ABC

A mãe do Python

Luciano @Ramalho.org

Uma expedição arqueológica

Estes slides e mais recursos no github

Uma expedição arqueológica

Estes slides e mais recursos no github

Luciano Ramalho, autor de Python Fluente

- Texto integral oficial em PythonFluente.com

Características da linguagem*

- 5 tipos de dados que podem ser combinados
- tipagem forte, sem declarações de tipos
- inteiros e strings de tamanho limitado só pela memória (raridade na época)
- *“refinements for top-down programming”*
- blocos por indentação
- concisão (¼ do equivalente em Pascal ou C)

* destacadas por Steven Pemberton

Características do ambiente*

- Não há necessidade de arquivos: procedimentos, funções e variáveis globais permanecem após o encerramento da sessão.
- A mesma interface é exibida ao usuário em todos os momentos, seja ao executar comandos, editar ou inserir dados em um programa.
- Undo!

* destacadas por Steven Pemberton

Luciano Ramalho

38. ABC: the mother of Python

Google
Microsoft
SurveyMonkey
nasuni
facebook
Dropbox
GONDOR
Eventbrite
nebula
New Relic
heroku


38
Basic Types
Collection Types


38. ABC: the mother of Python
Collection Types
List
Table
Exa
Jargon
1:02 / 7:09


38
Basic Types
Collection Types


38
Basic Types
Collection Types

ABC language
the mother of Python

"It all started with ABC, a wonderful teaching language that I had helped create in the early eighties. It was an incredibly elegant and powerful language, aimed at non-professional programmers."

Guido van Rossum, ABC team member and creator of Python[1]

ABC 1.05

```
>>> HOW TO SIEVE TO n:
HOW TO SIEVE TO n:
   PUT {2..n} IN numbers
   WHILE numbers <> {}:
      PUT min numbers IN p
      WRITE p
      FOR m IN {1..floor(n/p)}:
         IF m*p in numbers:
            REMOVE m*p FROM numbers

>>> SIEVE TO 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

SIEVE TO procedure definition

condition must be a **test**

21 years later, Python still has a lot of ABC in it

Python 3.2

```
>>> def sieve(n):
...     numbers = set(range(2, n+1))
...     while numbers:
...         p = min(numbers)
...         print(p, end=' ')
...         for m in range(1, int(n/p)+1):
...             if m*p in numbers:
...                 numbers.remove(m*p)
...
>>> sieve(50)
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

1. Foreword for "Programming Python" (1st ed.) http://www.python.org/doc/essays/foreword/

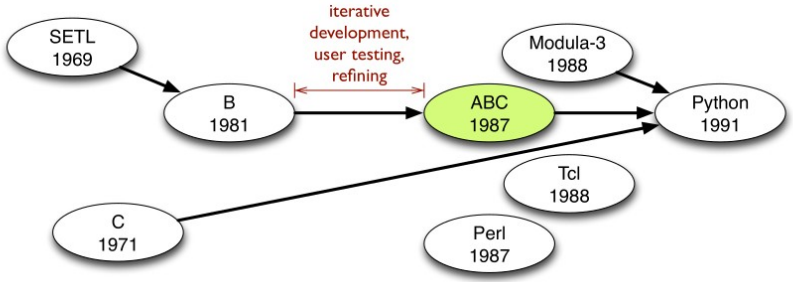
History

- ABC is the fourth iteration of work started in 1975 by Lambert Meertens and Leo Geurts at the CWI, then Stichting Mathematisch Centrum, in Amsterdam.

"[ABC] began as an attempt to design a suitable alternative to Basic for beginner programmers – a language that was still easy to learn, still interactive, but was easier to use and offered program structure."
Steven Pemberton (CWI)

"[B is] easy to use because it has powerful constructs without the restrictions professional programmers are trained to put up with but a newcomer finds irritating, unreasonable, or silly."
Steven Pemberton (CWI)

- The third iteration, called B, was developed in 1979-1981 with the collaboration of Robert Dewar of NYU, who brought ideas (such as mappings) from the SETL language.



- After 5 years of experience using and teaching B, the first and final version, called ABC, was released in 1987. Steven Pemberton and L. Meertens led the team during this time. Guido helped with design and implementation from 1982 to 1986.

"[ABC] was designed by first doing a task analysis of the programming task and then doing several iterations that included serious user testing. My own role in the ABC group was mainly that of implementing the language and its integrated editing environment.."
Guido van Rossum

- ABC 1.05 is copyrighted 1991. Funding was withdrawn around that time. As of Feb. 2012, the newest binary package has files dated Feb. 7, 2005.

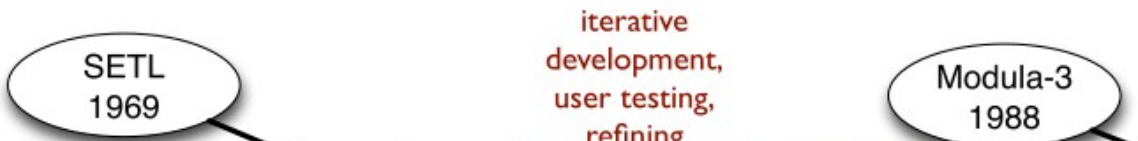
History

- ABC is the fourth iteration of work started in 1975 by Lambert Meertens and Leo Geurts at the CWI, then Stichting Mathematisch Centrum, in Amsterdam.

