ThoughtWorks®

Elegance begets simplicity

BEAUTIFUL PYTHON

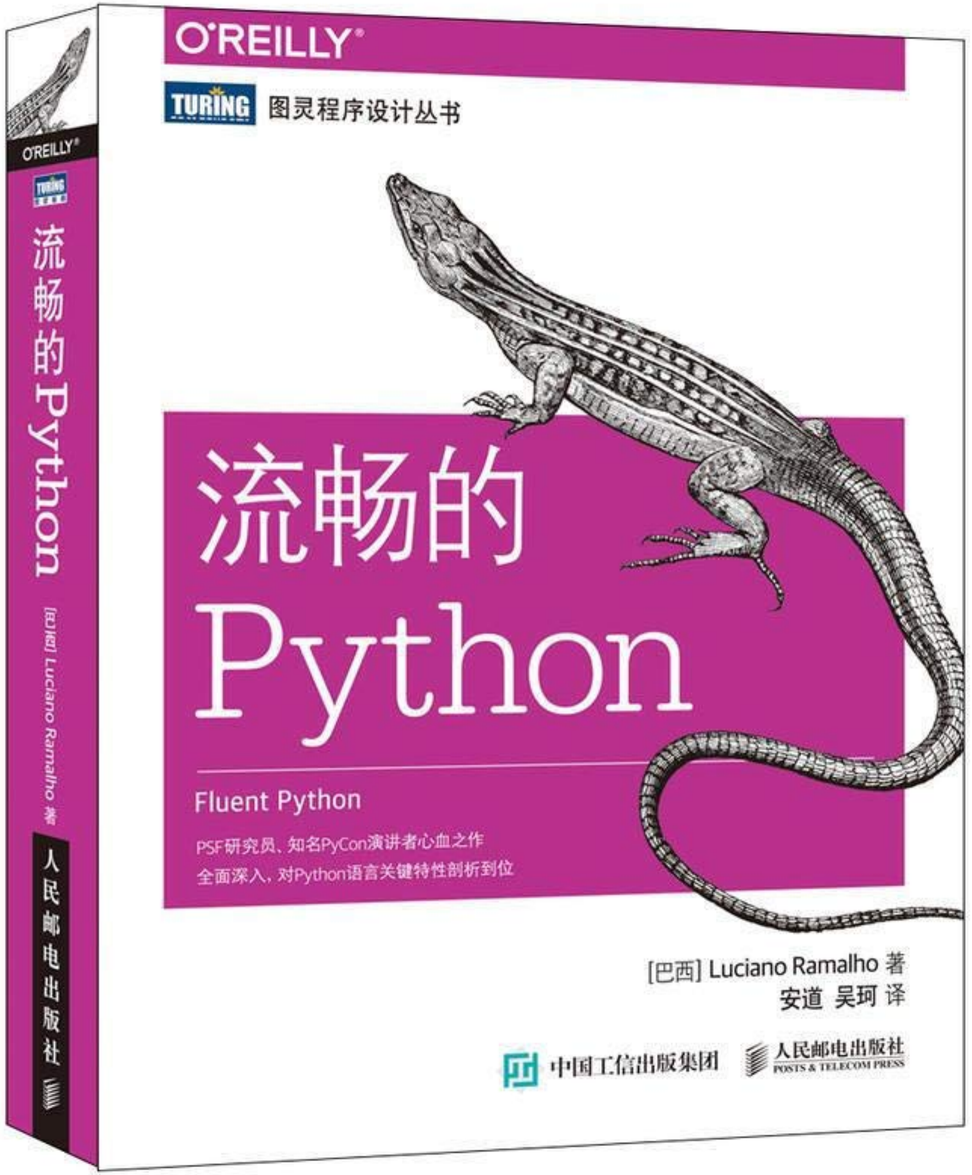
Great ideas of language design that make Python enjoyable and useful to so many people.

python™
brasil

pycon china

python™ brasil

FLUENT PYTHON



Published in 9 languages so far:

- Chinese (simplified)
- Chinese (traditional)
- English
- French
- Japanese
- Korean
- Polish
- Portuguese
- Russian

Book signing at 12:00 today on 3rd floor

2nd edition: Q3 of 2020

ThoughtWorks®

INTRODUCTION

To Love and create beauty are conditions to happiness.

— Walter Gropius (1883–1969), founder of Bauhaus School

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
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.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough
```

SIMPLE SYNTAX

A SIMPLE PROGRAM IN C

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for(int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

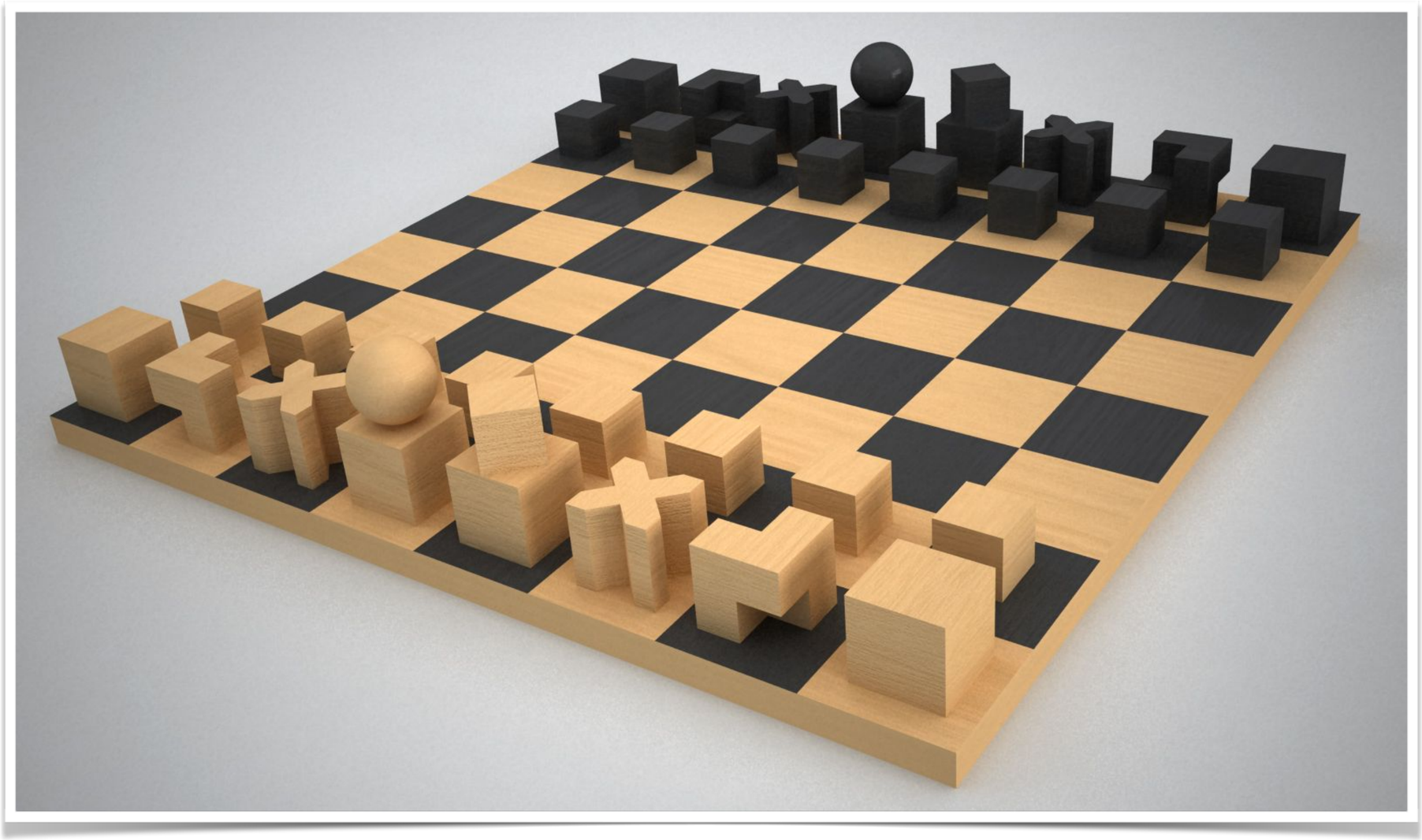
A SIMPLE PROGRAM IN C

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for(int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

