Computer Science 1001.py, Lecture 21[†]

Linked Lists; Finite and Infinite Iterators
Introduction to Digital Images

Instructors: Benny Chor, Daniel Deutch

Teaching Assistants: Ilan Ben-Bassat, Amir Rubinstein, Adam Weinstock

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

[†] © Benny Chor.

Appeals' Procedure: Clarifications

- We have recently been bombed by chains of appeals:
- The same student submits an appeal on question 2, say.
- Then, later, a separate appeal on question 3.
- Then on question 1, etc.
- (Fortunately this process must end, as we got only a finite number of questions in each assignment.)

- As of this time, each of you can submit a single, written, well documented appeal per assignment, within the appeals period. (so this is applicable to HW 5, 6, 7.)
- In case of utmost need (you are sure the grader's response to your appeal is incorrect / not justified / contradicts your freedom of speech, etc.) you can approach your TA and lay your claims.

Lecture 20 Highlights

- Linked Lists
- Iterators and Generators

Lecture 21, Plan

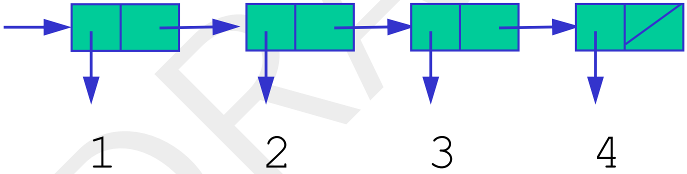
- Linked Lists, clarifications
- Iterators vs. Generators.
- Infinite iterators.
- Examples: Merging sorted iterators.

- Digital images: representation.

Linked Lists

An alternative to using a contiguous block of memory, is to specify, for each item, the memory location of the next item in the list.

We can represent this graphically using a boxes-and-pointers diagram:



Linked Lists vs. Regular Lists: Operations Complexity (reminder)

- Insertion after a given item requires O(1) time, in contrast to O(n) for regular lists.
- Deletion of a given item requires O(1) time, in contrast to O(n) for regular lists.
- Accessing the i-th item requires O(i) time, in contrast to O(1) for regular lists.
- Finding an item still requires O(n) time, but binary search is not applicable, even if the list is sorted (why?)

Perils of Linked Lists

With linked lists, we are in charge of memory management, and we may introduce cycles:

```
>>> L=linked_list()
>>> L.add_at_end(5)
>>> L.add_at_end(6)
>>> L.next.next.next=L
```

Can we check if a given list includes a cycle?

Detecting Cycles: First Variant

```python
def detect_cycle1(lst):
    p=lst.next
    dictio={}
    while True:
        if (p == None):
            return False
        elif (p.next in dictio):
            return True
        else:
            dictio[p.next]=1
            p = p.next
```

Note that we are adding the whole list element, both value and address of the next element (“box” in diagram) to the dictionary, and not just its contents.

Can we do it more efficiently? In the worst case we may have to traverse the whole list to detect a cycle, so O(n) time in the worst case is inherent. But can we detect cycles using just O(1) additional memory?

Detecting cycles: The Tortoise and the Hare Algorithm

```
def detect_cycle2(lst):
    """The hare moves twice as quickly as the tortoise
    Eventually they will both be inside the cycle
    and the distance between them will increase by 1 until
    it is divisible by the length of the cycle. """

    slow = fast = lst
    while True:
        if slow==None or fast==None:
            return False
        elif fast.next==None:
            return False
        slow = slow.next
        fast = fast.next.next
        # print("lst= ",id(lst),"slow= ",id(slow),"fast= ",id(fast)
        if (slow is fast):
            return True
```

Comment: We will (hopefully) see another manifestation of the tortoise and hare idea, in a completely different context: Pollard's ρ algorithm for factoring integers.

Detecting cycles: Execution

```
>>> L=linked_list()
>>> L.add_at_end(5)
>>> L.add_at_end(6)
>>> detect_cycle1(L)
False
>>> detect_cycle2(L)
False

>>> L.next.next.next=L
>>> detect_cycle1(L)
True
>>> detect_cycle2(L)
True
```

Detecting cycles: Execution, cont.

