

Python tutorial for analytic element developers

Mark Bakker, UGA, Athens, GA. mbakker@engr.uga.edu

Vic Kelson, WHPA, Bloomington, IN. vic@wittmanhydro.com

Willem Zaadnoordijk, Royal Haskoning, Rotterdam, NL. wj.zaadnoordijk@royalhaskoning.com

January 24, 2005

Contents

1	Introduction	1
2	Installation of Python	1
3	AEM tutorial	2
3.1	A simple well class	2
3.2	Adding the potential method	3
3.3	Adding the Aquifer class	4
3.4	Adding the Circle class and inheritance	6
3.5	Pythondoc	7
3.6	Solving for unknown strengths	9

1 Introduction

This document provides a tutorial for analytic element developers. The tutorial is meant for users of *Tim* an object-oriented computer program for the simulation of groundwater flow with analytic elements.

2 Installation of Python

The tutorial requires the installation of Python 2.1 or higher, which may be obtained from the Python website <http://www.python.org>. For *Tim*, the Numeric package NumPy and SciPy must also be installed. They may be obtained from the Numeric website <http://sourceforge.net/projects/numpy/> and the SciPy website <http://www.scipy.org>, respectively.

It is also possible to use the Enhanced Python distribution which has been made available by Enthought <http://www.enthought.com/python/>. It contains a lot of extras and therefor is quite large (some 84 Mbytes).

3 AEM tutorial

A small analytic element program will be created to calculate the discharge potential at a point due to discharge wells. The objective of this example is to introduce Python to analytic element developers through a learning-by-example approach. This example is by no means a thorough introduction into Python, but is merely intended to introduce some of the basic ideas of the Python language and to point out some of the pitfalls that the developers encountered. At several points reference is made to the Python Tutorial, available from the Python webpage and the book 'Learning Python', available from O'Reilly.

3.1 A simple well class

As a first step, a simple well class is created which has two methods: a constructor and a representation method. The constructor and representation methods have default names (underscore underscore init underscore underscore, for the constructor). To create the well class, start the Python shell (for example IDLE) and type the following text (formatted exactly the way it is shown) at the Python prompt:

```
>>> class Well:
    def __init__(self,xc,yc,Q,rw):
        self.xc=xc
        self.yc=yc
        self.rw=rw
        self.Q=Q
    def __repr__(self):
        return 'x,y,Q,rw: ' + str((self.xc,self.yc,self.Q,self.rw))
```

An additional carriage return must be given to finish entering the class. Note that >>> is the Python prompt in the Python shell.

Once the text has been entered, the Well class has been created but it does not have any instances yet. A well can be entered and it can be checked whether the data is stored correctly by typing

```
>>> w=Well(20,30,100,.1)
>>> w
x,y,Q,rw: (20, 30, 100, 0.1))
```

Individual attributes may be assessed as well. For example

```
>>> w.Q
100
```

So by just typing the name of an object, Python returns the object by calling its representation method. If no representation method is defined it uses a default one.

A few things have to be pointed out about the Python syntax. Variables do not have to be declared in Python, similar to MATLAB. The beginning and end of classes, methods and statements in Python are controlled by consistent indentation (the number of spaces used for indentation may be chosen freely, as long as you are consistent). The only special character that is used is the colon which is given at the end

of a statement that continues on the next line, such as the beginning of a class or method, but also after an `if`, `else` or `while` statement. Method names are preceded by the three letters `def`. The arguments of any method must include the instance itself by adding `self` to the argument list (yes, this seems a little inconvenient, but live with it). Furthermore, the `str` function converts a basic data type to a string. `str` takes only one argument, so that for this simple case `xc`, `yc`, `Q`, `rw` are grouped in a tuple (by separating them with commas and adding parentheses; p. 34, Python Tutorial).

The Well class may be typed in an editor (the developers like to use IDLE) and saved in a file with the `.py` extension, for example `aemex.py`. To import the file in Python (which is the same as typing all commands in by hand), the file must be placed in a directory that is specified in the `PYTHONPATH` environment variable (as explained in the Installation section).

