From ABC to Python
A Language Design Case Study

Guido van Rossum
guido@python.org

9th LASER summer school, Sept. 2012
A History Lesson

- Why? Because I can :-)
- No language is designed "from scratch"
- "Those who don't know history are destined to repeat it"
  --Edmund Burke (1729–1797)
About ABC (née B)

- Lambert Meertens and others at MC (now CWI)
- Late 1970s, early 1980s; I joined in 1982

Origins: Meertens used to teach classes in Algol–60 for the X8 computer to scientists

Observations:
- too much time wasted on "system" details
- no time left for writing the desired program
  - (e.g. data analysis for some experiment)
At the time, microcomputers were coming out with integrated Basic (e.g. Apple ][])
  ◦ extremely limited in memory
  ◦ notorious for PEEK and POKE operations
Meertens c.s. set out to design an alternative
  ◦ based on their experience with Algol
  ◦ both adopting good ideas and fixing problems
Originally meant for microcomputers
  ◦ by the time I joined, targeting UNIX systems
  ◦ initially named $B$
"With the choice between a language where it will take a week to write a program, and a language where it will take an afternoon, most people will choose the latter."

"ABC is even easier to learn and use than BASIC, and a more powerful tool than Pascal."

--The ABC Programmer's Handbook (Geurts, Meertens, Pemberton, 1990)
HOW TO PRINT CELSIUS FROM a TO b:
PUT a, b IN lo, hi
IF lo > hi:
    PUT hi, lo IN lo, hi \Swap hi and lo
FOR f IN \{lo..hi\}:
    PUT (f-32)*5/9 IN c
    WRITE f, "Fahrenheit =", 2 round c, "Celsius" /

PRINT CELSIUS FROM 40 TO 45

40 Fahrenheit = 4.44 Celsius
41 Fahrenheit = 5.00 Celsius
42 Fahrenheit = 5.56 Celsius
43 Fahrenheit = 6.11 Celsius
44 Fahrenheit = 6.67 Celsius
45 Fahrenheit = 7.22 Celsius
ABC Function Example

- HOW TO RETURN elements collection:
  PUT {} IN unique
  FOR element IN collection:
    IF element not.in unique:
      INSERT element IN unique
    RETURN unique
- WRITE elements "Mississippi"
- {"M"; "i"; "p"; "s"}
What Went Wrong?

- Design choices (UPPER CASE keywords)
- Implementation (optimized for huge strings)
- System Interface (no way to open a file)
- User Interface (integrated editor)
- Tools (there weren't any)
- Cooperation (hard to share programs)
- Evolution (hard to extend)
- Community (bootstrap problem)

- Perfectionistic and patronizing attitudes
ABC vs. Python

- Python started as "skunkworks" project
- Cut corners to get results quickly
- Aimed to please UNIX users
- Changes to ABC's data types
- Extension modules
- OS interface
- "Release early, release often" policy
- Listen to users; they are never wrong
- Plan to fix corner-cutting eventually
Some Examples of Differences

- **Numbers:**
  - ABC has arbitrary precision rationals and floats
  - Python started out with 32bit ints and floats

- **Strings:**
  - ABC represents strings as tree structures
  - Python represents them as a 1-malloc object

- **Lists and Dicts:**
  - ABC keeps them sorted
  - Python's lists are unsorted, dicts are hash tables

- **Tuples:**
  - In Python, also indexable, to use as immutable lists
A Big Difference

- ABC is statically typed
  - type inferencing happens during compilation
    - some form of Hindley–Milner
  - still, the runtime tags all values with their type
    - needed because of interactive program modification

- Python is dynamically typed
  - the compiler doesn't care about types
    - though it does care about scopes
  - uses object types instead of type tags on values
  - for good measure, *everything* is an object
    - e.g. types, functions, modules
Values are *computed* with infinite precision
- e.g. \( \frac{123456789}{234567890} \)
- Arjen Lenstra and other mathematicians *loved* this!

