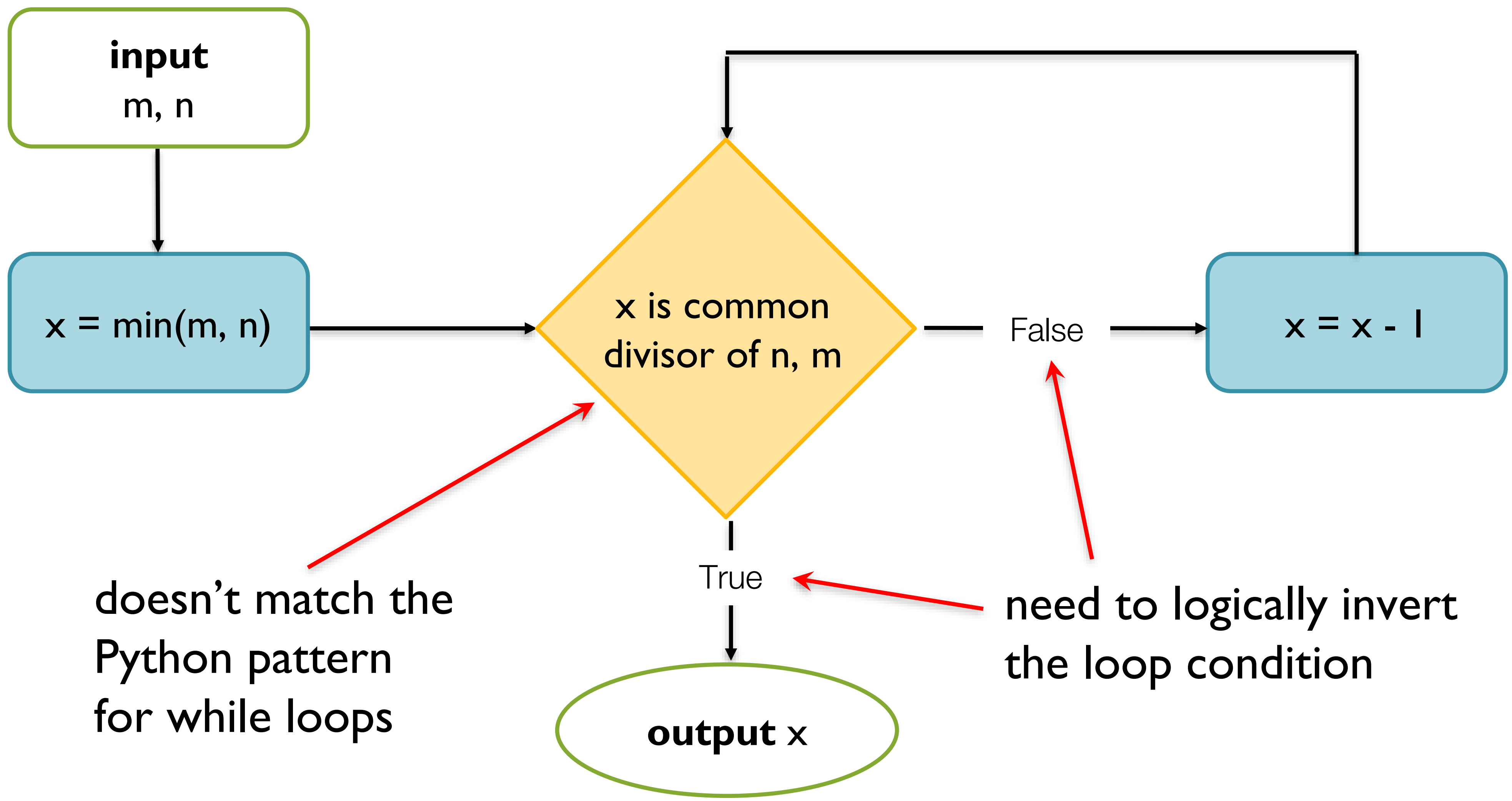
Common mistakes:

- Counter not incremented
- Tautological condition
- Forgot break



https://xkcd.com/1195/

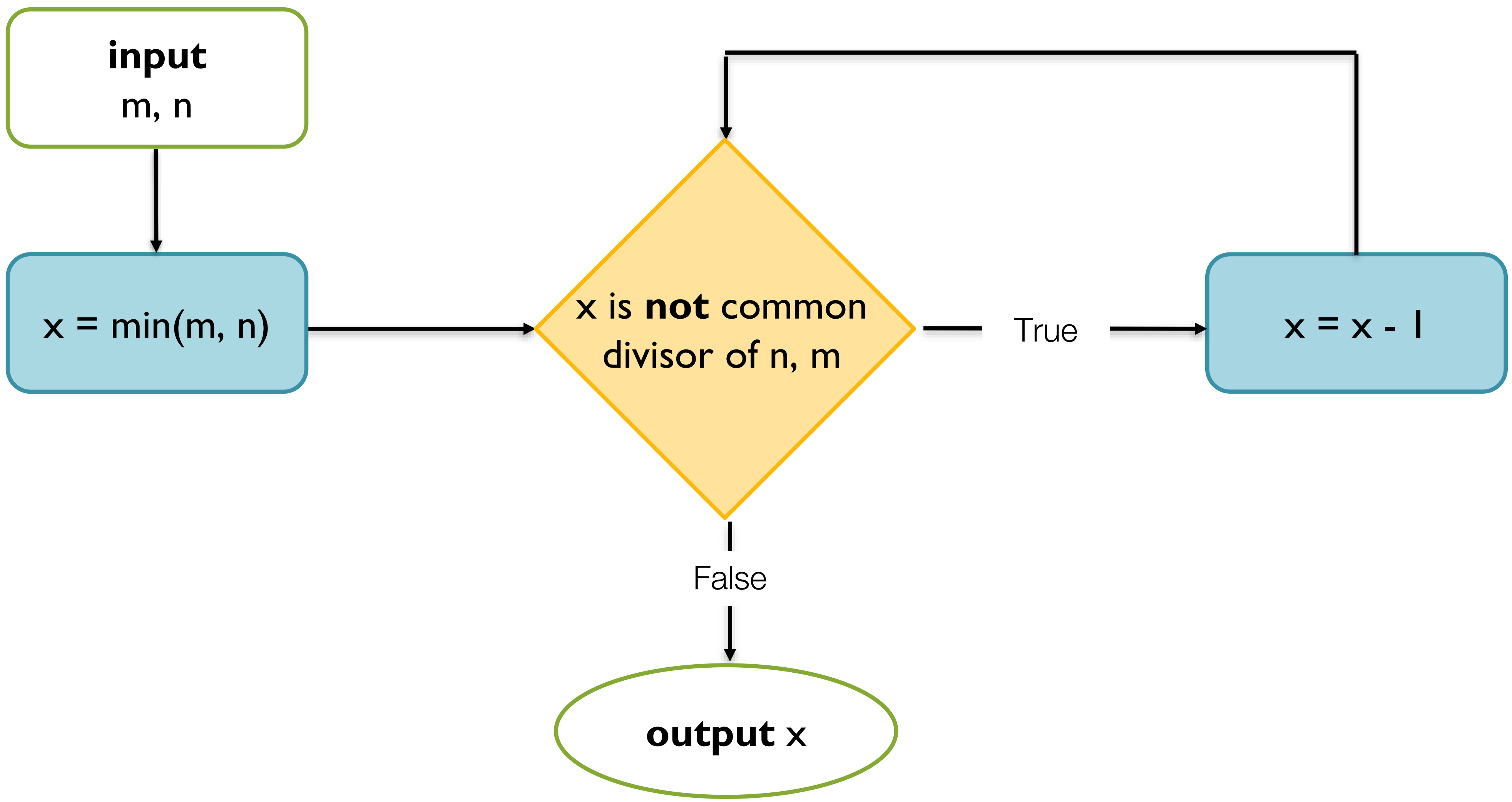
Using **while** to implement brute force GCD algorithm



https://flux.qa

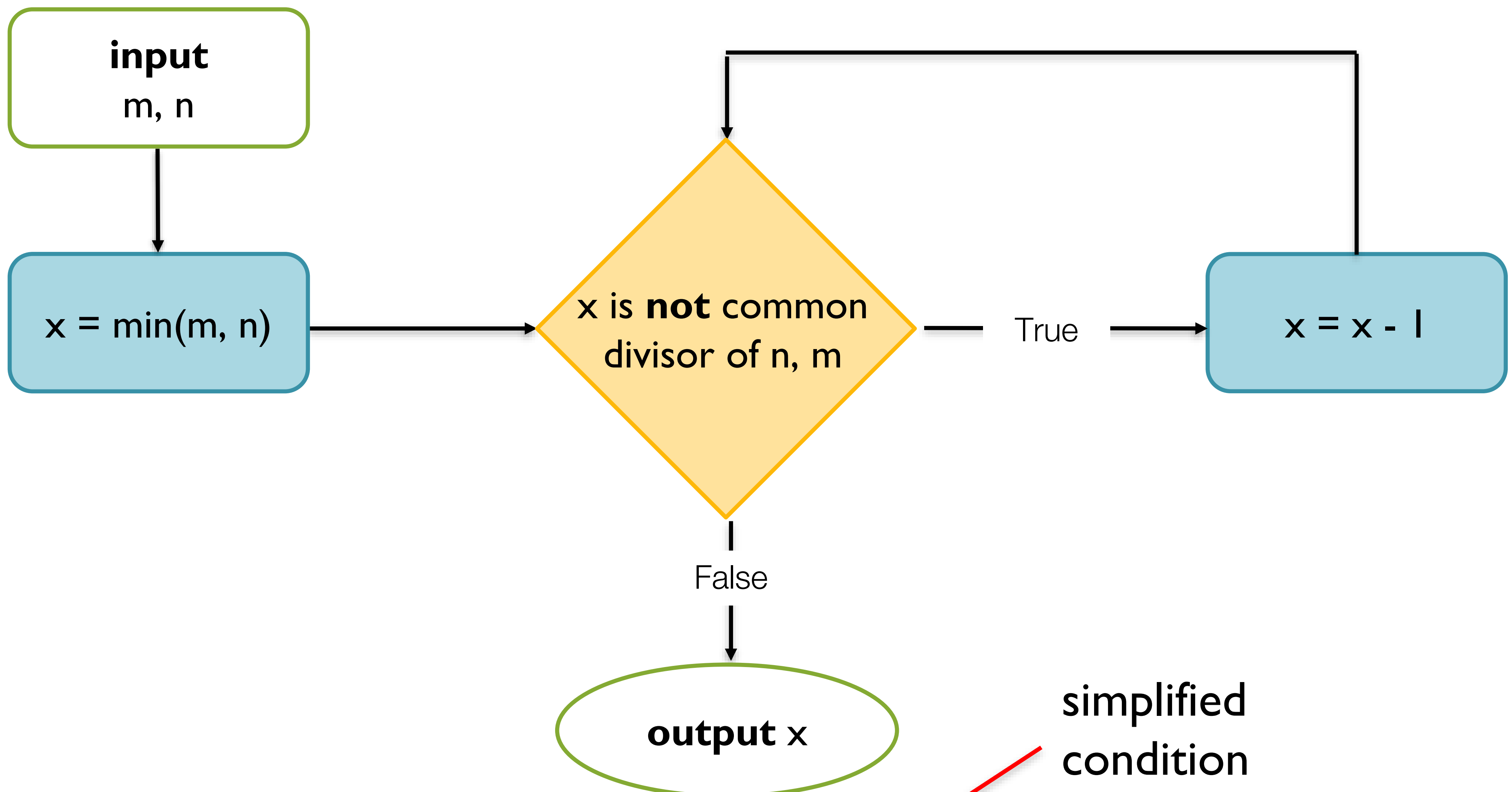
Clayton: AXXULH
Malaysia: LWERDE

Using **while** to implement brute force GCD algorithm




```
def gcd_brute_force(m, n):
    x = min(m, n)
    while not (m % x == 0 and n % x == 0):
        x = x - 1
    return x
```

Using **while** to implement brute force GCD algorithm



simplified condition

```
def gcd_brute_force(m, n):
    x = min(m, n)
    while (m % x != 0) or (n % x != 0):
        x = x - 1
    return x
```

Analysing our GCD algorithm

We have implemented an algorithm to find the GCD of two numbers.

