Computer Science 1001.py

Lecture 8[†]: Introduction to Object Oriented Programming; Recursion and Recursive Functions

Instructors: Haim Wolfson, Amiram Yehudai
Teaching Assistants: Yoav Ram, Amir Rubinstein

School of Computer Science
Tel-Aviv University
Spring Semester, 2012-13
http://tau-cs1001-py.wikidot.com

Lecture 7: Highlights

- Integer greatest common divisor.
- Euclid's gcd algorithm.
- Using gcd statistics to approximate $6/\pi^2 \approx 0.6079271$ and π.

- Searching in unordered lists and in ordered lists.
- Sequential search vs. binary search.

Lecture 7: Python Highlights

- The `sorted()` function for sorting lists.
- Employing a lambda expression:
  `key = lambda elem :  elem[0]` to guide `sorted()`.

Lecture 8: Plan

- Classes and methods (a very gentle intro to object oriented programming).
- Recursion, and recursive functions.

Object Oriented Programming (OOP)

OOP is a major theme in programming language design, starting with Smalltalk in the late 1970s (out of the legendary Xerox Palo Alto Research Center, or PARC, where many other ideas used in today's computer environment were invented). Other "OOP languages" include Eiffel, C++, Java, C#, and Scala.

Entities in programs are modeled as objects. They represent encapsulations that have their own attributes (also called fields), that represent their state, and functions, or operations that can be performed on them, termed methods. Creation and manipulation of objects is done via their methods.

Object Oriented Programming (OOP), cont.

The object oriented approach enables modular design. It facilitates software development by different teams, where each team works on its own object, and communication among objects is carried out by well defined methods' interfaces.

Python supports object oriented style programming (maybe not up to the standards of OOP purists). We'll describe some facets, mostly via concrete examples.

A more systematic study of OOP will be presented in Tochna 1, using Java.

Line Class in Python

```
class Line:
    """ Stores a line in the 2D plane. Line is defined
    by two points (x1,y1) and (x2,y2). It is represented
    by a,b where y=a*x + b is the line equation, or by x=c
    in case the line is parallel to the y axis """

    def __init__(self, x1,y1,x2,y2):
        assert type(x1)==float and type(y1)==float and \
               type(x2)==float and type(y2)==float and \
               (x1 != x2 or y1 != y2)
        self.point1=(x1,y1)
        self.point2=(x2,y2)
        if x1 != x2:
            self.slope=(y2-y1)/(x2-x1)
            self.offset= y1-x1*(y2-y1)/(x2-x1)
        else:
            self.slope="infty"
            self.offset=x1
```

The Line class is initialized by a pair of points in the 2D plan: `(x1,y1)` and `(x2,y2)`. The class has two fields: `slope`, and `offset`. These fields are computed internally from the two points. They can be accessed directly.

Line Class, Additional Methods

```
    def __repr__(self):
        if self.slope=="infty":
            output= "x="+str(self.offset)
        else:
            output= "y="+str(self.slope)+"*x"+"+"+str(self.offset)
        return output

    def __eq__(self,other):
        assert isinstance(other,Line)
        return self.slope==other.slope \
                    and self.offset==other.offset

    def is_parallel(self,other):
        assert isinstance(other,Line)
        return self.slope==other.slope
```

`__init__`, `__repr__`, `__eq__`, `is_parallel` are methods of this class. (The first three are special methods). These methods are correspondingly used to initialize an object in this class, describe how it is represented (when printing such an object), determine when two lines are equal, and when they are parallel.
`assert` evaluates its boolean argment and aborts if false.

Line Class, Example Execusions

```
>>> a=Line(1.,1.,2.,2.)
>>> b=Line(2.,2.,5.,5.)
>>> a.slope
1.0
>>> a.offset
0.0
>>> b.slope
1.0
>>> b.offset
0.0
>>> a==b
True
>>> Line.is_parallel(a,b)
True
>>> c=Line(2.,3.,5.,6.)
>>> c.slope
1.0
>>> c.offset
1.0
>>> a==c
False
>>> Line.is_parallel(a,c)
True
```

Student Class in Python

```
class Student:
    def __init__(self):
        self.name = generate_name()
        self.id = random.randint(2*10**7,6*10**7)

    def __repr__(self):
        return  "<{name}, {id}>".format(**self.__dict__)

    def __lt__(self, other):
        return self.name < other.name

    def is_given_name(self,word):
        return self.name.startswith(word+" ")
```

The Student class has two fields: `name`, and `id`. These fields can be accessed directly, and values can be assigned to them directly.

