Iteration: statement `while`

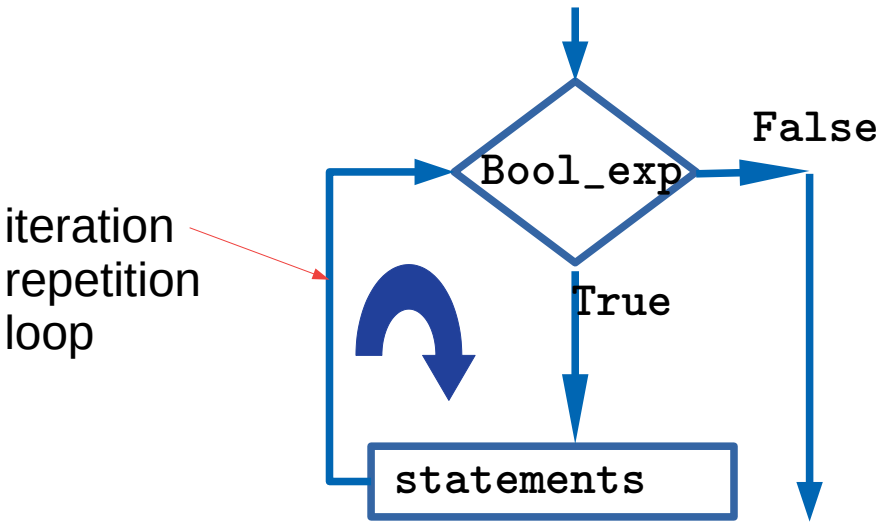
Iteration: statement `while`

Statement `for`:
Iteration in a controlled and secure way: traverses the structure items from the first to the last

```
for item in structure:
    process item
```

Statement `while`: (general iteration)
Allows to design all kinds of iteration

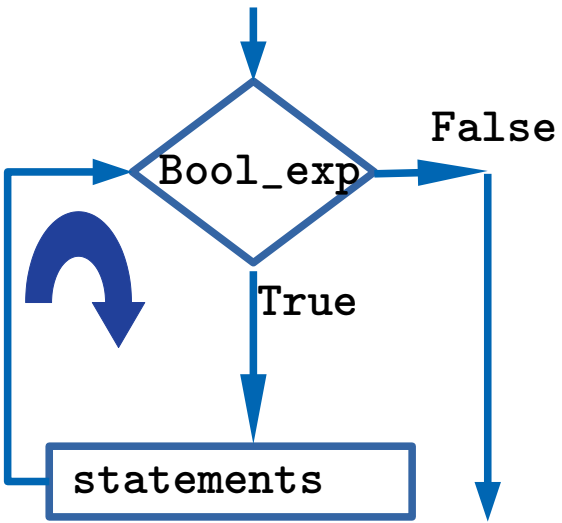
```
while Bool_exp:
    statements
```



Iteration: statement `while`

Statement `while`: (general iteration)
Allows to design all kinds of iteration

```
while Bool_exp:
    statements
```



Prone to errors:

- do not process exactly all items
- Bool_exp evaluates always to True:

endless loop

```
a = 5
while a < 10:
    a = a -1
```

Sequence based design

A **sequence** is a data set of any type arranged linearly

Sequence identification: to determine its data type

Sequence characterization:

- to determine its **first** element
- to determine its **next** element
- to determine its **ending property**: property met by all the sequence elements

Scheme identification: traversal or search

One variable will be assigned to all sequence items sequentially

Traversal scheme

Design of a `while` statement ...

```
statements before the while statement
while Bool_exp:
    statements in the body
```

... with the sequence based technique

```
get 1st element
while not ending_property:
    process element
    get next element
```

Exercise 1

Write function `sum_mult` that takes an integer, n, and returns the sum of all multiples of 3, positive and less than n. Save this function in file `sum_mult.py`. You can download this file with tests `test-sum_mult.txt`. Examples:

```
>>> sum_mult(10)
18
>>> sum_mult(30)
135
>>> sum_mult(100)
1683
```

Identification: sequence of multiples of 3 (integers)
characterization:

- first element: 3
- next: adding 3 to the previous element
- ending property: all items are less than n

Exercise 1

Write function `sum_mult` that takes an integer, n, and returns the sum of all multiples of 3, positive and less than n. Example:

```
>>> sum_mult(10)
18
>>> sum_mult(30)
135
>>> sum_mult(100)
1683
```

Identification: sequence of multiples of 3 (integers)
characterization:

- first element: 3
- next: adding 3 to the previous element
- ending property: all items are less than n

```
def sum_mult (n):
    summ = 0
    mult = 3
    while mult < n:
        summ = summ + mult
        mult = mult + 3
    return summ
```

Exercise 1: version 2

The characterized sequence is an arithmetic progression that can be represented with function `range`: `range(3, n, 3)`

Then, statement for can also be applied

```
def sum_mult (n):
    summ = 0
    mult = 3
    while mult < n:
        summ = summ + mult
        mult = mult + 3
    return summ
```

```
def sum_mult (n):
    summ = 0
    for mult in range(3, n, 3):
        summ = summ + mult
    return summ
```

Exercise 2

Write function **`multiples`** that takes an integer, n, and returns the minimum number of multiples of 3, positive and such that its sum is a value greater than or equal to n. Save this function in file **`multiples.py`**. You can download this file with tests **`test-multiples.txt`**.

