On Sunday, June 21, 1970, in an office building on Great Portland Street in London, a teletype sprang to life. Under the heading “HAPPY FAMILIES,” the machine rattled out a sequence of English sentences, such as “THE DOG SITS ON THE BABY” and “UNCLE TED PLAYS WITH SISTER.” The “Happy Families” program
that produced this output had been written that same weekend by someone with no prior programming experience, a participant in a workshop organized by the Computer Arts Society offering a course in “non-numerical programming.”¹

Almost fifty years later, on October 26, 2019, in Istanbul, the eyes of a young woman lit up as she managed to get her very first program to run. She was attending a workshop organized by Django Girls, an international non-profit organization aiming to empower women through free, one-day programming workshops.²

The programming language used at the London workshop in 1970 was TELCOMP,³ a simple unstructured language similar to BASIC—not BASIC as it is now, but unstructured BASIC as it was then.⁴ The programming language taught at the Istanbul workshop in 2019 was Python,⁵ a programming language designed by Guido van Rossum that has become wildly popular,⁶ steadily gaining in popularity since an inconspicuous public release in 1991. As far apart as these events are, both in time and geography, an arc of history connects them.

Way Back When

To fully explain the connection between these two events, I must first return to the computer programming scene in the late 1960s. The dramatis personae are Leo Geurts, then a programmer, and myself, then a junior researcher, both employed by the Mathematical Centre in Amsterdam, known at the time as “the MC,” but later renamed to Centrum Wiskunde & Informatica (CWI).⁷ The MC gave us considerable latitude to choose our own research topics, which were, generally speaking, concerned with exploring the frontiers of what computers could be made to do. For some time, we had been exploring the use of computers in art and music—not as an expensive spirograph or music box, but as a faithful servant creating works according to algorithmic rules of our own devising that, we hoped, embodied interesting creative principles. We joined the Computer Arts Society when it was founded in 1968 and became enthusiastic members,⁸ also
serving as instructors or mentors at tutorial workshops, such as the 1970 event in London. Personal computers did not yet exist and very few people had even seen a computer. The participants in the workshop—artists, composers and architects—did not know any programming languages; in two days we had to teach both a programming language and the principles of programming.

The common method of running programs in those days was batch processing. Programmers would deliver their programs as a pack of punched cards or a roll of punched paper tape at the counter of a computing center and—provided the program did not have a fatal error—could expect to collect the output in a couple of hours, or the following day. Such an arrangement would not have worked for a weekend course. But TELCOMP was what was referred to in those days as a “conversational language”; in today’s terminology, we would say that the implementation was interactive. The teletype served as a computer terminal, connected to a mainframe computer with a time-sharing system operated by Time Sharing, Ltd. After typing in the program as a numbered sequence of steps, the user could issue a DO command to run the program and obtain immediate results.

Learning the language was straightforward. TELCOMP had a repertoire of only fifteen commands and two data types: numbers and arrays of numbers. But using it to transform an idea into code became a bottleneck. The commands and types were all implemented at a very low level, a world apart from the ideas to which they were to give concrete form. As such, they did not offer a suitable means to properly structure and organize the material. Even the simplest projects quickly turned into what is called “spaghetti code”\(^9\)—an inextricable mess.

We knew, of course, that there were far better languages, such as ALGOL 60.\(^{10}\) The full ALGOL 60 language was, perhaps, more complicated than strictly necessary for this particular purpose, but we could have taught a simplified subset. We did not use ALGOL 60 because it had not been designed to be suitable for conversational use.
Programming Language Design and Simplicity

Apart from a few programming languages that were deliberately designed as a joke, such as INTERCAL, a language designer will not make a language hard to learn or hard to use by design. Ease of learning and ease of use are both desirable attributes in any programming language. Nonetheless, I have often felt that this aspect of language design does not always receive the attention it deserves. And what may seem easy to a designer may not necessarily be easy for a language learner.

