

## CSI31 Lecture 19

**Topics:** *Chapter 8. Loop Structures and Booleans*

8.3 (Continues) Common loop patterns: *for*, nested.

8.4. Computing with booleans.

8.5 Other common structures: *do-while*, *while* and *do-while*.

! No class on Wednesday, November 19<sup>th</sup>  
cancelled

## 8.3 (Continues) Common loop patterns: file, nested

### Work with files

Let's write a program that finds the average of all numbers in a file. We assume that the numbers are typed into a file one per line.

Here is the **algorithm**:

```
print an explanatory notice
take the file name from the user (where the numbers are stored)
open file
counter = 0
for loop over lines in the file
    covert the line into a float
    add to the sum
    increment the counter
print the average
```

See [average\\_file.py](#),

**Modification:**

if files contains characters other than numbers, then they will be ignored.

See [average\\_file\\_exptns.py](#)

## Nested loops

Loops may contain loops. Let's take a look at the program that reads in all the numbers from a file, no matter how the numbers are typed into a file, i.e. we may have something like this (more than one number per line, separated by space):

```
1 2 34 12 0 -12 23
```

```
3 5 456 23 09 8
```

```
1 4 -23 45 -89
```

Or like this (one number per line):

```
1
```

```
2
```

```
34
```

```
12
```

```
0
```

```
-12
```

```
23
```

### Algorithm:

```
take a file name from the user
```

```
open the file
```

```
numbers = [0] # list of all numbers from the file
```

```
for loop over the lines (L) in the file
```

```
    numbers_in_line=split L using space as a delimiter
```

```
    # numbers_in_line is the list of numbers now,
```

```
    # the last number has \n attached to it
```

```
    n = number of elements in the list numbers_in_line
```

```
    for i in range(n)
```

```
        convert the ith element of the list to a number
```

```
        append to the list of all numbers (numbers)
```

See [read\\_all\\_numbers.py](#)

## 8.4 Computing with booleans

Now we have two control structures, `if` and `while`, that use conditions, which are boolean expressions.

Boolean expression is either `True` (1) or `False` (0).

So far we used expressions that compare two values (`>=`, `<=`, `!=`, `==`, `>`, `<`).

And in the last program (`read_all_numbers.py`) you saw `... and ...`.

Let's take a look at `Boolean operators`. They are used to combine Boolean expressions to get a new Boolean expression.

`AND` (`&`)

`OR` (`&`)

`NOT` (`~`, `¬`)

`<expr1> and <expr2>`

`<expr1> or <expr2>`

`not <expr>`

`and` and `or` are `binary operators`, `not` is a `unary operator`.

The `and` of two expressions is `true` when both expressions are true.

The `or` of two expressions is `true` when at least one of the expressions is true.

The `not` operator computes the `opposite` of a Boolean expression.

P	Q	P and Q
T	T	T
T	F	F
F	T	F
F	F	F

P	Q	P or Q
T	T	T
T	F	T
F	T	T
F	F	F

P	not P
T	F
F	T

**Precedence rules (from high to low):** `not`, `and`, `or`

**Example:** `a and b or not a and b` is equivalent to `(a and b) or ((not a) and b)`

## Some of the properties of Boolean operations:

we can note the following correspondence:

true  $\approx$  1, false  $\approx$  0

and  $\approx$   $\times$ , or  $\approx$   $+$

Then:

$a$  or true == true (i.e.  $a + 1 \neq 1$  not working here)

$a$  and false == false (i.e.  $a \times 0 = 0$ )

$a$  and true ==  $a$  (i.e.  $a \times 1 = a$ )

Distributive rules:

$a$  or ( $b$  and  $c$ ) == ( $a$  or  $b$ ) and ( $a$  or  $c$ )

$a$  and ( $b$  or  $c$ ) == ( $a$  and  $b$ ) or ( $a$  and  $c$ )

A double negation rule: not (not  $a$ ) ==  $a$

DeMorgan's laws:

not( $a$  or  $b$ ) = (not  $a$ ) and (not  $b$ )

not( $a$  and  $b$ ) = (not  $a$ ) or (not  $b$ )

Boolean algebra (Boolean logic) is the algebra of truth values and operations on them.

It was developed by George Boole in the late 1830s.

One application of Boolean algebra is the analysis and simplification of Boolean expressions.

Let's write a program that takes a temperature value as an input, and output whehere it is hot, warm, cold or freezing today.

Assume that if it is above 90F then it is hot;  
if it is between 70F and 90F, then it is warm;  
it it is between 32F and 70F, then it is cold;  
and if it is bellow 32F then it is freezing.

Here is the **algorithm**:

take the temperature as an input (T)

if  $T \geq 90$

    output HOT

if  $T < 90$  and  $T \geq 70$

    output WARM

if  $T < 70$  and  $T \geq 32$

    output COLD

if  $T < 32$

    output FREEZING

See [temperature.py](#)

What if we want to allow user to enter temperatures as many times as he/she wants?

What would we do in this case?

See [temperature\\_infiniteLoop.py](#)

## 8.5 Other common structures: post-test, loop and half

the decision structure (**if**) along with the infinite (pre-test) loop (**while loop**) provide a complete set of control structures. Every algorithm can be expressed using just these.

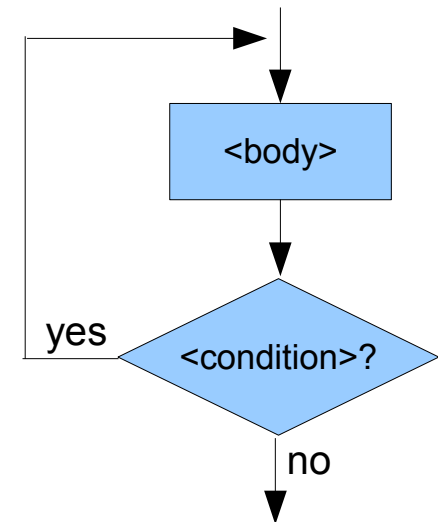
Sometimes, for certain kinds of problems, alternative structures are more convenient (**for loop** and **post-test loop**)

### Post-test loop

Syntax could be:

```
repeat  
    <body>  
until <condition>
```

- the condition test comes after the loop body  
(the body of the loop is always executed at least once)



Flowchart of a post-test loop

**! Python doesn't have a statement that directly implements a post-test loop.**

We can simulate the post-test loop:

initialize the variable(s) that are used in the while's condition to such value(s) that the while loop is entered.

**OR**

use `while True` and `break` statement

`while True` is an infinite loop, condition is always true  
`break` statement terminates a loop

For the first method see programs from lecture 18:

```
average_i.py    answer = "yes"  
                while answer[0]=='y':
```

```
average_s.py    next_value=0  
                while next_value != -1000:
```

```
average_s_mod.py ns='0'  
                 while ns != "":
```

For the second method see [temperature\\_infiniteLoop.py](#)

Try to **avoid** peppering the body of the loop with **multiple break statements**, because the logic of the loop might be lost.

However, there are times when this rule should be broken to provide most elegant solution.