ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

On the Performance of the Python Programming Language for Serial and Parallel Scientific Computations

Samriddha Prajapati[1], Chitvan Gupta[2]

Student, CSE, NIET, Gr. NOIDA, India[1]

Assistant Professor, NIET, Gr. Noida, India [2]

Abstract: In this paper the performance of scientific applications are discussed by using Python programming language. Firstly certain techniques and strategies are explained to improve the computational efficiency of serial Python codes. Then the basic programming techniques in Python for parallelizing scientific applications have been discussed. It is shown that an efficient implementation of array-related operations is essential for achieving better parallel [11] performance, as for the serial case. High performance can be achieved in serial and parallel computation by using a mixed language programming in array-related operations [11]. This has been proved by a collection of numerical experiments. Python [13] is also shown to be well suited for writing high-level parallel programs in less number of lines of codes.

Keywords: Numpy, Pypar, Scipy, F2py.

I BACKGROUND AND INTRODUCTION

Earlier there was a strong tradition among computational scientists to use compiled languages, in particular Fortran 77 and C, for numerical [2] simulation. As there is increase in demand for software flexibility during the last decade, it has also popularized more advanced compiled languages C++ and FORTRAN 95. Whereas in recent days many computational scientists programmers and engineers have moved away from compiled languages to considered problem solving environments, for example: Maple, Octave, Matlab, and R (or S-Plus). Matlab has been very popular and is regarded as the preferred development [8] platform for numerical software by a significant portion of the of the Computational Science and Engineering community. Lots of the problems are solved in Matlab. It may seem a paradox that computational scientists, who claim to demand as high performance as possible in their applications, use Matlab. The success of Matlab and similar interpreted environments is due to:

i. Integration of simulation and visualization
ii. a simple and clean syntax of the command language
iii. Built-in functions operating efficiently on arrays in compiled code
iv. Interactive execution of commands with immediate feedback
v. A rich standardized library of numerical functionality that is conveniently available
vi. Numerical operations that are fast enough in plain Matlab[2]
vii. Documentation and support

Many scientists normally feel more productive in Matlab than with compiled languages and separate visualization tools. The programming language Python is now coming up as a potentially competitive alternative to Matlab, Octave, and other environments. Python, when extended with numericaland visualization modules, shares many of Matlab's advantages are as mentioned above [2]. One of the particular advantage of Python is that the language is very rich and powerful, especially in comparison with Matlab, Fortran, and C. In particular, Python is an object-oriented language that supports operator overloading and offers a cross-platform interface to operator system functionality. Advanced C++ programmers can easily mirror their software designs in Python and even obtain more flexibility and elegance.

Although Matlab supports object-oriented programming but creating classes in Python is much more convenient. Convenience seems to be a key issue when scientists choose an interpreted scripting language over a compiled language today. Another advantage of Python is that it is much simpler than in most other environments rather than interfacing legacy software written in Fortran, C, and C++. This is because Python was designed to be extendible with compiled code for efficiency, and several tools are available which make the integration of Python and its libraries easier. So with the above properties and the interfacing capabilities, the Python represents a best

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940
IJARCCE
International Journal of Advanced Research in Computer and Communication Engineering
ICACTRP 2017
International Conference on Advances in Computational Techniques and Research Practices
Noida Institute of Engineering & Technology, Greater Noida
Vol. 6, Special Issue 2, February 2017

environment for doing parallel programming for computational science [11]. The language has already attracted significant interest among computational scientists for some years. A key question for the community is numerical efficiency of Python-based computing platforms. The aim of this paper is to quantify such numerical efficiency [2].
It is also need to be mentioned that Python is being used in a much broader context than scientific programming only. In computer science, the language has a strong and steadily increasing position as a general-purpose programming tool for diverse places such as system administration, dynamic web sites, distributed systems, software Engineering, graphical user interfaces, computational steering, search engines, education and also in large-scale business applications. Investigating a scientific problem with the aid of computers requires software which performs a vast range of different tasks: user interfaces (command line, files, web and the graphical windows), I/O management, numerical computing, data analysis, visualization, file and directory manipulation, and report generation. High computational efficiency is normally needed a few of these tasks and the number of statements calling for ultimate efficiency is often just a few percentage of a whole software package. The non-numerical tasks are usually more efficiently taken out in a scripting environment like Python than in compiled languages. The classical scripting way is to let a script of python call up stand-alone simulation programs and manage the whole computational and data analysis pipeline. However as computational scientists and Engineers move to the Python, they will probably like to implement the numeric in Python. In this paper it is explained that how to implement the numeric in Python using many techniques with different degrees of programming and computational efficiency.