First and foremost, a programming language is a tool for expressing algorithmic ideas in a form that lends itself to execution by an automaton. Functional tool design is not a science; it involves seeking a balance between potentially conflicting design objectives. A general-purpose programming language is also a tool kit, comprised of parts that must smoothly combine to work together in manifold ways, like the parts of a Meccano set can be endlessly combined. For all but the most basic languages, a designer needs to make hundreds of design decisions. Choosing judiciously from all the possibilities requires an awareness of the design space and a willingness to explore it, as well as the ability to identify trade-offs and examine alternatives. Here, a design philosophy can provide guidance to the designer. It can be laid down in a number of design rules, which cannot be hard-and-fast rules, but must still reflect the designer’s objectives.

The simplicity of a tool has two different aspects: the ease with which its functions can be learned and the ease with which it can be used once that functionality is understood. The task of a programmer is to translate an idea into code. This process will be considerably easier if the elements of a programming language are task-oriented and correspond to concepts in the programmer’s mind. At first glance, these two aspects of simplicity might appear to be at odds. Indeed, a minimalist language, while easy to learn, can make the programmer’s task difficult. Not only are more lines of code needed, but writing correct code becomes increasingly difficult as program size increases. Conversely, attempting to make something akin to a Swiss Army knife and including every imaginable feature
yields a complex language that will be difficult to learn, but surely easy to use. Or will it? Perhaps not.

A programmer has to decompose the original task into subtasks, which in turn are decomposed into sub-subtasks, and so on, until a level is reached in which the task at hand can be directly addressed using the tool set of the programming language. If that language is very rich in features, there will be many ways of doing this, and selecting among all the different options may become a serious mental burden. Making a language increasingly rich in this manner yields diminishing returns—until eventually any benefits, no matter how meager, are lost. Rather than cramming all sorts of potentially useful features into a tool set, a much better approach is to select a smaller number of powerful features that are task-oriented and work well in combination.

The ABC Project

Over the course of several years spent teaching programming workshops for artists, mostly in the UK and the Netherlands, Geurts and I became increasingly frustrated with the shortcomings of the languages available to us.\(^{12}\) “Why is there no language,” we wondered, “that combines the best of two worlds: the power and simple elegance of ALGOL 60 with the modest size and suitability for conversational use of BASIC and TELCOMP?” Our frustration provided the impetus to start a project in 1975 with the goal of designing such a language. I had observed the design process for ALGOL 68,\(^{13}\) intended as an ambitious successor to ALGOL 60, from close quarters and had also witnessed its revision between 1970 and 1974. In both cases, obvious improvements had not been incorporated because they had been proposed “too late,” so I knew it would be wise to allow for several iterations during the design process. We also decided not to settle on a name for the new language before we were happy with the result. To refer to the language, we selected the variable name \(B\) for an uninitialized string variable, serving as a placeholder until the design process was completed. The letter “B” was chosen because it is the first letter of the word “beginner” and
because the project was meant to become a language for teaching programming to absolute beginners. In keeping with the mathematical convention for variable names, we rendered it in italics. The initial approximation, $B_0$, was not yet given much thought; it was more an experiment to see what we liked and did not like. In the end, it was finished and published within a year.\textsuperscript{14} While quite different from the final results, a few things have remained the same through all iterations. The $B_0$ command \texttt{PUT 1, 2 IN A, B} assigned the value 1 to a variable named \texttt{A} and the value 2 to a variable named \texttt{B}. The command \texttt{PUT A, B IN B, A} swaps the contents of these two variables.\textsuperscript{15} In most other languages, such a swap requires three steps, using a third, auxiliary variable: (1) \texttt{PUT A IN X}; (2) \texttt{PUT B IN A}; and (3) \texttt{PUT X IN B}. In later iterations, when input-output devices allowing the mixed use of upper and lower case had become common, this became \texttt{PUT a, b IN b, a}, but the command structure for assignments and the collateral assignment of multiple values remained. More striking is the use of indentation. Although it was common in programs written in ALGOL 60 or its descendants, such as Pascal, to use indentation as a typographical layout feature for clarifying the grouping of commands, this was an entirely optional presentation choice, made purely for the benefit of the human reader. In an article by P. J. Plauger entitled “Signal and Noise in Programming Languages,”\textsuperscript{16} we found the (then) radical idea to “have the compiler read the same signal as we human beings, and let the indenting control grouping,” a suggestion we followed with enthusiasm. Indentation to indicate that a suite of commands belong together subsequently became mandatory in $B_0$ programs, a design choice that has been maintained throughout all iterations.\textsuperscript{17}

