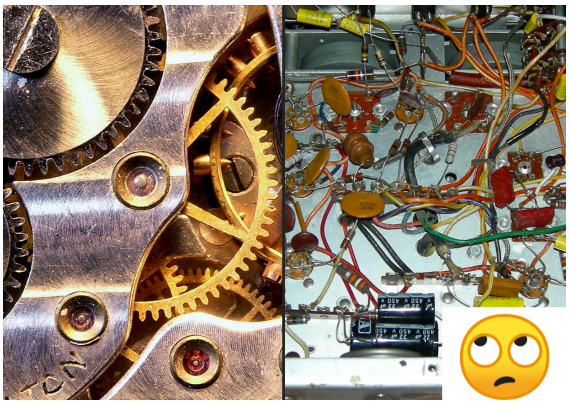
Python: Scalable from Microcontroller to Supercomputer

This post summarises a talk I gave to PyCon MEA on 19 October, 2023.

I'm really pleased to be able to speak to you in this impressive city of Dubai, and I'd like to take this opportunity to thank the organisers for the invitation.

I'm Steve Holden, and my life has for almost thirty years been intimately bound up with the Python language and the mission of its Foundation, the PSF. I no longer have any formal position with the foundation but I do, however, still strongly subscribe to its mission. I've been involved with Python for a long time - getting on for thirty years - and I'll be describing why as a part of this talk.

I started programming before the majority of people in this room were born. I'll mention the date shortly. This is a happy thought because, after a career mostly spent building software systems, it is still something I enjoy doing a great deal. From that point of view I have always considered myself incredibly lucky, although age now prevents me from being as productive a programmer as I once was.

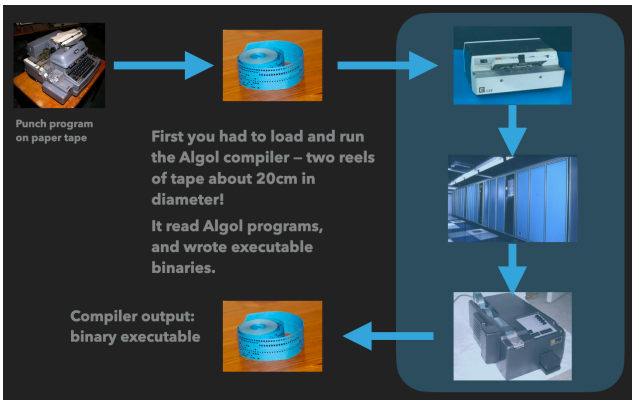


As a child I was fairly carefree, and reasonably clever, but I was never particularly happy at school. I much preferred to take apart old clocks and discarded radios that I found, or bought at jumble sales. They looked much like these pictures, although invariably grubbier.

The clocks were fun, because you could pretty much see how they worked — though I never managed to restore one to full working order! The electronics piqued my interest. What were all these strange-looking parts? What did those funny glass bottles do (no transistors back in those days)? How did these things all work?

I left school at 15 and became a trainee production engineer in a TV factory, with ambitions to work in technical development, but I soon learned those jobs were reserved for the sons of management. After a happy year in the Work Study department, learning about factory life and all the processes of TV manufacture, the company told me it had decided I should train in mechanical engineering. What? I started looking for another job - though while I was looking I did learn how to file a flat surface on a bar of metal.

I started looking around, and almost by accident I found a job at my local university's Computer Laboratory. I'd thought I'd be working with the electronics, but they actually wanted someone to operate the computer. Much of my time was spent keeping the computer running, but I also learned Algol 60, and in March 1967 I wrote a simple program to print the squares of the integers from 1 to 10. Please raise your hand if you were born after then.



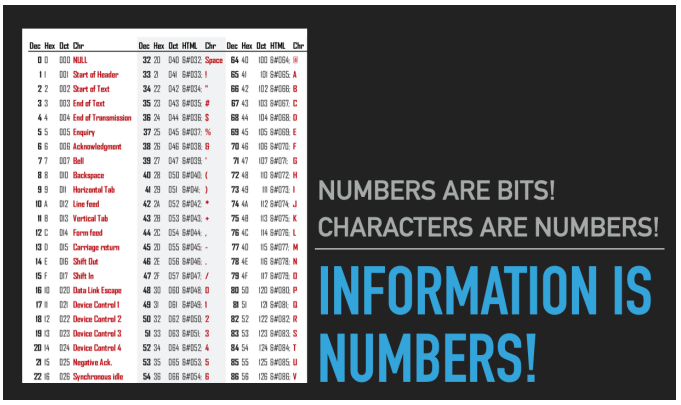
Computer operations were very different in those days. The computer only had paper tape peripherals. You had to key your program in on a Flexowriter, which punched it on to paper tape. If you've ever had to edit paper tape you would never complain about vi or emacs again.

