Scientific Programming

Lecture A03 – Structured programming

Andrea Passerini

Università degli Studi di Trento

2019/09/22

Acknowledgments: Alberto Montresor

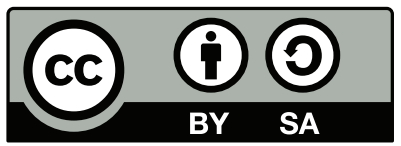
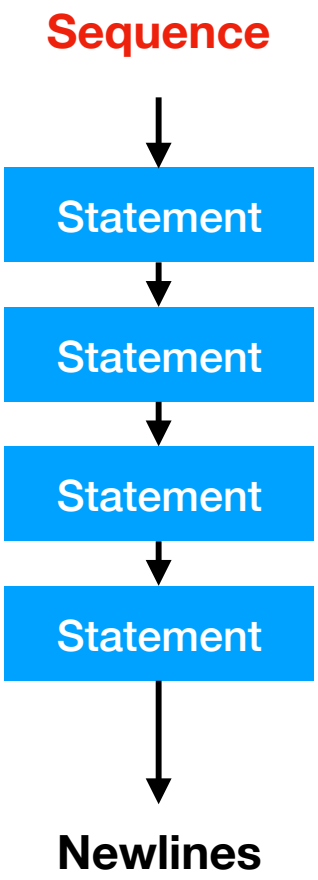
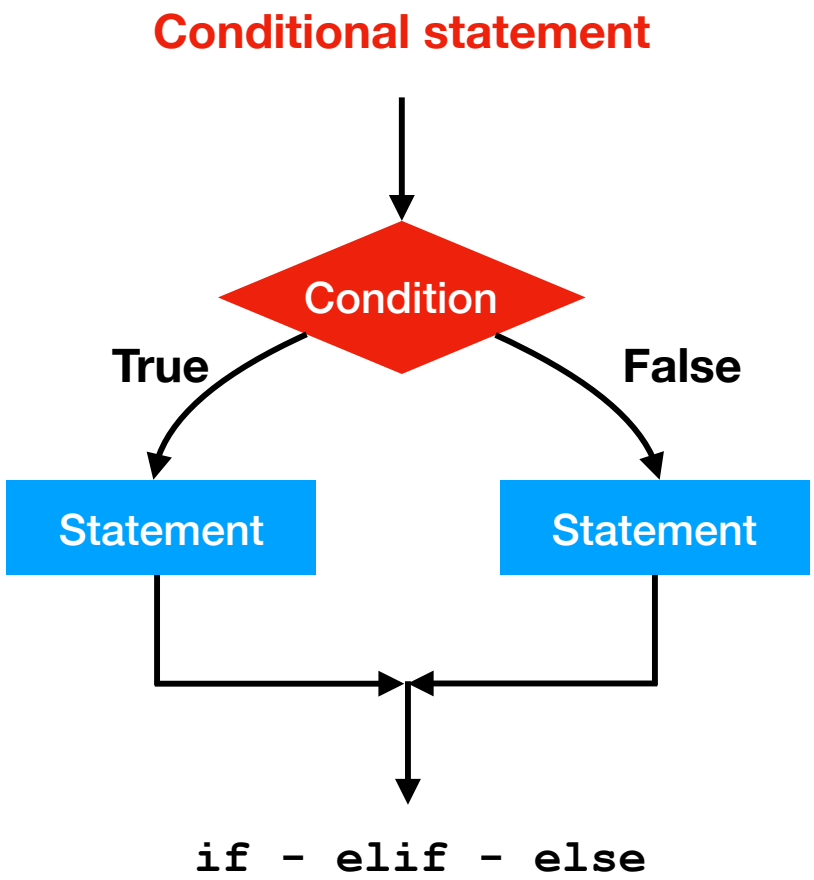


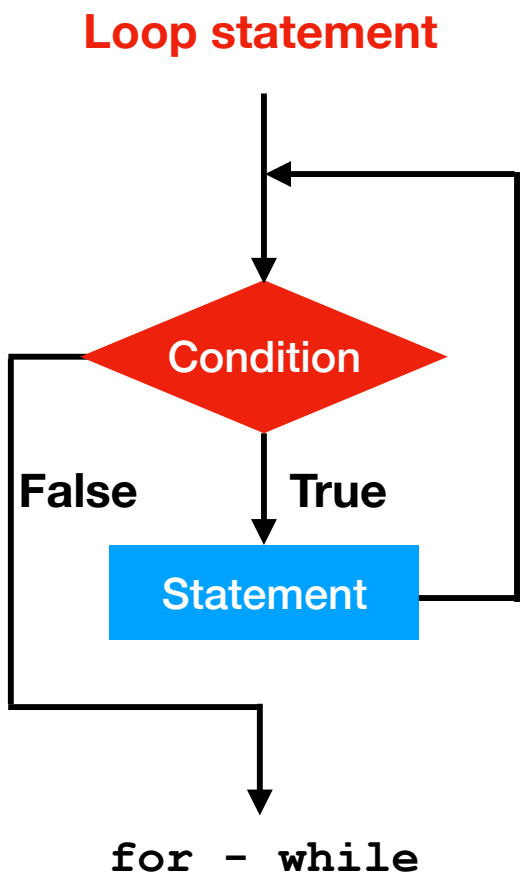
Table of contents

Structured programming

After having surveyed the basic data items, we now survey the basic control structures:





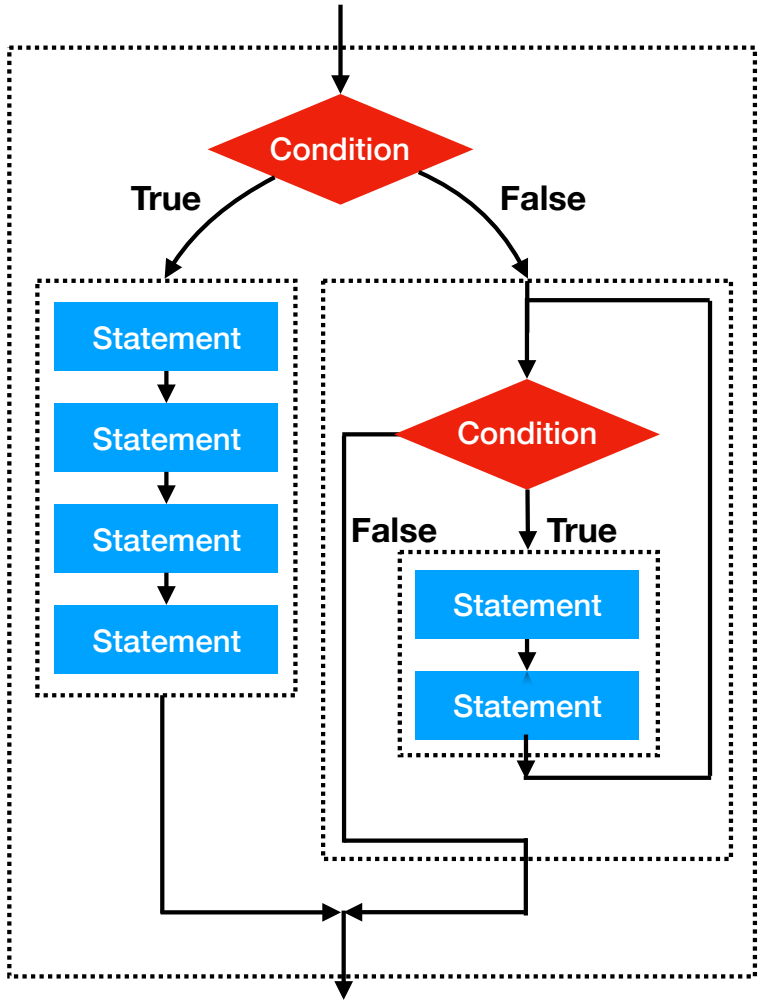


Structured programming

Blocks

Blocks are used to enable groups of statements to be treated as if they were one statement.

Block-structured languages have a syntax for enclosing blocks in some formal way.



Some simple examples

for loop

```
L = [1,2,3,4]
print("---------------")
for val in L:
    print(">> ", end="")
    print(val, val*val)
print("---------------")
```

Some simple examples

for loop

```
L = [1,2,3,4]                    ---------------
print("---------------")         >> 1 1
for val in L:                    >> 2 4
    print(">> ", end="")         >> 3 9
    print(val, val*val)          >> 4 16
print("---------------")         ---------------
```

if - else statement

```
answer = input()
if (answer.lower() == "yes"):
    print("Good!")
else:
    print("Too bad!")
```

Python vs rest-of-the-world: Indentation

Python

Blocks are identified by indentation

```
L = [1,2,3,4]
print("----")
for val in L
    print("** ", end="")
    print(val, val*val)
print("----")
```

Java

Blocks are identified by curly braces

```
System.out.println("--------")
for (int val=1; val <= 4; i++) {
  System.out.print("** ");
  System.out.println(val + " "
      + val*val)
}
System.out.println("--------")
```

Rules

- End-of-line is end of statement
- End-of-indentation is end of block

Common errors

```
L = [1,2,3,4]
print("---------------")
for val in L:
print("** ", end="")
print(val, val**2)
print("---------------")
```

```
[andrea@praha ~]$ python errors.py
  File "errors.py", line 4
    print("** ", end="")
        ^
IndentationError:  expected an indented block
```

Common errors

```
L = [1,2,3,4]
print("---------------")
for val in L:
      print("** ", end="")
    print(val, val**2)
print("---------------")
```

```
[andrea@praha ~]$ python errors.py
  File "errors.py", line 5
    print(val, val**2)
                     ^
IndentationError:  unindent does not match any
outer indentation level
```