We did not set out with a clear design philosophy; it grew naturally out of the leading design principles during the design process. To serve our intended users, absolute beginners, we sought to hide low-level implementation details and instead to provide powerful high-level, task-oriented features. One of the things we valued most was consistency, both as uniformity in the presentation of the constructs of the language—“syntax”—and the absence of quirks in their meanings—“semantics.” Most programmers do not truly understand all the different aspects of their favorite language, and one cannot reasonably expect a beginner to quickly master a new language. In some ways, learning a new programming language is similar to
learning a new natural language; irregularities and exceptions are a stumbling block on the road to mastery. The natural way of learning is to learn from examples, and then to extrapolate from these examples to general rules. A toddler might say, for example, “The dog bited him,” incorrectly applying the regular rule for forming the past tense to an irregular verb. While consistency may seem desirable as well as achievable for a constructed language, this was rarely seen in practice. Most languages, or their implementations, suffered from restrictions and exceptions that were both awkward and seemingly arbitrary in nature. These make a language harder to learn and harder to use. As we progressed with the design for our new language, we saw patterns in the design decisions we took. We tried to catch each pattern and codify it as a design rule, an approach that would help improve consistency. Some of these rules had been identified before by others, such as in the article by Plauger.

The requirement for syntactic uniformity was codified in the Uniformity Rule, expressed by Plauger as the dictum: “similar things should be said in similar ways.” A related, more semantic design rule was dubbed the Fair-Expectation Rule. There is a property of human visual perception, known as the Good Continuation principle that reflects one of the Gestalt laws of grouping. When an object is only partially visible, the human visual system extrapolates from the shape of the visible part to form an expectation of what is occluded. In order to make the shape whole, the invisible part is inferred from the context. When an opaque disk covers a corner of a rectangle, for example, one tends to assume that the sides of the rectangle continue beneath it, and that the rectangle is whole. When the object occluding the view is removed, an observer may discover that they have been fooled: in reality the other object was not whole. But this is then a surprise, exploited as a trick by magicians. We wanted to spare our learners such surprises. The Fair-Expectation Rule stated that if it was fair—based on extrapolation from similar constructs—for a user to expect that some conceivable construct was valid and had a certain meaning, then the construct will be valid and have that meaning. The more closely a design follows this rule, the fairer a user’s expectation becomes that the rule will also be honored on the next occasion. In effect, this approach exerts a self-strengthening force on the development of the language. Yet another
The design requirement was the Economy-of-Tools Rule: different tools should serve genuinely different functions. If it turns out that separate functions overlap conceptually, those tools should be unified. The application of this rule led to more gaps being filled in the overall functionality of the language so as to meet the Fair-Expectation Rule.

In 1981, after two iterations, we felt confident enough to publish the design in the form of a Draft Proposal, after which I quickly wrote an implementation—meant to be a throwaway product—so that we could gather feedback from teaching the language to beginners.

The following year, I accepted a one-year appointment as Visiting Associate Professor in the Computer Science department at the Courant Institute of New York University (NYU), where Robert Dewar was an enthusiastic supporter of the $B$ effort and had overseen some student projects that had produced implementations of $B_1$. The MC invited Steven Pemberton—in what was supposed to be a one-year appointment as Guest Research Scientist—to become the development lead of a serious implementation effort during this interim period. While I was in New York, the implementation team in Amsterdam was strengthened with a new hire, Guido van Rossum.