The compiler was a large program, contained on two 20cm rolls of tape. We had to read that into memory, and it would then read each program, and punch the resulting executable program onto tape. Another tape contained your compiler error messages, which had to be printed up on a Flexowriter so you knew what was wrong with your program

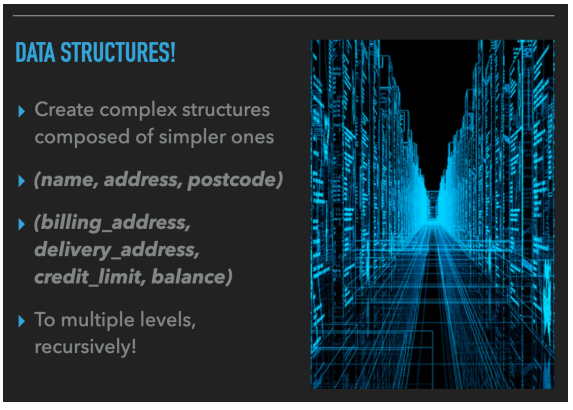
Then you had to take each executable program, read it into memory, link it with library routines (another 20cm tape reel) and run it. It would punch its results tape, and finally you had to run that through the Flexowriter to print it out. It wasn't a very convenient process!

A later program I was asked to write converted eight-hole paper tape to seven-hole, so we could use an older Flexowriter, that used a different code, to print the output when the computer got busy. Algol wasn't good at I/O so I had to learn assembly language.

The idea that the same information could be represented in different ways intrigued me. Algol 60 was a fairly limited language, and paper tape was an amazingly limiting medium, but they were quite enough for a beginner.



In my second and third jobs I got to use PL/1, which had features for describing record structures. I was introduced to disk files, and indexed sequential file access.

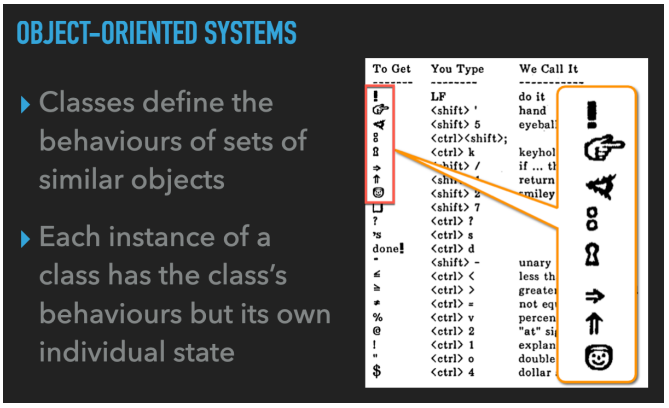


I started to read about the work on things like compiler-compilers, new languages like LISP and SNOBOL, as well as research into different types of computer such as stack-based architectures. Those were heady times for a young man.

I realised that real-world objects could be represented by data structures, and real-world relationships by references between the

structures. In essence, any information could be represented in digital form (modulo quantisation noise for analogue values)!

Data structures could refer to each other, recursively, and it was possible to create arbitrarily precise descriptions of complex objects in large sets of complex relationships.



After eight years happily learning and working I realised that my career options would be limited without a degree. So I went to university at the age of 23, which was relatively unusual in those days, and it was there I came across the early research papers on SmallTalk from Alan Kay's group at Xerox PARC.

At this stage SmallTalk wasn't available in the UK, so instead I learnt Simula, an object-oriented simulation language that had inspired SmallTalk. It was a less exciting language, but it let me experiment with object-oriented concepts.

After graduation I spent a couple of years as a research assistant and another couple building commercial systems. Then I started teaching Computer Science at Manchester, and visited Xerox PARC on a research trip, returning with a tape containing the SmallTalk 80 virtual machine. A research student ported it to the Vax-11/750, with an early Three Rivers PERQ workstation as a display.