`__init__`, `__repr__`, `__lt__`, `is_given_name` are methods of this class. As can be guessed from the names of the first two, they are used for initializing an object in this class, describe how it is represented (when printing such an object).

Student Class in Python, cont.

```
class Student:
    def __init__(self):
        self.name = generate_name()
        self.id = random.randint(2*10**7,6*10**7)

    def __repr__(self):
       return  "<"+self.name+","+str(self.id)+">"

    def __lt__(self, other):
        return self.name < other.name

    def is_given_name(self,word):
        return self.name.startswith(word+" ")
```

`__init__`, `__repr__`, `__lt__`, `is_given_name` are methods of this class. The third method, `__lt__`, describes how comparison ($<$, less than) of two such objects is determined. This enables us to sort lists containing objects of this class, for example.
The fourth method, `is_given_name`, checks if the `name` field starts with the argument in `word`, followed by a blank.

Student Class in Python: Example

```
>>> students_list
[<Or, 39316939>, <Yana, 52061841>, <Amir, 49419212>,
<Roee, 40604275>, <Noa, 24908823>, <Gal, 56592426>,
<Barak, 29548638>, <Rina, 52552066>, <Tal, 57995311>,
<Lielle, 43513357>, <Shady, 46042015>, <Yuval, 32125900>,
<Walt Disney, 27907836>]
>>> len(students_list)
13
>>> students_list[0].is_given_name("Yuval")
False
>>> students_list[11].is_given_name("Yuval")
False
>>> students_list[12].is_given_name("Walt")
True
```

In case you did not notice, Walt Disney has joined our class (no. 107, and hopefully the last one to join). There are some plausible reasons why you never saw him present. The most reliable one, IMHO, is that he stays home and watches the video footage on YouTube.

Generating Names At Random: Python Code

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'

def generate_name():
    """ generate a random first (given) name with 3-6 letters,
        space, and a random family name with 4-8 letters """

    first=random.sample(alphabet ,random.randint(3,6))
    family=random.sample(alphabet ,random.randint(4,8))
    name=str.join("", first) + " " + str.join("", family)
    return str.title(name)

>>> random.sample(alphabet ,random.randint(3,6))
['h', 'o', 'n', 'x', 'g', 's']
>>> random.sample(alphabet ,random.randint(3,6))
['f', 'h', 'd', 'j']

>>> for i in range(5):
generate_name()

'Oudwab Ngyzb'
'Slhbm Jnypu'
'Sxufj Drhbs'
'Cjdrhm Wqhxve'
'Snoc Lvcso'
```

Tinkering with the Student Class

```
def students(n):
    return [ Student() for i in range(n)]

>>>names=["Or","Yana","Amir","Roee","Noa","Gal","Barak",
       "Rina","Tal","Lielle","Shady","Yuval","Walt Disney"]
>>>students_list=students(13)
>>>for i in range(13):
   students_list[i].name=names[i]
>>>for i in range(13):
   print(students_list[i])

<Or, 39316939>
<Yana, 52061841>
<Amir, 49419212>
<Roee, 40604275>
<Noa, 24908823>
<Gal, 56592426>
<Barak, 29548638>
<Rina, 52552066>
<Tal, 57995311>
<Lielle, 43513357>
<Shady, 46042015>
<Yuval, 32125900>
<Walt Disney, 27907836>
```

Testing Equality for the Student Class

```
>>> students_list[11]==students_list[12]
False   # why should one expect equality?
>>> students_list[11].name=students_list[12].name
>>> students_list[11].id=students_list[12].id
>>> students_list[11].name==students_list[12].name
True
>>> students_list[11].id==students_list[12].id
True
>>> students_list[11]==students_list[12]
False   # well, this IS unexpected
```

We conclude that equality of the two different fields does not imply equality of the corresponding Student objects.

We can, however, define equality explicitly, as an additional method of the class. This method will be called __eq__. It will describe how equality (==) of two such objects is determined. The interpreter "understands" that this method will be used when a conditional involving == of two Student objects is evaluated.

Defining and Retesting Equality for the Student Class

