ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Lesson 5

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

30 September 2024

Michael Lodi

Department of Computer Science and Engineering

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

One-slide recap of lesson 4

`list` is a compound/structured type (group of values each of its type). Mutable sequence.
Square brackets. Same operations of other immutable sequences, plus…

You can modify the value preserving the identity, Assignment with a selection on the LHS. No changes in the name-values binding, but change in the selection of the list on LHS. Powerful system but be careful with aliasing and side effects.

Never use `for` loop when the list length changes inside the iteration
(`for` freezes the identity, not the value).

Command `del` removes elements. Many methods on lists can modify the list (e.g. `append`). Often they return `None` (so do not use on RHS of assignment). Unlike operations, these methods do not create new objects.

Augmented assignment (e.g. `a += k`) is like a shorthand (e.g. `a = a + k`) but: 1) operation in place if possible (i.e. `extend` instead of `+` for lists); 2) LHS evaluated only once
3) LHS evaluated first (unlike any other assignment)

Functions can modify global mutable objects by accessing through parameters
or through the scope (global scope is visible inside functions – more on this today…)

Next lectures for ILAI

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 ;-)

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 (eg. Theoretical), 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

Recursion

Functions may call functions in their body

Hence a function may call *itself* in its body

In this case, we say that that function is *recursive*

The name of a function is in scope within the function

recursion: ingredients

base case
recursive case

termination :
the recursive case "recurs" on a *simpler* case

simpler :
closer to the base case (whatever this means)

As in the case of `while`:
no linguistic guarantee that this is the case
It is the responsability of the programmer

Recursion: a form of repetition

```
def printrec(n):
    ''print "OK" n times,
     then print "done!" '''
    if n==0:
        print(done!')
    else:
        print('OK')
        printrec(n-1)
```

Recursion: factorial

$n! = n*(n-1)*\ldots*2*1$

$0! = 1$

$n! = n*(n-1)!$

$3! = 3*2! = 3*(2*1!) = 3*(2*(1*0!)) = 3*(2*(1*1)) = 6$

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

Recursion: factorial

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

Trace the execution of fact(3) on pythontutor

Multiple active frames on the stack

Multiple instances of the local names

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

recursion: example

Verify if a string is palindromic

recursion

A general theorem of the theory of computation:

Any computation that is expressible through iteration is also expressible through recursion. And vice versa.

recursion: home exercises

Write recursive functions for the following problems. Also, write an iterative solution.

1. sum of a sequence of integers
2. linear search: given an unordered sequence S and an object el, return True iff el is an element of S
3. Your recursive solution to (2) probably uses slices, that is *copies* of portions of S.
   Write a completely *in place* version (no slices).

   Hint: write a recursive auxiliary function
   `ric_lin_aux(S,index,el)`
   which returns `True` iff `el` is element of the portion of `S` from index to the end.
   Then the real function `ric_lin(S,el)` is simply an initial call to
   `ric_lin_aux(S,0,el)`

file search-rec.py

A new problem: frequency count

Given the string

```
s='When in the Course of human events'
```

count the frequency of each letter

W appears 1 time

h appears 3 times

e appears 5 times

' ' appears 6 times

etc.

First try: use lists

A paradigmatic problem: frequency count

Much easier if…

we could directly use the elements of s (characters of a string) as index of objects (integer counts) in a suitable sequence/container

Dictionaries

A list is (mutable) mapping between

indexes and objects

indexes = initial segment of natural numbers

A dictionary is a generalization:

A (mutable) mapping between

immutable objects and objects

Dictionaries: mapping

Compound type

Mutable

A finite mapping between

immutable objects: the *keys* of the dictionary

arbitrary objects: the *values* of the dictionary

Generalize lists: mapping betwen [0:len(S)] and objects

Dictionaries: dict

Values: Finite mappings between….

Presentation:

{} the empy dict

{k1:ob1, k2:ob2, …, kn:obn}

the mapping sending k*i* into ob*i*

The pairs *ki:obi* are the *items* of the dictionary

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dictionaries: dict

{k1:ob1, k2:ob2, …, kn:obn}

Example:

```
{1:10,2:20,'one':20}
```

Keys:

all distinct

all of immutable type (also any component, if any, should be immutable)

Dictionaries

Main operations:

len(D): numbers of items (pairs)

selection: D[k] k is a key (error if k not a key)

deleting: del D[k] k is key

mutable:

hence selection may be LHS of assignment

D[k] = object

set in D the pair k:object, *creating it if necessary*

k in D True sse k *is a key in* D

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dictionaries

Which of the following raise an error?

