

Chapter 3: Loops

Stephen Huang
January 26, 2023

Introduction

- Loops allow a block of statements to be executed repeatedly.
- Sometimes, but not always, there is an “index” that changes from one iteration to another.
- Lists and loops go together. Unavoidably, we will use lists in some examples with a loop.

Iterations

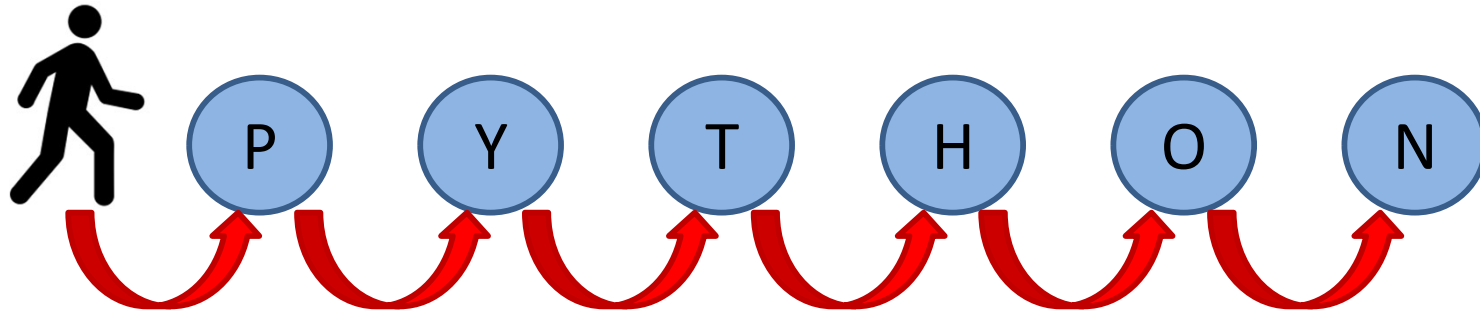
- There are two types of iteration:
 - **Definite** iteration: the number of repetitions is specified explicitly in advance, not necessarily a constant.
 - **Indefinite** iteration: the code block executes until some condition is met.
- Python has two main kinds of loops:
 - For-loop (ideal for Definite Iteration)
 - While-loop (ideal for Indefinite Iteration)
- Example: Repeatedly reading integer numbers until a negative number is entered.

Contents

1. [Iterator and Range](#)
2. [For Loop](#)
3. [While Loop](#)
4. [Nested Loops](#)
5. [Additional Statements](#)
 - Break
 - Else
 - Continue
 - Enumerate() function

1. Iterator

- An **iterator** is an object that contains a countable number of values.
- It can be iterated upon, meaning you can traverse through all the values.
- The values are arranged in some order, and you can go from one to the **next**.



Formally

- An **iterable** object is an object that implements `__iter__`, which is expected to return an **iterator** object.
- An **iterator** is an object that implements `next`, expected to return the next element of the iterable object.
- Example of iterable containers:
 - strings,
 - lists,
 - tuples

Iterable



Iterator

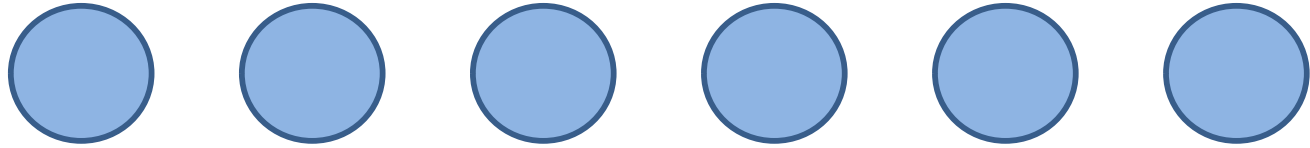
- You can get an **iterator** from an **iterable** container by calling the `iter()` method.
- The `for` loop creates an iterator object and executes the `next()` method for each loop.

Iter-*

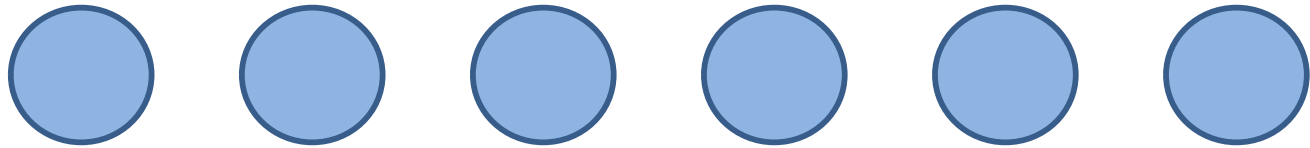
- In Python, an iterable is anything you can iterate over, and an iterator is a thing that does the actual iterating.
- Iterables can be iterated over. Iterators are the agents that perform the iteration.
- You can get an iterator from any iterable in Python using the `iter` function.
- Iterators are stateful. Once consumed, it's gone!
Exhausted!

Iterator

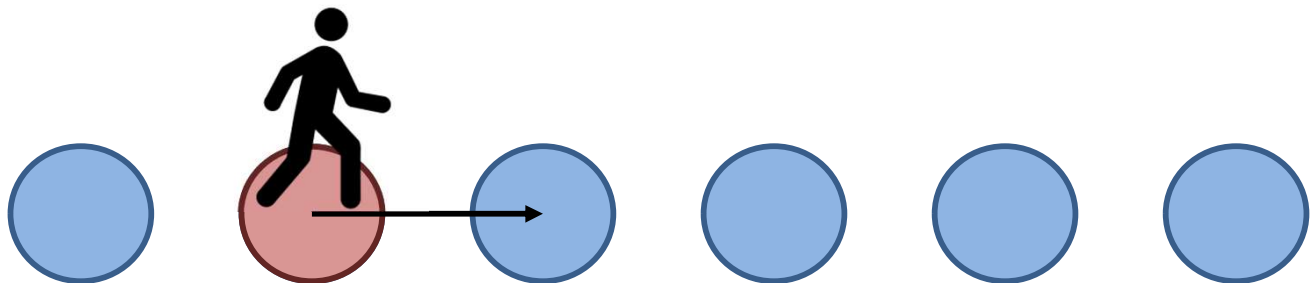
iterable



iterator



Init & next



Lists

- Lists, tuples, dictionaries, and sets are all iterable objects.
- They are iterable *containers* which you can get an iterator from.

apple
banana
cherry

```
mytuple = ("apple", "banana", "cherry")  
myit = iter(mytuple)
```

```
print(next(myit))  
print(next(myit))  
print(next(myit))
```

You get the iterator of the object

The first time you call it, you get the first one

Then you get the next one

Iterable

- Strings are also iterable objects containing a sequence of characters.
- This is special because you can pass a string literal to the `iter()`.

```
myit = iter("Apple")  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))
```

- The for-loop creates an iterator and executes the `next()` method.
- You are NOT going to use `iter()` and `next()`.

