Lecture 13: More Recursion: Memoization

School of Computer Science
Tel-Aviv University
Winter Semester, 2013-15
http://tau-cs1001-py.wikidot.com
Lecture 11-12 : Highlights

Recursion, and recursive functions
• Basic examples and definition of recursion
• Fibonacci
• factorial
• Binary search - revisited
  • Sorting
  • QuickSort
• MergeSort
• Towers of Hanoi
• Towers of Hanoi nightmare
• Tail recursion
Lecture 13 - Plan

Recursion, and recursive functions

Revisiting the results of fibonacci
   Memoization
   Recursion depth limit

Two additional examples for recursion:
   Ackermann function
   Mutual recursion

Classes and methods (a very gentle intro to object oriented programming).
Pitfalls of Using Recursion

Every modern programming language, including, of course, Python, supports recursion as one of the built-in control mechanism.

However, recursion is not the only control mechanism in Python, and surely is not the one employed most often.

Furthermore, as we will now see, cases where “naive recursion" is highly convenient for writing code may lead to highly inefficient run times. For this reason, we will also introduce techniques to get rid of recursion (in addition to the elimination of tail recursion that we already saw). We note, however, that in some cases, eliminating recursion altogether requires very crude means.
Computing Fibonacci Numbers

We coded Fibonacci numbers, using recursion, as following:

```python
def fibonacci(n):
    """ plain Fibonacci, using recursion """
    if n < 2:
        return 1
    else:
        return fibonacci(n-2) + fibonacci(n-1)

But surely **nothing** could go wrong with such simple and elegant code... To investigate this, let us explore the running time:
Computing Fibonacci Numbers

But surely **nothing** could go wrong with such simple and elegant code... To investigate this, let us explore the running time:

```python
>>> fibonacci(30)
1346269
>>> elapsed( "fibonacci(30)" )
1.3944975126913148
>>> elapsed( "fibonacci(35)" )
15.288596020470468
>>> elapsed( "fibonacci(40)" )
170.328407752612   # almost 3 minutes!!
```
**Inefficiency of Computing Fibonacci Numbers**

What is causing this *exponential growth* in running time?

```python
def fibonacci(n):
    """plain Fibonacci, using recursion""
    if n < 2:
        return 1
    else:
        return fibonacci(n-2) + fibonacci(n-1)
```

Going over the computation *mentally* (or inserting print commands to track execution "physically"), we observe that `fibonacci(1), fibonacci(2), fibonacci(3),` etc. are computed over and over. This is highly wasteful and causes a **huge overhead**.
Inefficiency of Computing Fibonacci Numbers
Count: Measuring the Inefficiency

We can easily modify the code, so it also counts the number of function invocations, using a global variable, count.

def count_fibonacci (n):
    """ recursive Fibonacci + counting no. of function invocations """
    global count
    count +=1
    if n <2:
        return 1
    else :
        return count_fibonacci (n -2)+ count_fibonacci (n -1)
Count vs. `fibonacci(n)`

```python
>>> count=0
>>> count_fibonacci(20)
10946
>>> count
21891
>>> count=0
>>> count_fibonacci(30)
1346269
>>> count
2692537
>>> count=0
>>> count_fibonacci(40)
165580141
>>> count
331160281      # over 300 million invocations
```

Can you see some relation between the returned value and count? Do you think this is a coincidence? Try to prove, using induction: \( \text{count}(n) = 2 \cdot F_n - 1. \)
Intuition for Efficiently Computing Fibonacci Numbers

Instead of computing from scratch, we will introduce variables fib[0], fib[1], fib[2], .... The value of each such variable will be computed just once. Rather than recomputing it, we will fetch the value from memory, when needed. The technique of storing values instead of re-computing them has different names in different contexts: It is known as memorization, a term coined by Donald Michie in 1968. In programming languages like Lisp (of which Scheme is a variant), where recursion is used heavily, there are programs to do this optimization automatically, at run time. These are often termed memoization.

In other contexts, such as stringology, this technique (remember and reuse computed values, rather than re-computing them) is often used as part of dynamic programming.
Python's Dictionary (**dict**)

In the next version of the function, **fibonacci2**, we will exploit a highly useful data structure in Python, the dictionary (class **dict**). This is a mutable class, storing key:value pairs. The keys (but not the values) should be immutable.