The comparison of performance for Python with corresponding implementations in Fortran 77 and C is also explained. As we go along, we shall also point out inefficient constructs in Python programs. It is fair to say that the core of this Python programming language is'nt exactly suitable for scientific applications involving intensive computations. This is mainly due to slow execution of long nested loops and the lack of efficient array data structures. However, the add-on package Numerical Python, often referred to as **NumPy**, provides contiguous multi-dimensional array structures with a large library of array operations implemented efficiently in C. The **NumPy** array evaluating facilities resemble those of Matlab, with respect to functions and computational efficiency.The problem with slow loops can be highly relieved through vectorization i.e., expressing the loop semantics via a group of basic **NumPy** array operations, where each problem involves a loop over array entries efficiently implemented in C. The same technique is well-known from programming in Matlab and other interpreted environments where loops run slowly.
Vectorised Python code may still run a way of 3–10 slower than optimized implementations in pure Fortran or C/C++ [7]. In some of cases, or in cases where vectorization of an algorithm is inconvenient, computation-intensive Python loops should be terminated directly to Fortran or C/C ++ . In a Python program one cannot distinguish between a function implemented in C/C++/FORTRAN and a function implemented in pure Python. With the F2PY[9] tool,
Coupling Python with Fortran is done almost automatically. As a conclusion, combining core Python with NumPy and Fortran/C/C++ code migration constitute a convenient and efficient scientific computing environment on serial computers.

II. PERFORMANCE OF SERIAL PYTHON CODES

We will discuss in this paragraph how to manage serial scientific applications using Python. In particular, the using of efficient array objects from the **NumPy** package, the efficiency of Python for-loops, and mixed-language programming will be studied. These issues are also of fundamental importance for an efficient parallelization of scientific applications in Python, which is to be addressed later. For any computation-based applications that are implemented in a traditional language, such as Fortran 77 or C, the main data construction are normally made up of arrays. The central of the computations is in the form of traversing the arrays and carrying out computing operations in (nested) loops, such as do-loops in FORTRAN and for-loops in C. Therefore, our efficiency investigations in the present section focus on the typical loops that are used as the building blocks in scientific codes. Array computing in Python employ the **NumPy** package. This package has a primary module defining the array data structure and efficient C functions operating on arrays. Presently two basic versions of this module exist. Numeric is long established module from the mid 1990s, while numarray is a more proper new implementation. The latter is meant as a replacement of the former, and no further development [8] of Numeric takes place. However, there are many numerical Python [2] code utilizing Numeric than we expect both modules to co-exist for more time. A lot of scripts written for Numeric will automatically work for numarray, since the programming interface of the two modules are very same. However, there are unfortunately some differences between Numeric and numarray, which may require manual editing to replace one module by the other. (The **py4cs.numpytools**module helps writing scripts that can run unaltered with both modules.) Many tools for scientific computing with Python, including the F2PY [9] program and **pypar**module to be used later in this paper. The very useful **SciPy** package with lots of numerical

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

computing functionality also works best with Numeric. Therefore, most of the experiments reported in this paper involve Numeric.

All experiments reported in this section are collected in software that can be downloaded and executed in own computing environment. The source code files also document precisely how our experiments are implemented and conducted.

Loops over NumPy Arrays. To illustrate computing with one-dimensional arrays in Python, we first generate an array of coordinates $x_i = i/(n-1), i = 0,\ldots,n-2$, and then we compute $y_i = \sin(x_i)\cos(x_i) + x_i^2$. A pure Python implementation employing a plain for-loop is given next.

