Journal of Fundamental and Applied Research **Vol. 1**, Issue 1 (2021) 20210003

Received: 15 December 2021 / Accepted: 20 December 2021
Published online: 22 December 2021
© The Author(s) 2021

Python scripting and scientific computing

Askar B. Abdikamalov[1, 2, 3, *]

[1] *Center for Field Theory and Particle Physics and Department of Physics, Fudan University, 200438 Shanghai, China*
[2] *Ulugh Beg Astronomical Institute, Astronomy Str. 33, Tashkent 100052, Uzbekistan*
[3] *Institute of Fundamental and Applied Research, National Research University TIIAME, Kori Niyoziy 39, Tashkent 100000, Uzbekistan*

The manuscript presents important aspects of the Python scripting language that make it a powerful and attractive tool for scientific computing. Several methods are described that accelerate various types of scientific computing in Python.

I. INTRODUCTION

Development of scientific programs often takes huge fraction of research efforts in computational sciences. Employing efficient coding tools and techniques to save development time is therefore of enormous importance. In recent years, Python ([1]) has attracted enormous attention as a language for human-efficient writing of numerical codes. Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C. The language provides constructs intended to enable clear programs on both a small and large scale. Python supports multiple programming paradigms, including object-oriented, imperative and functional programming styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library. In the following, we will review the salient aspects of Python that make them particularly useful language for scientific computing. We will also outline ways of overcomes one of the main concerns about Python: its computational efficiency.

II. SCRIPTING AND TRADITIONAL PROGRAMMING

The aim of this section is to discuss some key differences between traditional programming and scripting with particular focus on the Python language. These are two quite distinct programming approaches with distinct aims. Traditional programming, sometimes also called as system programming, refers to building - usually large and homogeneous - applications using languages such as Fortran, C, C++, etc. Within the scope of this paper, by scripting we will refer to programming at a high abstraction level using languages like Perl, Python, Ruby, Scheme, or Tcl. We will also discuss the features that distinguish these programming styles. During the last several years, the popularity of scientific computing environments such as IDL, Maple, Mathematica, Matlab, and Octave, has increased tremendously [4]. Scientists and engineers tend to find such environments to be more productive. One of the main reasons is the simple and clean syntax of the languages in these environments. Another important reason is the tight integration of calculation and visualization in these environments, in which user can quickly and conveniently visualize what he/she just have calculated. One of the main problems with the above-mentioned environments is that it is hard to make them work with other types of numerical and visualization software in an easy way. The programming languages used are also somewhat simple or primitive. This is where Python scripting comes in. Python not only offers the clean and simple syntax of the popular scientific computing environments, but also a very powerful language and plenty of tools for easy incorporating of simulation, visualization, and data analysis programs in easy ways. Computational scientist's

*Electronic address: aaskar17@fudan.edu.cn

job involves much more tasks than writing computationally intensive number-crunching codes and performing computations with such codes. Quite frequently, it is about moving data in and out of different tools, converting one data format to another, extracting numerical data from a text file, and managing numerical computations that involve a large number of data files and directories. Such tasks are much faster and convenient to accomplish in a language like Python than in Fortran, C, C++, C#, or Java. Fortran and C programmers may find novel programming paradigms and languages very attractive, but they may also want to reuse their old efficient and well-tested codes. Here again Python offers an attractive solution. Rather than porting those programs to new languages like C++, Java, etc., one can wrap such codes in a suitable scripting interfaces. Calling Fortran, C, or C++ codes from Python is particularly easy. Moreover, the Python interfaces can take advantage of object-oriented design and easy linking to graphical user interfaces (GUIs), visualization, or other modules. Despite these advantages, environments such as Matlab (or Octave, Scilab, Rlab, Euler, Tela, Yorick, etc.) is not always sufficient. Matlab is currently the most popular and fairly standard software of this kind. Matlab and Python have indeed many features in common such as simple, clean, and convenient syntax, no variable declaration, easy creation of GUIs, and coupling f calculation and visualization. Nevertheless, in our opinion Python has some definite advantageous over Matlab and similar programs:

- the Python programming language is more powerful,
- the Python environment is open and designed for easy integration with external tools,
- a complete toolbox or module with lots of functions and classes can be contained in a single file (unlike a number of M-files in Matlab),
- simpler transfer functions as arguments to functions,
- more convenient object-oriented programming,
- simple construction and use of nested and heterogeneous data structures,
- simpler interfacing to C, C++, and Fortran codes
- scalar functions work with array arguments to a larger degree,
- the Python source code is free and runs on more architectures.

