

Steve Holden

Chief Technology Officer
Felix (Global Stress Index)
sh@felix.com

Python: Scalable from Microcontroller to Supercomputer

Ore\ Conference, 30 March, 2019. Athens, Greece

Good morning everyone!



It's a pleasure and an honour to be opening Python Day for the very first **Ore** conference. This is my first visit to the beautiful city of Athens, but I sincerely hope it will not be the last. The opening picture was taken in the Amazonian jungle, while I was on a trip to Argentina and Brazil

as the keynote speaker to those two countries' annual Python conferences. So Python has on occasion been very good to me.

I'm Steve Holden, and my life was for twenty years intimately bound up with the Python language and the mission of its Foundation. I notice I was billed as representing the PSF, so I should point out that I no longer have any formal position with the foundation and therefore do not speak for it. I do, however, still strongly subscribe to its mission.



Here you see the short version of that mission's statement, and today's audience demonstrates that the mission is being fulfilled.

It's strange the way talks develop. After I'd decided on the title of the talk I was asked to focus on Python's design. If this isn't what you were expecting,

don't worry, it isn't really what I was expecting either, but I hope it will be of interest to a Python and cloud audience. It's what emerged from the process. Python audiences are usually friendly, anyway.

You might wonder what makes someone so passionate about a programming language that they dedicate such a large amount of their life to it. In order to explain that, I would like to go right back to the beginnings of my passion for programming, which I started before the majority of people in this room were born. You can check the date shortly.



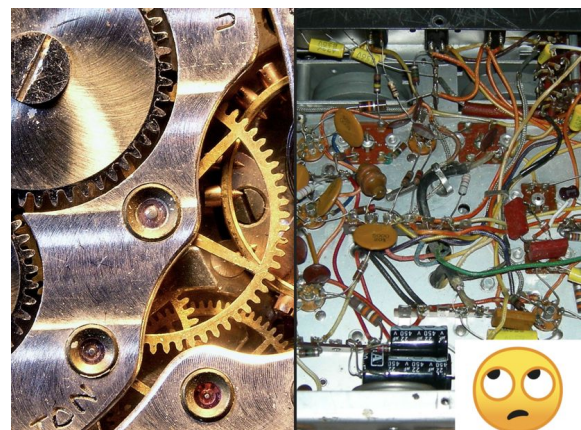
This is a somewhat humbling thought, because 50 years experience has taught me that I still make avoidable mistakes on a daily basis. It's also quite a happy thought because, after a career mostly spent building software systems, it is still something I enjoy doing a great deal. From that point of view I have always considered myself incredibly lucky.

The theme for today is "The Practical Applications and Theoretical Foundations of Python". I have always believed that practical experience should come before theoretical knowledge. It isn't appropriate to get too deeply into practical details in a keynote address, though I will be showing you some code, but if this is a typical Python conference then I am sure that you will be hearing plenty about those details. If you are new to the language I hope you will be both fascinated and impressed by its abilities.

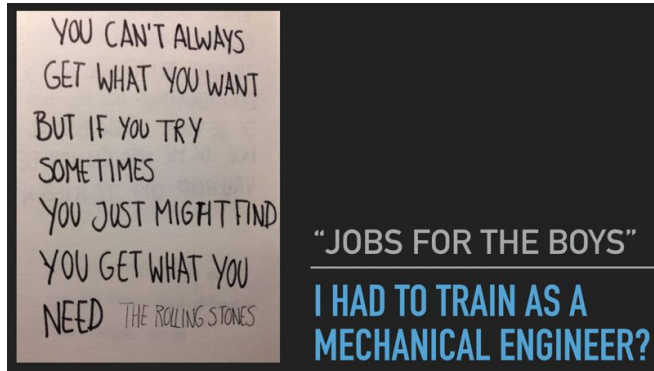
Despite its title, this talk has ended up being more about Python's fundamentals than the many uses to which it is nowadays being put. Your favourite search engine will tell you all you need to know about everything people are doing with Python.

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?