Example 1 *Filling a one-dimensional array using Python loops.*

```
from Numeric import arange, zeros, Float

n = 1000001
dx = 1.0/(n-1)
x = arange(0, 1, dx)      # x = 0, dx, 2*dx, ...
y = zeros(len(x), Float)  # array of same length as x, zeros
                          # as Float (=double in C) entries
from math import sin, cos  # scalar sin and math functions
for i in xrange(len(x)):   # i=0, 1, 2, ..., length of x - 1
    y[i] = sin(x[i])*cos(x[i]) + x[i]**2
```

The arange function allocates a **NumPy** array with values from a start value to a stop value with a specified increment. Showing a double precision real-value array of lengthis done by the zeros (n, Float) call. Traversing an array can be done by a for loop as shown, where x range is a function that returns indices 0, 1,2, and up to the length of x in this case. Indices in **NumPy** arrays always start at 0. The for-loop in Example 1 requires about 9 seconds on our test computer for an array length of one million. As a comparison, a corresponding plain for loop in Matlab is about 16 times faster than the plain Python for-loop. Switching from Numeric to numarray increases the CPU[5] time by 32%

Example 2 *Filling a one-dimensional array by vectorized code.*

```
from Numeric import arange, sin, cos
n = 1000001
dx = 1.0/(n-1)
x = arange(0, 1, dx)        # x = 0, dx, 2*dx, ...
y = sin(x)*cos(x) + x**2
```

In the present example, there is no need to allocate y beforehand, because an array is created by the vector Expression sin(x)*cos(x)+ x**2. Table 1 shows performance results for various vectorized versions and other implementations of the present array computation. From Table 1 we see that this vectorized version runs about 9 times faster than the pure Python version in Example 1.

The reader should notice that the expression sin(x)*cos(x)+ x**2 works for both scalar and array arguments. (In contrast, Matlab requires special operators like.* for array arithmetics.) We can thus write a function I(x), which evaluates the expression for both scalar and array arguments, as illustrated below.

Example 3 *Filling a one-dimensional array by calling a function.*

```
from Numeric import arange, sin, cos, Float
n = 1000001;   dx = 1.0/(n-1);   x = arange(0, 1, dx)

def I(x):
    return sin(x)*cos(x) + x**2

y = I(x)

# could also compute with a (slow) loop:
y = zeros(len(x), Float)
for i in xrange(len(x)):
    y[i] = I(x[i])
```

The function call I(x[i]) in the above for-loop adds an overhead of about 8% compared with inlining the mathematical expressions in Example 1. In general, function calls in Python are expensive.

III. PARALLELIZING SERIAL PYTHON CODES

Before running on any parallel [11] program on a computing machine, a serial program must be parallelized first. In this section, we will explain how to parallelize serial structured Python [13] computations. Here we will see that the high-level programming of Python [13] gives rise to parallel codes [12] of a clean and compact style.

A. **Parallelization in Message-passing** [10] There are several different programming approaches to implementing parallel [11] programs. In this paper, however, we have chosen to restrict our attention to message-passing[10] based parallelization . This is because the message-passing[10] approach is most widely used and has advantages with respect to both performance and flexibility. We note that a message between two neighboring processors contains simply a sequence of data items, typically a vector of numerical values. For example of message passing[10] programming, let us consider the simple case of parallelizing a five-point-stencil operation, which is in fact a two-dimensional simplification. That is, the stencil operation is carried out on the entries of a two-dimensional global array um, and the results are stored in another two-dimensional global array u.

Work Load Division.

