ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Lesson 1

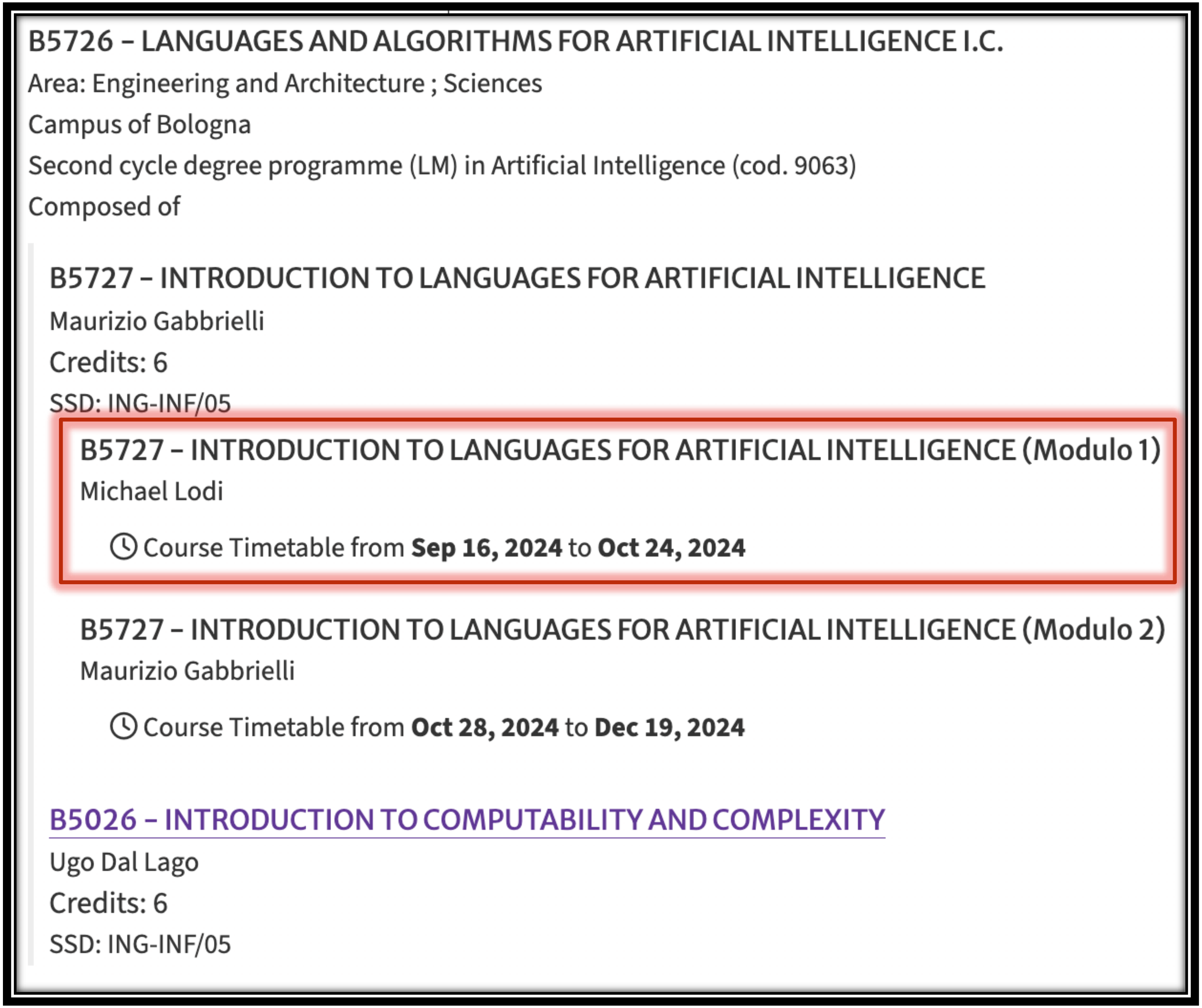
ILAI (M1) @ LAAI I.C. @ LM AI

16 September 2024

Michael Lodi

Department of Computer Science and Engineering

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



B5726 – LANGUAGES AND ALGORITHMS FOR ARTIFICIAL INTELLIGENCE I.C.

Area: Engineering and Architecture ; Sciences

Campus of Bologna

Second cycle degree programme (LM) in Artificial Intelligence (cod. 9063)

Composed of

B5727 – INTRODUCTION TO LANGUAGES FOR ARTIFICIAL INTELLIGENCE

Maurizio Gabbrielli

Credits: 6

SSD: ING-INF/05

B5727 – INTRODUCTION TO LANGUAGES FOR ARTIFICIAL INTELLIGENCE (Modulo 1)

Michael Lodi

🕓 Course Timetable from **Sep 16, 2024** to **Oct 24, 2024**

B5727 – INTRODUCTION TO LANGUAGES FOR ARTIFICIAL INTELLIGENCE (Modulo 2)

Maurizio Gabbrielli

🕓 Course Timetable from **Oct 28, 2024** to **Dec 19, 2024**

B5026 – INTRODUCTION TO COMPUTABILITY AND COMPLEXITY

Ugo Dal Lago

Credits: 6

SSD: ING-INF/05

Instructors

Michael Lodi (myself ☺)

PhD, CS (Ed) 2020

Assistant professor (junior)

Research on CS Education and CS Epistemology

I teach CS Education (to Master and PhD students
and pre-service High School teachers)

I taught Python and CS to Math bachelors

Two tutors to be formally assigned (error in the call)

Who to contact

For questions on exercises, on Python, tutoring/mentoring

1. *contact* **federico.ruggeri6@unibo.it**
   or **mohammadrez.hossein3@unibo.it**
2. *if you still have questions, contact me:* *michael.lodi@unibo.it*

For bureaucratic/administrative problems related to this module 1 only

1. *check very well the course/university websites*
2. *contact me: michael.lodi@unibo.it*

For matters related to the whole course or integrated course:

contact Prof. Gabbrielli or Prof. Dal Lago

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Most important

Ask questions,

make comments,

interrupt me!