## SETL and High-Level Data Types

During an earlier visit to NYU, I had become acquainted with SETL, a very high-level programming language that was designed by the visionary polymath, and author of several fundamental books in mathematics, Jack Schwartz. In some ways, SETL was almost the complete opposite of $B$, which was then in its first iteration, $B_1$. Based on concepts from mathematical set theory, SETL's target audience appeared to be mathematicians. By contrast, when designing $B$ we did not assume any knowledge of mathematics beyond the most elementary arithmetic and algebra. Yet the two languages had something in common on a deeper level: a design philosophy centered on providing
programmers with only a handful of tools, but very powerful ones. SETL embodied a radical departure from a mood that until then had hung over high-level language design, including that of ALGOL 68. The designers of these languages and their colleagues came from an era when processor instructions were not timed in microseconds but in milliseconds. Computations that took an hour in the early 1980s would have taken many weeks in the early 1950s. For decades, computer time had been viewed as incomparably more precious than programmer time, and this had left a mark on the mindset of a whole generation of programmers and the languages they designed. The constructs of programming languages were thus designed to have a rather straightforward mapping to the hardware instruction sets of computers. But, of course, this mapping was not perfect; some loss in execution speed, in comparison to carefully hand-crafted programs written directly in machine code, was inevitable. Diatribes against the pernicious influence of computer-time-consuming high-level languages were not uncommon even in the 1960s. The design of SETL was predicated on the philosophy that, as computers became faster and faster, human time would be more valuable than computer time, and that the efficiency of the tool set SETL provided should not be measured in terms of speed of computation, but in their conceptual power.

Having coded a few algorithms in SETL, I had experienced its power firsthand—a power that stemmed entirely from its high-level inbuilt data types. Particularly powerful were sets and maps, also known as “associative arrays,” containing data that can be indexed not only by consecutive integers but by arbitrary values. A programmer could introduce a simple database of quotations named whosaid, in which the value “Descartes” could be stored in the location

whosaid[“cogito ergo sum”]. These high-level types made it possible to express algorithms that required many steps in $B_1$ using just a few steps in SETL. In a clear violation of the Fair-Expectation Rule, $B_1$ allowed only integers as array indices. This design decision had been driven by fear: we had been concerned that aiming too high would make our language unimplementable on the small personal computers that were starting to appear on the market. But Dewar, in particular, convinced me that this meant we were designing for the past, not the future. This led us to redesign the system of data types for our beginners’ language. This time
we used only the criteria of ease of learning and ease of use to select candidate systems. The winner turned out to be remarkably similar to the data type system of SETL. The set of possible data type systems to choose from was very large, and to make the process more manageable I had written a program to select the competitive (Pareto-optimal) candidate systems. Interestingly, but quite incidentally, that selection program itself was written in SETL. The winning type system became that of $B_2$, and remained unchanged in the final iteration, released in 1985 under the name “ABC.” Expressed symbolically, we executed the command

```
PUT “ABC” IN B.21
```

Ten years had now elapsed since the start of the $B$ project.

An End and a Beginning

In September 1983, I returned to Amsterdam, to the MC, which had been renamed during my absence and was now the CWI. Computer science, *informatica* in Dutch, had played an important role in the research undertaken at the MC throughout its existence; the name change made this explicit. Pemberton remained on the team as the third principal designer of ABC. In the final iteration of the design, leading from $B_2$ to ABC, we reexamined almost every aspect of the language to see if any further improvements were possible, no matter how small. We held weekly team meetings in which all team members participated. Van Rossum regularly proposed changes. Some of his suggestions were adopted, many were not, but all were seriously considered. In programming-language design it is rare that a proposed change has only advantages; it almost always has some advantages and disadvantages. Weighing these factors against one another to reach a verdict can be a delicate matter. In many cases, a proposal would violate one of the design rules, which were an imperfect codification of the overarching design philosophy.

In 1984, the project obtained an IBM PC, and in less than a year we had ABC running on the machine and ready to be distributed to the world.