Range

- Most of the time, a for-loop is associated with an index within a **range**.

```
range (start, stop [, step])
```

- The three parameters must be integers.
- The default step is 1 and can be omitted.
- If the start is omitted, it defaults to 0.
 - `range (stop)`
 - `range (start, stop)`
 - `range (start, stop, step)`
- `range(n)` is equivalent to `range(0,n,1)`.

Range

- The range type represents an immutable sequence of numbers.

```
class range(stop)
```

```
class range(start, stop[, step])
```

- The arguments to the range constructor must be integers.

```
– range(x, y, z)
```

```
– range(x, y )      range(x, y, 1)
```

```
– range( y )      range(0, y, 1)
```

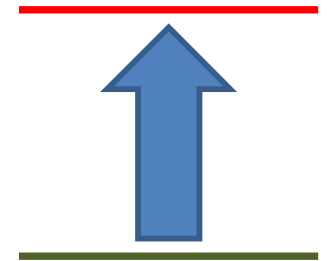
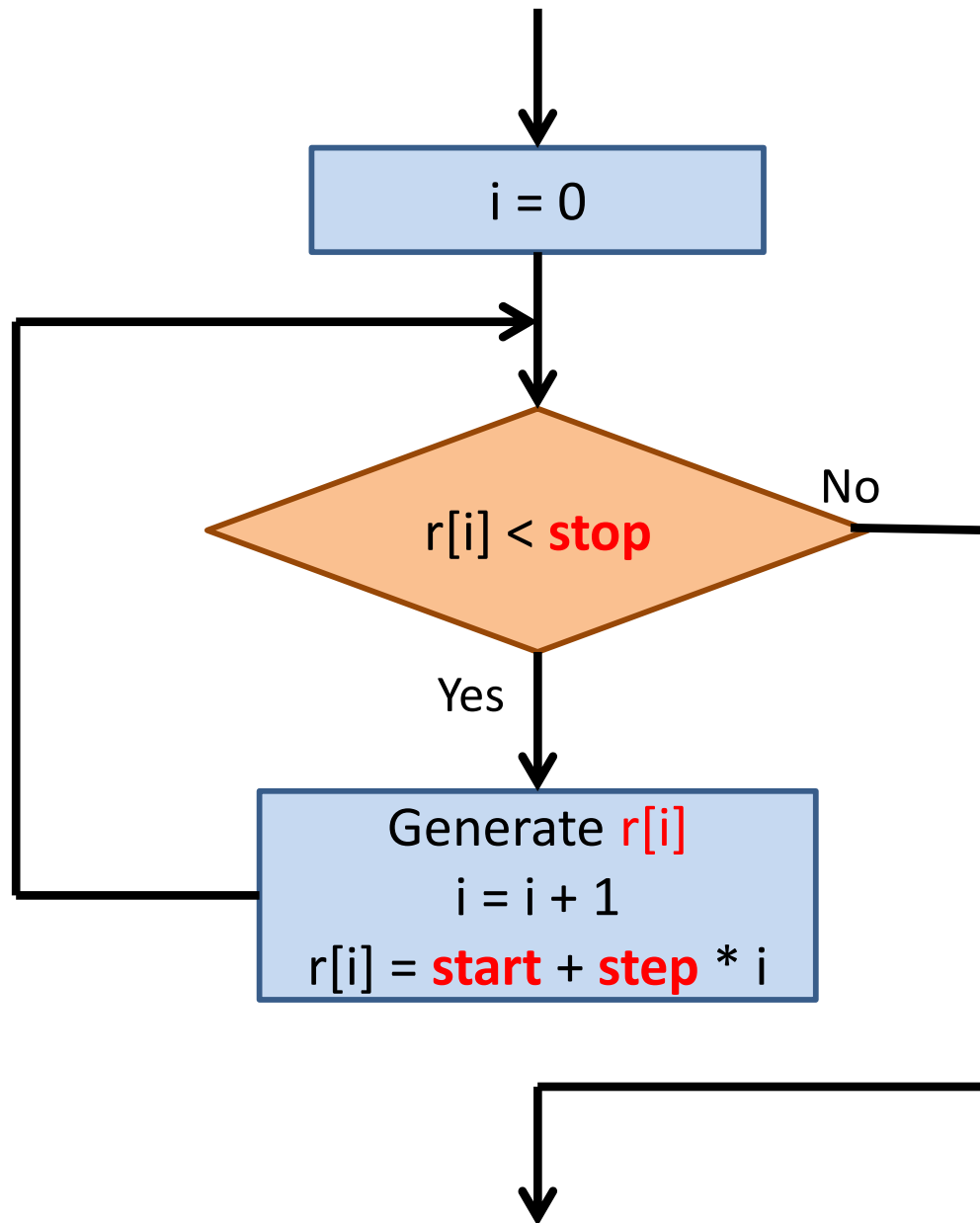
Range

- `Range ()` is commonly used for looping a specific number of times in for loops.
- Technically, it returns a class. We are not going to discuss classes at this time.
- It generates a sequence of numbers (0 or more) based on the parameters given. For example, `range (5)` generates **0**, 1, 2, 3, 4.

