KOÇ
ÜNİVERSİTESİ

05 – Repetition Structures

COMP125 Programming with Python

Recording Disclaimer

The synchronous sessions are recorded (audiovisual recordings). The students are not required to keep their cameras on during class.

The audiovisual recordings, presentations, readings and any other works offered as the course materials aim to support remote and online learning. They are only for the personal use of the students. Further use of course materials other than the personal and educational purposes as defined in this disclaimer, such as making copies, reproductions, replications, submission and sharing on different platforms including the digital ones or commercial usages are strictly prohibited and illegal.

The persons violating the above-mentioned prohibitions can be subject to the administrative, civil, and criminal sanctions under the Law on Higher Education Nr. 2547, the By-Law on Disciplinary Matters of Higher Education Students, the Law on Intellectual Property Nr. 5846, the Criminal Law Nr. 5237, the Law on Obligations Nr. 6098, and any other relevant legislation.

The academic expressions, views, and discussions in the course materials including the audio-visual recordings fall within the scope of the freedom of science and art.

Control Structures *(revisited)*

- There are three types of control structures
  - Sequence statements, which are executed sequentially
  - Conditional (decision) statements: **`if, if-else, if-elif-else`**
  - Repetition statements: **`for, while`**
- These statements are combined by either *sequencing* or *nesting*

Repetition Statements (Loops)

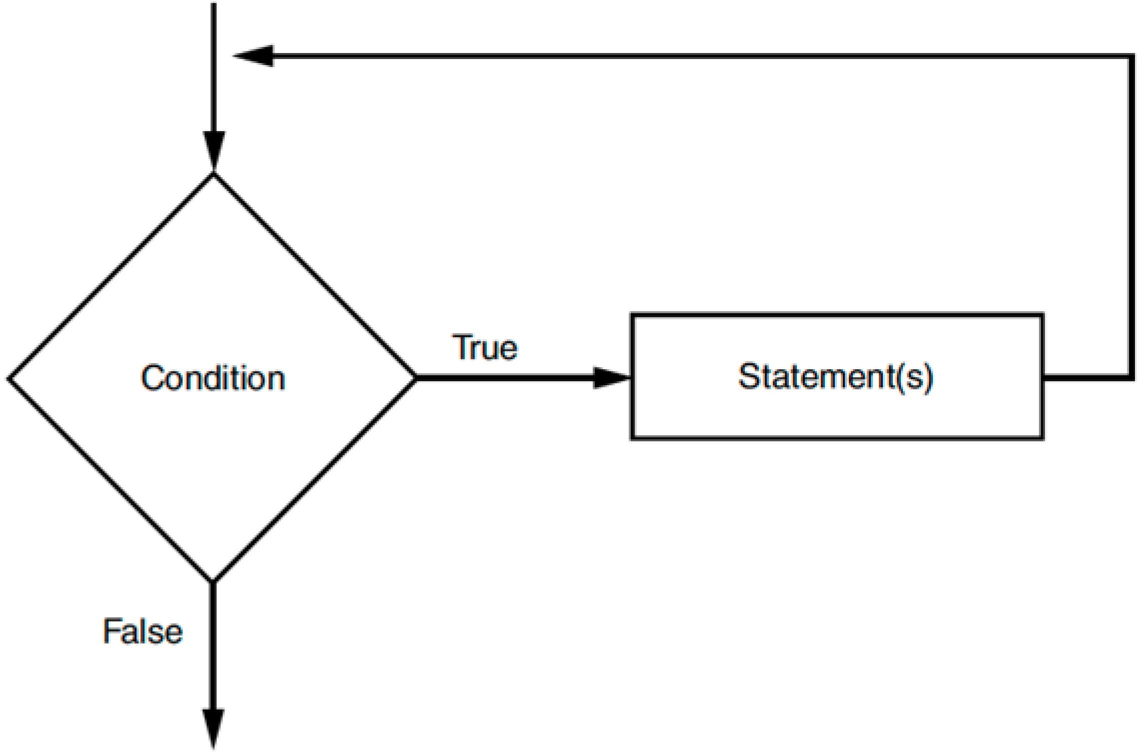
1. Condition-controlled loops
   - Use *True / False* condition to control the number of repetitions

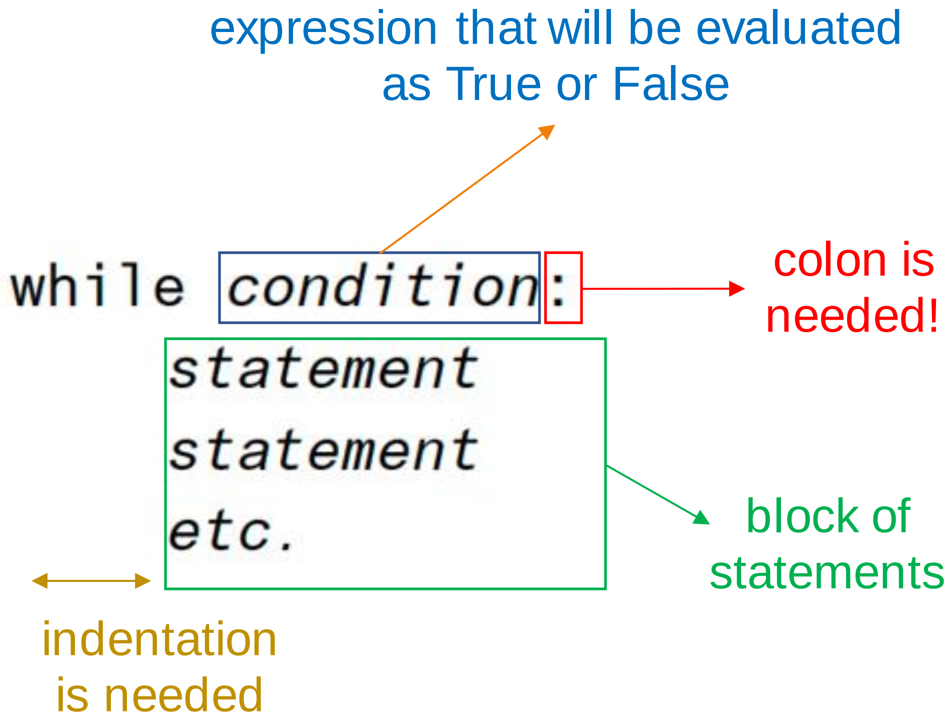
2. Count-controlled loops
   - Repeat a specific number of times

Condition-Controlled Loops

The `while` Statement

- Repeat while a condition remains True
- The **`while`** Statement





expression that will be evaluated as True or False

```
while condition:
    statement
    statement
    etc.
```

colon is needed!

block of statements

indentation is needed

Example 1

- Suppose you are given a total budget to cover your expenses at a bookstore.

  Write a program that first takes your total budget and then your book orders one by one, in the form of the book title and its price. You may assume that the budget and prices are entered as positive numbers.

  For each order, the program checks the remaining budget and allows placing this order only if there is sufficient budget left. The program stops when the first invalid order is placed.

Pseudocode (first version)

```
Take the total budget

Take the first book order (title1, price1)
If price1 <= total budget, order the book, update the budget
Else stop

Take the second book order (title2, price2)
If price2 <= remaining budget, order the book, update the budget
Else stop

Take the third book order (title3, price3)
If price3 <= remaining budget, order the book, update the budget
Else stop

…
```

These statements are repeated as long as there is sufficient budget left

Example 1

Pseudocode (first version)

```
Take the total budget

Take the first book order (title1, price1)
If price1 <= total budget, order the book, update the budget
Else stop
```

These statements are repeated as long as there is sufficient budget left

```
Take the second book order (title2, price2)
If price2 <= remaining budget, order the book, update the budget
Else stop

Take the third book order (title3, price3)
If price3 <= remaining budget, order the book, update the budget
Else stop

…
```

Pseudocode (second version)

```
Step 1.  Take the total budget

Step 2.  Take a book order (title, price)
Step 3.  Check price <= total budget, if no, go to Step 7
Step 4.  Order the book
Step 5.  Total budget = total budget - price
Step 6.  Go to Step 2

Step 7.  STOP
```

Example 1

Pseudocode (second version)

```
Step 1.  Take the total budget

Step 2.  Take a book order (title, price)
Step 3.  Check price <= total budget, if no, go to Step 7
Step 4.  Order the book
Step 5.  Total budget = total budget - price
Step 6.  Go to Step 2

Step 7.  STOP
```

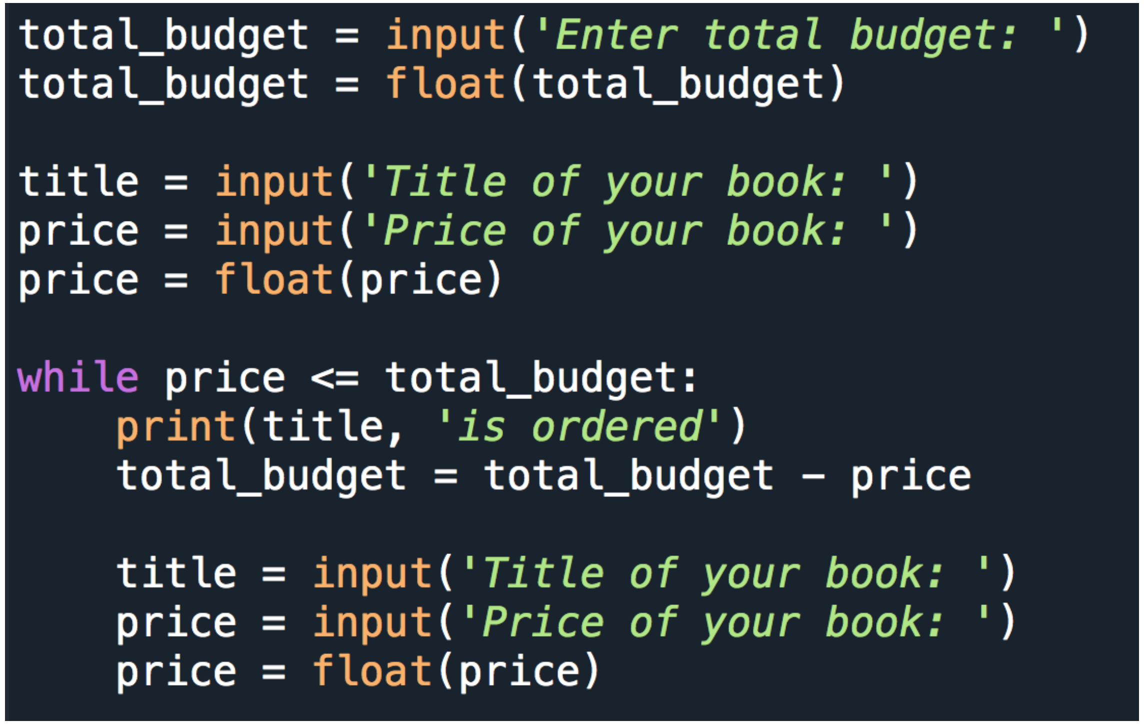
Pseudocode (another version)

```
Take the total budget

Take a book order (title, price)
While price <= total budget
    Order the book
    Total budget = total budget - price
    Take a book order (title, price)
```

Example 1

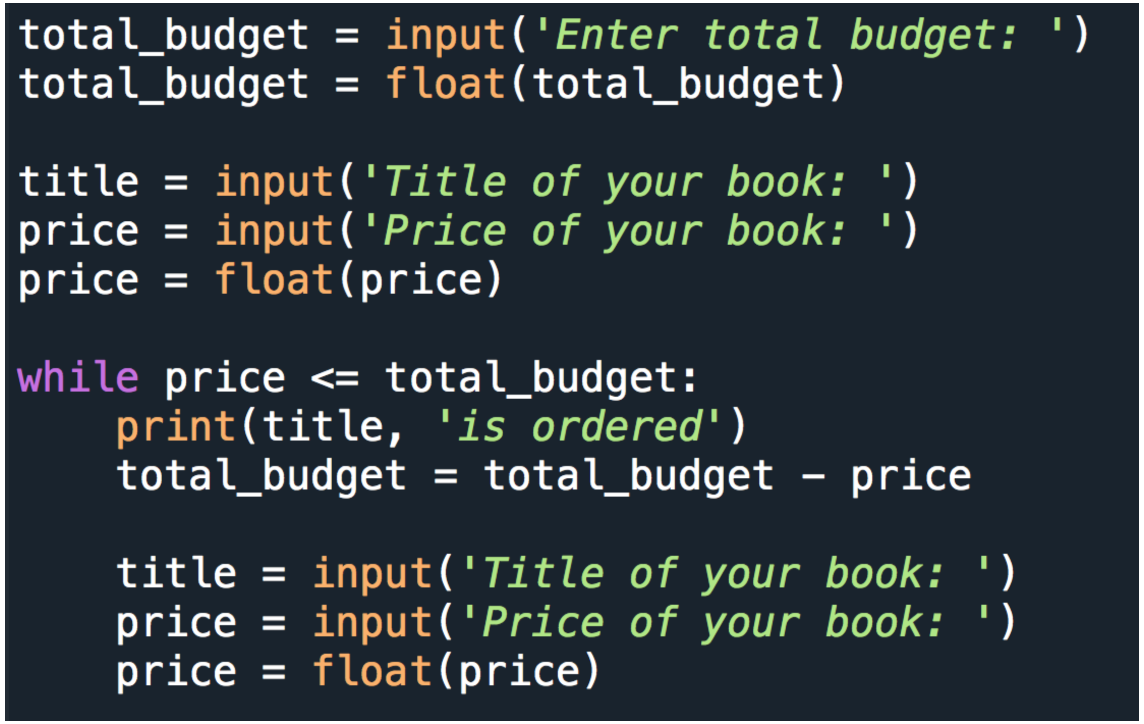
Pseudocode (another version)

```
Take the total budget

Take a book order (title, price)
While price <= total budget
        Order the book
        Total budget = total budget - price
        Take a book order (title, price)
```



```python
total_budget = input('Enter total budget: ')
total_budget = float(total_budget)

title = input('Title of your book: ')
price = input('Price of your book: ')
price = float(price)

while price <= total_budget:
    print(title, 'is ordered')
    total_budget = total_budget - price

    title = input('Title of your book: ')
    price = input('Price of your book: ')
    price = float(price)
```

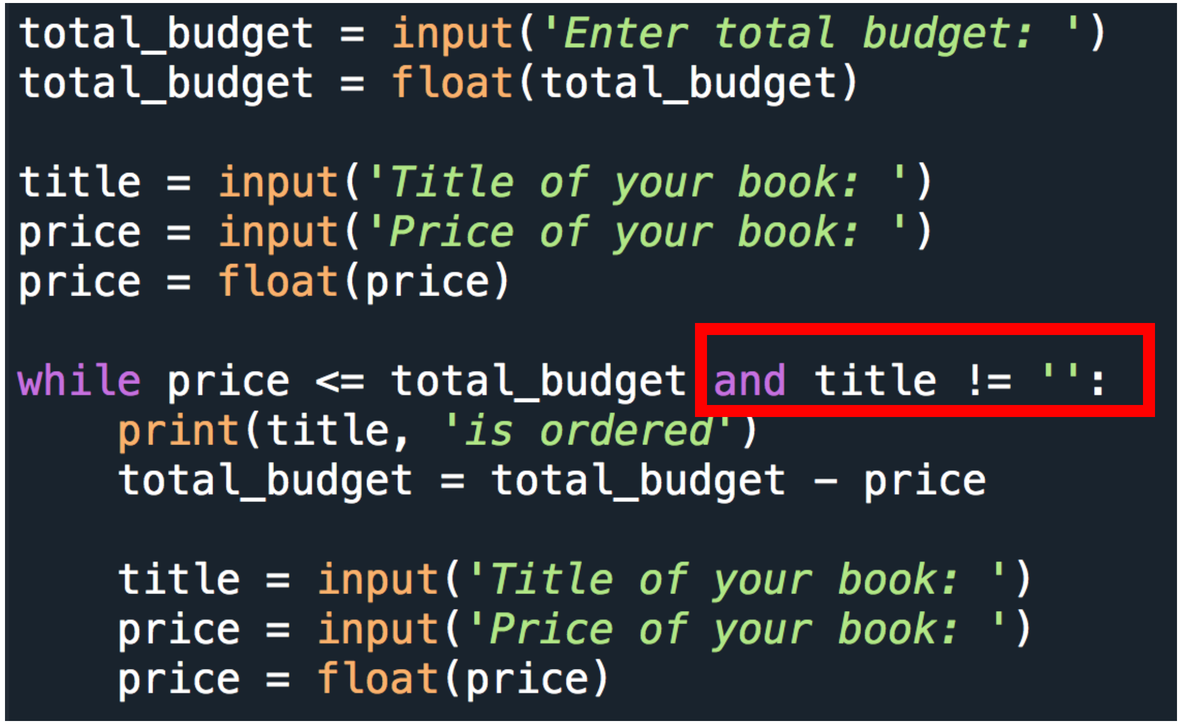
Example 1



```
total_budget = input('Enter total budget: ')
total_budget = float(total_budget)

title = input('Title of your book: ')
price = input('Price of your book: ')
price = float(price)

while price <= total_budget:
    print(title, 'is ordered')
    total_budget = total_budget - price

    title = input('Title of your book: ')
    price = input('Price of your book: ')
    price = float(price)
```