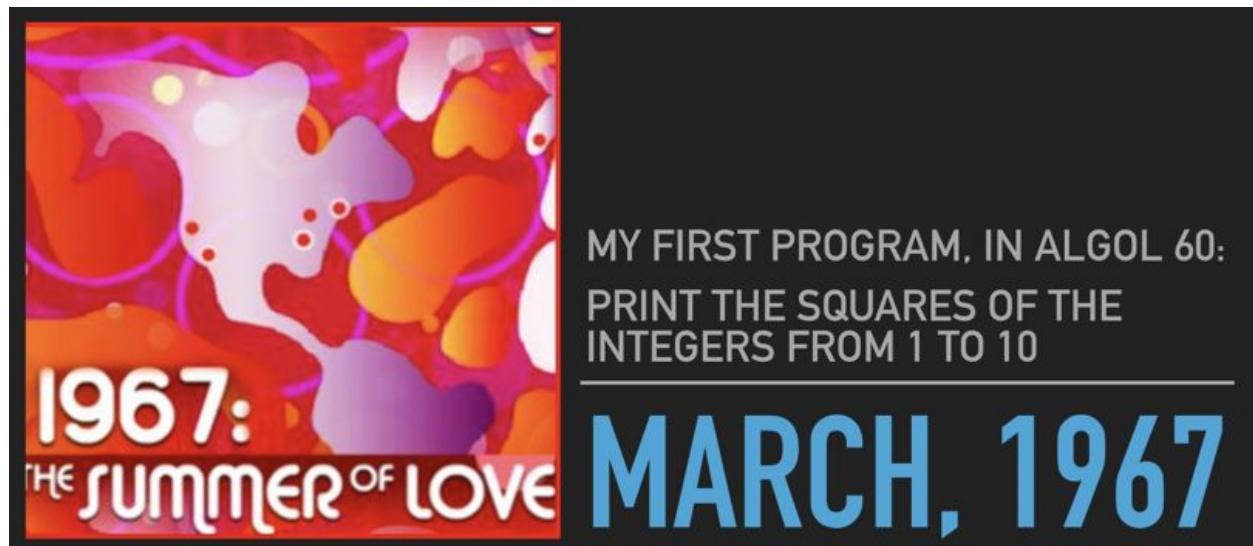


I left school at 15 and became a trainee production engineer in a TV factory, with ambitions to work in technical development. I soon learned those jobs were reserved for the sons of management, and so I should forget about them.

I spent a happy year attached to the Work Study department, learning about factory

life and all the processes of TV manufacture. At that point, the company told me it had decided I should train in mechanical engineering. What? I started looking for other work - though while I was looking I did learn how to file a flat surface on a bar of metal.

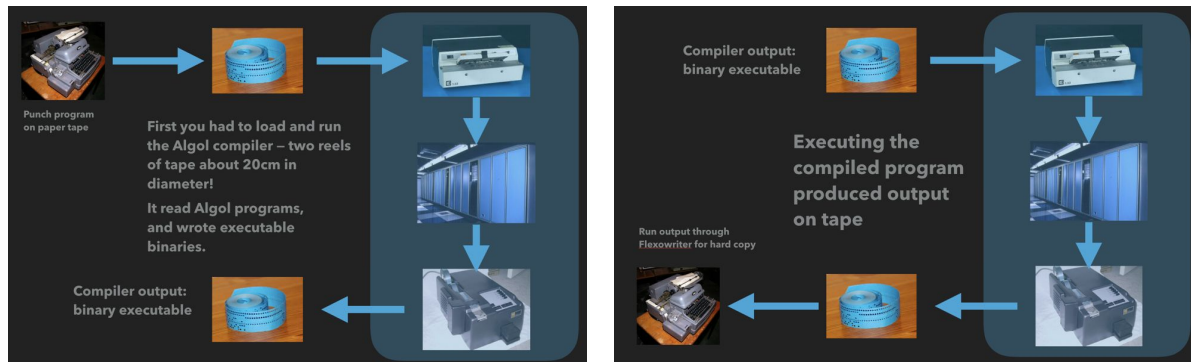
Almost by accident I found a job at my local university's Computer Laboratory. I had thought I'd be working with the hardware, but they actually wanted someone to operate the computer.



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. *[Over 90% of hands were raised]*.

Computer operations were very different in those days. The computer only had paper tape peripherals. You had to key your program in to a Flexowriter, which punched it on to paper tape.

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, if you were lucky.



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. If you've ever had to edit paper tape you would never complain about vi or emacs again!

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!

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!