Examples:

```
>>> multiples(9)
2
>>> multiples(10)
3
>>> multiples(18)
3
```

Identification: sequence of multiples of 3

Characterization:
first element: 3
next: adding 3 to the previous element
ending property: sum of elements greater than or equal to n

Process: counting and adding the sequence elements

Exercise 2

Write function `multiples` that takes an integer, n, and returns the minimum number of multiples of 3, positive and such that its sum is a value greater than or equal to n.

Examples

```
>>> multiples(9)
2
>>> multiples(10)
3
>>> multiples(18)
3
```

```
def multiples(n):
    q = 0
    summ = 0
    m3 = 3
    while summ < n:
        q = q + 1
        summ = summ + m3
        m3 = m3 + 3
    return q
```

Identification: sequence of multiples of 3

Characterization:

- first element: 3
- next: adding 3 to the previous element
- ending property: sum of elements greater than or equal to n

Process: counting and adding the sequence elements

- `for` statement is NOT applicable
- sequence cannot be represented with function `range`

Exercise 3

Write function `pow_two` that takes an integer, x, positive and less than 10, that represents a digit, and another integer, n, and returns the number of powers of 2 which are less than n and that contain this digit. Save this function in file `pow_two.py`. You can download this file with tests `test-pow_two.txt`.

Examples:

```
>>> pow_two(2, 1000)
5
>>> pow_two(3, 1000)
1
>>> pow_two(1, 1000)
4
```

Identification: sequence of powers of 2 (integers)
Characterization:
first element: 1
next element: multiplying by 2 the previous element
ending property: elements are less than n

Process: filter application (contains digit) and counting.

Exercise 3

Write function `pow_two` that takes an integer, x, positive and less than 10, that represents a digit, and another integer, n, and returns the number of powers of 2 which are less than n and that contain this digit. Example:

```
>>> pow_two(2, 1000)
5
>>> pow_two(3, 1000)
1
>>> pow_two(1, 1000)
4
```

Identification: sequence of powers of 2 (integers)
Characterization:
first element: 1
next: multiplying by 2 the previous element
ending property: elements are less than n

Process: filter application (contains digit) and counting.

```
def pow_two (x, n):
    c = 0
    pow2 = 1
    while pow2 < n:
        if str(x) in str(pow2):
            c = c + 1
        pow2 = pow2 * 2
    return c
```

Exercise 4

We want to sample a mathematical function y= f(x)= x^2+4x+2 in the interval [a, b] and with d abscissa increments. Write function **`posneg`** that takes the mentioned values a, b, d (integer or float), and returns two integers corresponding respectively to the number of points with positive or zero ordinate value and the number of points with negative ordinate value. Save this function in file **`posneg.py`**. You can download this file with tests **`test-posneg.txt`**.

Examples:

```
>>> posneg(-5, 5, 1)
(8, 3)
>>> posneg(-5, 5, 2)
(4, 2)
>>> posneg(-4, 4, 2)
(4, 1)
```

Identification: sequence of abscissas
Characterization:
first element: a
next: adding d to previous abscissa
ending property: abscissa > b

Process: ordinate computation, filter application (positive or negative) and computing 2 counters

Exercise 4

Two counters initialization

Identification: sequence of abscissas
Characterization:
first element: a
next: adding d to previous abscissa
ending property: abscissa > b

Process: ordinate computation, filter application (positive or negative) and computing 2 counters

```
def posneg (a, b, d):
    npos = 0
    nneg = 0
    x = a
    while x <= b:
        y = x**2 + 4*x + 2
        if y >= 0:
            npos = npos + 1
        else:
            nneg = nneg + 1
        x = x + d
    return npos, nneg
```

Searching scheme

```
found = False
get 1st element
while not ending_property:
     found = search_condition(element)
     if found:
          break
     get next element
if found:
    statements block for case found
else
    statements block for case not found
```

Exercise 5

We want to sample a mathematical function y= f(x)= x^2+4x+2 in the interval [a, b] and with d abscissa increments. Write function `root` that takes the mentioned values a, b, d (integer or float), and an epsilon tolerance, eps (float). This function returns the first root of the given mathematical function in the given interval. An abscissa will be considered a root if its corresponding ordinate is less than the given epsilon tolerance. If no roots are found, the function must return the string 'no root'. Save this function in file `root.py`. You can download this file with tests `test-root.txt`. Examples:

```
>>> round(root(-4, 4, 0.001, 1E-3), 2)
-3.41
>>> round(root(-1, 4, 0.001, 1E-3), 2)
-0.59
>>> root(-0, 4, 0.01, 1E-3)
'no root'
>>> root(-4, 4, 0.01, 1E-2)
'no root'
>>> round(root(-4, 4, 0.01, 0.05), 2)
-3.43
```

Identification: sequence of abscissas
Characterization:
first element: a
next: adding d to previous abscissa
ending property: abscissa > b

searching property: abs(y) < epsilon

Exercise 5

Identification: sequence of abscissas
Characterization:
first element: a
next: adding d to previous abscissa
ending property: abscissa > b

searching property: abs(y) < eps

```
def root (a, b, d, eps):
    x = a
    while x <= b:
        y = x**2 + 4*x + 2
        if abs(y) < eps:
            return x
        x = x + d
    return 'no root'
```