ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Lesson 4

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

25 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 3

Object: value enveloped in an identity. Different objects (identities) may have the same value
Identity accessible with `id()`. Identities compared with `is`. NB: `==` compares values, instead

Objects with same type have same methods. Methods are invoked on objects with the dot `.`

`tuple` is a compound/structured type (group of values each of its type). Immutable sequence. Same operations of other imutable sequences

Generalized assignment: <sequence of # names> = <sequence with the same # of elements>

Slice: a «window» on a sequence. `[b:e:p]` between `b` included and `e` excluded, with step `p` (default: 1. Can be negative to select backward)

Operations create new objects. On the contrary, assignments can create aliasing situations

`range`: a special sequence (not expanded). Indexes work like in slices. `for i in range()`… to iterate over numbers – usually indexes of a sequence

`while`: unbound iteration. I need to take care of initializing and modifying the guard.

Next lectures for ILAI

Monday, 30 September 15:00 - 18:00 (Regular, room 0.5)

Thursday 3 Oct: NO LECTURE (prof. Lodi busy)

Monday 7 Oct: NO LECTURE (cancelled in all ING area, I believe, on 7-8-9)

Thursday, 10 October 09:00 - 11:30 (Regular, room 2.8)

ALL UPDATES ALREADY
ON THE COURSE WEBSITE

Use this time for trying exercises on Virtuale ;-)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

A new data type:

lists

Lists: `list`

- structured type (like tuple)

- *mutable* sequences of objects

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Lists

Values: finite sequences of values, mutable

Presentation: [<object1>, … , <objectk>]

[] is the empty list

a list with a single element: [<element>]

(or also [<element>,])

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

but also…

Mutable

Contrary to all types we have seen so far:

we may modify the *value* of an object preserving its identity

Example:

```
L=[10,20,30]
L[1] = 200
```

The object bound to L did not change

Its value changed

Warning: LHS of = is *not* a name

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Assignment, again

The simplest form (from L1):

<name> = <expression>

Now:

<selection> = <expression>

<selection> must be on a mutable object (and cannot be a name)

Plus the extended versions we saw in L3:

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

A,B = B,A

L[i+1],L[i]=L[i],L[i+1]

Assignment, again

Now:

<selection> = <expression>

<selection> must be on a mutable object (and cannot be a name)

Semantics:

- evaluate *<expression>* and obtain an object *o*
- evaluate <selection> and obtain a component *p* of a modifiable object *m*
- the component *p* of *m* is modified in *o*

Observe: there is *no* modification of the bindings between names and values in the internal state

Function that takes a list of int and modifies every 0 into 100

State evolution

```
L = [1, 2]
V = [10, 20]
W = L + V
L = L + [3]
L[0] = 100
Z = ['s', V, 10]
Z[1][0] = 0
print(V)
```

State evolution

```
L = [1, 2]
V = [10, L, 20]
W = [30, L, 40]
printV)
print(W)

LL = L[:]
VV = V[:]
z = …
```

Example: double the value of elements

Given a list L of int, modify any element, transforming it in its double

=> file double.py

given L=[10,20,30,40],

transform L in [20,40,60,80]

Aliasing and side effects

```
L = [10,20,30]
V = L
L[0]=100
print(V)
```

Since there is aliasing (V and L are alias for the same object),
V has been modified by *side effect*

We also say:
`L[0]=100`
has the side effect to modify `V`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Aliasing and side effects, 2

We may have aliasing also on tuples (or int, or float etc.):

```
T = (10,20,30)
R = (100, T, 300)
```

the “same” on lists:

```
L = [10,20,30]
W = [100, L, 300]
```

On list we may modify L, and this will be seen also from W

Aliasing and side effects

Small examples on

aliasing_example.py

Aliasing and side effects, moral

Aliasing and side effects are present in the language *by design*

They are important to do meaningful things!

Yet, they are a delicate, subtle ingredient of the language

- to be understood
- to be controlled
- because they are source of errors which are difficult to locate and discover

Deleting elements from a list: `del`

Command `del`:

```
del L[i]
```

Semantics:

L[i] is eliminated from L

the elements "following" L[i] are "shifted to the left"

(indexes are recomputed)

Given list L of int, delete the zeros

```
L = [1,2,0,0,3,0,5,0]
```

Given list L of int, delete the zeros

```
L = [1,2,0,0,3,0,5,0]
i=0
while i<len(L):
    if L[i]==0:
        del L[i]
    else:
        i=i+1
# here L is [1,2,3,5]
```

Observe that the index is incremented *only* when `del` is *not executed* (file removezero.py)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

It does not work with for!

```
L = [1,2,0,3,0,5,0]
i=0
while i<len(L):
    if L[i]==0:
        del L[i]
    else:
        i=i+1
# here L is [1,2,3,5]
```

```
L = [1,2,0,3,0,5,0]
for i in range(len(L)):
    if L[i]==0:
        del L[i]
# here L is ??
```

WRONG: IT DOES NOT WORK!

Don’t use for on lists in presence of side effects!