Dec	Hex	Oct	Chr	Dec	Hex	Oct	Chr	Dec	Hex	Oct	Chr
0	00	000		128	80	000	Space	64	40	000	A
1	01	001	Start of Header	129	81	001	A	65	41	001	B
2	02	002	Start of Text	130	82	002	A	66	42	002	B
3	03	003	End of Text	131	83	003	A	67	43	003	C
4	04	004	End of Transmission	132	84	004	A	68	44	004	C
5	05	005	Enquiry	133	85	005	A	69	45	005	D
6	06	006	Acknowledgment	134	86	006	A	70	46	006	E
7	07	007	Flag	135	87	007	A	71	47	007	F
8	08	008	Breakspace	136	88	008	A	72	48	008	G
9	09	009	Horizontal Tab	137	89	009	A	73	49	009	H
10	0A	010	Line Feed	138	8A	010	A	74	4A	010	I
11	0B	011	Vertical Tab	139	8B	011	A	75	4B	011	J
12	0C	012	Form Feed	140	8C	012	A	76	4C	012	K
13	0D	013	Carriage Return	141	8D	013	A	77	4D	013	L
14	0E	014	Shift Out	142	8E	014	A	78	4E	014	M
15	0F	015	Shift In	143	8F	015	A	79	4F	015	N
16	10	016	Device Control 0	144	90	016	A	80	50	016	O
17	11	017	Device Control 1	145	91	017	A	81	51	017	P
18	12	018	Device Control 2	146	92	018	A	82	52	018	Q
19	13	019	Device Control 3	147	93	019	A	83	53	019	R
20	14	020	Device Control 4	148	94	020	A	84	54	020	S
21	15	021	Device Control 5	149	95	021	A	85	55	021	T
22	16	022	Device Control 6	150	96	022	A	86	56	022	U

NUMBERS ARE BITS!
CHARACTERS ARE NUMBERS!
**INFORMATION IS
NUMBERS!**

The second 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 was an interesting challenge. The second 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.

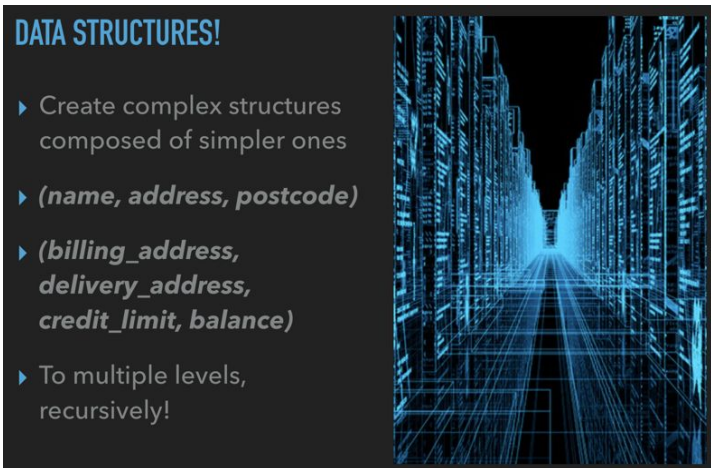
In my second and third jobs I got to use PL/1, which had features for describing record structures. I was introduced to disk files, and indexed sequential file access.

While programming commercial systems I started to read about the research that was going on into 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 programmer.

I realised that real-world and imaginary objects could be represented by data structures, and real-world and imaginary 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 had discovered enough about the way the world worked to realise that I would be limited in my career options without a degree.

So I went to university at the age of 23, and it was there I came across the early research papers on SmallTalk from Alan Kay's group at Xerox PARC. At that stage SmallTalk wasn't even ASCII-based, but it introduced me to object-oriented principles. It seemed quite natural to me that an object's description should include a list of available behaviours.



OBJECT-ORIENTED SYSTEMS

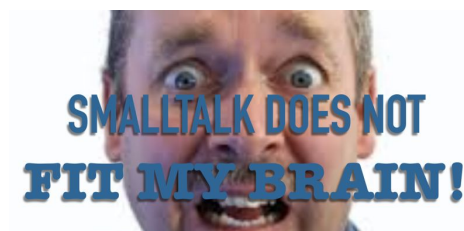
- ▶ Classes define the behaviours of sets of similar objects
- ▶ Each instance of a class has the class's behaviours but its own individual state

To Get	You Type	We Call It
LF	<shift> '	do it
<shift> 5	<ctrl><shift>;	hand
<ctrl> k	<shift> /	eyeball
<shift> 2	<shift> 7	keyhol
<ctrl> ?	<ctrl> s	if ... th
<ctrl> s	<ctrl> d	return
<ctrl> d	<shift> -	smiley
<shift> -	<ctrl> <	
<ctrl> <	<ctrl> >	
<ctrl> >	<ctrl> =	unary
<ctrl> =	<ctrl> v	less th
<ctrl> v	<ctrl> 2	greater
<ctrl> 2	<ctrl> 1	not eq
<ctrl> 1	<ctrl> 0	percen
<ctrl> 0	<ctrl> 4	"at" si
<ctrl> 4		explan
		double
		dollar

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

After I graduated 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 and I 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 message-passing technique made operator precedence impossible, 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.

My training work increasingly took me to the USA, and I ended up moving to the 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. Procedural programming also seemed quite natural in Python, and I felt lots of people could understand and use it.