Practical experience with SmallTalk left me feeling it was insufficiently expressive — it didn't feel like a natural way to express algorithms, which is ultimately what programming is about. Its lack of operator precedence wan unnatural, and the Smalltalk Virtual Machine was pretty much a hermetically sealed environment, with no easy way to communicate with the host operating system's resources.

For someone as practically-minded as me, it seemed far from ideal. It seemed as though there was no object-oriented system approachable enough for widespread use, so I moved into other areas. Over the next ten years I worked for a small startup called Sun Microsystems, created a business selling electronic publishing systems, and eventually became a consultant and technical trainer.

Almost thirty years into my careerYou're perhaps wondering where Python comes into the story, a! My training work was increasingly taking me to the USA, and I ended up moving to

the Washington, DC area in 1995. One of the classes I taught used some Python in its setup, and I'd developed a certain curiosity about it. One evening while waiting for a restaurant table I came across a copy of Learning Python and bought it.

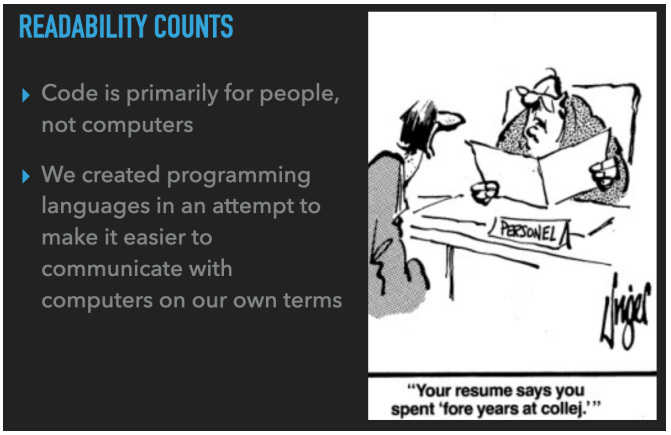
Within three days of writing my first Python program I was convinced I had found the language that would make object-oriented concepts accessible to a broad audience and yet support procedural programming in a natural way. Things don't succeed without significant effort, and over the next few months I decided I would make it my business to make Python a widely-used, popular language.

I had finally found a programming language I loved, for which I have always been grateful to Guido van Rossum, the language's inventor. You may ask, what was it about Python that appealed so much to me? If I was forced to sum it up in a single sentence, I suppose I'd say: I like the language's elegance and economy of design.

It has most of the properties I liked about Smalltalk: it is a modular, dynamic language, with late name binding. It uses exceptions, which are an elegant way to handle complex control flows and error conditions. On top of that it has sensible string handling and a judicious selection of basic data types.

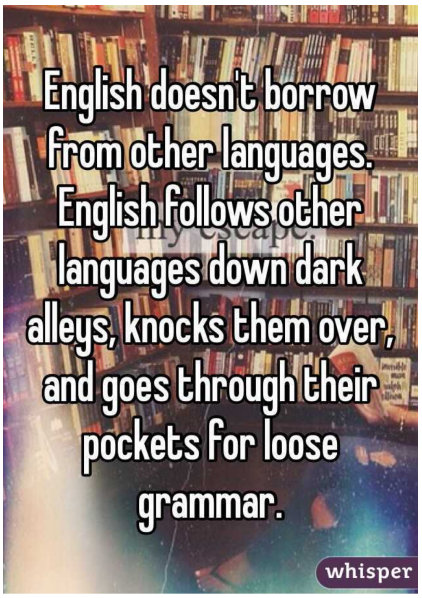
Unusually, and in the past even controversially, the block structure of a program is indicated by indentation rather than using braces or keywords to delimit code blocks. There's a standard library that gives access to network protocols, arbitrarily complex stored data and many other features, and the ability to package complex functionality in ways that encourage sharing and re-use. So what are the design principles that has made this language so widely used?

When you write a piece of code there's a relatively simple way to find out whether it makes sense to the computer: you just key it in and run it. Unfortunately there is no similar simple test to find out whether code makes sense to human beings.



Its emphasis on readability is an important part of Python's appeal. There are two contrasting approaches to language design: either make it hard to write bad programs, or make it easier to write good ones. Python very definitely falls into the latter category.

It is sometimes jokingly described as “a language for consenting adults,” because it tries to give the programmer as much freedom as possible. Rather than trying to protect you from your mistakes, it tries to make it less likely you’ll make them.