But our algorithm is very inefficient!

Is there a better algorithm?

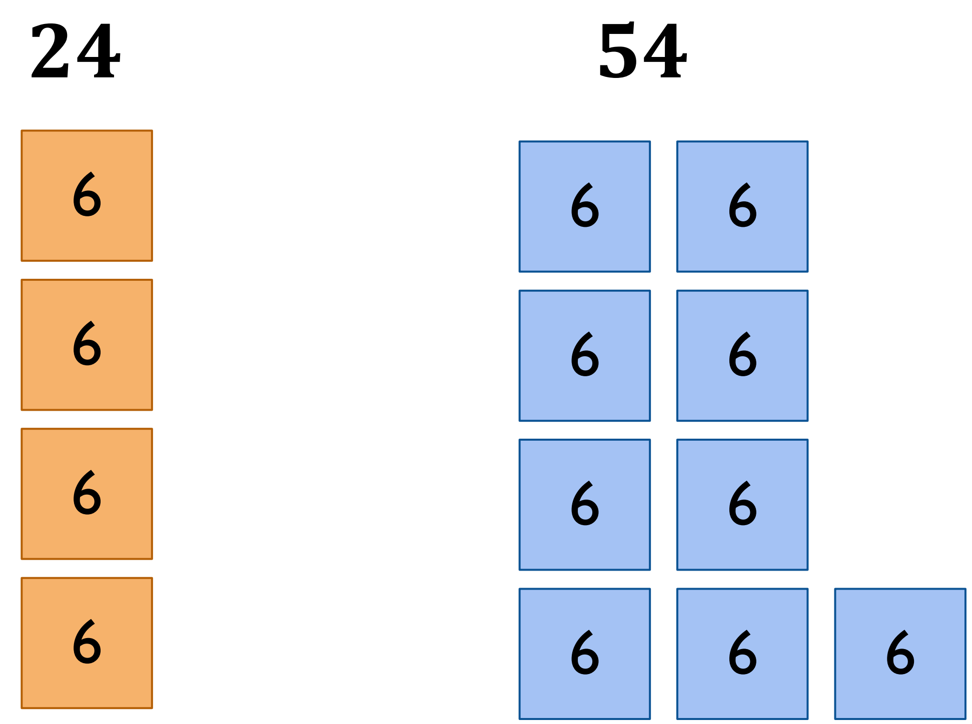
Where am I?

1. Greatest Common Divisor
2. While loops
3. Euclid's Algorithm

Let's *analyse the problem* to derive smarter algorithm

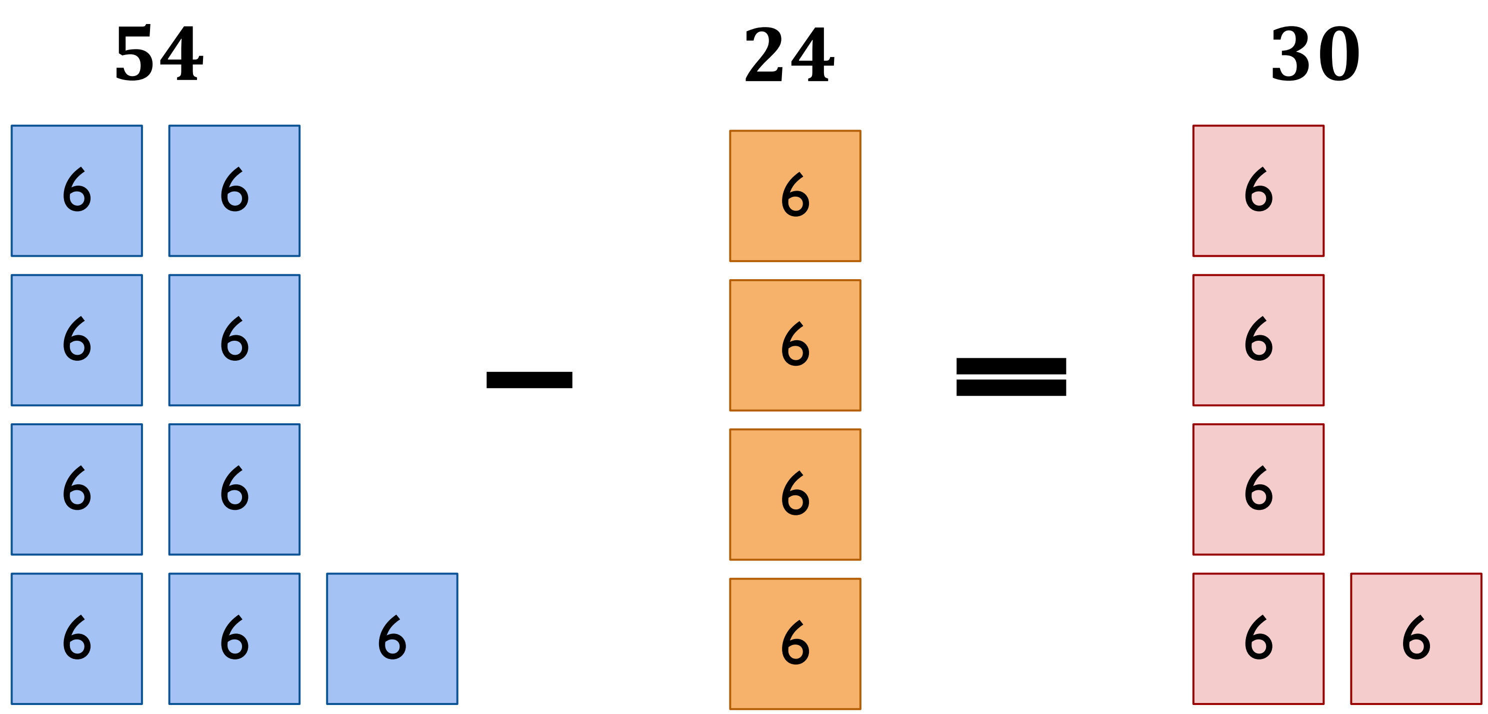
Orange: 24 = 4x6
Blue: 54 = 9x6

Both stacks are made of 6s.