A simple example may serve to give an impression of the vast difference between
TELCOMP and ABC. Bubble sort is a well-known but rather inefficient sorting algorithm, particularly for larger data sets. Yet, because of its simplicity, it is often used to illustrate elementary algorithmic concepts and to provide a first-glance comparison between programming languages. The following code examples show how bubble sort looks expressed in TELCOMP and in ABC.

**TELCOMP**

1.01 DO PART 2 FOR I = 1:1:N-1

2.01 DO PART 3 FOR J = 1:1:N-1


4.01 SET X = A[J]
4.03 SET A[J+1] = X

**ABC**

FOR i IN {1..#a-1}:
    FOR j IN {1..#a-1}:
        IF a[j] > a[j+1]:
            PUT a[j+1], a[j] IN a[j], a[j+1]

The numbering preceding each of the TELCOMP commands is indispensable; it
serves as a label to specify a continuation of the computation. There is no requirement for anything like this in ABC. The TELCOMP program consists of four numbered “parts,” which are like subroutines named by numbers. Each part consists of one or more numbered steps. After completing the steps of each part, control returns to the continuation of the step from which the part was invoked, a **DO PART** command. The TELCOMP version requires that the value of the variable N has been set in advance to the length of array A. After entering these commands, the user still has to give the direct command **DO PART 1** to start the sorting process, whereas the ABC command can be used directly.

ABC did not become the success we had hoped for. When we started the project, we naively expected it was only a matter of time before learning the first principles of computer programming became part of the standard high-school curriculum, for which ABC would be the perfect fit. In some schools, teachers did offer experimental programming classes using ABC. But, when “informatica” finally became part of the Dutch high-school curriculum at the end of the 1980s, it turned out to be nothing but a course in using a word-processor (WordPerfect) and a spreadsheet (Lotus 1-2-3), both products now only a faint memory. A serious obstacle we faced was finding a way to inform the remaining plausible target group of users—non-professional computer hobbyists—that the language existed. The Internet as we know it did not yet exist. Its precursor, ARPANET, was reserved for academic and military use. For whatever reason, the CWI directors did not approve the idea of placing an ad in *Dr. Dobb’s Journal*, then the leading journal for computer hobbyists. The only thing we could do was to mail copies of the software on floppy disks to the lucky few who had heard of our project and had contacted us, a few hundred people in total. In their wisdom and against our wishes, the directors of the institute had insisted that the startup screen displayed the message “Copyright (c) Stichting Mathematisch Centrum, Amsterdam, 1985,” actively discouraging users from giving copies to their friends. This, by the way, was the same year in which Richard Stallman published the “GNU Manifesto” in *Dr. Dobb’s Journal*,\(^{22}\) which became a rallying call for the free software movement.

In 1988, van Rossum was assigned to the Amoeba project, a joint project by the
Free University and CWI, the aim of which was to develop a distributed operating system. One morning in January 1990, he invited me to a demonstration of his newest brainchild, a language named “Python,” which he had developed over the recess between Christmas and New Year. I do not recall being particularly enthralled at first; the point of this new language eluded me at the time. It was only after I noticed that the programmers involved in the project had become enthusiastic users—in particular Jack Jansen, after van Rossum the world’s first Python programmer\(^{23}\)—that I began paying closer attention.

**Python and the Secret of Success**

DESIGNERS SOMETIMES TRY to capitalize on the success of other products by imitating them. Although this formula may work for a limited time with fashion articles or films, it does not make any sense for programming languages. A successful language will give rise to a substantive collection of software written in that language and a workforce of programmers that are proficient in that language. Switching to another language means that the software base of an organization needs to be rewritten and programmers need retraining. This process engenders a very real cost, and will only be a profitable undertaking if the new language is decisively better in some important way. It cannot simply be a mere imitation.