Common errors

```
L = [1,2,3,4]
print("---------------")
for val in L:
    print("** ", end="")
      print(val, val**2)
print("---------------")
```

```
[andrea@praha ~]$ python errors.py
  File "errors.py", line 5
    print(val, val**2)
    ^
IndentationError:  unexpected indent
```

Other formatting quirks

Content of parenthesis `()`, `[]`, `{ }` can be split among multiple lines. Without parenthesis, you get an error.

```
L = [ "A Game of Thrones", "A Clash of Kings",
      "A Storm of Swords", "A Feast for Crows",
      "A Dance with Dragons"
]
S1 = ("abcefghijklmnopqrstuvwxyz" +
       "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
S2 = "abcefghijklmnopqrstuvwxyz" +
       "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

[andrea@praha ~]$ python prova.py
  File "prova.py", line 12
    S2 = "abcefghijklmnopqrstuvwxyz" +
SyntaxError:  invalid syntax
```

Other formatting quirks

In-line sequences

Multiple statements are normally written one per line. Is it possible to use `;` to separate multiple statements in a single line.

```
a = 1 ; b = 2 ; c = a + b
```

In-line structured commands

It is possible to avoid indentation in structured statements, by writing a single statement on the same line

```
if val > maximum: maximum = val; pos = i
```

Both approaches are considered non-pythonic and advised against

`if` – `else` statements

Conditional statement

A conditional statement allows to write code that gets executed if and only if some condition is satisfied.

```
print("Guess what number I'm thinking: ")
guess = input()
if guess == 5633839494:
    print("It's correct!")
else:
    print("Sorry, wrong number")
```

BTW, do you see a problem in this code?

`if` – `else` statements

Conditional statement

The conditional statements allow to write code that gets executed if and only if some condition is satisfied.

```
print("Guess what number I'm thinking: ")
guess = int(input())
if guess == 5633839494:
    print("It's correct!")
else:
    print("Sorry, wrong number")
```

The return of `input()` is a string, compared to a number returns `False`

if – elif – else statements

`if`, `elif` and `else` form a "chain": only one of the branches is executed.

```
print("Insert a number between 1 and 4")
value = int(input())
if value == 1:
    print("Very good choice")
elif value == 2:
    print("The first even number")
elif value == 3:
    print("The first Fermat number")
elif value == 4:
    print("A highly totient number")
else:
    print("I said, between 1 and 4")
```

`if` – `elif` – `else` statements

`if`, `elif` and `else` form a "chain": only one of the branches is executed. The first one that matches is executed.

What is the difference between these two pieces of code?

```
value = 2
if (value == 2):
  print("Even and prime")
elif (value % 2 == 0):
  print("Even")
```

```
Even and prime
```

```
value = 2
if (value == 2):
  print("Even and prime")
if (value % 2 == 0):
  print("Even")
```

```
Even and prime
Even
```

if – elif – else statements

What is printed by this piece of code?

```
if condition1:
    print("A")
elif condition2:
    print("B")
else:
    print("C")
```

	condition2==True	condition2==False
condition1==True	A	A
condition1==False	B	C

`if` – `elif` – `else` statements

How we can re-obtain the behavior described by the table, without using `elif` / `else`, but using `and`/`or`/`not` operators?

	`condition2==True`	`condition2==False`
`condition1==True`	A	A
`condition1==False`	B	C

```
if condition1:
    print("A")
if not condition1 and condition2:
    print("B")
if not condition1 and not condition2:
    print("C")
```

For Loop

For loop

The `for` loop is a generic iterator in Python: it can step through the items of any ordered sequence or other iterable object.

- Works on strings, lists, tuples, dictionaries, and other built-in iterables
- Is it possible to define new iterable objects

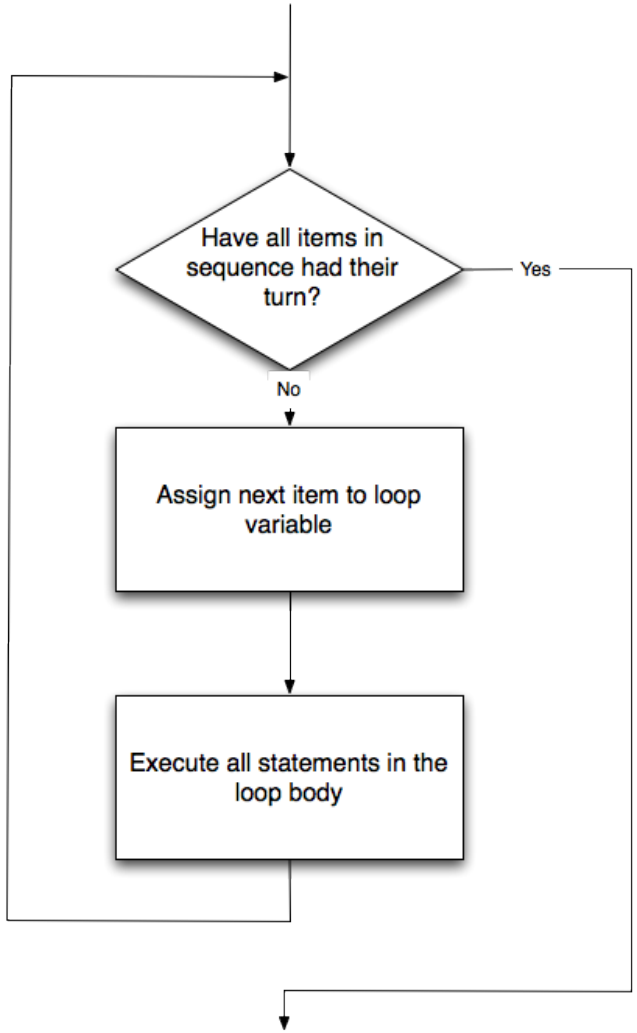
```
L = ["Venezia", "Verona", "Padova", "Vicenza",
     "Treviso", "Belluno", "Rovigo"]
for provincia in L:
    print(provincia, "si trova in Veneto")
```