1. Extend the program such that it also stops taking an order when an empty string is entered as the book title

Example 1

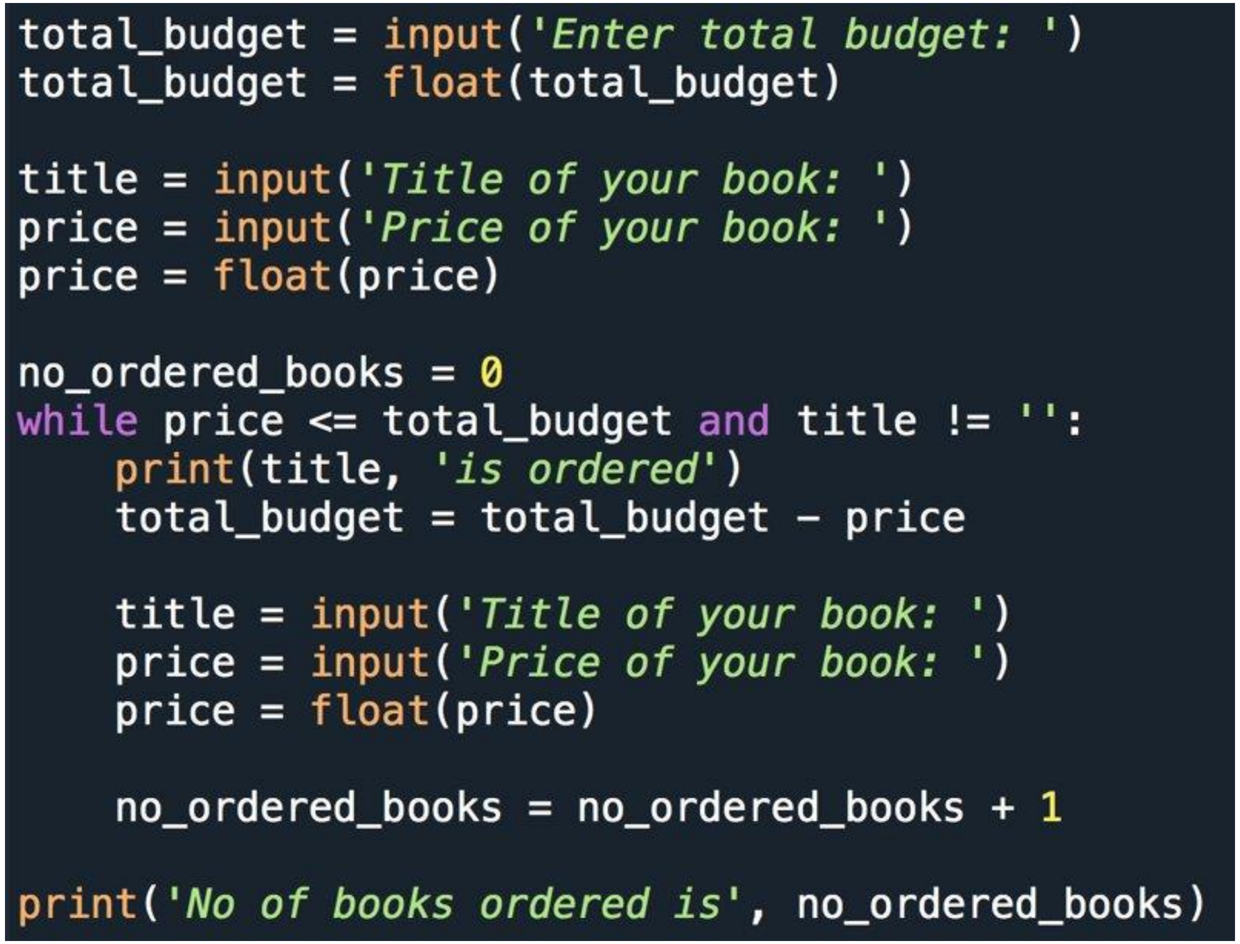


```python
total_budget = input('Enter total budget: ')
total_budget = float(total_budget)

title = input('Title of your book: ')
price = input('Price of your book: ')
price = float(price)

while price <= total_budget and title != '':
    print(title, 'is ordered')
    total_budget = total_budget - price

    title = input('Title of your book: ')
    price = input('Price of your book: ')
    price = float(price)
```

1. Extend the program such that it also stops taking an order when an empty string is entered as the book title

2. Extend the program such that it also counts the number of the ordered books and displays it

Example 1



```python
total_budget = input('Enter total budget: ')
total_budget = float(total_budget)

title = input('Title of your book: ')
price = input('Price of your book: ')
price = float(price)

no_ordered_books = 0
while price <= total_budget and title != '':
    print(title, 'is ordered')
    total_budget = total_budget - price

    title = input('Title of your book: ')
    price = input('Price of your book: ')
    price = float(price)

    no_ordered_books = no_ordered_books + 1

print('No of books ordered is', no_ordered_books)
```

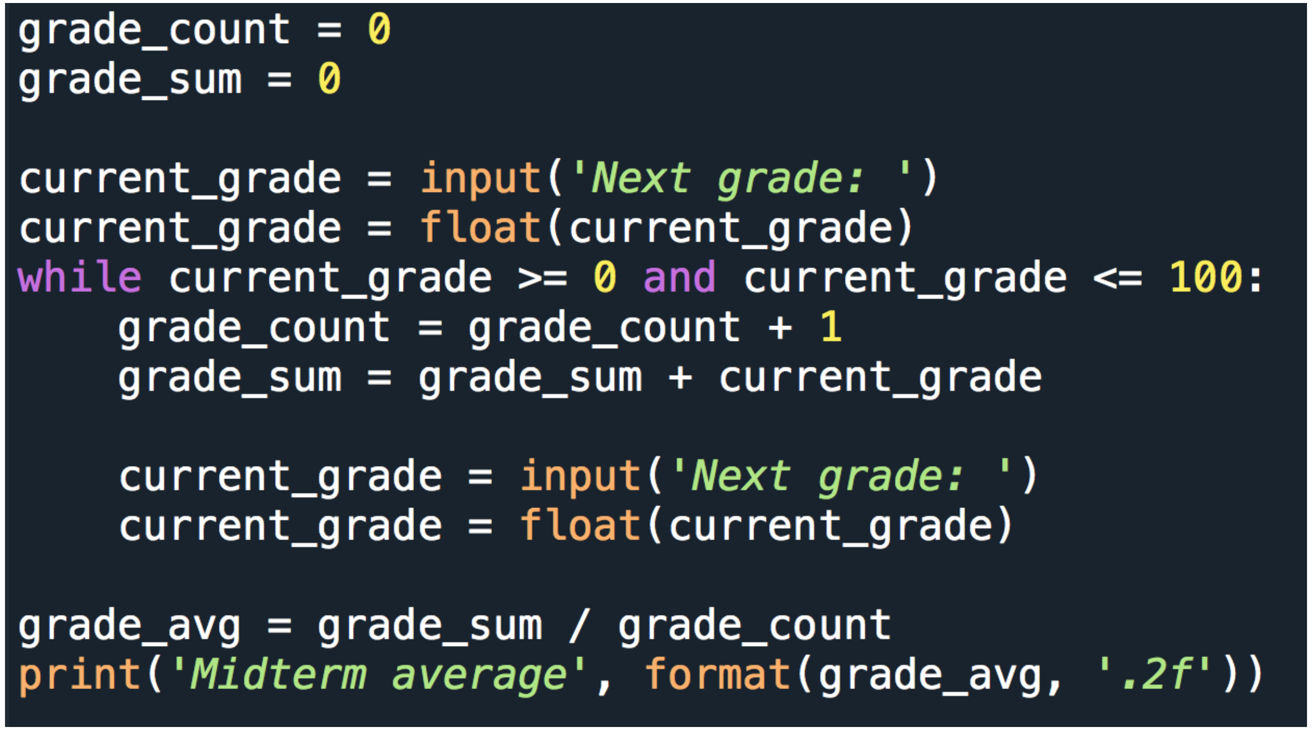
1. Extend the program such that it also stops taking an order when an empty string is entered as the book title

2. Extend the program such that it also counts the number of the ordered books and displays it

Example 2

○ Write a program that takes the grades of students one by one, calculates the average midterm grade, and displays the average on the screen.

A valid grade is in between 0 and 100 and the program continues taking the grades until it receives an invalid grade.



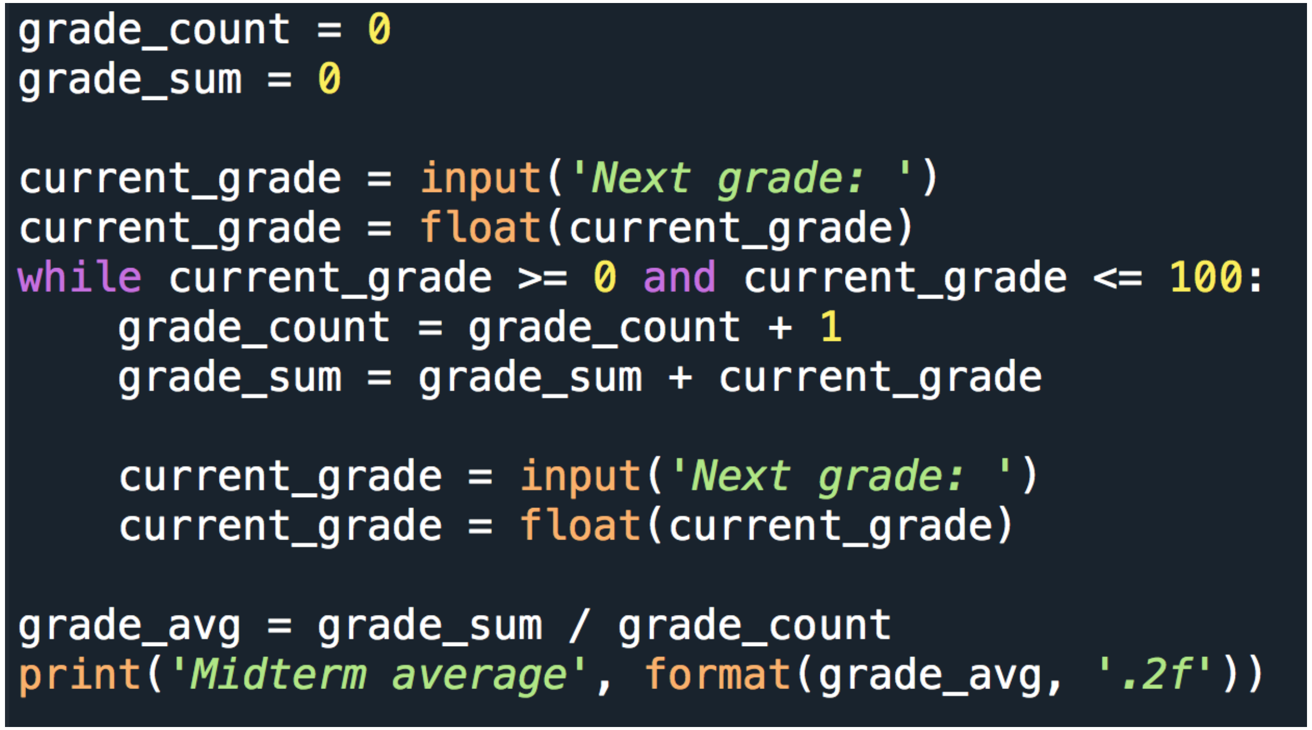
```
grade_count = 0
grade_sum = 0

current_grade = input('Next grade: ')
current_grade = float(current_grade)
while current_grade >= 0 and current_grade <= 100:
    grade_count = grade_count + 1
    grade_sum = grade_sum + current_grade

    current_grade = input('Next grade: ')
    current_grade = float(current_grade)

grade_avg = grade_sum / grade_count
print('Midterm average', format(grade_avg, '.2f'))
```

What is wrong with this code?

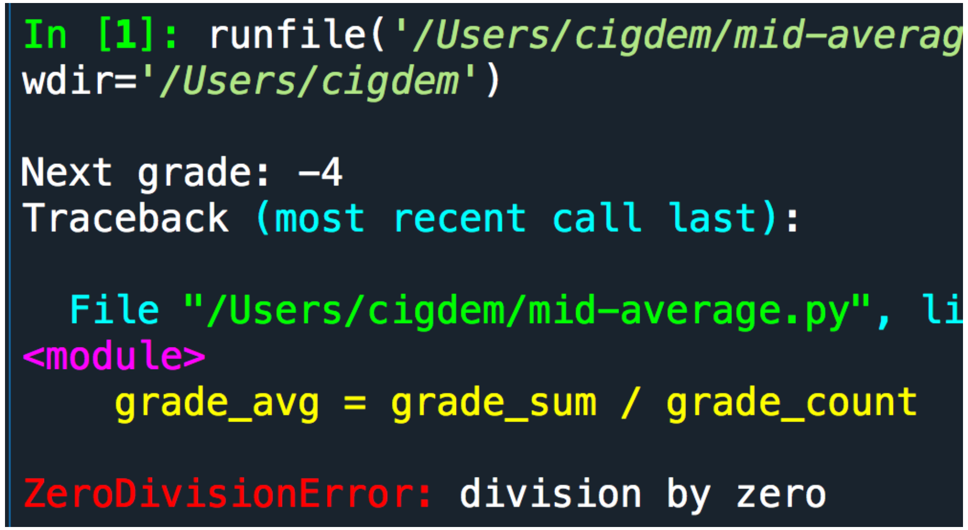
Example 2

○ Write a program that takes the grades of students one by one, calculates the average midterm grade, and displays the average on the screen.