D1= {1:10, 2:20}

D2={1:10, 'a':3.14, 3:3}

D3={1:[1,2,3], 3:{1:10,2:20,3:30}, 2:(1,2,3)}

D3[5]='pippo'

D3[1][0]=100

D3[2][1]=20 *error (tuple is not mutable)*

D4={(1,2):[1,2]}

D5={[1,2]:(1,2)} *error: key is mutable*

D6={(1,[2]):23} *error: key is mutable*

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dictionaries

No slices

No concatenation

they are *not sequences*

D1==D2 : True sse D1 and D2 have the same items, independently from the order of the items

Listing the pairs of a dictionary respects the insertion order (only from Python 3.7)…

… but we will always consider dict as UNORDERED.

Dictionaries vs lists of pairs

D={1:10, 'a':3.14, 3:30, 5:50}

L=[(1,10), ('a',3.14), (3,30), (5,50)]

Same information

Different efficiency of the access operation by key

list: sequential, hence O(len(L)) time

dict: hash access, constant time

Access by key to a dict is as efficient as an access by index to a list

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dictionaries

for e in D:

e varies on the keys of *D*

in which order?

From Python 3.7: in the order in which keys have been inserted in D

Before: in an arbitrary order depending on the machine

From L4:

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 dict in presence of side effects!

Never (never !)

use `for` on dictionaries,

if their length

is modified in the `for` body

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

This time it is an *absolute* prohibition:

a modification to a dict which is used to control a for
raises a run time error!

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The method get()

Let's remember the frequency count on a sequence S

This is wrong:

```
Freq={}
for e in S:
   Freq[e] += 1
```

Raises a key error

The method get()

Let's remember the frequency count on a sequence S

```
Freq={}
for e in S:
    if e in Freq:
        Freq[e] += 1
    else:
        Freq[e] = 1
```

This is wrong:

```
Freq={}
for e in S:
    Freq[e] += 1
```

Raises a key error

Common situation: check if key present, otherwise update

The method get()

```
Freq={}
for e in S:
   if e in Freq:
        Freq[e] += 1
    else:
         Freq[e] = 1
```

This is wrong:

```
Freq={}
for e in S:
   Freq[e] += 1
```

Raises a key error

Using instead the method `get()`

`D.get(key, default)`

returns `D[key]` if it exists; otherwise returns `default`

```
for e in S:
   Freq[e] = Freq.get(e,0)+1
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dictionaries: some methods

```
D.keys()
D.values()
D.items()
```

They return *view objects* (dynamic lists)
on keys, values, items (respectively)

(See the meaning of "view object" in Thonny)

On these views the operator "in" is defined

We may freeze these dynamic views:

```
L=list(D.keys())          T=tuple(D.values())
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Method update(): a substitute for concatenation

Concatenation does not exist for dictionaries

The method update does something similar, when possibile

Method update(): a substitute for concatenation

`D1.update(D2)`

extends `D1` with the items of `D2`,

if there are equal keys in D1 and D2, the ones in D2 ("update") are chosen

```
>>>
D1={"house":"casa","cat":"gatto","red":"rosso"}
>>> D2 = {"cat":"micio","grey":"grigio"}
>>> D1.update(D2)
>>> print(D1)
{'house': 'casa', 'cat': 'micio', 'red':
'rosso', 'grey': 'grigio'}
```

From dict to list

Use methods

```
D.keys()
D.values()
D.items()
```

and freeze them into lists. In particular

```
list(D.items())
```

returns as a list of tuples the items of the dictionary

```
list({1:'uno',2:'due',3:'tre'}.items())
```

returns

```
[(1, 'uno'), (2, 'due'), (3, 'tre')]
```

From dict to list

Warning:

```
list(D)
```

is equivalent to

```
list(D.keys())
```

From list to dict

`dict(…)` on a list of pairs returns a dict

```
dict([(1, 'uno'), (2, 'due'), (3, 
'tre')])
```

returns

```
{1:'uno',2:'due',3:'tre'}
```

Accessing scopes: globals(), locals(),etc.