The first step of parallelizing the five-point-stencil operation is to divide the computational work among P processors, which are supposed to form an Nx×Ny=P lattice. A straight forward work division is to partition the interior array entries of u and um disjointly into P=Nx×Ny small rectangular portions, using horizontal and vertical “cutting lines”. If the dimension of u and um is (nx+1)×(ny+1), then the (nx−1)×(ny−1)interior entries are divided into Nx×Nyrectangles. For a processor identified by an index tuple (l, m), where l<Nx and m<Ny, it is assigned with (nlx−1)×(nmy−1)interior entries. Load balancing requires that the processors have approximately

Copyright to IJARCCE DOI 10.17148/IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

the same work amount, i.e., $nlx-1 \approx (nx-1)/Nx$ and $nmy-1 \approx (ny-1)/Ny$.

Local Data Structure

The above work division means that processor (l, m) is only responsible for updating $(nlx-1)\times(nmy-1)$entries of u. To avoid having to repeatedly borrow um values from the neighboring processor when updating those u entries lying immediately beside each internal boundary, as depicted in Fig. 1, the data structure of the local arrays should include one layer of “ghost entries” around the $(nlx-1)\times(nmy-1)$ assigned entries. That is, two local two-dimensional arrays u_loc and um_loc, both of dimension $(nlx+1)\times(nmy+1)$, areused on processor(l, m). Note that no processor needs to construct the entire u and um arrays, and that computing the layer of ghost entries in u_loc is the responsibility of neighboring processors.

Local Computation

Assuming that the dimensions nlx and nmy are represented as variables nx_loc and ny_loc in a Python[13] program. The parallel[11] execution of a five-point-stencil operation, distributed on $P=Nx\times Ny$ processors, can be implemented as the following Python[13] code segment:

```
u_loc = zeros((nx_loc+1,ny_loc+1), Float)
um_loc = ones((nx_loc+1,ny_loc+1), Float)
for i in xrange(1,nx_loc):
    for j in xrange(1,ny_loc):
        u_loc[i,j] = um_loc[i,j-1] + um_loc[i-1,j] \
                     -4*um_loc[i,j] + um_loc[i+1,j] + um_loc[i,j+1]
```

Example 4. F2PY python implementation in Local computation

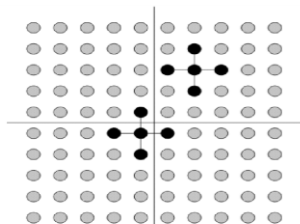

Fig1. Local Python implementation of a five-point-stencil

Need for Communication

After the code segment in fig1. is concurrently executed on each processor, we have only updated the $(nlx-1)\times(nmy-1)$ interior entries of u_loc . In a typical parallel application[12], the u_loc array will probably participate in later computations, with a similar role as that of the um_loc array in Example 4. Therefore, we also need to update the layer of ghost entries in u_loc . This additional update operation means that two and two neighboring processors exchange values across their internal boundary. More specifically, when processor (l, m)has a neighbor (l+1,m)in the upper x-direction (eg. l<Nx-1), processor(l, m) needs to send a vector containing values of u_loc[nx_loc-1,:], as a message, to processor(l+1,m). In return, processor (l, m) receives a vector of values from processor (l+1,m), and these received values are assigned to the entries of u_loc[nx_loc,:]. This procedure of message exchange [10] across an internal boundary has also to be carried out with each of the other possible neighbors, i.e. processors (l−1,m),(l, m+1), and (l, m−1).

IV. PYTHON IMPLEMENTATION OF PARALLELIZATION TASKS

Although the parallelization example from Section III is extremely simple, it nevertheless demonstrates the generic tasks that are present in any parallel[11] numerical code[2] . Using an abstract description, we can summarize thesegeneric tasks as follows:

Workload partitioning, i.e., divide the global data into local data to be owned exclusively by the processors

Serial computations using only local data items on each processor

1) Preparation of the outgoing messages, i.e., extract portions of some local data into vectors of values, where each vector works as a message
2) Message exchange between neighboring processors
3) Extraction of the incoming message
4) Update portions of some local data (e.g. the ghost entries) using values of the incoming messages of the above five generic tasks, the first task normally only needs to be executed once, in the beginning of a parallel application, whereas the second task concerns purely serial codes. Therefore, under the assumption that the serial codes have good serial performance, the overall performance of a parallelized application depends heavily on the last three tasks. We will show in this section how these communication related tasks can be implemented in Python. The efficiency of the parallel [12] Python[13] implementation is studied by detailed measurements.