For loop

```
for var in sequence:
    statement1
    statement2
    ...
```

- `var` is the loop variable; it is assigned to each of the values in the list, until all the values are used
- `sequence` is the loop sequence;
- the statements to be executed are called the loop body

Codelens



Iteration

`str`	`for` iterates over the characters
`list`	`for` iterates over the elements
`tuple`	`for` iterates over the elements
`dict`	`for` iterates over the keys

```
for k in [1,2,3,4]:
    print(k, end=' ')
print("")
for k in ('a','b','c'):
    print(k, end=' ')
print("")
for k in "IBM":
    print(k, end='.')
print("")
```

```
1 2 3 4
a b c
I.B.M.
```

Range

Range

The `range()` built-in function returns an iterable object that can be used to obtain a list of integers.

`range(stop)`

Returns a list of integers between `0` and `stop-1`

```
for k in range(4):
    print(k, end = ' ')
print("")
```

```
0 1 2 3
```

Range

Range

The `range()` built-in function returns an iterable object that can be used to obtain a list of integers.

`range(start, stop)`

Returns a list of integers between `start` and `stop-1`

```
for k in range(1,5):
    print(k, end = ' ')
print("")
```

```
1 2 3 4
```

Range

Range

The `range()` built-in function returns an iterable object that can be used to obtain a list of integers.

`range(start, stop, increment)`

Returns a list of integers between `start` and `stop-1`, with increment `increment`

```
for k in range(2,10,2):
    print(k, end = ' ')
print("")
```

```
2 4 6 8
```

Range - Differences between 2.x and 3.x

```
Python 2.7.13 (default, Apr 23 2017, 16:50:35)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on d
Type "help", "copyright", "credits" or "license" for more infor
>>> L = range(10)
>>> print(L)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
Python 3.5.3 |Anaconda 4.4.0 (x86_64)| (default, Mar  6 2017, 1
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwi
Type "help", "copyright", "credits" or "license" for more infor
>>> L = range(10)
>>> print(L)
range(0, 10)
```

Range - Differences between 2.x and 3.x

Python 2.x

`range()` returns a list that contains the desired integers. The list is stored in memory, making it inefficient to use `range()` for a very large number of iterations.

If you want to use the 3.x approach in 2.x, use `xrange()` instead.

Python 3.x

In Python 3.x, `range()` generates the numbers as the iterator `for` ask for them, without storing the list.

If you do want to create the list, use the `list()` built-in function:

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

Examples of `for` – Summation

The following code sums the numbers contained in the list `L`

```
L = [1, 25, 6, 27, 57, 12]
total = 0
for number in L:
    total = total + number
print("The sum is", total)
```

Alternative code (using built-in functions)

```
L = [1, 25, 6, 27, 57, 12]
print(sum(L))
```

Examples of `for` – Maximum

The following code returns the maximum value contained in `L`

```
L = [1, 25, 6, 27, 57, 12]
max_so_far = L[0]
for number in L:
    if number > max_so_far:
        max_so_far = number
print("The maximum is", max_so_far)
```

Alternative code (using built-in functions)

```
L = [1, 25, 6, 27, 57, 12]
print(max(L))
```

Examples of `for` – Fibonacci

The following code computes the first 20 Fibonacci numbers

$$F[n] = \begin{cases} F[n-1] + F[n-1] & n > 2 \\ 1 & n \leq 2 \end{cases}$$

```
F = [1,1]
for k in range(18):
    F.append(F[-1]+F[-2])
print(F)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
610, 987, 1597, 2584, 4181, 6765]
```