```
class Student:
    def __init__(self):
        self.name = generate_name()
        self.id = random.randint(2*10**7,6*10**7)

    def __repr__(self):
       return  "<"+self.name+","+str(self.id)+">"

    def __eq__(self, other):
        return self.name == other.name \
               and self.id == other.id

    def __lt__(self, other):
        return self.name < other.name

    def is_given_name(self,word):
        return self.name.startswith(word+" ")

>>> students_list[11].name=students_list[12].name
>>> students_list[11].id=students_list[12].id
>>> students_list[11]==students_list[12]
True          # desired effect
```

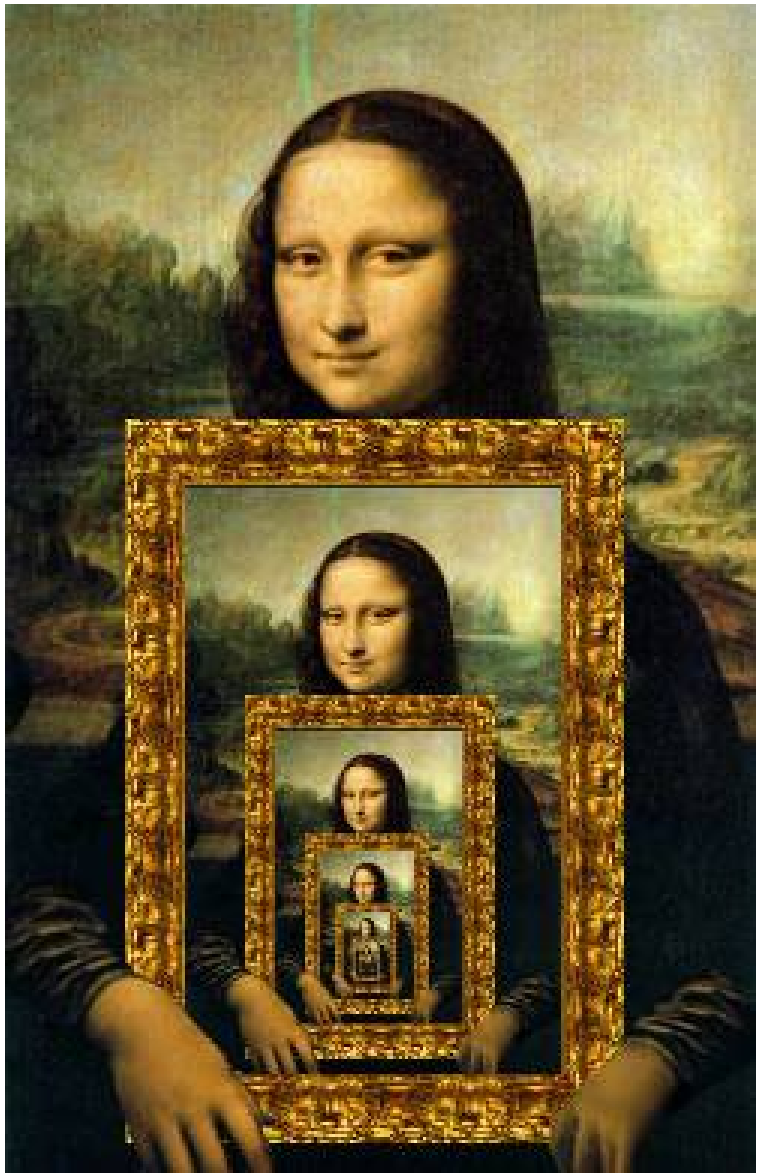
And Now to Something Completely Different: Recursion

'If seven maids with seven mops
Swept it for half a year,
Do you suppose,' the Walrus said,
'That they could get it clear?'
'I doubt it,' said the Carpenter,

And shed a bitter tear.
'O Oysters, come and walk with us!'
The Walrus did beseech.
'A pleasant walk, a pleasant talk,
Along the briny beach:
We cannot do with more than four,
To give a hand to each.'

Through the Looking-Glass and What Alice Found There:
Lewis Carroll, 1871.

And Now For Something Completely Different: Recursion



(taken from http://www.dominiek.eu/blog/?m=200711)

Recursion

A function $f(\cdot)$, whose definition contains a call to $f(\cdot)$ itself, is called recursive.

A simple example is the factorial function, $n! = 1 \cdot 2 \cdot \ldots \cdot n$.
It can be coded in Python, using recursion, as following:

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

A second simple example is the Fibonacci numbers, defined by $F_1 = 1,\ F_2 = 1$, and for $n > 2,\ F_n = F_{n-2} + F_{n-1}$.
A Fibonacci numbers function can be programmed in Python, using recursion, as following:

```
def fibonacci(n):
    if n<=2:
        return 1
    else:
        return fibonacci(n-2)+fibonacci(n-1)
```

Recursion and Convergence

At first sight, one may suspect that the recursive definitions above will lead nowhere. Or in other words, that they are cyclical and will never converge. This surely is not the case, and for specific instances one can simply run the code and get the (correct) answers.

```
>>> factorial(19)
121645100408832000
```

```
>>> fibonacci(21)
10946
```

There are two keys to correct design of recursive functions. The first one is to have a base case (one or more), which is the halting condition (no deeper recursion). In the factorial example, the base case was the condition `n==1`. In the Fibonacci example, it was `n<=2`. The second "design principle" is to make sure that all executions, or "runs", of the recursion lead to one of these base cases.

Recursion and Cyclicity

Recursive definitions that are cyclical will naturally not converge. A famous example is the following "dictionary definition" of recursion:

Recursion
re·cur·sion
n. Mathematics
See "Recursion".
[from Latin recursus, past participle of recurrere, to run back; see **recur**.]

You may also explore Google's version.

Choice of Base Cases

Recursive definitions that are seemingly proper may in fact diverge (lead to an infinite loop), due to unforeseen conditions in which the base cases are never reached. Consider, for example, the factorial function,

```
>>> factorial(1.9)
>>> factorial(0)
```

will both create an infinite loop.

(How would you fix it? Does it need fixing in the first place?)

More Complicated Recursion

A pair of functions $f(\cdot),\ g(\cdot)$, where the definition of the first function, $f(\cdot)$, includes a call to the second function, $g(\cdot)$, and the definition of the second function, $g(\cdot)$, includes a call to the first function, $f(\cdot)$, are also called recursive functions.
This definition generalizes to more functions as well.

The following example is due to Douglas R. Hofstadter, the well known cognitive scientist (and physicist, mathematician, computer scientist, etc.), who is possibly best known as the author of the book "Gödel, Escher, Bach: an Eternal Golden Braid". It involves the Female (F) and Male (M) sequences, defined on the next slide.

Hofstadter Female and Male Sequences

Are defined as following:

$$F(0) = 1,\ M(0) = 0,$$
$$F(n) = n - M(F(n-1)), \quad n > 1$$
$$M(n) = n - F(M(n-1)), \quad n > 1$$

The following Python code computes these sequences:

```
def female(n):
    if n<=0:
        return 1
    else:
        return n - male(female(n-1))

def male(n):
    if n<=0:
        return 0
    else:
        return n - female(male(n-1))
```

Binary Search, Revisited

We saw an iterative version of binary search.

```
def binary_search(key,lst):
    """ iterative binary search
        lst better be sorted for binary search to work"""
    n=len(lst)
    lower=0
    upper=n-1
    outcome=None   # default value
    while lower<=upper:
        middle=(upper+lower)//2
        if key==lst[middle].name:     # item found
            outcome=lst[middle]
            break  # gets out of the loop if key was found
        elif key<lst[middle].name:    # item cannot be in top half
            upper=middle-1
        else:          # item cannot be in bottom half
            lower=middle+1
    if not outcome:  # holds when the key is not in the list
        print(key, "not found")
    return outcome
```

Binary Search, Recursively

Here is a recursive implementation of the same task. This code follows the iterative code closely. It passes the key, the original list and two indices (`lower, upper`) to the recursive call.