A valid grade is in between 0 and 100 and the program continues taking the grades until it receives an invalid grade.



```python
grade_count = 0
grade_sum = 0

current_grade = input('Next grade: ')
current_grade = float(current_grade)
while current_grade >= 0 and current_grade <= 100:
    grade_count = grade_count + 1
    grade_sum = grade_sum + current_grade

    current_grade = input('Next grade: ')
    current_grade = float(current_grade)

grade_avg = grade_sum / grade_count
print('Midterm average', format(grade_avg, '.2f'))
```



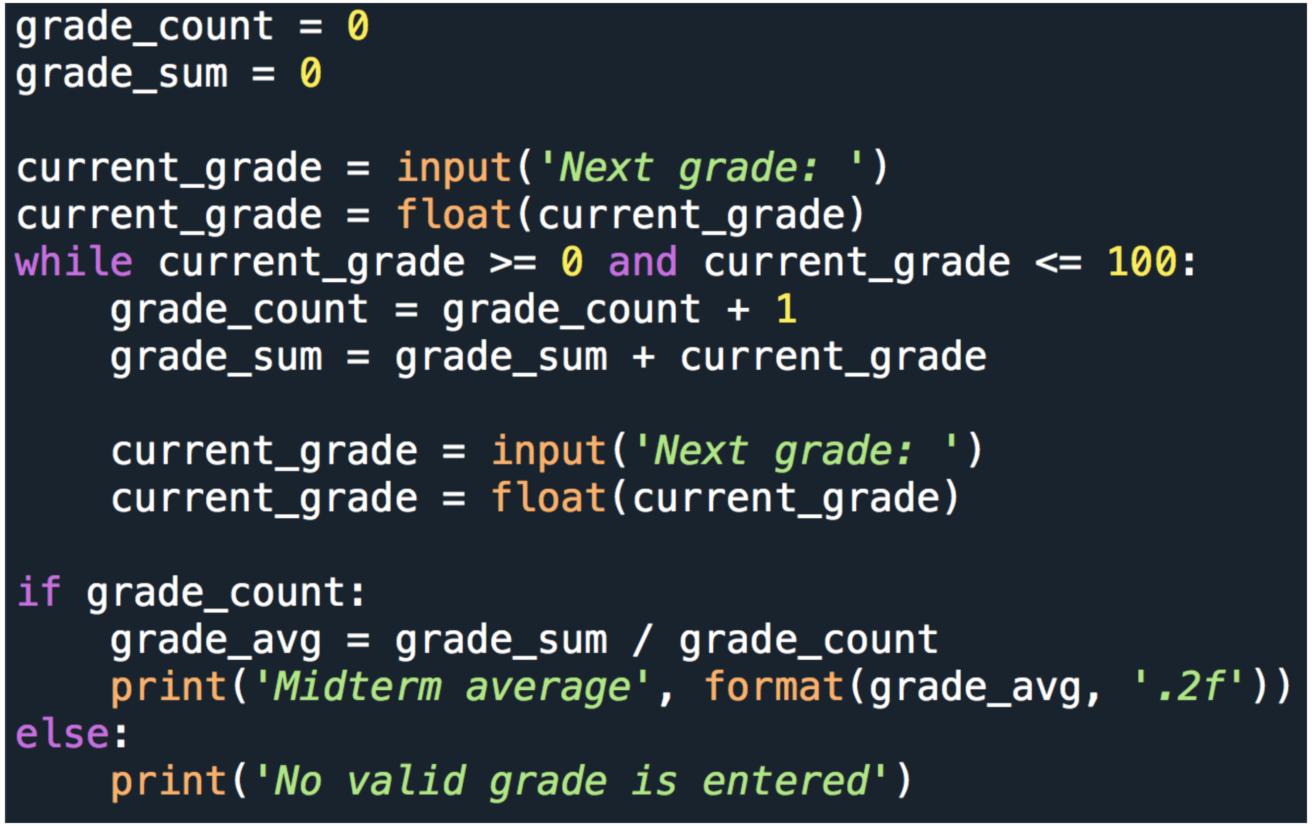
```
In [1]: runfile('/Users/cigdem/mid-averag
wdir='/Users/cigdem')

Next grade: -4
Traceback (most recent call last):

  File "/Users/cigdem/mid-average.py", li
<module>
    grade_avg = grade_sum / grade_count

ZeroDivisionError: division by zero
```

How can you fix it?

Example 2

○ Write a program that takes the grades of students one by one, calculates the average midterm grade, and displays the average on the screen.

A valid grade is in between 0 and 100 and the program continues taking the grades until it receives an invalid grade.

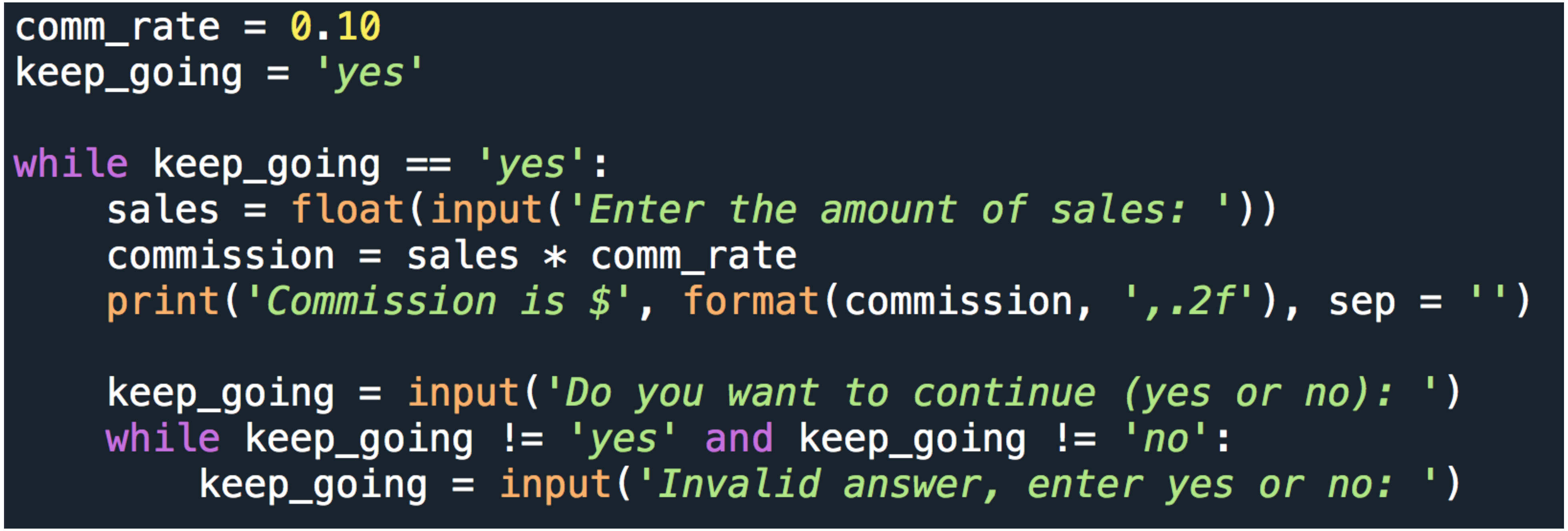


```
grade_count = 0
grade_sum = 0

current_grade = input('Next grade: ')
current_grade = float(current_grade)
while current_grade >= 0 and current_grade <= 100:
    grade_count = grade_count + 1
    grade_sum = grade_sum + current_grade

    current_grade = input('Next grade: ')
    current_grade = float(current_grade)

if grade_count:
    grade_avg = grade_sum / grade_count
    print('Midterm average', format(grade_avg, '.2f'))
else:
    print('No valid grade is entered')
```

Example 3

- Write a program that calculates a 10 percent sales commission for several salespeople. The program calculates the commission for the first salesperson. It then asks the user if s/he wants to perform the same operation for another salesperson. If so, the program repeats the calculation, otherwise it terminates.

  The program should only accept *'yes'*, 'or *'no* as an answer.



```
comm_rate = 0.10
keep_going = 'yes'

while keep_going == 'yes':
    sales = float(input('Enter the amount of sales: '))
    commission = sales * comm_rate
    print('Commission is $', format(commission, ',.2f'), sep = '')

    keep_going = input('Do you want to continue (yes or no): ')
    while keep_going != 'yes' and keep_going != 'no':
        keep_going = input('Invalid answer, enter yes or no: ')
```

Nested Loops

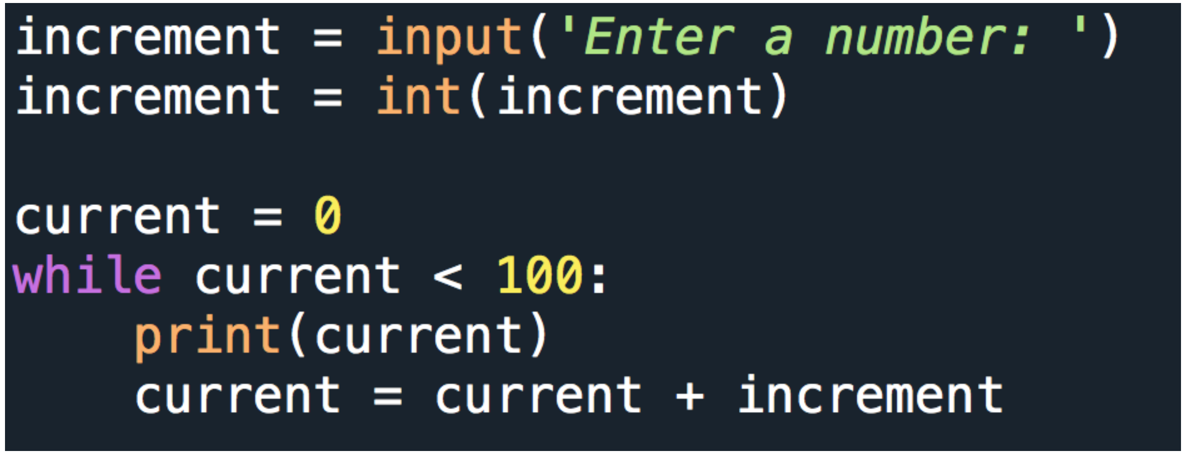
- You can write a loop inside another one
- Indeed, any control structures can be nested

```
while condition1:
    statement
    while condition2:
        statement
        if condition3:
            statement
        elif condition4:
            statement
    if condition5:
        statement
```

```
if condition1:
    statement
    while condition4:
        statement
        …
    statement
elif condition2:
    for i in range(n):
        if condition5:
            statement
        statement
else:
    statement
```

Example 4

- Write a program that takes an integer from the user and displays all numbers from 0 to 100 with the increments of that input



```python
increment = input('Enter a number: ')
increment = int(increment)

current = 0
while current < 100:
    print(current)
    current = current + increment
```

Anything wrong with this code?

Infinite loops are those that never terminate (the condition remains always true)

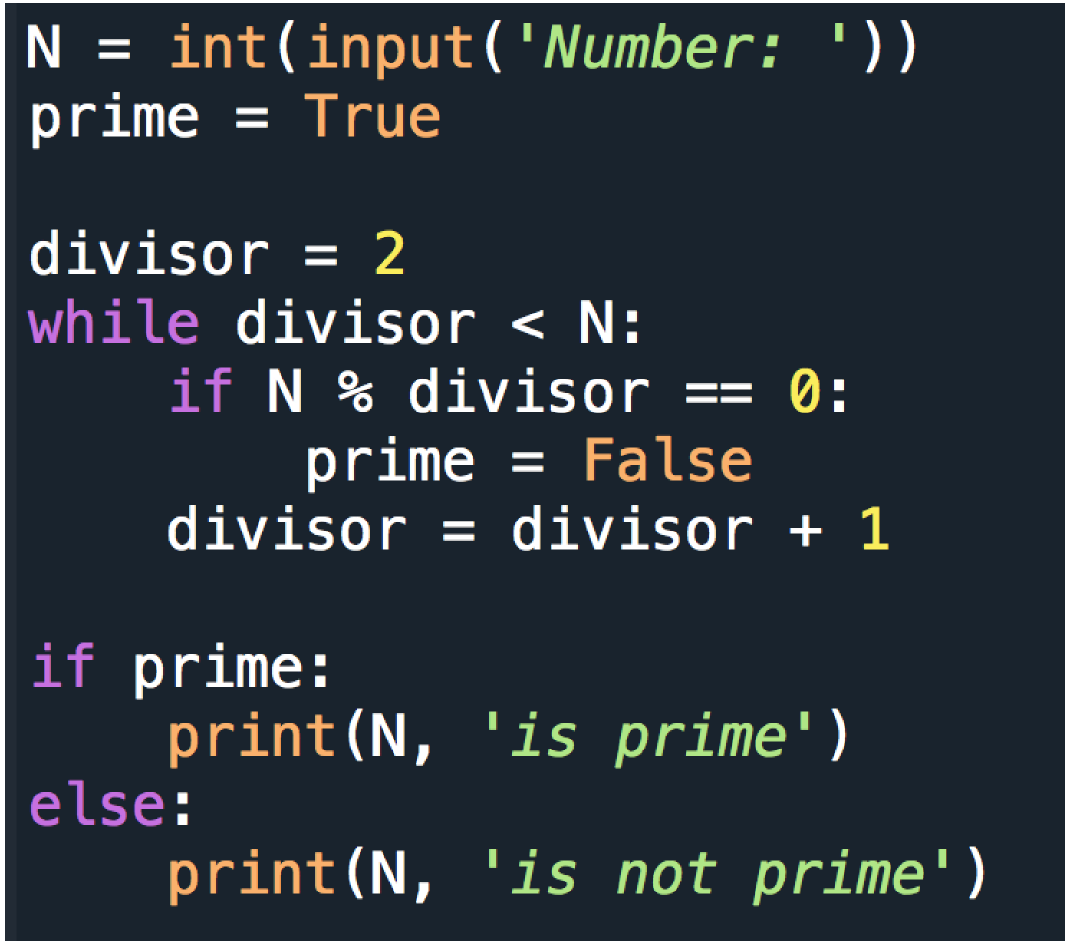
In our case, check how the program works with a non-positive increment (i.e., when the user enters a negative integer or zero for the increment)

This is a logic error

Type Ctrl + C to interrupt any program (also to interrupt the infinite loops)

Example: Prime Number

- Write a program that takes a positive integer from the user and displays whether it is prime or not



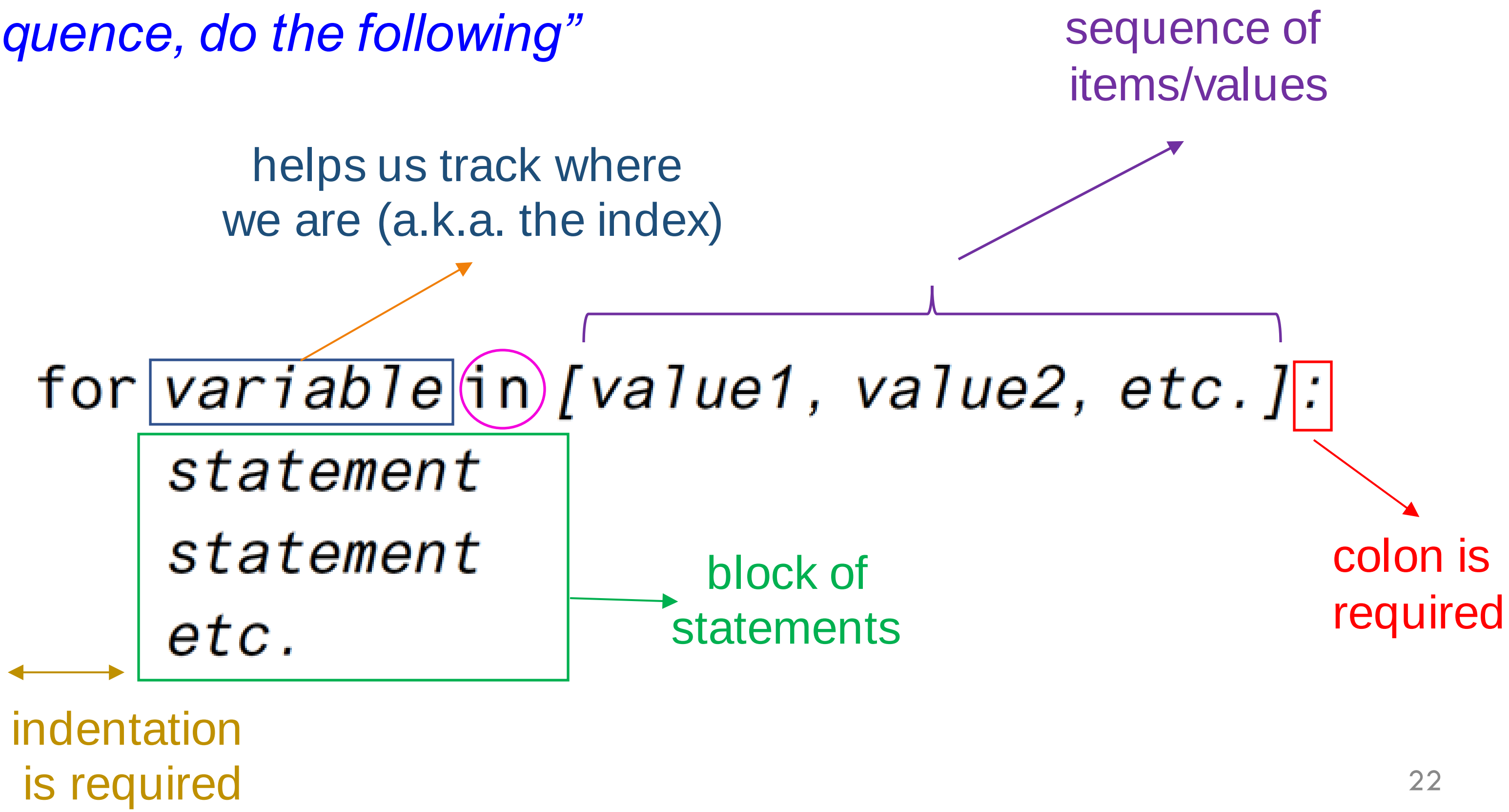
```python
N = int(input('Number: '))
prime = True

divisor = 2
while divisor < N:
    if N % divisor == 0:
        prime = False
    divisor = divisor + 1

if prime:
    print(N, 'is prime')
else:
    print(N, 'is not prime')
```

This is an example of count-controlled loops (in this loop, you consider all possibilities of divisor from 2 to N–1). The `for` statement is commonly preferred to implement the count-controlled loops.