"[ABC] began as an attempt to design a suitable alternative to Basic for beginner programmers – a language that was still easy to learn, still interactive, but was easier to use and offered program structure."
Steven Pemberton (CWI)

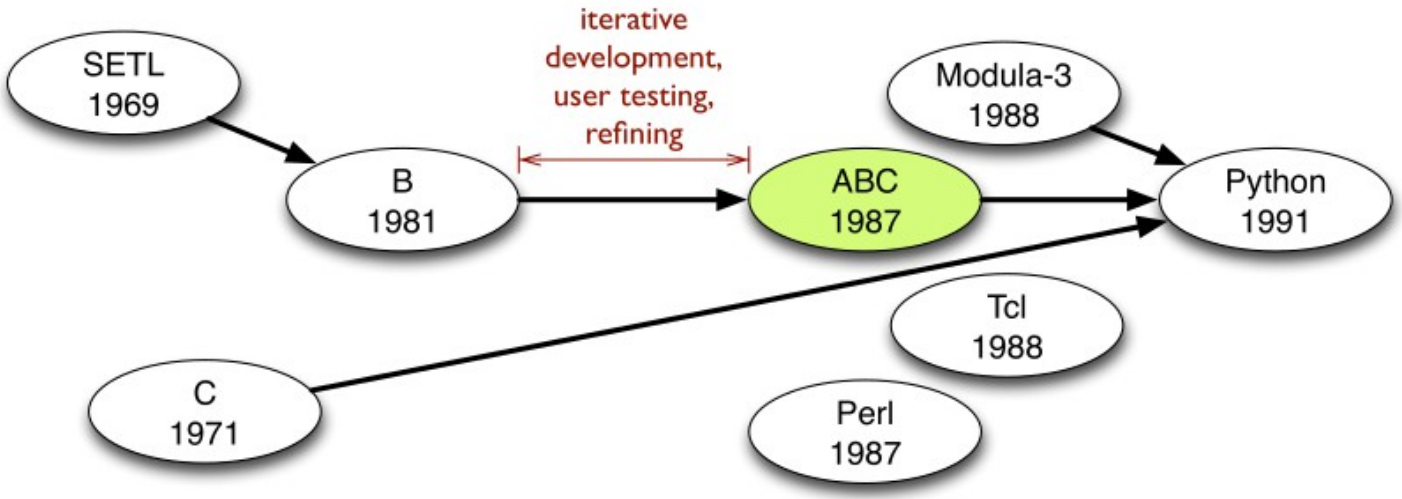
"[B is] easy to use because it has powerful constructs without the restrictions professional programmers are trained to put up with but a newcomer finds irritating, unreasonable, or silly."
Steven Pemberton (CWI)

- The third iteration, called B, was developed in 1979-1981 with the collaboration of Robert Dewar of NYU, who brought ideas (such as mappings) from the SETL language.



up with but a newcomer finds irritating, unreasonable, or silly."
Steven Pemberton (CWI)

NYU, who brought ideas (such as mappings) from the SETL language.



- After 5 years of experience using and teaching B, the first and final version, called ABC, was released in 1987. Steven Pemberton and L. Meertens led the team during this time. Guido helped with design and implementation from 1982 to 1986.

"[ABC] was designed by first doing a task analysis of the programming task and then doing several iterations that included serious user testing. My own role in the ABC group was mainly that of implementing the language and its integrated editing environment.."
Guido van Rossum

- ABC 1.05 is copyrighted 1991. Funding was withdrawn around that time. As of Feb. 2012, the newest binary package has files dated Feb. 7, 2005.

Basic Types

Numbers

- Numbers are either **exact** or **inexact**.
- Exact numbers are stored as ratios of integers of arbitrary length.
- Basic arithmetic (+, -, *, /) with exact numbers always produce exact results.
- Some functions, such as root, sin, log etc. give inexact results.

ABC 1.05

```
>>> PUT 0.70 IN charge
>>> PUT 0.05 IN tax
>>> PUT charge * (1+tax) IN total
>>> WRITE total
0.7350
>>> WRITE 2 round total
0.74
>>> WRITE */total
147
>>> WRITE /*total
200
```

numerator / denominator

rounding error

Python 3.2

```
>>> charge = 0.70
>>> tax = 0.05
>>> total = charge * (1+tax)
>>> print(total)
0.735
>>> print(round(total, 2))
0.73
>>> print(format(total,'.18f'))
0.734999999999999987
```

Text

- Mutable ASCII strings of arbitrary length.
- First character position is @ 1.
- Slicing is done with operators @ and | (pipe), see examples below.

similar to list slice assigment in Python

ABC 1.05

```
>>> PUT 'mutable' IN text
>>> PUT 'nt' IN text@5
>>> WRITE text
mutant
>>> PUT 'ili' IN text@2|1
>>> WRITE text
militant
```

ABC 1.05

```
>>> WRITE 'elephant'@3
ephant
>>> WRITE 'elephant'|3
ele
>>> WRITE 'elephant'@2|3
lep
>>> WRITE 'elephant'|3@2
le
```

Collection Types

now = compound with date and time

ABC 1.05

```
>>> PUT now IN start
>>> WRITE start
(2012, 3, 5, 6, 19, 46.359)
>>> PUT start IN y, m, d, hour,
min, sec
>>> WRITE hour, ":", min, ":",
round sec
6 : 19 : 46
```

List

- Sequence of ordered values of same type.
- Individual items may be retrieved and removed.

ABC 1.05

```
>>> PUT {7; 2; 3; 1; 4} IN lotto
>>> FOR number IN lotto: WRITE number
1 2 3 4 7
>>> INSERT 'coin' IN lotto
*** Can't cope with problem in your command
    INSERT 'coin' IN lotto
*** The problem is: incompatible types "" and 0
>>> WRITE #lotto \ length of list
5
>>> WRITE lotto item 5 \ 5th item
7
```

type check

\ comments

Compound

- Collection of values of same or different types, assigned to a single address.
- A compound may unpacked into several addresses.
- There is no other way to get at the individual parts.

ABC 1.05

```
>>> PUT {} IN tel
>>> PUT 11 IN tel['Jack']
>>> PUT 12 IN tel['Abe']
>>> PUT 13 IN tel['Bob']
>>> WRITE tel
{["Abe"]: 12; ["Bob"]: 13; ["Jack"]: 11}
>>> FOR name IN keys tel:
        WRITE name << 7, tel[name] /

Abe     12
Bob     13
Jack    11
>>> PUT 'We are all mad.' IN quote
>>> WRITE split quote
{[1]: "We"; [2]: "are"; [3]: "all"; [4]: "mad."}
```

sorted keys

Table

- Mapping of sorted, unique keys and values.
- Every key must have the same type; values also. But keys and values need not be of the same type.
- Iteration obtains the values. Use **keys** to get a list of keys.