```
$ ./args alpha bravo charlie
./args
alpha
bravo
charlie
```

A TRADITIONAL CHESS SET

A BAUHAUS CHESS SET BY JOSEF HARTWIG



A SIMPLE PROGRAM IN PYTHON

```
import sys

for arg in sys.argv:
    print arg
```

PYTHON: A MODERNIST LANGUAGE

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for(int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

```python
import sys

for arg in sys.argv:
    print arg
```

PEP 3099 -- Things that will Not Change in Python 3000

PEP:	3099
Title:	Things that will Not Change in Python 3000
Author:	Georg Brandl <georg at python.org>
Status:	Final
Type:	Process
Created:	04-Apr-2006
Post-History:	

Contents

- It will not be forbidden to reuse a loop variable inside the loop's suite.

> *Thread: elimination of scope bleeding of iteration variables https://mail.python.org/pipermail/python-dev/2006-May/064761.html*

- The parser won't be more complex than LL(1).

> *Simple is better than complex. This idea extends to the parser. Restricting Python's grammar to an LL(1) parser is a blessing, not a curse. It puts us in handcuffs that prevent us from going overboard and ending up with funky grammar rules like some other dynamic languages that will go unnamed, such as Perl.*

- No braces.

> *This is so obvious that it doesn't need a reference to a mailing list. Do `from __future__ import braces` to get a definitive answer on this subject.*

- No more backticks.

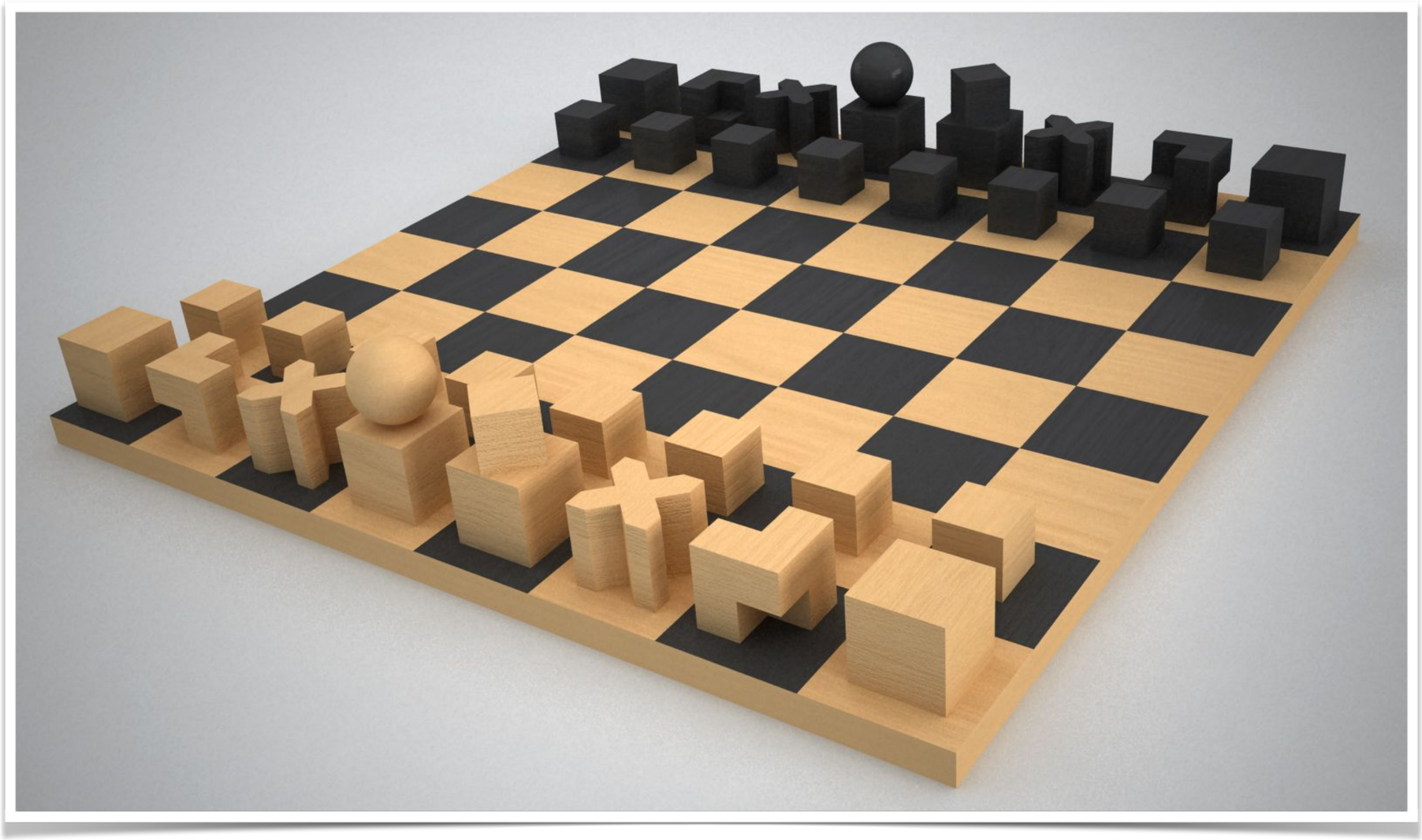
> *Backticks (`) will no longer be used as shorthand for `repr` -- but that doesn't mean they are available for other uses. Even ignoring the backwards compatibility confusion, the character itself causes too many problems (in some fonts, on some keyboards, when typesetting a book, etc).*
>
> *Thread: "new operators via backquoting", https://mail.python.org/pipermail/python-ideas/2007-January/000054.html*

PYTHON: A MODERNIST LANGUAGE

- The parser won't be more complex than LL(1).

> *Simple is better than complex. This idea extends to the parser. Restricting Python's grammar to an LL(1) parser is a blessing, not a curse. It puts us in handcuffs that prevent us from going overboard and ending up with funky grammar rules like some other dynamic languages that will go unnamed, such as Perl.*

MODERNIST