Our attention will be restricted to parallelizing numerical applications that are associated with structured computational meshes. This is because ensuring the parallelization quality of an unstructured computing application requires the same principles as for a structured computing application. Like a structured computing application, an unstructured computing application also uses arrays to constitute its data structure. The typical difference is that an unstructured computing application only uses flat one-dimensional arrays, and traversing the entries of an array is often in an unstructured fashion, and the number of neighbors may also vary considerably from processor toprocessor.

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

V PYTHON MPI BINDINGS THROUGH PYPAR

Let us start with looking at message exchanges between the processorsin a Python [13] program. For the purpose of efficiency, we have chosen the**pypar**package because of its light-weight implementation and user-friendly syntax. As we have mentioned before, the **pypar**package provides a Python[13] wrapper of a subset of the standard MPI routines. The underlying implementation of **pypar**is done in the C programming language, as an extension module that can be loaded into Python. The following is an example of using the two most important functions in **pypar**, namely **pypar.send**and **pypar.receive**:

```
import pypar                    # import the pypar module
myid = pypar.rank()             # rank of a processor
numprocs = pypar.size()         # total number of processors
msg_out = zeros(100, Float)     # a NumPy array to be communicated

if myid == 0:
    pypar.send (msg_out, destination=1)
    msg_in = pypar.receive(numprocs-1)
else:
    msg_in = pypar.receive(myid-1)
    pypar.send (msg_out, destination=(myid+1)%numprocs)

pypar.finalize()                # finish using pypar
```

Example 5 Functions of pypar

In the following Example 6 there is Array Slicing in Preparing and Extracting Messages is shown.When parallelizing array-based structured computations,

```
if has_upper_x_neighbor:
    pypar.send (u_loc[nx_loc-1,:], destination=upper_x_neighbor_id,
    bypass=True)
    u_loc[nx_loc,:] = pypar.receive (upper_x_neighbor_id, \
                                     buffer=buffer_x, bypass=True)

if has_lower_x_neighbor:
    pypar.send (u_loc[1,:], destination=lower_x_neighbor_id, bypass=True)
    u_loc[0,:] = pypar.receive (lower_x_neighbor_id, \
                                buffer=buffer_x, bypass=True)

if has_upper_y_neighbor:
    pypar.send (u_loc[:,ny_loc-1], destination=upper_y_neighbor_id,
    bypass=True)
    u_loc[:,ny_loc] = pypar.receive (upper_y_neighbor_id, \
                                     buffer=buffer_y, bypass=True)

if has_lower_y_neighbor:
    pypar.send (u_loc[:,1], destination=lower_y_neighbor_id, bypass=True)
    u_loc[:,0] = pypar.receive (lower_y_neighbor_id, \
                                buffer=buffer_y, bypass=True)
```

Example 6. Array Slicing in Preparing and Extracting Messages

we only use portions of a local array (such as its ghost layer) in the message exchanges. The slicing functionality of **NumPy** arrays is very important for both the task of preparing an outgoing message and the task of extracting an incoming message. The resulting Python implementation is extremely compact and efficient. Let us give a **pypar**implementation of the operation that updates the ghost layer of the local two-dimensional array u_loc, which is needed after executing the code as given above in example 6. It should be noted that the above example has merged together the three generic tasks, namely preparing outgoing messages, exchanging messages, and extracting incoming messages. The actual sequence of invoking thesend and receive commands may need to alternate from processor to processor for avoiding dead locks. It is of vital importance for the performance to use the slicing functionality to prepare outgoing messages, instead of using for -loops to copy a desired slice of a multi-dimensional array, entry by entry, into an outgoing message. Similarly, extracting the values of an incoming message should also be accomplished by the slicing functionality. For a receive command that is frequently invoked, it is also important to reuse the same array object as the message buffer, i.e., buffer_x for the x-direction communication and buffer_y for the y -direction. These buffer array objects should thus be constructed once and for all, using contiguous underlying memory storage. The measurements in Section V will show that the array slicing functionality keeps the extra cost of preparing and extracting messages at an acceptably low level. This is especially important for parallel [12] three-dimensional structured computations.