By commenting out the two print commands, we can explicitly watch the addresses of `lst, slow, fast`

```
>>> L=linked_list()
>>> L.add_at_end(5)
>>> L.add_at_end(6)

>>> detect_cycle2(L)
lst=  4356349200 slow=  4356349648 fast=  4356349712
False

>>> L.next.next.next=L

>>> detect_cycle2(L)
lst=  4356349200 slow=  4356349648 fast=  4356349712
lst=  4356349200 slow=  4356349712 fast=  4356349648
lst=  4356349200 slow=  4356349200 fast=  4356349200
True
```

Detecting cycles: A Weird Case

Last class, the following way of creating a cyclical linked list was attempted during the lecture.

```
>>> L=linked_list()
>>> L.add_at_end(5)
>>> L.add_at_end(6)
>>> L.add_at_end(L)
```

Running both cycle detection algorithms on this list produces `False`.

```
>>> detect_cycle1(L)
False
>>> detect_cycle2(L)
lst=  4356349904 slow=  4356349776 fast=  4356349840
lst=  4356349904 slow=  4356349840 fast=  4296614560
False
```

On the other hand, if we try to print this list, we run into an infinite loop.

Detecting cycles: A Weird Case, cont.

Is this a bug or a feature?

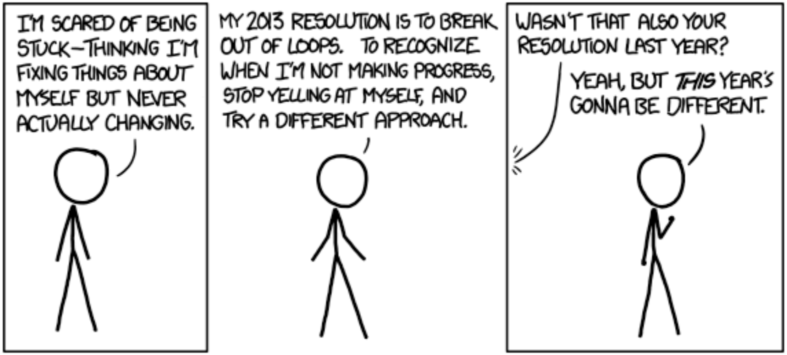
```
>>> L=linked_list()
>>> L.add_at_end(5)
>>> L.add_at_end(6)
>>> L.add_at_end(L)
```

What was created not a “cycle” in the standard sense. Our code follows the `.next` pointer, and it only looks for cycles created by going along these pointers.

So this is neither a bug, nor a feature. The code did what it was meant to do. We can definitely modify it so it does catch such scenarios, but we won't!

Life (and programming) are full of unexpected dangers, yet we try to avoid the dangers and fully enjoy both...

And Now to Something Completely Different: Iterators and Generators


I'M SCARED OF BEING STUCK—THINKING I'M FIXING THINGS ABOUT MYSELF BUT NEVER ACTUALLY CHANGING.
MY 2013 RESOLUTION IS TO BREAK OUT OF LOOPS. TO RECOGNIZE WHEN I'M NOT MAKING PROGRESS, STOP YELLING AT MYSELF, AND TRY A DIFFERENT APPROACH.
WASN'T THAT ALSO YOUR RESOLUTION LAST YEAR?
YEAH, BUT THIS YEAR'S GONNA BE DIFFERENT.

Source: http://xkcd.com/1154

Iterators and Generators

- Linked lists and Python's built-in lists (arrays) are two ways to represent a collection of elements. There are others, such as trees, dictionaries, and more. The same abstract data type (students, books,...) can be captured using any of these representations.
- It is desirable that functions that use the data as part of a computation should be as oblivious as possible to such internal representation, which may change over time.
- This general idea is captured in a concrete way by Python's iterators.
- Iterators will provide a generic access to a collection of items. So generic that it will even allow us to access an infinite collection (also known as stream)!
- Python's generators are tools to create iterators.

Iterables

An iterable is an object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`), some non-sequence types like `dict` and `file`, and objects of any user defined classes with an `__iter__()` or `__getitem__()` method.