The `aemex.py` file may be imported by typing `>>> import aemex`. Python will treat the `aemex.py` file as a module, and all classes defined in the file as part of the module. To create a well, the module name must now be added to create an instance of the class, for example `>>> w=aemex.Well(20,30,100,.1)`.

3.2 Adding the potential method

The Well class is edited to add a method that returns the discharge potential such that the file looks like this

```
import math
class Well:
    def __init__(self,xc,yc,Q,rw=0.1):
        self.xc=xc; self.yc=yc; self.Q=Q; self.rw=rw
    def __repr__(self):
        return 'x,y,Q,rw: ' + str((self.xc,self.yc,self.Q,self.rw))
    def potential(self,x,y):
        rsquare=(x-self.xc)**2+(y-self.yc)**2
        if rsquare <= (self.rw**2):
            return 0.0
        else:
            return self.Q/(4*math.pi)*math.log(rsquare/self.rw**2)
```

To import this modified module without exiting Python, the `reload` command must be used: `>>> reload(aemex)` (of course if you restarted the Python interpreter you can just do `>>> import aemex`). After a well is entered, the discharge potential, $\Phi = Q/(4\pi) \ln(r^2/r_w^2)$, can be calculated at any point x, y as follows:

```
>>> w=aemex.Well(20,30,100)
>>> w
x,y,Q,rw: (20, 30, 100, 0.1)
>>> w.potential(100,200)
119.977429376
>>> w.potential(100,-140)
119.977429376
```

To find out which methods are available in a module, the build-in `dir` command can be used

```
>>> dir(aemex.Well)
['_doc__', '__init__', '__module__', '__repr__', 'potential']
```

The `aemex` module starts with importing the `math` module, which comes with the standard distribution of Python, so that mathematical functions can be used. Several modules in a big project may all need the `math` module; Python recognizes a multiple import and ignores the statement. It must be declared in the file, however, otherwise Python does not have access to the module; this is an essential difference with, for example, MATLAB. The constructor has been modified slightly as well by making the radius of the well optional, with a default value of 0.1. Also, all four statements in the constructor are gathered on one line to save some space. Several statements may be given on one line as long as they are separated by semicolons.

3.3 Adding the Aquifer class

A simple Aquifer class is added. The Aquifer class keeps track of all the elements in the aquifer and has a method to calculate the potential at a point due to all elements. Upon construction, an empty `elementList` is created. An `elementList` is of data type `List`, a basic data type in Python that can store a list of objects; lists should be surrounded by square brackets, and the objects separated by commas, for example `alist=[well1,well2]`. `List` has a number of useful build-in methods (see Python Tutorial, p. 29). An `addElement` method is made such that an element can be added to the list; the `addElement` method makes use of the build-in `append` method of the `List` data type. In addition, the `Well` class has been modified to assign an aquifer as a parent to the element. The code now looks as follows.

```
import math
class Well:
    def __init__(self,parent,xc,yc,Q,rw=0.1):
        self.xc=xc; self.yc=yc; self.Q=Q; self.rw=rw
        self.parent=parent
        parent.addElement(self)
    def __repr__(self):
        return 'x,y,Q,rw: ' + str((self.xc,self.yc,self.Q,self.rw))
    def potential(self,x,y):
        rsquare=(x-self.xc)**2+(y-self.yc)**2
        if rsquare <= (self.rw**2):
            return 0.0
        else:
            return self.Q/(4*math.pi)*math.log(rsquare/self.rw**2)

class Aquifer:
    def __init__(self):
        self.elementList = []
    def __repr__(self):
        return str(self.elementList)
    def addElement(self,el):
        self.elementList.append(el)
```

```

def potential(self,x,y):
    pot = 0.0
    for el in self.elementList:
        pot = pot + el.potential(x,y)
    return pot

