ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Lesson 2

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

19 September 2024

Michael Lodi

Department of Computer Science and Engineering

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

One-slide Recap of L1

Computations are performed by (abstract) machines. They are a sequence of elementary operations that the machine can perform. Computations are described in a language (the language the machine can execute – in our case, the Python machine).

Elementary operations acts on values/objects, organized in types (set of values, their presentation, the elementary operations, the name of the type)

- integers (`int`, no overflow), strings (`str`, immutable), subset of rationals (`float`), truth values (`bool`, presented with capital, lazy eval of `and`/`or`)

Expression: a phrase for obtaining a value we are interested in.

Evaluated from left to right.

- elementary operations on numbers/strings/bools, `input()` (transfer to internal state, gives a `str`)

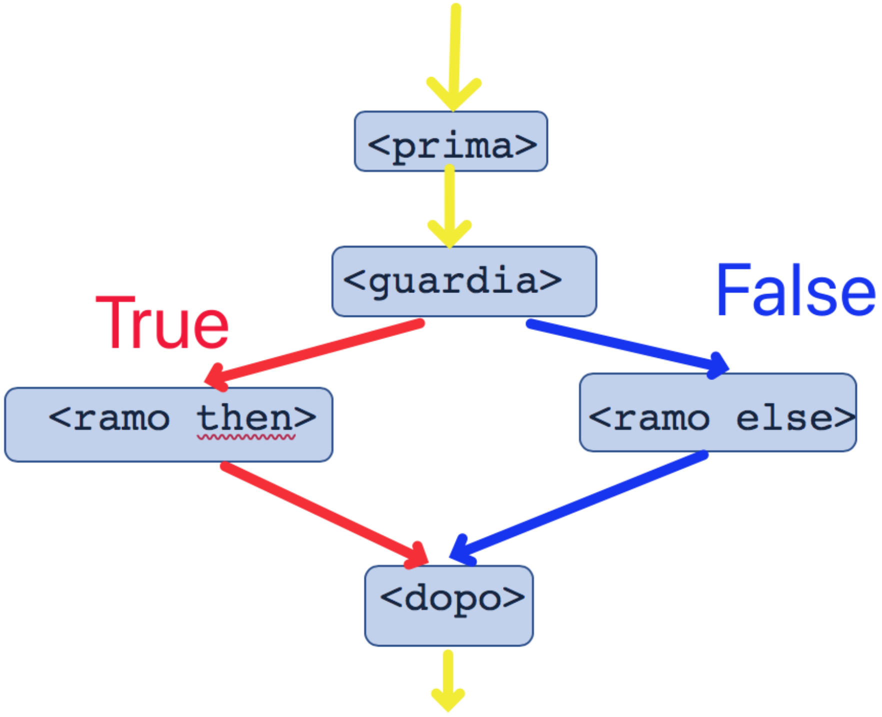
Command: a phrase we use to change the state of the machine, not interested in value

- assignment (evaluate right side first, change the internal states i.e. names assoc values); `print()` (transfer to the external state), `if`, `if-else`, `if-elif-else`

