ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Lesson 3

ILAI (M1) @ LAAI I.C. @ LM AI

23 September 2024

Michael Lodi

Department of Computer Science and Engineering

These slides draw very heavily from Simone Martini's slides.

One-slide recap of Lecture 2

Function: sequence of commands with a name. Returns a value.

Defined with `def`, name, parenthesis, formal parameters (names)

Called with name and actual parameters (*arguments*, are expressions)

Value of actual parameters bound to the formal parameters, which are local names

Also any name occurring to the left of an assignment is local to the function

Function without return, return the value `None` (of type `NoneType`): a value of expressions for which the value is irrelevant (we are interested in «side effects», actions on the internal state).

We can import names (and functions) from libraries with `import` or `from … import`

We can use `for <name> in <sequence>`: to scan every element of a sequence. `<name>` is just a name like all the others (no local scope).

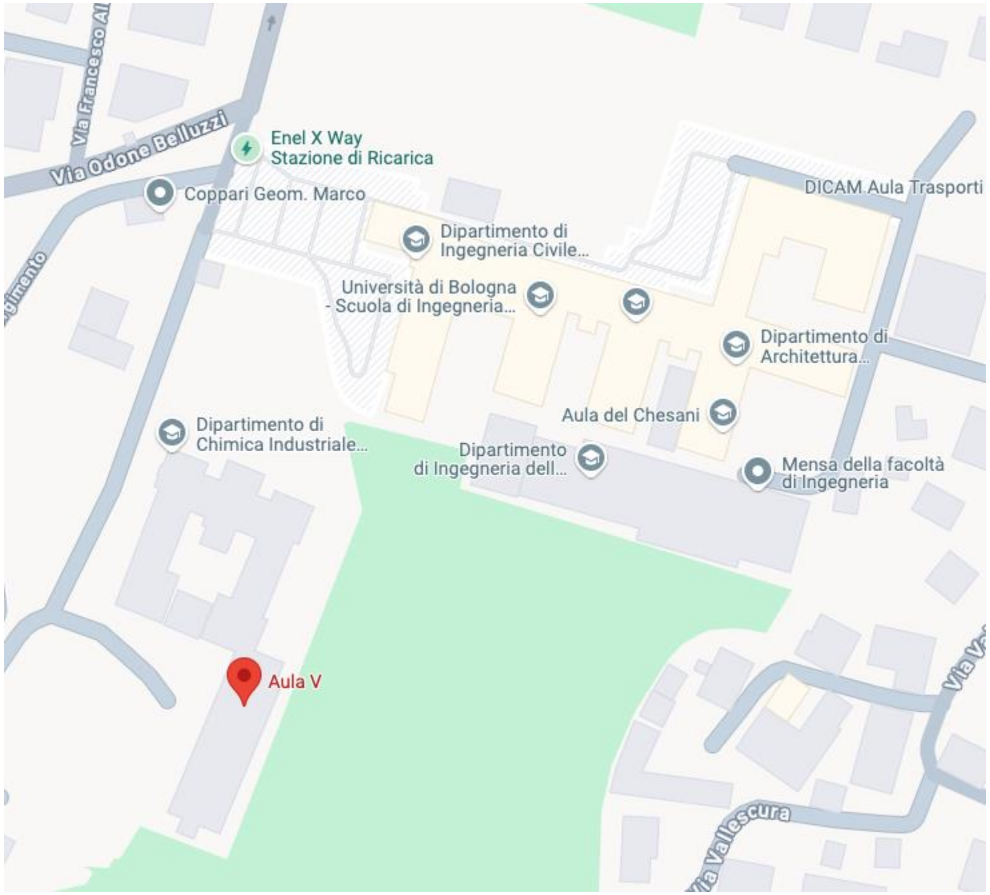
The «`in`» in the `for` should not be confused with the boolean operator `in`, which returns `True` if an element (for strings only, also substrings) appear in a sequence.

This week lectures for ILAI

Wednesday 25 Sept, 9.15 – 11.30, Room V, Via Risorgimento 4 (NEW!)