```

After this module has been imported (or reloaded), an aquifer must be created first. Wells can then be added to the aquifer. In the following example a well and an image well are placed on either side of the x axis, so that the potential is constant along the x axis. Comments in the code or in statements are preceded by a #.

```

>>> reload(aemex)
<module 'aemex' from 'c:\directory\aemex.pyc'>
>>> aq=aemex.Aquifer()          # Creates an Aquifer
>>> w1=aemex.Well(aq,20,30,100)
>>> w2=aemex.Well(aq,20,-30,-100)
>>> aq                          # Calls Aquifer representation method
[x,y,Q,rw: (20, 30, 100, 0.1), x,y,Q,rw: (20, -30, 100, 0.1)]
>>> w1.potential(50,0)
96.2944075201
>>> w2.potential(50,0)
-96.2944075201
>>> aq.potential(50,0)          # Potential is zero along x-axis
0.0
>>> aq.potential(100,0)
0.0

```

Note that two parentheses should always follow the name of a method, even if no arguments are passed on to the method (as, in this case, for the Aquifer constructor).

Python stores objects by reference. Once no reference exist to an object within the code, the garbage collector will automatically remove the object. The previous model of two wells could have been entered without giving the well and image well separate names, or even without giving them a name at all. Normally, if an object is not given a name it is created and then immediately removed by the garbage collector since no reference to the object remains. But since a reference to a Well object is created in the elementList, it will not be removed and can still be accessed through the elementList. For example:

```

>>> aq=aemex.Aquifer()
>>> aemex.Well(aq,20,30,100)
x,y,Q,rw: (20, 30, 100, 0.1)
>>> aemex.Well(aq,20,-30,-100)
x,y,Q,rw: (20, -30, -100, 0.1)
>>> aq
[x,y,Q,rw: (20, 30, 100, 0.1), x,y,Q,rw: (20, -30, -100, 0.1)]
>>> aq.potential(50,0)

```

```
0.0
>>> aq.potential(50,50)
-13.7311729587
```

Notice that if an object is created without a name, then the reference method is automatically called.

3.4 Adding the Circle class and inheritance

A Circle class is added to describe the geometry of a Circle. The circle class has one method, `isInArea`, which returns either a 1 if the point x,y is inside or on the circle, or a 0 if it is outside. The Well class, which has the geometry of a circle, will be derived from the Circle class as follows:

```
import math
class Circle:
    def __init__(self,xc,yc,R):
        self.xc = xc; self.yc = yc; self.R = R
        self.Rsquare = R*R
    def isInArea(self,x,y):
        rsquare = (x-self.xc)**2 + (y-self.yc)**2
        return rsquare <= self.Rsquare

class Well(Circle):
    def __init__(self,parent,xc,yc,Q,rw=0.1):
        Circle.__init__(self,xc,yc,rw)
        self.Q=Q; self.parent=parent
        parent.addElement(self)
    def __repr__(self):
        return 'x,y,Q,rw: ' + str((self.xc,self.yc,self.Q,self.R))
    def potential(self,x,y):
        if self.isInArea(x,y):
            return 0.0
        else:
            return self.Q/(4*math.pi) * \
                math.log( ((x-self.xc)**2 + (y-self.yc)**2) / self.Rsquare )
```

The Well class is derived from the Circle class by specifying the Circle class between parentheses at the beginning of the class. Once a well is created, the constructor of the Circle class must be called explicitly. In the Circle class, the radius of the circle is called `R` instead of `rw`, so that the code has to be changed somewhat. The square of the radius is also stored as an attribute in the Circle class, so that it also becomes available to the Well class; the `rw` attribute has disappeared, however:

```
>>> aq=aemex.Aquifer()
>>> w=aemex.Well(aq,20,30,100)
>>> w.rw          # This should give an error, since rw is not an attribute anymore
Traceback (innermost last):
```

```

File "<pyshell#24>", line 1, in ?
    w.rw
AttributeError: rw
>>> w.R
0.1
>>> w.Rsquare
0.01

```

The calculation of the discharge potential did not fit on one line anymore so that the first line is ended with a backslash, which allows a statement to continue on the next line. It is noted that this may not be the most efficient way to compute the discharge potential due to a well, since the r^2 is calculated twice (once to determine whether the point lies inside the circle, and once to calculate the potential), but we thought this was a simple example of inheritance. Also, note that the `Aquifer` class remains unchanged.