Input can be decreased to *smaller input* with *same output*

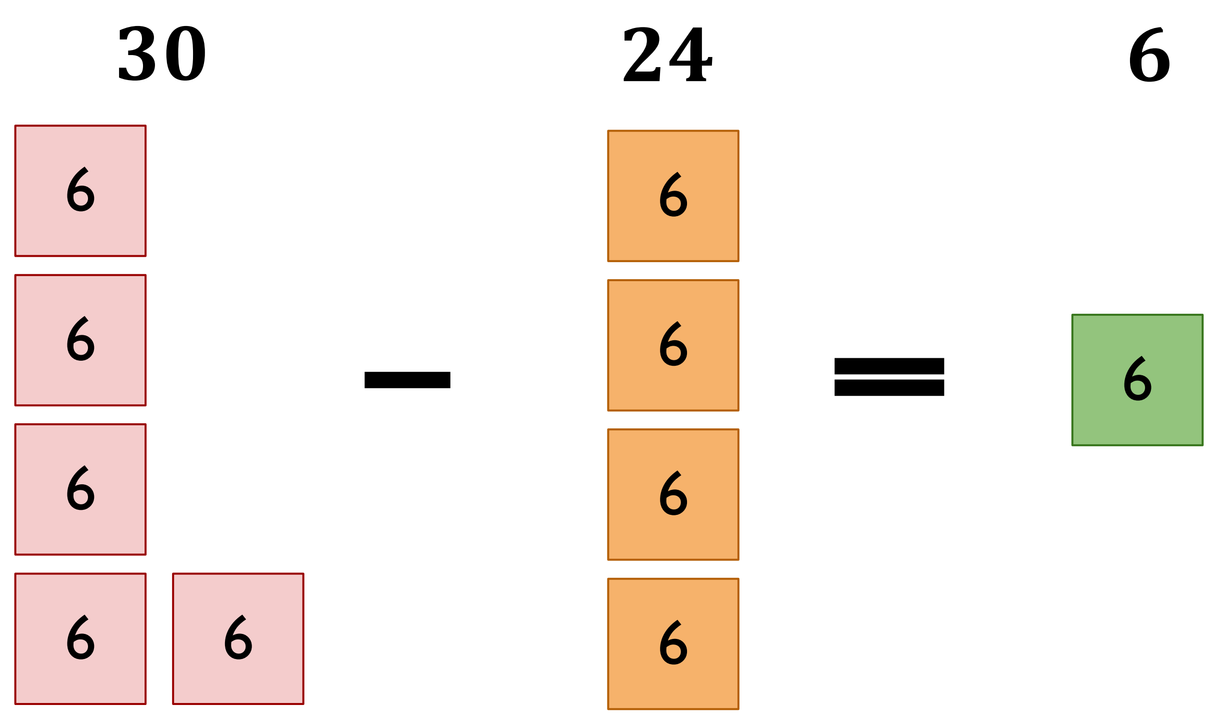
If we *subtract* the smaller stack from the bigger stack, the result will also be made of 6s:



We have (almost) shown that gcd(54, 24) = gcd(24, 30)

This reduction can be applied repeatedly

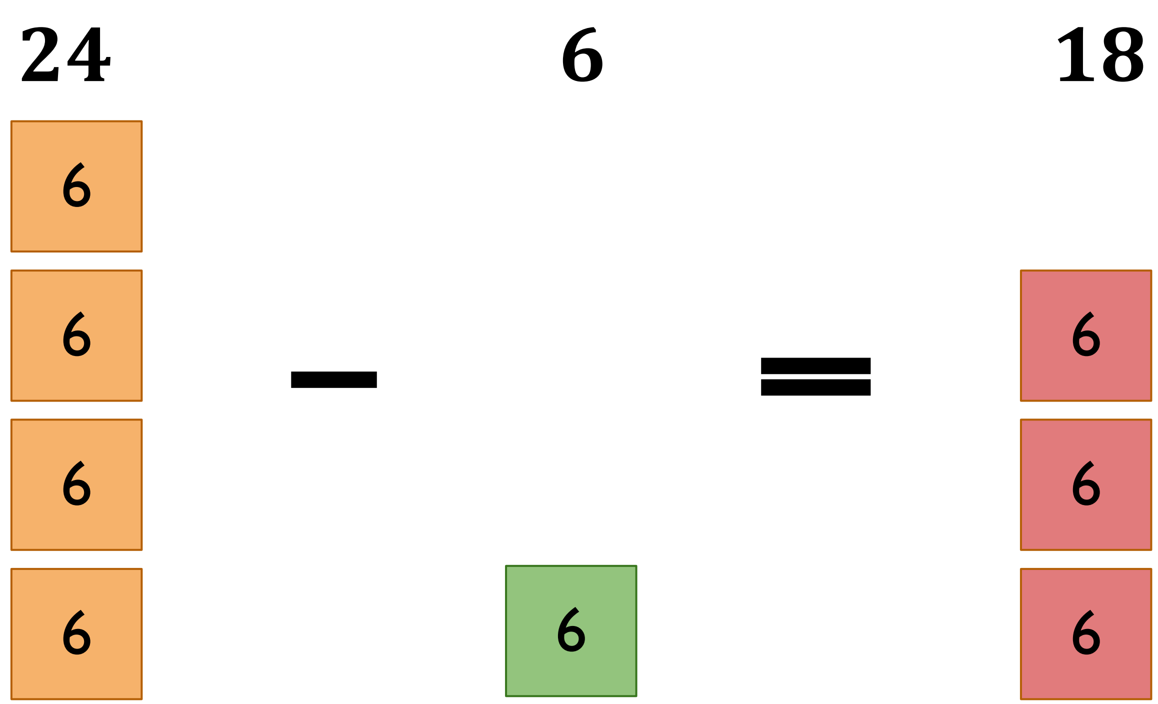
We repeat the process, subtracting the new smallest stack from the previous smallest stack:



We have shown that gcd(54, 24) = gcd(24, 6)

…but we don't have to stop there

We repeat the process, subtracting the new smallest stack from the previous smallest stack:



We have shown that gcd(54, 24) = gcd(24, 6)

Eventually we end up at a simple case

The of a non-zero *m* number and zero is simply *m*

6 **0** **6**

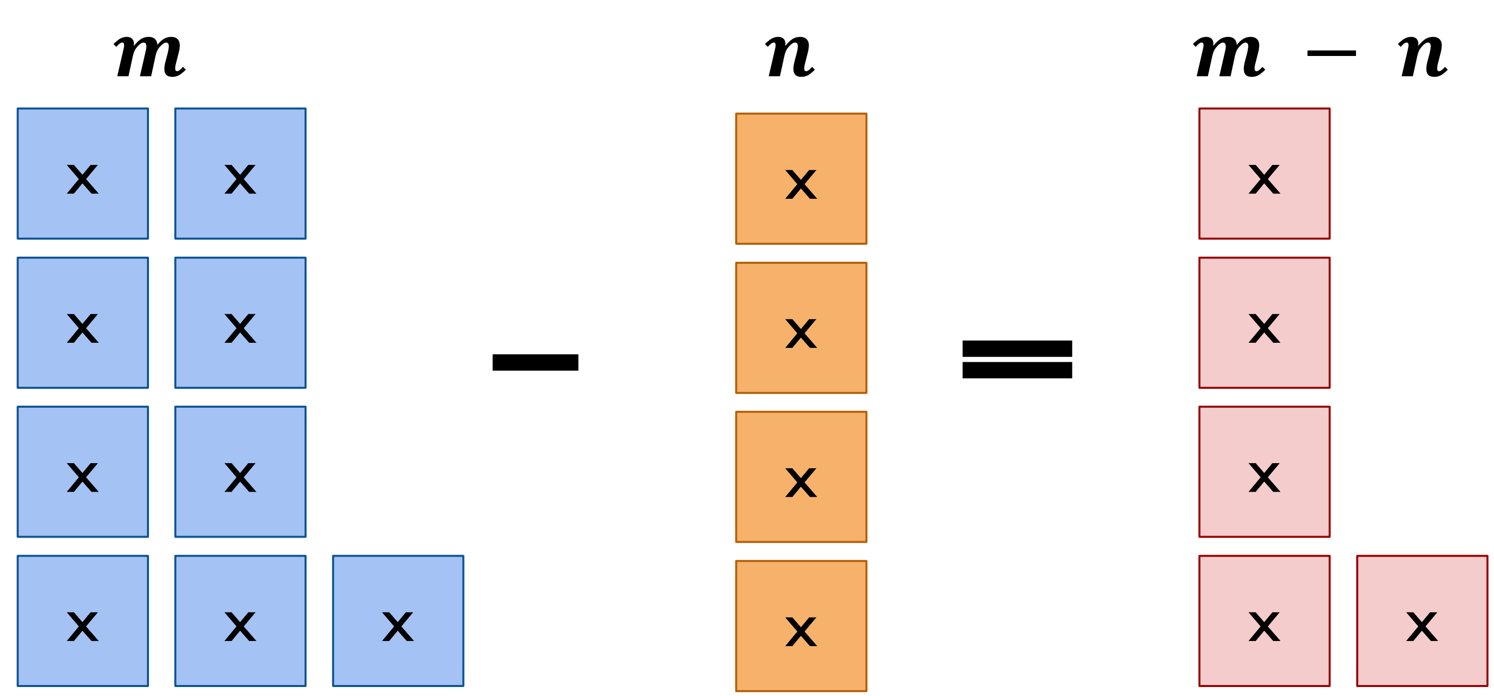
$\boxed{6} - \quad = \boxed{6}$

We have shown that gcd(6, 0) = 6

Pattern also holds for *unknown number* in each box

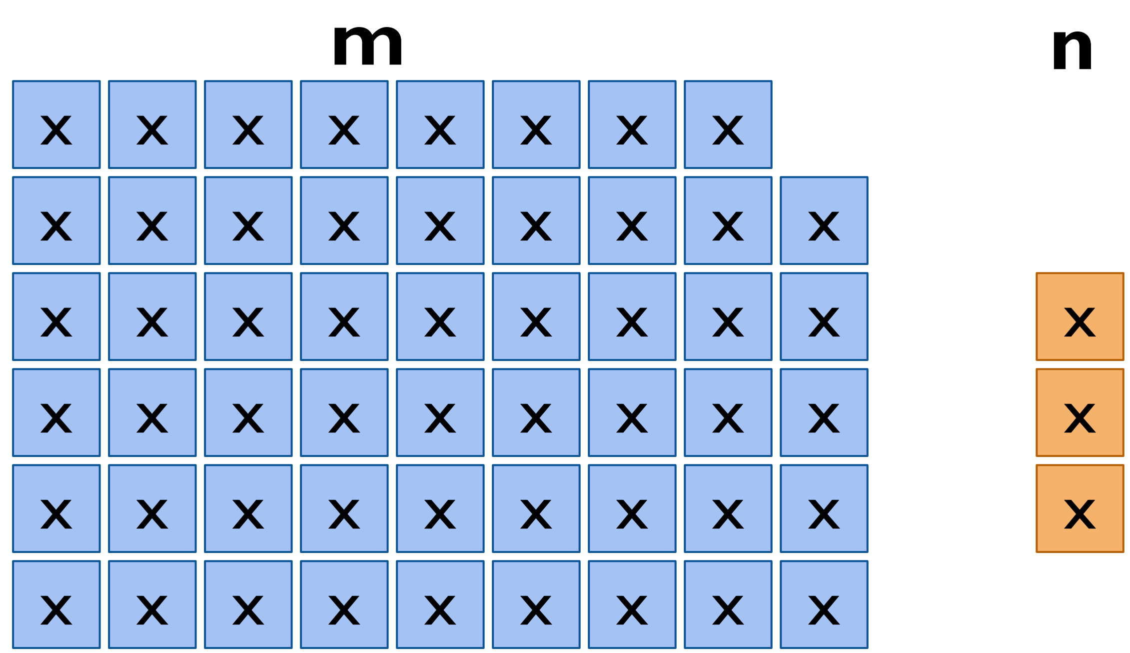
We have shown that: $\gcd(9x, 4x) = \gcd(4x, 9x - 4x)$