```python
>>> students = {"Dzutv Rztvsud": 48322167, "Wpbo Fgrv": 26753752}  # creating a dictionary
>>> "Dzutv Rztvsud" in students  # membership query
True
>>> "Al Capone" in students
False
>>> students["Dzutv Rztvsud"]  # retrieving value of existing key
48322167
>>> students["Al Capone"] = 48322167  # inserting a new key + value
>>> students
{'Wpbo Fgrv': 26753752, 'Dzutv Rztvsud': 48322167, 'Al Capone': 48322167}
>>> type(students)
<class 'dict'>
```

Python's **dict** does not support having different items with the same key (it keeps only the most recent item with a given key).
Fibonacci Numbers: Recursive Code with Memoization

We will use a dictionary -- an indexed data structure that can grow dynamically. This dictionary, which we name fib, will contain the Fibonacci numbers already computed. We initialize the dictionary with fib dict[0]=1 and fib dict[1]=1.

```python
fib_dict = {0:1, 1:1}    # initial fib dictionary, first two elements
def fibonacci2 (n):
    """ recursive Fibonacci, employing memorization in a dictionary ""
    # print (n)    # diagnostic printing
    if n not in fib_dict :
        res = fibonacci2 (n -2) + fibonacci2 (n -1)
        fib_dict [n] = res
    return fib_dict [n]
```

fib_dict = {0:1, 1:1}    # initial fib dictionary, first two elements
```
Recursive Fibonacci Code with Memoization: Execution

`fibonacci2` is recursive, with exactly the same control flow of `fibonacci`, only it stores intermediate values that were already computed. This small change implies a huge performance difference:

```python
>>> elapsed( "fibonacci(30)"
1.382285332480808
>>> elapsed( "fibonacci2(30)"
0.0002703864389275168
>>> elapsed( "fibonacci2(30)"
0.00011875635387781358    # fib_dict has all the values ,
    # so no computation is needed
```
Recursive Fibonacci Code with Memoization: Execution (cont.)

```python
>>> elapsed ( " fibonacci (35) " )
15.576800843539942
>>> fib_dict ={0:1 ,1:1}; elapsed ( " fibonacci2 (35) " )
0.0002703864389275168
>>> fib_dict ={0:1 ,1:1}; elapsed ( " fibonacci2 (50) " )
0.0003061366215830838

>>> fib_dict ={0:1 ,1:1}; elapsed ( " fibonacci2 (5) " ); fib_dict
0.0001590266215830838
{0: 1, 1: 1, 2: 2, 3: 3, 4: 5, 5: 8}  # in case you were wondering
```
Pushing Performance to the Limit

```python
>>> fib_dict = {0: 1, 1: 1}; elapsed ("fibonacci2 (1000) " )
0.0050543361684186995
>>> fib_dict = {0: 1, 1: 1}; elapsed ("fibonacci2 (1500) " )
0.0048369586210128546
>>> fib_dict = {0: 1, 1: 1}; elapsed ("fibonacci2 (1980) " )
0.009931565110125717
>>> fib_dict = {0: 1, 1: 1}; elapsed ("fibonacci2 (2000) " )
Traceback (most recent call last):
    # removed most of the error message
    File "C:\Users\.....\fib.py", line 36, in fibonacci2
        res = fibonacci2(n-2) + fibonacci2(n-1)
RuntimeError: maximum recursion depth exceeded
```

What the $#*%& is going on?
Python Recursion Depth

While recursion provides a powerful and very convenient means to designing and writing code, this convenience is not for free. Each time we call a function, Python (and every other programming language) adds another “frame" to the current “environment". This entails allocation of memory for local variables, function parameters, etc.

Nested recursive calls, like the one we have in fibonacci2, build a deeper and deeper stack of such frames.