```
def rec_binary_search(key,lst,lower,upper):
    """ recursive binary search.
        passing lower and upper indices"""
#    print(lower,upper)   # for debugging purposes
    if lower>upper:
        return None
    elif lower==upper:
        if key==lst[lower].name:
            return lst[lower]
        else:
            return None
    elif key==lst[(lower+upper)//2].name:
        return lst[(lower+upper)//2]
    elif key<lst[(lower+upper)//2].name:
         # item cannot be in top half
        return rec_binary_search(key,lst,lower,(lower+upper)//2-1)
    else:          # item cannot be in bottom half
        return rec_binary_search(key,lst,(lower+upper)//2+1,upper)
```

Recursive Binary Search, Example Runs

Input list contains objects from the Student class.

```
>>> from binary_search import *
>>> stkist=students(10**5)
>>> srt=sorted(stkist)
>>> binary_search(srt[11].name,srt)
<Abe Kwip, 35285412>
>>> rec_binary_search(srt[11].name,srt,0,len(srt))
<Abe Kwip, 35285412>

>>> elapsed("binary_search(srt[11].name,srt)",number=10000)
0.2976080000000003
>>> elapsed('rec_binary_search(srt[11].name,srt,0,len(srt))',
 number=10000)
0.4218190000000017  # 40% slower

>> binary_search('Al Capone',srt) # Al Capone is not in the list
>>> rec_binary_search('Al Capone',srt,0,len(srt))

>>> elapsed("binary_search('Al Capone',srt)",number=10000)
0.2718720000000001
>>> elapsed("rec_binary_search('Al Capone',srt,0,len(srt))",
 number=10000)
0.4183180000000002 # 40% slower
```

Binary Search, Recursively, Using Slicing

Here is a second recursive of the same task.

```
def rec_slice_binary_search(key,lst):
    """ recursive binary search.
        lst better be sorted for binary search to work"""
    n=len(lst)
    if n<=0:
        return None
    elif n==1:
        if key==lst[0].name:
            return lst[0]
        else:
            return None
    elif key==lst[n//2].name:
        return lst[n//2]
    elif key<lst[n//2].name:  # item cannot be in top half
        return rec_slice_binary_search(key,lst[0:n//2])
    else:                     # item cannot be in bottom half
        return rec_slice_binary_search(key,lst[n//2:n])
```

A Second Binary Search, Recursively with Intermediate Printing

```
def disp_rec_slice_binary_search(key,lst):
    """ recursive binary search, displaying intermediate results.
        lst better be sorted for binary search to work"""
    n=len(lst)
    print(n,lst[n//2])
    if n<=0:
        print(key," not found")
        return None
    elif n==1:
        if key==lst[0].name:
            return lst[0]
        else:
            print(key," not found")
            return None
    elif key==lst[n//2].name:
        return lst[n//2]
    elif key<lst[n//2].name:    # item cannot be in top half
        return disp_rec_slice_binary_search(key,lst[0:n//2])
    else:                       # item cannot be in bottom half
        return disp_rec_slice_binary_search(key,lst[n//2:n])
```

Recursive Binary Search, Example Runs

Input list contains objects from the Student class.

```
>>> st_list=students(10**5)  # names, id generated at random
>>> srt=sorted(st_list)
>>> srt[5000-2:5000+2]  # sample slice
[<Mxqf Ydxpb, 20983125>, <Mxqnl Asmlk, 47035596>,
<Mxqra Djtks, 34760839>, <Mxqs Ihqszbmg, 25600859>]

>>> disp_rec_slice_binary_search(srt[10**4//2-1].name,srt)
10000 <Mxqra Djtks, 34760839>
5000 <Gleadh Fxhq, 46713613>
2500 <Jsyzbi Wpskvagm, 57644362>
1250 <Lgn Cdwemf, 33269908>
625 <Mbf Zrvfbs, 40191094>
313 <Mlw Jdiskpr, 39567152>
157 <Mrdiog Agwntkr, 31437982>
79 <Muiegd Eyjx, 40804133>
40 <Mwhc Nhzk, 24117233>
20 <Mwxrjs Lpfg, 39950463>
10 <Mxjq Lwkqoh, 28878839>
5 <Mxnh Cmfyu, 39175743>
3 <Mxqf Ydxpb, 20983125>
2 <Mxqnl Asmlk, 47035596>
<Mxqnl Asmlk, 47035596>
```

Recursive Binary Search, Two Additional Example Runs