We can generalise: $\gcd(m, n) = \gcd(n, m - n)$



We can improve efficiency

What would happen if our m and n started like this:

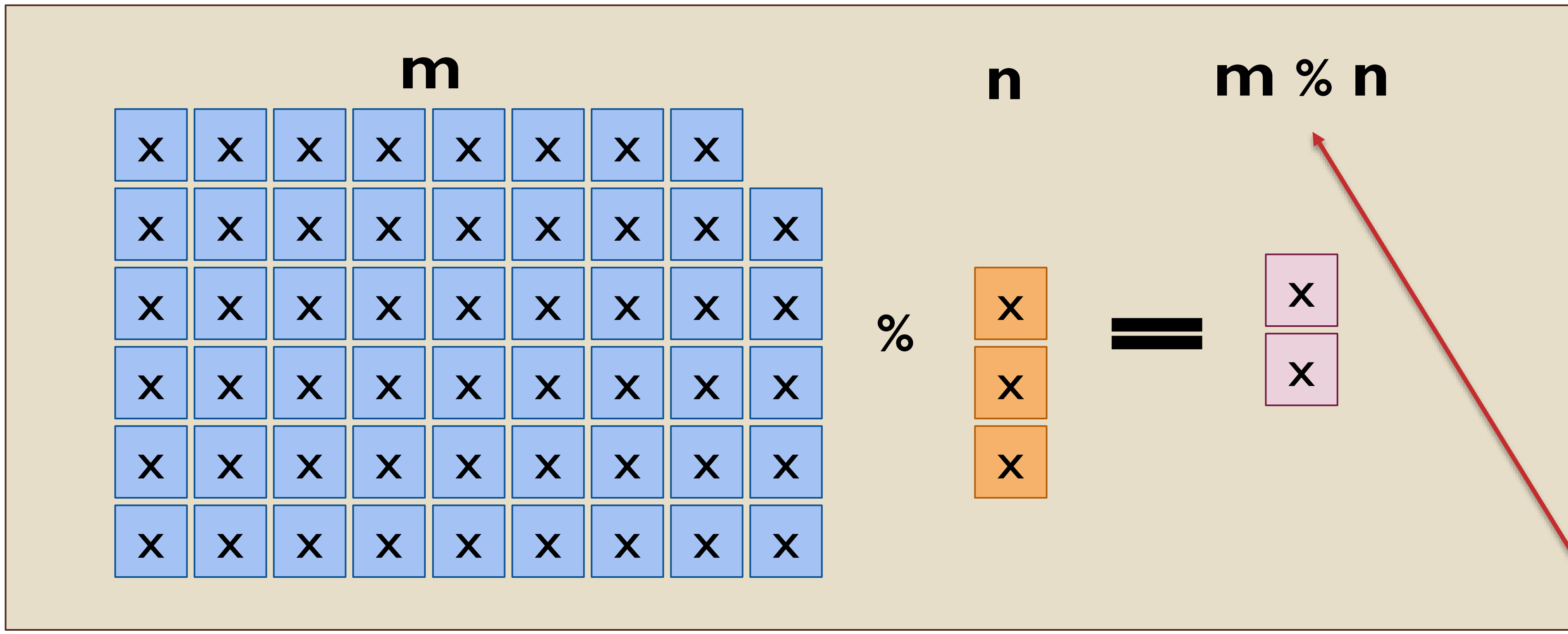


We would have to subtract n 17 times!

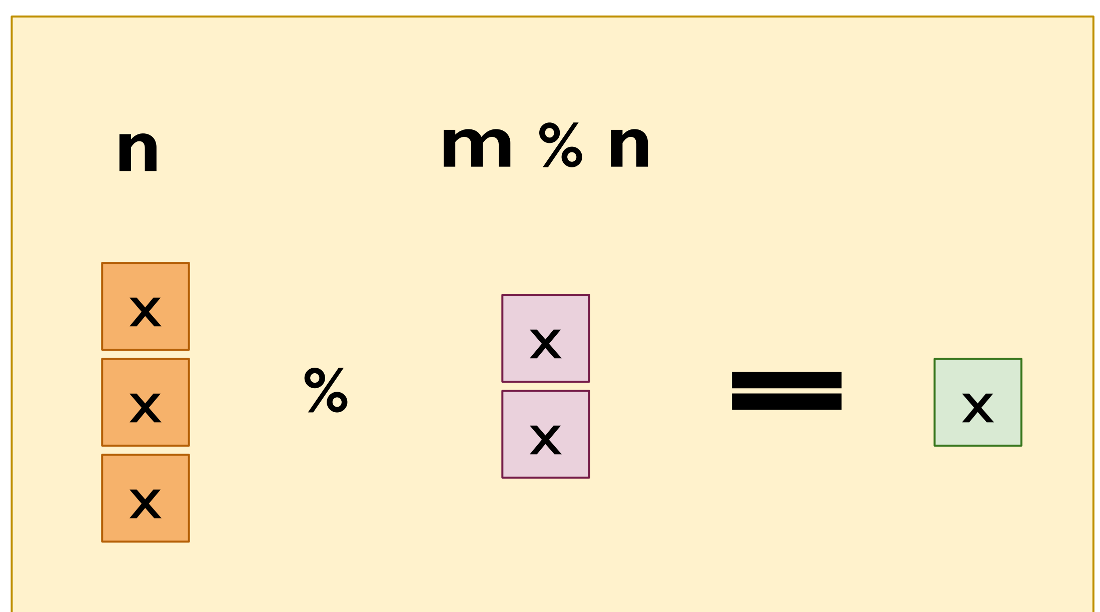
Instead of subtracting, we get the same result if we take the integer remainder of dividing m by n, i.e., $m \ \% \ n$ in Python.