```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough
```

ANCIENT AND MODERNIST

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough
```

ThoughtWorks®

ENHANCED FOR

THAT SAME SIMPLE PROGRAM IN JAVA < 5

```
class Arguments {
    public static void main(String[] args) {
        for (int i=0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

Specialists are people who always repeat the same mistakes.

— Walter Gropius

FOREACH: SINCE JAVA 5

The official name of the *foreach* syntax in Java is *"enhanced for"*

```
class Arguments2 {
    public static void main(String[] args) {
        for (String arg : args)
            System.out.println(arg);
    }
}
```

FOREACH: SINCE JAVA 5

The official name of the *foreach* syntax in Java is *"enhanced for"*

```
class Arguments2 {
    public static void main(String[] args) {
        for (String arg : args)
            System.out.println(arg);
    }
}
```

```
import sys

for arg in sys.argv:
    print arg
```

FOREACH: SINCE JAVA 5

The official name of the *foreach* syntax in Java is *"enhanced for"*

year: 2004

```
class Arguments2 {
    public static void main(String[] args) {
        for (String arg : args)
            System.out.println(arg);
    }
}
```

year: 1991

```
import sys

for arg in sys.argv:
    print arg
```

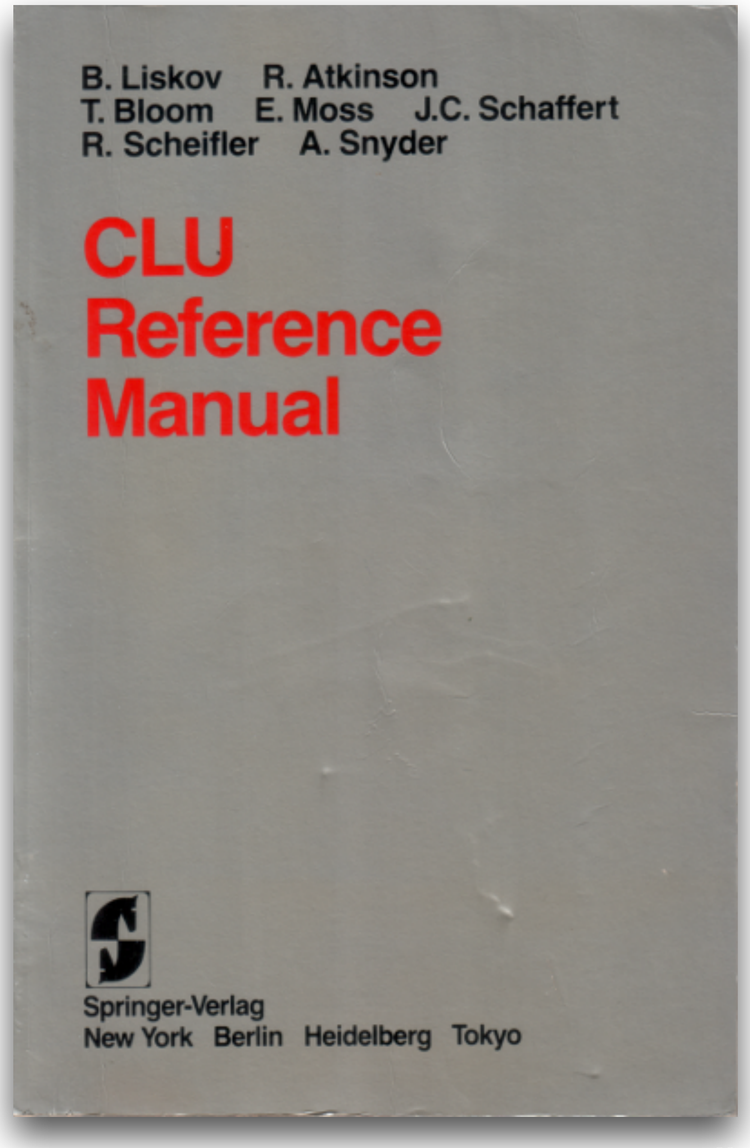
FOREACH IN BARBARA LISKOV'S CLU

Barbara Liskov,
Institute Professor at MIT

FOREACH IN BARBARA LISKOV'S CLU

Prof. Liskov and her students

Barbara Liskov,
Institute Professor at MIT



CLU REFERENCE MANUAL

year: 1975

types of results can be returned in the exceptional conditions. All information about the names of conditions, and the number and types of arguments and results is described in the *iterator heading*. For example,

leaves = **iter** (t: tree) **yields** (node)

is the heading for an iterator that produces all leaf nodes of a tree object. This iterator might be used in a **for** statement as follows:

```
for leaf: node in leaves(x) do
    ... examine(leaf) ...
end
```

CLU Reference Manual, p. 2
B. Liskov et. al. — © 1981 Springer-Verlag

ThoughtWorks®

ITERABLES

ITERABLE OBJECTS: THE KEY TO FOREACH

CLU, Python, Java, etc. let we create **iterable** objects

```
for item in an_iterable:
    process(item)
```

ITERABLE OBJECTS: THE KEY TO FOREACH

CLU, Python, Java, etc. let we create **iterable** objects

```
for item in an_iterable:
    process(item)
```

Some languages don't offer this flexibility

- C has no concept of iterables
- In Go, only some built-in types are iterable and can be used with foreach (written as the **for … range** special syntax)

avoidable
believable
extensible
fixable
iterable
movable
readable
playable
washable

iterable, adj. — Capable of being iterated.

ThoughtWorks®

THE ITERATOR PATTERN

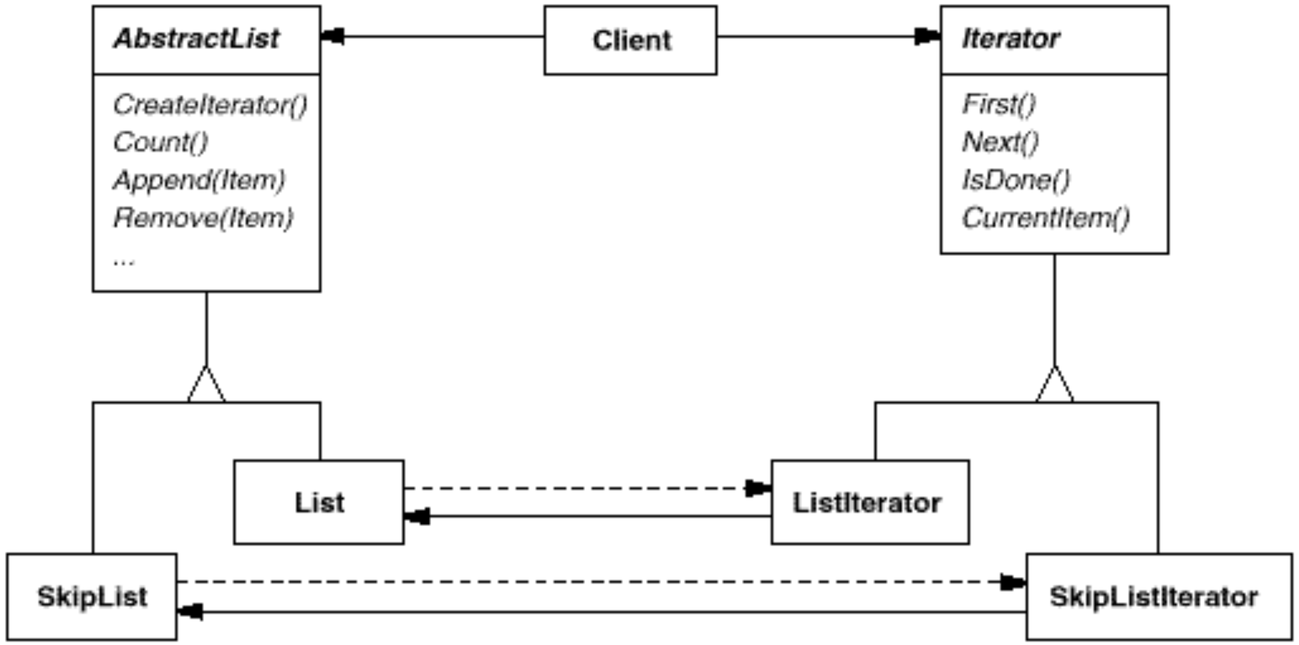
The classic recipe

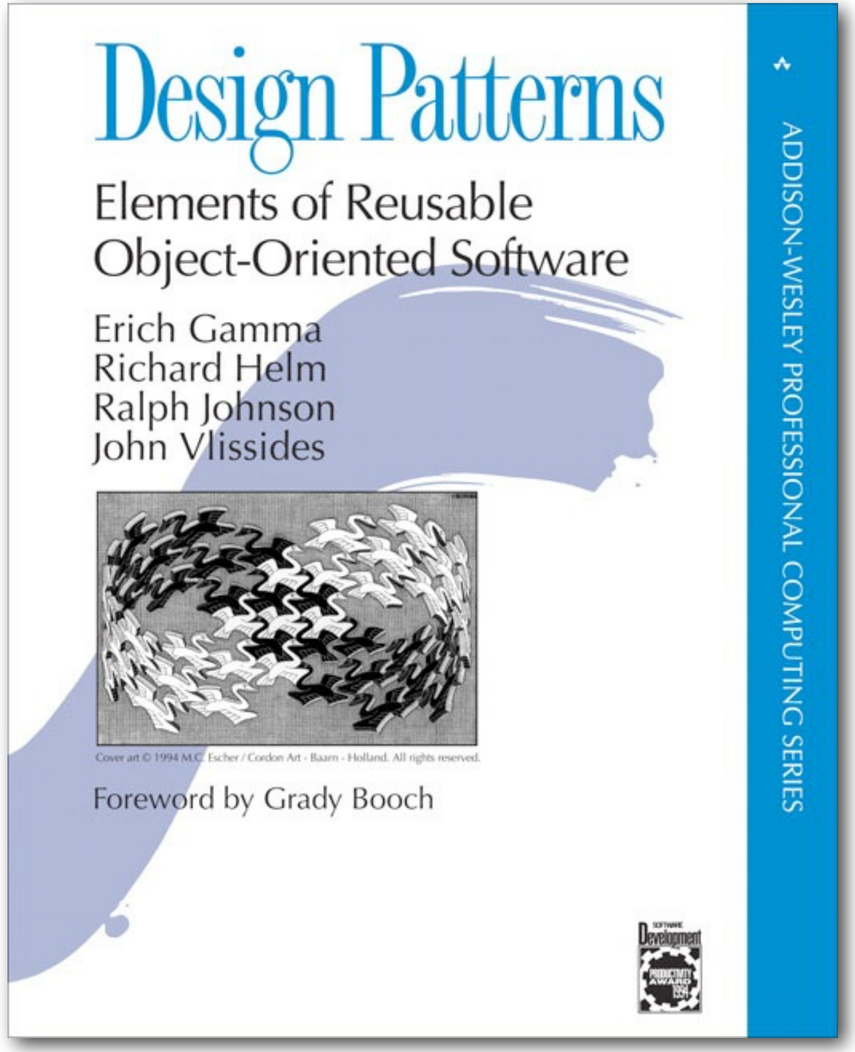
THE ITERATOR FROM THE GANG OF FOUR

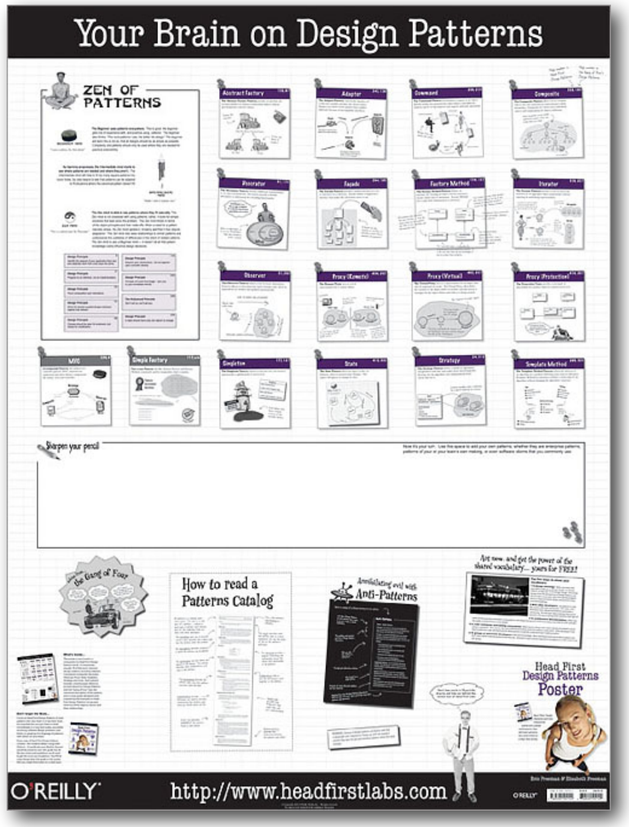
Design Patterns

Gamma, Helm, Johnson & Vlissides

©1994 Addison-Wesley



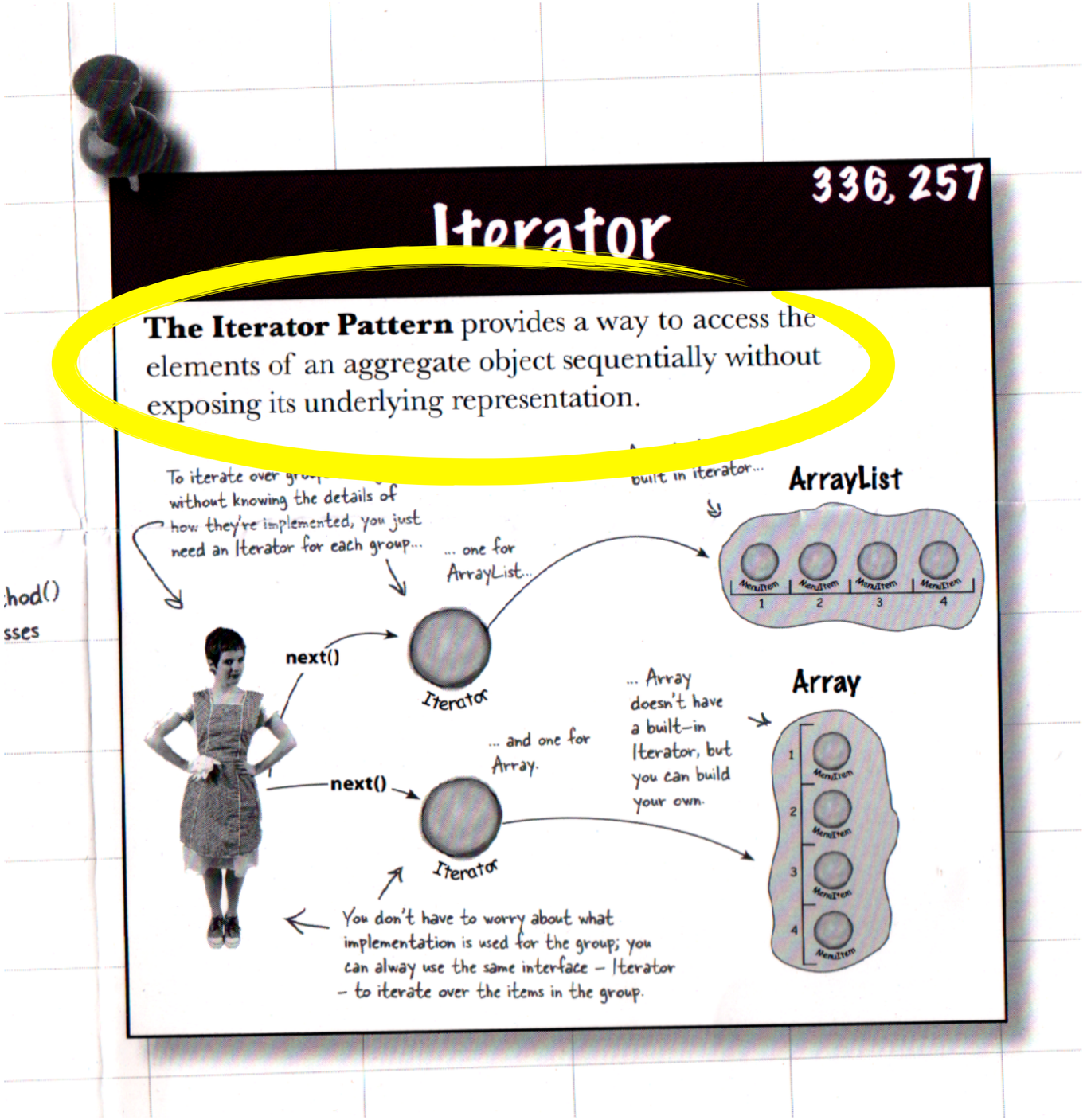




Head First Design Patterns Poster

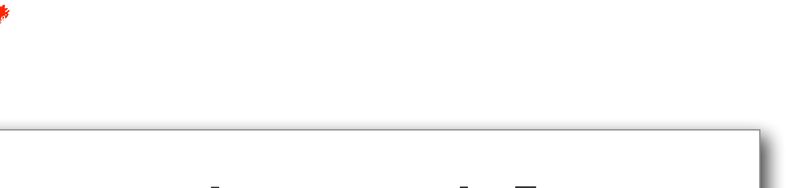
O'Reilly

ISBN 0-596-10214-3



THE FOR LOOP MACHINERY

- In Python, the **for** loop, automatically:
  - Obtains an **iterator** from the **iterable**
  - Repeatedly invokes **next()** on the **iterator**, retrieving one item at a time
  - Assigns the item to the loop variable(s)