(by raising your hand)

But also important

Pay attention (in silence)

while I talk

or I write programs

Slide format

Some material (especially: code)
will be constructed during the lecture

Slides are essentially just a memory prop for the instructor

At the end of the lecture, the slides and the code will be uploaded on virtuale.unibo.it

Recording of the lectures will be provided on Virtuale

Please register on virtuale.unibo.it (you need a @studio.unibo.it)
https://virtuale.unibo.it/course/view.php?id=66180

- ANNOUNCEMENTS
- ALL THE MATERIAL

Computing

A collection of *applications*

A *technology* which makes possible those applications

A *science* founding that technology

Computer *Science* and *Engineering*

Important linguistic aspect

Knowing a language is to know *how to use* that language

This course (= module 1)

Fundamentals elements of Python (“*the grammar*”)

basics, functions, iteration

data, objects, classes

NumPy

Fundamentals of programming in Python (“*the use*”):

little time for this ☹

Exercise on your part, at home, *is necessary*

This course (= module 1)

It is a "service" to you

- to review (or learn) some Python's features you may not know

- to review some programming skill

(but we cannot insist on this!)

If you know Python enough

- you don't need this module

However

- we will discuss the language as computer scientists

Structure

Lectures:

24 hours: mostly theory
last lecture on Oct 24

Autonomous lab/exercises:

on `virtuale.unibo.it`

- elementary tests
- suggested exercises

automatic correction, on test data, on virtuale

Tutoring “on request”: mail to
federico.ruggeri6@unibo.it
mohammadrez.hossein3@unibo.it

Platforms

https://virtuale.unibo.it

main repository and announcements

https://thonny.org

our working Python IDE (Integrated Development Environment)

http://www.pythontutor.com

for visualising the step-by-step execution of a program

You may use any Python you like

IDLE, Anaconda, Jupyter notebook, PyCharm, etc.

Textbook?

Not really…

Any material on Python

www.python.org

Textbook

Introductory level:

John V. Guttag
Introduction to Computation and Programming Using Python
(Second Edition: With Application to Understanding Data)
MIT Press, 2016

[esiste anche la traduzione italiana]

Textbook

(Very) elementary level:

Allen B. Downey
Think Python 2e.
O'Reilly Media, 2012. ISBN 978-1449330729.
On-line manuscript:
https://greenteapress.com/wp/think-python-2e/

Jessen Havill
Discovering Computer Science: Interdisciplinary Problems, Principles, and Python Programming
Chapman and Hall/CRC. ISBN 9781482254143

Textbook

Reference:

Mark Lutz
    Learning Python 5e.
    O'Reilly Media, 2013

Good reference on almost all elements of the language

Textbook

Fantastic reference on how programming languages work:

M. Gabbrielli, S. Martini

Programming Languages: Principles and Paradigms (2nd ed)

Springer, 2023

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Exam (for module 1 *only*)

Programming test, closed books (except official py&numpy docs), in lab room:

- (easy-to-medium difficulty) programming exercises
- questions on Python, e.g.:

  Complete the following program so that it prints XXX

Exams will be given on `EOL.unibo.it`

exercises are automatically checked against test data

same environment of `virtuale.unibo.it`

Outcome (of module 1): **pass/fail**

Prof. Gabbrielli and Dal Lago will give details on the whole Integrated course grades.

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Test dates (Module 1)

Early test – DATE TO BE CONFIRMED

end Oct/beginning Nov

9 Jan 2025 – 9:00 – Lab 4

29 Jan 2025 – 9:00 – Lab 4

1 in June or July

1 in September

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Timing

On Mondays:

the lecture will start at **15.0x**, please be seated and ready by that time *(as soon as the room is free from previous lecture)*

On Thursdays:

the lecture will start at **9.15**, please be seated and ready by that time

Lectures: changes

No lecture:

Sep 26
Oct 3

Additional lectures:

MAYBE 25 Sept, 9.15-11.30 (to be confirmed)

22 Oct, 9.15-12.00, room 0.5 (covering Prof Chesani, confirmed)

Other changes may be announced on Virtuale

see the always updated calendar on
https://corsi.unibo.it/2cycle/artificial-intelligence/timetable

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Let’s get to know each other

https://forms.office.com/e/bhFauqn8ZC

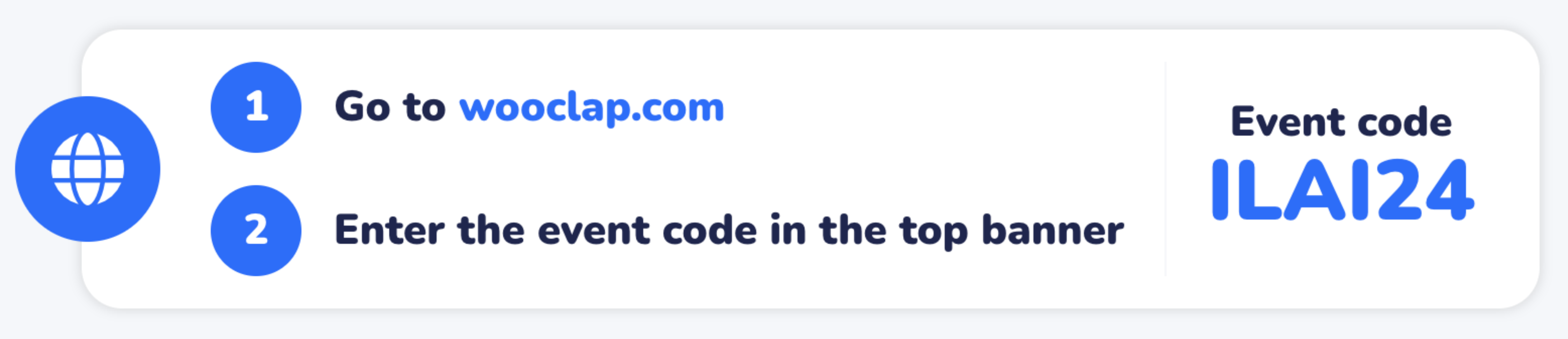
(link also on Virtuale)

Let's get to know each other (ILAI (M1) @ LAAI I.C. @ LM AI 2024)

We will use ANONYMOUS quizzes.

Always the same link OR qr code

Let’s try it now ☺



1 Go to wooclap.com

2 Enter the event code in the top banner

Event code

ILAI24

Most important

Ask questions,

make comments,

interrupt me!