But *printed* as decimal approximations
- e.g. 0.5263158098919677; or 0.52
- (to get the fraction, use num/denom operators)

Easily create *huge* numerators/denominators
- Computation slows down to a crawl
  - long division is slow!
- Hard to debug unless familiar with the problem
- Common pitfall for non-mathematical users
Python's Evolving Integers

- **Stone age: simplest implementation**
  - 32-bit int; no overflow checking
  - no automatic coercions between int and float!

- **Bronze age: more user friendly**
  - exception on overflow; automatic float coercion

- **Iron age: the "long" type: more functionality**
  - arbitrary precision; manual conversion needed

- **Industrial age: even more user friendly**
  - automatic extend int to long on overflow

- **Python 3: single int type (== old long)**
Other Numeric Types

- Float (real; mapped to C "double")
- Complex (a favor to numpy)
- Decimal (IEEE 754; decimal *floating point*)
- Fraction (rational; two ints, gcd-normalized)

Being a number is a "protocol"
  - define __add__, __sub__, __mul__, __div__, etc.
  - define __float__ to convert to float
  - define __int__ to convert to integer (may truncate)
  - define __index__ to convert to integer (if exact)
  - define __eq__, __hash__ to be comparable w. others
Trouble With ABC's Strings

- ABC's string data type internally used a tree
  - This is great when slicing and joining huge strings
  - We used asymptotically optimal algorithms!
    - We even had a mathematical proof they were optimal!!
    - Sadly the constant factor was rather large...

- Python's strings optimize for shorter strings
  - One allocation per string object
  - C–speed O(N) operations
  - This is good for many practical situations
  - E.g. words in a huge file (millions of short strings)
ABC lists were "bags" or "multisets"
  ◦ Again, a theoretically optimal tree structure
  ◦ Always kept sorted
  ◦ But that's not what you need most often!

To represent e.g. "the lines of a file"
  ◦ E.g. if you’re building a text editor
  ◦ ABC must use a table mapping linenumber → text
  ◦ This is awkward when inserting or moving lines

ABC tables were similarly sorted by key
  ◦ That's nice, but hash tables are faster!
ABC slices only apply to strings

Notation: \textit{string}@lower, \textit{string}|upper
- 'abc'@2 = 'bc'; 'abc'|2 = 'ab'; 'abc'@2|1 = 'b'
- i.e. 1–based indexing, closed interval slicing
- Invariant: \textit{s}|x \land \textit{s}@|(x+1) = \textit{s}

Python uses 0–based, half–open intervals
- 'abc'[1:] == 'bc'; 'abc'[:2] == 'ab'; 'abc'[1:2] == 'b'
- Nicer invariant: \textit{s}|[x] + \textit{s}[[x:]] == \textit{s}
- Select individual characters: 'abc'[1] == 'b'
- Negative counts from the end: 'abc'[-1:] == 'c'
- You can slice \textit{any sequence}
The Awkward UPPER CASE

- Maybe a leftover trait from Algol
  - which printed keywords in **bold or underlined**
- Also distinguishes functions from procedures
  - like Algol–68 used bold for types
- But C, and even Pascal, didn't need this
- In UNIX, upper case was frowned upon
  - more effort to type (hold shift key)
  - reminded of mainframe punched cards
- To modern eyes it looks like SHOUTING
- Also didn't anticipate code colorization
Community, Internet

- ABC distributions were by magnetic tape
  - Information mostly spread by word of mouth
- Python was lucky to have the Internet
- In 1991, email and usenet abounded
  - First source distribution on comp.sources.unix (?)
- We quickly set up a mailing list (1992)
  - python-list@cwi.nl; now @python.org
- Later we added a newsgroup (1994)
  - comp.lang.python (still very much alive)
  - comp.lang.python.announce (1998, moderated)
But The Biggest Difference Is...

- Openness and extensibility
  - Python lets you look under the hood
  - Introspection APIs abound
  - Extension modules are easy to write

- "Every complex system should have at least two levels of extensibility"
  - In Python: regular modules; extension modules
  - Handy response to unwanted feature requests:
    - "You can do that yourself, just write an extension"