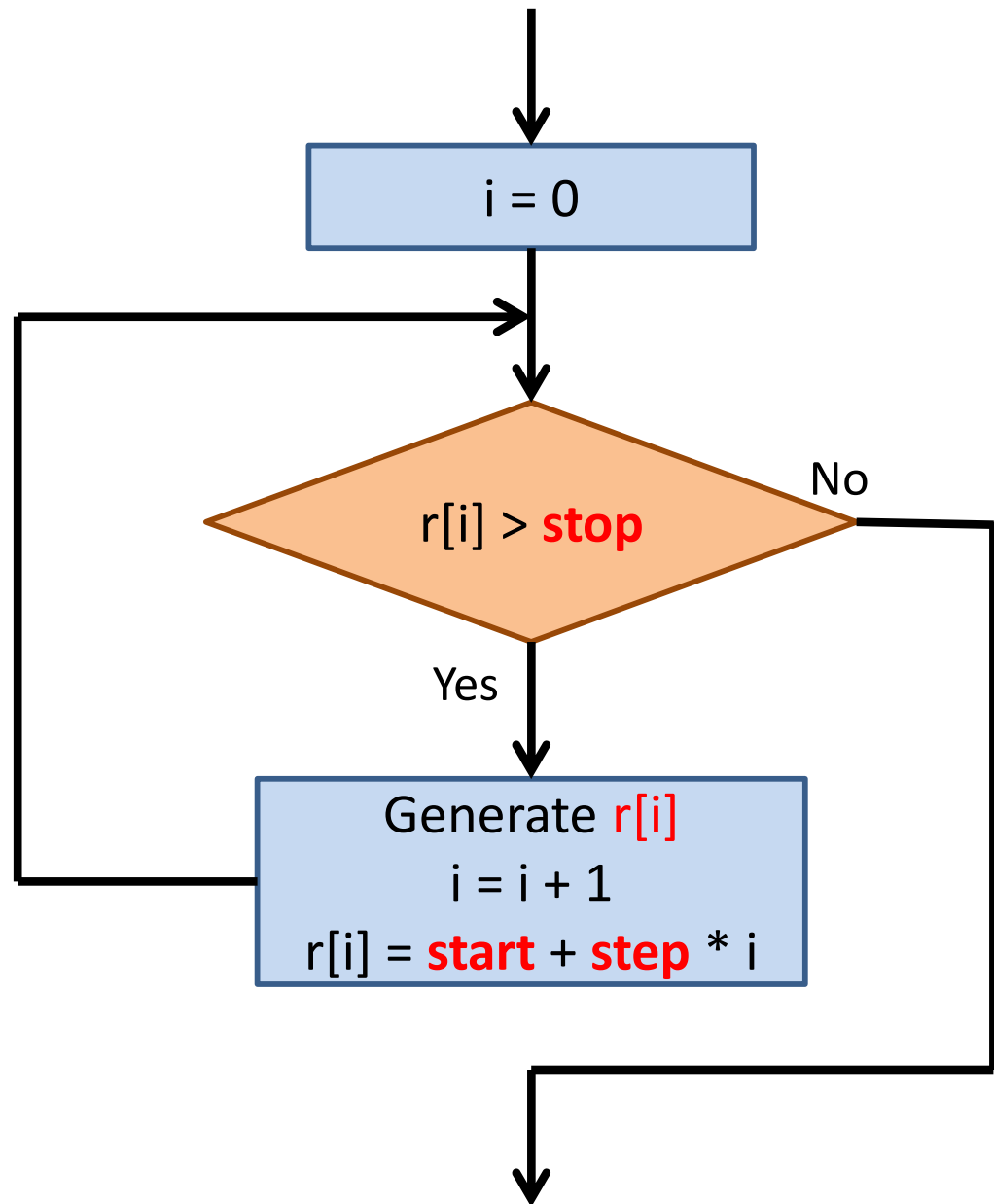
Step

- For a positive *step*, the contents of a range *r* are determined by the formula
 - $r[i] = \text{start} + \text{step} * i$
 - where $i \geq 0$ and $r[i] < \text{stop}$.
- For a negative *step*, the contents of the range are still determined by the formula
 - $r[i] = \text{start} + \text{step} * i$,
 - Where $i \geq 0$ and $r[i] > \text{stop}$.

Range (positive step)



Range (negative step)



Examples

```
print(list(range(5)))           [0, 1, 2, 3, 4]
print(list(range(1, 5)))       [1, 2, 3, 4]
print(list(range(0, 5, 2)))    [0, 2, 4]
print(list(range(0, 10, 3)))   [0, 3, 6, 9]
print(list(range(5, 0, -1)))   [5, 4, 3, 2, 1]
print(list(range(0, 5, -2)))   []
```

Using Range in For-Loop

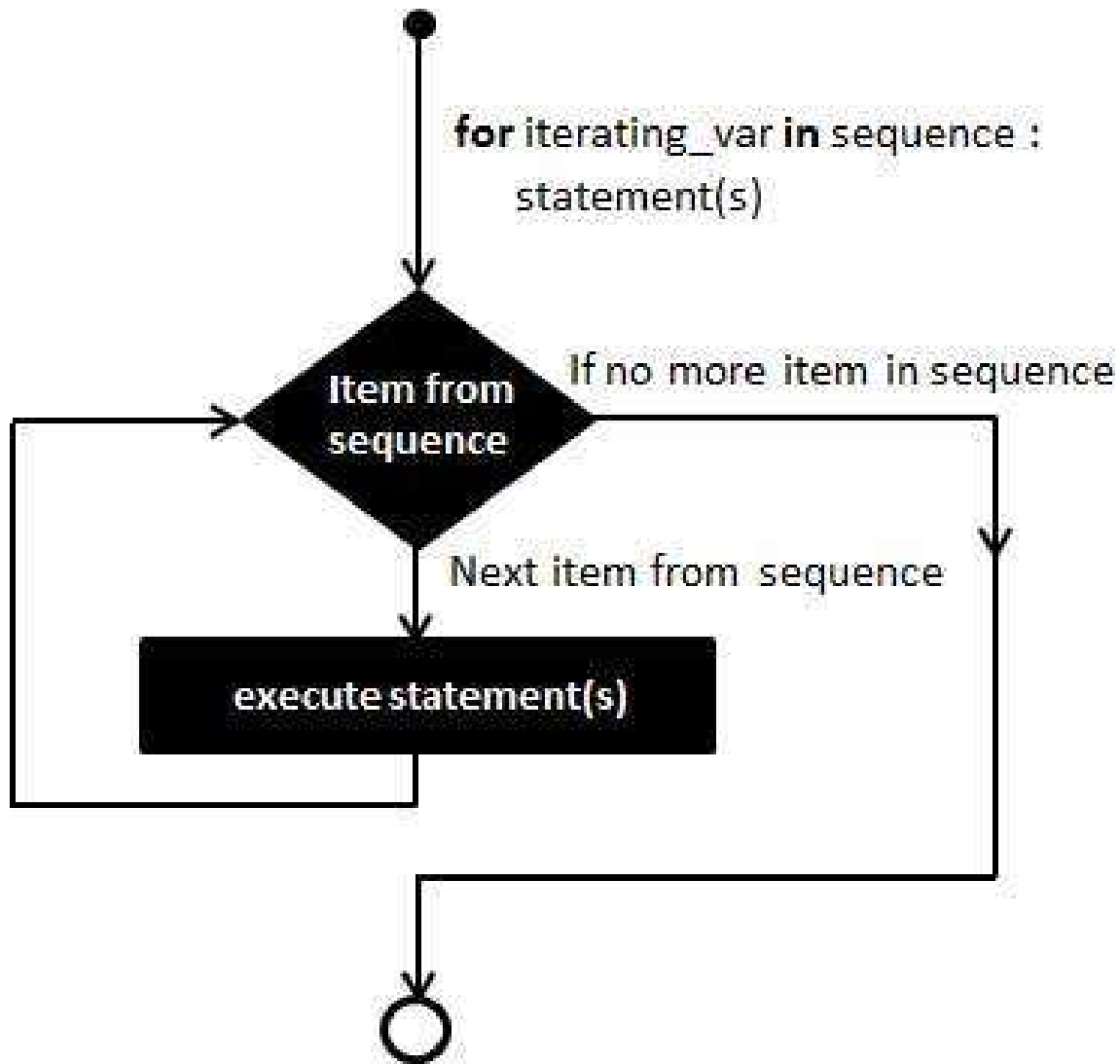
- If you need to iterate over a sequence of numbers in a `for`-loop, the built-in function `range()` comes in handy.
- Other iterators:
 - A list (tuple, dictionary, set)
 - A string

2. For loop

- Python's for-statement iterates over the items of any sequence (a list or a string, for now) in the order that they appear in the sequence.
- Syntax:

```
for <iterating_var> in <iterable>:  
    <statement(s)>
```
- A sequence (such as a string or list) is iterable.