Never (never !)

use `for` on lists,

if their length

is modified in the `for` *body*

Don’t use for on lists in presence of side effects!

The heading of the `for` *command freezes the sequence object (its identity)*

not its value

`del`

In the "modifying assignment"

<selection> = <expression>

<selection> *must be an explicit selection*

In

`del` <selection>

<selection> *must be an explicit selection*

Compare:

```
L=[10,20,30]
del L[1]
```

```
L=[10,20,30]
e=L[1]
del e
```

Beware: `del` *`<name>`* will remove *`<name>`* from the internal state!

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Add/delete elements: methods

Invoking a method on an object of type list

may *modify the object*

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Some methods on lst

L.append(x) L[len(L):]=[x]

append x as last element of L; returns None

L.insert(i,x) L[i:i]=[x]

insert x in L at index i (shifting indexes),
or append if i>=len(L); returns None

L.extend(V) L[len(L):]=V

extends L with list V (in place concatenation)

L.clear() del L[:]

empties L

➔➔➔ all of them return **None**

Add/delete elements: slice

<selection> = <expression>

<selection> may be a *slice*

If <selection> is a slice:

- *RHS must be a list, whose elements are inserted for the slice*
- *as a result of the assignment, the list at LHS may grow, or shrink*

Add/delete elements: slice

Selection may be a *slice. Examples:*

`L=[1,2,3]`	1	2	3			
`L[0:2]=[10,20]`	10	20	3			
`L[0:1]=[5,6]`	5	6	20	3		
`L[-1:]=[4]`	5	6	20	4		
`L[1:1]=[1,2]`	5	1	2	6	20	4
`L[0:2]=[]`	2	6	20	4		
`L[0:0]=[1]`	1	2	6	20	4	
`L[len(L):]=[30]`	1	2	6	20	4	30

Warnings

1)

del L[i:j]   is equivalent to   L[i:j]=[]

2)

L[1:2]=[4,5,6] and L[1]=[4,5,6]

are two (very!) different things

- L[1:2]=[4,5,6] *extends* L

- L[1]=[4,5,6] leaves unchanged the length of L,: the element L[1] is replaced by the (list) object [1,2,3]

Warnings

Semantically equivalent to slicing

- In general, using methods is more efficent

Careful

```
L=L+[elemento]
```

is not the same as

```
L.append(elemento)
```

In the first, a new object is created
In the second, no new object

Then…

Don't use for on lists in presence of side effects!

The heading of the `for` command freezes the sequence object (its identity) *not its value*

Compare:

```
L=[10]
for e in L:
    L.append(e)
print(L)
```

```
L=[10]
for e in L:
    L=L+[e]
print(L)
```

Warning!

The heading of the `for` command freezes the sequence object (its identity)

not its value

Compare:

```
L=[10]
for e in L:
    L.append(e)
print(L)
```

```
L=[10]
for e in L:
    L=L+[e]
print(L)
```

The first is an infinite loop, the second prints [10,10]

Careful

```
L=L+[elemento]
```

is not the same as

```
L.append(elemento)
```

Suppose there is aliasing on the object bound to L:

```
L=[10,20,30]
W=L
L.append(40)
print(W)
L=L+[60]
print(W)
print(L)
```

One example with methods

Read from the keyboard a series of positive integers, until the user inserts 9999
Store the numbers in a list

file inser.py

assignment: one more variant

augmented assignment

In iteration scans one often uses:

```
res = res + val
```

or also

```
count = count*val
```

We may use *contracted forms*

assignment: one more variant

augmented assignment

In iteration scans one often uses:

```
res = res + val
```

or also

```
count = count*val
```

We may use *contracted forms*

```
res += val

count *= val
```

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Augmented assignments

+= -= *= /= //= %= **=

and even others

Operations are overloaded:

T += v

is addition, or concatenation, etc.

depending from the type of T

Augmented assignments

`x+=v` `x=x+v`

Similar, *but not the same*, to its "normal" analogue

1. the operation is done *in place* if possible
2. x *is evaluated only once*
3. *LHS is evaluated first*

Let’s dig deeper

Augmented assignments

Similar, but *not the same* as their "normal" analogues

1. The operation is done *in place* if possible

```
L = [1,2,3]
L +=   [4]
```

```
V = [1,2,3]
V = V + [4]
```

are two *different things:*

+= is extend (append) on lists!

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Augmented assignments

Similar, but *not the same* as their "normal" analogues

2. x *is evaluated only once*

Let f(x) be a function

```
L[f(i)] +=  val

L[f(i)] = L[f(i)] + val
```

+= calls f(i) only once

➔ if f() has side effects, they happen once only!

