ABC *vs.* Python

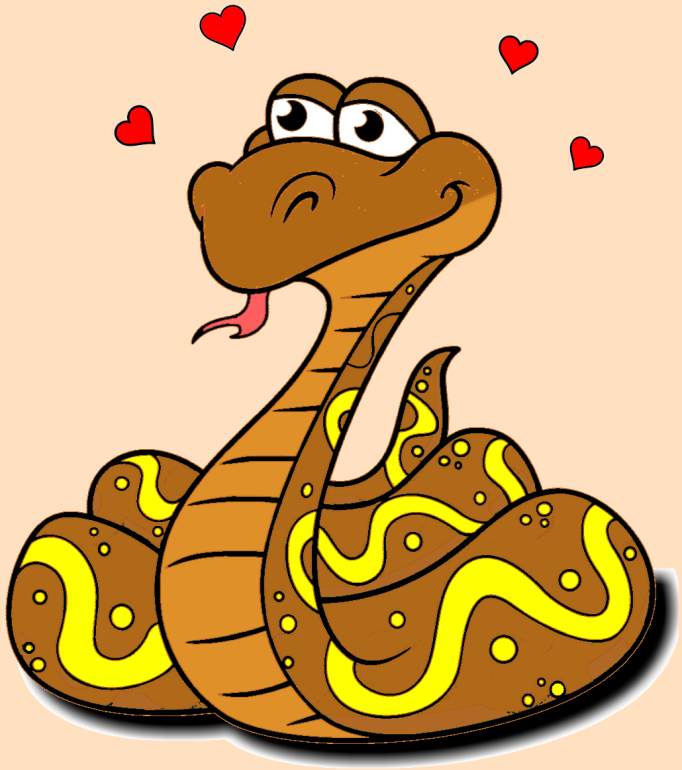
Lambert Meertens

CWI Lectures on Programming

21 November 2019


ABC


AEC



ABC is the precursor of Python

> While sharing software with the world today only takes a few clicks, in the 1980s it was an altogether more laborious affair, with van Rossum recalling the difficulties of trying to distribute Python precursor ABC.

Nick Heath, “Python is eating the world”.
ZDnet, August 6, 2019

Which Python features come from ABC?

- The infamous colon
- The equally infamous indentation
- The prompt (>>>)
- and …?

Thesis:

All good things in Python come from ABC.

Corollary:

All bad things in Python come from Guido.

Differences and commonalities

The divergence between ABC and Python stems mainly from *differences in their design objectives*.

I’ll come to these differences. But let me focus first on the *commonalities*.

Commonalities

The commonality between ABC and Python stems mainly from the commonality in their *design philosophy* and *design rules*.

(Aside:

Language design is not an exact science.

That is why I prefer the term "*design philosophy*" over "*design methodology*".)

Design philosophy

A designer needs to make hundreds of design decisions, choosing from an ocean of possibilities. How to choose judiciously from this plethora?

- Awareness of the *design space* and willingness to *explore* it.
- Awareness of *trade-offs* and the need to strike a *balance*.
- Recognizing the value of *consistency* (but not a *foolish* consistency...).

Design rules

The design rules governing the design of ABC crystallized out during the design process.

To understand and appreciate these design rules, it is essential to understand the *design objectives*.

To explain the design objectives, I need to go back to the origins of the ABC project.

Origins of the ABC project

Dramatis personae:

Leo Geurts, programmer

Lambert Meertens, junior researcher

Location:

Mathematical Centre

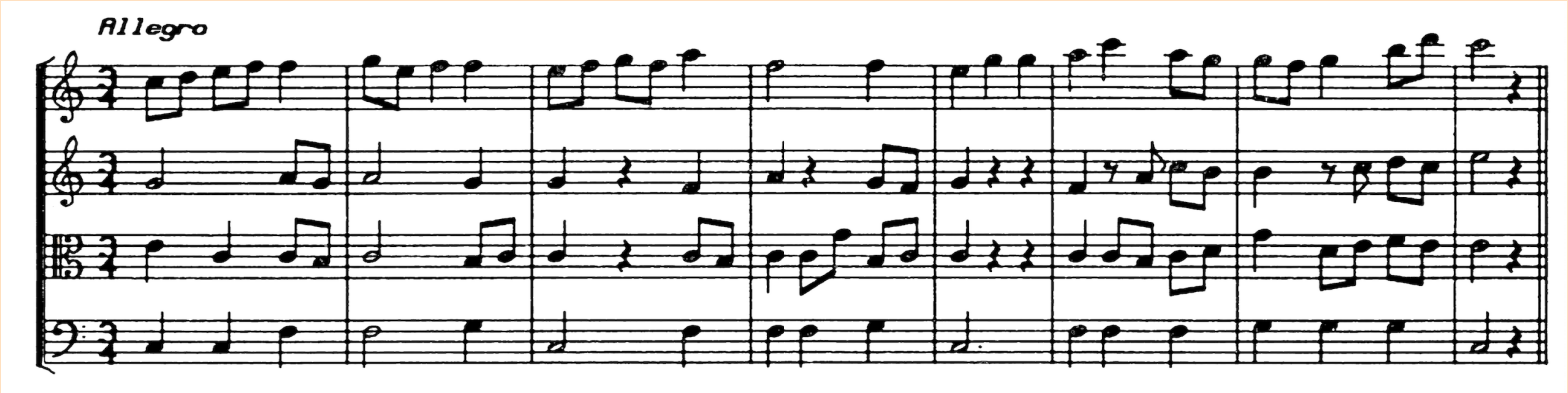
Period:

1968–1975

Computer art exploration

1968: Lambert Meertens

String quartet No. 1 in C major



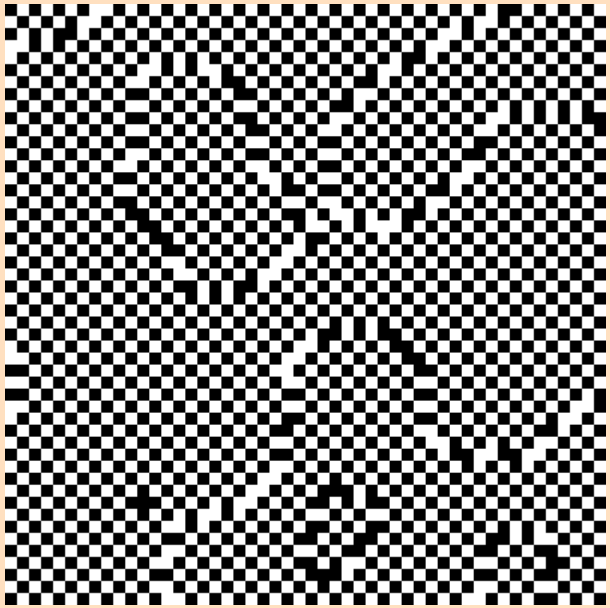
1970: Louis Andriessen, Leo Geurts,
Lambert Meertens

Sonata Opus 2 No. 1

Computer art exploration (continued)

1970: Leo Geurts, Lambert Meertens

Kristalstructuren



Computer Arts Society

1968: Founded in London

1970: Dutch branch (CASH)

Activities:

- newsletter (*PAGE*)
- meetings
- conferences
- exhibitions
- **practical courses**

Practical courses

- Most participants had never used a computer.
- We taught TELCOMP II, very similar to BASIC.
- The participants used teletypes, which were hooked up to a time-sharing service.
- The number of available teletypes was rather limited, so participants had to take turns.

Report on a 1970 course

THE CAS WEEKEND COURSE

Some 10 people spent the week-end of 20/21 June in the CAS course and workshop on "non-numerical programming with special emphasis on text processing and music", at Time Sharing Limited, 187 Great Portland Street, London W1. Some of them had experience in programming, some had not. After an explanation of the terminal system and the Telcomp2 language on Saturday morning, 6 terminals were used to produce a variety of music and especially text.

Output of a Telcomp2 program by one of those without previous programming experience:

HAPPY FAMILIES

MOTHER SITS ON MOTHER
MOTHER LEAP-FROGS OVER THE LODGER
LITTLE JOHNY LOVES SISTER
MOTHER LOVES GRANNY
THE DOG SLEEPS WITH GRANNY
THE LODGER RUNS TO BROTHER
THE DOG SITS ON THE BABY
THE BABY JUMPS ON GRANNY
UNCLE TED PLAYS WITH SISTER
THE LODGER JUMPS ON UNCLE BOB
FATHER SLEEPS WITH THE BABY
MOTHER JUMPS ON THE BABY
THE LODGER JUMP-FROGS OVER FATHER
UNCLE TED LEAP-FROGS OVER MOTHER

Another course and workshop will be announced in PAGE by October.