Most programming languages' implementations limit this recursion depth. Specifically, Python has a nominal default limit of 1,000 on recursion depth. However, the user (you, that is), can modify the limit (within reason, of course).
Pushing Performance to the Limit

You can import the Python sys library, and out what the limit is, (and also change it).

```python
>>> import sys
>>> sys.getrecursionlimit() # find recursion depth limit
1000
```

From the examples we ran above, it seems that the recursion depth of fibonacci2 is \( n/2 \). Why? Note that the assignment with the recursive calls is:

\[
\text{res} = \text{fibonacci2}(n -2) + \text{fibonacci2}(n -1)
\]

What if we changed it to:

\[
\text{res} = \text{fibonacci2}(n -1) + \text{fibonacci2} (n -2)
\]
Running fibonacci2 with
res = fibonacci2(n -1) + fibonacci2 (n -2)

>>> fib_dict ={0:1 ,1:1}; elapsed ( " fibonacci2 (900) “ )
0.0045616411224078035

>>> fib_dict ={0:1 ,1:1}; elapsed ( " fibonacci2 (990) “ )
0.004007718752081502

>>> fib_dict ={0:1 ,1:1}; elapsed ( " fibonacci2 (1000) “ )
Traceback (most recent call last):
  # removed most of the error message
    File "C:\Users\......\fib.py", line 36, in fibonacci2
 res = fibonacci2(n-1) + fibonacci2(n-2)
    RuntimeError: maximum recursion depth exceeded
Changing Python Recursion Depth (back to original version)

```python
>>> import sys
>>> sys.getrecursionlimit()             # find recursion depth limit
1000
>>> sys.setrecursionlimit(20000)   # change limit to 20,000
>>> fibonacci2(3000)
664390460366960072280217847866028384244163512452783259
405579765542621214161219257396449810982999820391132226
802809465132446349331994409434926019045342723749188530
316994678473551320635101099619382973181622585687336939
784373527897555489486841726131733814340129175622450421
605101025897173235990662770203756438786517530547101123
748849140252686120104032647025145598956675902135010566
909783124959436469825558314289701354227151784602865710
780624675107056569822820542846660321813838896275819753
281371491809004412219124856375121694811728724213667814
577326618521478357661859018967313354840178403197559969
056510791709859144173304364898001    # hurray
```
Fibonacci Numbers: Iterative (Non Recursive) Solution

We saw that memoization improved the performance of computing Fibonacci numbers dramatically (the function `fibonacci2`). We now show that to compute Fibonacci numbers, the recursion can be eliminated altogether:

This time, we will maintain a list data structure, denoted `fibb`. Its elements will be `fibb[0], fibb[1], fibb[2], ..., fibb[n]` (n + 1 elements altogether for computing $F_n$).

Upon generating the list, all its values are set to 0. Next, we initialize the values `fibb[0]=fibb[1]=1`. And then we simply iterate, determine the value of the k-th element, `fibb[k]`, after `fibb[k-2]`, and `fibb[k-1]` were already determined.

No recursion implies no nested function calls, hence reduced overhead, and no need to confront Python's recursion depth limit!
Iterative Fibonacci Solution: Python Code

def fibonacci3 (n):
    """ iterative Fibonacci, employing memorization in a list """
    if n <2:
        return 1
    else :
        fibb = [0 for i in range (n +1)]
        fibb [0]= fibb [1]=1  # initialize
        for k in range (2,n +1):
            fibb [k] = fibb [k -1] + fibb [k -2]
            # update next element
        return fibb [n]
Recursive vs. Iterative: Timing

Let us now do some performance comparisons: `fibonacci2` vs. `fibonacci3`:

```python
>>> import sys
>>> sys.setrecursionlimit (20000)
>>> elapsed ( "fibonacci2 (2000)" )
0.007919692762108422
>>> elapsed ( "fibonacci3 (2000)" )
0.003019041286940194
>>> elapsed ( "fibonacci2 (2000)" )
0.00015984851784622833
>>> elapsed ( "fibonacci3 (2000)" )
0.003127113678182525
```

As we saw, `fibonacci2` runs faster the second time (when `fib_dict` has already been computed).
Finally: Iterative Fibonacci Solution Using \( O(1) \) Memory

No, we are not satisfied yet.

