Extended Introduction to Computer Science  
CS1001.py

Lecture 17: Finite and Infinite Iterators  
Introduction to Digital Image Processing

Instructors: Amir Rubinstein, Amiram Yehudai  
Teaching Assistants: Yael Baran, Michal Kleinbort

Founding instructor: Benny Chor

School of Computer Science  
Tel-Aviv University  
Winter Semester, 2015-16  
http://tau-cs1001-py.wikidot.com
Lecture 16 Highlights

- The **dictionary** problem (find, insert, delete).
- Python's hash.
- Hash functions and hash tables.
- The 8 queens problem
Lecture 17, Plan

• Iterators.
• Lazy (delayed) evaluation.
• Infinite iterators.
• Examples: Merging sorted iterators.
• Handling Errors: try and except

• Introduction to Digital Image representation.
• Generating synthetic images

Next time
• Basics of Digital Image Processing.
• Noise, and local noise reductions
• Segmentation
And Now to Something Completely Different: Finite and Infinite Iterators
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.
• 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-iterable](http://docs.python.org/dev/glossary.html#term-iterable))

range is a special iterable class.

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

An iterator is an object representing a stream of data. Repeated calls to the built-in function next(it), where it is an iterator (or calls to the iterator’s __next__() method) 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 next(it) just raise StopIteration exception again. (see http://docs.python.org/dev/glossary.html#term-iterator)

```python
>>> 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

We can create an iterator by calling the function iter with an iterable object argument (like list, tuple, str, dict, range, etc.) 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().

```python
>>> 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'
```
Iterables and Iterators, cont.

```python
>>> next(it)
'raanan'
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    next(it)
StopIteration

>>> table={"benny":72,"rani":82,"raanan":92}
>>> for key in table:               # an iterator is created
      print(key)                         # "under the hood"
          # more details later
      rani
      benny
      raananan
```
Iterables and Iterators, cont.

As we see from this example, a dictionary (when transformed into an iterator), returns the keys one by one. Files return the lines one by one, etc.

We can turn an iterator into a list as well. This list will reflect the current state of the iterator, not its original state.

```python
>>> table={"benny":72,"rani":82,"raanan":92}
>>> it=iter(table)
>>> next(it)
'rani'
>>> list(it)
['benny', 'raanan']
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
    next(it)
StopIteration
```
Thou Shalt Not Modify an iterable during Iteration

If we add or remove elements from an iterable during iteration, strange things may happen. For example

```python
>>> elems = ['a','b','c']
>>> for e in elems:
    print(e)
    elems.remove(e)

a
c
```

```python
>>> elems
['b']
```

adapted from

http://unspecified.wordpress.com/2009/02/12/thou-shalt-not-modify-a-list-during-iteration/
Iterators as a Tool for Abstraction

The use of iterators hides the implementation of data collections. For example, when we see the code

```
for x in SomeCollection:
    ....
```

We do not know if SomeCollection is a list, a dict, or any user defined data collection. Furthermore, we can later modify the implementation of SomeCollection, for example change it from a list to a dict, and the code using it (called client code) will not have to be changed.

Similarly when we use `next(it)`, it may be an iterator of any kind of a data collection, with any order of traversal.
Iterables, Iterators, and Generators: More Examples

```python
>>> mylist=[x for x in range(10**8)]
>>> it1=iter(mylist)
>>> it2=(x for x in range(10**8)) # note the () instead of []

>>> type(mylist)
<class 'list'>
>>> type(it1)
<class 'list_iterator'>
>>> type(it2)
<class 'generator'> # list comprehension syntax inside () is one way to create a generator. The other will be shown next

>>> # mylist
    # typing this without the comment will clobber your screen
    # and most likely will cause your Python shell to crash
>>> it1
<list_iterator object at 0x17027d0>
>>> it2
<generator object <genexpr> at 0x1704f08>
```
iter1 and iter2 were iterators representing the first $10^8$ integers. These integers can fit in just under 1GB RAM. But, can we have iterators representing even more items?

```python
>>> it3=(x for x in range(2**100))
```

```python
>>> next(it3)
0
```

```python
>>> next(it3)
1
```

An iterable with $2^{100}$ elements will not fit in Amazon, Google, and NASA computers, even if taken together.

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

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 (assuming the function does not refer to a global variable, which may have changed). This is now going to change.