Global, enclosing, local scopes (or *namespaces*)

Scope: moment in the execution where some name is available

At any moment in the run of a program we have *four* scopes for names:

1. Built-In
2. Global
3. Enclosing
4. Local

Builtins are created when the interpreter starts up

Try

```
dir(__builtins__)
```

*Function `dir()` lists the *attributes* of the class of its argument

Global, enclosing, local scopes (or *namespaces*)

```
a=10
def f(s):
    x=20
    def g(u):
        return u
    return s+g(x)
w=f(a)
print(w)
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Global, enclosing, local scopes (or *namespaces*)

```
a=10
def f(s):
    x=20
    def g(u):
        return u
    return s+g(x)
w=f(a)
print(w)
```

`a` and `w` are global:
defined at the top level

Global, enclosing, local scopes (or *namespaces*)

```
a=10
def f(s):
    x=20
    def g(u):
        return u
    return s+g(x)
w=f(a)
print(w)
```

`a` and `w` are global:
defined at the top level

`s` and `x` are local to `f`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Global, enclosing, local scopes (or *namespaces*)

```
a=10
def f(s):
    x=20
    def g(u):
        return u
    return s+g(x)
w=f(a)
print(w)
```

`a` and `w` are global:
defined at the top level

`s` and `x` are local to `f`

`u` is local to `g`

`s` and `x` are non local to `g` and non global
(or: in the enclosing scope)

LEGB

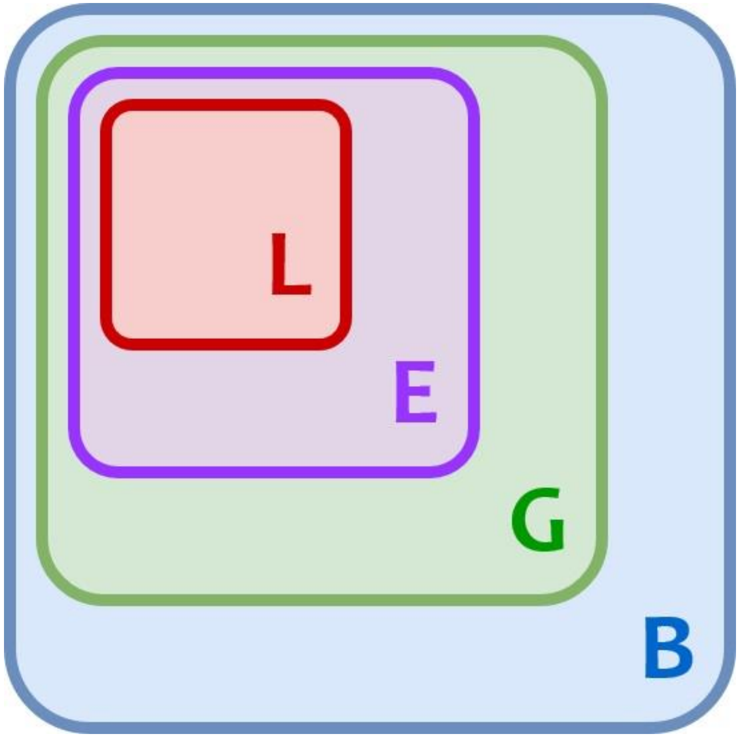
Searching for a name in the internal state works inside-out:

first Local

then Enclosing

then Global

finally Built-in



The `global` command

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The `global` command

In a function definition

```
global <name>
```

signals that `<name>` should be taken from the global scope

```
A=10
def f():
    global A
    A=2000
f()
print(A)      #prints 2000
```

The `nonlocal` command

In a function definition

```
nonlocal <name>
```

signals that `<name>` should be taken from the nonlocal scope

```
A=10
def f():
    A=100
    def g():
        nonlocal A
        A=2000
    g()
    return A
print(f())   #prints 2000
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The `local` and `nonlocal` commands

Use them sparingly!

Accessing scopes

`globals()`

returns the dict of the global names with their binding

`locals()`

returns the dict of the local names with their binding

There is *not* a similar `nonlocals()` function

`globals()` returns the actual dictionary

used by the abstract machine

➔ we may modify it

(not for locals…)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Comprehension

In Set Theory

the comprehension principle

is the operation by which,

given a condition expressible by a formula $\phi(x)$,

we form the set of all those x meeting that condition:

$$\{x \mid \phi(x)\}$$

Comprehension: list

We usually form lists out of other sequences

E.g. The doubles of the numbers from 0 to 100

```
doubles=[]
for i in range(0,101):
    doubles.append(2*i)
```

It would be nice to have a compact way to express this
Something like: *the list of 2*i for i in the range(0,101)*

```
doubles=
```

Comprehension: list

We usually form lists out of other sequences

E.g. The doubles of the numbers from 0 to 100

```
doubles=[]
for i in range(0,101):
    doubles.append(2*i)
```

It would be nice to have a compact way to express this
Something like: *the list of 2*i for i in the range(0,101)*