(by raising your hand)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Computations, Machines, Languages

Computation is a combinatorial manipulation of symbols from a finite alphabet

Manipulation is done through simple, combinatorial, effective elementary operations

Computations and machines

Computations are *performed*
by machines

Computations use *elementary operations*
which operations depends on the machine

Computations are *described*
in a language: natural or artificial
programming language

Each PL has its own *abstract machine*

Computations and machines

Computations are *performed*

We call *machine* the agent performing the computation

The machine for a computation must be able to perform the *elementary operations* that the computation is built on

Computations and machines

Computations are *performed*

We call *machine* the agent performing the computation

The machine for a computation must be able to perform the *elementary operations* that the computation is built on

To specify a computation we must specify the elementary operations *and* the sequence of those operations that should be applied

So we have a *description* of the computation. This description is done is a certain language. And the machine should be able to ”understand” (better: execute) that language

Machines and languages

We have languages describing computations, as sequences of elementary operations to be applied on some data

We have machines able to execute those descriptions

Let suppose that we have a language for computations, L

We *call : abstract machine for L any agent able to execute the computations described in L (the “programs” written in L)*

Machines and languages

We *call abstract machine for L*, any agent able to execute the computations described in L

There are machines that are able to compute any computable function.

These are the machines whose languages are *general purpose programming languages*.

One of these machines is the Python machine, which we will describe in this course

Python

Some remarks for those who already know a programming language

Python: high level language

Flexible and large set of data types

No direct connection between Python data and machine representation

Higher-order: functions are first-class citizens,
hence it may used as a functional programming language
(Any value is first-class.)

Easy to learn (at least the basics)

Python: it is meant to be interpreted

A Python machine provides an interactive interpreter, which

- evaluates a Python command
- shows its result

Programs as simple as single expressions:

```
523
```

is a legal Python complete expression
which evaluates to the integer 523

Programs (scripts) are lines of text,
each line being a legal Python command

Simple compiler produces VM bytecode, which is then interpreted

Python: dynamic types

Any value (object) comes with its type,
which is maintained and checked at run-time

No type associated to names,
no declaration of names

No static check on type compatibility

support for type annotations, un-checked by the run-time

Dynamic data structures:

sequences (e.g lists) grow and shrink at run-time
no standard («fixed size») arrays

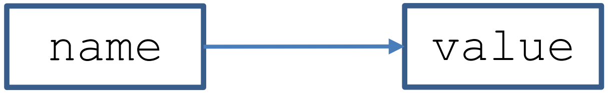
… we will use NumPy at the end of the course

Data is garbage collected, like in Java (no malloc, no free())
Simple reference count algorithm

Python: reference model for names

State of the machine

a list of associations:



name

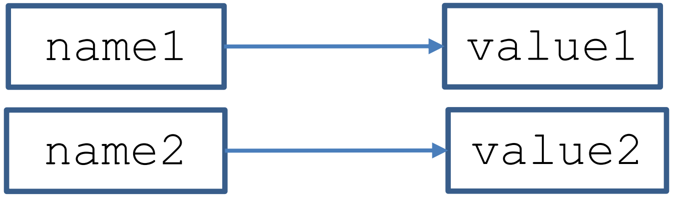
value

A "variable" (name) is a "pointer to its associated value rather than a named container for that value

Like Java's names for objects

Python: reference model for names

In particular, starting from



name1

value1

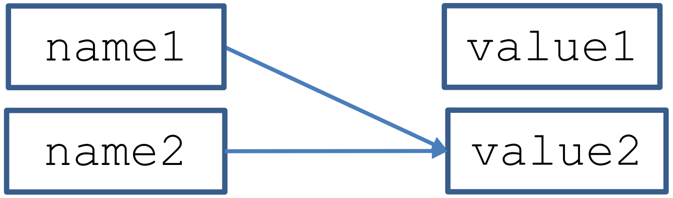
name2

value2

if we assign

```
name1=name2
```

we will be left into the state



name1

value1

name2

value2

if `value2` is modifiable, this makes side-effects possible

Python: object-oriented

everything is an object

multiple inheritance

Classes are present (and they are objects, too)

but they are less normative than in Java

e.g.: we may add attributes to single instances

Python: control on the system

The programmer may change almost any aspect of the system

Environments are available and modifiable

Extended "reflection" mechanisms

Large set of libraries

"We are all consenting adults"

The Python machine

It is an agent able to execute any description of a computation written in the Python programming language

1 Elementary operations

2 How these operations could be ordered

3 How this ordering (a *program*) should be described

Elementary ops act on values (or *objects*)

Values are organized in *data types* which also fix the elementary operations

Data types

Data and elementary operations are organised in *(data) types*

A type is given by

- the collection of the *values* of that type
- how those values are *presented*
- the collection of *elementary operations* on those values
- the *name* of the type

Data types: int

The type of integer values:

- *values* : the signed integers *from math*
- *presented* : like in math
- *elementary operations* : addition (`+`), subtraction (`-`), multiplication (`*`), integer division (`//`), integer modulo (`%`), exponentiation (`**`)
- the *name* of the type : `int`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Expressions

We may group several operations in an *expression*

Expressions are *evaluated*

- *from left to right*
- *respecting the precedence rules of math*
- *parentheses may be used to force grouping*

```
>>>  (2+3)*3**8//2
16402
```

Precedences

precedences are the same as in standard math

they can be changed by using parenthesis

computation is always performed left to right, when possible

In order of decreasing precedence:

- parenthesis
- **
- + - (unary)
- * // %
- + - (binary)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Data types: str

The type of the strings:

- *values* : finite *sequences* of characters (from a fixed alphabet)
- *presented* : enclosed in *quotes* (`'`), or (`"`) or (`'''`) or even (`"""`)
- *elementary operations* : concatenation (`+`), repetition (`*`),
  length (`len(…)`), selection (`[…]`)
- *name* : `str`

Selection on strings

Strings are *sequences* of characters, that is
are correspondences between *indexes* and characters
indexes start at *zero* (hence, e.g. the third element is at index 2)

the second element of `'pine'` is `'i'` :

```
>>> 'pine'[1]
'i'
```

Selection on strings

General form

string [*index*]

where

string is any expression evaluating to a `str`

index is any expression evaluating to an `int`

```
>>> ('pine' + 'apple')[4*3//2+1]
'l'
```

Length

The *length* of a string is the *number of its characters*:

```
>>> len('pine')
4
```

(quotes are *not* part of the string: they are just for representation)

The last element of a string `S` is at index `len(S)-1`

```
>>> 'pineapple'[len('pineapple')-1]
'e'
```

Shorthand: *negative indexes are counted from the end*

```
>>> 'pineapple'[-1]
'e'
```

Selection, again

Selecting *"after"* the length is an error

```
>>> 'pine'[10]
IndexError: string index out of range
```

The string with no characters:

```
>>> len('')
0
>>> 'bologna' + ''
'bologna'
```