Guido van Rossum wasn’t afraid to use concepts from other languages. This is a perfectly reasonable thing to do, given that there is nothing new under the sun. Python’s object model is remarkably like SmallTalk’s. The first, and still the dominant Python implementation is written in C, and many of Python’s early capabilities were and still are based directly on standard C libraries.

Python’s comprehensions were inspired by Haskell, its dictionaries by Perl hashes, and its scoping rules by LISP. There is little point trying to be novel when perfectly adequate solutions already exist. The genius of Python’s design is to weave all these features together so naturally.

The simpler something is, the easier it is to use. Python’s simplicity makes it accessible to a wider audience. Here’s the classical “simplest program” in C and Python. Here are some questions a beginner might reasonably ask about the C:

```
C
#include <stdio.h>
int main()
{
  printf("Hello, World!");
   return 0;
}

Python
print(”Hello, world!”)
```

What does the “include” do, and why does it have a hash mark before it? Why do I have to write a function with a specific name, main - and what’s a function, by the way?. What are the curly braces for/? Why printf instead of print, and what does return 0 imply?

In Python, by contrast, the beginning programmer doesn’t have to conform to any structure, making the solution about as simple as it could possibly be.

This is a very important point for beginners: learning to program places a huge cognitive load on the learner. By minimising the initial load Python lets beginners easily see some early results, encouraging them to try more - it almost draws them into programming!

When Python was invented it wasn’t unusual for programming errors to result in debugging tasks that involved poring over a memory dump

Python handles whatever comes up, including errors from operating system calls and libraries, by raising exceptions. Programs can catch and handle all kinds of exceptional conditions appropriately.

At the lower right you see the simplest recursive function: it simply calls itself. The single call to the function triggers a potentially infinite recursion. Eventually the recursion level hits a defined limit (also under programmer control) and raises a RecursionError exception, which the program catches and continues.

Python has never been obsessed with speed because it accepted the overhead of late name binding right from the start. It didn't really start to focus hard on efficiency until version 1.5 in around 1995, which ran at twice the speed of the preceding release.

Early users were attracted to it because of its utility rather than its speed — it proved very effective for writing network-based programs, for example (the original YouTube site was programmed almost exclusively in Python). It was relatively good at text handling, and its object-oriented capabilities simplified complex tasks.

Python made some difficult tasks relatively easy, and there are many areas of computing where performance isn't especially important. In such areas, you can get a lot of utility with relatively poor performance.

Another thing I like about the language is its catholic approach to computing.

The Python world is happy to acknowledge that it is only one of the many choices for today's programmer. While it would be foolish to deny that many wheels have been reinvented in Python, there are some things which it is unlikely to be terrifically good at any time soon

An extremely popular need among scientists and technologists is numerical computing, which has come to the fore dramatically as more and more scientists and technologists are finding a need to deploy computers in their work.

Python was initially thought unsuitable because it allocates objects, including numbers, dynamically, and each carries a significant memory overhead. For large numerical computations Python itself had no native way of efficiently working with large numeric arrays packed into consecutive memory locations.

Over twenty years ago libraries to handle such problems were developed in Fortran and other languages. The Python scientific community has collected those libraries and packaged them together so you can install them with a single short command line: `pip install numpy`.

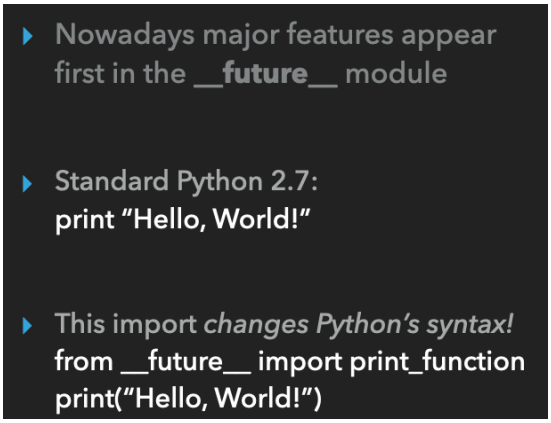
Python embraces its ability to give the user almost direct access to operating system capabilities and I/O. This helps programmers by offering a unified interface to common

functions such as filestore and networks across a range of platforms, meaning most Python programs are naturally portable.