Think about the algorithm's execution ow. Suppose we have just executed the assignment

\[ \text{fibb}[4] = \text{fibb}[2] + \text{fibb}[3] \]

The entry \( \text{fibb}[4] \) will subsequently be used to determine \( \text{fibb}[5] \) and \( \text{fibb}[6] \). But then we make no further use of \( \text{fibb}[4] \). It just lies, basking happily, in the memory.

The following observation holds in "real life" as well as in the "computational world": Time and space (memory, at least a computer's memory) are important resources that have a fundamental difference:

Time cannot be re-used, while memory (space) can be.
Iterative Fibonacci Reusing Memory

At any point in the computation, we can maintain just two values, \( \text{fibb}[k-2] \) and \( \text{fibb}[k-1] \). We use them to compute \( \text{fibb}[k] \), and then reclaim the space used by \( \text{fibb}[k-2] \) to store \( \text{fibb}[k-1] \) in it.

In practice, we will maintain two variables, previous and current. Every iteration, those will be updated. Normally, we would need a third variable \text{next} \ for keeping a value temporarily. However Python supports the “simultaneous" assignment of multiple variables (first the right hand side is evaluated, then the left hand side is assigned).
Iterative Fibonacci Reusing Memory: Code

def fibonacci4(n):
    """fibonacci in O(1) memory """
    if n < 2:
        return 1  # base case
    else:
        previous = 1
        current = 1
        for i in range(n - 1):  # n -1 iterations (count carefully)
            current, previous = previous + current, current
        # simultaneous assignment
        return current

>>> for i in range(0, 7):
    print(fibonacci4(i))  # sanity check
1
1
2
3
5
8
13
Iterative Fibonacci Code, Reusing Memory: Performance

Reusing memory can surely help if memory consumption is an issue. Does it help with runtime as well?

```
>>> elapsed ( " fibonacci3 (10000) " ,number =100)
1.8180336248959341
>>> elapsed ( " fibonacci4 (10000) " ,number =100)
0.6884749629554126
>>> elapsed ( " fibonacci3 (30000) " ,number =100)
9.804906322193794
>>> elapsed ( " fibonacci4 (30000) " ,number =100)
4.1275352837865995
>>> elapsed ( " fibonacci3 (100000) " ,number =10)
7.492711482876487
>>> elapsed ( " fibonacci4 (100000) " ,number =10)
3.6488584185927238
```

We see that there is about 50-70% saving in time. Not dramatic, but significant in certain circumstances.

The difference has to do with different speed of access to different level cache in the computer memory. The `fibonacci4` function uses O(1) memory vs. the O(n) memory usage of `fibonacci3`. 
And to really conclude our Fibonacci excursion, we note that there is a closed form formula for the n-th Fibonacci number,

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}}$$

You can verify this by induction. You will even be able to derive it yourself, using generating functions (studied in the discrete mathematics course).
Closed Form Formula: Code, and Danger

def closed_fib (n):
    """ code for closed form Fibonacci number """
    return round (((1+5**0.5)** (n+1) - (1 -5**0.5)** (n+1) )/ (2**n *5**0.5))

>>> for i in range (1 ,6):
    print (i*10 , fibonacci4 (i*10) , closed_fib (i *10))      # sanity check
10 89 89
20 10946 10946
30 1346269 1346269
40 165580141 165580141
50 20365011074 20365011074

However, being aware that floating point arithmetic in Python (and other programming languages) has finite precision, we are not convinced, and push for larger values:

>>> for i in range (40 ,90):
    if fibonacci4 (i) != closed_fib (i):
        print (i, fibonacci4 (i), closed_fib (i))
        break

70 308061521170129 308061521170130                             Bingo!
Reflections: Memoization, Iteration, Memory Reuse

In the Fibonacci numbers example, all the techniques above proved relevant and worthwhile performance wise. These techniques won't always be applicable for every recursive implementation of a function.

Consider quicksort as a specific example. In any specific execution, we never call quicksort on the same set of elements more than once (think why this is true).