By that time I was experienced enough in technology trends to realise that things don't succeed without significant effort, so I determined I would invest twenty years in working to make Python a widely-known language. Interestingly, quite a few other people appear to have made similar decisions around that time, and many are still involved with Python.



**A FLOWER CANNOT BLOSSOM
WITHOUT SUNSHINE, AND MAN
CANNOT LIVE WITHOUT LOVE.**

Max Muller

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.

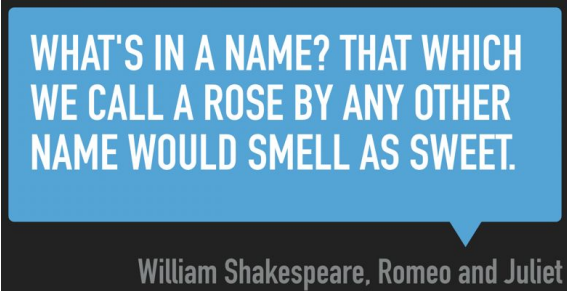
It has most of the things I liked about Smalltalk in it: it is a modular, dynamic language, with late name binding. It uses exceptions, which are an elegant way to handle complex control flows and error conditions. On top of that it has sensible string handling and a judicious selection of basic data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

Unusually, and somewhat controversially, the block structure of a program is indicated by indentation rather than using braces or keywords to delimit code blocks. It also has 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 a nicely modular form. What are the design principles that has made this language so widely used?

Name binding strategies are very important in determining the characteristics of a programming language, and Python chose a late binding strategy.

In C and other *static* languages the compiler determines memory allocations in advance, at compile time. This in turn requires it to know the sizes of the values to be stored, so programs have to include type declarations for each variable. If a program needs to use memory dynamically, it has to incorporate its own logic for allocation and deallocation, typically using something like the **malloc** library.

In Python and similar *dynamic* languages, name bindings are not set in advance, and are looked up in so-called namespaces at run-time. Variables are simply the names of references to data objects (and therefore all the same size), while the objects themselves are automatically allocated from, and returned to, a storage heap. Programmers do not have to concern themselves with the details of memory allocation *at all*, and the data type is associated with the value, not the name. Polymorphism is easy.



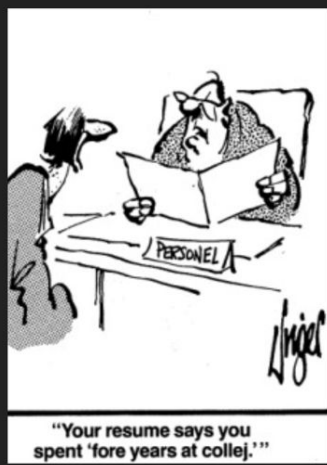
WHAT'S IN A NAME? THAT WHICH
WE CALL A ROSE BY ANY OTHER
NAME WOULD SMELL AS SWEET.

William Shakespeare, *Romeo and Juliet*

Late name binding is a mixed blessing: it dispenses with the necessity for declarations and makes polymorphism practical. But the need to look up names adds significant overhead to execution times.

READABILITY COUNTS

- ▶ Code is primarily for people, not computers
- ▶ We created programming languages in an attempt to make it easier to communicate with computers on our own terms



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 appears to be no similarly reliable test to find out whether code makes sense to human beings.

Its emphasis on readability is an important part of Python's appeal.

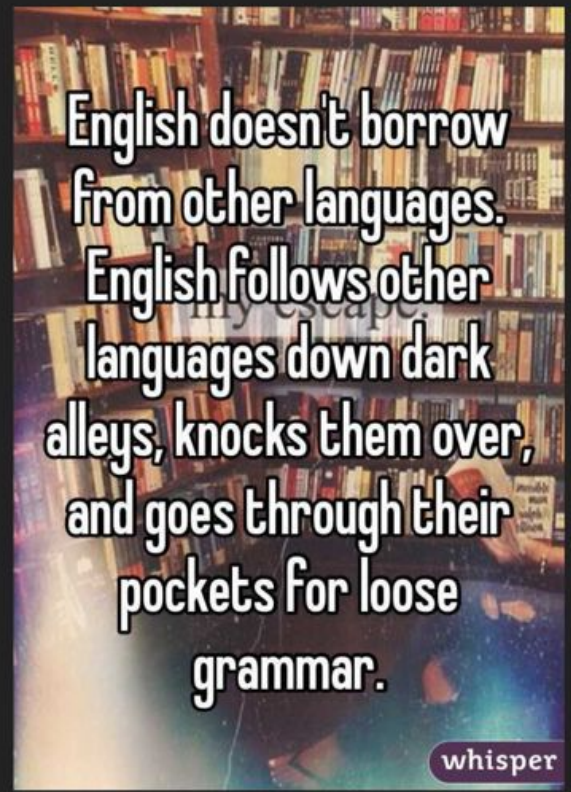
There are two contrasting approaches to language design: either make it hard to write bad programs, or make it easier to write good ones. Python very definitely falls into the latter category.