Program: plain text, each line a single command/expression, need to be run

Review: conditional command

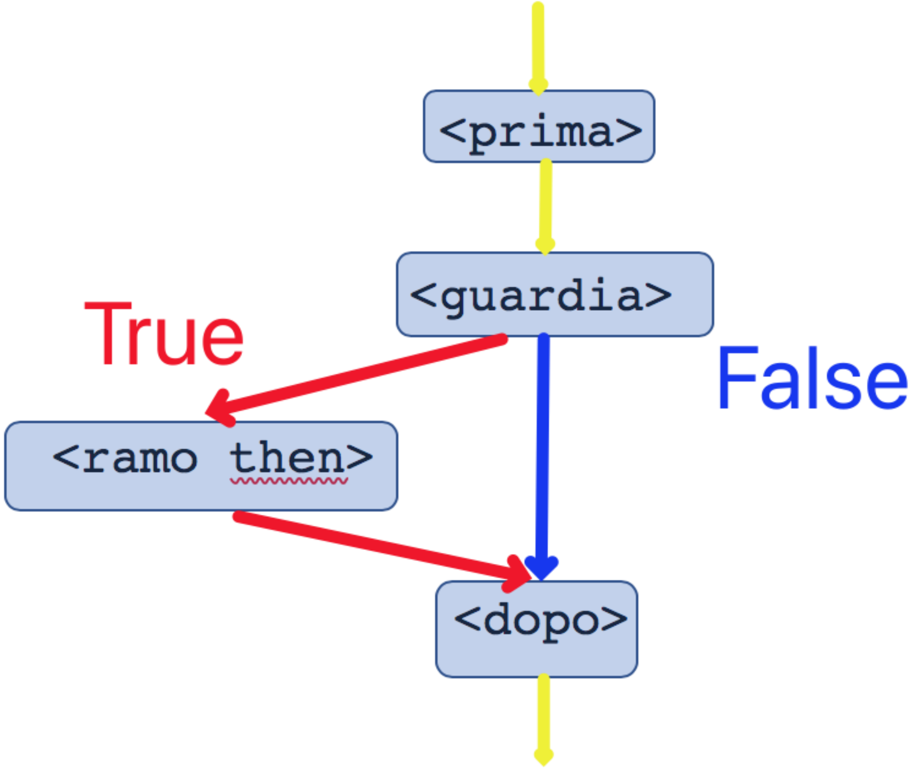
```
<prima>
if <guard>:
    <then branch>
else:
    <else branch>
<dopo>
```



Review: conditional command

else is optional

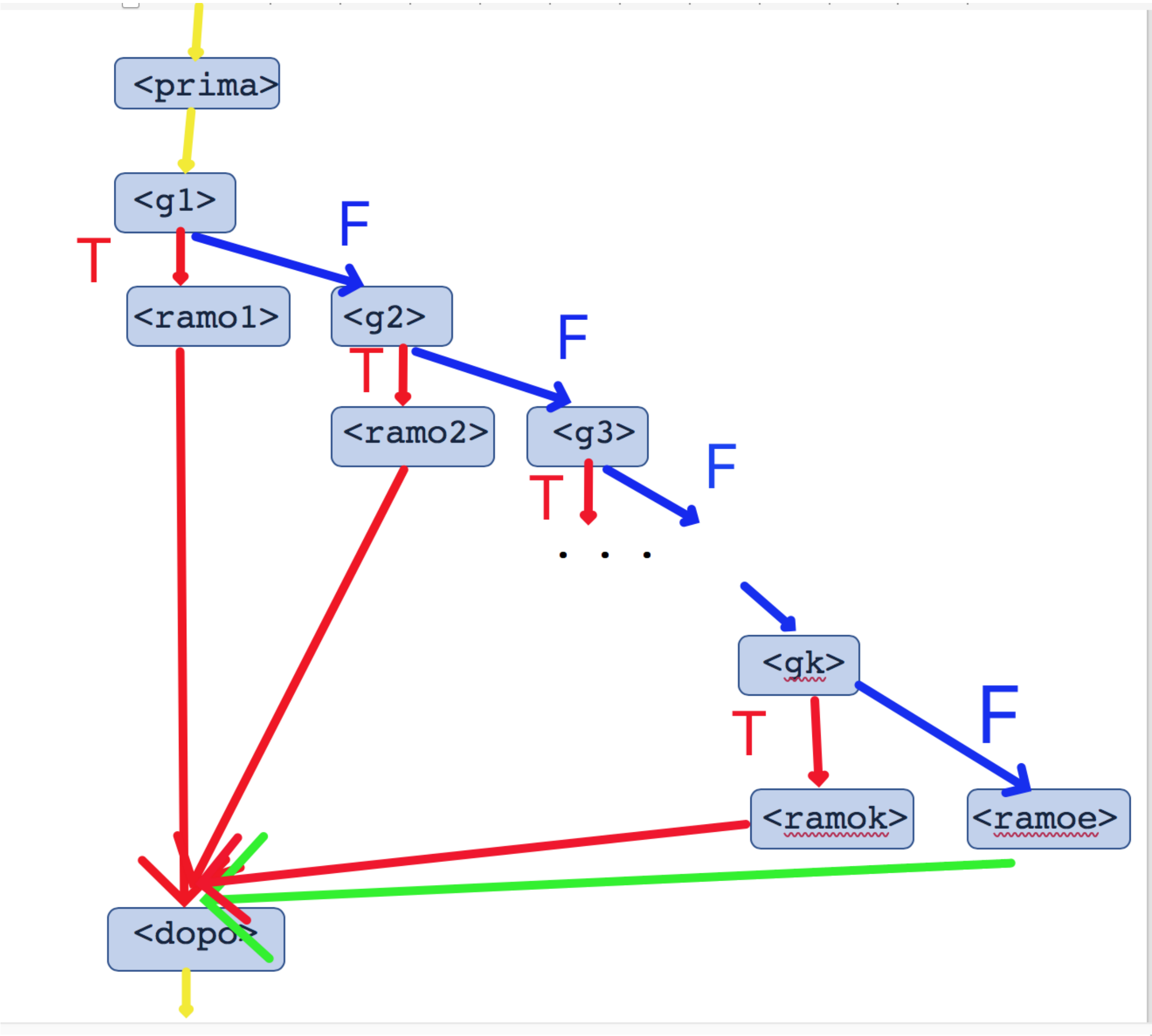
```
<prima>
if <guard>:
    <then branch>
<dopo>
```