(see http://docs.python.org/dev/glossary.html#term-iterator)

`range` is a special iterable class.

```
>>> a=range(10)
>>> type(a)
<class 'range'>
>>> a
range(0, 10)
>>> a[2]
2
```

Iterators (reminder)

An iterator is an object representing a stream of data. Repeated calls to the iterators `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a StopIteration exception is raised instead. At this point, the iterator object is "exhausted", and any further calls to its `__next__()` method just raise StopIteration exception again.

(see http://docs.python.org/dev/glossary.html#term-iterator)

```
>>> it=iter([0,1,2])
>>> next(it)
0
>>> next(it)
1
>>> next(it)
2
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    next(it)
StopIteration
```

Iterables and Iterators (reminder)

An iterable object (such as a `list`, `tuple`, `str`, `dict`, `range`, etc.) can be made into an iterator by calling the function `iter`. This function does not modify the original iterable object. In fact, when we loop over an iterable using `for`, an iterator is created first, and then the items are called, one by one, using `next()`.

```
>>> table={"benny":72,"rani":82,"raanan":92}
>>> next(table)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    next(table)
TypeError: dict object is not an iterator
>>> it=iter(table)
>>> next(it)
'rani'
>>> next(it)
'benny'
>>> next(it)
'raanan'
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    next(it)
StopIteration
```

Generators for Infinite Streams

Iterators and generators represent streams, but produce only one element at a time. Therefore, there is no problem representing a 2^{100} long stream.

In fact, there is no problem representing streams with countably many elements. To do that, we will introduce generator functions.

So far, our functions contained no state, or memory. Successive calls to the function with the same arguments produced the same results. This is now going to change.

```
def natural():
    """ a generator for all natural numbers """
    n=1
    while True:
        yield n
        n+=1
```

Generators for Infinite Streams, cont.

So far, our functions contained no state, or memory. Successive calls to the function with the same arguments produced the same results. This is now going to change.

```
def natural():
    """ a generator for all natural numbers """
    n=1
    while True:
        yield n
        n+=1
```

A function that contains a yield statement is termed a generator function. When a generator function is called, the actual arguments are bound to function–local formal argument names in the usual way, but no code in the body of the function is executed. Instead, a generator–iterator object is returned.

```
>>> natural()
<generator object natural at 0x16f60d0>
>>> Nat=natural()
>>> Nat
<generator object natural at 0x16f60a8>
```

Generators, cont.

```
>>> Nat=natural()
>>> Nat
<generator object natural at 0x16f60a8>
```

Nat is a generator–iterator. To get its “returned value”, which is specified by the yield statement, we invoke next.

```
>>> next(Nat)
1
>>> next(Nat)
2
>>> [next(Nat) for i in range(10)]
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

We see that Nat has a state, which is retained, unchanged, between successive calls.
We can have additional instances of the same generator function.

```
>>> Nat2=natural()
>>> next(Nat2)
1
>>> next(Nat)
13
```

A Fibonacci Numbers Generator

```
def fib():
    """ a generator for all Fibonacci numbers"""
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a+b

>>> Fib=fib()
>>> Fib
<generator object fib at 0x1704fa8>
```

Again, `Fib` is a generator–iterator, so to get its “returned value”, which is specified by the `yield` statement, we invoke `next()`.

```
>>> next(Fib)
1
>>> next(Fib)
1
>>> next(Fib)
2
>>> [next(Fib) for i in range(10)]
[3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

Execution Specification

If a yield statement is encountered, the state of the function is frozen, and the value of expression_list is returned to the caller of `__next__()`. By "frozen" we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time `next()` is invoked, the function can proceed exactly as if the yield statement were just another external call.