```python
def natural():
    """ a generator for all natural numbers """
    n=1
    while True:
        yield n
        n+=1
```
Generators for Infinite Streams, cont.

A function that contains a yield statement is termed a generator function. When a generator function is called, the actual arguments are bound to the function’s 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.

```python
>>> natural()
<generator object natural at 0x16f60d0>
>>> Nat=natural()
>>> Nat
<generator object natural at 0x16f60a8>
```
Generators, cont.

Nat is a generator--iterator. To get its "returned value", which is specified by the \texttt{yield} statement, we invoke \texttt{next}.

\begin{verbatim}
>>> next(Nat)
1

>>> next(Nat)
2

[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
\end{verbatim}

We see that Nat has a \texttt{state}, which is retained, unchanged, between successive calls.

We can have additional instances of the same generator function.

\begin{verbatim}
>>> Nat2=natural()

>>> next(Nat2)
1

>>> next(Nat)
13
\end{verbatim}
Lazy Evaluation

In programming language theory, lazy evaluation or call-by-need is an evaluation strategy, which delays the evaluation of an expression until its value is actually required, and also avoids repeated evaluations by memoization (caching).

The "opposite" of lazy actions is eager evaluation, sometimes known as strict evaluation. Eager evaluation is the evaluation behavior used in most cases in most programming languages.
Lazy Evaluation (cont.)

Python's iterators and generators employ lazy evaluation. The next item is evaluated only when it is required, by means of executing `next()`. We remark that it would not be possible to handle finite but very large iterators/generators, or infinite iterators/generators, without the lazy evaluation mechanism.

Good old Scheme has a special syntax, enabling the delay and force of an evaluation of an expression. These make the use of very large and of infinite streams in Scheme possible.
Eager evaluation

We have been using eager evaluation all the time and did not know it!

Similar to
Le Bourgeois Gentilhomme, by Molière:
“Good heavens! For more than forty years I have been speaking prose without knowing it.”

(Quotations taken from Wikipedia)
A Fibonacci Numbers Generator

```python
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()` .

```python
>>> 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

When a `yield` statement

```
yield expression_list
```

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

This is similar to what is called co-routine, in contrast with a normal function call which is also called sub-routine.
Merging Sorted, Infinite Iterators

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

```python
def merge(iter1, iter2):
    """ on input \texttt{iter1, iter2}, two infinite sorted 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)
```

23
Merging Sorted, Infinite iterators: Execution

```python
>>> nat1=natural()
>>> nat2=natural()
>>> nat3=merge(nat1,nat2)
nat3 , too is a generator--iterator, so to get its “returned value", which is specified by the `yield` statement, we invoke `next` .

```python
>>> next(nat3)
1
```python
>>> next(nat3)
1
```python
>>> next(nat3)
2
```python
>>> next(nat3)
2
```python
>>> [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?

```python
>>> 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/IntroCS2011/Code/intro17/lecture17.py", line 30, in merge
    right=next(iter2)
StopIteration
```

What went wrong?
The merged iterator, `nat3`, 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.

```python
>>> 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 execution. 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**.
Handling Errors: **try** and **except**: example

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

Let us now apply this function in two different cases:

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

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

We will employ this error handling mechanism to enable merging any non-empty sorted iterators, finite or infinite.
More on **try and except**

The example in the previous slide is not so good – we can solve this problem with an **if** statement. The following example shows a situation where we would need to write many if statements, so **try/except** is better.

```python
def compute(...):
    try
        # a long computation, with several steps
        # that may cause zero divide
    except ZeroDivisionError:
        print("division by zero")
```

We will also use **try/except** when it is either **impossible** or **expensive** to check for the condition in advance. Example – when we invert a matrix, checking in advance that it is not singular would take as much time as inverting, so it makes more sense to try to invert, and raise an **exception** if we discover that the matrix is singular while we do it.