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Review: conditional command

```
<prima>
if <g1>:
      <ramo1>
elif <g2>:
      <ramo2>
…
elif <gk>:
      <ramok>
else:
      <ramoe>
<dopo>
```



Blocks

Block:

- a sequence of lines
- each line with a single command (assignment, print, if…)
- all of them at the same *indentation*

(same distance from the left margin)

It is used to «*structure*» the execution in with compound commands (e.g., `if`)

Other programming languages use parenthesis `{,}`.

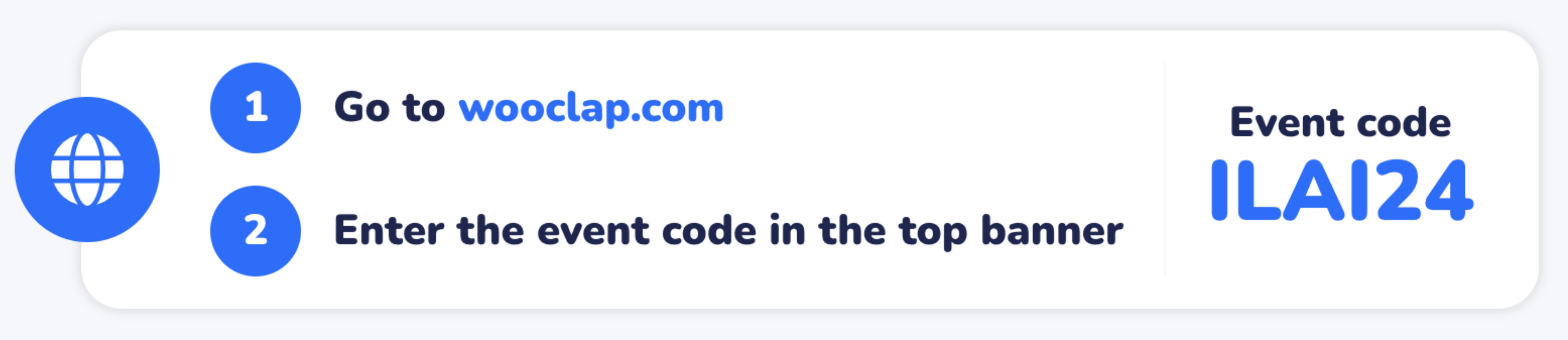
Python uses indentation

No local scopes, unless for functions or classes

scope: portion of program (execution) where that name is visible

QUIZ TIME!