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.

I got an extra smile from this cartoon when I realised the personnel manager couldn't spell either.

BORROW FROM ELSEWHERE ...

- ▶ ... whenever it makes sense
- ▶ Good ideas are good wherever they come from!



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.

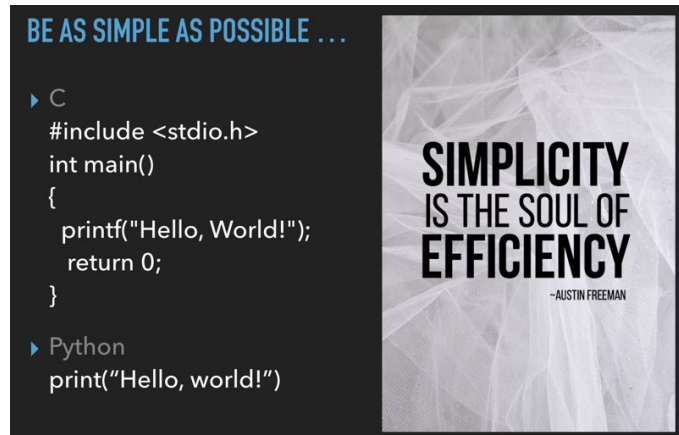
The first, and still the dominant Python implementation is written in C, and many of Python's early capabilities were and still are based directly on standard C libraries.

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 are available. The genius of Python's design is to weave all these features together so naturally.

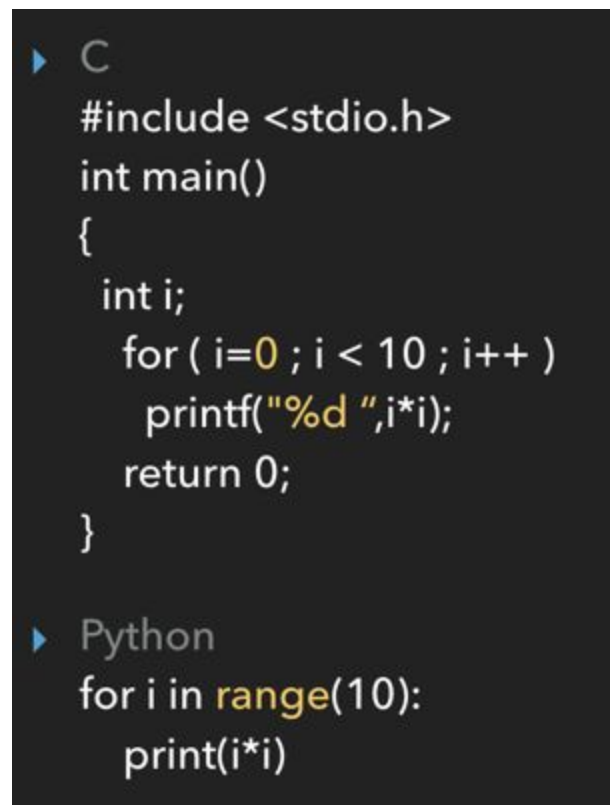
The simpler something is, the easier it is to use. Python's simplicity makes it accessible to a wider audience. Compare the classical "simplest program" in C and Python. I am obliged to point out that my C knowledge is seriously out of date, so my examples may not represent best practice. Also, please consider my remarks about C as being representative of static languages generally rather than as aimed at C in particular.

Note how the C program requires the programmer to reference a library “include” file for access to even the simplest I/O functions, and to write a function with a specific name, **main**.

In Python, 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 beginner: learning to program places a *huge* cognitive load on the learner. By minimising the initial load Python lets beginners easily see some results, encouraging them to try more.



These slightly more complicated programs show some other aspects of Python’s simplicity and elegance of design.

On the left we see the C and Python equivalents of my first Algol program. The C example has to explicitly set the initial value for the loop, provide a test that must be true for the loop to proceed, and give an expression to evaluate between iterations. It also has to explicitly convert the number to a string before its value can appear in the outside world.

In Python the programmer simply iterates over an object that produces a sequence of integers, printing each one’s square.

The example on the right emphasises even further the differences in **for** loop design.

In C, the programmer has to say how many elements are in the array, even though the elements are right there in the program. The array is a simple data structure that carries no semantic information, so its length has to be explicitly used again in the loop specification.