Thursday 26 Sept, NO LECTURE

ALL UPDATES ALREADY
ON THE COURSE WEBSITE



Pending question: if Python interpretes code line by line, how does he know the local names inside a fucntion?

```
def f(x):
    print(a)
    print(x)
    a = 42
    return a + x
```

- You interpret each **command**, non each line («def» is a compound command) [shell demo]

- Python implementations usually have a compilation step.

- «Static»/«lexical» scope

Cpython implementation

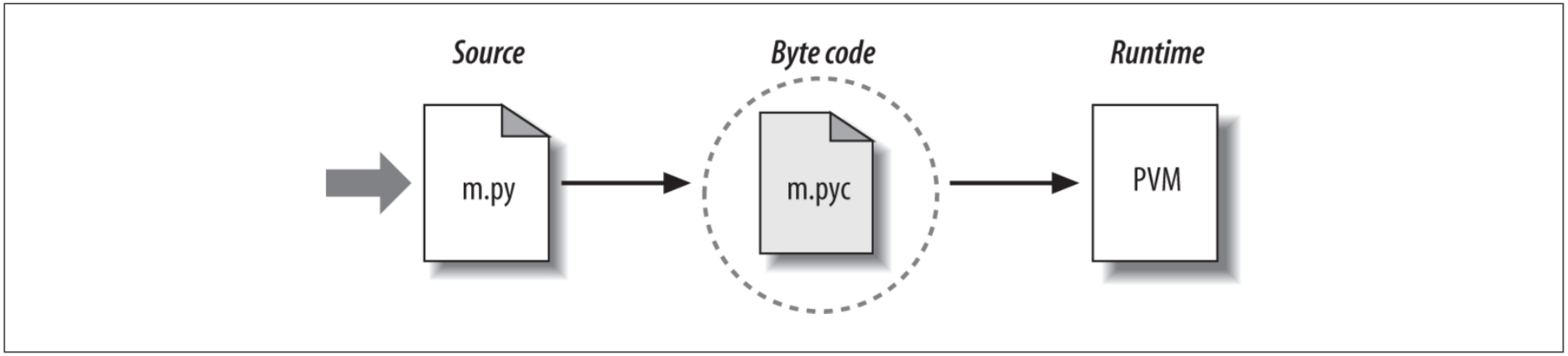


Figure 2-2. Python's traditional runtime execution model: source code you type is translated to byte code, which is then run by the Python Virtual Machine. Your code is automatically compiled, but then it is interpreted.

Mark Lutz, Learning Python, p. 32

«Disassembling» (from dis import dis)

```
def f():
    print(a)
    a = 1
```

```
 0           0 RESUME                   0

  2           2 LOAD_CONST               0 (<code object f at
0x10ad34ff0, file "<dis>", line 2>)
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (f)
              8 LOAD_CONST               1 (None)
             10 RETURN_VALUE

Disassembly of <code object f at 0x10ad34ff0, file "<dis>", lin
2>:
  2           0 RESUME                   0

  3           2 LOAD_GLOBAL              1 (NULL + print)
             14 LOAD_FAST                0 (a)
             16 PRECALL                  1
             20 CALL                     1
             30 POP_TOP

  4          32 LOAD_CONST               1 (1)
             34 STORE_FAST               0 (a)
             36 LOAD_CONST               0 (None)
             38 RETURN_VALUE
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

«Disassembling» (from dis import dis)

```
b = 2
def f2():
    print(b)
    if b == 2:
        print("ok")
```

```
 0           0 RESUME                   0

  2          2 LOAD_CONST               0 (2)
             4 STORE_NAME               0 (b)

  3          6 LOAD_CONST               1 (<code object f2 at
0x10912e790, file "<dis>", line 3>)
             8 MAKE_FUNCTION            0
            10 STORE_NAME               1 (f2)
            12 LOAD_CONST               2 (None)
            14 RETURN_VALUE

Disassembly of <code object f2 at 0x10912e790, file "<dis>",
line 3>:
  3          0 RESUME                   0

  4          2 LOAD_GLOBAL              1 (NULL + print)
            14 LOAD_GLOBAL              2 (b)
            26 PRECALL                  1
            30 CALL                     1
            40 POP_TOP