Count-Controlled Loops

The `for` Statement

- Preferred way to code **count-controlled loops**
- In Python, it is actually a **"for each"** loop
  - Iterates once over items in a sequence
  - i.e., *"for each item in the sequence, do the following"*
- General format:



```
for variable in [value1, value2, etc.]:
    statement
    statement
    etc.
```

sequence of items/values

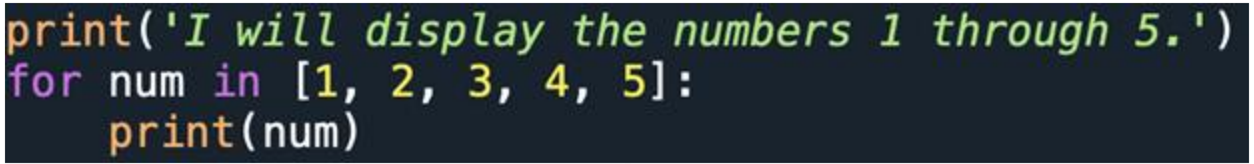
helps us track where we are (a.k.a. the index)

block of statements

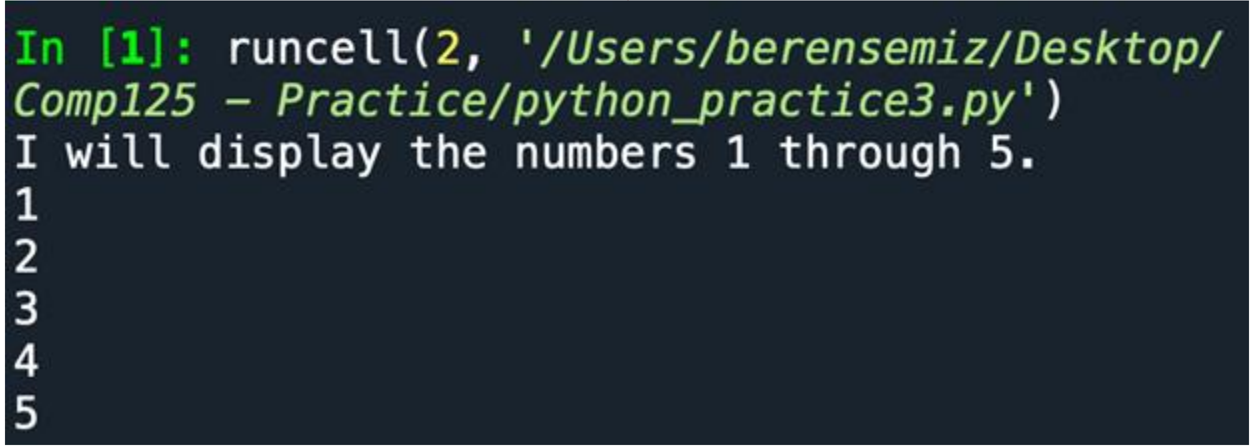
colon is required

indentation is required

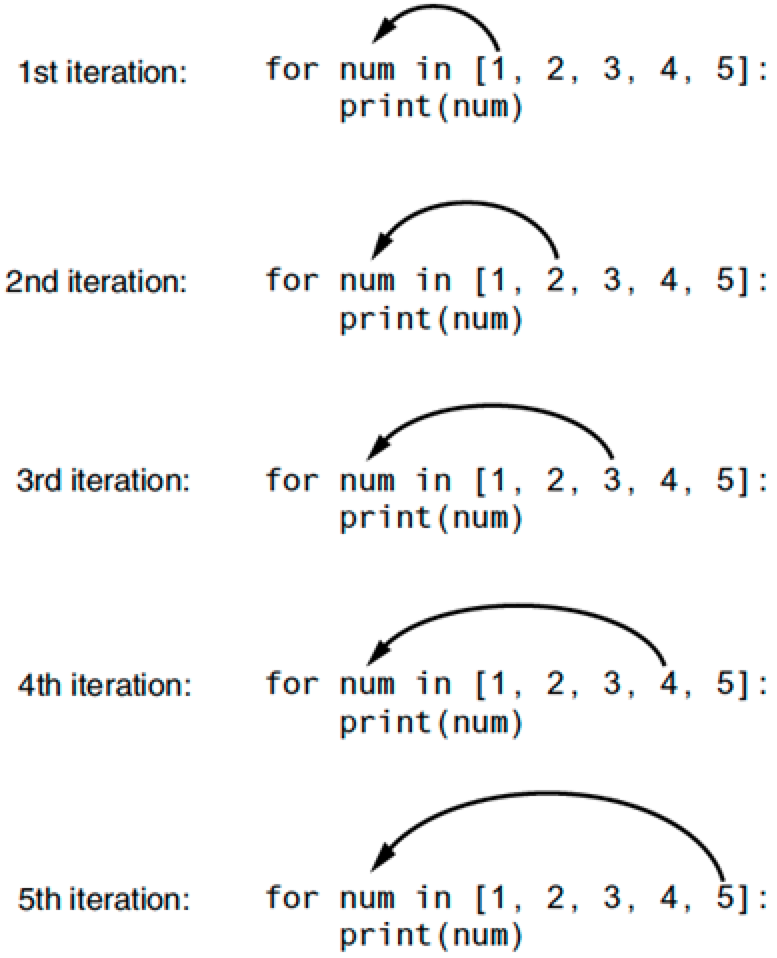
The `for` Statement



```python
print('I will display the numbers 1 through 5.')
for num in [1, 2, 3, 4, 5]:
    print(num)
```



```
In [1]: runcell(2, '/Users/berensemiz/Desktop/
Comp125 - Practice/python_practice3.py')
I will display the numbers 1 through 5.
1
2
3
4
5
```



1st iteration:

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

2nd iteration:

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

3rd iteration:

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

4th iteration:

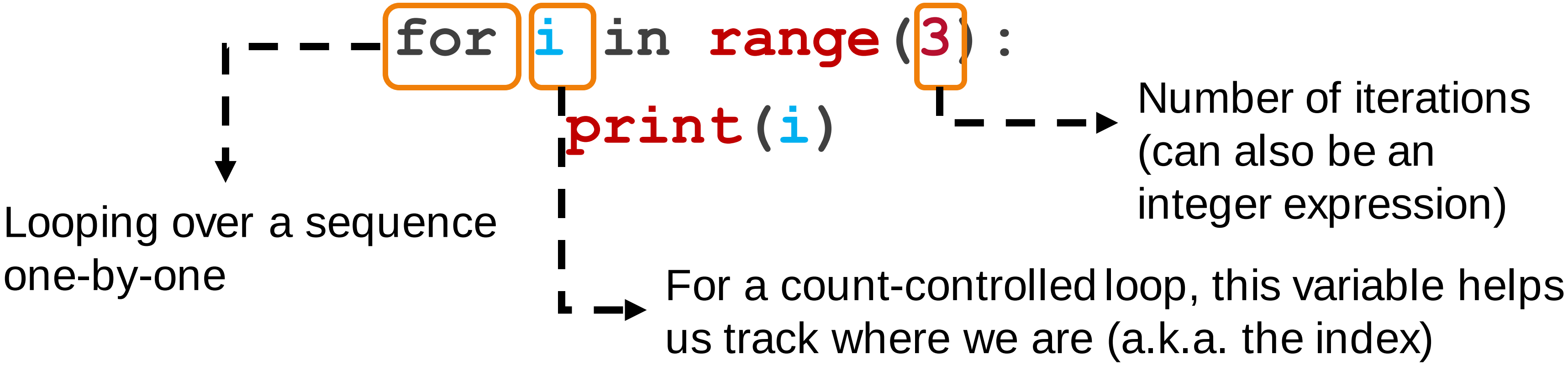
```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

5th iteration:

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

Built-in `range` Function

- `range` function creates an ordered sequence of numbers
  - `range(N)` creates the sequence [0, 1, 2, … N – 1]
  - e.g., `range(3)` creates the sequence [0, 1, 2]
  - Python uses zero-based indexing
- It returns an *iterable* object
  - Iterable: contains a sequence of values that can be iterated over
- Frequently used with the **for** loop to create count-controlled loops



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

Looping over a sequence one-by-one

For a count-controlled loop, this variable helps us track where we are (a.k.a. the index)

Number of iterations (can also be an integer expression)

Built-in `range` Function

- `range(end_index)`
  - with a single argument
  - creates an iterable that returns integers *from 0 to end_index – 1 with increments of 1*
  - creates an empty sequence if end_index <= 0

```
range(5) : [ 0, 1, 2, 3, 4 ]
range(-3): [ ]
```

Built-in `range` Function

- **`range(start_index, end_index)`**
  - with two arguments
  - creates an iterable that returns integers *from start_index to end_index – 1 with increments of 1*
  - creates an empty sequence if end_index <= start_index

```
range(2, 6) : [ 2, 3, 4, 5 ]
range(-3, 2): [ -3, -2, -1, 0, 1 ]
range(4, 2) : [ ]
range(4, 4) : [ ]
```

Built-in `range` Function

- `range(start_index, end_index, step)`
  - with three arguments
  - creates an iterable that returns integers *from start_index to end_index – 1 with increments of step*
  - step can also be a negative number

```
range(2, 11, 3) : [ 2, 5, 8 ]
range(7, -1, -2): [ 7, 5, 3, 1 ]
range(2, 6, 5)  : [ 2 ]
range(4, 1, 2)  : [ ]
range(4, 1, -2) : [ 4, 2 ]
```

Built-in `range` Function

- `range(3) -> [0,1,2]`
- `range(4,7) -> [4,5,6]`
- `range(4,9,2) -> [4,6,8]`
- `range(4,8,2) -> [4,6]`
- `range(9,4,-2) -> [9,7,5]`
- `range(9,5,-2) -> [9,7]`

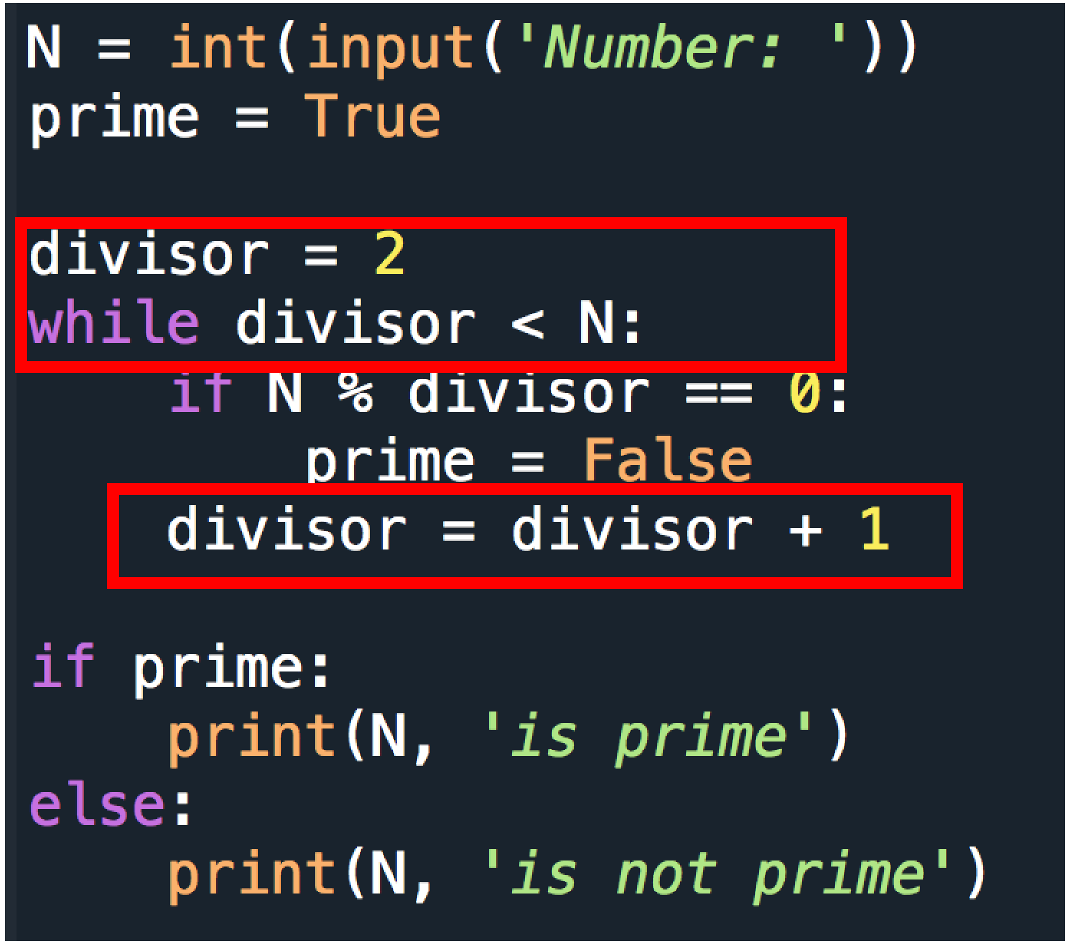
- `range(-1) -> []`
- `range(7,4) -> []`
- `range(4,9,-2) -> []`
- `range(4,8,3) -> [4,7]`
- `range(9,4,2) -> []`
- `range(9,4,-3) -> [9,6]`

All of these can be used in a for loop!

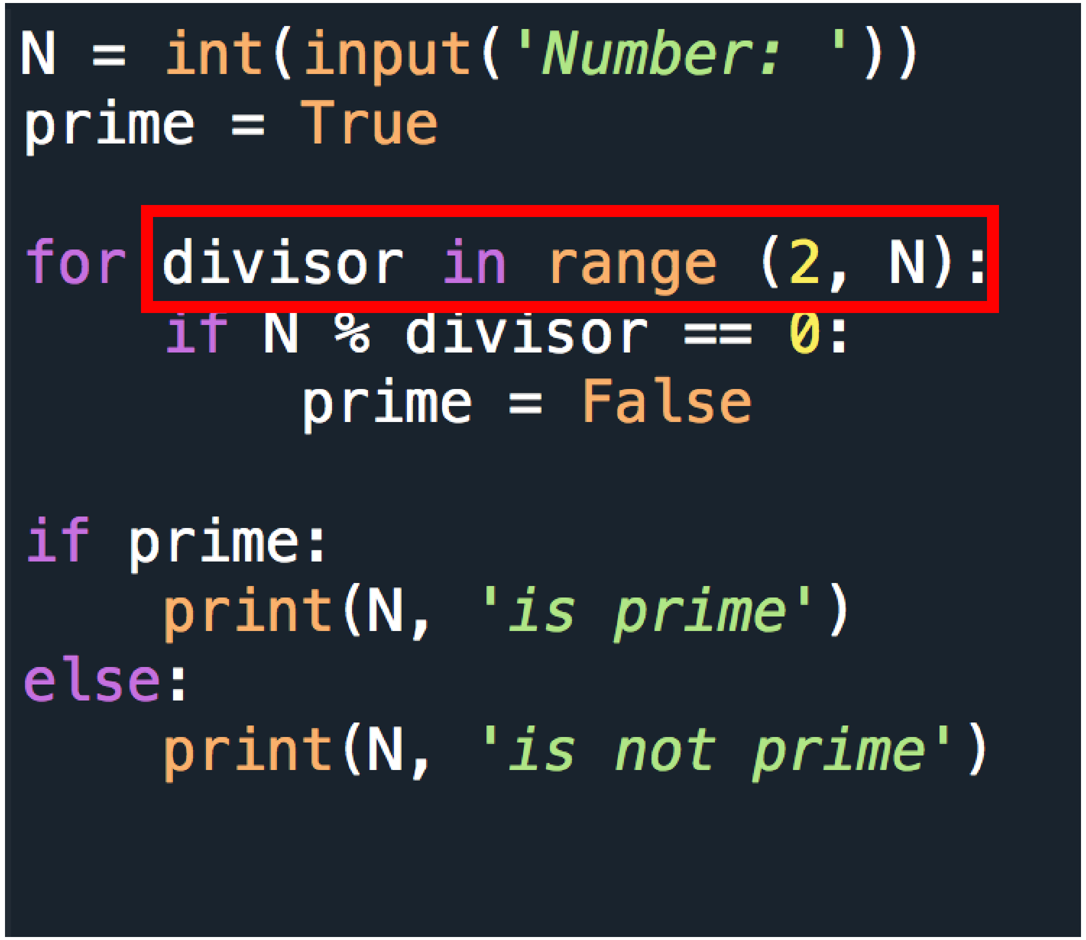
Example: Prime Number

○ Write a program that takes a positive integer from the user and displays whether it is prime or not

Implement the same program with the for statement



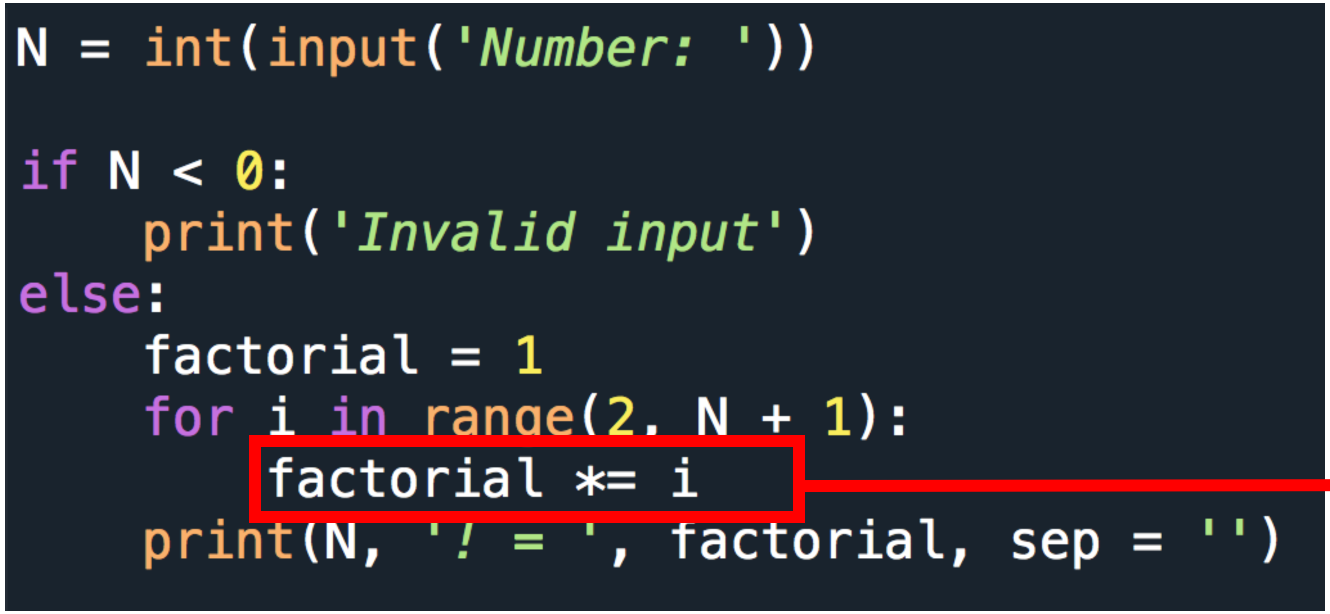
```
N = int(input('Number: '))
prime = True