Overloading

Same symbols are used for different operations

```
>>> 3 + 4
7
>>> '3' + '4'
'34'
```

We say that the symbol (`+`, in this case) is *overloaded*

The machine disambiguates depending on the context (depending on the *types of the arguments*)

Expressions

Expressions are sequences of operations on values, using operations and parenthesis

```
len(str(123))+2**(len('BO')+1)
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Data types: float

The type of rational values:

- *values* : a *subset* of the rational number
- *presented* : like in math; also exponentional notation
  `3.1415, -1.2, 3.4567e3, 3456.7`
- *elementary operations* : addition (`+`), subtraction (`-`),
  multiplication (`*`), division (`/`), exponentiation (`**`)
- the *name* of the type : `float`

Data types: float

- *values* : a *subset* of the rational number

approximations

```
>>> 0.1+0.2
0.30000000000000004
```

limited interval of representation

```
>>> 3.**1000
OverflowError: (34, 'Result too large')
>>> 3**1000
13220708194808066368904552597521443659654220327521
481676649203682268285973467048995407783138506...
```

Converting types

The names of the types can be used as "type casts"

```
>>> float(3)
3.0
>>> int(3.54)
3
>>> str(3.14)
'3.14'
>>> int('123')
123
>>> int('bologna')
ValueError: invalid literal for int()
```

Mixed expressions

The Python machine will apply the necessary transformations of types so that the expression makes sense

E.g.

`7/3` is equivalent to `float(7)/float(3)`

Mixing `int` and `float` is ok

```
>>> 7.0/3
2.3333333333333335
>>> 2**(0.5)
1.4142135623730951
```

expressions

One possible legal phrases of the language:

it expresses a computation for obtaining a *value*

we are interested in that value

It may use values and operations of different *compatible* types

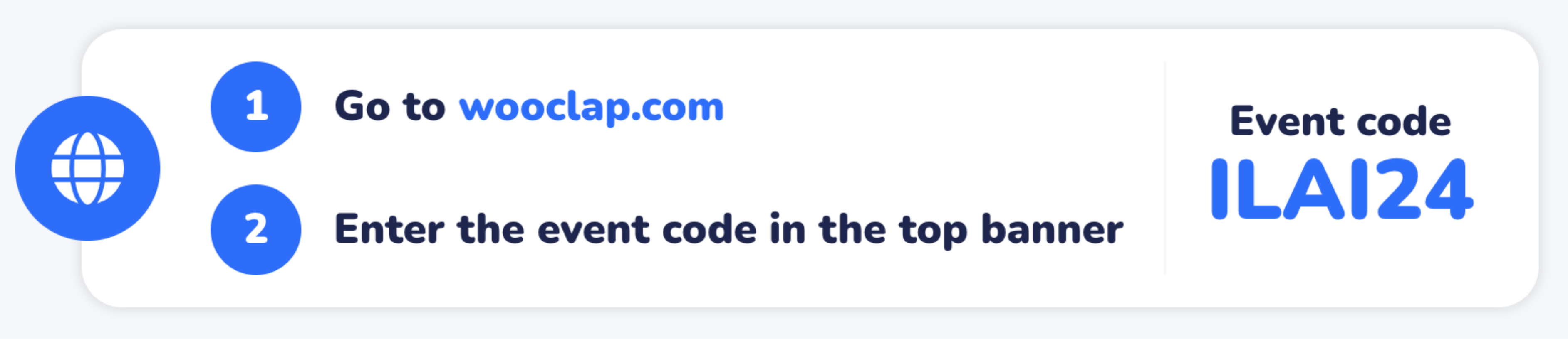
When evaluated in the shell, the shell prints its value

When evaluated in a script, its value is lost, unless explicitly used

In a Jupyter notebook, the value of the last expression is printed

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

Names and the assignment

Assignment

<name> = <expression>

A <name> is a (finite) sequence of letters, digits, and _, which does not start with a digit

The assignment is that elementary statement of the language which associates a name with a value

Semantics of the assignment

<name> = <expression>

1. Evaluate <expression> , determining a value V
2. Check if <name> is already present

    2.1 If not present, create it

3. Bind V to name

The Python machine maintains an internal state

Names and the assignment

Sometimes is could be useful to give a *name* to a value

We do this via an *assignment*

name = value

What is a name?

A name is a sequence of:
- letters (upper and lower case)
- digits
- the caracter _

it must begin with a letter

Effect of the assignment   name=expression
- Python checks if the name has already been used
- Python evaluates the expression to the right of =, and obtains a value V
- Python *binds* (associates) V to the name, in its *internal state*

Evaluation of a name: the association (binding) for that name is used and the associated value is used as value for the name

Internal state

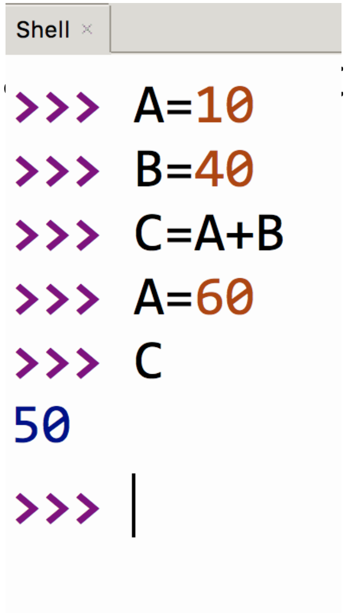
The Python machine maintains a list of *associations (bindings)* between names and values

This list is called the *internal state* of the machine

We assume for the moment that when we start a Python machine, its internal state is *empty*

Then any assignment on a *new* name will add he internal state

An assignment on an *existing name* changes the binding for that name only



```
>>> A=10
>>> B=40
>>> C=A+B
>>> A=60
>>> C
50
>>> |
```

Evolution of the internal state: example

Evolution of the internal state

The internal state evolves as an ordered list:
new names are added at the end of the list

Assignments to already used names modify the binding for those names *only*

Keeping track of the evolution of the internal state is an important part of the programming task

=====

Types are an attribute of values, not of names (the type of a value bound to a name may change freely)

===

We may inspect the type of a value by using the operation (function) type(…)

Expressions vs Commands

An *expression* is a phrase in the language of which we are interested in its value

```
12+3
len(‘bologna’)+1
```

A *command* is a phrase in the language which we use for its modification of the (internal) state