  5         42 LOAD_GLOBAL              2 (b)
            54 LOAD_CONST               1 (2)
            56 COMPARE_OP               2 (==)
```

«Disassembling» (fro

```
b = 2
def f2():
    print(b)
    if b == 2:
        print("ok")
```

```
              8 MAKE_FUNCTION            0
             10 STORE_NAME               1 (f2)
             12 LOAD_CONST               2 (None)
             14 RETURN_VALUE

Disassembly of <code object f2 at 0x10912e790, file "<dis>",
line 3>:
  3           0 RESUME                   0

  4           2 LOAD_GLOBAL              1 (NULL + print)
             14 LOAD_GLOBAL              2 (b)
             26 PRECALL                  1
             30 CALL                     1
             40 POP_TOP

  5          42 LOAD_GLOBAL              2 (b)
             54 LOAD_CONST               1 (2)
             56 COMPARE_OP               2 (==)
             62 POP_JUMP_FORWARD_IF_FALSE    17 (to 98)

  6          64 LOAD_GLOBAL              1 (NULL + print)
             76 LOAD_CONST               2 ('ok')
             78 PRECALL                  1
             82 CALL                     1
             92 POP_TOP
             94 LOAD_CONST               0 (None)
             96 RETURN_VALUE

  5     >>   98 LOAD_CONST               0 (None)
            100 RETURN_VALUE
```

Objects

An object is a value envelopped in an identity

This identity is unique during a run of the Python machine

In Python, *any value is an object* (even int values)

We may have distinct objects (with different identities) and same value

Identity of an object is completely controlled by the Python machine:

we may inspect the identity of an object : function `id( )`

but we cannot change it

Objects

An object is a value envelopped in an identity
This identity is unique during a run of the Python machine

`==` is equality between *values*

(nothing to do with identities)

`is` : comparison operator

`True` iff identities are the same

Obvious: if two objects are identical on `is`, they are also `==`

example: isequal.py

Objects: identity

- At this stage of the course, the `is` relation is not much relevant
- it will become relevant when *mutable* objects will be introduced

Important:

we can never assume identity of objects from the fact that they have the same value

On the contrary *we must always assume* that

any operation creates new objects

For: ``freezing" the sequence

```
for <name> in <sequence>:
    <block>
```

(0) <sequence> *is computed and "frozen"*

What is frozen is the object (its identity). Therefore in:

```
s='Lodi'
for c in s:
    print(c)
    s='zzzz'
```

the assignment `s='zzzzz'` happens,
but it has no effect on the repetition,
because the object `'zzzz'` is not the one controlling the for

What will this program print?

Try on your own (without running it)