Nested `for` loops

Two or more loops may be "nested", meaning that one is contained in the other.
Consider the following example, that lists all possible "quarters of hour"

```
L = []
for h in range(24):
    for m in range(0, 60, 15):
        L.append(str(h)+":"+str(m))
print(L)
```

```
['0:0', '0:15', '0:30', '0:45', '1:0', '1:15', '1:30',
 '1:45','2:0', '2:15', '2:30', '2:45', '3:0', '3:15',
 '3:30', '3:45', ...
```

Nested `for` loops

The following code should check whether there are two repeated values inside a list

```
L = [1,3,5,6]
for x in L:
    for y in L:
        if x == y:
            print(x, "is repeated")
```

Do you see any problems?

```
1 is repeated
3 is repeated
5 is repeated
6 is repeated
```

Nested `for` loops

The following code checks whether there are two repeated values inside a list

```
L = [1,3,5,6,1,8,3]
for i in range(len(L)):
    for j in range(i+1,len(L)):
        if L[i] == L[j]:
            print(L[i], "is repeated at ", i, j)
```

```
1 is repeated at 0 4
3 is repeated at 2 6
```

Nested `for` loops

The following code checks whether there are two repeated values inside a list

```
L = [1,3,5,6,1,8,3,4,3]
for i in range(len(L)):
    for j in range(i+1,len(L)):
        if L[i] == L[j]:
            print(L[i], "is repeated at ", i, j)
```

What is printed?

```
1 is repeated at 0 4
3 is repeated at 2 6
3 is repeated at 2 8
3 is repeated at 6 8
```

Exercise

Given a list `L` of integers, print `True` if `L` contains two distinct values whose sum is equal to 17

Example: If `L=[3,7,12,10,8,32,7,5]`, print `True` because `7+10=17`.

```
found = False
for i in range(len(L)):
    for j in range(i+1,len(L)):
        if L[i]+L[j] == 17:
            found = True
print(found)
```

Exercise

Given a list `L` of integers, print the indexes of a couple of values whose sum is equal to 17

Example: If `L=[3,7,12,10,8,32,7,5]`, print either `2,7` or `1,3` because `L[2]+L[7]=17` and `L[1]+L[3]=17`.

```
found = False
for i in range(len(L)):
    for j in range(i+1,len(L)):
        if L[i]+L[j] == 17:
            found = True
            pos_i = i
            pos_j = j
if found:
    print(pos_i, pos_j)
```

Exercise

Given a list `L` of integers, print the indexes of the first couple of values whose sum is equal to 17

Example: If `L=[3,7,12,10,8,32,7,5]`, print `1,3` because `L[1]+L[3]=17`.

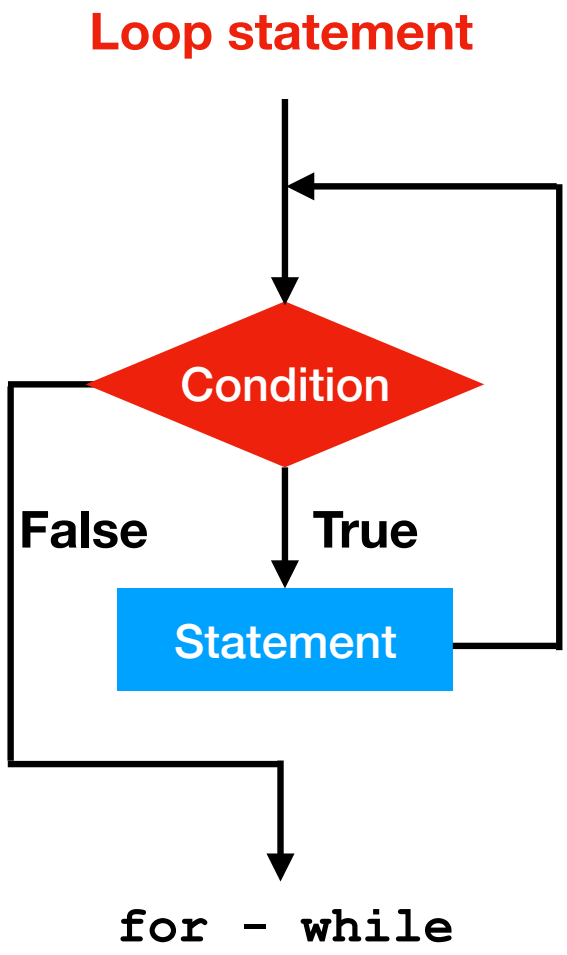
```
found = False
for i in range(len(L)):
    for j in range(i+1,len(L)):
        if L[i]+L[j] == 17 and not found:
            found = True
            pos_i = i
            pos_j = j
if found:
    print(pos_i, pos_j)
```

While loops

While

The `while` statement allows to write code that repeats as long as a certain condition is true. `while` stops iterating as soon as the condition is not true anymore.

```
while condition:
    # statements that may change
    # the condition from
    # True to False
```



While loops

Differences between `while` and `for`

- `for` element `in` collection: executes n times, where n is the length of collection.
- `while` condition: executes an indefinite number of times, as long as the condition is true.