(This operation is also called modulo).

Final problem reduction: gcd(m, n) = gcd(n, m % n)



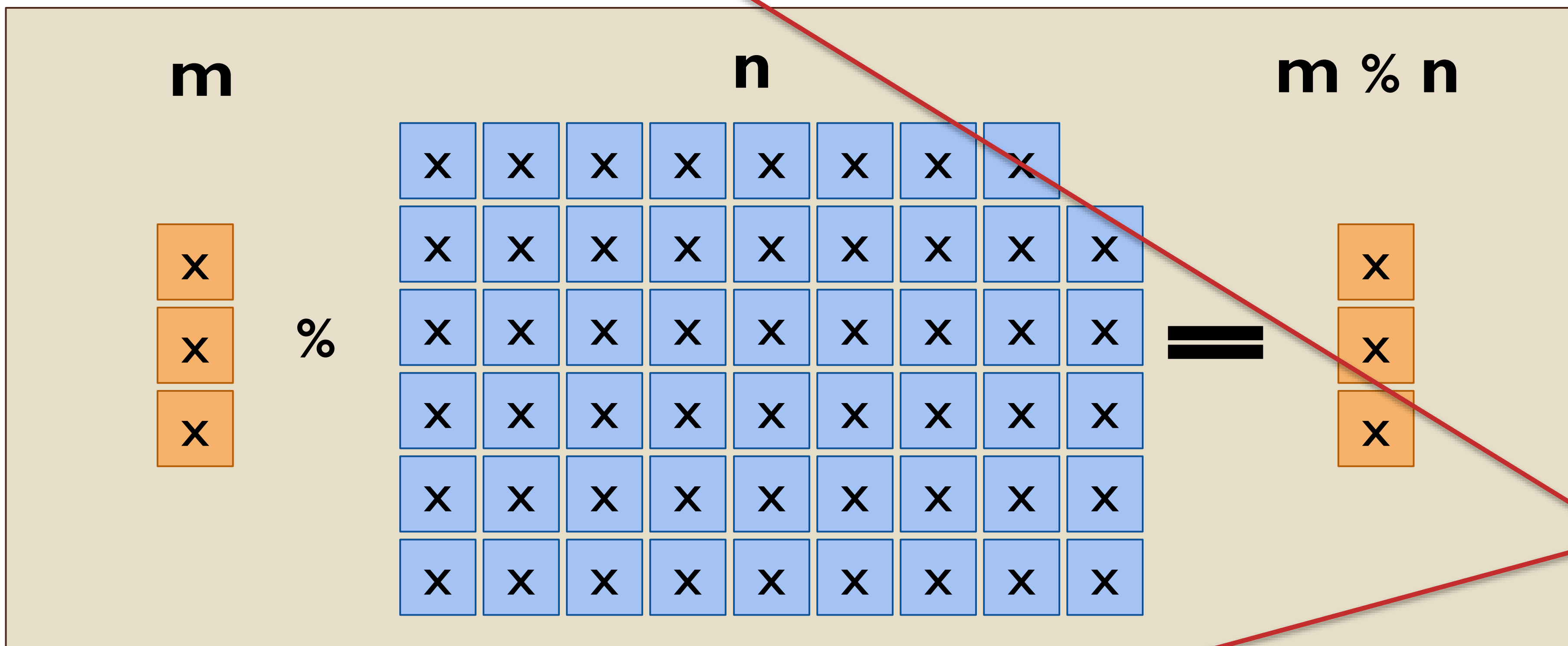
Seems we need
m >=n
for this to work.