*doubles=[2*i for i in range(1,101)]*

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Comprehension: list

Or also:

the even numbers out of a given list of integers LN

LN=…..

```
even=[]
for i in LN:
    if i%2==0:
        even.append(i)
```

Something like: *the list of those i in LN if i%2==0*

even=…

Python allows this form of expression
which is called "comprehension" in set-theory, in math

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Comprehension: list

Or also:
the even numbers out of a given list of integers LN
LN=.....

```
even=[]
for i in LN:
    if i%2==0:
        even.append(i)
```

Something like: *the list of those i in LN if i%2==0*
even=[i for i in LN if i%2==0]

Python allows this form of expression
which is called "comprehension" in set-theory, in math

Examples

1. The list of lists of numbers from 0 to i, for i less than 20

observe this is a nested comprehension

```
LL=[]
....
```

[[],[0],[0,1],[0,1,2],...,[0,1,2,...,19]]

Examples

1. The lists of numbers from 0 to i, for i less than 20

```
[ [k for k in range(i)] for i in range(20)]
```

observe this is a nested comprehension

```
LL=[]
for i in range(20):
    AL=[]
    for k in range(i):
        AL.append(k)
    LL.append(AL)
```

Examples

1. The lists of numbers from 0 to i, for i less than 20

```
    [ [k for k in range(i)] for i in
range(20)]
```

observe this is a nested comprehension

```
2. H = [(x,y) for x in range(2)
              for y in range(3)]
```

```
3. from math import pi
P = [round(pi,i) for i in range(6)]
```

Equivalent form

```
T = [(x,y) for x in range(2)
           for y in range(3)
               if x<y]
```

```
T=[]
for x in range(2):
    for y in range(3):
        if x<y:
            T.append((x,y))
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

list comprehension

The general definition is a bit obscure

A simple version:

`[`*expression* `for` *name* `in` *sequence*`]`

expression in a comprehension
is *always* evaluated

- after the `for` *name* `in` *sequence*
- in its local frame:

name is local to the comprehension

on Pythontutor…

list comprehension

A more complex version

```
[expression for name in sequence
                    series of for/if ]
```

equivalent to

```
res=[]
for name in sequence:
    series of for/if
        res.append(expression)
return res
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

list comprehension: exercises

1. Given a unary function f and a sequence S, return the list of the application of f to the elements of S

2. Given two sequences S1 and S2 return the list of the elements present in both S1 and S2 (don't bother about multiplicity: if e appears in both S1 and S2, maybe many times in S1, or in S2, or in both, it may appear with whatever multiplicity in the result)

3. Pythagorean triples: (a,b,c) such that $a^2+b^2=c^2$, for $a+b+c \leq k$

Next lectures for ILAI

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 ;-)

Challenge: permutations (via comprehension!)

def perm(L):

"return the list of all the permutations of the elements of L; does not modify L'''

Example:

perm([1,2,3])

returns

[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

Hint: argue recursively (so you can have a very simple base case)

The permutations of L are obtained by taking as first element, any element of L, and taking then all the permutations of….

You need an auxiliary function erase(e,L) which returns a copy of L from which e has been removed

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Other comprehensions

On dicts:

{*key*:*val* for *name* in *sequence*}

Examples

```
{i:i**2 for i in range(10)}
{i:i**2 for i in range(10) if i%2==0}
{i:[k for k in range(i)]
                    for i in range(10)}
```


ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Other comprehensions

On sets (we don't cover sets in this course)

Sets are mutable collections of objects

without repetition

and unordered

They are iterable

Be careful: no comprehension on tuples

There is no comprehension on tuples

Be careful: no comprehension on tuples

There is no comprehension on tuples

Yet expressions which *seem*

"tuple comprehension"

are legal Python code…

Be careful: no comprehension on tuples

There is no comprehension on tuples

Yet expressions which *seem*

"tuple comprehension"

are legal Python code…

```
H=(i*2 for i in range(10))
```

Generators: super simplified view

```
G=(i*2 for i in range(3))
```

A generator is a kind of "*potential sequence/tuple*"

which is able to *generate* the elements of the sequence, through the predefined function next(…)

`next(G)` *: evaluates to 0*

`next(G)` *: evaluates to 2*

`next(G)` *: evaluates to 4*

`next(G)` *: the generator is exhausted*

an error is raised

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Generators: super simplified view

```
G=(i*2 for i in range(3))
```

Important use: a generator may be the <sequence> used in a for:

```
for e in G:
    print(e)
```

for calls next(G) at any iteration (and assign the result to e)

NB: a generator is a particular case of *iterator*, the general structure on which next() is defined and on which we may iterate with a for

Loop patterns

https://csed-unibo.github.io/#!pages/pattern_cicli.md