So memoization is not applicable to quicksort. And replacing recursion by iteration, even if applicable, may not be worth the trouble and surely will result in less elegant and possibly more error prone code.

Even if these techniques are applicable, the transformation is often not automatic, and if we deal with small instances where performance is not an issue, such optimization may be a waste of effort.
Not for the **Soft At Heart:**
the Ackermann Function

This recursive function, invented by the German mathematician Wilhelm Friedrich Ackermann (1896–1962), is defined as following:

\[
A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\
  A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 
\end{cases}
\]

This is a total recursive function, namely it is defined for all arguments (pairs of non-negative integers), and is computable (it is easy to write Python code for it). However, it is what is known as a non primitive recursive function, and one manifestation of this is its huge rate of growth.

You will meet the inverse of the Ackermann function in the data structures course as an example of a function that grows to infinity very very slowly.
Ackermann function: Python Code

Writing down Python code for the Ackermann function is easy -- just follow the definition.

```python
def ackermann(m,n):
    """Ackermann function"""
    if m==0:
        return n+1
    elif m>0 and n==0:
        return ackermann(m-1,1)
    else:
        return ackermann(m-1,ackermann(m,n-1))
```

However, running it with \( m \geq 4 \) and any positive \( n \) causes runtime errors, due to exceeding Python's maximum recursion depth. Even \( \text{ackermann}(4,1) \) causes such an outcome.
Ackermann function and Python Recursion Depth

However, running it with $m \geq 4$ and any positive $n$ causes runtime errors, due to exceeding Python's maximum recursion depth. Even `ackermann(4,1)` causes such a outcome.

However, even with this much larger recursion limit, say 100,000, only on Linux did `ackermann(4,1)` ran to completion (returning just 65,533, by the way). On an 8GB RAM machine running either MAC OSX 10.6.8 or Windows 7, it simply crashed (reporting "segmentation fault").

Adventurous? `ackermann(4,2)` will exceed any recursion depth that Python will accept, and cause your execution to crash. This is what happens with such rapidly growing functions.
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. (Sometimes they are called mutually recursive).

This definition generalizes to more functions as well. Here is an example – a very inefficient way to determine if a positive integer is even or odd.

```python
def even(x):
    if x==0:
        return True
    else:
        return odd(x-1)

def odd(x):
    if x==0:
        return False
    else:
        return even(x-1)
```
Recursion in Other Programming Languages

Python, C, Java, and most other programming languages employ recursion as well as a variety of other flow control mechanisms. By way of contrast, all LISP dialects (including Scheme) use recursion as their major control mechanism. We saw that recursion is often not the most efficient implementation mechanism.

Taken together with the central role of eval in LISP, this may have prompted the following statement, attributed to Alan Perlis of Yale University (1922-1990): "LISP programmers know the value of everything, and the cost of nothing".

In fact, the origin of this quote goes back to Oscar Wilde. In The Picture of Dorian Gray (1891), Lord Darlington defines a cynic as "a man who knows the price of everything and the value of nothing".
And Now to Something Completely Different: OOP

The sea was wet as wet could be,
The sands were dry as dry.
You could not see a cloud, because
No cloud was in the sky:
No birds were flying overhead -
There were no birds to fly.
The Walrus and the Carpenter
Were walking close at hand;
They wept like anything to see
Such quantities of sand:
"If this were only cleared away,"
They said, "it would be grand!"

Through the Looking-Glass
and What Alice Found There:
Lewis Carroll, 1871.
Object Oriented Programming (OOP)

OOP is a major theme in programming language design, starting with Simula, a language for discrete simulation, in the 1960s. Then 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.
Classes and Objects

We already saw that classes represent data types. In addition to the classes/types that are provided by python, programmers can write their own classes. A class is a template to generate objects. The class is a part of the program text. An object is generated as an instance of a class.

As we indicated, a class includes data attributes (fields) to store the information about the object, and methods to operate on them.
Line Class in Python

Line is a simple example of a class. (By convention, class names start with an upper case letter).

An object of the class Line has to represent a line in 2D plane. A line includes the points \((x,y)\) that satisfy \(y = ax + b\), for some \(a\) and \(b\). So we can represent the line by the slope \(a\) and offset \(b\). These will be the fields of the class line. So each line object will have its own values of slope and offset.