(see http://www.python.org/dev/peps/pep-0255/)

Merging Sorted, Infinite Iterators

Suppose iter1 and iter2 are sorted iterators, and both are infinite. We wish to produce a new sorted iterator which is the merge of both.

```
def merge(iter1,iter2):
    """ on input iter1, iter2, two infinite orted iterators,
    produces the sorted merge of the two iterators """

    left=next(iter1)
    right=next(iter2)
    while True:
        if left<right:
            yield(left)
            left=next(iter1)
        else:
            yield(right)
            right=next(iter2)

>>> Nat1=natural()
>>> Nat2=natural()
>>> Nat3=merge(Nat1,Nat2)
```

Merging Sorted, Infinite iterators: Execution

Nat3, too is a generator–iterator, so to get its “returned value”, which is specified by the yield statement, we invoke next.

```
>>> next(Nat3)
1
>>> next(Nat3)
1
>>> next(Nat3)
2
>>> next(Nat3)
2
>>> [next(Nat3) for i in range(10)]
[3, 3, 4, 4, 5, 5, 6, 6, 7, 7]
```

An Attempt to Merge Sorted, Finite iterators

Should the iterators in `merge` really be infinite?

```
>>> Nat1=natural()
>>> Nat2=(n-2 for n in range(3))
>>> Nat3=merge(Nat1,Nat2)
>>>
>>> next(Nat3)
-2
>>> next(Nat3)
-1
>>> next(Nat3)
0
>>> next(Nat3)
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    next(Nat3)
  File "/Users/benny/Documents/InttroCS2011/Code/intro17/lecture17.
    right=next(iter2)
StopIteration
```

What went wrong is that the merged iterator was not yet exhausted, yet one of the arguments to `merge`, `Nat2` was exhausted. The merging procedure still invoked `next(iter2)`. This has caused a StopIteration error.

Handling Errors: try and except

Python provides an elaborate mechanism to handle run time errors. For example, division by zero causes a ZeroDivisionError.

```
>>> 5/0
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    5/0
ZeroDivisionError: int division or modulo by zero
```

Such errors disrupt the flow of control in a program execusion. We may want to detect such error and allow the flow of control to continue. This may not be so important in the small programs written in this course, but becomes meaningful in large software projects. Python enables such detection, using the keywords try and except.

```
def division(a,b):
    try:
        return a/b
    except ZeroDivisionError:
        print("division by zero")
```

Handling Errors: try and except, cont.

```
def division(a,b):
    try:
        return a/b
    except ZeroDivisionError:
        print("division by zero")
```

Let us now apply this function in two different cases:

```
>>> division(5,6)
0.8333333333333334

>>> division(5,0)
division by zero
```

We will employ this error handling mechanism to enable merging any non-empty sorted iterators, finite or infinite.

Merging Any Non-Empty, Sorted iterators

```
def merge3(iter1,iter2):
    """ on input iter1, iter2, two non-empty sorted iterators, not
    necessarily infinite, produces sorted merge of the two iterators

    left=next(iter1)
    right=next(iter2)
    while True:
        if left<right:
            yield(left)
            try:
                left=next(iter1)
            except StopIteration:  # iter1 is exhausted
                yield(right)
                remaining=iter2
                break
        else:
            yield(right)
            try:
                right=next(iter2)
            except StopIteration:  # iter2 is exhausted
                yield(left)
                remaining=iter1
                break
    for elem in remaining:
        yield(elem)
```

Merge3: Examples of Executions

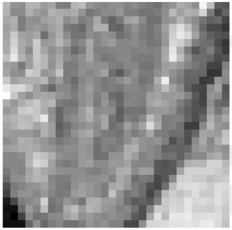
```
>>> iter1=(x**2 for x in range(4))
>>> iter2=natural()
>>> merged=merge3(iter1,iter2)

>>> [next(merged) for i in range(14)]
[0, 1, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10]

>>> iter1=(x**2 for x in range(5))
>>> iter2=(x**3 for x in range(6))
>>> merged=merge3(iter1,iter2)

>>> [next(merged) for i in range(11)]
[0, 0, 1, 1, 4, 8, 9, 16, 27, 64, 125]
```

Finally, lets see what happens when the original iterators/generators are not sorted.

```
>>> iter1=((-1)**x*x**2 for x in range(5))
>>> iter2=(x**3 for x in range(6))
>>> merged=merge3(iter1,iter2)

>>> [next(merged) for i in range(11)]
[0, 0, -1, 1, 4, -9, 8, 16, 27, 64, 125]
      # garbage in, garbage out
```