The Python programmer doesn't care how long the list is: each list knows how many elements it has, and when used as the object

of a **for** loop produces each member in sequence. While Python does very little in the way of implicit conversions, all Python objects can represent themselves as some kind of string, though for complex objects the string isn't terribly meaningful. Not only that, but the elements in Python lists don't even have to all be of the same type, a fact that sometimes blows people's minds.

I should remind you, before we get carried away, that once compiled all the C examples will have finished before the Python interpreter has got a significant way through its initialisation and well before it even knows what program it's supposed to run, so I am not suggesting that Python is ideal for all applications. In the interests of harmonious relations between languages, I will also point out that Python and C agree, as do all sane languages, that zero-based indexing is the way to go.

When Python was invented it wasn't unusual for programming errors to result in messages that were completely inscrutable without a programmer investing a significant chunk of time. I well remember my first PL/1 core dump.

```

▶ C
#include <stdio.h>
int main()
{
    int array[8] = {1, 2, 3, 4, 5, 6, 7, 8}
    int loop;
    for ( loop=0 ; loop < 8; loop++ )
        printf("%d ", array[loop]);
    return 0;
}

▶ Python
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
for number in numbers:
    print(number)

```


Early users were attracted to it because of its utility rather than its speed – it proved very useful 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.

While there has sometimes been joking talk about Python's "march towards world domination," 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

PLAY WELL WITH OTHERS

- ▶ Python is not perfect
- ▶ Other languages will be better at some tasks
- ▶ Incorporating other languages gives Python more power!
- ▶ e.g., Numpy, a widely used numerical package
- ▶ Uses code written in Python, C++, C and Fortran!



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.

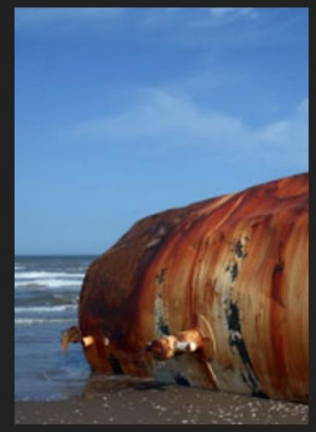
Over twenty year ago libraries to handle such problems were developed in Fortran and other languages. With a great deal of work the Python scientific community has collected those

libraries and packaged them together, Pythonically, so you can install them with a single short command line: **pip install numpy**.

Unlike SmallTalk, which existed in splendid isolation, Python embraces its ability to give the user almost direct access to operating system capabilities and I/O. This helped programmers by offering a unified interface to common functions like filestore and network interfaces across a range of platforms, meaning many Python programs are portable.

DON'T FIGHT THE ENVIRONMENT

- ▶ Go with the flow
- ▶ Offer a consistent interface across platforms
- ▶ Platform-specific features are OK
 - ▶ But don't use them in programs you want to be 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.

Each platform does have some Python libraries not available for others, but if you avoid those you can easily make your programs portable.

DON'T STRIVE FOR PERFECTION

- ▶ "Good enough" is often just that
- ▶ Optimise for low development times rather than fast execution

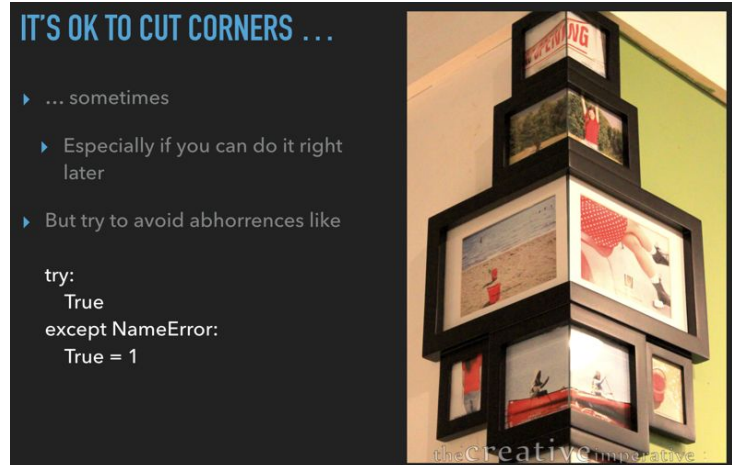
**GOOD
ENOUGH
IS THE NEW
PERFECT**

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 could be solved by rewriting relatively small performance-critical sections of the code in C as an extension module. Python developed comparatively fast in its first fifteen years. Its early development history, while not particularly chaotic, did have a few missteps, and occasionally rapid development of the language meant that features arrived less than fully ready for prime time.

The code sample on the right was used for a while, because the Boolean constants `True` and `False` were first defined in Python 2.2.3(?), so if you wanted to use them you had to make sure they were defined in earlier versions.