Commands

Input/Output

`WRITE «expr»`
Write to screen; / before or after «expr» gives new line

`READ «address» EG «expr»`
Read value from terminal to «address»; «expr» is example of type to be accepted

`READ «address» RAW`
Read line of text

Data Handling

`PUT «expr» IN «address»`
Put value of «expr» in «address»

`REMOVE «expr» FROM «list»`
Remove one element from «list»

`INSERT «expr» IN «list»`
Insert in right place, keeping the «list» sorted

`DELETE «address»`
Delete permanent location or table entry

`SET RANDOM «expr»`
Start random sequence for **random** and **choice**

Termination

`QUIT`
Terminate command or leave the ABC environment

`RETURN «expr»`
Leave function returning value of «expr»

`REPORT «test»`
Leave predicate reporting outcome of «test»

`SUCCEED | FAIL`
Leave predicate reporting success or failure.

Flow Control

`CHECK «test»`
Check «test» and stop if it fails (like assert)

```
IF «test»:
    «commands»
```
If «test» succeeds, execute «commands»; no ELSE allowed

```
SELECT:
    «test»: commands
    ...
    «test»: commands
```
Select one alternative: try each «test» in order (one must succeed; the last test may be ELSE)

```
WHILE «test»:
    «commands»
```
As long as «test» succeeds execute «commands»

```
FOR «name», ... IN «train»:
   «commands»
```
Take each element of «train» in turn and execute «commands»; may unpack compound elements

`PASS`
Do nothing

`«KEYWORD» «expr» «KEYWORD» ...`
Execute user-defined command

`«KEYWORD»`
Execute refinement command

"We did requirements and task analysis, iterative design, and user testing. You'd almost think programming languages were an interface between people and computers."
Steven Pemberton (CWI)

Commands

Input/Output

`WRITE «expr»`
Write to screen; / before or after «expr» gives new line

`READ «address» EG «expr»`
Read value from terminal to «address»; «expr» is example of type to be accepted

`READ «address» RAW`
Read line of text

Data Handling

`PUT «expr» IN «address»`
Put value of «expr» in «address»

`REMOVE «expr» FROM «list»`
Remove one element from «list»

Flow Control

`CHECK «test»`
Check «test» and stop if it fails (like assert)

```
IF «test»:
    «commands»
```
If «test» succeeds, execute «commands»; no ELSE allowed

```
SELECT:
    «test»: commands
    ...
    «test»: commands
```
Select one alternative: try each «test» in order (one must succeed; the last test may be ELSE)

```
WHILE «test»:
    «commands»
```
As long as «test» succeeds execute «commands»

`FOR «name», ... IN «train»:`

PUT «expr» IN «address»
Put value of «expr» in «address»

REMOVE «expr» FROM «list»
Remove one element from «list»

INSERT «expr» IN «list»
Insert in right place, keeping the «list» sorted

DELETE «address»
Delete permanent location or table entry

SET RANDOM «expr»
Start random sequence for **random** and **choice**

Termination

QUIT
Terminate command or leave the ABC environment

RETURN «expr»
Leave function returning value of «expr»

REPORT «test»
Leave predicate reporting outcome of «test»

SUCCEED | FAIL
Leave predicate reporting success or failure.

Select one alternative: try each «test» in order (one must succeed; the last test may be ELSE)

WHILE «test»:
**   «commands»**
As long as «test» succeeds execute «commands»

FOR «name», ... IN «train»:
**   «commands»**
Take each element of «train» in turn and execute «commands»; may unpack compound elements

PASS
Do nothing

«KEYWORD» «expr» «KEYWORD» ...
Execute user-defined command

«KEYWORD»
Execute refinement command

"We did requirements and task analysis, iterative design, and user testing. You'd almost think programming languages were an interface between people and computers."
Steven Pemberton (CWI)

Functions

Numeric

~x
Approximate value of x

exactly x
Exact value of x

exact x
Test if x is exact

+x, x+y, x-y, -x, x*y, x/y, xy**
Arithmetic operators (** = power)

root x, n root x
Square root, n-th root

abs x, sign x
Absolute value, sign (-1, 0, or +1)

round x, floor x, ceiling x
Rounded to whole number

n round x
x rounded to n digits after decimal point

a mod n
Remainder of a when divided by n

***/x, /*x**
Numerator, denominator of exact number x

andom
Random approximate number r, 0 <= r < 1

e, exp x
Base of natural logarithm, exponential function

log x, b log x
Natural logarithm, logarithm to the base b

pi, sin x, cos x, tan x, arctan x
Trigonometric functions, with x in radians

angle (x, y), radius (x, y)
Angle of and radius to point (x, y)

c sin x, c cos x, c tan x,
c arctan x, c angle (x, y)
Similar, with the circle divided into c parts
(e.g. 360 for degrees)

Text

x<<n, x><n, x>>n
x converted to text, aligned left, center, right in width n

t^u
t and u concatenated

t^^n
t repeated n times

lower t, upper t
lower "aBc" = "abc"

stripped t
Strip leading and trailing spaces

split t
Split text t into table of words

Train

#train
Number of elements in train

e#train
Number of elements equal to e

e in train, e not.in train
Test for presence or absence

min train
Smallest element of train

e min train
Smallest element larger than e

max train, e max train
Largest element

train item n
n-th element

choice train
Random element

keys table
List of all keys in table

Now

now
e.g. (1999, 12, 31, 23, 59, 59.999)

Functions

`~x`
Approximate value of x

`exactly x`
Exact value of x

`exact x`
Test if x is exact

`+x, x+y, x-y, -x, x*y, x/y, x**y`
Arithmetic operators (** = power)