And Now to Something Completely Different: Digital Images Representation

Digital Images Representation



By now, we all know what this is a part of.

```
[183, 148, 143, 181, 178, 156, 165, 126, 123, 181, 189, 148, 139, 135, 142]
[178, 138, 138, 175, 158, 147, 179, 171, 168, 173, 147, 117, 127, 139, 139]
[168, 176, 123, 147, 142, 161, 165, 176, 140, 154, 125, 136, 169, 122, 99]
[161, 173, 127, 147, 154, 144, 161, 170, 122, 113, 105, 138, 177, 175, 104]
[164, 200, 162, 158, 167, 111, 151, 174, 140, 115, 117, 141, 133, 146, 140]
[170, 208, 171, 158, 204, 152, 158, 182, 159, 126, 138, 169, 134, 147, 157]
[171, 219, 170, 140, 199, 166, 143, 157, 134, 106, 123, 164, 129, 129, 133]
[192, 232, 180, 156, 200, 181, 149, 168, 140, 130, 144, 167, 135, 120, 125]
[160, 154, 122, 157, 194, 181, 145, 175, 122, 124, 141, 148, 144, 144, 128]
[165, 145, 140, 205, 197, 150, 157, 197, 124, 128, 144, 133, 145, 162, 111]
[199, 160, 172, 174, 175, 184, 136, 156, 125, 108, 135, 145, 133, 121, 129]
[234, 215, 218, 193, 159, 129, 104, 137, 135, 118, 141, 156, 135, 122, 126]
[254, 199, 160, 142, 163, 168, 163, 147, 140, 127, 144, 151, 127, 144, 109]
[173, 152, 173, 188, 199, 175, 182, 124, 117, 116, 141, 154, 122, 150, 126]
[154, 163, 200, 206, 197, 172, 142, 102, 124, 128, 155, 180, 138, 142, 139]
```

Brief “Historical” Context

At the early days of personal computers, say in the early 1980s, processors were relatively slow and quite expensive. Memory was even more expensive in relative terms.

Early e-mail (1970s to early 1980s) messages were plain ascii texts.

The situation is reflected by the following saying, often attributed (apparently incorrectly) to Bill Gates, in 1981:
“640KB ought to be enough for anybody”.

This was supposedly said when talking about IBM PC’s 640KB RAM, which was a significant breakthrough over the previous 8-bit systems that were typically limited to 64KB RAM.

A Brief Context, 30 Some Years Later

With the proliferation of strong, inexpensive processors, larger and faster RAMs, and especially of large, non-volatile memory chips (*e.g.* flash memory, commercialized from mid 1990s) it became possible to efficiently store, process, and transmit large digital images.

Facebook has stored 60 billion photos by the end of 2010. This number is expected to become 100 billion by summer 2011. (Consequently, it was called "the biggest image junk yard".)

By way of comparison, Photobucket stored 8 billion photos by 2010 end, Picasa 7 billion, and Flickr 5 billion. (source: this web page.)

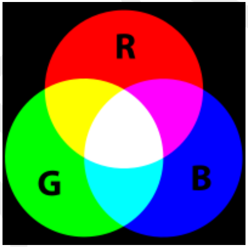
Basic Model of a Digital Image

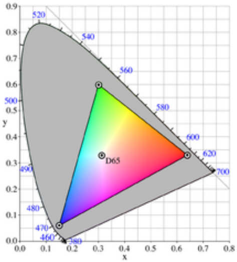
A digital image is typically encoded as a k-by-ℓ rectangle, or matrix, M, of either grey–level or color values.

For videos (movies), there is a third dimension, “time”. For each point t sampled in time, the frame at time t is nothing but a “regular” image.

Basic Model of a Digital Image, cont.

Each element $M[x, y]$ of the image is called a pixel, shorthand for picture element. For grey level images, $M[x, y]$ is a non negative real number, representing the light intensity at the pixel. For standard (RGB) color images, $M[x, y]$ is a triplet of values, representing the red, green, and blue components of the light intensity at the pixel.