```
>>> disp_rec_slice_binary_search(srt[10**4//2].name,srt)
10000 <Mxqra Djtks, 34760839>
<Mxqra Djtks, 34760839>  # smack in the middle

>>> disp_rec_slice_binary_search("Al Capone",srt)# key not existing
10000 <Mxqra Djtks, 34760839>
5000 <Gleadh Fxhq, 46713613>
2500 <Dgkybc Fzsiyuaq, 20836310>
1250 <Bqk Cmpz, 48888472>
625 <Aue Qtzosne, 33283402>
312 <Alk Yifzbeph, 37115360>
156 <Agiy Sodifj, 46931518>
78 <Aikjq Bcygumlh, 45702032>
39 <Ajzt Ekut, 38230430>
20 <Aksbtp Ztmdyp, 27808941>
10 <Akw Pradbxth, 54052265>
5 <Aldxsi Jwao, 49743185>
2 <Aldgo Aqorpdbx, 37914905>
1 <Akw Pradbxth, 54052265>
Al Capone  not found
```

Binary Search, Iterative vs. Recursive

Lets put the two functions to the test of the clock.

```
>>> from binary_search import *
>>> st_list=students(10**6)
>>> srt=sorted(st_list)

>>> elapsed('rec_slice_binary_search("Al Capone",srt)',number=1)
0.07140300000000366
>>> elapsed('rec_slice_binary_search("Al Capone",srt)',number=100)
6.905284999999992
>>> elapsed('binary_search("Al Capone",srt)',number=200000)
6.167489000000175

>>> srt[5*10**5+1].name
'Nad Frowzgx'
>>> elapsed('rec_slice_binary_search("Nad Frowzgx",srt)',number=1)
0.06951200000000313
>>> elapsed('binary_search("Nad Frowzgx",srt)',number=2000)
0.06343100000000845
```

So the iterative version is approximately 2000 times faster than the recursive, sliced version (both for existing and non existing keys).

Food for thought: Why? What is the complexity of the sliced recursive version?

Sort and Search

As we saw, binary search requires preprocessing – sorting.

We will introduce one approach to sorting (out of very many).

This approach, quicksort, employs both randomization and recursion.

(Contents include Game Board, 6 Moving Pieces, 6 Tile Holders, 30 Colored Tiles, Over 300 Topic Cards, Sand Timer & Instructions. For 2 to 6 players, ages 12 & up.)

Quicksort

Our input is an unsorted list, say
`[28, 12, 32, 27, 10, 12, 44, 20, 26, 6, 20, 21]`.
We choose a pivot element, simply one of the elements in the list.

For example, suppose we chose 20 (the second occurance). We now compare all elements in the list to the pivot. We create three new lists, termed `smaller, equal, greater`. Each element from the original list is placed in exactly one of these three lists, depending on its size with respect to the pivot.

```
smaller = [12, 10, 12, 6].
equal = [20, 20].
greater = [28, 32, 27, 44, 26, 21].
```

Note that the `equal` list contains at least one element, and that both `smaller` and `greater` is strictly shorter than the original list.

Quicksort (cont.)

What do we do next? We recursively sort `smaller` and `greater`, and then we append the three lists, in order (recall that in Python + means append for lists).

Note that `equal` need not be sorted.

```
return quicksort(smaller) + equal + quicksort(greater)
```

`quicksort(smaller) = [6, 10, 12, 12]`.

`equal = [20, 20]`.

`quicksort(greater) = [21, 26, 27, 28, 32, 44]`.

Final result:

```
[6, 10, 12, 12] + [20, 20] + [21, 26, 27, 28, 32, 44]
= [6, 10, 12, 12, 20, 20, 21, 26, 27, 28, 32, 44].
```

(Original list was

`[28, 12, 32, 27, 10, 12, 44, 20, 26, 6, 20, 21]`.)