Always the same link OR qr code



1 Go to wooclap.com

2 Enter the event code in the top banner

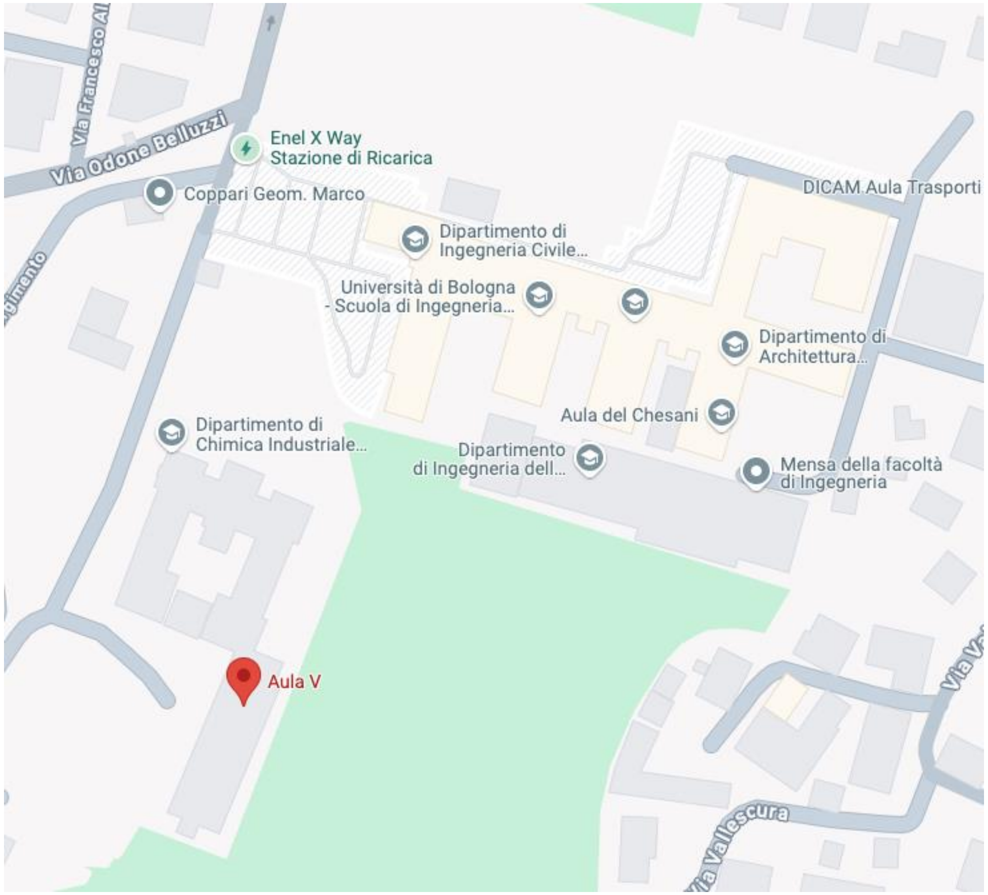
Event code
ILAI24

Next week lectures for ILAI

Monday 23 Sept, 15-18, room 0.5 (regular)

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

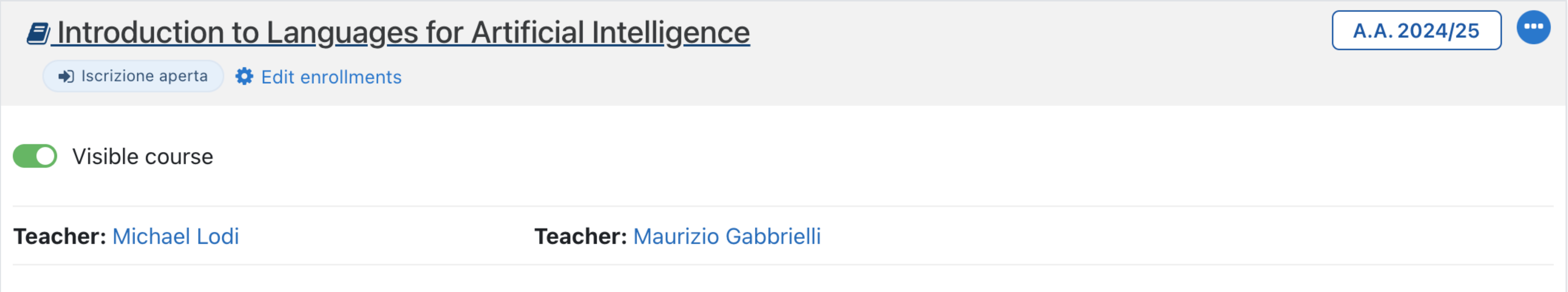
Thursday 26 Sept, NO LECTURE



ALL UPDATES ALREADY
ON THE COURSE WEBSITE

Remember to join Virtuale!