(images from Wikipedia)

Grey Level Images

For the sake of simplicity, the remaining of this presentation will deal with grey scale images only. However, what we will do is applicable to color images as well.

To enable representation on bounded precision, digital devices, real numbers expressing grey levels have to be discretized.

A good quality photograph (that is, good by human visual inspection) has 256 grey-level values (8 bits) per pixel.,The value 0 represents black, while 255 represents white (not very intuitive, I agree :-).

For each pixel, the closer its value is to 0, the blacker it is. So 128 is a perfect grey.

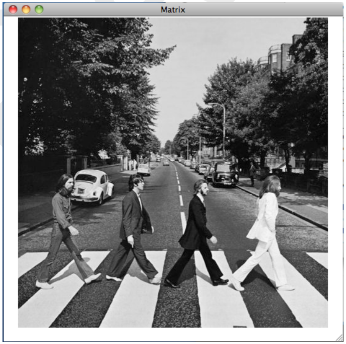
We remark that in some applications, such as medical imaging, 4096 grey levels (12 bit) are used.

Loading and Displaying Images

Our matrix.py package has (rather simple) methods for loading a digital image (making it into an instance of the Matrix class and displaying them.

```
m=Matrix.load("abbey_road.bitmap")
>>> m=Matrix.load("abbey_road.bitmap")
>>> m.display()
```

This opens a new, graphical window.
To run additional code in the shell, this window has to be closed first.



Loading and Displaying Images, cont.

In the display method, we can zoom in (but zoom factor has to be an integer)

```
>>> m=Matrix.load("abbey_road.bitmap")
>>> m=Matrix.load("abbey_road.bitmap")
>>> m.display(zoom=3)
>>> m.display(zoom=2)

>>> m.display(zoom=1.5)
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    m.display(zoom=1.4)
  File "/Users/benny/Dropbox/Eshnav2012/Code/Eshnav10/matrix.py", l
    tl = tk_worker(root)
  File "/Users/benny/Dropbox/Eshnav2012/Code/Eshnav10/matrix.py", l
    pi = pi.zoom(zoom)
  File "/Library/Frameworks/Python.framework/Versions/3.2/lib/pytho
    self.tk.call(destImage, 'copy', self.name, '-zoom',x,y)
_tkinter.TclError: expected integer but got "1.5"
```

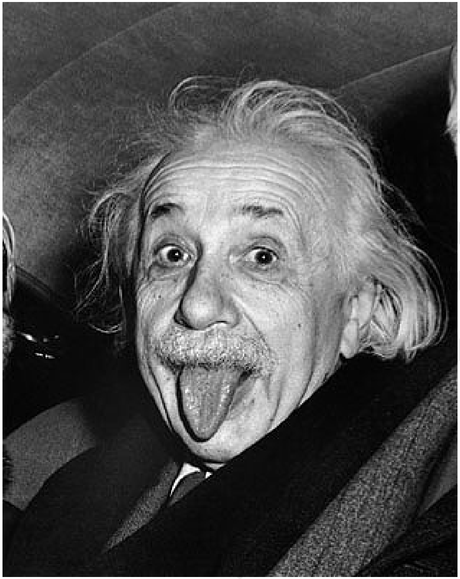
Grey Level Images - Another Example

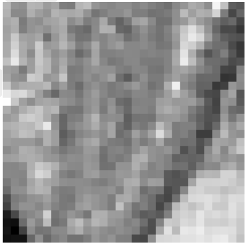
```
>>> Albert=Matrix.load("albert-einstein-1951.bitmap")
>>> Albert.display(zoom=2)
>>> Albert.display()

>>> Tongue=Albert[260:300,130:160]
        # a slice of the original

>>> T.display(zoom=6)
>>> Tongue.dim()
(40, 30)
```

The Tongue, in Numbers

```
>>> Tongue=Albert[260:300,130:160]
        # a slice of the original

>>> T.display(zoom=6)
>>> Tongue.dim()
(40, 30)

>>> for i in range(40):
print([T[i,j] for j in range(30)])
```

A Grey Level Image and a Slice Thereof





Tinkering with a Real Image: Lena

```
for i in range (100):
    for j in range (100):
        Lena[i,j ]=0

Lena.display()
# black square at upper left corner
```

(this operator is applicable to any grey level image whose dimensions are at least 512-by-512.)