`root x, n root x`
Square root, n-th root

`abs x, sign x`
Absolute value, sign (-1, 0, or +1)

`round x, floor x, ceiling x`
Rounded to whole number

`n round x`
x rounded to n digits after decimal point

Text

`x<<n, x><n, x>>n`
x converted to text, aligned left, center, right in width n

`t^u`
t and u concatenated

`t^^n`
t repeated n times

`lower t, upper t`
lower "aBc" = "abc"

`stripped t`
Strip leading and trailing spaces

`split t`
Split text t into table of words

Train

`#train`
Number of elements in train

`e#train`
Number of elements equal to e

Absolute value, sign (-1, 0, or +1)

Numeric

round x, floor x, ceiling x
Rounded to whole number

n round x
x rounded to n digits after decimal point

a mod n
Remainder of a when divided by n

***/x, /*x**
Numerator, denominator of exact number x

andom
Random approximate number r, 0 <= r < 1

e, exp x
Base of natural logarithm, exponential function

log x, b log x
Natural logarithm, logarithm to the base b

pi, sin x, cos x, tan x, arctan x
Trigonometric functions, with x in radians

angle (x, y), radius (x, y)
Angle of and radius to point (x, y)

c sin x, c cos x, c tan x,
c arctan x, c angle (x, y)
Similar, with the circle divided into c parts
(e.g. 360 for degrees)

Split text t into table of words

Train

#train
Number of elements in train

e#train
Number of elements equal to e

e in train, e not.in train
Test for presence or absence

min train
Smallest element of train

e min train
Smallest element larger than e

max train, e max train
Largest element

train item n
n-th element

choice train
Random element

keys table
List of all keys in table

Now

now
e.g. (1999, 12, 31, 23, 59, 59.999)

How To's

HOW TO «KEYWORD» ... :
«statements»
QUIT
 	
«refinements»
	Define a command (procedure) «KEYWORD». The signature may be formed by multiple uppercase keywords interleaved with lowercase parameter names.
	The QUIT command is optional: it is used to terminate the procedure before falling off the end. Refinements are code blocks which start with an identifier and a colon.
	The example shows the definition of a command named DISPLAY/INDENTED which takes a **train** and a number (**n**) as arguments; **spaces** is a function refinement: it returns a text made of **n** spaces.
		`>>> HOW TO DISPLAY train INDENTED n:`
		`HOW TO DISPLAY train INDENTED n:`
		`  FOR item IN train:`
		`  WRITE spaces, item /`
		`spaces: RETURN ' '^^n`
		
		`>>> DISPLAY {11; 22; 33} INDENTED 10`
		`  11`
		`  22`
		`  33`

HOW TO RETURN «name» ... :
«statements»
RETURN «value»
 	
«refinements»
	Define a function «name». The signature is formed by one name with 0, 1 or 2 arguments according to the syntax:
	• no arguments: «name»
	• one argument: «name» «arg»
	• two arguments: «arg1» «name» «arg2».
	A mandatory RETURN command terminates the function.
		`>>> HOW TO RETURN side1 hypotenuse side2:`
		`HOW TO RETURN side1 hypotenuse side2:`
		`  RETURN root (side1*side1 + side2*side2)`
		
		`>>> WRITE 3 hypotenuse 4`
		`5`

HOW TO REPORT «name» ... :
«statements»
SUCCEED
FAIL
REPORT «test»
 	
«refinements»
	Define a predicate «name». The signature may be formed by one name with 0, 1 or 2 arguments according to the function syntax (see above). Predicate execution must terminate with SUCCEED (test condition is true), FAIL (test condition is false) or REPORT «test», in which case the condition evaluates to the result of «test».
	The example shows a predicate named **only.consonants** which succeeds if given a **text** argument with no vowels. Within **only.consonants** there is a predicate refinement named **vowel** which reports whether the value of **char** at the point of invocation is one of 'AEIOU' (after converting **char** to uppercase).
		`>>> HOW TO REPORT only.consonants text:`
		`HOW TO REPORT only.consonants text:`
		`  FOR char IN text:`
		`  IF vowel: FAIL`
		`  SUCCEED`
		`vowel: REPORT upper char in 'AEIOU'`
		
		`>>> CHECK only.consonants 'Ni!'`
		`*** Your check failed in your command`
		`  CHECK only.consonants 'Ni!'`
		`>>> PUT 'Ng' IN name`
		`>>> IF only.consonants name:`
		`  WRITE "I can't pronounce ", name`
		
		`I can't pronounce Ng`

Environment

>> list workspaces

== list permanent locations

:: list how to's

"[…] the integrated structured editor, which [ABC] users almost universally hated.[1]"
Guido van Rossum

syntax directed editor

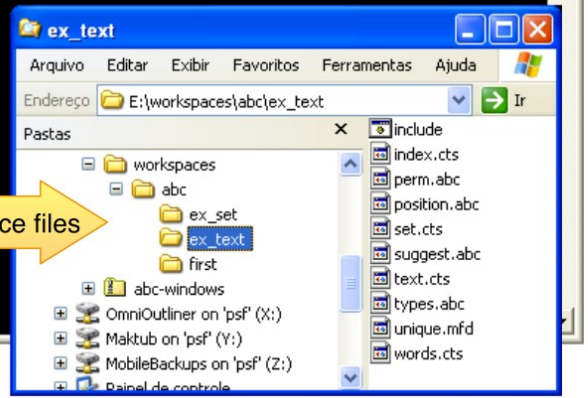
```
C:\WINDOWS\system32\cmd.exe - abc.bat
ABC Release 1.05.02 (central workspace).
Copyright (c) Stichting Mathematisch Centrum, Amsterdam, 1991
Type '?' for help.
>first
>>> >>
ex.set ex.text >first
>>> >ex.text
>>> ==
index set text words
>>> ::
HOW TO INCLUDE element IN set:
HOW TO RETURN unique train:
>>> :INCLUDE ? IN ?
HOW TO INCLUDE element IN set:
   IF element not.in set:
      INSERT element IN set

>>> :unique
HOW TO RETURN unique train:
   PUT {} IN set
   FOR item IN train:
      IF item not.in set:
         INSERT item IN set
   RETURN set
```