https://virtuale.unibo.it/course/view.php?id=66180



Introduction to Languages for Artificial Intelligence

Teacher: Michael Lodi

Teacher: Maurizio Gabbrielli

… and do the exercises I post ☺

I will publish past exams (⚠) very soon

Pending questions on how integers are represented inside the Python machine

There will be a lecture on this (probably L8, penultimate lecture)

- we will have to understand objects in Python first
- we will talk about operation costs in Python

In Python, everything is an object. Integers are instances of class ‘int’ (!= Java, e.g.)

Small integers in the range [-5 , 256] are pre-allocated at initialisation time. Any such object is never duplicated *(but we don’t care and always assume different objects)*

Larger integers will be allocated as (different) objects.

- integers in the range [-2^{30} , 2^{30}] have some optimizations (using underlying 32bit memory word…)

Structuring a program: functions

A *function* (in the context of programming languages) is a *named sequence of commands*, which performs a computation

Functions help to structure a program in "*subprograms*", each of which performs a simpler task

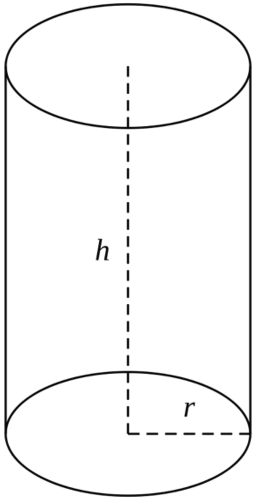
Structuring a program: functions

Compute the volume of a cylinder:

area of its basis circle

*

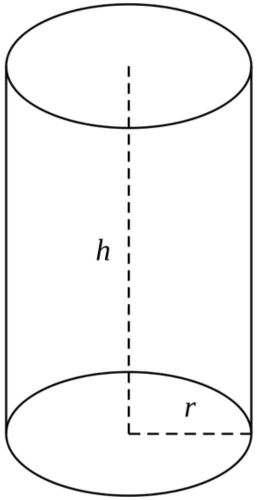
the height of the cylinder



Structuring a program: functions

```
def cylinder_vol(r,h):
    pi=3.1415
    return (pi*r**2)*h

print(cylinder_vol(2,3))
print(cylinder_vol(6,7))
```

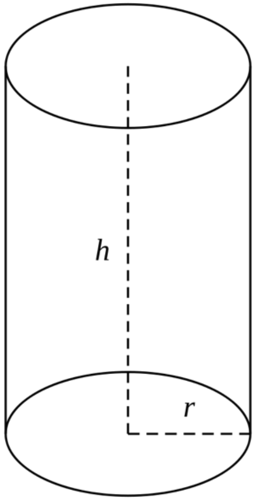


ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Structuring a program: functions

```
def cylinder_vol(r,h):
    pi=3.1415
    return (pi*r**2)*h
```

However, we may split the problem in two subproblems (*of course, this is a toy example!*)

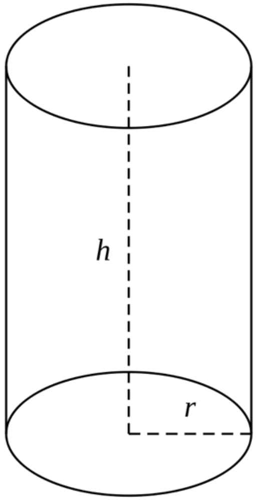


Structuring a program: functions

```
def area_circle(r):
    pi=3.1415
    return pi*r**2

def cylinder_vol(r,h):
    return area_circle(r)*h

print(cylinder_vol(2,3))
print(cylinder_vol(6,7))
```



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Functions: definitions vs use

1. Functions should be *defined*

```
def cylinder_vol(r,h):
    return area_circle(r)*h
```

A binding between the name and the *body* is inserted into the internal state. *No computation is done*

2. Functions may be *used* (*called*):

```
print(cylinder_vol(2+1,3*2))
```

arguments are bound to the parameters, and the body *is executed.* A value is *returned*.