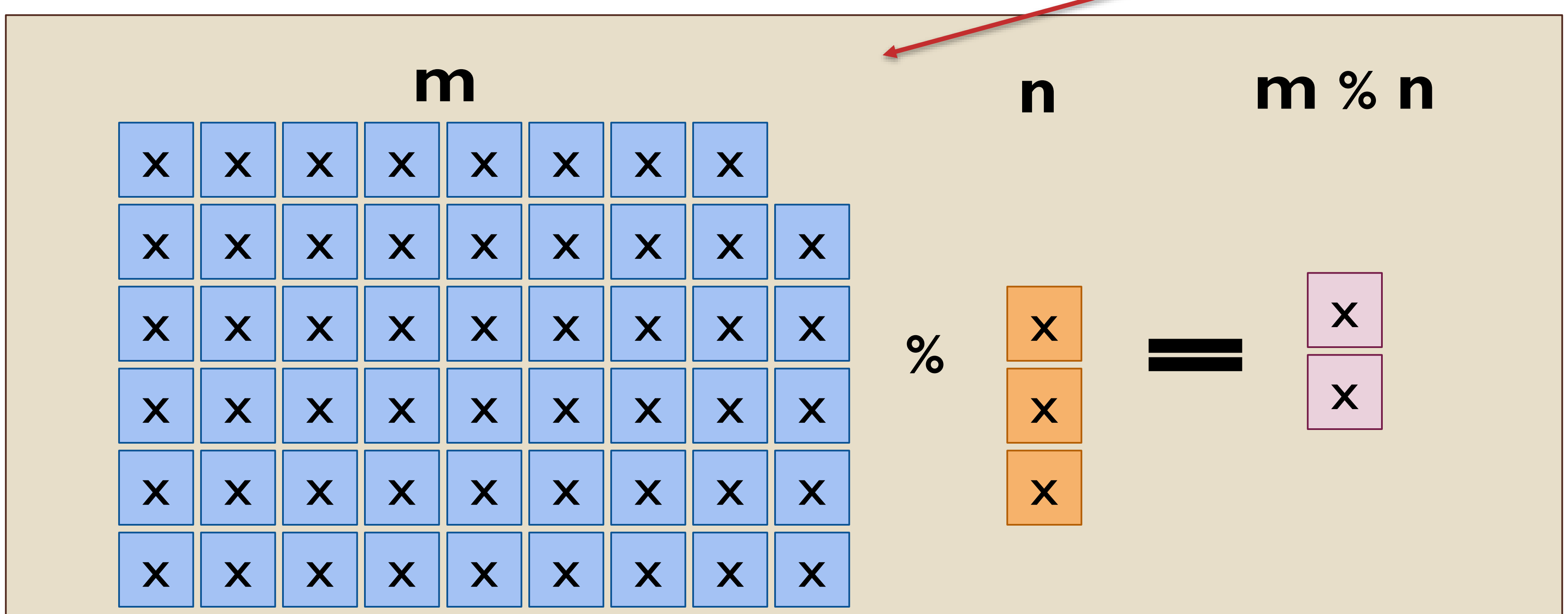
https://flux.qa

Clayton: AXXULH
Malaysia: LWERDE

Final problem reduction: gcd(m, n) = gcd(n, m % n)

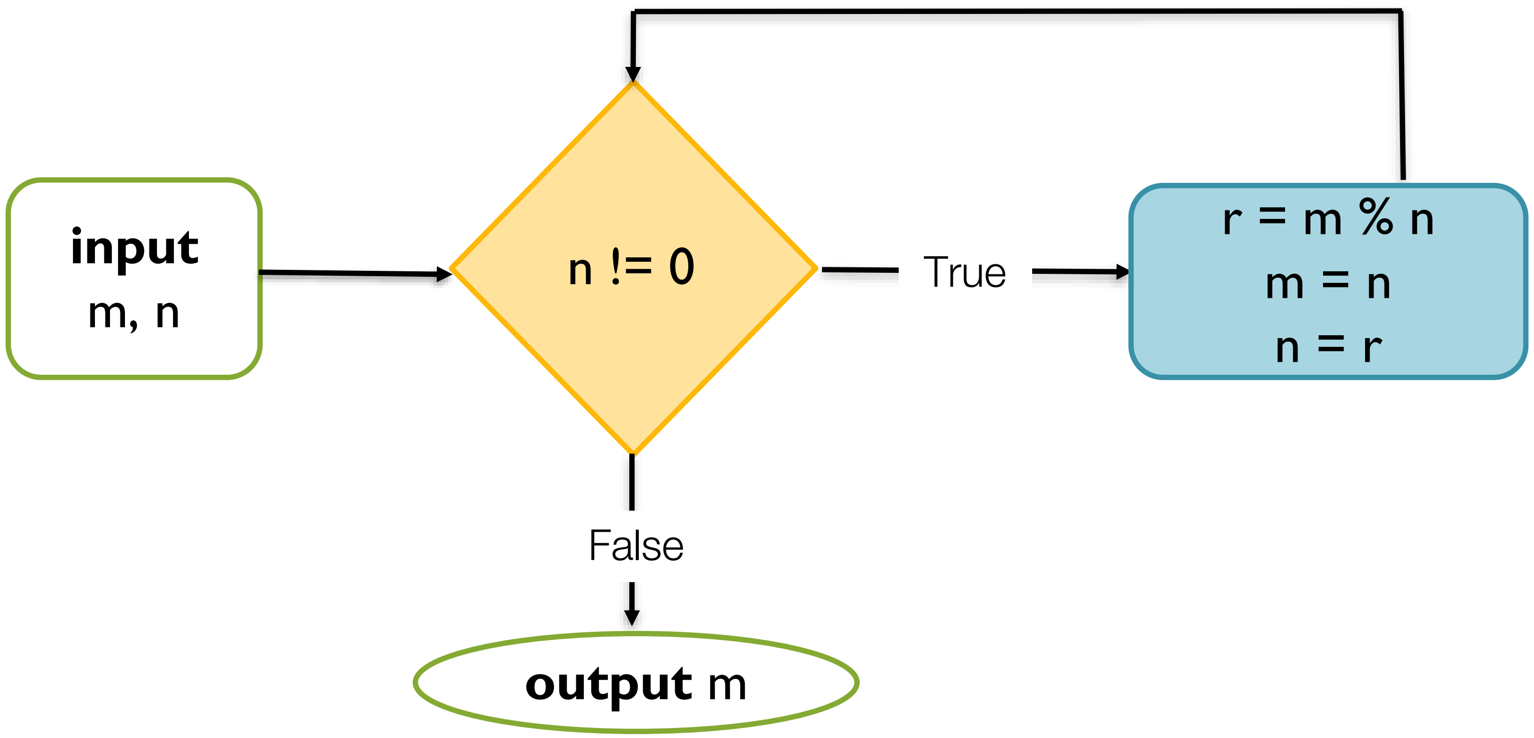


But if m <=n reduction simply flips arguments



Thus gcd(m, n) = gcd(n, m%n) is generally correct and reduces problem size after at most two applications

Euclid's Algorithm



Eukleides of Alexandria
3xx BC – 2xx BC

```
def gcd(m, n):
    """
    Input : integers m and n such that not n==m==0
    Output: the greatest common divisor of m and n
    """
    while n != 0:
        r = m % n
        m = n
        n = r
    return m
```

Recommended reading

"*Introduction to Computing using Python: An Application Development Focus*", by L. Perkovic

- §2.3
- §5.3

FIT1045/53 Workbook

- Chapter 2, §2.2.1
- Chapter 3, §§3.1-3.3

Check point for this week

- By the end of this week you should be able to do the following:
- Implement Python programs to:
  - Calculate the average of a list
  - Find a given item in a list
  - Compute specific sums and products

https://flux.qa

Clayton: AXXULH
Malaysia: LWERDE

Coming Up

- More loops and sequence types
- Tables and matrices