first: default workspace

>ex.text: select workspace

:keyword: edit how to

workspace files



1. Foreword for "Programming Python" (1st ed.) http://www.python.org/doc/essays/foreword/

Tests

```
>>> WRITE s1
{0; 2; 4; 6}
>>> WRITE s3
{3; 4}
>>> HOW TO REPORT all.even train:
HOW TO REPORT all.even train:
   REPORT EACH n IN train HAS even
even: REPORT n mod 2 = 0

>>> CHECK all.even s1
>>> CHECK all.even s3
*** Your check failed in your command
    CHECK all.even s3
>>> HOW TO FIND.ODD train:
HOW TO FIND.ODD train:
   SELECT:
      EACH n IN train HAS n mod 2 = 0:
         WRITE "no odd number found"
      ELSE:
         WRITE "found:", n

>>> FIND.ODD s1
no odd number found
>>> FIND.ODD s3
found: 3
```

ABC 1.05

Quantifiers

SOME «name»,… IN «train» HAS «test»
Sets «name», ... on success. May unpack compound element

EACH «name»,… IN «train» HAS «test»
Sets «name», ... on failure. May unpack compound element

NO «name»,… IN «train» HAS «test»
Sets «name», ... on failure. May unpack compound element

Built-in Predicates

e in train, e not.in train
Test for presence or absence

exact x
Test if x is exact

Expressions

x < y, x <= y, x >= y, x > y
x = y, x <> y, x <= z < y
Order tests (<> is 'not equals')

«pred», «pred» x, x «pred» y
Outcome of predicate «pred» (no permanent effects)

«pred»
Outcome of refinement predicate «pred» (no permanent effects)

«test» AND «test» AND ...
Fails as soon as one of the tests fails

«test» OR «test» OR ...
Succeeds as soon as one of the tests succeeds

NOT «test»
Succeeds if «test» fails

Python 3.2

```
>>> def all_even(seq):
...     return all(n%2==0 for n in seq)
...
>>> s1
[0, 2, 4, 6]
>>> s3
[3, 4]
>>> all_even(s1), all_even(s3)
(True, False)
>>> def first_odd(seq):
...     for n in seq:
...         if n%2: return n
...
>>> def find_odd(seq):
...     found = first_odd(seq)
...     if found is None:
...         print('no odd number found')
...     else:
...         print('found:', found)
...
>>> find_odd(s1)
no odd number found
>>> find_odd(s3)
found: 3
```

Tests

```
>>> WRITE s1
{0; 2; 4; 6}
>>> WRITE s3
{3; 4}
>>> HOW TO REPORT all.even train:
HOW TO REPORT all.even train:
   REPORT EACH n IN train HAS even
even: REPORT n mod 2 = 0

>>> CHECK all.even s1
>>> CHECK all.even s3
*** Your check failed in your command
    CHECK all.even s3
>>> HOW TO FIND.ODD train:
HOW TO FIND.ODD train:
   SELECT:
      EACH n IN train HAS n mod 2 = 0:
         WRITE "no odd number found"
      ELSE:
         WRITE "found:", n

>>> FIND.ODD s1
no odd number found
>>> FIND.ODD s3
```

Quantifiers

SOME «name»,… IN «train» HAS «test»
Sets «name», ... on success. May unpack compound element

EACH «name»,… IN «train» HAS «test»
Sets «name», ... on failure. May unpack compound element

NO «name»,… IN «train» HAS «test»
Sets «name», ... on failure. May unpack compound element

Built-in Predicates

e in train, e not.in train
Test for presence or absence

exact x
Test if x is exact

Python 3.2

```
      EACH n IN train HAS n MOD 2 = 0:
         WRITE "no odd number found"
      ELSE:
         WRITE "found:", n

>>> FIND.ODD s1
no odd number found
>>> FIND.ODD s3
found: 3
```

ABC 1.05

e in train, e not.in train
Test for presence or absence

exact x
Test if x is exact

Python 3.2

```
>>> def all_even(seq):
...     return all(n%2==0 for n in seq)
...
>>> s1
[0, 2, 4, 6]
>>> s3
[3, 4]
>>> all_even(s1), all_even(s3)
(True, False)
>>> def first_odd(seq):
...     for n in seq:
...         if n%2: return n
...
>>> def find_odd(seq):
...     found = first_odd(seq)
...     if found is None:
...         print('no odd number found')
...     else:
...         print('found:', found)
...
>>> find_odd(s1)
no odd number found
>>> find_odd(s3)
found: 3
```

Expressions

x < y, x <= y, x >= y, x > y
x = y, x <> y, x <= z < y
Order tests (<> is 'not equals')

«pred», «pred» x, x «pred» y
Outcome of predicate «pred» (no permanent effects)

«pred»
Outcome of refinement predicate «pred» (no permanent effects)