Functions: definition

```
def <name> (<formal parameters>):
    <block>
```

header

body

def is a reserved name

It has the structure of a standard name but *cannot be used as a name* because of its special role in the header of a function. *Other reserved names we met:* if, and, or, not, …

<formal parameters> : comma separated ***names***
(can be empty – but still needs parentheses)

<block> : is a sequence of commands all at the same indentation

A ”def” is a compound command

Functions: use

To use (call) a function we use the syntax:

<name>(<list of arguments>)

which is an *expression*

<list of arguments> : a comma separated list of **expressions** (**in the same number** of the formal parameters in the header of the function)

(can be empty – but still needs parentheses)

Also known as: *actual parameters*

return

```
return <expression>
```

is a command that may appear *only* inside the body of a function

When executed, *it forces the termination* of the execution of the body; the value of <expression> is “returned” as the value of the function call

`return` is a reserved name

Functions: semantics

```
def f(x):
    body
```

A def of function introduces into the internal state a binding between the name of the function and its body

```
f(exp)
```

When a function is called:

`exp` is evaluated to a value `v`

in the internal state (*in the local frame for* `f`),
`x` (the formal parameter of f) *is bound to*
`v` (the value of the actual parameter)

`body` is executed in this state

the evaluation of any `return <exp2>` causes the termination of the function.
The value of `<exp2>` is "returned" as value of the call

Functions: *formal* and *actual* parameters

Formal parameters (or *parameters*, for short) are *names*.
Comma separated if more than one.
If there are no parameters, then the def has the form

```
def f():
    body
```

Actual parameters (or *arguments*, for short) are *expressions*.
Same number of the actual parameters.
At the moment of the call they are bound to the formal parameters, respecting their order

Functions: the evolution of the state

Functions and the internal state

Any call to a function creates a new frame in the internal state

When a function “returns” its frame is erased from the state

Frames are added and removed from the state like dishes from a stack of dishes. Hence the name for the internal state:
the *frame stack*, or the *call stack*

Formal parameters and any name occurring to the left of an assignment in the body of a function is *local* to that function:

- the binding is created inside the frame for that function
- the binding is deleted together with the frame, at return time

Example: frames on the stack

Check on pythontutor.com:

```
def perimeter(base,h):
    P=(base+h)*2
    return P
    return 100  #never executed! (this is a comment)
def z(h):
    return 0
def f(x):
    N=z(1)+1+x
    return N+1

per=0
per=perimeter(10,2)
w=f(9)
print(per,w)
```

Local names

Check on pythontutor.com:

```
A=10
def foo(x):
    return A+x
def fie(y):
    A=100
    return A+y

print(foo(10))      # prints 20
print(fie(10))      # prints 210
print(A)            # prints 10
```

Functions without return

What if execution of (the body of) a function terminates without a return?

```
def triple(x):
    x = abs(x) #pre-defined function
    print('three times x is: ', 3*x)


triple(3)
```

Functions without return

If the execution of a function reaches the end of its body, an implicit `return` is inserted by the machine

```
def triple(x):
    x = abs(x) #pre-defined function abs()
    print('three times x is: ', 3*x)
    return     #this is not necessary

triple(3)
```

Functions without return: the value None

Let's `print` the value returned by such a function:

```
def triple(x):
    x = abs(x) #pre-defined function abs()
    print('three times x is: ', 3*x)
    return     #this is not necessary

print(triple(3))   #not a very good idea
```

Functions without return: the value None

`None` is a special value (of type `NoneType`)

It is the value of those expressions for which

we are interested in the action they have on the state

because their value is irrelevant

```
return
```

is just an abbreviation of

```
return None
```

NoneType

The type of None:

- *values* : a single element
- *presented* : `None`
- *elementary operations* : no operations
- *name* : `NoneType` (it cannot be used as a converter)