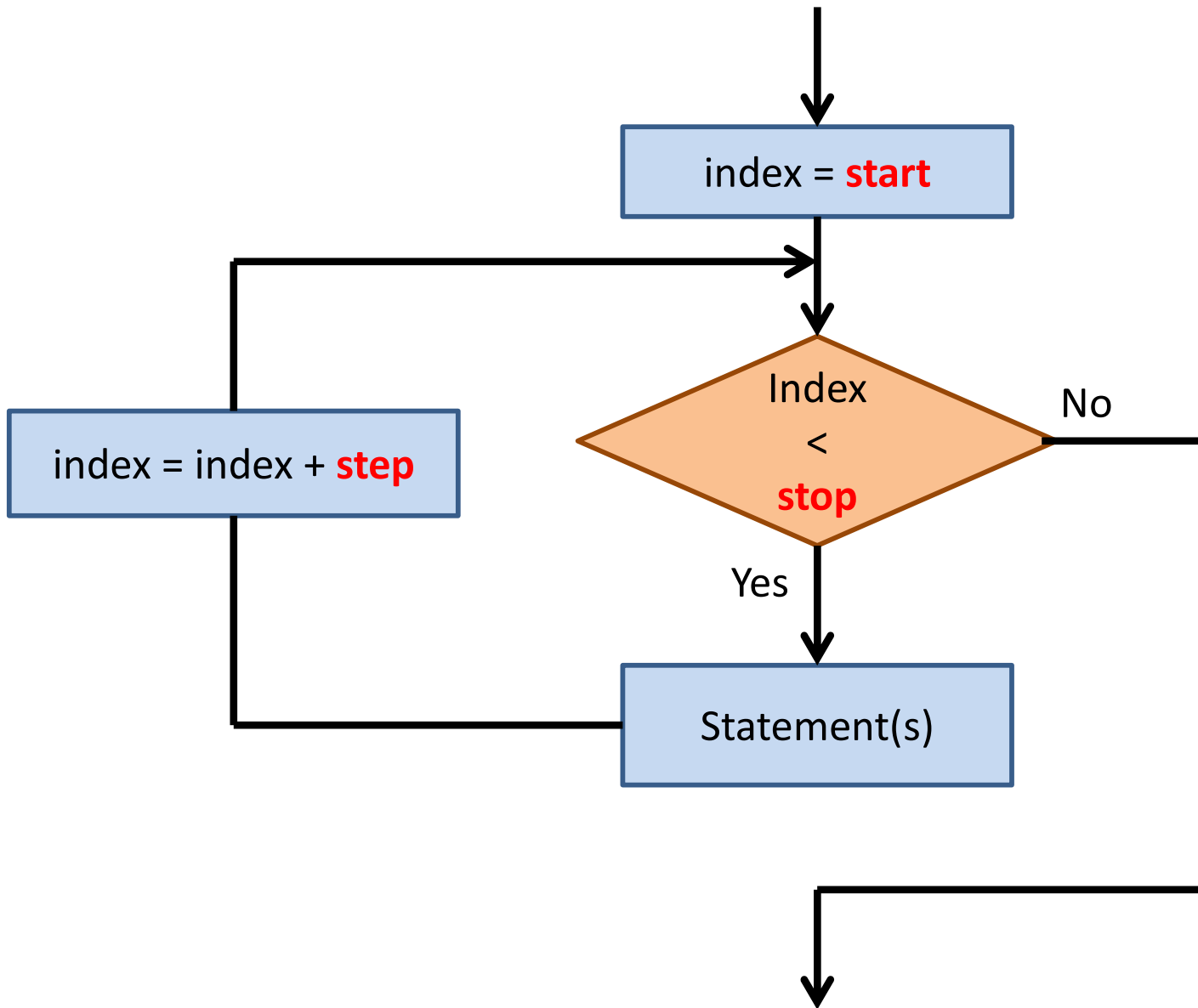
Flow Chart



Range

- Some authors considered `range()` as iterable.
- It is almost iterable with some minor differences.
- The following number is generated as you request it. There is no prepared sequence of all the numbers. (why?)
 - The reason is that the next “item” is predictable; add one to the last number.
 - It also saves memory. Think large: `Range(1000000)`.

Flow chart (Range+)



Examples

```
for letter in 'Python':  
    print ('Current Letter :', letter)
```

```
fruits = ['banana', 'apple', 'mango']  
for fruit in fruits  
    print ('Current fruit :', fruit)
```

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

```
for i in range(10):  
    print (i)
```

For Loop

- The for statement is used to iterate over the elements of a sequence (a string, range, tuple, or list) or other iterable objects.
- The expression list that generates the list is evaluated only once.
- Fortunately, Python
 - gets the iterator for the for-loop automatically, and
 - calls next() repeated
 - Until there is no more next item.

Caution

- It's better not to change the index of a range. Range(n) returns a sequence that cannot be changed (even though you can change n).
- It is dangerous to modify the sequence inside the loop. Please don't do it.
- If you need to modify the sequence you are iterating over inside the loop, it is recommended that you first make a copy of the variable.

Caution

- The first loop is the same as the second one.
- They both print 0 to 4.
- What if you want to print 1 to 5, not 0 to 4? See the last two loops.
- How about 4 down to 0?

```
for i in range(5):  
    print(i, end=' ')
```

```
for i in range(0, 5, 1):  
    print(i, end=' ')
```

```
for i in range(5):  
    print(i+1, end=' ')
```

```
for i in range(1, 6):  
    print(i, end=' ')
```

[0, 5)

[1, 6)

With Index or Not

```
fruits = ['apple', 'banana', 'cherry']
```

```
for i in range(len(fruits)):  
    print(fruits[i])
```

```
# with no index, better
```

```
for fruit in fruits:  
    print(fruit)
```

Else

- Python allows an optional else statement associated with a loop statement.
- If the else statement is used with a for-loop, the else statement is executed when the loop has exhausted iterating the list (**normal exit**).
- Read "else" as "on normal exist." More to come later.

```
for i in range(1,10,2):  
    print(i)  
else:  
    print('End of the loop')
```

```
1  
3  
5  
7  
9  
End of the loop
```

More

- So far, all the data types you have encountered that are collection or container types are iterable.
- Many objects built into Python or defined in modules are designed to be iterable.
- For example, open files in Python are iterable.
- As you will see later in the lecture on file I/O, iterating over an opened file object reads data from the file.