It is convenient to have a term to refer to the languages used for traditional scientific programming and the scripting languages. [4] proposed to use type-safe languages and dynamically typed languages, respectively. These terms classify the languages by the variable flexibility, i.e., if variables must be declared with a specific type or whether variables can hold data of any type. This is a clear and important difference of the functionality of the two types of programming languages [4]. There are many other ways of differentiating programming languages. Examples are 1) compiled versus interpreted, 2) high and low level languages, etc. However, in this paper, our focus is programming style instead of languages. Therefore, from now one, we will use the distinction in terms of type declaration discussed above. In type-safe languages, each variable is required to be explicitly declared with a type. The compiler then uses this declaration to make sure that the right data type is combined with the appropriate algorithms. Such languages are also often called as statically typed and strongly typed languages. Static, unlike dynamic, implies that a type of a variable is fixed at compilation time. This distinguishes, e.g., Python from C. Strong or weak typing refers to situation if a variable of a given type can be automatically used as a variable of another type, i.e., if implicit type conversion can take place. One advantage of type-safe languages is that is it less likely to produce bugs and it is generally safer to program. This, however, has a cost of reduced flexibility. In big development projects with many coders, the static typing can help to handle complexity. On the other hand, code reuse is not always easily achieved by static typing because codes work only with a given data type. Object-oriented and generic programming provide convenient tools to overcome such restrictions of a statically typed paradigm. Dynamically typed languages do not require variables to be declared to be of any type, and thus do not impose the above-discussed restrictions on how variables and functions are sued. When programmer needs a variable, he/she just assigns it a value. There is no requirement to declare the type. This results in a lot of flexibility. This drawback of this is negative side effects from typing errors. On the other hand, dynamically typed languages often carry out detailed checks during run time if variables and functions are consistently used. Moreover, a piece of code in dynamically typed languages can be used for many purposes. This leads to smaller code and thus the fewer bugs.

Copyright © 2021

A. Productive pairs of programming languages: Example of Unix and C

Unix evolved into a highly productive environment for software development thanks to two distinct programming tools: the traditional programming language C and the Unix shell for combining C programs into new applications. A user can apply a set of only a few of elementary C programs together with a suitable shell program to solve new problems. For instance, there is no simple Unix tool that allows a user to browse a sorted disk usage list in the directories of a user. However, one can easily build such a tool as a shell script that combines three C programs: (i) du for summarizing disk usage, (2) sort for sorting lines of text, and (3) less for browsing text files [4]: du -a $HOME — sort -rn — less. This glues three programs that are independent of each other. Without the gluing ability of Unix shells, one would need to develop a C program to solve the present problem, which, however, would be much more complicated. This epitomizes the power of Unix in a nutshell. A Unix command interpreter, or shell, is a language for combining applications written in traditional languages. There are many interpreters: Bourne shell (sh) and C shell (csh), Bourne Again shell (bash), Korn shell (ksh), and Z shell (zsh). Although the Unix shells have many useful high-level features, the shells are quite primitive programming languages, especially compared to modern programmers.

B. Scripts uses less code

Python, like many dynamically typed languages, supports a number of high-level data structures and constructs that enables users to write codes which are far shorter than codes with similar functionality written in traditional languages like Fortran, C, C++, or Java. Stated differently, on average each statement does more work. A simple example that demonstrates this is the task of reading the number of real numbers in a file, in which several numbers may appear at one line and empty lines are allowed. This task is fulfilled by two Python statements: F = open(filename, 'r'); n = F.read().split(). Doing this in Fortran, C, C++, or Java requires a loop at least. In some of the languages, a number of statements needed for dealing with a changing number of real numbers in a line. The are many other situation where scripting languages such as Python accomplishes a given task with far less number of points. [3] points out a number several applications where the ratio of code size and the implementation time between type-safe languages and the dynamically typed Tcl language vary from 2 to 60. For example, a database application implementation in C++ consumed two months, while the same thing in Tcl was done only in one day. A database library was developed in C++ in 2-3 months and reimplemented in Tcl in about one week. The Tcl implementation of an application for displaying oil well curves required two weeks of labor, while the reimplementation in C needed three months.

C. Mixed language programming

Another important advantage of Python is that it allows mixed-language programming. Employing distinct languages for different tasks within a given software is often a good approach. Dynamically typed language themselves are usually developed in C and thus have well-known methods for extending the language with new functions coded in C. Python can also easily be incorporated with other languages such as C++ and Fortran. Static type languages can also be incorporated together. However, for example, calling a C function from Java is a more involved than calling the same function from Python. This stems from the way the languages were initially designed: Python was designed to be extended with new C and C++ codes, while Fortran, C, C++, and Java were meant to build large software in one language. Such distinct thinking results in dynamically typed languages being simpler and more flexible for mixed-language programming.

III. COMPUTATIONAL EFFICIENCY OF PYTHON

One of the major concerns about scripting languages is their computational efficiency. When executed, scripts are compiled to hardware-independent byte-code, which is then interpreted. Type-safe language codes, (except Java), are compiled, meaning the code is translated into hardware-dependent machine instructions. The interpreted, high-level, flexible data structures employed in scripting languages lead to decrease in execution speed penalty, especially when reading data structures of considerable size [2]. However, for a wide range of tasks, dynamically typed languages are efficient enough on modern computers. When a script is executed in a matter of a dew seconds, a factor of a few slower code is not not crucial. Moreover, dynamically typed languages can yield optimal efficiency in many applications. The above discussed one-line Python code for splitting a file into numbers calls up highly optimized C code to perform

Copyright © 2021

the splitting. It is very hard to beat the efficiency of Python in this example in e.g. C. As a result, dynamically typed codes often provides highest efficiency both in terms of development and computer time.