It makes sense to think of the realm of programming languages as an ecosystem in which languages occupy their own niches. FORTRAN’s niche is high-performance scientific programming, involving heavy-duty numerical computation; that of COBOL is administration, based on files of data records. The C language is designed for systems programming, originally developed specifically for the Unix operating system. Just as there is no such thing as a general-purpose transportation vehicle, a truly one-size-fits-all general-purpose programming language does not exist; for a given highly specialized application domain it will always be possible to design a language tailored to, and better suited for, the specific needs of that domain.
As part of the Amoeba project, there was a need to connect independently developed pieces of software. At the time, operating systems such as Unix came with a program, called the “shell,” to issue direct instructions to the operating system. These included the kind of instructions needed to connect applications, such as feeding the output of one into the input channel of another. A sequence of shell commands, known as a “script,” could be stored in a file, whose name could then be used as a new shell command. This was a form of programming, but the shell languages, as they were at the time, had not been designed for programming, and using them for this purpose could be extremely awkward. Even worse, there was no direct way of interacting with Amoeba processes; stubs had to be written in C. Although switching to C may have seemed a better approach, it would also have forced the programmer into an awkward edit–store–compile–run–debug loop.

There was a clear need for a well-designed “scripting language” that allowed for interfacing with external processes. Python was originally designed to serve as a high-level scripting language for the Amoeba project. ABC was completely unsuitable for this purpose; it lived in a world of its own, shielding its users—by design—from the outside world. Python was expressly designed to interface with that outside world. The only other contender in the niche for high-level scripting languages at that time was an early version of Perl, but it was not properly documented and continued support for the language could not be guaranteed. Also, while Perl did what it was meant to do, it featured an eclectic mix of features adopted from other languages that had been integrated with little regard for consistency.

In spite of the radically different application domain, van Rossum had been inspired by the design philosophy and design rules of the ABC project. The design philosophy of Python would later be described somewhat poetically by Tim Peters, one of its early evangelists, in a text now known as “The Zen of Python.” Many of its aphorisms echo ABC’s design philosophy: “Errors should never pass silently” and “In the face of ambiguity, refuse the temptation to guess” correspond to a design rule we had dubbed the Logic-Error Rule. Similarly, the Fair-Expectation Rule is reflected in the assertion that “Special cases aren’t special enough to break the rules”; and “There should be one – and preferably only one – obvious way to do
it” embodies the spirit of the Economy-of-Tools Rule. In common with ABC, Python users could type in some code and immediately run it; the language had a concise and clear syntax, high-level data types, and variable declarations were not needed. This made the workflow so easy that its earliest users, system programmers, also began to use the language for many tasks outside its originally intended domain. A somewhat enhanced version, with proper documentation describing how to install the system under Unix, was posted as an open-source release on Usenet in early 1991, and picked up by programmers around the world.

The ability to evolve and adapt to changing needs is an essential attribute for the long-term survival of any programming language. FORTRAN, which first appeared in 1957, and COBOL, dating back to 1960, are both still very much alive today, but have undergone a considerable evolution. Likewise, Python, now available as version 3, is no longer the Python of 1990, but has still managed to hold on to its strong features, including its consistency, absence of unnecessary restrictions, and economy of tools. Powerful data types play a central role in Python, as they do in ABC, and in SETL beforehand.

The following code examples show two versions of the bubble sort algorithm, this time expressed in ABC and Python.

**ABC**

```plaintext
FOR i IN {1..#a-1}:
    FOR j IN {1..#a-1}:
        IF a[j] > a[j+1]:
            PUT a[j+1], a[j] IN a[j], a[j+1]
```

**Python**

```python
FOR i in range(1, #a):
    for j in range(1, #a):
        if a[j] > a[j+1]:
            a[j], a[j+1] = a[j+1], a[j]
```
for i in range(len(a)-1):
    for j in range(len(a)-1):
        if a[j] > a[j+1]:
            a[j], a[j+1] = a[j+1], a[j]