We can have multiple except clauses; a list of exceptions to be handled in each clause; and the last clause may omit exception names (to handle all others)
For loop

We mentioned that a for loop over an iterable using for, actually uses an iterator. We now show an example:

```python
elems = ['a', 'b', 'c']

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

Is the same as

```python
it = iter(elems)
while True:
    try:
        print(next(it))
    except StopIteration:
        break
```
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
    # end of the while loop
    for elem in remaining:
        # protects against StopIteration
        yield(elem)
Merge3: Examples of Executions

```python
>>> 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, let's see what happens when the original iterators/generators are not sorted.
```python
>>> 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, 9, 16, 27, 64, 125]
```

# garbage in, garbage out
And Now to Something Completely Different:
Introduction to Digital Image Processing
Digital Images Representation

Any guesses as to what this image is (or is part of)?

[100 112 88 ..., 134 145 166]
[ 80 132 134 ..., 130 184 158]
[ 44 51 56 ..., 132 122 9]
...
[ 14 17 15 ..., 206 204 184]
[ 21 11 12 ..., 203 176 185]
[ 24 13 16 ..., 200 180 182]
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, 35 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 stores about 350 million photos DAILY (reported September 2013). 1.1 billion photos were uploaded on 2013 New Years Eve.

The total number of photos shared on Instagram is 16 billion. On average, 55 million photos are posted daily (reported December 2013). (Instagram was launched on October 2010 !!).

On Flickr the average upload of images per MONTH in 2012 was about 43 million.
Basic Model of a Digital Image

A digital image is typically encoded as a \( k \)-by- \( \ell \) 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 nonnegative 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 remainder of this presentation will deal with grey scale images only. However, what we will do is applicable to color images as well. Real numbers expressing grey levels have to be discretized in order to enable their representation on bounded precision digital devices.

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 bits) are used.
class Matrix:

    ''' Represents a rectangular matrix with n rows and m columns. '''

def __init__(self, n, m, val=0):
    ''' Create an n-by-m matrix of val's.
    Inner representation: list of lists (rows) '''
    assert n > 0 and m > 0
    self.rows = [[val]*m for i in range(n)]

    def dim(self):
        return len(self.rows), len(self.rows[0])

    def __repr__(self):
        if len(self.rows) > 10 or len(self.rows[0]) > 10:
            return "Matrix too large, specify submatrix"
        return "<Matrix {}>".format(self.rows)

def __eq__(self, other):
    return isinstance(other, Matrix) and self.rows == other.rows
class Matrix - Indexing

# cell/sub-matrix access/assignment

```python
def __getitem__(self, ij):
    #ij is a tuple (i,j). Allows m[i,j] instead m[i][j]

    i, j = ij

    if isinstance(i, int) and isinstance(j, int):
        return self.rows[i][j]

    elif isinstance(i, slice) and isinstance(j, slice):
        M = Matrix(1,1)  # to be overwritten
        M.rows = [row[j] for row in self.rows[i]]
        return M

    else:
        return NotImplemented
```

def __setitem__(self, ij, val):
    #ij is a tuple (i,j). Allows m[i,j] instead m[i][j]
    i, j = ij
    if isinstance(i, int) and isinstance(j, int):
        assert isinstance(val, (int, float, complex))
        self.rows[i][j] = val
    elif isinstance(i, slice) and isinstance(j, slice):
        assert isinstance(val, Matrix)
        n, m = val.dims()
        s_rows = self.rows[i]
        assert len(s_rows) == n and len(s_rows[0][j]) == m
        for s_row, v_row in zip(s_rows, val.rows):
            s_row[j] = v_row
    else:
        return NotImplemented
class Matrix - item access and assignment

```python
>>> m = Matrix (10, 10)  # 10 x 10 matrix of zeros
>>> m[4, 5]  # same as m.__getitem__((4,5))
0
>>> m[4, 5] = 45  # same as m.__setitem__((4,5),45)
>>> m[4, 5]
45
```

Note: the code in the matrix.py file contains an additional feature: accessing and assignment of a whole slice.

```python
>>> m[3:5, 4:8]
<Matrix [[0, 0, 0, 0], [0, 45, 0, 0]]>
```
The .bitmap image file format

There are several common image file formats, such as jpg, bmp, tiff, png, etc. Each format represent images somewhat differently. In particular, most of them employ lossy compression to save on memory.