A. Approaches to speeding up Python programs

Here, we will outline some of the recent approaches to increasing computational efficiency of Python. Many numerical schemes use loops over array data structures. Standard Python loops over large lists or arrays run slower in Python. The slower speed may be tolerable in many application such as solving ordinary differential equations in one space dimension. However, there are many computationally-intensive scientific applications where standard Python code needs to run for hours, whereas a Fortran or C code implementing the same algorithm finishes within minutes. One method of making loops over large arrays faster is to resort to vectorization, that is to replace the loops by a bunch of operations on entire arrays. This can lead to huge speed-up in some case [5]. Another method is move the loops to a compiled code, either in Fortran, C, or C++. There exist a wide range of tools for easily combining Python and compiled languages. For example, F2PY is a tool for combining Fortran 77/90 with Python. Cython [6] is a recent approach that extends Python with some new elements that allow automatic compilation of pieces of code to machine code and thus leading considerable speed-up of some computations. Another option is to write manually the loops that need to be migrated in C. There is also PyPy. PyPy is a Python interpreter and just-in-time compiler. PyPy focuses on speed, efficiency and compatibility with the original CPython interpreter. PyPy started out as a Python interpreter written in the Python language itself. Current PyPy versions are translated from RPython to C code and compiled. The PyPy JIT compiler is capable of turning Python code into machine code at run time. For a computational scientist who needs to write a code in Python, the central question goes as follows: What method to resotrt to speed up loops over arrays? When to vectorize loops? Or should one resort to the new tool Cython and implement the loops in Python with some extra commands? Which cases suits PyPy? We addressed these question in our next paper for several reference computational problems.

Summary and Conclusion

We have argued that Python has many attractive capabilities. It has many advantages over other dynamically typed languages:

1. Because of its clean syntax, Python is easy to learn;
2. Detailed run-time checks help to detect code errors and reduce development time;
3. Coding with nested, heterogeneous data structures is simple;
4. Object-oriented programming is relatively easy;
5. If offers various approaches for efficient numerical computing, and;
6. The integration of Python with C, C++, Fortran, and Java is very well designed and supported. Fortran, C, C++, or Java programmers you will find the following aspects of Python scripting especially attractive:
7. Because the type of variables and function arguments are not declared, a code piece has a wider application and more likely to be reused.
8. Dynamic memory does not need to be administered. Variables are created when needed, while Python destroys them automatically when appropriate.
9. Keyword arguments allow better call flexibility and enhance code documentation.
10. It is easy to set up and deal with arbitrarily nested, heterogeneous lists and dictionaries. This usually overcomes the need to code separate classes to describe complicated data structures.
11. Any Python data structure can be printed out to a file or on the screen with a single command. This represents a enormously convenient capability for debugging or storing data.
12. GUI capabilities is easily accessible at a high level.
13. Python offers many advanced C++ option: classes, templates, inheritance, namespaces, and operator overloading.
14. Python uses regular expressions for many goals and this simplifies the code readability significantly.

 Copyright © 2021

15. Python offers an interactive shell that makes it easy to test and debug different code pieces before incorporating them into a source code.

16. Dynamically typed languages are frequently used for small software, but Python's modular system makes it well suited for large-scale codes also.

17. Python is far more dynamic than compiled languages. Programmers can generate code, e.g., by adding new variables to classes at run time.

18. Development a program in Python takes less time than in Fortran, C, C++, or Java. This leads to far shorter code and thus shorter development time.

For these reasons, we believe that using Python in scientific computing is highly desirable and recommended.

acknowledgements

The research is supported in part by Grants F-FA-2021-432, F-FA-2021-510, and MRB-2021-527 of the Uzbekistan Ministry for Innovative Development.

[1] G. Van Rossum *et al. The python programming language, http://python.org.*
[2] M.-J. Dominus. Why not translate Perl to C? Perl.com, 2001. See http://www.perl.com/pub/a/2001/06/27/ctoperl.html.
[3] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. IEEE Computer Magazine, 1998. See http://home.pacbell.net/ouster/scripting.html.
[4] H. P. Langtangen. Python Scripting for Computational Science Springer-Verlag Berlin Heidelberg, 2004.
[5] I. Wilbers, H. P. Langtangen, and Å. Ødegård Using Cython to Speed up Numerical Python Programs In: Proceedings of MekIT'09, ed. by B. Skallerud and H. I. Andersson, pp. 495-512, NTNU, Tapir (ISBN: 978-82-519-2421-4).
[6] G. Ewing, R. Bradshaw, S. Behnel, D. S. Seljebotn et al. Cython: C-extensions for Python, http://cython.org.

Copyright © 2021