A Simple Python Class Hierarchy using box partitioning

We can observe that four of the five generic tasks in most parallel numerical applications are independent of the specific serial computations. Therefore, to simplify the coding effort of parallelizing array-based structured computations, we have devised a reusable class hierarchy in Python. The name of its base class is **BoxPartitioner**, which provides a unified Python interface to the generic task of work load partitioning and the three generic communication-related tasks. In addition to several internal variables, two of the major functions in **BoxPartitioner**are declared as:

i. Prepare_communication
ii. Update_internal_boundaries.

The first function is for dividing the global computational work and preparing some internal data structures (such as allocating data buffers for the incoming messages), whereas the latter is meant for the update operation

VI CONCLUSION

The discussion and analysis presented, together with the measurements in Section V, given us reasons to believe that Python is sufficiently efficient for scientific computing. However, “Python” implies in this context the core language together with the Numerical Python [2] package and frequent migration of nested loops to C/C++ and Fortran extension modules. In addition, the programmer must avoid expensive constructs in Python. In data structures we should use Python arrays and

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

computation-intensive code segments with compiled code, either in a ready-made library such as Numerical Python or in tailored C/C++ or FORTRAN code. Moreover, the fully comparable measurements of Python-related parallel wave [4] simulations against a pure C implementation also encourage the use of Python in developing parallel applications. The results obtained in this paper suggest a new way of creating future scientific computing applications. Python, with its clean and simple syntax, high-level statements, numerous library modules, vectorization capabilities, bindings to MPI, can be used in large portions of an application where performance is not first priority or when vectorized expressions are sufficient. This will lead to shorter and more flexible code, which is easier to read, maintain, and extend. The parts dealing with nested loops over multidimensional arrays can be migrated to a compiled extension module using python. Our performance tests show that such mixed-language applications may be as efficient as applications written entirely in low level C or Fortran 77.

REFERENCES

[1] D. Arnold, M.A. Bond, Chilvers and R. Taylor,Hector: Distributed objects in Python, In Proceedings of the 4th International Python Conference, 1996.
[2] D. Ascher, P.F. Dubois, K. Hinsen, J. Hugunin and T. Oliphant, Numerical Python, Technical report, Lawrence Livermore National Lab.,CA, 2001. http://www.pfdubois.com/numpy/numpy.pdf.
[3] D. Beazley,Python Essential Reference, (2nd ed.), New Riders Publishing, 2001.
[4] D. Blank, L. Meeden and D. Kumar,Python Robotics: An environment for exploring robotics beyond LEGOs, In Proceedings of the 34thSIGCSE technical symposium on Computer Science Education, 2003, pp. 317–321.
[5] Blitz++software, http://www.oonumerics.org/blitz/.
[6] W.L. Briggs,A Multigrid Tutorial. SIAM Books, Philadelphia, 1987.
[7] A.M. Bruaset,A Survey of Preconditioned Iterative Methods, Addison-Wesley Pitman, 1995.
[8] D. Beazley et al.,Swig 1.3 Development Documentation, Technical report, 2004. http://www.swig.org/doc.html.
[9] F2PY software package, http://cens.ioc.ee/projects/f2py2e.
[10] Message Passing Interface Forum, MPI: A message-passing interface standard.Internat. J. Supercomputer Appl.8(1994), 159–416.
[11] A. Grama, A. Gupta, G. Karypis and V. Kumar, Introduction to Parallel Computing, (2nd ed.), Addison–Wesley, 2003.
[12] W. Gropp, E. Lusk and A. Skjellum,Using MPI – Portable Parallel Programming with the Message-Passing Interface, (2nd ed.), The MITPress, 1999.
[13] D. Harms and K. McDonald,The Quick Python Book, Manning, 1999.