Other (given) elements may be added, just like the discharge well. As long as any additional element calls the `addElement` method upon it's creation, it is stored in the `elementList` and the potential due to all elements can be calculated.

3.5 Pythondoc

Python code may be documented by placing text between three quotes (either single or double quotes, as long as you are consistent). For example, the well class may be documented as follows.

```

class Well(Circle):
    """
    The well class contains wells with given discharge
    """
    def __init__(self,parent,xc,yc,Q,rw=0.1):
        """
        Creates well and adds well to elementList
        """
        Circle.__init__(self,xc,yc,rw)
        self.Q=Q; self.parent=parent
        parent.addElement(self) # Add element to elementList
    def __repr__(self):
        return 'x,y,Q,rw: ' + str((self.xc,self.yc,self.Q,self.R))
    def potential(self,x,y):
        """
        The potential function for a well returns the discharge potential
        at a given point x,y. Inside the well and at the radius of the well
        the potential is zero
        """
        if self.isInArea(x,y):
            return 0.0

```

```

else:
    return self.Q/(4*math.pi) * \
        math.log( ((x-self.xc)**2 + (y-self.yc)**2) / self.Rsquare )

```

Python 2.1 has an incredibly convenient built-in documentation browser named pydoc. Go to the directory where you installed Python 2.1, then go to the directory `Tools/Scripts` and double click on the pydoc icon (it would be convenient to put a shortcut to this icon somewhere else on your computer). A small window is opened where you can either type the name of the python file, or just click on the 'open browser' button and a webserver will be opened from where you can click your way to the documentation of any file in the `Pythonpath`, for example to read the documentation on the Numeric package! Note that any comments in the code (preceded by a `#`) will not be included in the documentation. Furthermore, there will be all kinds of handy connections made. For example, the `Well` class discussed here inherits from `Circle`. And thus pydoc automatically makes a connection to the `Circle` class and you can simply click on `Circle` and get the documentation from `Circle`. The documentation for the `Well` class looks roughly as follows (color, bold, and font size changes are omitted, so it looks much prettier, and more readable, in real life)

```
class Well(Circle)
```

The well class contains wells with given discharge

```

__init__(self, parent, xc, yc, Q, rw=0.10000000000000001)
Creates well and adds well to elementList

__repr__(self)

isInArea(self, x, y) from Circle

potential(self, x, y)
    The potential function for a well returns the discharge potential
    at a given point x,y. Inside the well and at the radius of the well
    the potential is zero

```

In addition, pydoc provides on-line help at the Python prompt. At the Python prompt, type `>>> from pydoc import *`. If now the `aemex.py` file with the documented `Well` class are imported, on-line help on the `Well` class is available at the Python prompt:

```

>>> help(aemex.Well)
Help on class Well in module aemex:

class Well(Circle)
|   The well class contains wells with given discharge
|
|   __init__(self, parent, xc, yc, Q, rw=0.10000000000000001)

```

```

|     Creates well and adds well to elementList
|
|     __repr__(self)
|
|     isInArea(self, x, y) from aemex.Circle
|
|     potential(self, x, y)
|         The potential function for a well returns the discharge potential
|         at a given point x,y. Inside the well and at the radius of the well
|         the potential is zero

```

or if you are just interested on help for one specific function:

```
>>> help(aemex6.Well.potential)
```

```
Help on method potential in module aemex:
```

```

potential(self, x, y) unbound aemex.Well method
    The potential function for a well returns the discharge potential
    at a given point x,y. Inside the well and at the radius of the well
    the potential is zero