The syntactic similarity between the languages is clear. Another similarity is that there is no requirement in either language to include declarations of the variables used in a program—a burden lifted, but not at the expense of type safety. In almost all programming languages, code for the bubble sort algorithm will work for only one type of element, such as numbers. If strings of text, for example, also need to be sorted, another piece of code has to be written. In Python, as was also the case in ABC, the code above can be used for any element types for which the comparison operator > is defined, such as strings. Whereas many programming languages do not accommodate a ten-digit integer, neither ABC nor Python impose an upper bound on integer size. The value of the expression \(10^{999} + 1 - 10^{999}\) is exactly equal to 1, but step-by-step evaluation requires the subtraction of two numbers that, if written out in decimal notation, each use one thousand digits. In most languages, the program will bail out or produce 0 instead of 1. Both Python and ABC produce the correct value. This is not merely a property of the implementations; it is prescribed by the language specifications.

No brief overview of Python’s features,\(^{26}\) however appealing these may be, can do justice to the overall design of the language—it cannot be judged by considering its components in isolation. A better gauge of Python’s quality are the judicious choices that have been made about what features to include and what features not
to include. Its real strength can be measured in how well these components can be combined to suit the tasks of programmers.

The growth in popularity of Python, from its inception thirty years ago as a one-person effort flying under the radar, has been phenomenal, but not meteoric. Instead it has been a long, slow, and steady rise. Python’s ease of learning gave it a competitive advantage in a period when there was a perpetual need for more programmers. Its clean syntax and semantics make it easier to maintain a software base written in Python than other languages—an important consideration given that the cost of maintaining software dwarfs the cost of creating new software. Its increasing popularity has also been driven by the availability of extensive application libraries—collections of modules offering domain-specific functionality—developed and maintained by the Python community in the spirit of open and free software, and easily accessible through the searchable Python Package Index. Whether a start-up company quickly wants to develop a website or an AI application, Python libraries can provide a head start. There are modules for astronomy, cryptography, data science, financial services, genomics, multimedia, and many other areas. The design of Python makes using libraries and modules easy, an area in which the design of ABC was lacking. Python also makes the creation of new libraries easy, and thereby encourages users to contribute newly developed application libraries to the common repository.

By almost any measure, Python is now among the top three most popular programming languages worldwide—and often tops the list. Whatever the intrinsic qualities of Python, the phenomenal success of the language cannot be explained without taking into account the role played by its unassuming originator and lead designer, Guido van Rossum. Acting with the support and consensus of the Python community, van Rossum was required to sign off on all changes to the language, which led to him being bestowed the jocular title of “Benevolent Dictator for Life.” Throughout the evolution of Python, van Rossum’s steady guiding hand has helped safeguard the integrity of its design.
A Prequel and an Epilogue

I first met van Rossum in the early 1980s, when we both served on a team designing and developing a system for automating the membership administration of the Pacifistisch Socialistische Partij, a Dutch political party that later merged with several other small parties to form the GroenLinks (Green Left) party. The membership of the party had doubled in the preceding few years, and the three employees, who until then had been doing all the administrative work by hand, were overwhelmed. Luckily, the price of minicomputers had been dropping rapidly, and so the party had decided to buy one. The party’s budget did not allow for a commercial software house to develop the system they needed; instead, a small group of volunteers began developing the system in-house, using COBOL as the implementation language. A serious development bottleneck was the awkwardness of the editor that came with the minicomputer; it only allowed users to see one line of code at a time. Since the computer could not multitask, compilation—which was incredibly slow—and testing had to wait until an editing session was finished. As a result, progress could only be made at a snail’s pace. Van Rossum, a university student at the time, surprised the rest of the team by unexpectedly producing a new editor that was not only much easier to use, but also utilized most of the available screen space to good effect. He coded the editor in COBOL, a language he had only become acquainted with through this project. At the time, the ABC project was seeking to expand the development team, and, recognizing his talent for programming, I encouraged van Rossum to apply for the job. After graduating, he began his professional career on November 1, 1982 as a member of the B team, which he would also surprise by unexpectedly producing a new editor—a structure editor for ABC, fully cognizant of its syntax.