When a line is generated, we need to initialize it. We can choose different ways to do it. Here we decided that the programmer will supply two points in the 2D plane: \((x_1,y_1)\) and \((x_2,y_2)\). So when the line object is generated, we will have to compute and store the slope and the offset. The method that is used to create and initialize an object has a special name, `__init__`. 
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)

    point1 = (x1, y1)  # point1 is local to the method
    point2 = (x2, y2)  # point2 is local to the method

    if x1 != x2:
        self.slope = (y2-y1)/(x2-x1)  # assigns to field slope
        self.offset = y1 - x1 * (y2-y1)/(x2-x1)  # to field offset
    else:
        self.slope = "infty"  # assigns to field slope
        self.offset = x1  # assigns to field offset
Line Class, Additional Methods

assert evaluates its boolean argument and aborts if false.

__repr__ is another special method of the class (starts and ends with two _ symbols). It is used to describe how it is represented (when printing such an object).

Special methods have specific roles in any class in which they appear. We will see how they are called.

def __repr__(self):
    if self.slope == "infty":
        return "x=" + str(self.offset)
    else:
        return "y=" + str(self.slope) + "*x" + "+" + str(self.offset)
Line Class, Additional Methods (cont)

__eq__ is a special method that determines when two objects (in this case) lines are equal. is_parallel is a (non special) method that determines if two lines are parallel.

```python
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
```
Note on fields and parameters

The first parameter of every method represents the current object (an object of the class which includes the method). By convention, we use the name `self` for this parameter.

So the variable `self.slope` is the field named `slope` in the (current) object. Unlike other languages (eg. Java), we do not explicitly declare the names of the fields of the class we are defining. They exist because they are mentioned – initialized in the `__init__` method. Note that `point1` and `point2` are not fields. `other.slope` is the slope of the object pointed to by `other`.

We have seen that we call a function by its full name, including the name of the module (class) that includes it, for example `Line.is_parallel`. But we can also use the OO notation, which moves the first actual parameter (for `self`) to the position before the name of the method.
Line Class, Example Executions

```python
>>> a = Line(1., 1., 2., 2.)  # allocate, then call Line.__init__
>>> a                      # Line.__repr__(a) same as a.__repr__( )
y = 1.0*x + 0.0

>>> b = Line(2., 2., 5., 5.)

>>> a.slope  # accessing the field slope of the object
1.0  # that a points to

>>> a.offset
0.0

>>> b.slope
1.0

>>> b.offset
0.0
```
Line Class, Example Executions (cont.)

```python
>>> a==b        # __eq__ is called, same as a.__eq__(b)
True
>>> a is b
False
>>> a.is_parallel(b)  # same as 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
```
Further executions of Line

```python
>>> a = Line (1.0, 1.0, 2.0, 2.0)
>>> a
y = 1.0*x + 0.0
>>> a.slope
1.0
>>> a.slope = 2  # do we want to allow this?
>>> a
y = 2*x + 0.0  # the line has changed
```

What if we do not want to allow changes in the slope and offset of the line after it was created?
Information hiding

One of the principles of OOP is information hiding: The designer of a class should be able to decide what information is known outside the class, and what is not. In most OOP languages this is achieved by declaring fields and methods as either public or private.

In python, a field whose name starts with two _ symbols, will be private. It will be known inside the class, but not outside.

A private field cannot be written (assigned) outside the class, and its value cannot be read (inspected), because its name is not known.
A modified Line class

Suppose we change the names of slope and offset to __slope and __offset, respectively, throughout the class. Then we will get:

```python
>>> a = Line (1. ,1. ,2. ,2.)
>>> a
y = 1.0*x + 0.0
>>> a.__slope
Traceback (most recent call last):
  File "<pyshell#55>", line 1, in <module>
    a.__slope
AttributeError: 'Line' object has no attribute '__slope'
```

We wanted to block the ability to change the fields, but now their values are also hidden.
Private fields

If we want to let client code (code outside the class, that uses it) read the value of a private field, we need to write a method that will return the value. This is called a getter.