We will be using our own format in this course: the .bitmap format.

\[
\begin{align*}
M_{0,0} & \quad M_{0,1} & \quad M_{0,2} & \quad \ldots & \quad M_{0,m-1} \\
M_{1,0} & \quad M_{1,1} & \quad M_{1,2} & \quad \ldots & \quad M_{1,m-1} \\
\vdots & \quad \vdots & \quad \vdots & \quad \ddots & \quad \vdots \\
M_{n-1,0} & \quad M_{n-1,1} & \quad M_{n-1,2} & \quad \ldots & \quad M_{n-1,m-1}
\end{align*}
\]

Our matrix.py package has (rather simple) methods for loading and saving digital images in this format.
The .bitmap image file format

Our matrix.py package has (rather simple) methods for loading and saving digital images in this format.

# load image into an instance of the Matrix class
>>> m = Matrix.load("some_image.bitmap")
# save an instance of Matrix class into an image in .bitmap format
>>> m. save("my_copy.bitmap")

Use ./ for the current directory, and ../ to get to the parent directory:

>>> m = Matrix.load ("./ some_image.bitmap")
>>> m = Matrix.load( "../../some_image.bitmap" )
Converting to and from the .bitmap format

Our .bitmap format employs no compression, thus it requires large amount of storage, and consequently is not used outside this course. Images in this format are provided in the course website, so you can work directly with them.

If you are interested in converting "real" images to and from our .bitmap format, you can use the functions `image2bitmap` and `bitmap2image`, which are both part of the `images.py` file.

To run them, you will need to first install the PIL (Python's Image Library) package (see next slide).

But you will not need this for the homework.
Installing PIL – for reference only

As of python version 3.4, python packages can be installed using the following command:
python -m pip install <PackageName>

For installing PIL, use
python -m pip install pillow

(in Windows do:
    Accessories => Command Prompt,
    cd <Python directory>
    python -m pip install pillow
)

More details are available here:
https://docs.python.org/3.4/installing/

Note that if multiple Python versions are installed on your computer, the installation command is slightly different.
Converting to and from the .bitmap format, cont.

We provide these two functions to do the conversions, so we can work with "real" images:

```python
def image2bitmap (path)
...
def bitmap2image (path)
...```

```python
>>> image2bitmap ("./real_image.jpg")
# This creates a file with the same name in .bitmap format
>>> im = Matrix.load("./real_image.bitmap")
>>> ...
>>> # manipulate the image
>>> bitmap2image("./real_image.bitmap")
# This created a file with the same name in .bmp format
```

You do not need to understand the fine print of these conversions (although they are rather simple). Now you'll be able to impress your friends and family with your own homemade mini version of photoshop!
Displaying .bitmap Images

Our matrix.py package also has a method for displaying the image it represents.

```python
>>> 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. Again, you do not need to understand the fine print of how the display() method works.
In the display method, we can zoom in. Note that the zoom factor has to be a positive integer.

```python
>>> m = Matrix.load("abbey_road.bitmap")
>>> m.display(zoom=3)
>>> m.display(zoom=2)
>>> m.display(zoom=1.5)

>>> Exception in thread Thread-1:
Traceback (most recent call last):
  File "C:\Python34\lib\threading.py", line 921, in _bootstrap_inner
    self.run()
  File "C:\Users\amiram\Documents\amiram\IntroCS\introFall2015\lec17Amiram\code\matrix.py", line 201, in run
    func(root)
  File "C:\Users\amiram\Documents\amiram\IntroCS\introFall2015\lec17Amiram\code\matrix.py", line 155, in tk_worker
    pi = pi.zoom(zoom)
  File "C:\Python34\lib\tkinter\__init__.py", line 3437, in zoom
    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

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

If we want to see the numbers:
>>> for i in range (40):
    print ([[ tongue [i,j] for j in range (30)])
Tinkering with an Image

```python
>>> m = Matrix.load("albert-einstein-1951.bitmap")
>>> bs = black_square(m)
>>> bs.display()

>>> m = Matrix.load("abbey_road.bitmap")
>>> ts = three_squares(m)
>>> ts.display()
```
def copy(mat):
    """ brand new copy of matrix """
    n, m = mat.dim()
    new = Matrix(n, m)
    for i in range(n):
        for j in range(m):
            new[i, j] = mat[i, j]
    return new

def black_square(mat):
    """ add a black square at upper left corner """
    n, m = mat.dim()
    if n<100 or m<100:
        return None
    else:
        new = copy(mat)
        for i in range(100):
            for j in range(100):
                new[i, j] = 0
        return new
def three_squares(mat):
    ''' add a black square at upper left corner, grey at middle, and white at lower right corner'''
    n, m = mat.dim()
    if n<500 or m<500:
        return None
    else:
        new = copy(mat)
        for i in range (100):
            for j in range (100):
                new[i,j] = 0  # black square
        for i in range (200,300):
            for j in range (200,300):
                new[i,j] = 128  # grey square
        for i in range (400,500):
            for j in range (400,500):
                new[i,j] = 255  # white square
        return new
Simple Synthetic Images: Lines and More