Use of `while`

The while statement is useful when the value of `condition` can not be known beforehand, for instance when interacting with a user

```
while input("Do you want me to stop? ").lower() != "yes":
    print("Then I'll keep going!")
```

Exercise

Write a program that ask the user a question (e.g., what is the capital of Italy) and keep repeating the question until the answer is correct

```
answer = ""
while (answer != "rome"):
    print("What is the capital of Italy?")
    answer = input().lower()
    if answer != "rome":
        print("Sorry, wrong answer; retry")
```

Exercise

Write a program that ask the user a question (e.g., what is the capital of Italy) and repeats the question three times or until the answer is correct.

```
answer = ""
attempts = 0
while (answer != "rome" and attempts < 3):
    print("What is the capital of Italy?")
    answer = input().lower()
    if answer != "rome":
        print("Sorry, wrong answer; retry")
        attempts = attempts+1
if (attempts == 3):
    print("The capital is Rome! Goat!")
else:
    print("Very good")
```

Exercise

Let L be a list of integers and let $L_{sum}(k) = \sum_{i=0}^{k-1} L[i]$. Write a program that takes L and an integer value *threshold* as input and prints the number of elements k such that $L_{sum}(k) \geq$ *threshold*. Print -1 if the sum of all elements is smaller than *threshold*.

For example,

- If $L = [1, 4, 3, 12, 7]$ and *threshold* $= 7$, the output should be 3, meaning that the sum of the first three elements (8) is larger than *threshold*.
- If $L = [1, 2, 3, 4]$ and *threshold* $= 11$, the output should be -1, meaning that the sum of all elements (10) is smaller than *threshold*.

Exercise

```
L = list(range(10))
threshold = 20
tot = 0
i = 0
while (tot <= threshold and i < len(L)):
    tot = tot + L[i]
    i = i+1
if (tot >= threshold):
    print(i-1)
else:
    print(-1)
```

Exercise

Write a program that creates a list with all the Fibonacci number smaller than 1,000,000.

```
F = [1,1]
over = False

while not over:
    next = F[-1]+F[-2]
    if next > 1000000:
        over = True
    else:
        F.append(next)
```

```
F = [1,1]
while F[-1]<1000000:
    F.append(F[-1]+F[-2])
F.pop(-1)
```

Exercise

Exercise

The $3n+1$ sequence is defined like this: given a number n, compute a new value for n as follow: if n is even, divide n by 2. If n is odd, multiply it by 3 and add 1. Stop when you reach the value of 1.

Example: for $n = 3$, the sequence is $[3, 10, 5, 16, 8, 4, 2, 1]$.

Write a program that creates a list D , such that for each value n between 1 and 50, $D[n]$ contains the length of the sequence so generated. In case of $n = 3$, the length is 8. In case of $n = 27$, the length is 111.

Use of `while` – Exercise

```
MAX = 51
L = [0]*MAX
for n in range(1,MAX):
    count = 0
    start = n
    while n > 1:
        if (n % 2 == 0):
            n = n // 2
        else:
            n = 3*n+1
        count = count + 1
    L[start] = count
print(L[1:])
```

Break and continue – hate and love

break

Inside a loop (either `for` or `while`), a `break` statement interrupts the execution of the loop.

```
sums = []
tot = 0
for x in range(1,100):
    tot = tot+x
    if tot>300:
        break
    sums.append(tot)
print(sums)
```

```
[1, 3, 6, 10, 15, 21, 28,
 36, 45, 55, 66, 78, 91,
 105, 120, 136, 153, 171,
 190, 210, 231, 253, 276,
 300]
```

Break and continue – hate and love

break

This version prints the same output as before; it is much better without a `break`!

```
sums = []
tot = 0
x = 1
while (tot+x <= 300):
    tot = tot + x
    sums.append(tot)
    x = x + 1
print(sums)
```

```
[1, 3, 6, 10, 15, 21, 28,
 36, 45, 55, 66, 78, 91,
 105, 120, 136, 153, 171,
 190, 210, 231, 253, 276,
 300]
```

Break and continue – hate and love

continue

Inside a loop (either `for` or `while`), a `continue` statement interrupts the current iteration of the loop and skip to the next one.

```
LS = []
for x in range(1,31):
    if x%2 != 0 and x%3 != 0:
        continue
    LS.append(x)
print(LS)
```

```
[2, 3, 4, 6, 8, 9, 10, 12,
 14, 15, 16, 18, 20, 21,
 22, 24, 26, 27, 28, 30]
```

Break and continue – hate and love

continue

This version prints the same output as before; it is much better without a `continue`!

```
LS = []
for x in range(1,31):
    if x%2 == 0 or x%3 == 0:
        LS.append(x)
print(LS)
```

```
[2, 3, 4, 6, 8, 9, 10, 12,
 14, 15, 16, 18, 20, 21,
 22, 24, 26, 27, 28, 30]
```

Break and continue behavior

```
for number in range(10):
    print(number, end=" ")
print("")
```

```
0 1 2 3 4 5 6 7 8 9
```

```
for number in range(10):
    print(number, end=" ")
    break
print("")
```

```
0
```

```
for number in range(10):
    print(number, end=" ")
    continue
print("")
```

```
0 1 2 3 4 5 6 7 8 9
```

Break and continue behavior

```
for number in range(1,11):
    print(number, end=" ")
    if number % 4 == 0:
        break
print("")
```

```
1 2 3 4
```

```
for number in range(1,11):
    if number % 4 == 0:
        break
    print(number, end=" ")
print("")
```

```
1 2 3
```

Avoid break and continue!