divisor = 2
while divisor < N:
    if N % divisor == 0:
        prime = False
    divisor = divisor + 1

if prime:
    print(N, 'is prime')
else:
    print(N, 'is not prime')
```



```
N = int(input('Number: '))
prime = True

for divisor in range (2, N):
    if N % divisor == 0:
        prime = False

if prime:
    print(N, 'is prime')
else:
    print(N, 'is not prime')
```

Example: Factorial

- Write a program that takes N and calculates N! = 1 × 2 × … × N
- This program should also check the input validity
  - The factorial of a negative number is not defined
  - 0! = 1



```
N = int(input('Number: '))

if N < 0:
    print('Invalid input')
else:
    factorial = 1
    for i in range(2, N + 1):
        factorial *= i
    print(N, '! = ', factorial, sep = '')
```

`factorial = factorial * i`

Augmented Assignment Operators

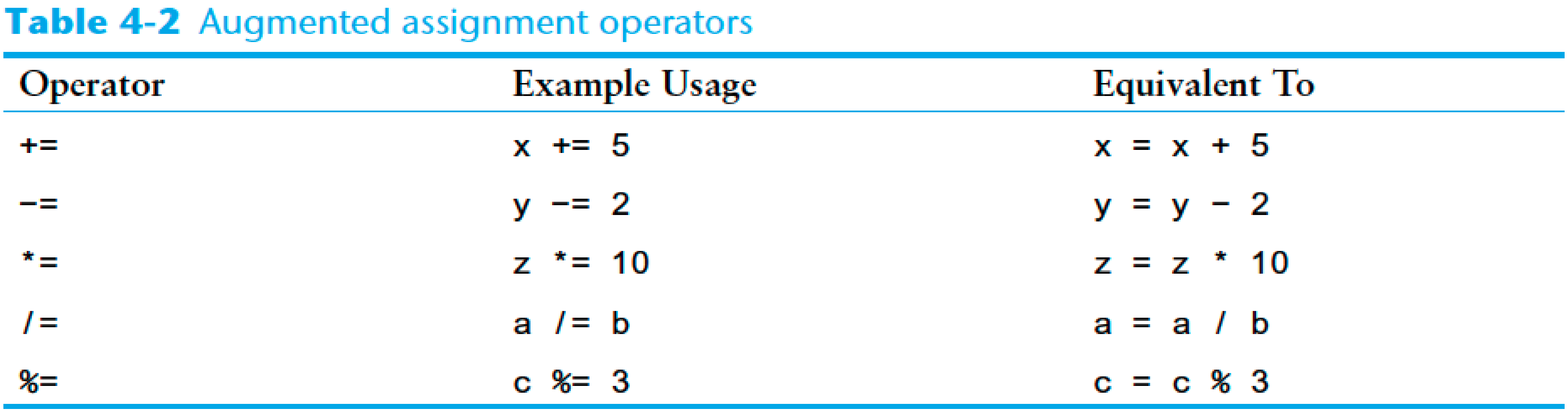
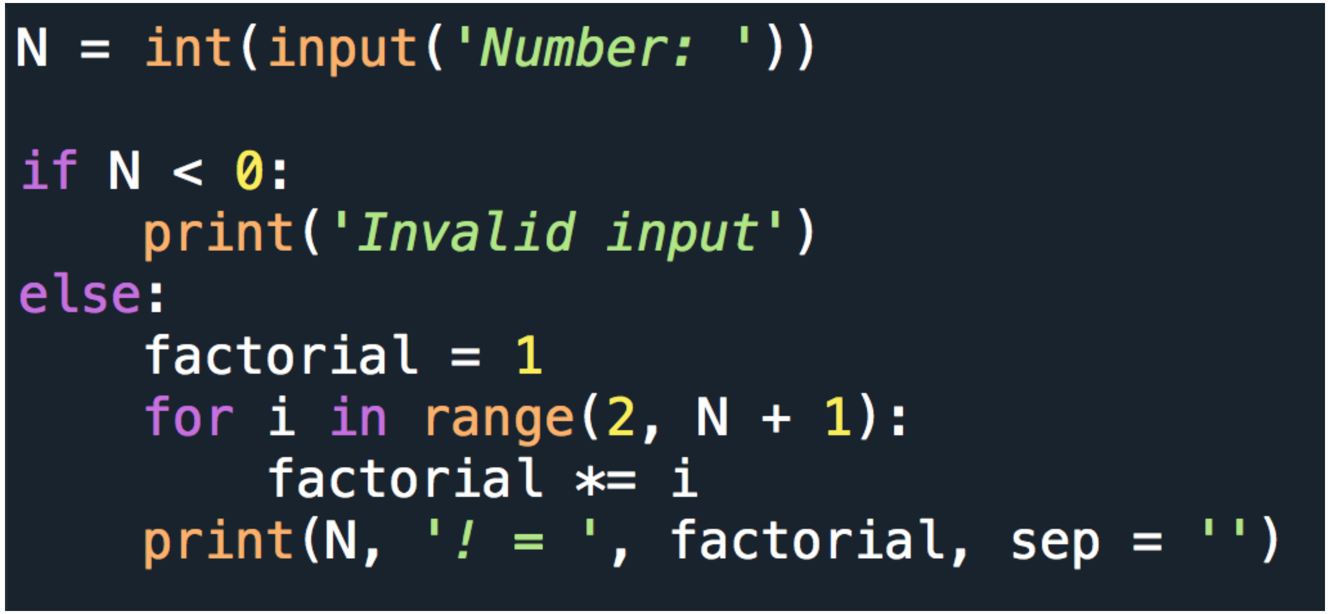


Table 4-2 Augmented assignment operators

Operator	Example Usage	Equivalent To
+=	x += 5	x = x + 5
−=	y −= 2	y = y − 2
*=	z *= 10	z = z * 10
/=	a /= b	a = a / b
%=	c %= 3	c = c % 3

Example: Factorial

- Write a program that takes N and calculates N! = 1 × 2 × … × N
- This program should also check the input validity
  - The factorial of a negative number is not defined
  - 0! = 1

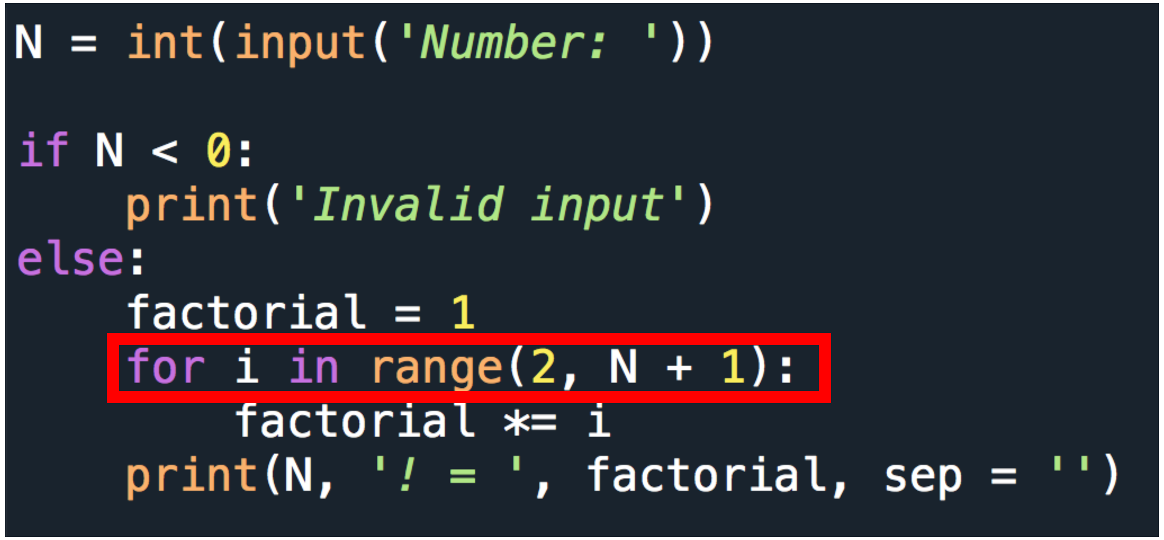


```python
N = int(input('Number: '))

if N < 0:
    print('Invalid input')
else:
    factorial = 1
    for i in range(2, N + 1):
        factorial *= i
    print(N, '! = ', factorial, sep = '')
```

Is it necessary to consider N = 0 as a special case? (i.e., is it necessary to write an if clause for this special case?)

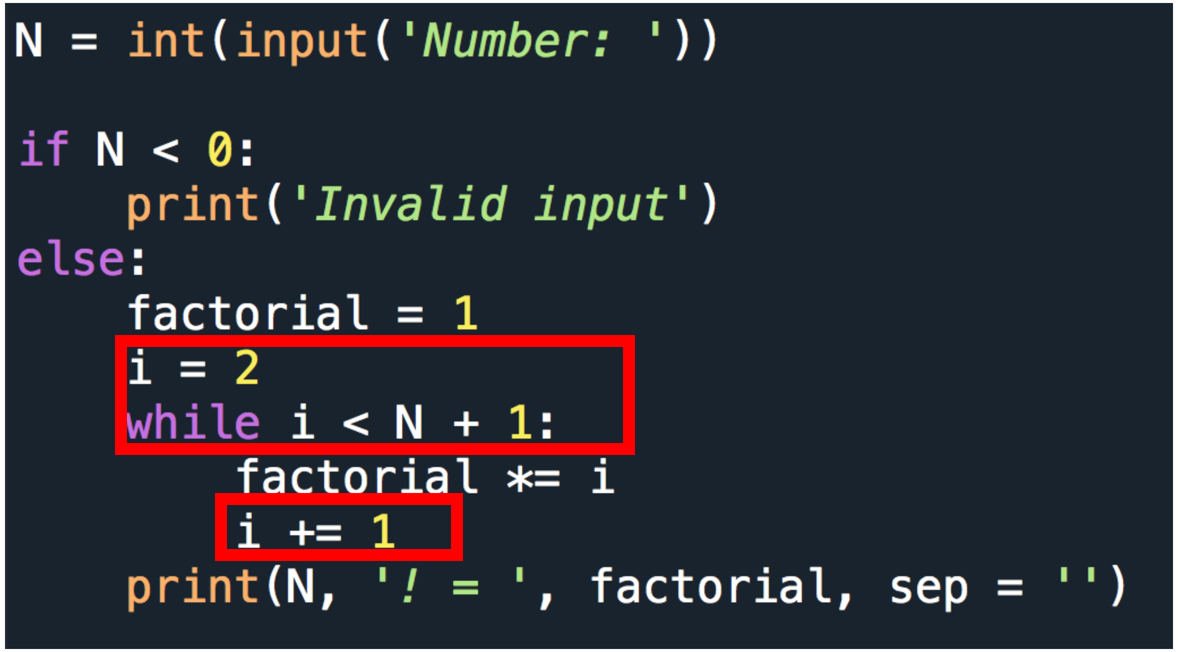
Example: Factorial

- Write a program that takes N and calculates N! = 1 × 2 × … × N
- This program should also check the input validity
    - The factorial of a negative number is not defined
    - 0! = 1



```python
N = int(input('Number: '))

if N < 0:
    print('Invalid input')
else:
    factorial = 1
    for i in range(2, N + 1):
        factorial *= i
    print(N, '! = ', factorial, sep = '')
```

Implement it with the while loop

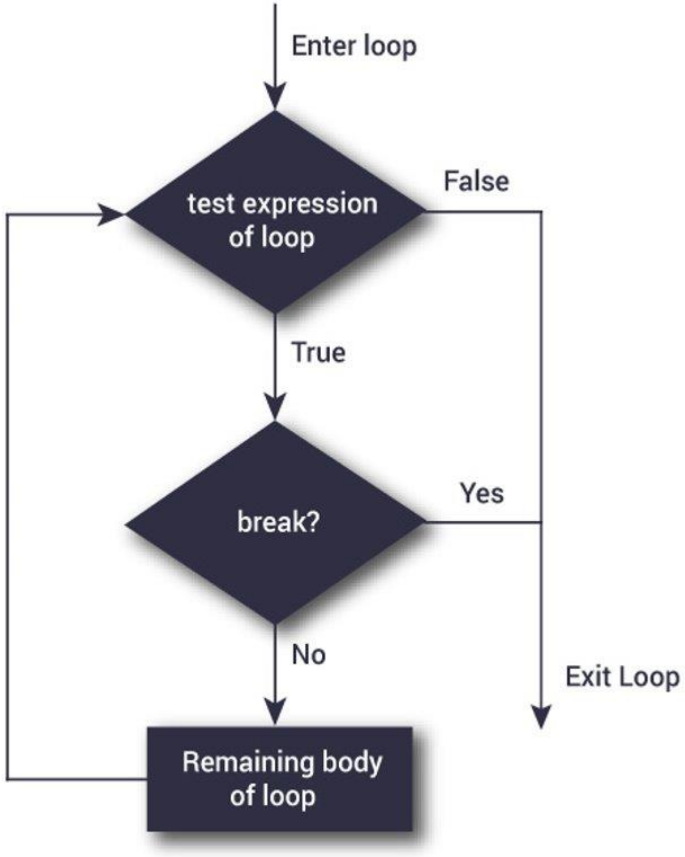


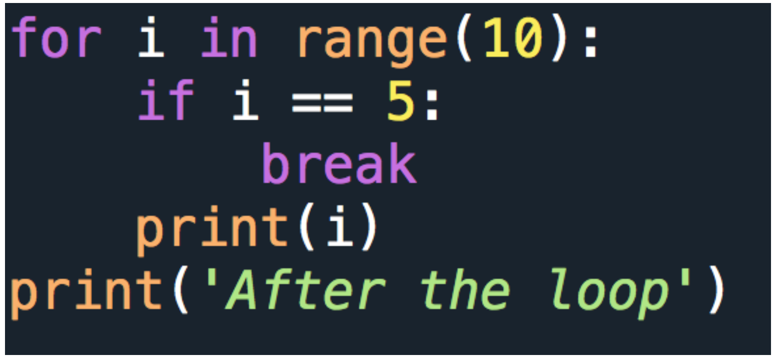
```python
N = int(input('Number: '))