«test» AND «test» AND ...
Fails as soon as one of the tests fails

«test» OR «test» OR ...
Succeeds as soon as one of the tests succeeds

NOT «test»
Succeeds if «test» fails

Jargon

term	meaning in ABC	Python perspective
address	Name or expression that may fill the 2nd **hole** of PUT/INTO to receive a value. Examples: total, phones[name], word@3|2.	These are like the expressions that may appear on the left side of an assignment; also called L-values in CS theory. Once an address is bound to a value it's type cannot change.
command	A built-in command or a user-defined procedure created with the HOW TO command, taking any number of arguments.	ABC commands receive the parameters by reference, and can change the value of all actual arguments passed. Command names are always uppercase (enforced by the editor).
compound	Similar to a record but without field names. Used in PUT commands with multiple values, as composite keys in tables and as arguments for **functions** or **predicates** that require more than two parameters.	ABC compounds are like Python tuples. Packing and unpacking is supported, but not item access or iteration.
formula	An expression composed of one operator or user defined **function** and zero, one or two operands or arguments.	Formula syntax is the same for operators and functions. A function that takes one argument is used like a prefix operator; if it takes two arguments, it is used like an infix operator.
function	A function that returns a value (of any type). A function may take zero, one or two arguments. Execution must end with a RETURN command.	Functions and predicates are side-effect free by definition: they receive copies of all actual arguments. Function names are always lowercase (enforced by the editor).
hole	In the ABC environment, a hole is a missing element in the syntax of the line being edited. Holes are marked with **?**	Python does not include a syntax-directed console or editor, so there's no analog of holes.
how to	A user defined subroutine (**command**, **function** or **predicate**). The command HOW TO «keyword» starts the definition of a how to, and changes the editing mode of the environment. If «keyword» is RETURN, a function is defined; if REPORT, a predicate is defined; otherwise, a command named «keyword» is defined. See also **refinement**.	ABC distinguishes between commands (procedures which can change the environment and do not return a value), functions (which cannot affect the environment and must return a value) and predicates (cannot affect the environment either, and can only be used in tests). A **refinement** is yet another subroutine-like construct.
predicate	A **function** that tests a condition on zero, one or two arguments. Execution of a predicate must end with REPORT, SUCCEED or FAIL	It is not possible to store the result of a predicate: there is no boolean data type in ABC. Predicates can only be called where a **test** is expected and the result is used immediately.
refinement	A subroutine defined and accessible only within the body of another subroutine. Refinements provide syntactic support to "stepwise refinement" in top-down programming. Refinements are written at the end of a HOW TO, and do not have parameters or local variables; they share the names defined in the enclosing scope.	There is no good analog for ABC refinements in Python. A refinement is like a function defined within another function, sharing the same scope of the outer function, and therefore able to change any variable of the outer function. ABC refinements are an example of dynamic scoping: free variables in refinements are bound at the point of invocation. See the predicate example in the How To's panel.
test	Expressions used as conditions in the IF, SELECT, WHILE and CHECK commands. Tests are built using comparison operators (=, <>, >, >=, <, <=) or predicate calls. Tests may be combined with the boolean operators AND, OR and NOT.	There is no boolean data type in ABC. Tests can only appear where a condition is expected. There is no way to assign the result of test to a variable.
train	The iterable data types, which can be used with the FOR command: **text**, **list** and **table**.	Similar to sequences in Python. However, iterating over a table gets the values, not the keys.
workspace	ABC programs are organized in workspaces, where HOW TOs and the contents of global variables are stored. Because of this feature, global variables are called "persistent locations" in ABC. The layout of a workspace in the filesystem is an implementation detail.	Each workspace is a directory with several files, one per HOW TO and global variable, plus an index and other auxiliary files. ABC has no support for file handling under user control. The only way to move bulk data in and out of an ABC program is by reading and writing the workspace files, which store readable representations of ABC data structures.

Jargon

term	meaning in ABC	Python perspective
address	Name or expression that may fill the 2nd **hole** of PUT/INTO to receive a value. Examples: total, phones[name], word@3|2.	These are like the expressions that may appear on the left side of an assignment; also called L-values in CS theory. Once an address is bound to a value it's type cannot change.
command	A built-in command or a user-defined procedure created with the HOW TO command, taking any number of arguments.	ABC commands receive the parameters by reference, and can change the value of all actual arguments passed. Command names are always uppercase (enforced by the editor).
compound	Similar to a record but without field names. Used in PUT commands with multiple values, as composite keys in tables and as arguments for **functions** or **predicates** that require more than two parameters.	ABC compounds are like Python tuples. Packing and unpacking is supported, but not item access or iteration.
formula	An expression composed of one operator or user defined **function** and zero, one or two operands or arguments.	Formula syntax is the same for operators and functions. A function that takes one argument is used like a prefix operator; if it takes two arguments, it is used like an infix operator.
function	A function that returns a value (of any type). A function may take zero, one or two arguments. Execution must end with a RETURN command.	Functions and predicates are side-effect free by definition: they receive copies of all actual arguments. Function names are always lowercase (enforced by the editor).
hole	In the ABC environment, a hole is a missing element in the syntax of the line being edited. Holes are marked with **?**	Python does not include a syntax-directed console or editor, so there's no analog of holes.
how to	A user defined subroutine (**command**, **function** or **predicate**). The command HOW TO «keyword» starts the	ABC distinguishes between commands (procedures which can change the environment and do not return a value), functions (which cannot affect the environment and must