```

3.6 Solving for unknown strengths

Determining unknown strengths in a general way may all seem a little complicated so let's do a simple example. The example of section 3.3 will be used as a starting point. The Well class is modified to include a potentialInfluence function and a potentialContribution function. Furthermore a getMatrixCoefficients and a getMatrixContribution method are added. Either one returns an empty list, because a well with given discharge does not have any unknown parameters. The Aquifer class is also modified. The potential method is modified so that it calls the potentialContribution function of every element, and a solve method is added. The solve method is not complete, but just compiles the matrix as a list of MatrixRow objects. A MatrixRow class is defined, which has two attributes, a list, called A, representing a list of matrix row coefficients, and a right-hand-side, called b. The code now looks like this:

```

import math
class Well:
    def __init__(self,parent,xc,yc,Q,rw=0.1):
        self.xc=xc; self.yc=yc; self.Q=Q; self.rw=rw
        self.parent=parent
        parent.addElement(self)
    def __repr__(self):
        return 'x,y,Q,rw: ' + str((self.xc,self.yc,self.Q,self.rw))
    def potentialInfluence(self,x,y):
        rsquare=(x-self.xc)**2+(y-self.yc)**2
        if rsquare <= (self.rw**2):

```

```

        return 0.0
    else:
        return 1/(4*math.pi)*math.log(rsquare/self.rw**2)
def potentialContribution(self,x,y):
    return self.Q * self.potentialInfluence(x,y)
def getMatrixCoefficients(self,fa,x,y):
    return []
def getMatrixContribution(self,elementList):
    return []

class Aquifer:
    def __init__(self):
        self.elementList = []
    def __repr__(self):
        return str(self.elementList)
    def addElement(self,el):
        self.elementList.append(el)
    def potential(self,x,y):
        pot = 0.0
        for el in self.elementList:
            pot = pot + el.potentialContribution(x,y)
        return pot
    def solve(self):
        rows=[]
        for e in self.elementList:
            rows = rows + e.getMatrixContribution(self.elementList)
        return rows

class MatrixRow:
    def __init__(self,A=[],b=0.0):
        self.A = A
        self.b = b

    A HeadWell class is created for head-specified wells. The HeadWell is derived from the Well class, and
    only the constructor, representation, getMatrixCoefficients, and getMatrixContribution methods have to be
    overloaded.

class HeadWell(Well):
    def __init__(self,parent,xc,yc,head,rw=0.1):
        Well.__init__(self,parent,xc,yc,0.0,rw)
        self.head=head
    def __repr__(self):
        return 'x,y,h,rw: ' + str((self.xc,self.yc,self.head,self.rw))
    def getMatrixCoefficients(self,fa,x,y):

```

```

    return [fa(self,x,y)]
def getMatrixContribution(self,elementList):
    mr = MatrixRow()
    for e in elementList:
        mr.A = mr.A + e.getMatrixCoefficients\
            (lambda z,x,y:z.potentialInfluence(x,y),self.xc,self.yc)
    mr.b = self.head - self.parent.potential(self.xc,self.yc)
    return [mr]

```

The `getMatrixContribution` returns the `MatrixRow` for the head-specified well. It loops through all elements and from each element it gets the `potentialInfluence` function evaluated at the center of the well. Notice that the other elements do not have to know anything about this well, as long as they have a `potentialInfluence` method. The syntax of the matrix generation may need a little further explanation. It makes use of the lambda construction in Python. The lambda construction allows for the definition of a temporary function, that can also be passed as a function argument. The Python documentation states: Lambda forms (lambda expressions) have the same syntactic position as expressions. They are a shorthand to create anonymous functions; the expression `lambda arguments: expression` yields a function object that behaves virtually identical to one defined with

```

def name(arguments):
    return expression

```

Notice that the right-hand-side of the matrix is equal to the head at the well (this is a simple example where the head and potential are equal), minus the potential of all elements, including the element under consideration. This means, that we don't solve for the discharge Q of the well, but for the change in discharge ΔQ compared to the current value of Q . This will come in handy when solutions are obtained in an iterative manner.