if N < 0:
    print('Invalid input')
else:
    factorial = 1
    i = 2
    while i < N + 1:
        factorial *= i
        i += 1
    print(N, '! = ', factorial, sep = '')
```

Loop Controls

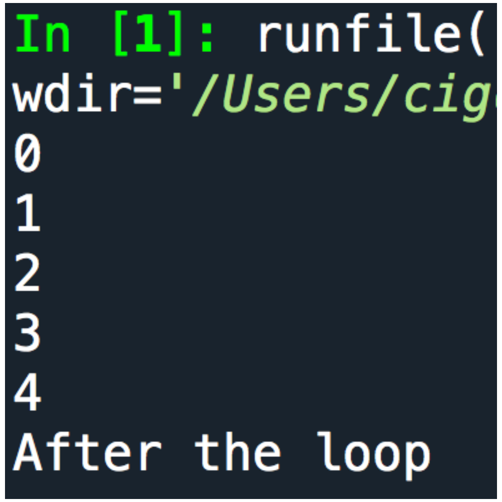
More Control over Loops: `break`

- **break** terminates the loop immediately
- Execution continues with the statements after the loop





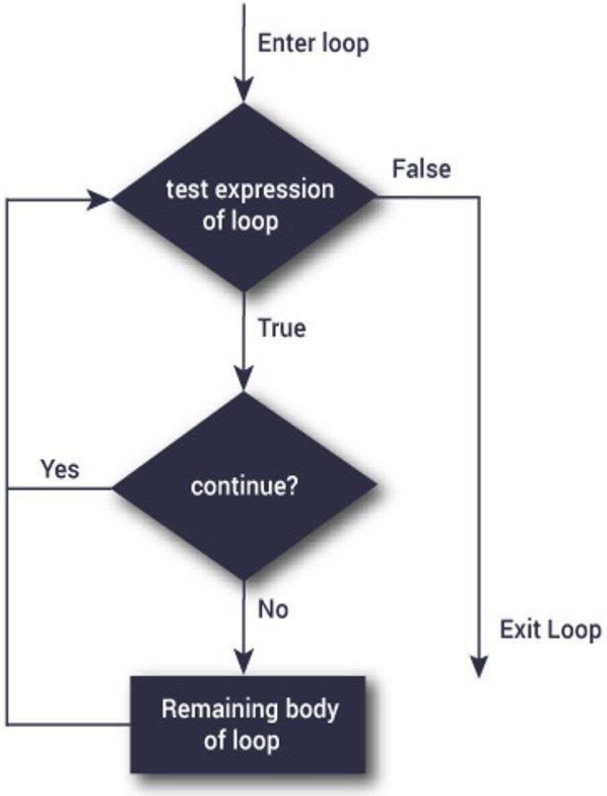
```python
for i in range(10):
    if i == 5:
        break
    print(i)
print('After the loop')
```

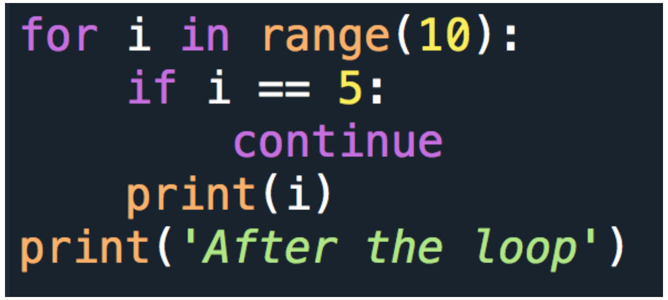


```
In [1]: runfile(
wdir='/Users/cig
0
1
2
3
4
After the loop
```

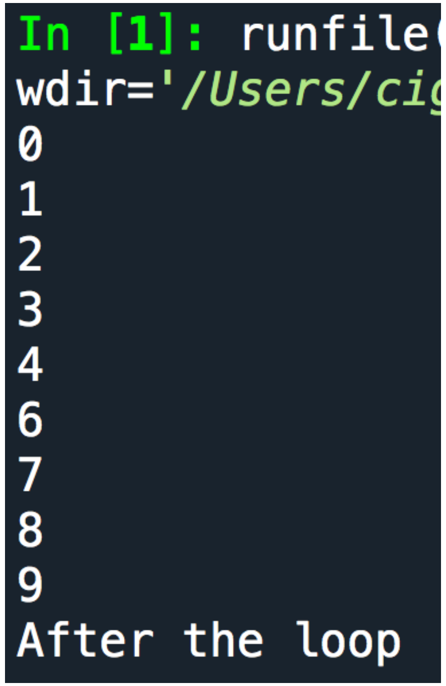
More Control over Loops: `continue`

- `continue` skips the remaining code within the loop and jumps to the start of the loop
- Loop continues with the next iteration





```python
for i in range(10):
    if i == 5:
        continue
    print(i)
print('After the loop')
```

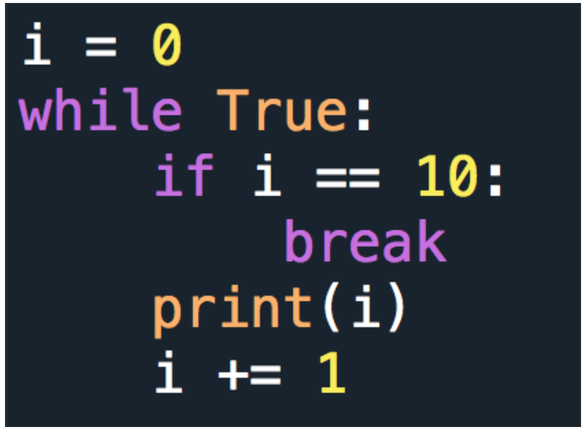


```
In [1]: runfile(
wdir='/Users/cig
0
1
2
3
4
6
7
8
9
After the loop
```

break and continue

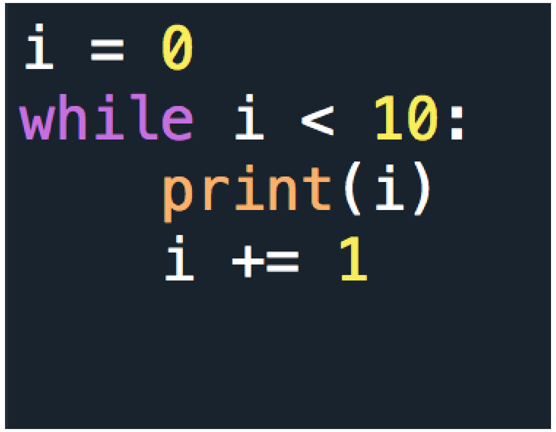
- Do not use **break** and **continue** unless they are really necessary (one good example is the last exercise of this slide set)
- Do not use them as an alternative of writing conditions of a loop statement

Example: Display numbers from 0 to 9



```python
i = 0
while True:
    if i == 10:
        break
    print(i)
    i += 1
```

Poor programming



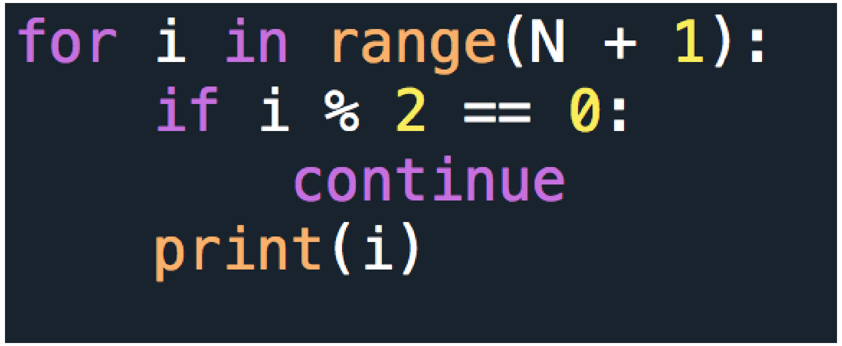
```python
i = 0
while i < 10:
    print(i)
    i += 1
```

Better!!!

break and continue

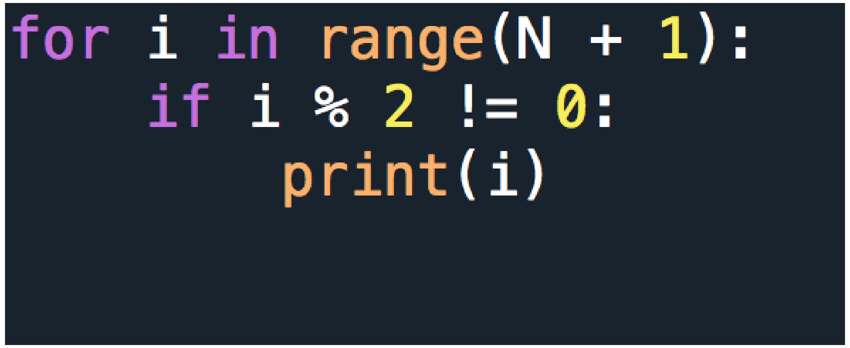
- Do not use **break** and **continue** unless they are really necessary (one good example is the last exercise of this slide set)
- Do not use them as an alternative of writing conditions of a loop statement

Another example: Display odd numbers from 0 to N



```python
for i in range(N + 1):
    if i % 2 == 0:
        continue
    print(i)
```

Poor programming

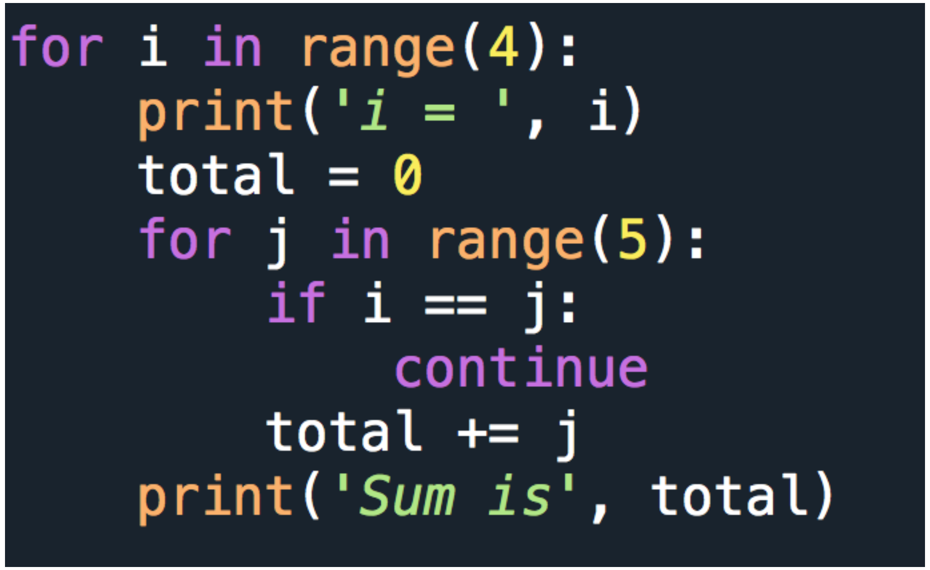


```python
for i in range(N + 1):
    if i % 2 != 0:
        print(i)
```

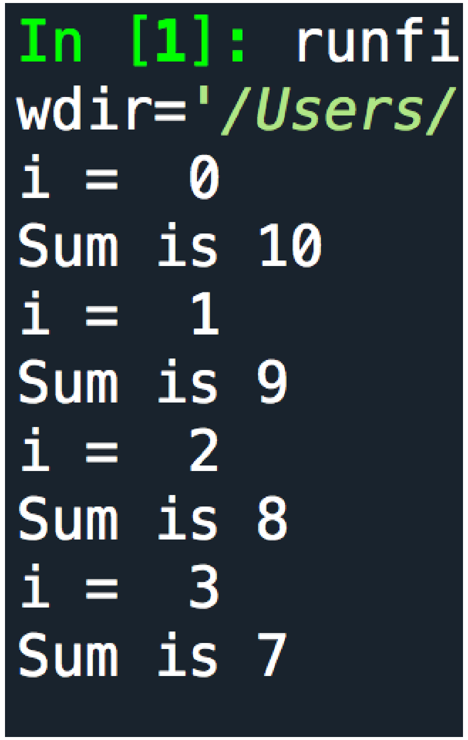
Better!!!

break and continue

- When they are used in nested loops, they affect the loop that they belong to



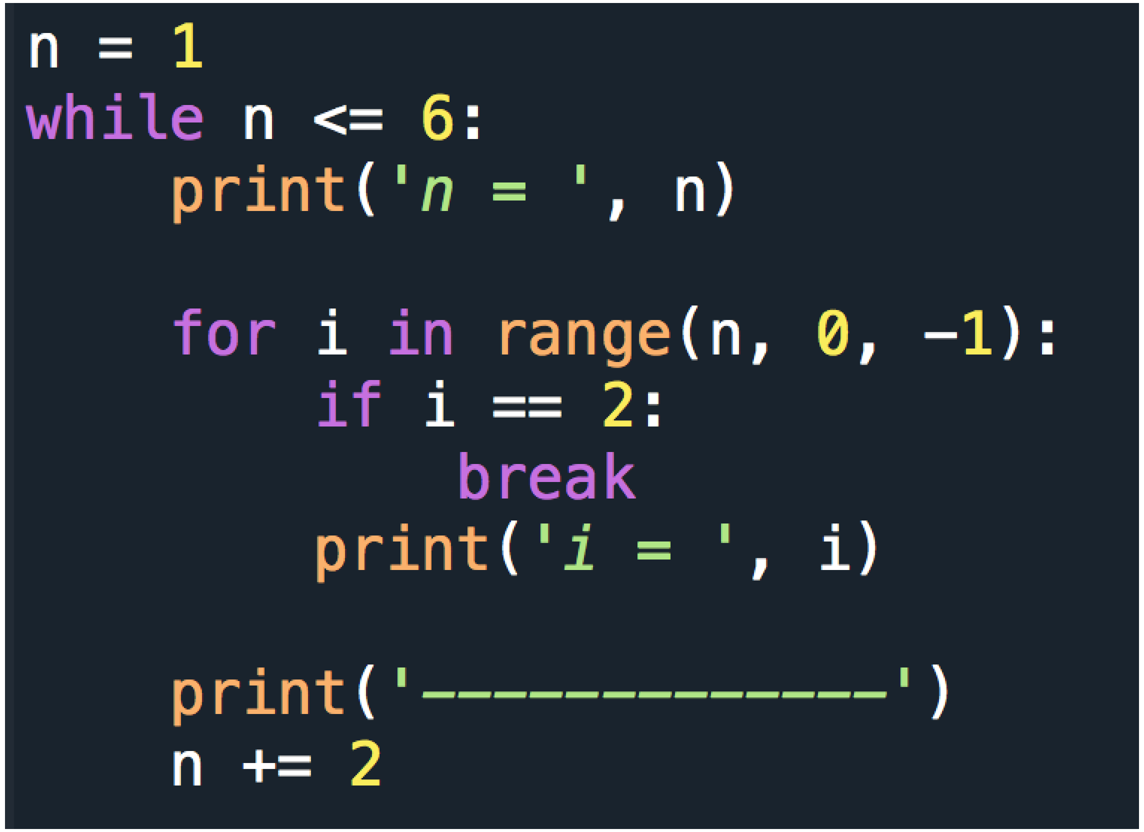
```
for i in range(4):
    print('i = ', i)
    total = 0
    for j in range(5):
        if i == j:
            continue
        total += j
    print('Sum is', total)
```



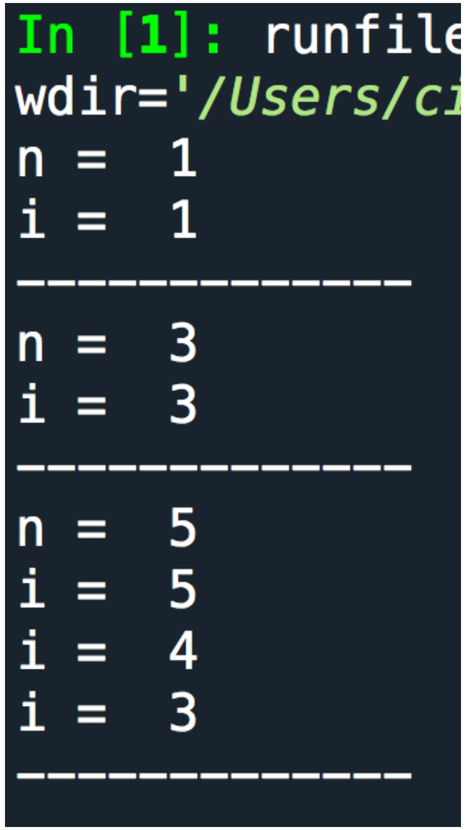
```
In [1]: runfi
wdir='/Users/
i =  0
Sum is 10
i =  1
Sum is 9
i =  2
Sum is 8
i =  3
Sum is 7
```

break and continue

- When they are used in nested loops, they affect the loop that they belong to