```
a = "Lodi"
for c in a:
  print(c)
  c = 'z'
  a = a + c
  print(c, a)
```

Objects: methods

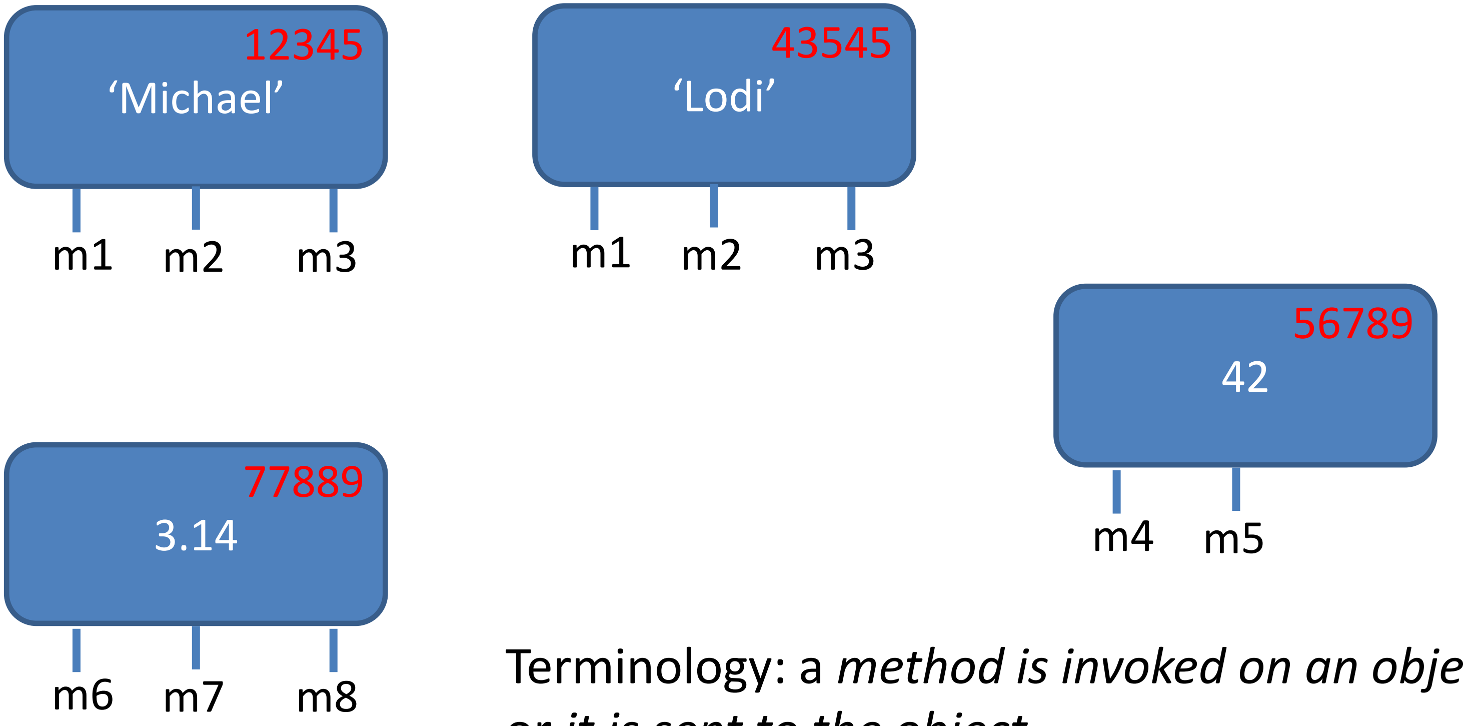
The envelope enclosing an object contains, besides its identity, also other information

An important information of an object are *its methods*

An object is an "active value":
we may stimulate the object through the *methods* offered by its envelope

Objects: methods

Objects of the same *type* share the same methods



Terminology: a *method is invoked on an object* or *it is sent to the object* or also (I like this less) it is *called on* the object

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Objects: methods

Methods act on the object on which they are invoked; they stimulate the object to do something (e.g., to return a value)

Syntax to invoke a method:

<object>.<method>(<optional arguments>)

Examples:

str offer the method upper()

```
s='bologna BO'
t=s.upper()
print(t)
```

Objects: methods

Methods act on the object on which they are invoked; they stimulate the object to do something (e.g., to return a value)

examples in Thonny (file metods_shell.txt)

Recall: the object receiving the method can be a constant, a name, an expression… (in summary: can be any Python expression evaluating to an object)

Some methods offered by str

See them at:

https://docs.python.org/3/library/stdtypes.html#string-methods

`str.capitalize()`

Return *a copy of the string* with its first character capitalized and the rest lowercased.

`str.isalpha()`

Return True if all characters in the string are alphabetic and there is at least one character, False otherwise.

`str.isdigit()`

Return True if all characters in the string are digits and there is at least one character, False otherwise.

Some methods offered by str

`str.lower()`

Return a copy of the string with all the cased characters converted to lowercase.

Methods with parameters (observe the notation used by the documentation) :

`str.find(sub[, start[, end]])`

Return the lowest index in the string where substring *sub* is found within the slice s[start:end]. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 if *sub* is not found.

`str.count(sub[, start[, end]])`

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Methods offered by float

`fl.as_integer_ratio()`

Return a pair of integers whose ratio is exactly equal to the original float and with a positive denominator

`fl.is_integer()`

Return True if the float instance is finite with integral value, and False otherwise:

Some methods offered by sequences, hence by str and other sequence types we will see

`s.index(x)`

Return the index of the first occurrence of *x* in *s;*

it is an error if x is not an element of s

`s.count(x)`

total number of occurrences of x in s

A new type: tuple

- `int, float, bool, str , NoneType`

  are simple types

- tuples are a structured type (compound type):

  its values are groups of values

  each of them of arbitrary type

- tuples are *immutable sequences* of objects
  - (strings are also immutable sequences)

tuple

Values: finite sequences of values
each of them of arbitrary type

Presentation: `(<object1>, … , <objectk>)`

or also without parentheses (⚠️) :
the comma `,` is the constructor for tuples

`()` is the empty tuple

the tuple with a single element is `(<element>,)`

Operations: concatenation (`+`), repetition (`*`), length `len()`, selection `[…]`

They are the common operations on most other types of sequences (eg, strings)

tuple

Examples:

```
(10.3, 100, 'simone')
(3,)
()
(1, 2, (100,200), 3)
(10, 20,)
10,20
```

Non examples:

```
(3)
(10,,20)   {10,20}    [10,20]    (,0)
```

Indexes and scan: like for str

Indexes on tuples: similar to `str`

- they start from 0
- negative indexes: count from the end

We may perform a

`for`

on a tuple

```
for e in (1,2,’bologna’,(3,4)):
    print(e)
```

Operations create new objects

```
T = (1,2,3)
T = T + (4,5)
```

Any operation creates a new object!

On the contrary…

```
V = (100,200)
S = V
```

`S` and `V` are two names for the same object

tuple() as type converter

As any name of a type, `tuple()` may be used to convert values into tuples

```
tuple()                ()
tuple('bolo')          ('b','o','l','o')
tuple((1,2,3))         (1,2,3)

tuple(100)             error

tuple(range(2,6))   (2,3,4,5)
```

Select and “name” elements

```
T = (1,2,(100,200),3)
X = (10,20)
W = (9,X,11)
Z = T[2]
```

➔try them in Pythontutor

aliasing is possible: different names (the aliases) refer to the same object

Aliasing on tuple is harmless, because tuple are immutable

Generalised assignment

Standard assignment:

```
<name>=<expression>
```

Generalised:

```
<name1>,…,<namek>=<sequence with exactly k elements>
```

Recall how an assignment is evaluated!

1. Evaluate the RHS, and obtain its value
2. Bind this value to the name(s) in the LHS

Examples

```
A,B = (1,2)
C, D, E = A+1, 5, B
C, E = C+1, E+1
```

Same number of elements on the two sides !

Examples

```
A,B = (1,2)
C, D, E = A+1, 5, B
C, E = C+1, E+1
```

Same number of elements on the two sides !

```
A,B = B,A
```

this the “Pythonic” way of swapping two bindings

Selecting portions of sequences: slice

A *slice* is a *subsequence*

At first sight: a generalization of "selection of one element"

```
s='Bologna b school'
print(s[0])           # 'B'
print(s[8:11])        # 'b s'
print(s[:7])          # 'Bologna'
print(s[10:])         # 'school'
```

Slice

`<sequence>[<start>:<end>]`

Semantics:

subsequence of *<sequence>* starting at *<start>* index (included) up to *<end>* index (*non included*)

Warning: *<start>* < *<end>*

otherwise the sequence is empy

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Slice

Examples:

-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7	8	9
b	o	l	o	g	n	a		B	O

```
s=‘bologna BO’
```

Who are they?

```
s[0:4]   bolo
s[3:-1]  ogna B
s[3:2]
s[3:4]     o
s[3:3]
```

Slice

Examples:

-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7	8	9
b	o	l	o	g	n	a		B	O

s[: <end>] starts at 0
s[<start> :] ends at len(S)+1, last element included

```
s=‘bologna BO’
```

Who are they?

```
s[:4]
s[3:]
s[:]
s[:-1]
```

Slice: is *not* a *generalised selection*

Warning:

a slice is a «window» on the sequence

-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7	8	9
b	o	l	o	g	n	a		B	O

`s=‘bologna BO’`

what is `s[6:12]` ?

Slice: step

<*sequence*>[<*first_index*> : <*last_index*> : <*step*>]

Semantics:
subsequence starting at <first_index>,
terminating at <last_index>-1,
each element <step> elements distant from the preceding one

<*sequence*>[<*first_index*>: <*last_index*>]
is shorthand for
<*sequence*>[<*first_index*>: <*last_index*> : 1]

Slice: step

subsequence starting at <first_index>, terminating at <last_index>-1, each element <step> elements distant from the preceding one

-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7	8	9
b	o	l	o	g	n	a		B	O