Let's do an example. Two head-specified wells are specified, one at $(x_1, y_1) = (0, 0)$ with a head of $\phi_1 = 10$, and one at $(x_2, y_2) = (10, 10)$ with a head of $\phi_2 = 20$; both wells have a radius of $r_w = 1$. The influence function of a well is written here as $\ln\{[(x - x_w)^2 + (y - y_w)^2]/r_w^2\}/(4\pi)$. When the influence function is evaluated at the well, the value is calculated at the well screen, and thus the influence function of the well at its own radius equals zero. The system of linear equations is:

$$\begin{bmatrix} 0 & \ln(200)/(4\pi) \\ \ln(200)/(4\pi) & 0 \end{bmatrix} \begin{bmatrix} \Delta Q_1 \\ \Delta Q_2 \end{bmatrix} = \begin{bmatrix} 10 - Q_2 \ln(200) \\ 20 - Q_1 \ln(200) \end{bmatrix} \quad (1)$$

Let's check whether the Python code generates the same result. The `Well`, `Aquifer`, `MatrixRow`, and `HeadWell` classes are gathered in a file named `solex.py`. The file `solex.py` is imported, the aquifer and wells are defined and the matrix constructed as follows

```

>>> import solex
>>> aq=solex.Aquifer()
>>> w1=solex.HeadWell(aq,0,0,10,1)
>>> w2=solex.HeadWell(aq,10,10,20,1)
>>> matrix=aq.solve()
>>> matrix          # matrix is a list of two MatrixRow objects

```

```
[<solex.MatrixRow instance at 874de0>, <solex.MatrixRow instance at 8741b0>]
>>> matrix[0].A # the 1st MatrixRow of the list, attribute A, gives the 1st matrix row
[0.0, 0.421626699478]
>>> matrix[1].A # the 2nd MatrixRow of the list, attribute A, gives the 2nd matrix row
[0.421626699478, 0.0]
>>> matrix[0].b # the 1st MatrixRow of the list, attribute b, gives the 1st RHS
10.0
>>> matrix[1].b # the 2nd MatrixRow of the list, attribute b, gives the 2nd RHS
20.0
```

When a discharge specified well is added, for example one at $(x,y) = (0,10)$, discharge $Q = 10$, radius $r_w = 1$, then only the right-hand-side will change:

```
>>> w3=solex.Well(aq,0,10,10,1)
>>> matrix=aq.solve()
>>> matrix[0].A
[0.0, 0.421626699478]
>>> matrix[0].b
6.3353220056
```

The previous solex example is the simplest case possible: elements have either one unknown and one control point (head-specified wells) or no unknowns at all (discharge-specified wells). But the procedure works the same for elements with more parameters. For example for an element with three parameter, the `getMatrixCoefficient` method returns a list of three coefficients instead of one.

Furthermore, if an element has multiple, say M , control points, the `getMatrixContribution` method loops through the control points and returns M `MatrixRow` objects to the solve method. Notice that when the row of a matrix is compiled (in the `getMatrixContribution` method) full use is made of the features of a `List`. For example adding a list with three coefficients to another list just adds three entries to the list. Furthermore, adding an empty list (returned by a discharge well) does not change the list at all, as shown below

```
>>> a=[1]
>>> b=[2]
>>> c=[]
>>> d=[3,4]
>>> a+b+c+d
[1, 2, 3, 4]
```

In the previous example a matrix and a right-hand-side were constructed. In addition, it has to be stored which column of the matrix corresponds to which parameter of which element. This information is stored in the `MatrixColumn` object. This object does not contain a column of the matrix (and thus the name may not be a very good choice). `MatrixColumn` has two attributes: `element` which is the element to which a matrix column corresponds, and `index` which corresponds to the index of the parameter of the element. For elements with only one unknown the index is 0 (lists and arrays start with index 0 in Python). But if an element has three unknown parameters, the index will be either 0, 1, or 2.

Once the entire matrix is compiled in solve, it is passed to the solver (in the `aem.matrix` module). The solver parses the list into an array (a data type defined in the Numeric package) and solves it either exactly

(for a square matrix) or in the least-squares sense (if there are more rows than columns). The solution vector is then passed back to the solve module in Aquifer. And the solve method in Aquifer passes the solution back to the elements, since their references are stored in a list of MatrixColumns.

It is important to note that the described procedure for obtaining a solution is very generic. Since a solution is obtained for elements in an elementList, it can be used to solve any part of the model, from one element to a string of elements to all elements in the aquifer. Furthermore, when a matrix with more rows than columns is constructed, a solution is obtained in the least-squares sense.