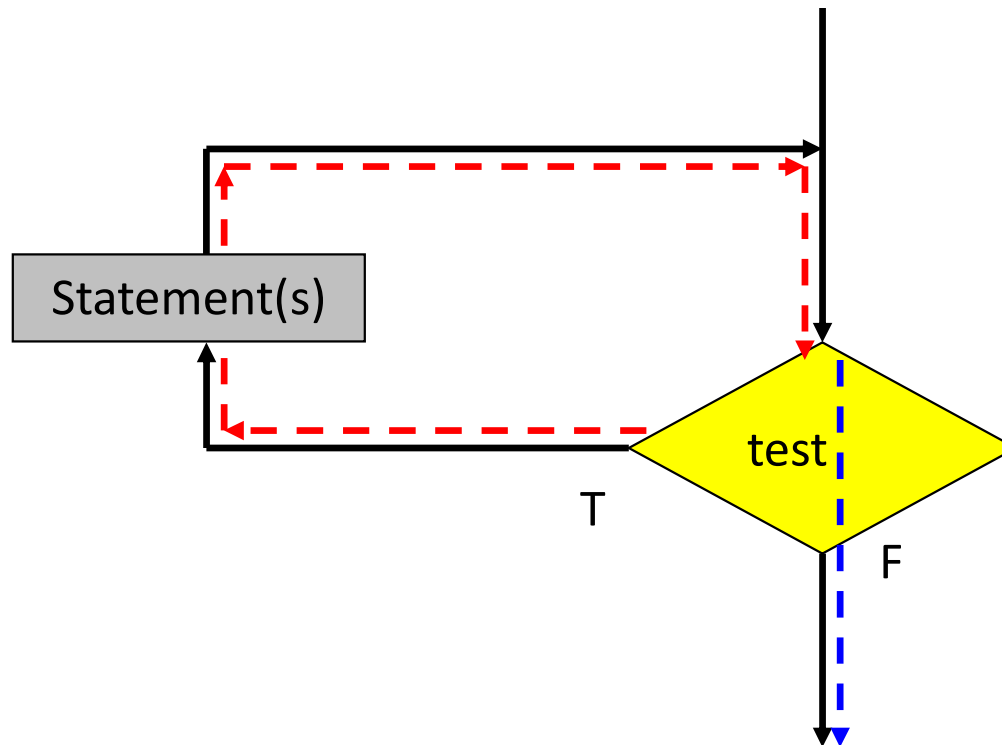
3. While Loop

- The syntax is:

```
while <expression>:  
    <statement(s)>
```

- The expression is evaluated as a Boolean expression.
- Statements should be appropriately indented. Otherwise, there is no way to know where the “block” ends.

While Loop

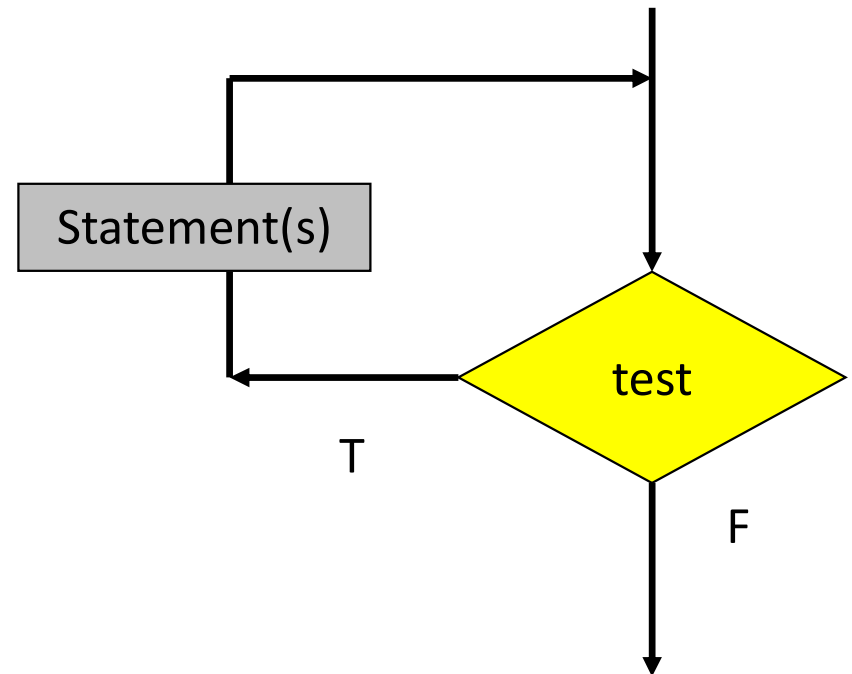


While Statement

- The while statement causes the loop body to be executed repeatedly as long as the test evaluates **True**.

```
while (test)
    statement(s)
```

```
a = 1
while (a < 10)
    print("a = ", a)
    a = a + 1
```



The execution of While

- First, the test is evaluated. If it evaluates to True, the loop body is executed next, and execution returns to the test.
- The test is then re-evaluated, and if it is True, the loop body is again executed, and the execution returns to the test.
- This process continues in a loop while the test evaluates to true.
- When the test evaluates to false, the execution proceeds to the next instruction past the loop body.

While Loop

- Somewhere in the loop body, there must be a change that may cause the test to be **False**. Otherwise, the test will remain true forever. This is called an “Infinite Loop.”
- Note that if the test evaluates to **False** the first time it is evaluated, this will result in execution jumping to the next instruction after the loop.
- Thus, the loop body of a while-statement may execute **0** or more times depending on when the test evaluates to False.

While Loop

- If you want more than one statement executed each time around the loop, indent the statements.
- It is common, though not required, to use an index variable with the while loop.

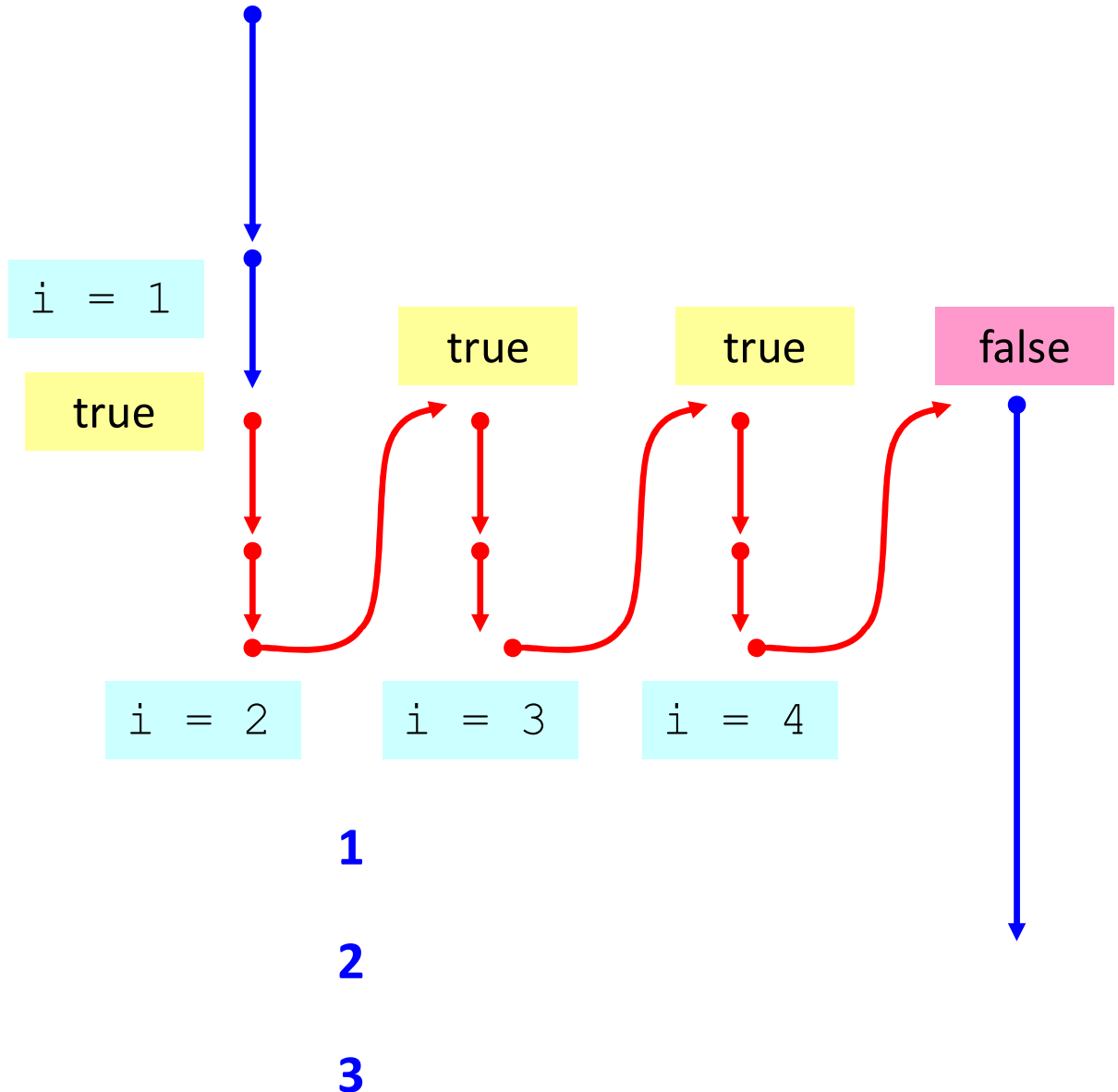
Example

```
i = 1
```

```
while (i <= 3) :
```

```
    print (i)
```

```
    i = i + 1
```