This led to an agreement among the core devs that no new features could be introduced in a bug fix release.



Python developed comparatively fast in its first fifteen years. Its early development history, while not in particularly chaotic, did have a few missteps, and occasionally rapid development of the language meant that features arrived less than fully ready for prime time.

The code sample above was used for a while, because the Boolean constants **`True`** and **`False`** were first defined in Python 2.2.3(?), so if you wanted to use them you had to make sure they were defined in earlier versions. This led to an agreement among the core devs that no new features could be introduced in a bug fix release. I like this discipline.

- ▶ Nowadays major features appear first in the **`__future__`** module
- ▶ Standard Python 2.7:
`print "Hello, World!"`
- ▶ This import *changes Python's syntax!*
`from __future__ import print_function`
`print("Hello, World!")`

Nowadays significant new features are handled in a rather more controlled way 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 "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.

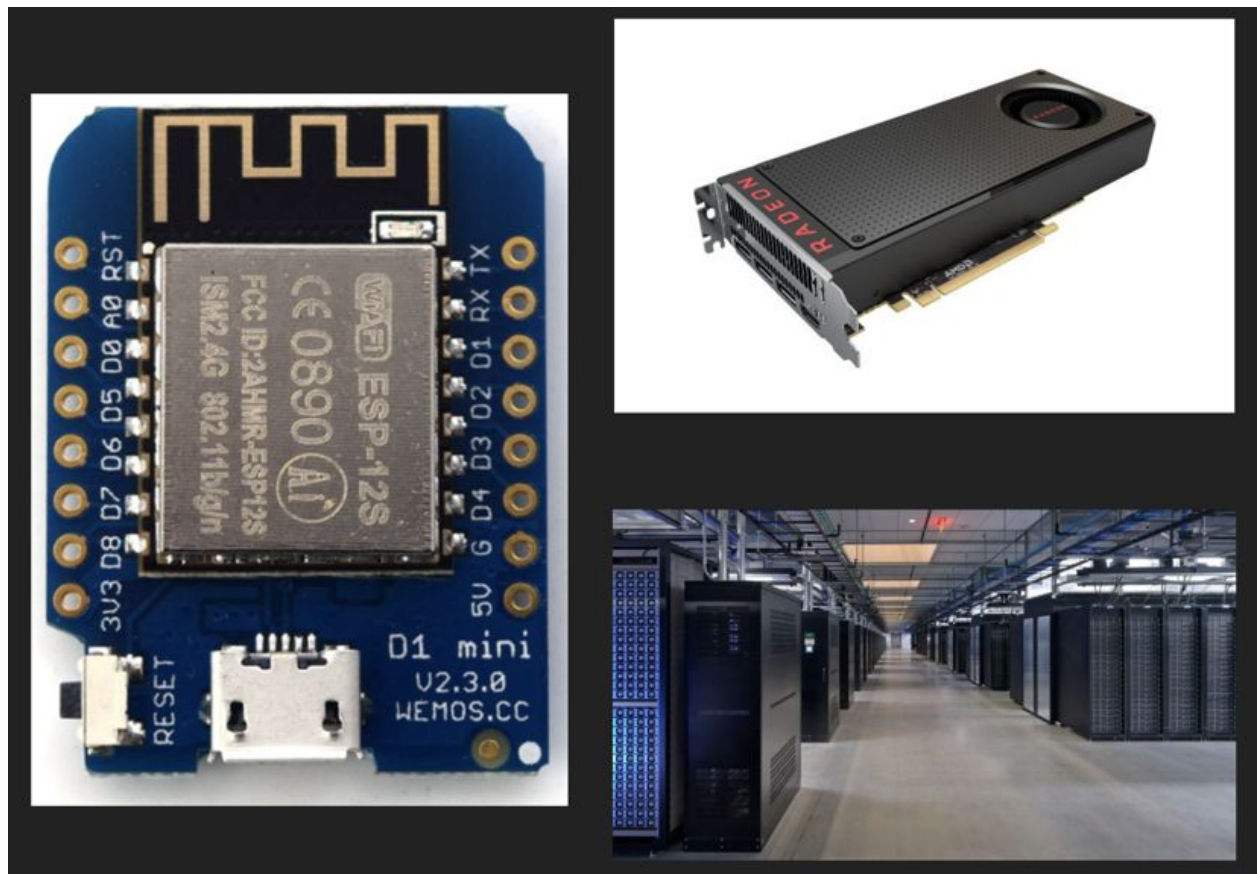


A further reason for Python's popularity nowadays is the huge range of applications and tools available. All of the projects on the left half of this graphic are curated by NumFocus Foundation, and they are all aimed at making scientists' lives easier. The web is well-represented, as is email - it's said that Mailman 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. Below on the left you see the smallest device I currently possess that runs Python - the D1 Mini, based on the ESP8266 chip, which is typical of an ever-expanding range of micro controller devices targeted by MicroPython.