We can also write a method that assigns a value to the field (called a setter). Instead, we can write a method that changes the state of the object in some structured way (can change one or more fields, but only in very specific way).

Many OOP programmers choose to make all the fields of a class private, and provide methods for the desired operations.
Modified class Line

class Line1:
    # same as Line, except change field name slope to __slope, # and offset to __offset, and add the following two methods
    def slope(self):
        return self.__slope
    def offset(self):
        return self.__offset

Now run it
>>> a= Line1(1. ,1. ,2. ,2.)
>>> a.slope()
1.0
>>> a.slope
<bound method Line1.slope of y= 1.0*x + 0.0>
Modiﬁed class Line (cont.)

>>> a.__slope
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    a.__slope
AttributeError: 'Line1' object has no attribute '__slope'

>>> a.slope=2      # but we can replace the method!!

>>> a.slope()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    a.slope()
TypeError: 'int' object is not callable
Designing classes in OOP

The recommended way to design a class is to first decide what operations (methods) the class should support. This would be the API (or contract) between the class designer and the clients.

We often distinguish between:

• **Queries** – return a value, do not change the state
• **Commands** – change the state, do not return a value
• **Constructors** – create the object and initialize it

Then decide how to **represent** the state of objects (which fields), and make the fields private.

Then **implement** (write code for) the methods.

This way we can later change the representation (eg. change from Cartesian to Polar representation of points), while the **client code** is unchanged.
Python provides the basic ingredients for OOP, including inheritance (that we will not discuss).

However, we do not have the full safety that strict OOP languages have (as seen in the example). “Private” fields are accessible with mangled names, a client may add a field to an object, etc. In short, there is no way to enforce data hiding in python, it is all based on convention.

The language puts more emphasis on flexibility.

In this course we will usually use public fields and public methods to simplify the code, rather than adhere to OOP. This is the common style in python.

The course Software 1 (in Java) places OOP at the center.
class Student:
    def __init__(self):
        self.name = generate_name()  # function shown later
        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 + ' ')
The Student class has two fields: name, and id. These fields can be accessed directly, and values can be assigned to them directly.

The methods of the class are:

•  __init__ used to create and initialize an object in this class
•  __repr__ used to describe how an object is represented (when printing such an object).
•  __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.
•  Is_given_name, checks if the name field starts with the argument in word, followed by a blank.
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 )

Note:
random.randint (k,m) returns a random integer n, k≤ n ≤ m
random.sample(str,m) returns a list of length m whose elements are characters chosen at random from the string str
str.join(st,lst) concatenates the (string) elements of lst, and places between each two the string st.
Generating Names At Random: Python Code (cont)

```python
>>> 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

The students list that we used in the search examples

def students (n):
    return [ Student () for i in range (n)]
    # for each element, Student() generates a student
    # object and puts a reference to it in the list.

>>> 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]
Tinkering with the Student Class (cont)

```python
>>> for i in range (13):
    print ( students_list [i])
<Or , 39316939 >
<Yana , 52061841 >
<Amir , 49419212 >
<Roe, 40604275 >
<Noa , 24908823 >
<Gal , 56592426 >
<Barak , 29548638 >
<Rina , 52552066 >
<Tal , 57995311 >
<Lielle , 43513357 >
<Shady , 46042015 >
<Yuval , 32125900 >
<Walt Disney , 27907836 >
```
Student Class in Python: Example

```python
>>> students_list
[<Or, 39316939>, <Yana, 52061841>, <Amir, 49419212>,
 <Roe, 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
```
Testing Equality for the Student Class

```python
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
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, by defining a method `__eq__` as was done in the class Line. It will be called each time we test equality (==) of two Student objects.
Defining and Retesting Equality for the Student Class

class Student:
    def __init__(self):
        # as before, body omitted
    def __repr__(self):
        # as before, body omitted
    def __lt__(self, other):
        # as before, body omitted
    def is_given_name(self, word):
        # as before, body omitted
    def __eq__(self, other):
        return self.name == other.name \
            and self.id == other.id

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