The Istanbul workshop in 2019 mentioned at the beginning of this essay was not the only one of its kind. It was, in fact, the eighth in an ongoing series of such workshops that were put on hold by the COVID-19 pandemic. In the course of these one-day workshops, the participants start with a trivial one-line program, such as `print('Hello, Django Girls!')`, and by the
end of the day have learned to program their own website. These Django Girls workshops are not specific to Istanbul; they are held all over the world. And Python is not only taught as a first language in these workshops; it has become the language of choice for introductory programming courses. Python now occupies the niche for which ABC was designed. Indeed, Python now occupies many niches. It has become a truly multi-niche language.

The success of any undertaking can rarely be ascribed to a single factor. In addition to those mentioned above, luck also plays a role. While ABC was perhaps too far ahead of its time, Python appeared at just the right time. Whatever the case, I believe that the design philosophy we developed in the process leading to ABC, inasmuch as it was also followed in Python, has had at least some positive influence.

DOI: 10.37282/991819.22.56


“TELCOMP,” Wikipedia.

“BASIC,” Wikipedia, see section “Unstructured BASIC.”

“Welcome to Python.org,” Python Software Foundation.


Centrum Wiskunde & Informatica.

The Computer Arts Society.
It is not clear when and by whom the term was coined, but it entered general colloquial use in the 1970s, not long after the publication of Edsger W. Dijkstra, “Notes on Structured Programming,” in *Structured Programming*, ed. Ole-Johan Dahl et al. (London: Academic Press, 1972), 1–82. 


“INTERCAL,” *Wikipedia*. 


A precursor can be found in Peter Landin’s ISWIM, a suite of gedankenexperiment languages that used optional indentation, not for readability, but mirroring its use in mathematics. See P. J. Landin, “A Correspondence Between ALGOL60 and Church’s Lambda-Notation: Part I,” *Communications of the ACM* 8, no. 2 (1965): 89–101, doi:10.1145/363744.363749. In 1974, Donald Knuth wrote:

It seems clear that languages somewhat different from those in existence today would enhance the preparation of structured programs. We will perhaps eventually be writing only small modules which are identified by name as they are used to build larger ones, so that devices like indentation, rather than delimiters, might become feasible for expressing local structure in the source language. 


Jack Jansen is still active in the Python community, in particular as the maintainer of MacPython, a port of Python to macOS, the operating system used by Apple's Mac desktop and laptop computers.

“Why was Python Created in the First Place?,” *General Python FAQ*, Python Software Foundation.


Readable introductions to Python that offer such an overview are easily found on the Internet. A list can be found on the Python wiki: “BeginnersGuide/Programmers,” *Python.org*. An easy introductory course not assuming any prior programming experience is given by: April Speight, *Bite-Size Python: An Introduction to Python Programming* (Indianapolis: John Wiley & Sons, 2020).

“Python Package Index,” *Python Software Foundation*.


“Python Creator Scripts Inside Google,” *eWEEK*, March 6, 2006. Since 2019, this role has been taken over by a Steering Council.


*Lambert Meertens* is a researcher at the Kestrel Institute in Palo Alto, California. While his main work was done at CWI, Amsterdam, and at the Kestrel Institute, he has held professorships at New York University, Delft University of Technology and Utrecht University, and was chair of IFIP Working Group 2.1 on Algorithmic Languages and Calculi from 1999 to 2009.

More on Computer Science
Wacky Jabber
Douglas Hofstadter
A sweetish suite of machine translations of a pseudo-Swedish paragraph concocted by Douglas Hofstadter. The resulting gobbledygook reveals the zombie-like nature of these highly vaunted programs.

An Antique Land
Benjamin Thierry
Benjamin Thierry reviews *Minitel: Welcome to the Internet* by Julien Maillard and Kevin Driscoll. The French Minitel project, according to the authors, was much, much more than simply a public works project gone awry.

When Existence is Inefficient
Allyn Jackson
How hard is it to solve a problem when a solution is guaranteed to exist? It is to address questions like this that the theoretical computer scientist Christos Papadimitriou invented the complexity class PPAD.