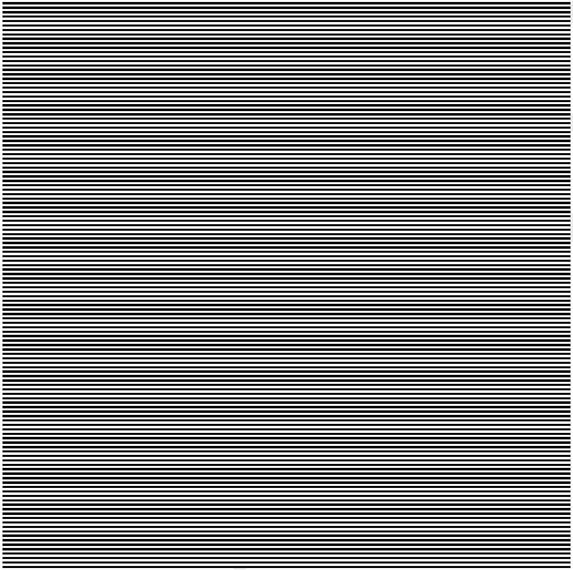
Tinkering with a Real Image: Lena, cont.

```
for i in range (200,300):
    for j in range (200,300):
        Lena[i,j ]=128

Lena.display()


for i in range (412,512):
    for j in range (412,512):
        Lena[i,j ]= 255

Lena.display()

# black square at upper left corner
# grey square at middle
# white square at lower right corner
```

Simple Synthetic Images: Lines and More

```
horizontal_lines=Matrix(512,512)

for x in range(412):
    if x % 2 == 0:
        for y in range(512):
            horizontal_lines[x,y]=255

horizontal_lines.display()

horizontal_lines.display(zoom=2)
```

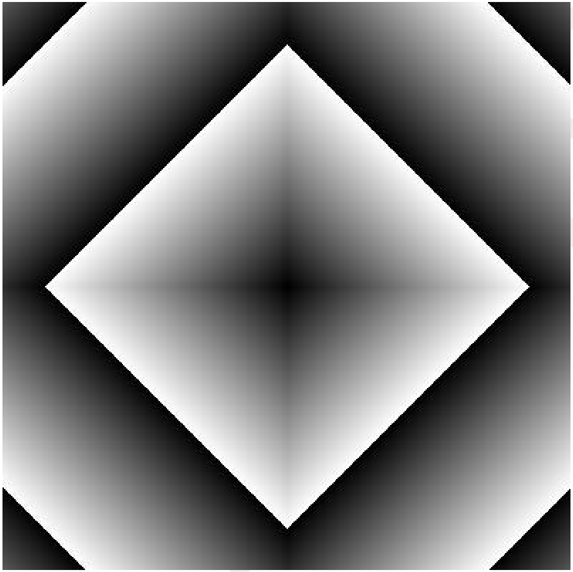
Displaying Synthetic Images: Lines and More



Simple Synthetic Images: Lines and More

```
abs_cont=Matrix(512,512)

for x in range(512):
       for y in range(512):
           abs_cont[x,y]=1.5*(abs(x-256)+abs(y-256))

abs_cont.display()
abs_cont.display(zoom=2)


for x in range(512):
       for y in range(512):
           abs_cont[x,y]=1.5*(abs(x-256)+abs(y-256))  % 256

abs_cont.display()
abs_cont.display(zoom=2)
```

Displaying Synthetic Images: Lines and More



Simple Synthetic Images: Scaling Lines

```
A= Matrix(512,512)
B= Matrix(512,512)
C= Matrix(512,512)

for i in range (512):
   for j in range (512):
      A[i,j]=(i+j -512) % 256

A.display()

for i in range (512):
   for j in range (512):
      B[i,j]=2*(i+j -512)  % 256

B.Display()

for i in range (512):
   for j in range (512):
      C[i,j]=4*(i+j -512)  % 256

C.display()
```