```
A=10
```

Programs

Little use of Python as an extended calculator

We may write *programs* (also: *scripts*)

A program is *a plain text*

Any line of this text is a single Python phrase (command, expression)

The Python machine may *run* (*evaluate, execute*) a program:

evaluate the program line by line, starting with line 1,
until the end of the text is reached

Run a program

The external state: print and input

Print(…) is a command

It transfers the value of its argument(s) to the *external state*

input() is an expression that is used to transfer values from the *external to the internal state. Input() always gives values of type str*

Transferring values from the internal to the external state

The command

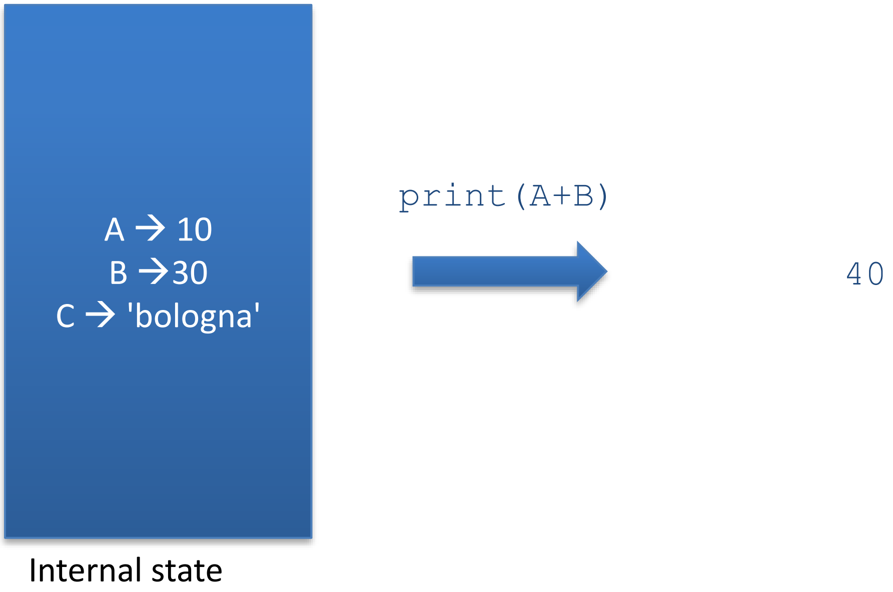
```
print(expression)
```

evaluates *expression* and

shows the resulting value in the *external state* (the *shell*)

```
A=10
B=A+20
C='bologna'
print(A+B)
```

Transferring values from the internal to the external state



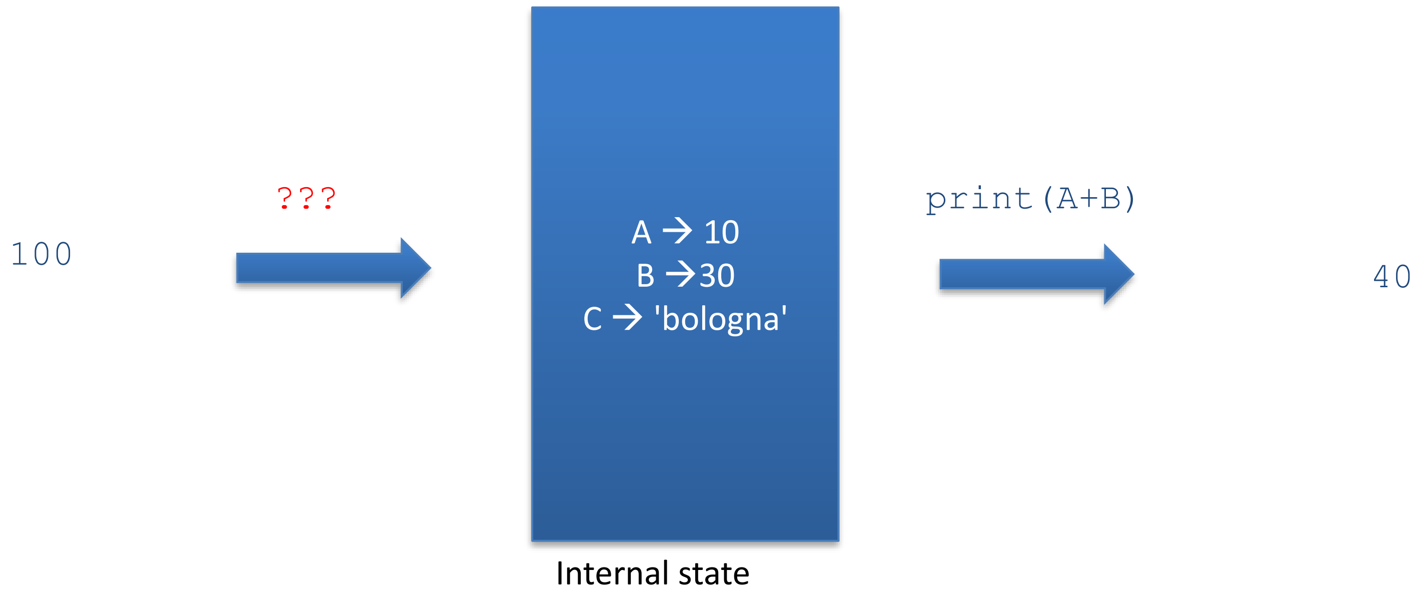
A → 10
B →30
C → 'bologna'

Internal state