```
for item in an_iterable:
    process(item)
```

  - Terminates when a call to **next()** raises **StopIteration**.

AN ITERABLE TRAIN

An instance of **Train** can be iterated, car by car

```
>>> t = Train(3)
>>> for car in t:
...     print(car)
car #1
car #2
car #3
>>>
```

CLASSIC ITERATOR IMPLEMENTATION

The pattern as described

by Gamma et. al.

```
>>> t = Train(4)
>>> for car in t:
...   print(car)
car #1
car #2
car #3
car #4
```

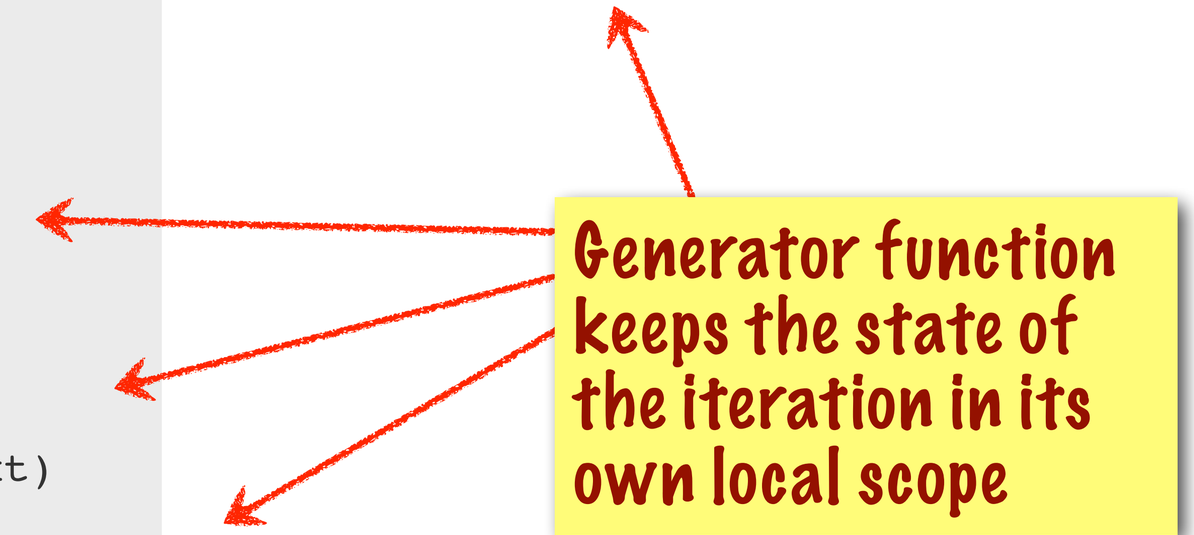
```
class Train:

    def __init__(self, cars):
        self.cars = cars

    def __iter__(self):
        return TrainIterator(self.cars)

class TrainIterator:

    def __init__(self, cars):
        self.next = 0
        self.last = cars - 1

    def __next__(self):
        if self.next <= self.last:
            self.next += 1
            return 'car #%s' % (self.next)
        else:
            raise StopIteration()
```

COMPARE: CLASSIC ITERATOR × GENERATOR METHOD

The classic Iterator recipe is obsolete in Python since v.2.2 (2001)

```
class Train:

    def __init__(self, cars):
        self.cars = cars

    def __iter__(self):
        return IteratorTrem(self.cars)

class TrainIterator:

    def __init__(self, cars):
        self.next = 0
        self.last = cars - 1

    def __next__(self):
        if self.next <= self.last:
            self.next += 1
            return 'car #%s' % (self.next)
        else:
            raise StopIteration()
```

```
class Train:

    def __init__(self, cars):
        self.cars = cars

    def __iter__(self):
        for i in range(self.cars):
            yield 'car #%s' % (i+1)
```