```
s[1:9:2]
s[ : 100 : 4]
s[3 : : 3]
s[3 :-1 : 3]
```

Slice: negative step

When the step is negative, the subsequence is extracted
still from <start> included to <end> excluded
from the end; in this case <*start*> > <*end*>
so it is extracted backwards

-10	**-9**	**-8**	**-7**	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
b	**o**	**l**	**o**	**g**	**n**	**a**		**B**	**O**

`s[5:2:-1]`	`ngo`
`s[ : 3:-1]`	from end of the sequence (last included!)
`s[6::-1]`	to start (NB: 0 included!)
`s[6:0:-1]`	(NB: 0 (<end>) excluded!)
`s[ : : -1]`	(copy of) s reversed
`s[8:2:-3]`	
warning: `s[8:-1:-1]`	

Operations create new objects

On integers is obvious:

```
x=8
x=x+1
```

In the second assignment, we RHS evaluates to 9, which is bound to name x

9 is a new value, is not (obviously!) a modification of 8

(which does not make any sense)

The sametrue for other types!

And then it appears less is obvious…

Operations create new objects

On tuples:

```
x=(10,20,30)
y = x
x= x + (40,)
```

In the second assignment, we RHS evaluates to (10,20,30,40) which is bound to name x

(10,20,30,40) is a new value, is not a modification of (10,20,30)

(which does not make any sense… because tuples are immutable!)

operations create new objects

Slices, like any other operation, create new objects!

```
S = (10,20,30)
W = S[:]
V = S
```

`S` and `V` are *two names* for the *same object* (they are *aliases*)

`W` is bound to a new object, which has the same value of `S`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Linear scanning of a sequence: backwards

```
S='bologna'
for c in S[::-1]:
    print(c)
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Exercises

Write on the whiteboard «Strings and tuples are immutable» 1000 times

- I need a «repeat 1000 times»

Write a funciton taking as argument a sequence S and returning True if and only if there are two consecutive equal elements

- I need to access and manipulate indexes of S

Special sequences: range

```
range(<start>,<end>,<step>)
```

sequence of integers between <start> and <end>-1, each <step> apart from the previous

*+ and * are not defined on ranges*

`range(<end>)` shorthand for `range(0,<end>,1)`

If `<end> < <start>` range is empty

negative step: same as for slices

Special sequences: range

```
range(<start>,<end>,<step>)
```

sequence of integers between <start> and <end>-1, each <step> apart from the previous

We may perform a `for` *on a range!*

Exercises

Write on the whiteboard «Strings and tuples are immutable» 1000 times

- I need a «repeat 1000 times»

Write a funciton taking as argument a sequence S and returnit True if and only if there are two consecutive equal elements

- I need to access and manipulate indexes of Python

introrange.py

Using range on other sequences

Range allows to access the indexes of a sequence

Given a tuple `T`,

count the number of *identical contiguous pairs*

Ex: on `(1,2,2,3,4,4,4,5)` return `3`

Example with range

Given a tuple T,

count the number of *identical contiguous pairs*

```
def countpairs(T):
    res=0
    for i in range(len(T)-1): #end one index before
        if T[i]==T[i+1]:
            res = res +1
    return res
```

Lab 1 - sequences

On virtuale

Unbounded iteration

Problem:

We randomly extract integers from 0 to 99

Given an integer x (between 0 and 99)

We want to count how many extractions we perform before extracting x

Unbounded iteration

Problem:

We randomly extract integers from 0 to 9 and count

Given an integer x (between 0 and 9)

We wanto to count how many extractions we perform before extracting x

We don’t know *beforehands* how many extractions!

bounded iteration is not the right construct!

file: while.py

while command
Unbounded iteration

```
while < bool-expr> :
    <block>
```

Semantics:

1. Evaluate < bool-expr> (the *guard*)

2.1 If the guard is False, terminate the command (i.e., proceed with the command following the block, at the indentation of «while») ("*we exit from while*")