```
n = 1
while n <= 6:
    print('n = ', n)

    for i in range(n, 0, -1):
        if i == 2:
            break
        print('i = ', i)

    print('-------------')
    n += 2
```



```
In [1]: runfile
wdir='/Users/ci
n =  1
i =  1
-------------
n =  3
i =  3
-------------
n =  5
i =  5
i =  4
i =  3
-------------
```

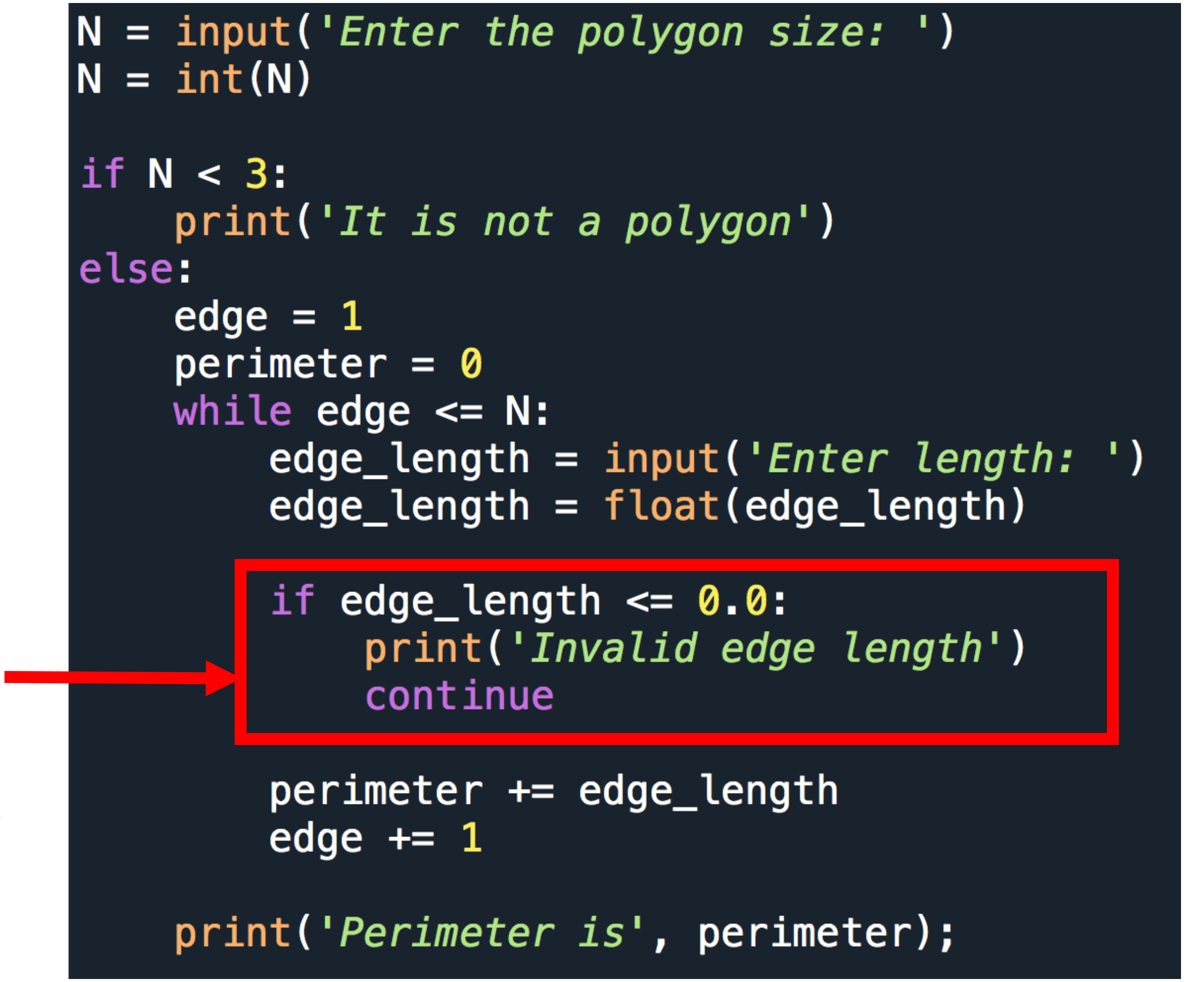
More Examples

Example: Polygon Perimeter

- Write a program to calculate the perimeter of a given polygon. This program takes the polygon size and the length of each edge as inputs. It then displays the perimeter of the polygon on the screen.

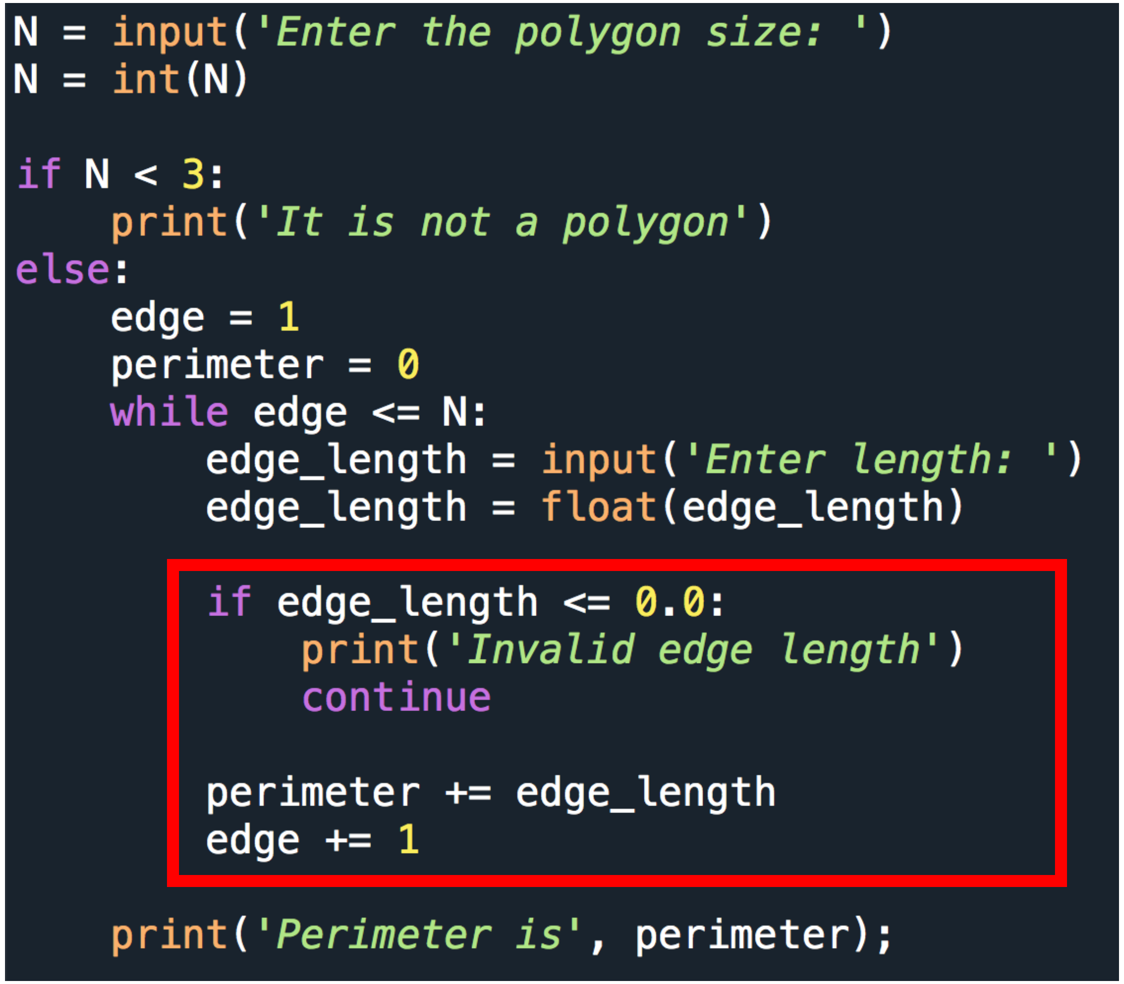
  It should perform validity check on the polygon size and accept only positive edge lengths.

This is an example use of **`continue`**. However, to have better readability, we do not recommend you to use **`break/continue`** in your codes. For example, you can implement this solution using an else clause.

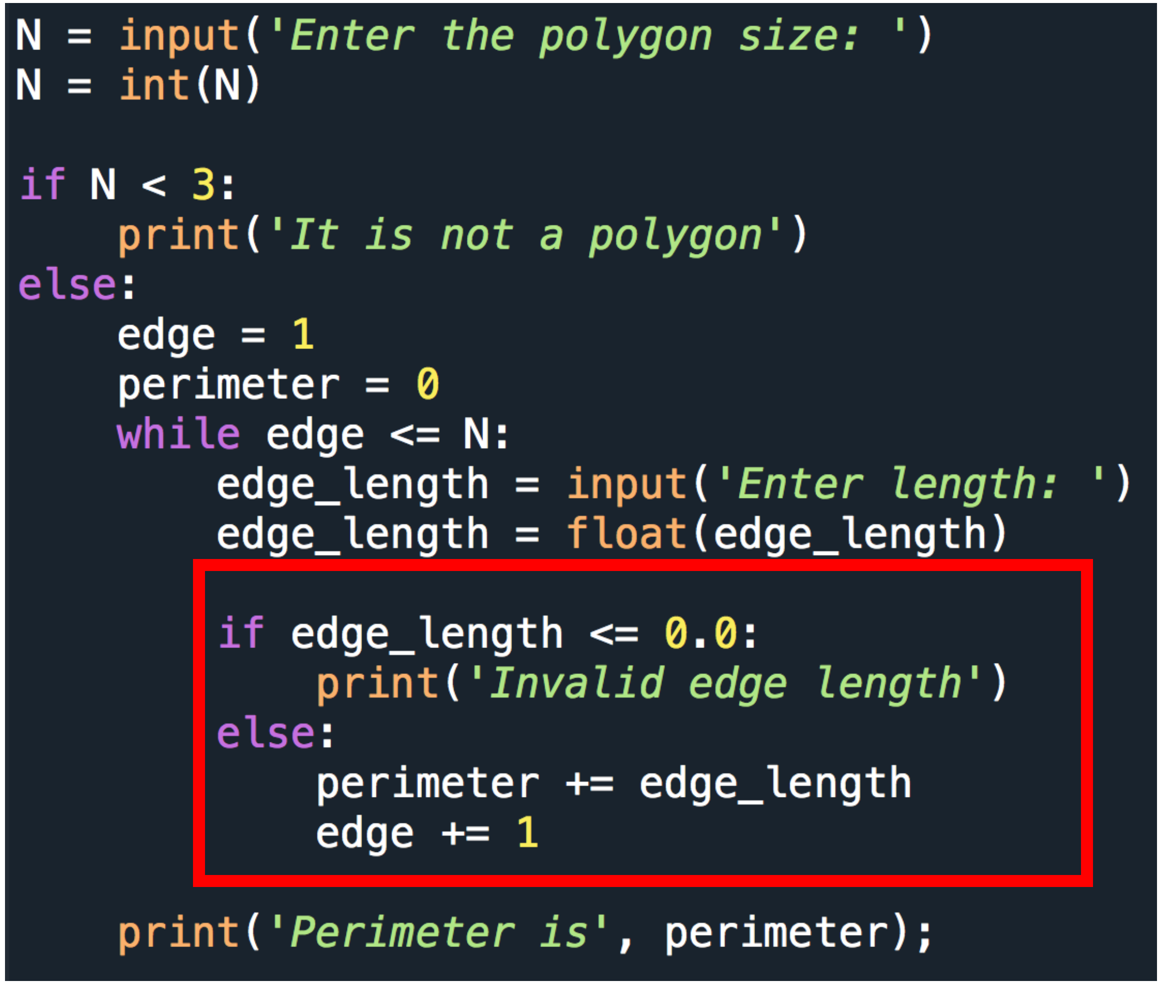


```python
N = input('Enter the polygon size: ')
N = int(N)

if N < 3:
    print('It is not a polygon')
else:
    edge = 1
    perimeter = 0
    while edge <= N:
        edge_length = input('Enter length: ')
        edge_length = float(edge_length)

        if edge_length <= 0.0:
            print('Invalid edge length')
            continue

        perimeter += edge_length
        edge += 1

    print('Perimeter is', perimeter);
```

Example: Polygon Perimeter



```python
N = input('Enter the polygon size: ')
N = int(N)

if N < 3:
    print('It is not a polygon')
else:
    edge = 1
    perimeter = 0
    while edge <= N:
        edge_length = input('Enter length: ')
        edge_length = float(edge_length)

        if edge_length <= 0.0:
            print('Invalid edge length')
            continue

        perimeter += edge_length
        edge += 1

    print('Perimeter is', perimeter);
```



```python
N = input('Enter the polygon size: ')
N = int(N)

if N < 3:
    print('It is not a polygon')
else:
    edge = 1
    perimeter = 0
    while edge <= N:
        edge_length = input('Enter length: ')
        edge_length = float(edge_length)

        if edge_length <= 0.0:
            print('Invalid edge length')
        else:
            perimeter += edge_length
            edge += 1

    print('Perimeter is', perimeter);
```

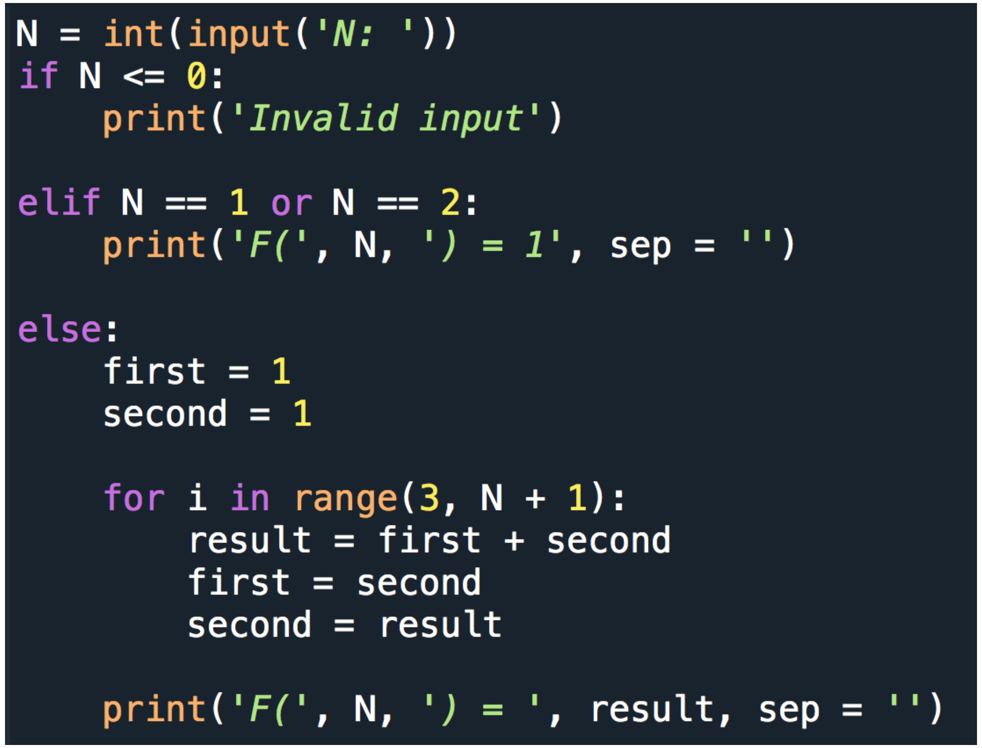
Example: Fibonacci Sequence

- Write a program that calculates and displays the N-th Fibonacci number F(N), which is defined as

  F(1) = 1

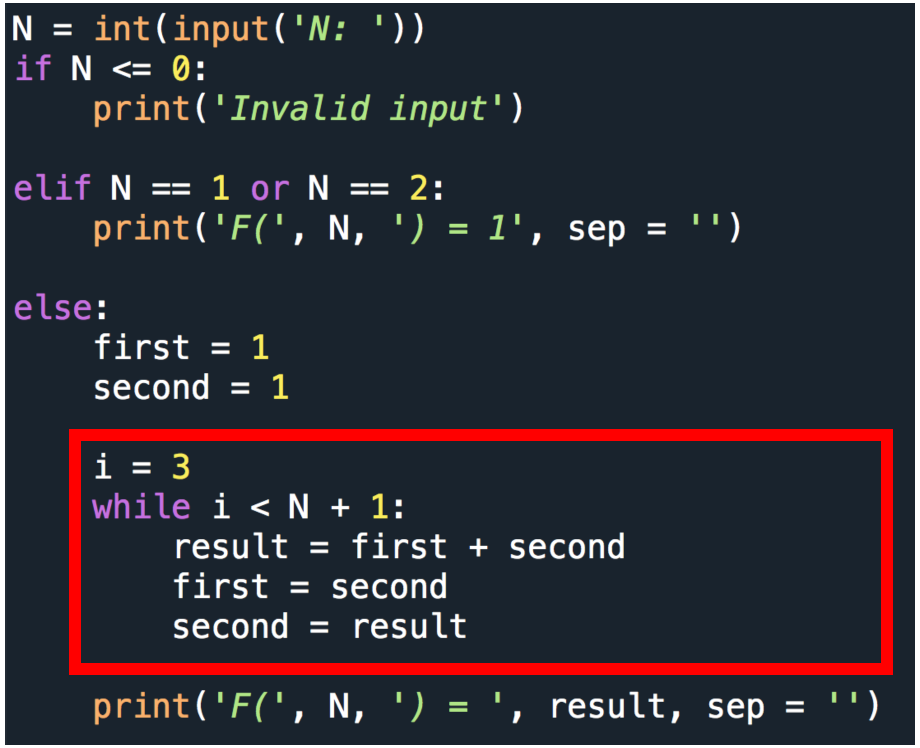
  F(2) = 1

  F(N) = F(N – 1) + F(N – 2) , for all N > 2