Sentinel

- A common task in programs is to input an indefinite number of values.
- A loop must be used, and there must be a way to signal that all the data has been entered.
- The user can enter as many data values as he chooses before stopping the process by entering the sentinel value.
- A for-loop is probably not the right choice.

Sentinel

- The programmer chooses a particular value that the user can enter to signal the end of input to achieve this.
- The value is a **sentinel** since it stands guard at the end of the data entry and stops the process.
- One usually does not want to process the sentinel value because it is not part of the data but just a way to stop the input.

Solution 1

```
sum = 0
num = 0
value = int(input("Enter an Positive int: "))

while value != -1:
    sum = sum + value
    num = num + 1
    value=int(input("Enter a positive int: "))

print("Sum of ", num, "numbers: ", sum)
```

Solution 2

```
sum = 0
num = 0

while True:
    value=int(input("Enter a positive int: "))
    if value == -1:
        break
    else:
        sum = sum + value
        num = num + 1

print("Sum of ", num, "numbers: ", sum)
```

Solution 3

```
sum = 0
num = 0
done = False

while not done:
    value = int(input("Enter a positive int: "))
    if value == -1:
        done = True      # one-way switch
    else:
        sum = sum + value
        num = num + 1

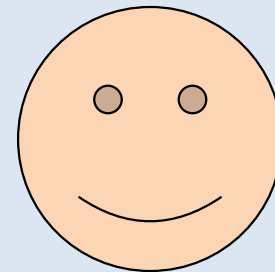
print("Sum of ", num, "numbers: ", sum)
```

while vs. for

```
i = 0
while i <= 4:
    print(i)
    i = i + 1
```

```
for i in range(5):
    print(i)
```

Which one do you like better?
Why?



Example

- Here is a crude description of an algorithm for computing the greatest common divisor (GCD) of 2 positive integers.
 - Repeatedly subtract the smaller one of the 2 numbers from the larger one until the resulting 2 numbers are equal.
 - The equal numbers are the greatest common divisor of the original 2 numbers.

Example

- For example, we would do the following to compute the greatest common divisor of 25 and 15 using this method.

```
num1  num2
```

```
25    15
```

```
10    15    <<<<subtract 15 from 25
```

```
10     5    <<<<subtract 10 from 15
```

```
 5     5    <<<<subtract 5 from 10
```

```
GCD is 5
```

Pseudo Code

- To make this into a Python program, we would need to prompt for and input the 2 numbers and then have a loop to do this process repeatedly.
- We should exit the loop when the 2 numbers are equal.
 - That means we should stay in the loop when the 2 numbers are not equal.
- Each time around the loop, we must determine which number is larger and subtract the smaller number.

Code

```
num1 = int(input("Number 1: "))
num2 = int(input("Number 2: "))

while num1 != num2:
    if num1 < num2:
        num2 = num2 - num1
    else:
        num1 = num1 - num2

print ("GCD is", num1)
```


4. Nested Loops

- Loops can be nested (to any number of levels).
 - For-loop inside for-loop
 - For-loop inside while-loop
 - While-loop inside for-loop
 - While-loop inside while-loop

Example

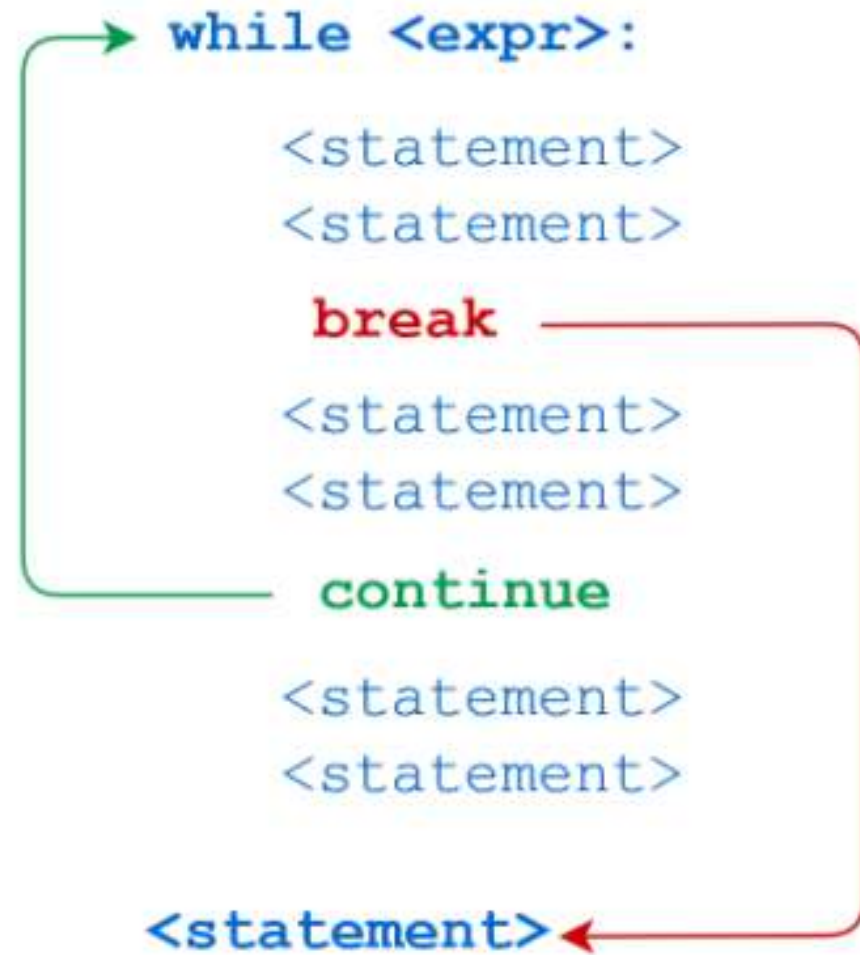
```
SIZE = 5
for numStars in range(1, SIZE+1):
    for i in range(numStars):
        print('*', sep = ' ', end='')
    print()
```