Augmented assignments

Similar, but *not the same* as their "normal" analogues

3. LHS is evaluated first, then RHS, then the binding in the state is modified

Let f(x) be a function

```
L[f(i)] +=  exp
```

If f(i) has side effects, they happen before the evaluation of exp

(file augmented.py)

More methods on list

L.remove(x)

remove from L the first element L[i] equal (==) to x
error if `x not in L`
returns `None`

L.copy() L[:]

returns a (shallow) copy of L

L.reverse()

reverses L (in place)
returns None
What is the difference with L=L[::-1] ??
remember the mantra: "operations create new objects"

L.sort()

sort L (in place)
compare with the builtin function M = sorted(L)
remember the mantra: "operations create new objects"

More methods on list

L.pop(i)

returns and removes the element at index i

pop() returns and removes L[-1]

returns a value: L[i]
modify in place the list, removing the element at index i
leave unchanged the identity of L

Double (nested) iteration

Sometimes we need to iterate *inside* another iteration.
This programming schema is called double (or nested) iteration

Trivial example: Given a tuple of tuples of integers, count the zeros

```
def zeros(ToT):
    count=0
    for t in ToT:
        for e in t:
            if e==0:
                count += 1
    return count
```

file tuples.py

Individual exercises at the end of the slides

Can functions modify global mutable objects?

Can functions modify global objects?

Observe that with *immutable* objects, this is impossible:

```
b=20
def f(a):
    a=10
    b=100
    return None
f(b)
print(b)
```

a,b in the body of f are local names!

Intermezzo: global (more in next lecture)

Observe that with *immutable* objects, this is impossible,

unless a global declaration is used

```
b=20
def f(a):
    global b
    a=10
    b=100
    return None
f(b)
print(b)
```

a is local; b is the one in the main frame

Can functions modify global mutable objects?

With mutable objects: possibile, in several ways

1. *Through a mutable actual parameter:*

```
b=[20]
def f(a):
    a[0]=100
    # return None
f(b)
print(b)
```

In Pythontutor…

Of course the assignment `a[0]=100` would results in an error when f is called on an immutable actual parameter

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Can functions modify global mutable objects?

With mutable objects: possibile, in several ways

2. *Through a global name, bound to a mutable object*

```
b=[20]
def f():
    b[0]=100
f()
print(b)
```

In Pythontutor…

Observe: at the left of = there is a *selection* not a name

Hence the rule: "name at the left of = is local" cannot be applied

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Some exercises and challanges

For home 1: ordered insertion

Write a function

```
def ins_ord(L,el):
   '''L: ordered list, in increasing order
insert el in L, in place'''
```

Example:

S=[2,4,6,8]

ins_ord(S,5) must modify S in [2,4,5,6,8]

Some solutions in file inserimento-in-lista-ord.py

For home 2: Floyd triangles

A *Floyd triangle* of order 5:

five rows, with the integers from 1; any row has one more element of the previous row

```
 1
 2  3
 4  5  6
 7  8  9 10
11 12 13 14 15
```

Write a function Floyd(n) which prints a Floyd triangle of order n

We want a *pretty print*

For home 2: Floyd triangles

We want a *pretty print*

print may have additional arguments "by keyword":

`sep`: string used to separate values *in the same print* (default: ' ')

`end`: string used to *terminate, after a print* (default: newline: '\n')

```
print(10,20)
print(30,40)
print(10,20,sep=' & ',end=' * ')
print(30,40,sep=' & ')
print(50)
```

```
10 20
30 40
10 & 20 * 30 & 40
50
```

\t is "tab"

`\n is "neline"`

`etc: look in the doc`

Double iteration: matrices

Python does not have a type for array/matrices

(but we will see and use the library NumPy)

AS TOY EXERCISE for nested iterations, encode a *nxm* matrix as:

- list of lists
    - a list of *n* elements (the rows),
    - each element is a a list of m elements (the *m* elements of the row)

$$A = \begin{pmatrix} 0 & 1 \\ 3 & 2 \\ 5 & 6 \end{pmatrix}$$

```
LA =
[ [0,1],
  [3,2],
  [5,6] ]
```

Double iteration: matrices

Element $a_{i,j}$ of matrix A
is encoded as element L[i-1][j-1] of list LA
(supposing that the indexes of A start at 1)
For simplicity: suppose that both start from zero

$$A = \begin{pmatrix} 0 & 1 \\ 3 & 2 \\ 5 & 6 \end{pmatrix}$$

```
LA =
[ [0,1],
  [3,2],
  [5,6] ]
```

Double iteration: matrices

Exercise 1:

Write a function

def is_sum_mat(A,B,S):

  '''A, B, S: matrices encoded as lists

verify that S is the matrix sum of A and B:

for any i and j: S[i][j]=A[i][j]+B[i][j]'''

Double iteration: matrices

Exercise 2:

Write a function

```
def sum_mat(A,B):
   '''A, B: matrices encoded as lists
```

return a new list S, the matrix sum of A and B, as a list of lists'''

Double iteration: matrices

Exercises 3-4-5:

(1) Function which computes and returns the *transpose* of a matrix, taken as a parameter

(2) Function which takes two *lists* (non matrices!) of the same length, and computes and returns the *scalar product of the two lists*

(3) Function which returns the *product of two matrices* A and B, given as parameters (you may assume they are in the correct format).

Hint: you may use the transpose to make the scalar product between the rows of A and the rows of the transpose of B