```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough
```

SPECIAL METHODS

ThoughtWorks®

THE PYTHONIC DEFINITION OF ITERABLE

iterable, adj. — (Python) An object from which the **iter()** function can build an **iterator**.

iter(*iterable*):

Returns iterator for *iterable* by:

- invoking **__iter__** (if available)
  or
- building an iterator to fetch items via **__getitem__** with 0-based indices (seq[0], seq[1], etc...)

SPECIAL METHODS

Table 1-1. Special method names (operators excluded)

Category	Method names
String/bytes representation	`__repr__`, `__str__`, `__format__`, `__bytes__`
Conversion to number	`__abs__`, `__bool__`, `__complex__`, `__int__`, `__float__`, `__hash__`, `__index__`
Emulating collections	`__len__`, `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`
Iteration	`__iter__`, `__reversed__`, `__next__`
Emulating callables	`__call__`
Context management	`__enter__`, `__exit__`
Instance creation and destruction	`__new__`, `__init__`, `__del__`
Attribute management	`__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__dir__`
Attribute descriptors	`__get__`, `__set__`, `__delete__`
Class services	`__prepare__`, `__instancecheck__`, `__subclasscheck__`

ThoughtWorks®

THE ZEN OF LEN

LEN FUNCTION AND __LEN__ METHOD

Why do we call **len(x)** and not **x.len()**?

For performance, the length of a sequence must be fast to get.

For built-in collections (**str**, **list**, **tuple**, **array**, **set**, etc.) and extensions (Numpy arrays), the interpreter uses the C API to get the length from the **ob_size** field in the **PyVarObject** struct in memory.

For collections defined in Python code, the interpreter calls the user-defined **__len__** special method.

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess
There should be one-- and preferably only one --obvious
Although that way may not be obvious at first unless you
Now is better than never.
Although never is often better than *right* now.

ThoughtWorks®

OPERATOR OVERLOADING

Table 13-1. Infix operator method names (the in-place operators are used for augmented assignment; comparison operators are in Table 13-2)

Operator	Forward	Reverse	In-place	Description
`+`	`__add__`	`__radd__`	`__iadd__`	Addition or concatenation
`-`	`__sub__`	`__rsub__`	`__isub__`	Subtraction
`*`	`__mul__`	`__rmul__`	`__imul__`	Multiplication or repetition
`/`	`__truediv__`	`__rtruediv__`	`__itruediv__`	True division
`//`	`__floordiv__`	`__rfloordiv__`	`__ifloordiv__`	Floor division
`%`	`__mod__`	`__rmod__`	`__imod__`	Modulo
`divmod()`	`__divmod__`	`__rdivmod__`	`__idivmod__`	Returns tuple of floor division quotient and modulo
`**`, `pow()`	`__pow__`	`__rpow__`	`__ipow__`	Exponentiation[a]
`@`	`__matmul__`	`__rmatmul__`	`__imatmul__`	Matrix multiplication[b]
`&`	`__and__`	`__rand__`	`__iand__`	Bitwise and
`|`	`__or__`	`__ror__`	`__ior__`	Bitwise or
`^`	`__xor__`	`__rxor__`	`__ixor__`	Bitwise xor
`<<`	`__lshift__`	`__rlshift__`	`__ilshift__`	Bitwise shift left
`>>`	`__rshift__`	`__rrshift__`	`__irshift__`	Bitwise shift right

[a] `pow` takes an optional third argument, `modulo`: `pow(a, b, modulo)`, also supported by the special methods when invoked directly (e.g., `a.__pow__(b, modulo)`).

[b] New in Python 3.5.

ThoughtWorks®

EXAMPLE: SET OPERATORS

SETS IN A FEW LANGUAGES AND PLATFORMS

Some languages/platform APIs that implement sets in their standard libraries

Python	**set**, **frozenset**: > 10 methods and operators
Ruby	**Set**: > 10 methods and operators
Elixir	**MapSet**: > 10 methods
.Net (C# etc.)	**ISet** interface: > 10 methods; 2 implementations

SETS IN A FEW LANGUAGES AND PLATFORMS

Some languages/platform APIs that implement sets in their standard libraries

Python	**set**, **frozenset**: > 10 methods and operators	😻
Ruby	**Set**: > 10 methods and operators	😻
Elixir	**MapSet**: > 10 methods	😻
.Net (C# etc.)	**ISet** interface: > 10 methods; 2 implementations	😻

SETS IN A FEW LANGUAGES AND PLATFORMS

Some languages/platform APIs that implement sets in their standard libraries

Python	**set**, **frozenset**: > 10 methods and operators	😻
Ruby	**Set**: > 10 methods and operators	😻
Elixir	**MapSet**: > 10 methods	😻
.Net (C# etc.)	**ISet** interface: > 10 methods; 2 implementations	😻
JavaScript (ES6)	**Set**: < 10 methods	😿
Java	**Set** interface: < 10 methods; 8 implementations	😿

SETS IN A FEW LANGUAGES AND PLATFORMS

Some languages/platform APIs that implement sets in their standard libraries

Python	**set**, **frozenset**: > 10 methods and operators	😻
Ruby	**Set**: > 10 methods and operators	😻
Elixir	**MapSet**: > 10 methods	😻
.Net (C# etc.)	**ISet** interface: > 10 methods; 2 implementations	😻
JavaScript (ES6)	**Set**: < 10 methods	😿
Java	**Set** interface: < 10 methods; 8 implementations	😿
Go	Do it yourself, or pick one of several packages…	😾

FULL SET OPERATORS IN PYTHON

```
In [8]:  1 f & p
Out[8]: {2, 3, 5}
```

Intersection

```
In [9]:  1 f | p
Out[9]: {0, 1, 2, 3, 5, 7, 8}
```

Union

```
In [10]:  1 f ^ p
Out[10]: {0, 1, 7, 8}
```

Symmetric difference (a.k.a. XOR)