*

**

5. Additional Clauses

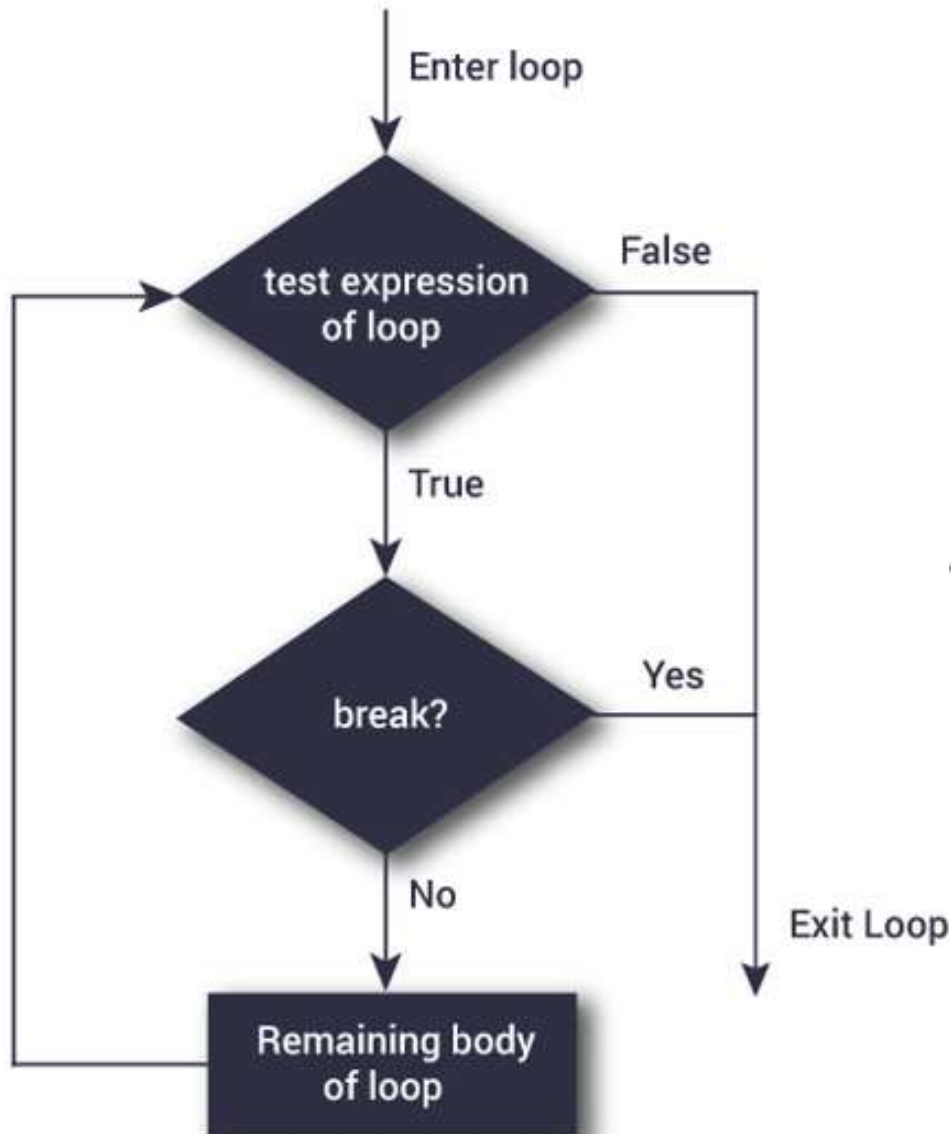
- Break
- Continue
- Else



Two ways to exit

- There are two ways to exit the loop:
 - **Normal** exit: when we reach the end of the iterable, or the while condition becomes False.
 - **Abnormal** exit: when we decided to exit in the middle of the loop's body. Break.

Break



```
for var in sequence:  
    # codes inside for loop  
    if condition:  
        break  
    # codes inside for loop  
# codes outside for loop
```

```
while test expression:  
    # codes inside while loop  
    if condition:  
        break  
    # codes inside while loop  
# codes outside while loop
```

Break Clause

- Breaks out of the innermost enclosing for or while loop.

```
for i in range(10):  
    for j in range(10):  
        print("(" + i + ", " + j + ") ", end="")  
        if i == j:  
            break  
    print()  
print()
```

Break Example

(0, 0)
(1, 0) (1, 1)
(2, 0) (2, 1) (2, 2)
(3, 0) (3, 1) (3, 2) (3, 3)
(4, 0) (4, 1) (4, 2) (4, 3) (4, 4)
(5, 0) (5, 1) (5, 2) (5, 3) (5, 4) (5, 5)
(6, 0) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (6, 6)
(7, 0) (7, 1) (7, 2) (7, 3) (7, 4) (7, 5) (7, 6) (7, 7)
(8, 0) (8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (8, 6) (8, 7) (8, 8)
(9, 0) (9, 1) (9, 2) (9, 3) (9, 4) (9, 5) (9, 6) (9, 7) (9, 8) (9, 9)