Using break and continue frequently makes code hard to follow. But if replacing them makes the code even harder to follow, then that's a bad change.

E.W. Dijkstra. *Go To Statement Considered Harmful.* Communications of the ACM, Vol. 11 (1968) 147-148

List Comprehension

List comprehension

The list comprehension operator allows to filter or transform a list. The original list is left unchanged. A new list is created instead.

The abstract syntax is:

```
filtered = [expression(element)
            for element in original
            if condition(element)]
```

List Comprehension

List comprehension as a filter

Given an arbitrary iterable object `original`, we can create a new list that only contains those elements of `original` that satisfy a given condition.

The abstract syntax is:

```
filtered = [element
            for element in original
            if condition(element)]
```

List comprehension - example

Example

Generate the list of natural numbers smaller or equal to 30 that are divisible by either 2 or 3

$$L = \{n : 1 \leq n \leq 30 \textbf{ and } (n \bmod 2 = 0 \textbf{ or } n \bmod 3 = 0)\}$$

```
L = [n for n in range(1,31) if n%2 == 0 or n%3 == 0]
```

```
[2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 16,
 18, 20, 21, 22, 24, 26, 27, 28, 30]
```

List comprehension - exercise

Exercise

Given a list of DNA sequences represented as strings, returns only those sequences that contain at least one adenosine ("A")

```
sequences = ["ACTGG", "CCTGT", "ATTTA", "TATAGC"]
L = [seq for seq in sequences if "A" in seq]
```

```
['ACTGG', 'ATTTA', 'TATAGC']
```

List comprehension - exercise

Exercise

Given a list of telephone numbers represented as strings, returns only those that are from Verona (prefix "045")

```
numbers = ["04599904523", "0461304534",
           "0288662244", "0458346157"]
L = [number for number in numbers if number.startswith("045")]
```

```
['04599904523', '0458346157']
```

Note

The name of the "temporary" variable holding the current element (in the examples above, `n` and `seq`, respectively) is arbitrary.

These pieces of code are identical:

```
L = [n for n in range(1,31) if n%2 == 0 or n%3 == 0]
```

```
L = [pippo for pippo in range(1,31)
           if pippo%2 == 0 or pippo%3 == 0]
```

The name of the variable does not make any difference. You are free to pick any name you like, but choose something meaningful.

List Comprehension

List comprehension as transformation

Given an arbitrary list `original`, we can use a list comprehension to also transform the elements in the list in some way.

The abstract syntax is:

```
transformed = [expression(element)
               for element in original]
```

The `expression` should be based on `element`, but sometimes it is possible that they are completely independent.

List comprehension - exercises

Exercise

Generate the list of the squares of the natural numbers between `1` and `30`.

$$L = \{n^2 : 1 \leq n \leq 30 \}$$

```
L = [n*n for n in range(1,31)]
print(L)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
 225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625,
 676, 729, 784, 841, 900]
```

List comprehension – exercises

Exercise

Given the list of strings representing (part of) the 3D structure of a protein chain, compute a list of lists which should hold, for each residue (that is, for every row of atoms), its coordinates.

```
atoms = [
    "SER A 96 77.253 20.522 75.007",
    "VAL A 97 76.066 22.304 71.921",
    "PRO A 98 77.731 23.371 68.681",
    "SER A 99 80.136 26.246 68.973",
    "GLN A 100 79.039 29.534 67.364",
    "LYS A 101 81.787 32.022 68.157",
]

coords = [row.split()[-3:] for row in atoms]
print(coords)
```

List comprehension – example

But! The results is a list of list of strings, I need float coordinates!
What we can do?

```
coords = [
    [float(coord) for coord in row.split()[-3:]]
    for row in atoms]
print(coords)
```

```
[[77.253, 20.522, 75.007], [76.066, 22.304, 71.921], ...
```

List comprehension – example

But! Nice list of lists, but I need a list of tuples - I will use them later in a dictionary!

```
coords = [
    tuple([float(coord) for coord in row.split()[-3:]])
    for row in atoms]
print(coords)
```

```
[(77.253, 20.522, 75.007), (76.066, 22.304, 71.921), ...
```

List Comprehension

All together, now!