```
print(A+B)
```

```
40
```

Transferring values from the external to the internal state



100

???

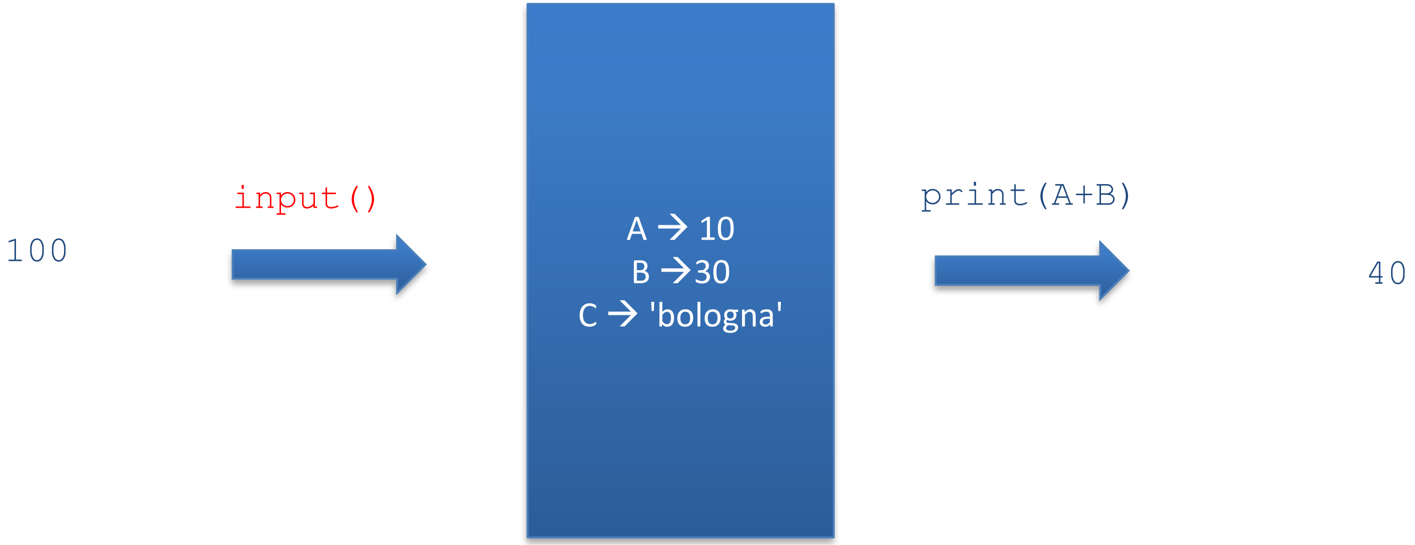
A → 10
B → 30
C → 'bologna'

print(A+B)

40

Internal state

Transferring values from the external to the internal state: input()



100

`input()`

A → 10
B → 30
C → 'bologna'

`print(A+B)`

40

Internal state

The input expression

```
input()
```

is an *expression* which transfers a *string* from the shell into the internal state

```
A=10
B=A+20
C='bologna'
D=input()
print(D)
```

The input expression

```
input()
```

is an *expression* which transfers a *string* from the shell into the internal state

If we want a *number* we must explicitly convert it:

```
D=int(input())
print(D+10)
```

Try also: `D = int(input("give me a number: "))`

Swapping two values

For swapping the bindings (values) of two names (using only the part of the language that we know so far) we must use a *third* name:

```
A=10
B=200

tmp=B
B=A
A=tmp
```

Visualize the effect of this code on the internal state!

Strings are immutable

One may be tempted to write

```
S='bologna'
S[0]='c'
```

This is an error: we cannot modify a value of type str! (Like we cannot modify a value of type int or float)

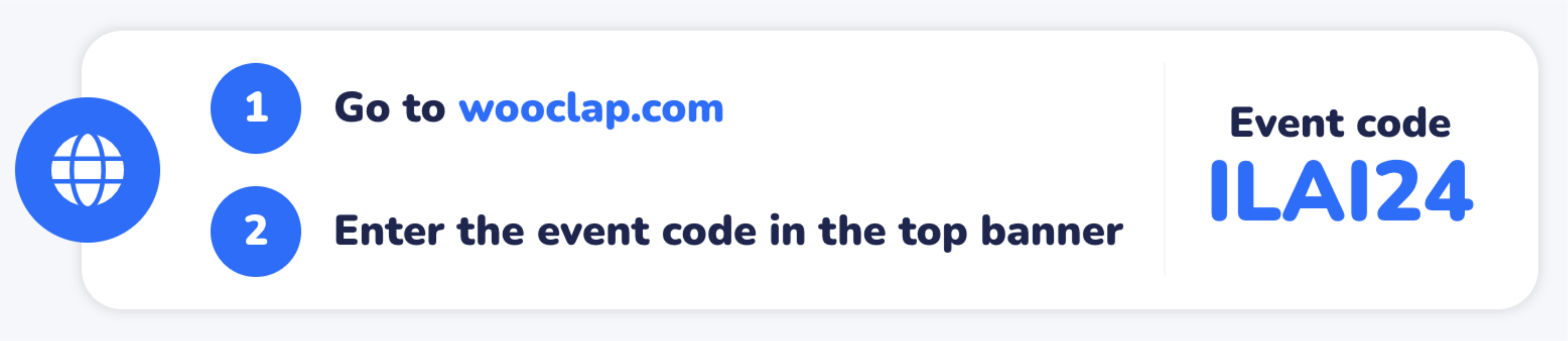
Of course we may modify the binding of a name:

```
S='cologna'
```

but the two values `'bologna'` and `'cologna'` exist independently and distinct

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

Conditional command

It is a *compound* command:

it is composed (also) from other commands

Its use: altering the sequential order of execution of a program, according to some *conditions*

Conditional command: first form

`if <`*condition*`>:`	guard
`    <t-block>`	then branch
`else:`	
`    <e-block>`	else branch

Syntax:

`if` and `else` are *reserved names*

A reserved name (or reserved word) has the structure of a standard name but *cannot be used as a name* because of its special role inside the language

Conditional command: first form

```
if <condition>:                         guard
      <t-block>                         then branch
else:
      <e-block>                         else branch
```

semantics:

We evaluate the guard. It the guard is true, we execute the “then” branch; if the guard is false, we execute the ”else” branch

After the then/else branch (only one of those!) the execution proceeds from what follows the conditional command

Conditions: boolean values

`if` <*condition*>`:` guard

What is this <*condition*> `?`

Syntactically:

any expression of type `bool`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Data types: bool

The type of the truth values:

- *values* : only two, *true* and *false*
- *presented* : True, False
- *elementary operations* : and (logical conjunction), or (logical disjunction), not (negation). They are defined by the usual truth tables
- *name* : bool

- Exp1 `and` Exp2: `True` iff both Exp1 and Exp2 are True
- Exp1 `or` Exp2: `False` iff bothExp1 and Exp2 are False

Logical operators

`not` (negation):

transforms `True` into `False` and `False` into `True`

`not True` has value `False`

`not False` has value `True`

`and` (conjunction: et):

<exp1> `and` *<exp2>* has value `True` iff
both *<exp1>* and *<exp2>* have value `True`

`or` (disjunction: vel):

<exp1> `or` *<exp2>* has value `False` iff
both *<exp1> and <exp2>* have value `False`

Comparison operators: producing bool values

less than:	<
less than or equal :	<=
greater than:	>
greater than or equal:	>=
Equal:	==
Non equal:	!=

Lazy evaluation of logical operators

Python expressions are

completely evaluated from left to right:

```
7*0*(int(input()))
```

the `input()`

is always performed, even if the result (0) is already known

Lazy evaluation of logical operators

Python expressions are
completely evaluated from left to right:

```
7*0*(int(input()))
```

the `input()`
is always performed, even if the result (0) is already known

Boolean expressions are an exception to this

lazy evaluation of Booleans: as soon as we find a value that allows to know the final result, the evaluation terminates

```
(0==0) and (0==1) and (0==int(input()))
```

not evaluated

Nested if

Write a program which input an int x and an int y and prints:

- 0 if x is 0
- y/x if y is negative
- -y/x if y is positive
- 100 if y is zero

nested “if”s

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

nested if

0 if x is 0

y/x if y negative

-y/x if y positive

100 if y is 0

```
x = int(input("Provide x: "))
y = int(input("Provide y: "))
if x == 0:
    print(0)
else:
    if y < 0:
        print(y/x)
    else:
        if y > 0:
            print(-y/x)
        else: #y is 0
            print(100)
```

nested if

0 if x is 0
y/x if y negative
-y/x if y positive
100 if y is 0

```
x = int(input("Provide x: "))
y = int(input("Provide y: "))
if x == 0:
    print(0)
else:
    if y < 0:
        print(y/x)
    else:
        if y > 0:
            print(-y/x)
        else: #y is 0
            print(100)
```

```
x = int(input("Provide x: "))
y = int(input("Provide y: "))
if x == 0:
    print(0)
elif y < 0:
    print(y/x)
elif y > 0:
    print(-y/x)
else: #y is 0
    print(100)
```

General form of the conditional

```
if <expr1 bool>:
    <block1>
elif <expr2 bool>:                    optional
    <block2>
…
elif <exprk bool>:                    optional
    <block>
else:                                 optional
   <block-e>
```

Blocks

Block:

- a sequence of lines
- each line with a single command
- 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`)

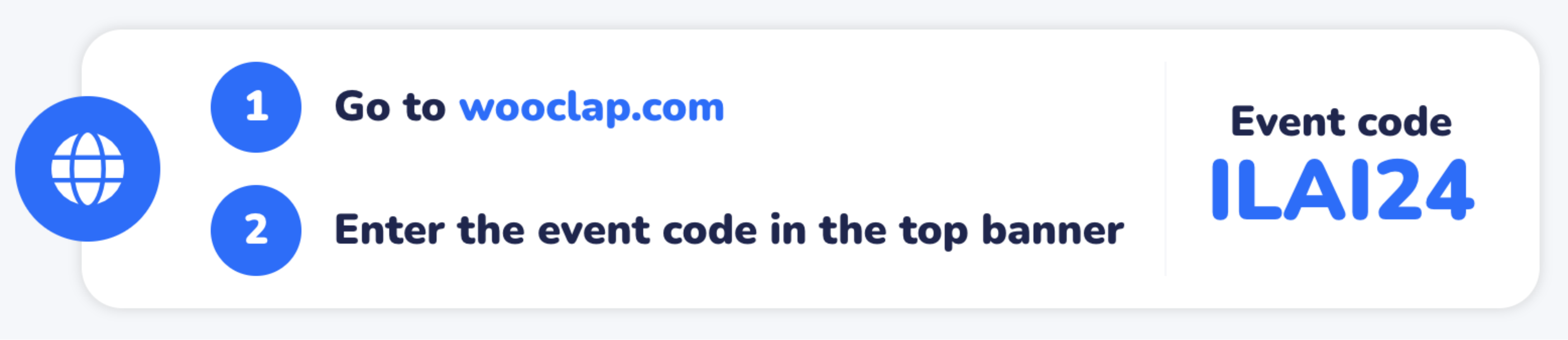
Other programming languages use parenthesis `{,}`.

Python uses indentation

No local scopes, unless for functions or classes

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

Review: blocks. example

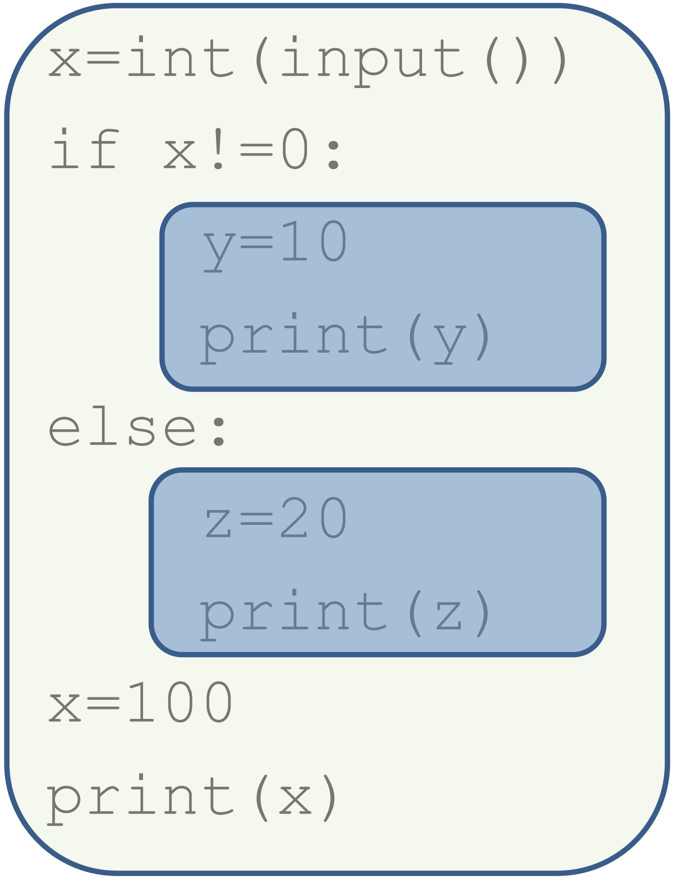
How many blocks?

What does it print on input 20?