None

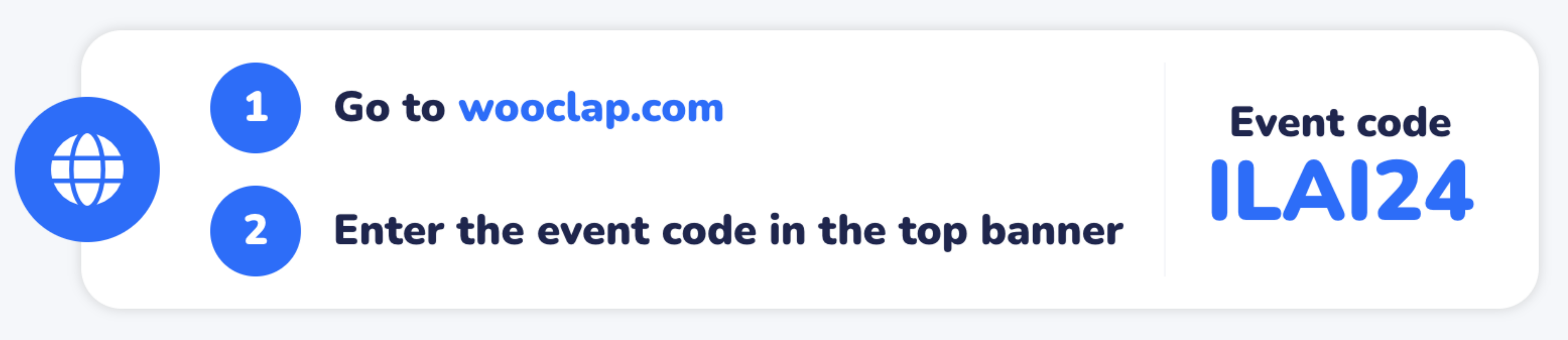
Try in Python:

```
print(print(10))
```

Be sure you understand what happens!

QUIZ TIME!

Always the same link OR qr code



1 Go to wooclap.com

2 Enter the event code in the top banner

Event code

ILAI24

Default and keyword parameters in functions

The predefined function `print` has optional parameters, to be passed *by keyword*

```
print(10, 20, sep=' *** ')     #default for end is '\n'
print(30, 40, end=' ===== ')   #default for sep is ' '
print(50, 60, end='\n\n^^\n')
print(70, 80)
```

prints

```
10 *** 20
30 40 ===== 50 60

^^
70 80
```

Default and keyword parameters in functions

Arguments

by keywords and

with *default*

can be used *with user-defined functions*

more on this in a following lesson

Predefined functions

The Python machine has many pre-defined functions

We know some: `len()`, `int()`, `float()`, `input()`, etc.
Some others: `max()`, `min()`, `abs()`, etc.

See the documentation:
https://docs.python.org/3/library/functions.html

Predefined functions: Libraries (modules)

The Python machine has many pre-defined functions

We know some: `len()`, `int()`, `float()`, `input()`, etc.
Some others: `max()`, `min()`, `abs()`, etc.

See the documentation:
https://docs.python.org/3/library/functions.html

Also a large collection of *libraries* (*modules*)
that is additional collections of functions, and,
more generally, pre-defined names

Libraries (modules)

Some available libraries

`math` : `sin()`, `cos()`, `pi`, `factorial()`, `gcd()`, etc.

`random` : `randint(a,b)`, `random()`, etc.

`string` : `punctuation`, `digits`, `ascii_uppercase`, etc.

The entire catalogue:

https://docs.python.org/3/library/

Importing *names* from a module/library

Command: `from`

```
        from <module> import <name-list>
```

Example:

```
        from math import sqrt, e
```

Importing all names from a module:

```
        from <module> import *
```

Importing *names* from a module/library

Command: `import`

```
import <module-name>
```

All fully qualified names of `<module-name>` are available

Example:

```
import math
root=math.sqrt(2)**math.e
```

or:

```
import math as m
root=m.sqrt(2)**m.e
```

Many more libraries

User-supplied libraries:

the Python package index

https://pypi.org/

In Thonny: Tools -> Manage packages

Some of these:

`Matplotlib` : visualization, graphs, plots

`NumPy` : numerical computations, arrays

`Pandas` : data analysis

. . .