Simple Synthetic Images: Circles and More

```
Circ256=Matrix(512,512)
for i in range (512):
   for j in range (512):
      Cirtc256[i,j]=((i -256)**2+(j -256)**2)//256
Circ256.display()

Circ16=Matrix(512,512)
for i in range (512):
   for j in range (512):
      Circ16[i,j]=((i -256)**2+(j -256)**2)//16
Circ16.display()

Circ4=Matrix(512,512)
for i in range (512):
   for j in range (512):
      Circ4[i,j]=((i -256)**2+(j -256)**2)//4
Circ4.display()
```

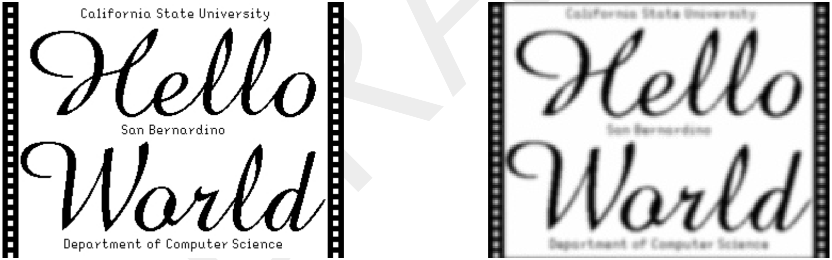
We urge you to try these (and other) functions by yourself.

Simple Synthetic Images: Miscellaneous

```
import sys
import math
import random
import cmath   # complex numbers

A=Matrix(512,512)
B= Matrix(512,512)
C= Matrix(512,512)

for i in range (512):
   for j in range (512):
      A[i,j]=(math.sin ((i-256)**2 + (j-256)**2 )* 16) % 256
A.display()

for i in range (512):
   for j in range (512):
      B[i,j]=256*math.sin(25*cmath.phase(complex(i-256,j-256))) % 2
B.display

for i in range (512):
   for j in range (512):
      C[i,j]=random.randint(0,255)

C.display()
```

Blur

The two major effects hampering image accuracy are termed blur and noise. Blur is an intrinsic phenomenon to digital image acquisition, resulting from limits on sampling rates. Blur has the effect of reducing the image's high-frequency components. To really understand it, a non negligible knowledge about signal processing is required. It is completely outside the scope of this course (and, unfortunately, of general CS studies as well).

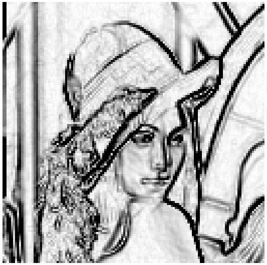


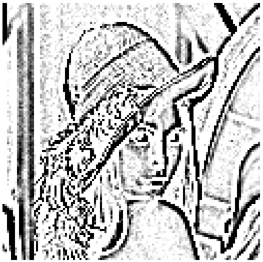
An original image (left) and a blurred version thereof (right). Taken from Wikipedia (which ran out of the "gidday mate" version).

Edges

Images of interest are usually not completely smooth. While most areas are smooth, parts of images exhibit sharp changes in intensity from one pixel to the next. These boundaries are termed edges, and often capture much of the meaningful information in an image.

The problem of edge detection is a central problem in image processing, and many algorithms attempt to solve it.

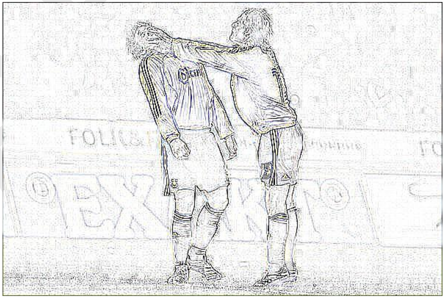




An original image (of the world famous Lena) (center), and the results of two different edge detection algorithms (Sobel, left and Laplacian, right). Images taken from http://www.pages.drexel.edu/∼weg22/edge.html.

Built in Filters in Python Image Library (PIL)

The PIL package has a number of built in filters. For example, it has a contours finding filter, whose result is shown here.



(see http://www.riisen.dk/dop/pil.html)