```
x=int(input())
if x!=0:
    y=10
    print(y)
else:
    z=20
    print(z)
x=100
print(x)
```

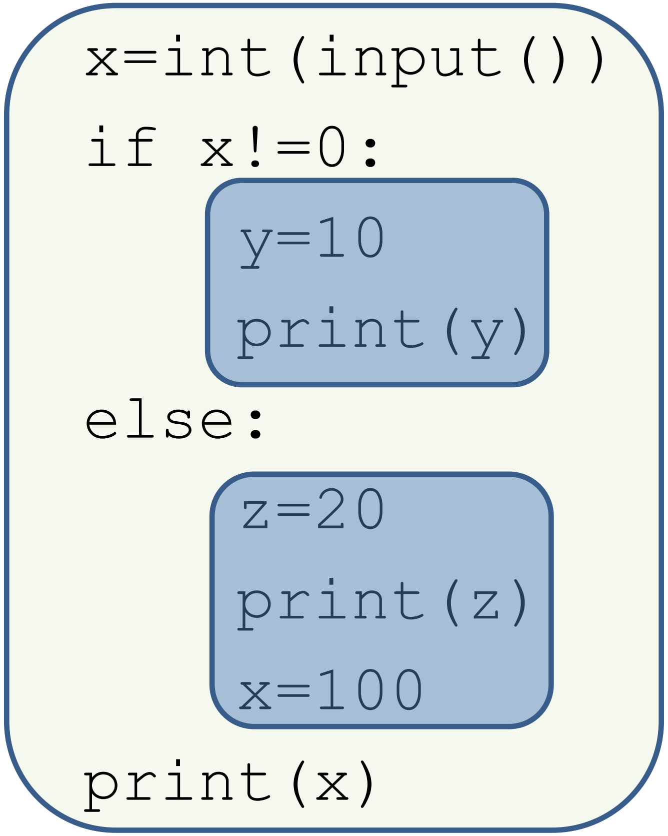
Review: blocks. example

How many blocks?



```
x=int(input())
if x!=0:
    y=10
    print(y)
else:
    z=20
    print(z)
x=100
print(x)
```

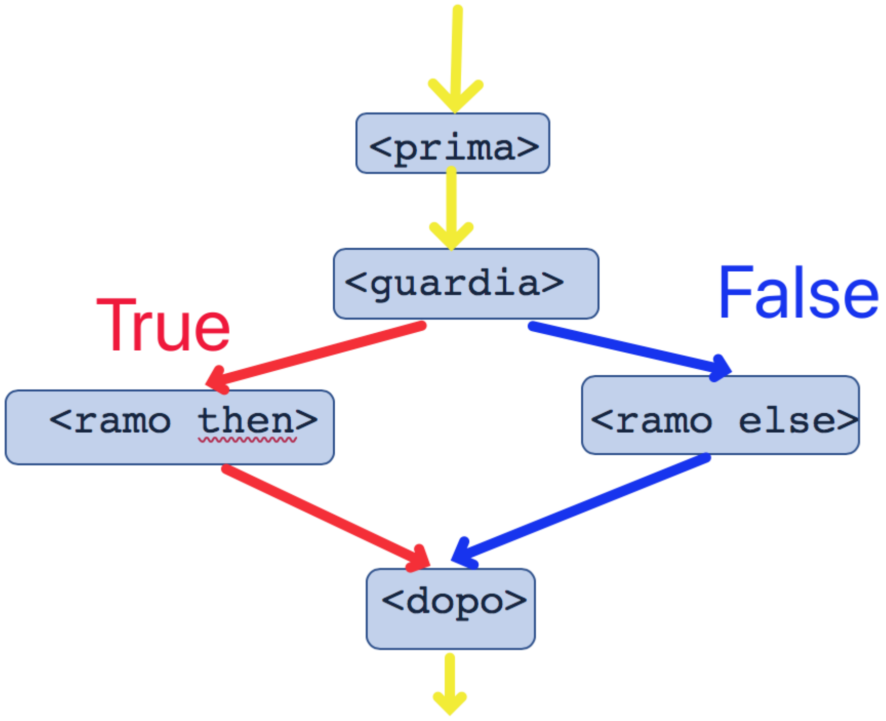
Review: blocks. another example
What does it print on input 20? And on input 0?



```
x=int(input())
if x!=0:
    y=10
    print(y)
else:
    z=20
    print(z)
    x=100
print(x)
```

Review: conditional command

```
<before>
if <guard>:
    <then branch>
else:
    <else branch>
<after>
```



<prima>

<guardia>

True

False

<ramo then>

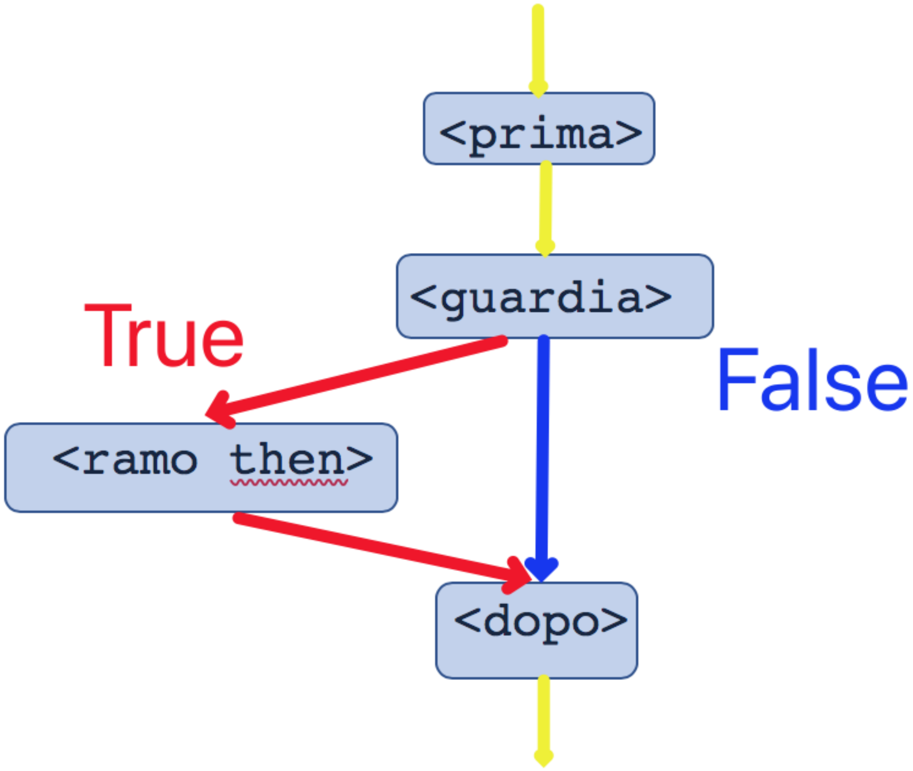
<ramo else>

<dopo>

Review: conditional command

else is optional

```
<prima>
if <guardia>:
    <ramo then>
<dopo>
```



Review: conditional command

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