Problem

Given the string ‘LODI’ we want to print its elements one by one, on a new line

```
s=‘LODI’
print(s[0])
print(s[1])
print(s[2])
print(s[3])
```

Another problem

Given a string from user input, we want to print its elements one by one, on a new line

```
s=input()
???????
```

bounded iteration (definite iteration): for

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

compound command (like `if`)

sequences: for the moment only strings

bounded iteration: for

```
s='COOP'
for c in s:
    print(c)
```

Express a repetion: in this case the command

```
print(c)
```

is executed on all elements (all characters) of the (value of) s

A linear scan

Task: print the non blank elements (i.e, `!=' '`) of a string `st`

```
st='bologna is a nice town'
for el in st:
    if el!=' ':
        print(el)
```

Example of a pervasive programming pattern:

linear scan of a sequence

For: a more complete view

Semantics:

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

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

(1) *If <name> does not exists, it is created*

(2) <name> is bound to the *first* element of the <sequence>

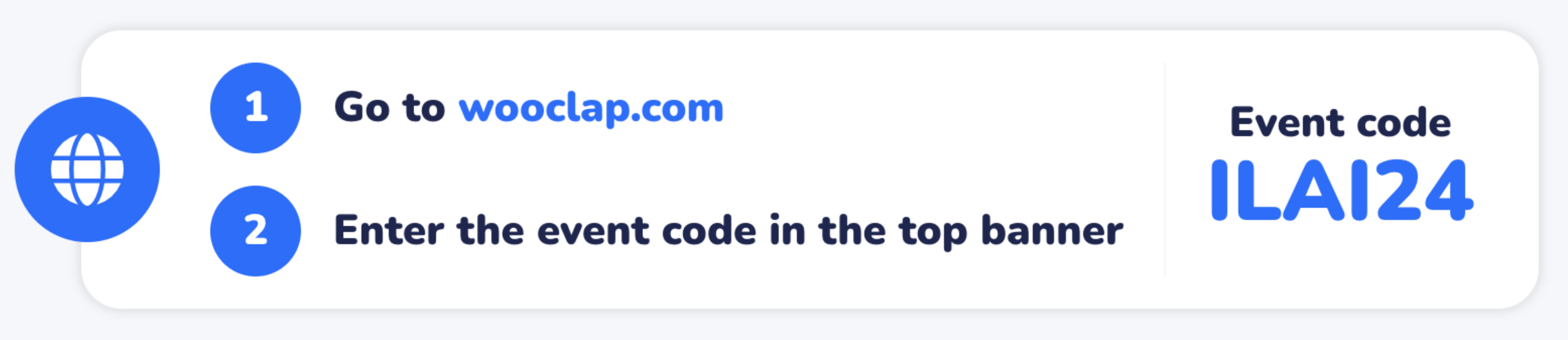
(3) Evaluate <block>

(4.1) If there is a *next* element in <sequence>, bind <name> to this element; repeat from (3)

(4.2) If there is *not* a *next* element in <sequence>, the evaluation of the command terminates (and hence the execution passes *after* the for)

QUIZ TIME!

Always the same link OR qr code



1 Go to wooclap.com

2 Enter the event code in the top banner

Event code

ILAI24

The boolean operator in

`<element> in <sequence>`

a boolean expression with value `True` iff
<element> is an element of <sequence>

Examples:

`‘p’ in ‘michael’`	has value `False`
`‘a’ in ‘aeiou’`	has value `True`
`‘A’ in ‘aeiou’`	has value `False`

Of course both <element> and <sequence> may be arbitrary expressions

Only if <sequence> is a string:

True if <element> *is a substring of <sequence>*

`‘ae’ in ‘michael’`	has value `True`

operator in

Do not confuse the two uses of the reserved name `in`:

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

is the header of an iteration

```
<element> in <sequence>
```

is a boolean expression

We cannot mix the two. Example:

```
        for s in ‘michael’ and s in ‘lodi’:  NO! WRONG!
```

It is not correct Python.

We may combine logical expressions:

```
        if s in ‘michael’ and s in ‘lodi’:
```