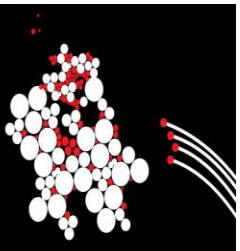
UNIVERSITY OF TWENTE.



Iterations

How to successively visit and operate on elements in a container

1

Iterations

So far, we have implicitly assumed that a task is done just once …

In real life, repeated tasks are everywhere.

For instance:

- Reading all lines of a CSV file and operating on them
- Intersecting a collection of polygons
- Calculating the buffer area around a set of points
- Determining a regression line using the least squares method
- Connecting to a database to retrieve data records from a table
- Checking whether polygons in a shapefile are geometrically valid
- Retrieving data from a citizen science portal
- Operating on a raster data set with a 3x3 kernel window

ITC UNIVERSITY OF TWENTE.

2

Iterations

The repetition of a task over multiple elements is called iteration

Iteration is performed in a statements that we call a loop

In Python, you can use two types of loops:
- The for loop
- The while loop

Normally, the task at hand will determine which of the two is the more appropriate.

UNIVERSITY OF TWENTE.

3

The while loop

A while loop uses the following syntax:

while condition :
body_of_statements

The body of statements (indented, obviously) will be repeated as long as the condition continues to evaluate to true. These statements code for the "*task-per-element*."

While loops are used for cases when one cannot predict how often the task-per-element will need to be executed.

Good code ensures that eventually the condition becomes false, when there is nothing to be done anymore, for instance because there are no more elements to be handled.

UNIVERSITY OF TWENTE.

4

The while loop

A simple while loop that prints the numbers starting at 1, until we find a non-prime number.

Its structure is typical of while loops:

```
i = 0                                   % initialization
while i < 100 :                         % (non-)termination condition
        if is_prime(i) is False:        % statements (ie, task for the
                i = i + 1               % element)
        else:
                print(i)
                i = i + 1               % progress step
```

Variable *i* controls the progress through the loop, it is called the loop variable

UNIVERSITY OF TWENTE.

6

While loop order of progress

One obviously can also count down:

```
i = 10
while i > 0:
        print(i)
        i = i - 1
```

We can wrap this in a function

```
def countdown(n)
        i = 10
        while i > 0:
                print(i)
                i = i – 1
        print("BOOM!")

countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
BOOM!
```

UNIVERSITY OF TWENTE.

7

Skipping an element inside the while loop

We can skip an iteration step with the continue statement

```
i = 0
while i < 9:
   i = i + 1
   if i==5 or i==7:
      continue                                  # Skip and go on
   else:
      print(i, "*", end="")
print(i+1)
```

*1*2*3*4*6*8*9*

Normally, *print()* puts a newline character behind what it is printing. With the *end="…"* clause used, it ends the printed string with whatever is in … position instead.

UNIVERSITY OF TWENTE.

9

While iteration over a list

We can iterate over lists, which is a common programming practice:

```
fruits = ["orange", "watermelon", "lemon", "coconut"]
i = 0
while i < len(fruits):
   print(fruits[i], end=" ")
   i = i + 1
```

orange watermelon lemon coconut

Observe the subtle use of *end=" ".*

Also observe that in this case we know upfront how often we will iterate (namely, *len(fruits)* times). In such cases, a *for* loop is usually the better coding choice. (See later.)

UNIVERSITY OF TWENTE.

10

Populating a container with while

We can populate lists

```
twenty_even_numbers = [ ]
i = 0
while i < 40:
        twenty_even_numbers.append(i)
        i = i + 2
```

While does the job again, but a choice to use a *for* loop is recommended.

UNIVERSITY OF TWENTE.

The for loop

The ***for*** loop has the following syntax:

```
for variable in sequence :
        body_of_statements
```

The variable value iterates over the complete sequence *(which can be a string, a list, a tuple, or any other type of container)* and assumes the value of its successive elements.

Good code

- makes use of the *variable* in the *body_of_statements*.
- uses the *for* loop in cases when upfront we know how often the *body_of_statements* will be executed for an element.

This is a common coding mechanism and you can use ***continue*** also in for loops.

UNIVERSITY OF TWENTE.

The for loop

An example with strings:

```
for char in "Hello world!":
    print(char)
```

H
e
l
l
o

w
o
r
l
d
!

Another example with strings:

```
my_string = "Can you reverse this?"
result = ""
for char in my_string:
    result = char + result
print(result)
```

?siht esrever uoy naC

Observe that here only one print statement is executed.

UNIVERSITY OF TWENTE.

Steering the for loop

We can finetune the scope of action with ***range()***:

for i in range(…) :

It is used as: *range([start], stop, [step])* All arguments are integer.
•*start* is optional and defaults to 0; *step* is also optional (defaults to 1)

The *range()* function gives us an iterator object, this is something magical over which you can iterate. With *list(range(…))* you will see the list of elements that the iterator gives you. *start* is a member of the iterator but *stop* is (just) not: *list(range(1,3)) == [1, 2]*

```
print(list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(list(range(2, 8)))
[2, 3, 4, 5, 6, 7]

print(list(range(0, 12, 2)))
[0, 2, 4, 6, 8, 10]
```

UNIVERSITY OF TWENTE.

Iteration over *n*-dimensional data structures

How to iterate over, for instance, a two-dimensional data structure?

```
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
rows = 3
cols = 3
the_sum = 0

for i in range(rows):
    for j in range(cols):
        the_sum += m[i][j]
        print(m[i][j])

print("The sum is: ", the_sum)
```

```
1
2
3
4
5
6
7
8
9
The sum is:  45
```

Nested iteration. Within iteration over rows, we iterate over columns.

Observe the use of +=, which is expressed as "*plus and becomes*."

Summary

- We use iterations for repetitive tasks
- In Python, iterations are done with for and while loops
- While loop is used typically when one cannot quite foresee when the process will end.
- For loop is used typically when the number of steps is known upfront, for instance to access elements of a sequence
- Nested loops are useful to iterate over *N*-dimensional data structures, for instance where lists are nested inside a list.