BASIC in the 60s and 70s

```
100 REM BUBBLE S0RT
110 F0R I = 1 TO N - 1
120 F0R J = 1 TO N - 1
130 IF A[J] <= A[J+1] THEN 170
140 LET X = A[J]
150 LET A[J] = A[J+1]
160 LET A[J+1] = X
170 NEXT J
180 NEXT I
```

We could teach the language in half a day.

However, the language was unstructured; programs became spaghetti code.

Most participants struggled with even the simplest programs.

The ABC design objectives

This explains why, back in 1975, Leo and I started the ABC project (then named *B* project).

Aim: to design a new “beginners’ programming language” as a replacement of BASIC. This new language was to be:

- simple to learn and to use;
- suitable for conversational use (interactive);
- suitable for structured programming.

What is a “beginner”?

- Basic assumption: someone who does not know any programming language and is not familiar with even basic programming concepts
- To serve such users:
  - hide low-level implementation details;
  - instead, provide powerful high-level (task-oriented) features;
  - make the implementation interactive.

Simplicity clarified

- Users need to learn only *a small number of constructions.*
- The concrete syntax of these constructions is suggestive of their meanings, making them *easy to read and easy to remember.*
- The *semantics* of each construction is as *straightforward* as can be.
- Learning more *complicated concepts can be postponed* until the simpler ones are understood.

Suitability for conversational use clarified

- The system signals errors as soon as possible, rather than wait for the program to be completed.
- It displays one face to the user, not a suite of subsystems (editor, file system, compiler), each having their own conventions and reactions.
- It does not leave the user uncertain whose turn it is but prompts promptly whenever user input is required.

On the importance of interactiveness

A quote from the first Python paper (1991):

One of Python's strengths is the ability for the user to type in some code and immediately run it: no compilation or linking is necessary. Interactive performance is further enhanced by Python's concise, clear syntax, its very-high-level data types, and its lack of declarations (which is compensated by run-time type checking). All this makes programming in Python feel like a leisure trip compared to the hard work involved in writing and debugging even a smallish C program.

Interactively testing remote servers using the Python programming language
Guido van Rossum en Jelke de Boer

Suitability for structured programming clarified

- The language allows you to write programs in such a way that it is easy to understand the *functionality* of the program by looking at the program *text*. In particular:
  - the various constructions faithfully embody intuitive task-oriented abstractions;
  - the language makes it easy to subdivide a task into subtasks;
  - the language allows the user to define problem-oriented abstractions.

Design by iteration

$$B_0 \rightarrow B_1 \rightarrow B_2 \rightarrow \cdots \rightarrow B_\infty.$$

1975: B_0
1978: B_1
1979: B_2
1985: B_∞ = ABC

1982: Two new faces on the team

- Steven Pemberton (1 September, attracted as a guest researcher for one year)
- Guido van Rossum (1 December)

The design of the language at the time (B_2) was already fairly advanced; apart from the inbuilt data types, it was *very* close to ABC, but we still did a lot of polishing to give it a high gloss.

Each proposed language change was extensively discussed by the whole team.

The ABC design rules

The design rules governing the design of ABC crystallized out during the design process.

These rules are “rules of thumb” rather than a set of principles that can be followed blindly. There are trade-offs, and finding good solutions often requires creativity.

The aim of the rules is to keep the language *both* simple to learn and simple to use.

The list of design rules

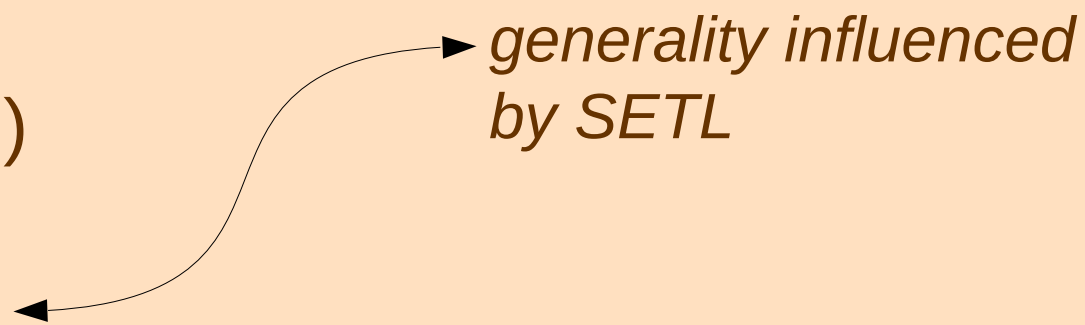
- Economy-of-Tools Rule
- Fair-Expectation Rule
- Uniformity Rule
- Logic-Error Rule
- One-Concept-at-a-Time Rule
- Semantic-Distance Rule

Economy-of-Tools Rule

The number of concepts (functions, features etc.) is small, but the concepts themselves are powerful and on the appropriate, task-oriented, abstraction level.