Given an arbitrary list `original`, we can use a list comprehension to transform and filter the elements in the list at the same time

The abstract syntax is:

```
transformed = [expression(element)
               for element in original
               if condition(element)]
```

List Comprehension

Using the example developed so far, we want just to list the coordinates of serines.

```
coords = [
    tuple([float(coord) for coord in row.split()[-3:]])
    for row in atoms
    if row.startswith("SER")]
print(coords)
```

```
[(77.253, 20.522, 75.007), (80.136, 26.246, 68.973)]
```

List comprehension and lists of lists

Let's say that we want to create a bi-dimensional matrix $n \times n$, initialized to zero. Let's write in this way

```
n = 4
L = [[0]*n]*n
print(L)
```

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

What happens if I execute `L[1][1] = 5`?

```
[[0, 5, 0, 0], [0, 5, 0, 0], [0, 5, 0, 0], [0, 5, 0, 0]]
```

Why?

List comprehension and lists of lists

The correct way of initializing a list of lists is to use list comprehension (or a loop)

```
n = 4
L = [[0]*n for i in range(n)]
L[1][1] = 5
print(L)
```

```
[[0, 0, 0, 0], [0, 5, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Exercise

Exercise

Given a number `n`, print all the possible ways to obtain it as a products of two integers. Pay attention at commutative repetitions (e.g., 4x9 and 9x4)

```
n = 36
L = []
for i in range(n+1):
    for j in range(i,n+1):
        if (i*j==n):
            L.append((i,j))
print(L)

# Through list comprehension
L = [(i,j) for i in range(n+1) for j in range(i,n+1) if i*j==n]
```

Exercise

Problem

Given a list of values, generate the sublist of values that only appear once in the list. For example, `L = [1,3,2,3,5,4,5,3,3]` should generate `[1, 2, 4]`

```
L = [1,3,2,3,5,4,5,3,3]
SL = [n for n in L if L.count(n)==1]
print(SL)
```

Exercise

Problem

Given a list of values, potentially with repeated values, generate the ordered sublist of values where repeated value are removed. For example, `L = [1,3,2,3,5,4,5,3,3]` should generate `L = [1,2,3,4,5]`

```
L = [1,3,2,3,5,4,5,3,3]
SL = []
for n in L:
    if (n not in SL):
        SL.append(n)
print(SL.sort())
```

Exercise

Problem

Given two lists of values (with no repetitions), generate the sublist of values that appear in both of them.

```
L1 = [1,7,9,3,23,11]
L2 = [7,23,2,4,8,16]

SL = []
for v in L1:
  if v in L2:
     SL.append(v)
print(SL)
```

Exercise

Problem

Given two lists of ordered values, generate the sublist of values that appear in both of them. Exploit the fact that the lists are ordered.

```
A = [1,2,4,8,16,32]
B = [4,8,12,16,20]
LS = []
a = b = 0
while (a < len(A) and b < len(B)):
    if A[a] == B[b]:
        LS.append(A[a])
        a = a+1
        b = b+1
    elif A[a] < B[b]:
        a = a+1
    else:
        b = b+1
print(LS)
```

Exercise

Exercise - Sudoku

Given a matrix 9×9, print `True` if the matrix is an acceptable Sodoku solution, print `False` otherwise.

```
L = [[1, 2, 3, 4, 5, 6, 7, 8, 9],
  [4, 5, 6, 7, 8, 9, 1, 2, 3],
  [7, 8, 9, 1, 2, 3, 4, 5, 6],
  [2, 3, 4, 5, 6, 7, 8, 9, 1],
  [5, 6, 7, 8, 9, 1, 2, 3, 4],
  [8, 9, 1, 2, 3, 4, 5, 6, 7],
  [3, 4, 5, 6, 7, 8, 9, 1, 2],
  [6, 7, 8, 9, 1, 2, 3, 4, 5],
  [9, 1, 2, 3, 4, 5, 6, 7, 8] ]
```

Exercise

Given the list of strings:

```
table = [
    "protein domain start end",
    "YNL275W PF00955 236 498",
    "YHR065C SM00490 335 416",
    "YKL053C-A PF05254 5 72",
    "YOR349W PANTHER 353 414",
]
```

- write a program that takes the column names from the first row of table;
- for each row creates a dictionary like this:

```
dictionary = {
    "protein": "YNL275W",
    "domain": "PF00955",
    "start": "236",
    "end":, "498"
}
```

- append each of these dictionaries to a list.

Exercise

Given a long text represented as a string, compute the frequencies of each words that appear in it and store it in a dictionary that associates words to frequencies.

```
text = """Nel pozzo di San Patrizio c'e' una pazza che lava
una pezza. Arriva un pazzo, con un pezzo di pizza e chiede
alla pazza se ne vuole un pezzo. La pazza rifiuta. Allora
il pazzo prende la pazza, la pezza e la pizza e li butta
nel pozzo di San Patrizio, protettore dei pazzi.
"""
```