Python network interfaces are based directly on the standard Berkeley socket library, making it relatively easy to adopt for programmers of other languages who are already familiar with it.

Many programs are written to determine a specific answer rather than being run repeatedly. Speed of development is, under many circumstances, much more important than maximum execution speed.

Besides, Python's extensibility and ability to work with programs in other languages meant that many performance problems can be solved by rewriting relatively small performance-critical sections of the code in C as an extension module.



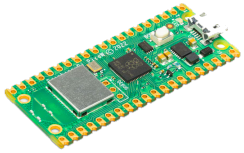
Nowadays significant new features are handled through the `__future__` module. The first example shows Python 2's standard behaviour. You will note that it differs from the earlier example, because in Python 2.7 **print** is a keyword that introduces a statement. The import in the second example represents a back porting of Python 3's print function. **print** is no longer a keyword, can be used in any expression, and even redefined if you choose to do so.

Imports from `__future__` are unusual in being able make quite intimate changes to the way the interpreter behaves, as the example shows. (That's why such imports have to be the very first thing a program does). `print_function` was introduced into the `__future__` in Python 2.6 to allow programmers to start writing code that would be Python 3-compatible.

This allows programmers to gain experience with upcoming features while making it explicit that the feature set is not frozen, and may be changed in, or entirely absent from, subsequent language versions.

A further reason for Python's popularity nowadays is the huge range of applications and tools available. All of the projects on the left are curated by NumFocus Foundation, and they are all aimed at making scientists' lives easier. The web is well-represented, as is email - it's said that the

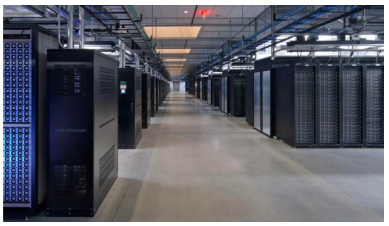
Mailman package handles about one email in six going across the Internet backbone.

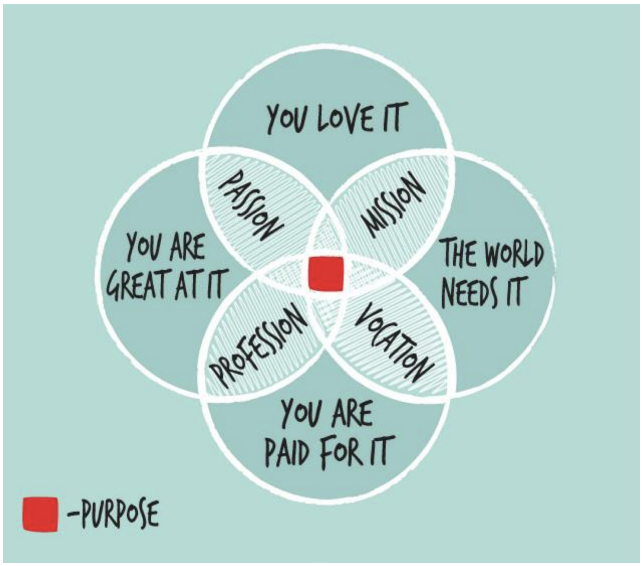


This brief summary of Python's advantages might help to explain the multitude of implementations now available. Here's a tiny device that runs Python - it's a complete computing system on a chip, which is typical of an ever-expanding range of micro controller devices targeted by MicroPython.

Python is also used to drive GPUs that offer the very fastest numerical and graphical performance available, so it's used in rendering farms by companies like Disney. Even on classical CPUs there are speedups available with Numba, Python and PyPy.

Python is quite at home running hugely parallel computations in a number of interesting and different ways. The fact that a single language can be well-suited to such a wide range of environments and applications speaks far more eloquently than I can about Python's merits. If you're looking for a new language to experiment with then I'd definitely encourage you to try it.





This graphic pretty well summarises how this mission to popularise Python gave purpose to my professional life - though it's actually the connections with people that has kept me using Python.

As Brett Cannon said: "I came for the language and stayed for the community." Without its vibrant community I think it's very unlikely that Python would be as popular as it is.

This explains, I hope, why I think Python is such a magnificent achievement, and why it's so popular today. I feel my 1995 self chose wisely to invest so much time in Python. My involvement with the language has brought me into contact with a large number of wonderful people, and has taken me to places I never thought I would visit. After all, if it weren't for Python I wouldn't be here in Dubai.