```
N = int(input('N: '))
if N <= 0:
    print('Invalid input')

elif N == 1 or N == 2:
    print('F(', N, ') = 1', sep = '')

else:
    first = 1
    second = 1

    for i in range(3, N + 1):
        result = first + second
        first = second
        second = result

    print('F(', N, ') = ', result, sep = '')
```

Implement it with the while loop



```
N = int(input('N: '))
if N <= 0:
    print('Invalid input')

elif N == 1 or N == 2:
    print('F(', N, ') = 1', sep = '')

else:
    first = 1
    second = 1

    i = 3
    while i < N + 1:
        result = first + second
        first = second
        second = result

    print('F(', N, ') = ', result, sep = '')
```

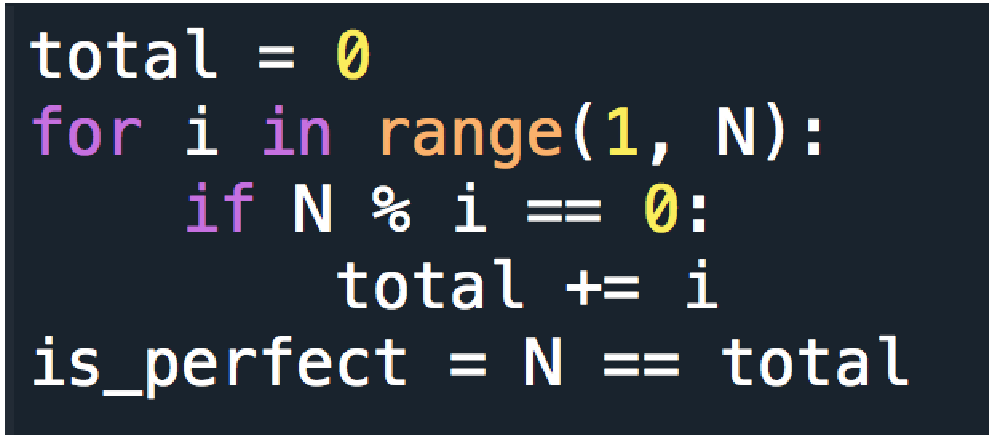
Infinite loop (one common mistake)

How can you fix it?

Example: Perfect Numbers

- A perfect number is a positive integer that is equal to the sum of its positive divisors, excluding the number itself
  - e.g., 28 is a perfect number (1 + 2 + 4 + 7 + 14 = 28)
- Write a program that displays the first M perfect numbers on the screen

First write a code fragment to understand whether a given positive number N is a perfect number

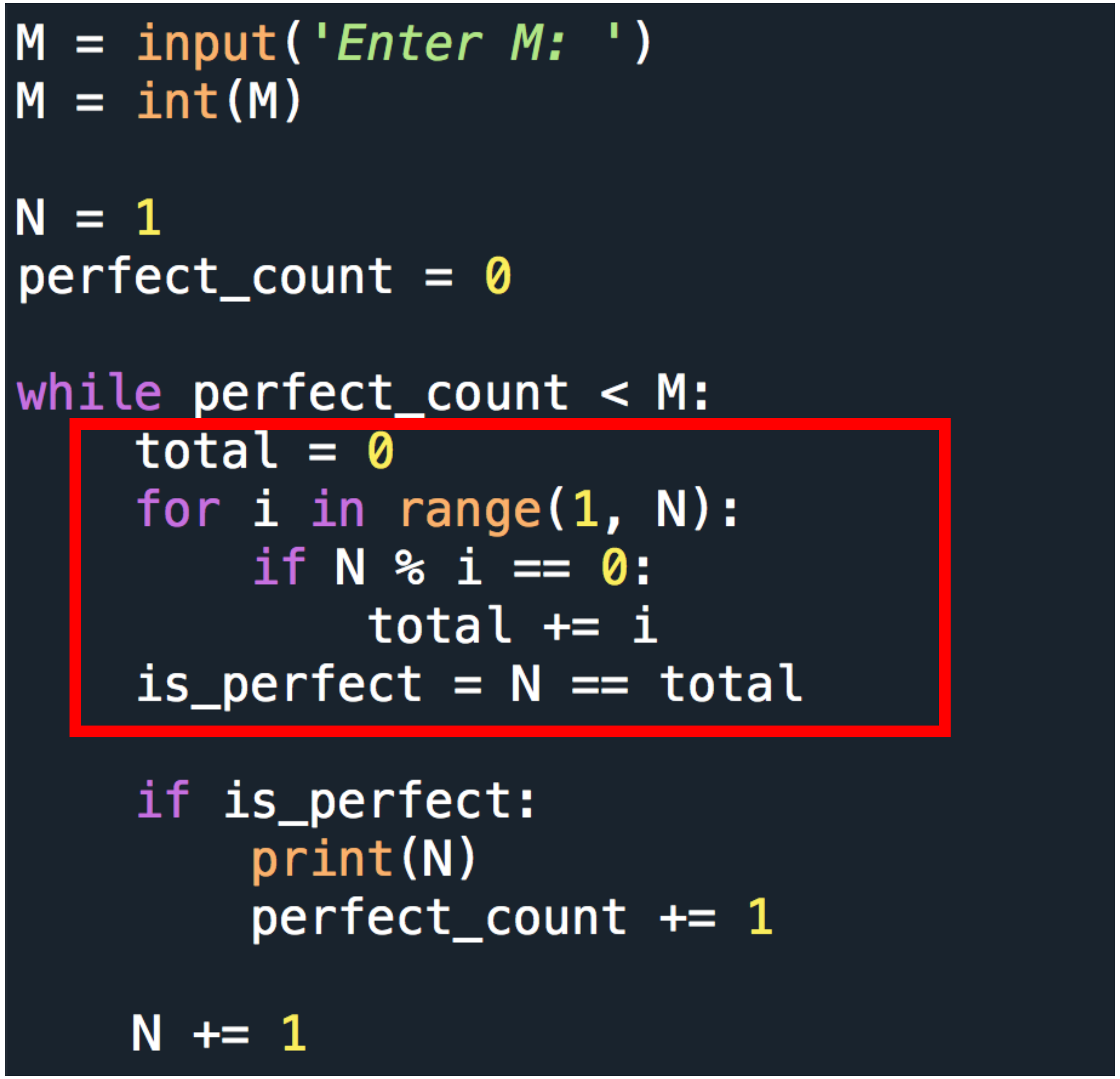


```
total = 0
for i in range(1, N):
    if N % i == 0:
        total += i
is_perfect = N == total
```

Now use this code fragment inside another loop to find the first M perfect numbers

Example: Perfect Numbers

- A perfect number is a positive integer that is equal to the sum of its positive divisors, excluding the number itself
  - e.g., 28 is a perfect number (1 + 2 + 4 + 7 + 14 = 28)
- Write a program that displays the first M perfect numbers on the screen

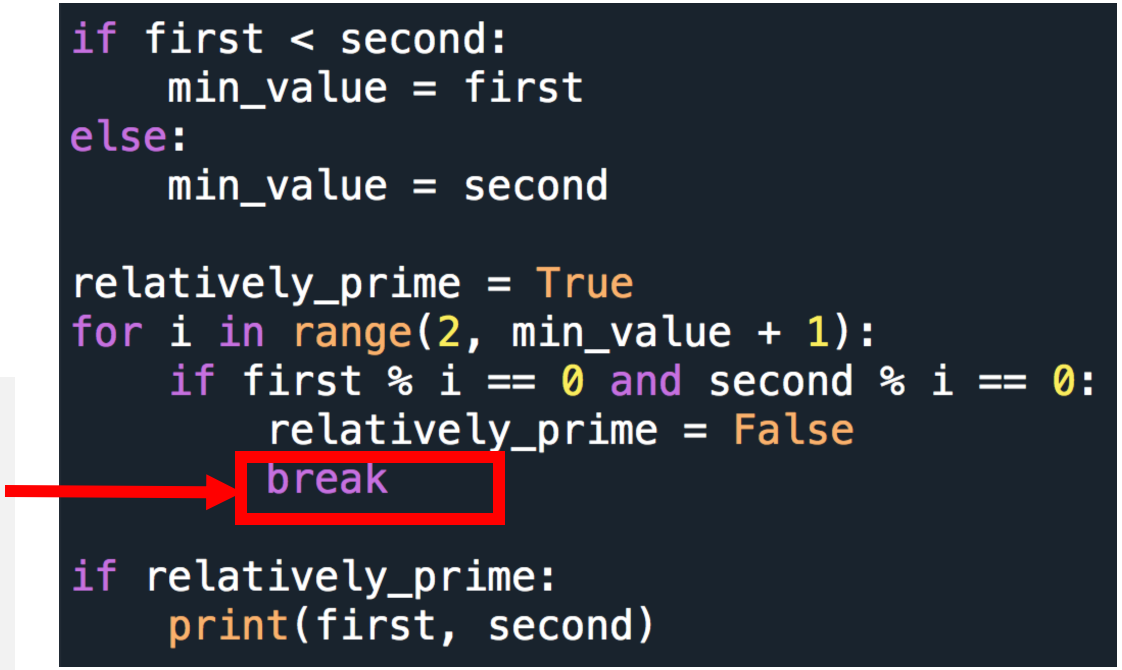


```
M = input('Enter M: ')
M = int(M)

N = 1
perfect_count = 0

while perfect_count < M:
    total = 0
    for i in range(1, N):
        if N % i == 0:
            total += i
    is_perfect = N == total

    if is_perfect:
        print(N)
        perfect_count += 1

    N += 1
```

Example: Co-prime Numbers

- Two integers are called co-prime (or said to be relatively prime to each other) if they do not have any common factors than 1
- Write a program that displays all co-prime number pairs from N to M

First write a code fragment to understand whether two given positive numbers (first and second) are relatively prime



```python
if first < second:
    min_value = first
else:
    min_value = second

relatively_prime = True
for i in range(2, min_value + 1):
    if first % i == 0 and second % i == 0:
        relatively_prime = False
        break

if relatively_prime:
    print(first, second)
```

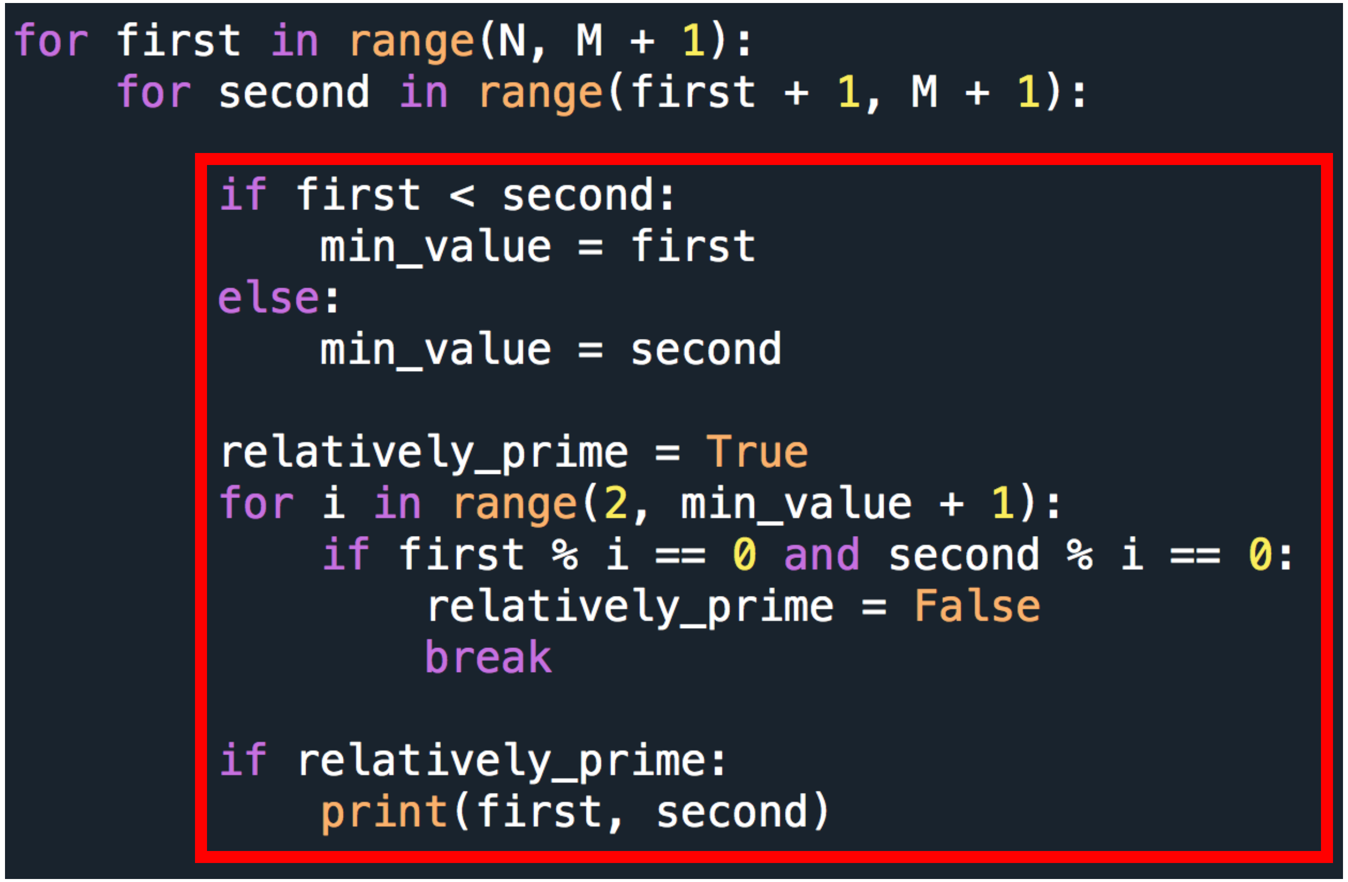
Here **break** is quite useful to prevent unnecessary calculations (once you have found a common factor, it means the numbers are not relatively prime, so no need to examine the other factors)

It will decrease the actual computational time

Now use this code fragment inside other loops to display all co-prime number pairs

Example: Co-prime Numbers

- Two integers are called co-prime (or said to be relatively prime to each other) if they do not have any common factors than 1
- Write a program that displays all co-prime number pairs from N to M



```
for first in range(N, M + 1):
    for second in range(first + 1, M + 1):

        if first < second:
            min_value = first
        else:
            min_value = second

        relatively_prime = True
        for i in range(2, min_value + 1):
            if first % i == 0 and second % i == 0:
                relatively_prime = False
                break

        if relatively_prime:
            print(first, second)
```