function	take zero, one or two arguments. Execution must end with a RETURN command.	they receive copies of all actual arguments. Function names are always lowercase (enforced by the editor).
hole	In the ABC environment, a hole is a missing element in the syntax of the line being edited. Holes are marked with **?**	Python does not include a syntax-directed console or editor, so there's no analog of holes.
how to	A user defined subroutine (**command**, **function** or **predicate**). The command HOW TO «keyword» starts the definition of a how to, and changes the editing mode of the environment. If «keyword» is RETURN, a function is defined; if REPORT, a predicate is defined; otherwise, a command named «keyword» is defined. See also **refinement**.	ABC distinguishes between commands (procedures which can change the environment and do not return a value), functions (which cannot affect the environment and must return a value) and predicates (cannot affect the environment either, and can only be used in tests). A **refinement** is yet another subroutine-like construct.
predicate	A **function** that tests a condition on zero, one or two arguments. Execution of a predicate must end with REPORT, SUCCEED or FAIL	It is not possible to store the result of a predicate: there is no boolean data type in ABC. Predicates can only be called where a **test** is expected and the result is used immediately.
refinement	A subroutine defined and accessible only within the body of another subroutine. Refinements provide syntactic support to "stepwise refinement" in top-down programming. Refinements are written at the end of a HOW TO, and do not have parameters or local variables; they share the names defined in the enclosing scope.	There is no good analog for ABC refinements in Python. A refinement is like a function defined within another function, sharing the same scope of the outer function, and therefore able to change any variable of the outer function. ABC refinements are an example of dynamic scoping: free variables in refinements are bound at the point of invocation. See the predicate example in the How To's panel.
test	Expressions used as conditions in the IF, SELECT, WHILE and CHECK commands. Tests are built using comparison operators (=, <>, >, >=, <, <=) or predicate calls. Tests may be combined with the boolean operators AND, OR and NOT.	There is no boolean data type in ABC. Tests can only appear where a condition is expected. There is no way to assign the result of test to a variable.
train	The iterable data types, which can be used with the FOR command: **text**, **list** and **table**.	Similar to sequences in Python. However, iterating over a table gets the values, not the keys.
workspace	ABC programs are organized in workspaces, where HOW TOs and the contents of global variables are stored. Because of this feature, global variables are called "persistent locations" in ABC. The layout of a workspace in the filesystem is an implementation detail.	Each workspace is a directory with several files, one per HOW TO and global variable, plus an index and other auxiliary files. ABC has no support for file handling under user control. The only way to move bulk data in and out of an ABC program is by reading and writing the workspace files, which store readable representations of ABC data structures.

Concluindo...

- Criadores do ABC (Meertens & Geurts) ensinaram programação para artistas por muitos anos
- Princípios de design
  - Consistência (“eu comi” → “eu fazi”)
  - Reduzir surpresas
  - Economia de ferramentas
  - Ferramentas adequadas às tarefas
  - Ambiente interativo, resultados imediatos

Idéias interessantes (?)

- Blocos por endentação
- Tratamento uniforme de sequências: laço for suporta “trens” (str, list, table)
- Tipagem forte: variáveis não podem mudar de tipo, listas só aceitam um tipo (1º elemento)
- Desempacotamento de tuplas

Lessons

Guido explains why ABC failed[3]

I had been part of the ABC development team in the early '80s, and in my head I had analyzed some of the reasons it had failed. Failure can be measured in many ways. On the one hand, upper management withdrew all funding from the project; on the other hand, there were few users. I had some understanding for the reasons for the latter, and to some extent Python is a direct response to that.

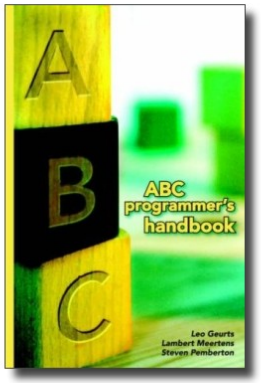
In part, of course, the reason for ABC's failure was that it was too early for such a high-level language. But I believe that several of its early major design decisions contributed to its demise:

- Unconventional terminology intended to make beginners more comfortable but instead threw off more experienced users
- A monolithic implementation that made it hard to add new features
- Too much emphasis on theoretically optimal performance
- Not enough flexibility in its interaction with other software running on the same computer (ABC didn't have a way to open a file from within a program)

Python addresses several of these issues by its object-oriented design and by making it really easy to write extension modules.

3 *An Interview with Guido van Rossum by Bruce Stewart 06/04/2002 http://onlamp.com/lpt/a/2431*

L. Geurts, L. Meertens, S. Pemberton
ABC Programmer's Handbook
ISBN 0-9547239-4-5



What ABC got right

- Focus on simplicity.
- Suitability for interactive use.
- Block structure by indentation.
- The for loop (a.k.a "enhanced for loop" in JSR 201, 25 years later!)
- Choice of built-in types.
- Tuple unpacking.
- Division (thanks to exact numbers)

Years of task analysis, user testing and iterative development.

How many programming languages enjoy the fruits of such a legacy?

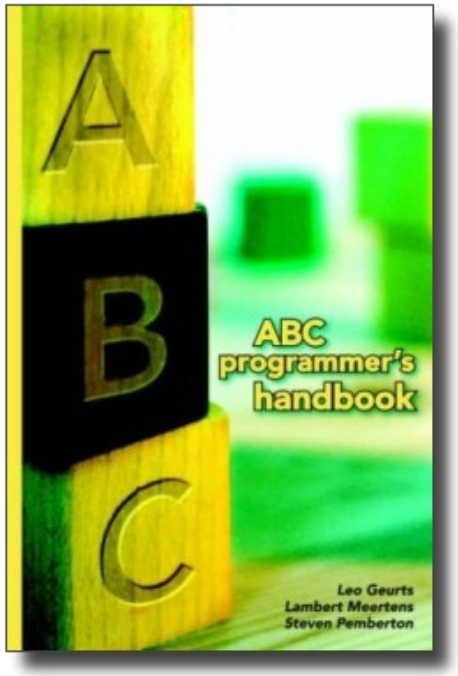
Lessons

Guido explains why ABC failed[3]

I had been part of the ABC development team in the early '80s, and in my head I had analyzed some of the reasons it had failed. Failure can be measured in many ways. On the one hand, upper management withdrew all funding from the project; on the other hand, there were few users. I had some understanding for the reasons for the latter, and to some extent Python is a direct response to that.

In part, of course, the reason for ABC's failure was that it was too early for such a high-level language. But I believe that several of its early major design decisions contributed to its demise:

L. Geurts, L. Meertens, S. Pemberton
ABC Programmer's Handbook
ISBN 0-9547239-4-5



response to that.

In part, of course, the reason for ABC's failure was that it was too early for such a high-level language. But I believe that several of its early major design decisions contributed to its demise:

- Unconventional terminology intended to make beginners more comfortable but instead threw off more experienced users
- A monolithic implementation that made it hard to add new features
- Too much emphasis on theoretically optimal performance
- Not enough flexibility in its interaction with other software running on the same computer (ABC didn't have a way to open a file from within a program)

Python addresses several of these issues by its object-oriented design and by making it really easy to write extension modules.