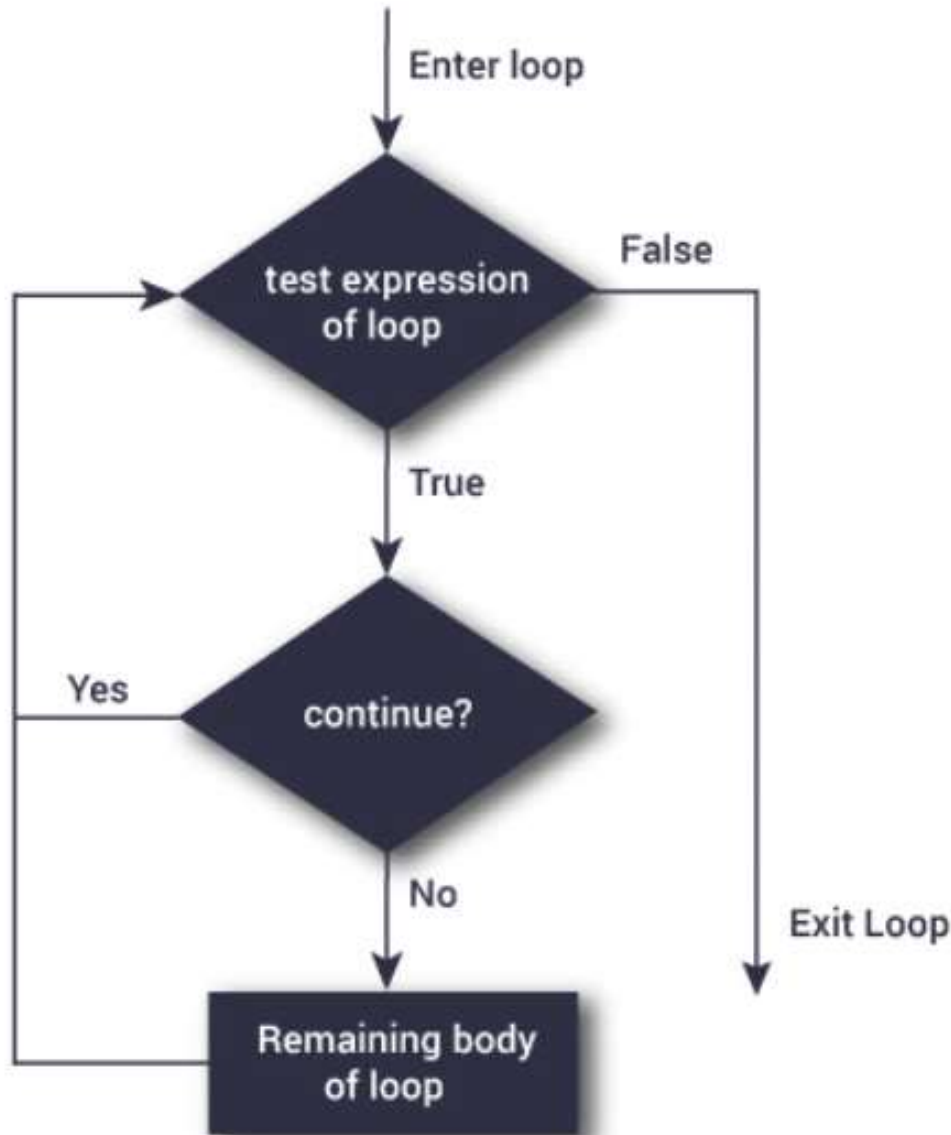
Break & Else

- Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when a break statement ends the loop.
- Normal exit vs. abnormal exit.
 - Normal exit => execute **else** clause
 - **Ab**normal exit => **don't** execute else clause

Example

```
for i in range(10):  
    for j in range(10):  
        print("(" + i + ", " + j + ") ", end="")  
    else: # do not indent further  
        print()  
else: # do not indent further  
    print()
```

Continue



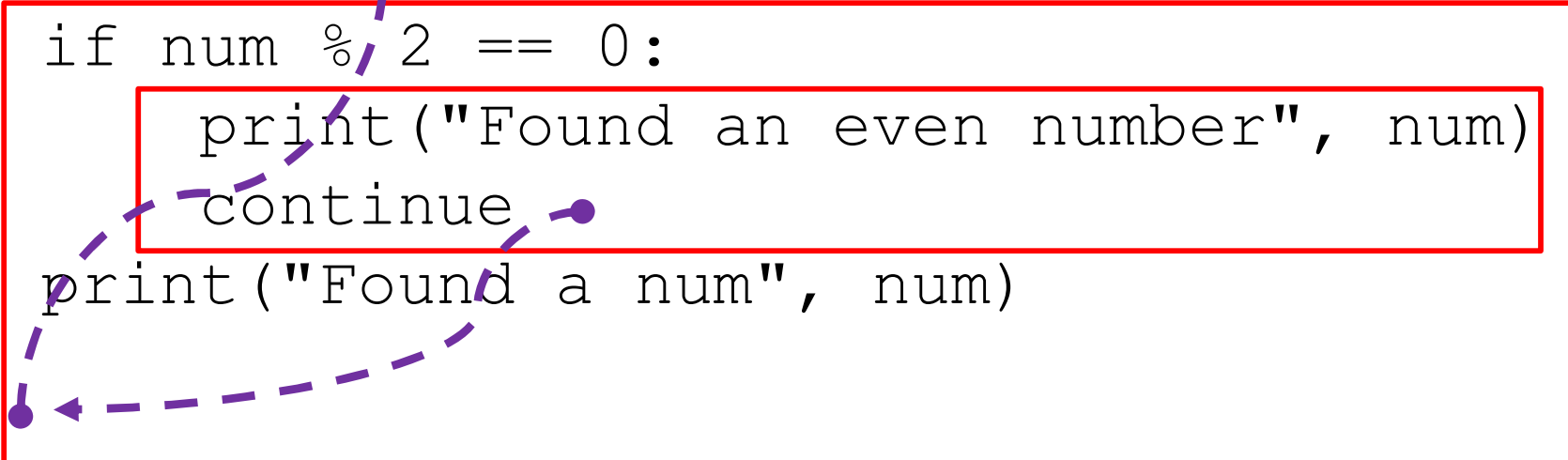
```
for var in sequence:  
    # codes inside for loop  
    if condition:  
        continue  
    # codes inside for loop  
  
# codes outside for loop
```

```
while test expression:  
    # codes inside while loop  
    if condition:  
        continue  
    # codes inside while loop  
  
# codes outside while loop
```

Continue

- The continue statement, borrowed from C, continues with the next iteration of the loop. "Continue to the next iteration."

```
for num in range(2, 10):  
    if num % 2 == 0:  
        print("Found an even number", num)  
        continue  
    print("Found a num", num)
```



Break and Continue

- Flow of Control
 - Recall how loops provide a "graceful" and clear flow of control in and out
 - In rare instances, it can alter the natural flow
- `break`;
 - Forces loop to exit immediately.
- `continue`;
 - Skips rest of loop body
- These statements violate the natural flow
 - Only used when necessary.

Why?

- Break and continue statements can alter the flow of a loop.
- Loops iterate over a code block until the test expression is false (normal exit). Still, sometimes we wish to terminate the current iteration or even the whole loop without checking the test expression (abnormal exit).
- The break and continue statements are used in these abnormal cases.

Else matters

```
i = 5
while (i <= 10) :
    print (i)
    i = i + 1

#

#
else:
    print ("Normal Exit")
print ("The next statement.")
```

Else matters

```
i = 5
while (i <= 10) :
    print (i)
    i = i + 1
    if i== 7:
        break # not the end
else:
    print ("Normal Exit")
print ("The next statement.")
```

Enumerate()

- It is convenient to use an iterator to get the elements in a for-loop.
- If it is necessary to have an index, we can use Range().
- Is there a way to combine the features (iterator and index)?
- Enumerate() is the answer.

Example

```
fruits = ["apple", "banana", "cherry",  
          "pear", "grape", "watermelon"]
```

```
for idx, fruit in enumerate(fruits):  
    print(idx, fruit)
```

```
for num, fruit in enumerate(fruits, start=1):  
    print(num, fruit)
```

```
for i in range(len(fruits)):  
    print(i, fruits[i])
```