For example: ABC has only five types:

- number
- text (string)
- compound (tuple)
- list
- table (dictionary)



Fair-Expectation Rule

If a concept may be lawfully used in context *X*, and the same concept is (conceptually) applicable in context *Y*, then it may be lawfully used in context *Y*, with the expected meaning.

Example of violation in ALGOL 60:

procedure *P*(*i*); **integer** *i*; …
P(123);

integer *i*;
i := 123;

procedure *Q*(*s*); **string** *s*; …
Q(‘hello’);

string *s*;
s := ‘hello’;

Uniformity Rule

Similar concepts are embodied (invoked) in a similar way.

`similar things should be said in similar ways`

Signal and noise in programming language.

P. J. Plauger

Logic-Error Rule

The embodiments of the concepts are such that errors cannot arise if this can be prevented by choosing the right embodiment. Insofar this is impossible, making errors is made hard. Errors are signalled, if possible, on concept invocation before anything else happens. If this is impossible, they are signalled before disaster strikes, and the action leading to disaster is aborted.

Therefore:

- No dangling ELSE problem and suchlike
- Static checks where possible
- Dynamic checks otherwise

```
Errors should never pass silently.
```

PEP 20 – The Zen of Python Tim Peters

One-Concept-at-a-Time Rule

Concepts do not mutually depend on each other to fully understand them.

(Only important for learning, not for use.)

Semantic-Distance Rule

No two concepts of the system are almost the same.

`different things should be said differently.`

Signal and noise in programming language.

P. J. Plauger

Corollary:

```
There should be one-- and preferably only
   one --obvious way to do it.
Although that way may not be obvious at
   first unless you're Dutch.
```

PEP 20 – The Zen of Python Tim Peters

Comparison of design objectives

	ABC	Python
target users	beginners	Guido*
target use	learning to program	scripting

---

* and people like Guido

(And Python started life as a one-person skunkworks project.)

The essential Guido nature

Guido may be an ex-BDFL, but he was, is and remains a BHFL.

Unfortunately, no one understands the original use of the term *hacker* anymore.

```
hacker
       A person who delights in having
       an intimate understanding of the
       internal workings of a system,
       computers and computer networks
       in particular.

    RFC 1392, Internet Users' Glossary
```

Another similarity: design by iteration

$$\mathrm{Py}_0 \rightarrow \mathrm{Py}_1 \rightarrow \mathrm{Py}_2 \rightarrow \mathrm{Py}_3 \rightarrow ???$$

Difference:

For Python, the process is driven by the user community.

Extra: The origins of indentation

- ISWIM (Landin 1966, an expression-oriented “paperware” language) introduced indentation for grouping, similar to the conventions of “mathematical communication”, which “require no explanation”.
- SASL (Turner 1972, a variant of the applicative subset of ISWIM) implemented this.

The origins of indentation (continued 1)

Perhaps we should go even further. Well-written programs are always indented to show who controls what. A very hard bug to find, in fact, is one where the indenting shows the intent but not the actuality:

```
if (a)
    if (b)
        ...
else ...
```

The else binds with the innermost if, even though that is not wanted here. Why not have the compiler read the same signal as we human beings, and let the indenting control grouping? It warrants consideration.

Signal and noise in programming language (1975)

P. J. Plauger

The origins of indentation (continued 2)

Plauger's argument is essentially one about the improved *readability* of the program text: the layout of the text faithfully represents its meaning.

Readability, rather than the convention in mathematical communication, was also the motivation for adopting this in ABC.

After all, as we all know:

```
Readability counts,
```

PEP 20 – The Zen of Python Tim Peters

The origins of indentation (continued 3)

The syntax of B0 is such that a new line where forbidden or no new line (but a space) where obligatory, never changes a valid program into another valid program. As a consequence, a B0 editor that is aware of the syntax and automatically indents at each new line, may also automatically increase the indentation level at each new line which is not obligatory, thus indicating continuation of the running statement. Similarly, at the end of each statement the editor can restore the old indentation level. As a result, **B0 programs always have a reasonable layout.**

Designing a beginners' programming language (1976)

Leo Geurts and Lambert Meertens

"On the Internet, nobody knows you're a benevolent dog."