3 *An Interview with Guido van Rossum*
by Bruce Stewart
06/04/2002
http://onlamp.com/lpt/a/2431

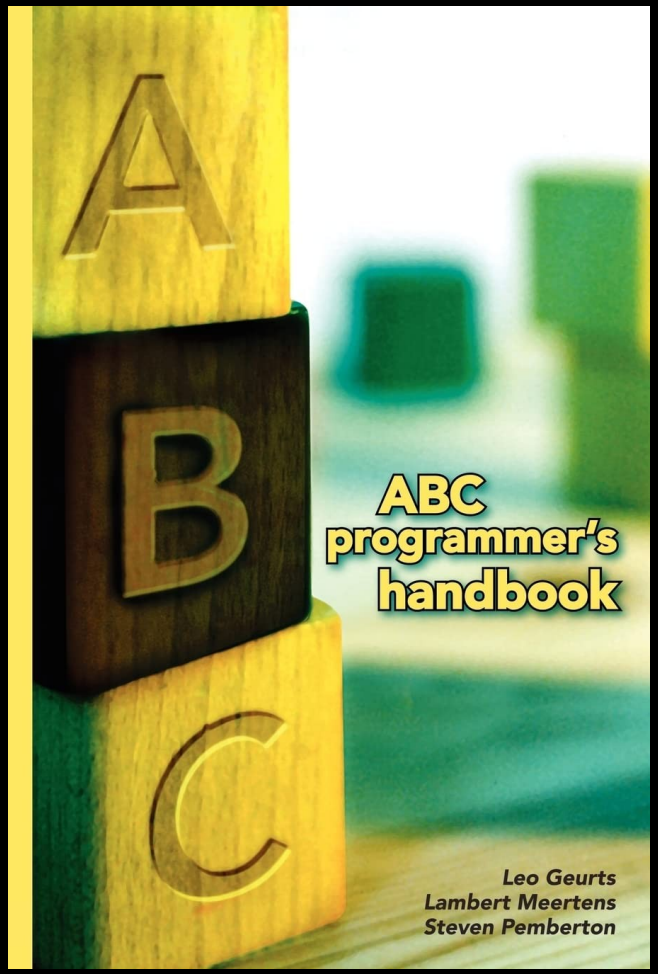
What ABC got right

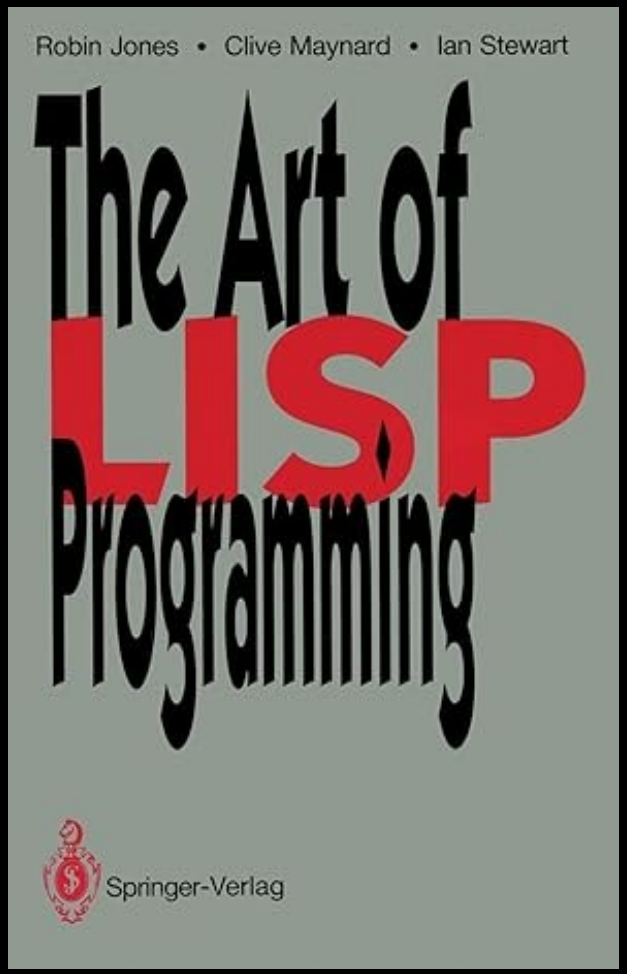
- Focus on simplicity.
- Suitability for interactive use.
- Block structure by indentation.
- The for loop (a.k.a "enhanced for loop" in JSR 201, 25 years later!)
- Choice of built-in types.
- Tuple unpacking.
- Division (thanks to exact numbers)

Years of task analysis, user testing and iterative development.

How many programming languages enjoy the fruits of such a legacy?

Referências





The ABC Programming Language: a short introduction

(Also available in Japanese)

New: The Origins of Python - An article by Lambert Meertens on the origins of ABC, and its influence on Python.

New: Implementation for the Raspberry Pi!.

The ABC Programmer's Handbook is available online.

ABC is an interactive programming language and environment for personal computing, originally intended as a good replacement for BASIC. It was designed by first doing a *task analysis* of the programming task.

ABC is easy to learn (an hour or so for someone who has already programmed), and yet easy to use. Originally intended as a language for beginners, it has evolved into a powerful tool for beginners and experts alike.

Here is an example function *words* to collect the set of all words in a document:

```
HOW TO RETURN words document:
   PUT {} IN collection
   FOR line IN document:
      FOR word IN split line:
         IF word not.in collection:
            INSERT word IN collection
   RETURN collection
```

Some features of the language:

- a powerful collection of only 5 data types that can easily be combined
- strong typing, yet without declarations
- no limitations (such as max int), apart from sheer exhaustion of memory

INFERENCE

COMPUTER SCIENCE / REVIEW ESSAY

VOL. 7, NO. 3 / NOVEMBER 2022

The Origins of Python

Lambert Meertens

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