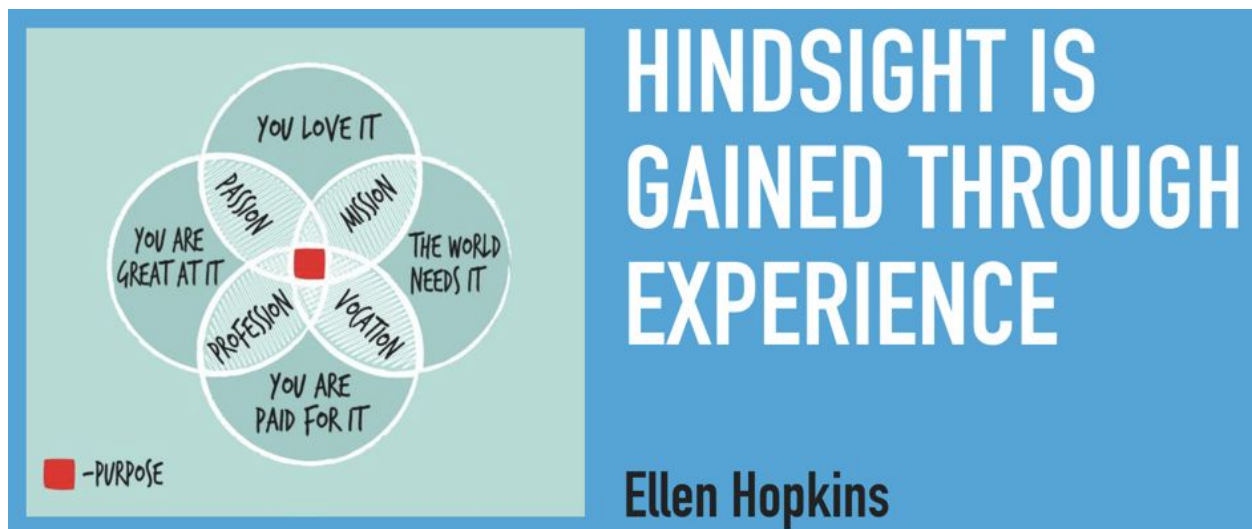
Python is also used to drive GPUs that offer the very fastest numerical and graphical performance available, so it's used in rendering farms by companies like Disney. Even on classical CPUs there are speedups available with Numba, Python and PyPy.



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 encourage you to try it.

That's my summary of why I think Python is such a magnificent achievement. I feel my twenty-five years younger self chose wisely invest so much time in Python. After all, if it weren't for Python I wouldn't be here in Athens.

The graphic on the next page pretty well describes how Python gave purpose to my professional life - though I should remind everyone who's passionate about programming that people are still much more important than computers. My involvement with Python has brought me into contact with a large number of wonderful people, and has taken me to places I never thought I would visit, and I'm happy I made the investment.



I enjoy software development so much. It still fascinates me after 50 years. Sometimes people ask me why, and I tell them it's because the only limit on what you can build are your imagination.

WHY I LOVE SOFTWARE DEVELOPMENT

- ▶ "Tell me what the rules are, I can build it for you!"
- ▶ You can (almost) build castles in the sky



It's quite possible that the image on this slide existed first as some kind of data model, which was then transformed by software into the visual representation we are now sharing.

Python engages people's imaginations, making it possible for them to think about their programs in ways that make sense in terms of their problem. There

are fewer details to keep track of, and with judicious choice of names a good Python program seems to read relatively naturally at a high level.

WHY I FEAR SOFTWARE DEVELOPMENT

- ▶ "Tell me what the rules are, I can build it for you!"
- ▶ You can enforce and monitor *any* system of rules



This freedom to create, however, has a dark side.

Perhaps the home of democracy is an appropriate place to wonder

exactly what it is that capitalist democracies are building. It seems to me we should abandon the hubristic assumption that we can somehow develop immunity from the law of unintended effects.

In my experience software developers don't take kindly to the suggestion they should accept responsibility for their creations, but "I was just doing my job" does not absolve anyone of responsibility. Being able to build something doesn't mean we necessarily should. With great power comes great responsibility, and I believe ethics will either be our saviour, or their absence our ruin. But that's a subject for another talk altogether.

Thank you very much for listening to my musings about Python. I was asked to save a little time for questions, so perhaps if there's anything you want to know, perhaps we could take some now.