2.2 If the guard is True, evaluate <block> (the «while» *body*) ("*we enter inside the while* ")

3. Go back to step (1)

Bounded (`for`) iteration: ingredients

```
for i in sequence:
    <body>
```

1. What to repeat: <body>
2. Initial value for `i` : first element of `sequence` (if any)
3. How to pass to next iteration: `i` takes *next* element of `sequence`
4. Termination: `sequence` is exausted

All these ingredients *must be given explicitely*
in an unbounded iteration (`while`)

unbounded (`while`) iteration: ingredients

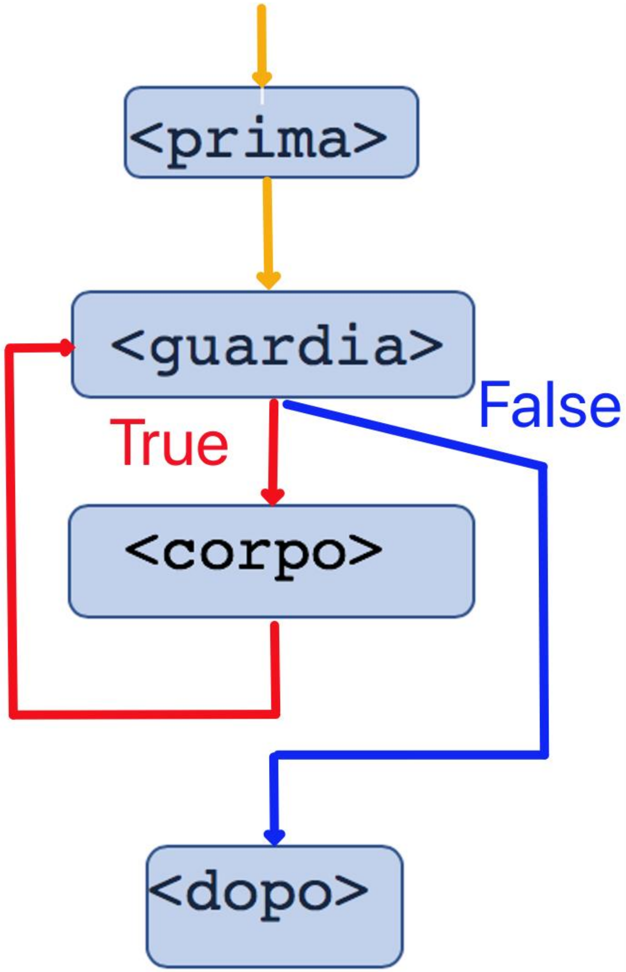
```
<before>
while  <guard> :
   <body>
```

1. What to repeat: <body>
2. Initial value for the guard : ➔ *before the while*
3. How to pass to next iteration: ➔ *inside the <body> the guard may (should) change*
4. Termination: ➔ *<guard>*

All these ingredients *must be given explicitely*
in an unbounded iteration (`while`)

while

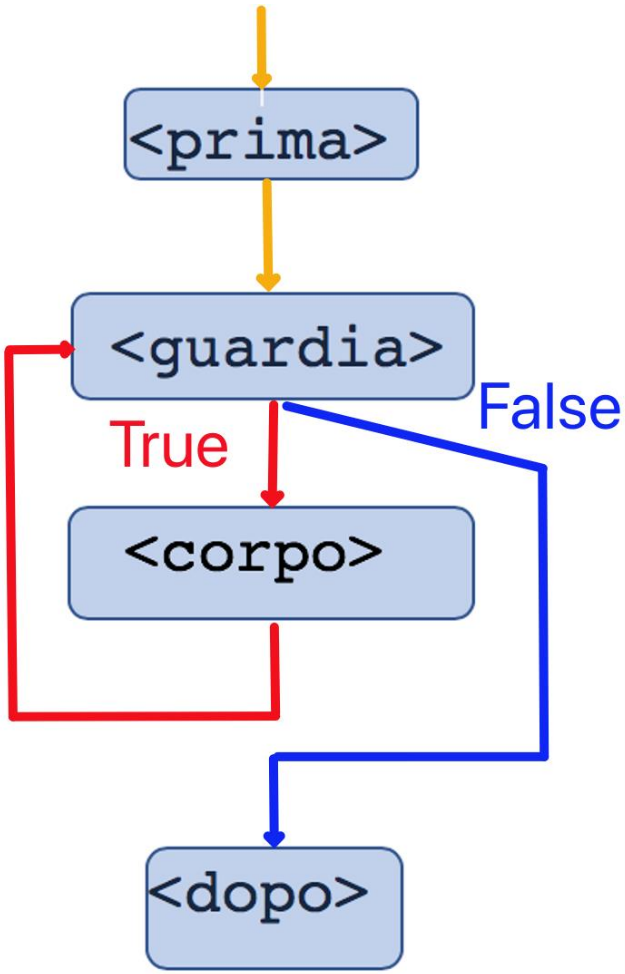
```
<before>
while <guard>:
    <body>
<after>
```