```python
def horizontal():
    horizontal_lines = Matrix(512,512)
    for i in range(512):
        if i % 2 == 0:
            for j in range(512):
                horizontal_lines[i,j] = 255
    return horizontal_lines

>>> a = horizontal()
>>> a. display()
>>> a. display(zoom=2)
```
Displaying Synthetic Images: Lines and More
Simple Synthetic Images: Diagonal Lines

def diagonals(c=1):
    surprise = Matrix(512,512)
    for i in range(512):
        for j in range(512):
            surprise[i,j] = (c*(i+j)) % 256
    return surprise

>>> a = diagonals ()
>>> a. display ()
>>> a = diagonals (c =3)
>>> a. display ()
Simple Synthetic Images: Product and Circles

```python
def product(c=1):
    surprise = Matrix(512,512)
    for i in range(512):
        for j in range(512):
            surprise[i,j] =
                (c*((i-256)*(j-256)))% 256
    return surprise

>>> a = product ()
>>> a. display ()
>>> a = product (c =2)
>>> a. display ()
```
def circles(c=1):
    surprise = Matrix(512,512)
    for i in range(512):
        for j in range(512):
            surprise[i,j] = round(c*((i-256)**2+(j-256)**2))% 256
    return surprise
def synthetic(n,m,func):
    ''' produces a synthetic image "upon request"
    '''

    new=Matrix(n,m)
    for i in range(n):
        for j in range(m):
            new[i,j]= func(i,j) % 256

    return new
Simple Synthetic Images: Miscellaneous

```python
>>> a = synthetic (512,512,lambda x,y: random.randint(0,255))
>>> a. display ()
>>> b = synthetic (512 , 512 , lambda x,y: \
    math . sin (16*((x -256)**2 + (y -256)**2)))
>>> b. display ()
>>> c = synthetic (512 , 512 , lambda x,y: \
    56* math . sin (32* cmath . phase ( complex (x -256 ,y -
    256)))))
>>> c. display ()
```

We urge you to try these (and other) functions by yourself.
Image processing is any form of signal processing for which the input is an image, such as a photograph or video frame; the output of image processing may be either an image or a set of characteristics or parameters related to the image. Most image-processing techniques involve treating the image as a two-dimensional signal and applying standard signal-processing techniques to it.

(text and figure taken from Wikipedia).
Digital Image Processing

- Signal processing for which the input is an image
- **Common problems:**
  - Noise reduction (denoising) - removing noise from an image.
  - Segmentation - partitioning a digital image into segments *(e.g. identify the boundaries of cells in a multi-cell image)*
  - Tracking - relate objects in subsequent frames of a film
  - Edge detection – detecting discontinuities in the image
  - Registration - transforming different images into one coordinate system *(e.g. minor shifts in the camera position in subsequent frames)*
  - Color correction.
- **Typical applications:**
  - Machine vision
  - Medical / biological image analysis
  - Face detection
  - Object recognition
  - Augmented reality
  - …
Capturing Images

Consider a specific pixel with coordinates $x, y$. Suppose $I(x, y)$ is the "true" value at pixel $x, y$. This is the value which would be observed by averaging the photon count on a long period of time, assuming the image source is constant over time.

The observed value, $S(x, y)$, is the result of the light intensity measurement, usually made by a CCD (charge coupled device, transforming light to electrical voltage) matrix, together with an optical light focusing system (lens or lenses). Each captor of the CCD is roughly a square area, in which the number of incoming photons is being counted for a fixed period corresponding to the obturation time.
CCD and DSP

Digital signal processing (DSP) takes the raw data from the sensor and assembles it in correct color space or bitmap structure. In doing so, it handles white balance, brightness, sharpness, contrast and noise levels.

(image and text taken from http://www.axis.com/edu/axis/)