while

```
<before>
while <guard>:
        <body>
<after>
```

1. <before> must guarantee an initial value to <guard>
2. <body> must modify the value of <guard>



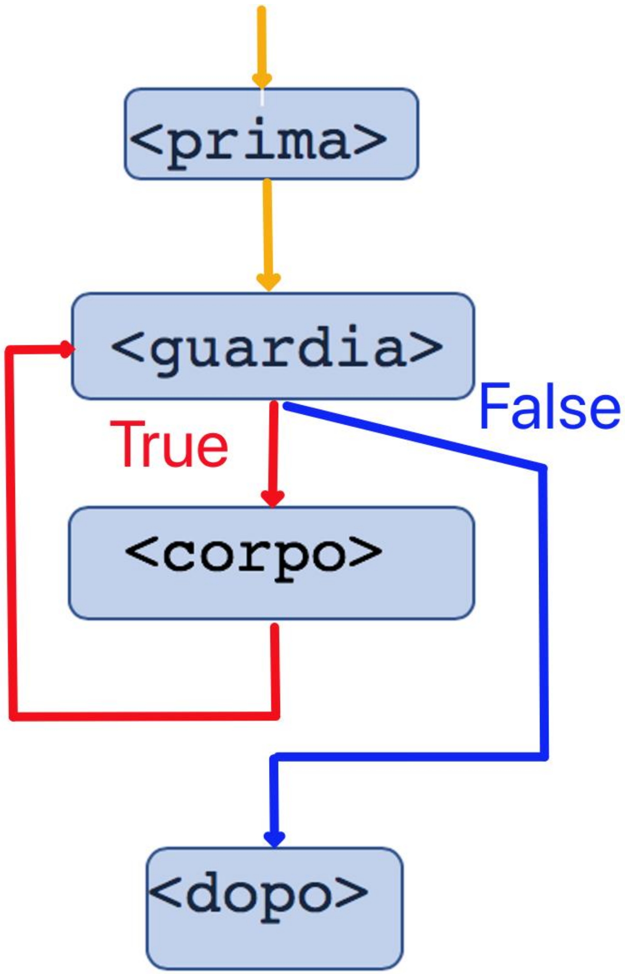
while: programming

```
<before>
while <guard>:
    <body>
<after>
```

A "while" command is never alone

- before initializations
- body must modify the guard

They are *not* grammatical requirements: good programming practice



for expressed through while

```
for i in range(len(s)):
    <body>
```

With some approximation corresponds to:

for expressed through while

```
for i in range(len(s)):
    <body>
```

With some approximation corresponds to:

```
i=0
while i < len(s):
    <body>
    i = i+1
```

It does not take into account the "freezing" of the sequence…

Unlike for, Python’s while is very similar to many other languages ☺

But compare:

```
sum=0
for e in s:
    sum=sum+e
print(sum)
```

```
sum=0
for i in range(len(s)):
    sum=sum+s[i]
print(sum)
```

```
i=0
sum=0
while i < len(s):
    sum=sum+s[i]
    i = i+1
print(sum)
```

Every time I can, I use:

- for without indexes (directly on the sequence)
- for instead of while

Is it a correct program?

```
while 1==1:
   x=100
print(x)
```

Is it a correct program?

```
while 1==1:
   x=100
print(x)
```

Syntactically correct

Semantically wrong: it is a infinite loop which never interacts with the external world