```
In [11]:  1 f - p
Out[11]: {0, 1, 8}
```

Difference

```
In [12]:  1 p - f
Out[12]: {7}
```

USING INFIX OPERATORS: DE MORGAN'S LAW

```
In [17]:
1 e = {n for n in range(10) if n % 2 == 0}
2 e
```

```
Out[17]: {0, 2, 4, 6, 8}
```

```
In [18]:
1 p & e
```

```
Out[18]: {2}
```

```
In [19]:
1 f - (p & e)
```

```
Out[19]: {0, 1, 3, 5, 8}
```

```
In [20]:
1 f - (p & e) == (f - p) | (f - e)
```

```
Out[20]: True
```

SET OPERATORS AND METHODS (1)

Table 3-2. Mathematical set operations: these methods either produce a new set or update the target set in place, if it's mutable

Math symbol	Python operator	Method	Description
S∩Z	`s & z`	`s.__and__(z)`	Intersection of `s` and `z`
	`z & s`	`s.__rand__(z)`	Reversed `&` operator
		`s.intersection(it, ...)`	Intersection of `s` and all sets built from iterables `it`, etc.
	`s &= z`	`s.__iand__(z)`	`s` updated with intersection of `s` and `z`
		`s.intersection_update(it, ...)`	`s` updated with intersection of `s` and all sets built from iterables `it`, etc.
S∪Z	`s | z`	`s.__or__(z)`	Union of `s` and `z`
	`z | s`	`s.__ror__(z)`	Reversed `|`
		`s.union(it, ...)`	Union of `s` and all sets built from iterables `it`, etc.
	`s |= z`	`s.__ior__(z)`	`s` updated with union of `s` and `z`
		`s.update(it, ...)`	`s` updated with union of `s` and all sets built from iterables `it`, etc.

SET OPERATORS AND METHODS (2)

Differences:

S\Z	`s - z`	`s.__sub__(z)`	Relative complement or difference between `s` and `z`
	`z - s`	`s.__rsub__(z)`	Reversed `-` operator
		`s.difference(it, …)`	Difference between `s` and all sets built from iterables `it`, etc.
	`s -= z`	`s.__isub__(z)`	`s` updated with difference between `s` and `z`
		`s.difference_update(it, …)`	`s` updated with difference between `s` and all sets built from iterables `it`, etc.
		`s.symmetric_difference(it)`	Complement of `s & set(it)`
S Δ Z	`s ^ z`	`s.__xor__(z)`	Symmetric difference (the complement of the intersection `s & z`)
	`z ^ s`	`s.__rxor__(z)`	Reversed `^` operator
		`s.symmetric_difference_update(it, …)`	`s` updated with symmetric difference of `s` and all sets built from iterables `it`, etc.
	`s ^= z`	`s.__ixor__(z)`	`s` updated with symmetric difference of `s` and `z`

SET TESTS

All of these return a bool:

Table 3-3. Set comparison operators and methods that return a bool

Math symbol	Python operator	Method	Description
		`s.isdisjoint(z)`	`s` and `z` are disjoint (have no elements in common)
e∈S	`e in s`	`s.__contains__(e)`	Element `e` is a member of `s`
S⊆Z	`s <= z`	`s.__le__(z)`	`s` is a subset of the `z` set
		`s.issubset(it)`	`s` is a subset of the set built from the iterable `it`
S⊂Z	`s < z`	`s.__lt__(z)`	`s` is a proper subset of the `z` set
S⊇Z	`s >= z`	`s.__ge__(z)`	`s` is a superset of the `z` set
		`s.issuperset(it)`	`s` is a superset of the set built from the iterable `it`
S⊃Z	`s > z`	`s.__gt__(z)`	`s` is a proper superset of the `z` set

ThoughtWorks®

CODE: UINTSET CLASS

UINTSET: A SET CLASS FOR NON-NEGATIVE INTEGERS

Inspired by the **intset** example in chapter 6 of *The Go Programming Language* **by A. Donovan and B. Kernighan**

An empty set is represented by zero.

A set of integers {a, b, c} is represented by on bits in an integer at offsets a, b, and c.

Source code:

https://github.com/standupdev/uintset

REPRESENTING SETS OF INTEGERS AS BIT PATTERNS

This set:

```
UintSet({13, 14, 22, 28, 38, 53, 64, 76, 94, 102, 107, 121, 136, 143, 150, 157,
169, 173, 187, 201, 213, 216, 234, 247, 257, 268, 283, 288, 290})
```

REPRESENTING SETS OF INTEGERS AS BIT PATTERNS

This set:

```
UintSet({13, 14, 22, 28, 38, 53, 64, 76, 94, 102, 107, 121, 136, 143, 150, 157,
169, 173, 187, 201, 213, 216, 234, 247, 257, 268, 283, 288, 290})
```

Is represented by this integer

```
250215800770294692189743128168123011668092585423464438593870336339645497189765
2283727872
```

REPRESENTING SETS OF INTEGERS AS BIT PATTERNS

This set:

```
UintSet({13, 14, 22, 28, 38, 53, 64, 76, 94, 102, 107, 121, 136, 143, 150, 157,
169, 173, 187, 201, 213, 216, 234, 247, 257, 268, 283, 288, 290})
```

Is represented by this integer

```
250215800770294692189743128168123011668092585423464438593870336339645497189765
283727872
```

Which has this bit pattern:

```
10100001000000000000010000000001000000000100000000000000000100100000000000010
0000000000100000000000001000000000000100010000000001000000100000010000001000
0000000000100000000000010000100000001000000000000000100000000000100000000000
1000000000000010000000010000010000000110000000000000
```

REPRESENTING SETS OF INTEGERS AS BIT PATTERNS

This set:

```
UintSet({290})
```

Is represented by this integer

```
1989292945639146568621528992587283360401824603189390869761855907572637988050133
502132224
```

Which has this bit pattern:

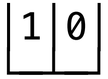
```
1000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
```

REPRESENTING SETS OF INTEGERS AS BIT PATTERNS (2)

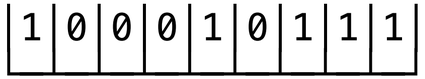
`UintSet() → 0`

`UintSet({0}) → 1`

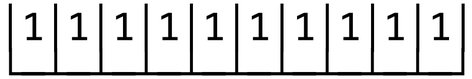
`UintSet({1}) → 2`



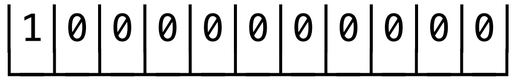
`UintSet({0, 1, 2, 4, 8}) → 279`



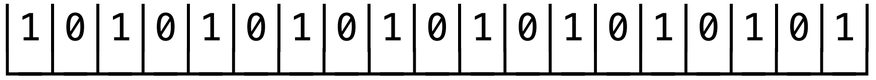
`UintSet({0, 1, 2, 3, 4, 5, 6, 7, 8, 9}) → 1023`



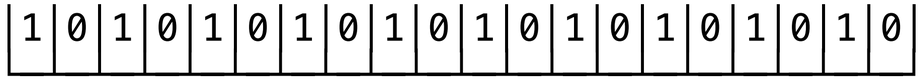
`UintSet({10}) → 1024`



`UintSet({0, 2, 4, 6, 8, 10, 12, 14, 16, 18}) → 349525`



`UintSet({1, 3, 5, 7, 9, 11, 13, 15, 17, 19}) → 699050`



SAMPLE METHOD: INTERSECTION OPERATOR &

```
76          def __and__(self, other):
77              cls = self.__class__
78              if isinstance(other, cls):
79                  res = cls()
80                  res._bits = self._bits & other._bits
81                  return res
82              return NotImplemented
```

https://github.com/standupdev/uintset

standupdev / uintset

Unwatch 1 | Star 14 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Security | Insights | Settings

A Python set type designed for dense sets of non-negative integers.

Edit

Manage topics

26 commits | 1 branch | 0 releases | 1 contributor | BSD-3-Clause

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

ramalho demo notebook and scripts — Latest commit e05b935 on May 3

demo	demo notebook and scripts	5 months ago
.gitignore	Refactored snippets to create short_long and trim.	last year
LICENSE	Initial commit	last year
README.md	new implementation of uintset using a (long) int as an array of bits	5 months ago
bitops.py	new implementation of uintset using a (long) int as an array of bits	5 months ago
test_bitops.py	new implementation of uintset using a (long) int as an array of bits	5 months ago
test_uintset.py	new implementation of uintset using a (long) int as an array of bits	5 months ago
uintset.py	new implementation of uintset using a (long) int as an array of bits	5 months ago

README.md

https://github.com/standupdev/uintset

THE BEAUTY OF DOUBLE DISPATCH

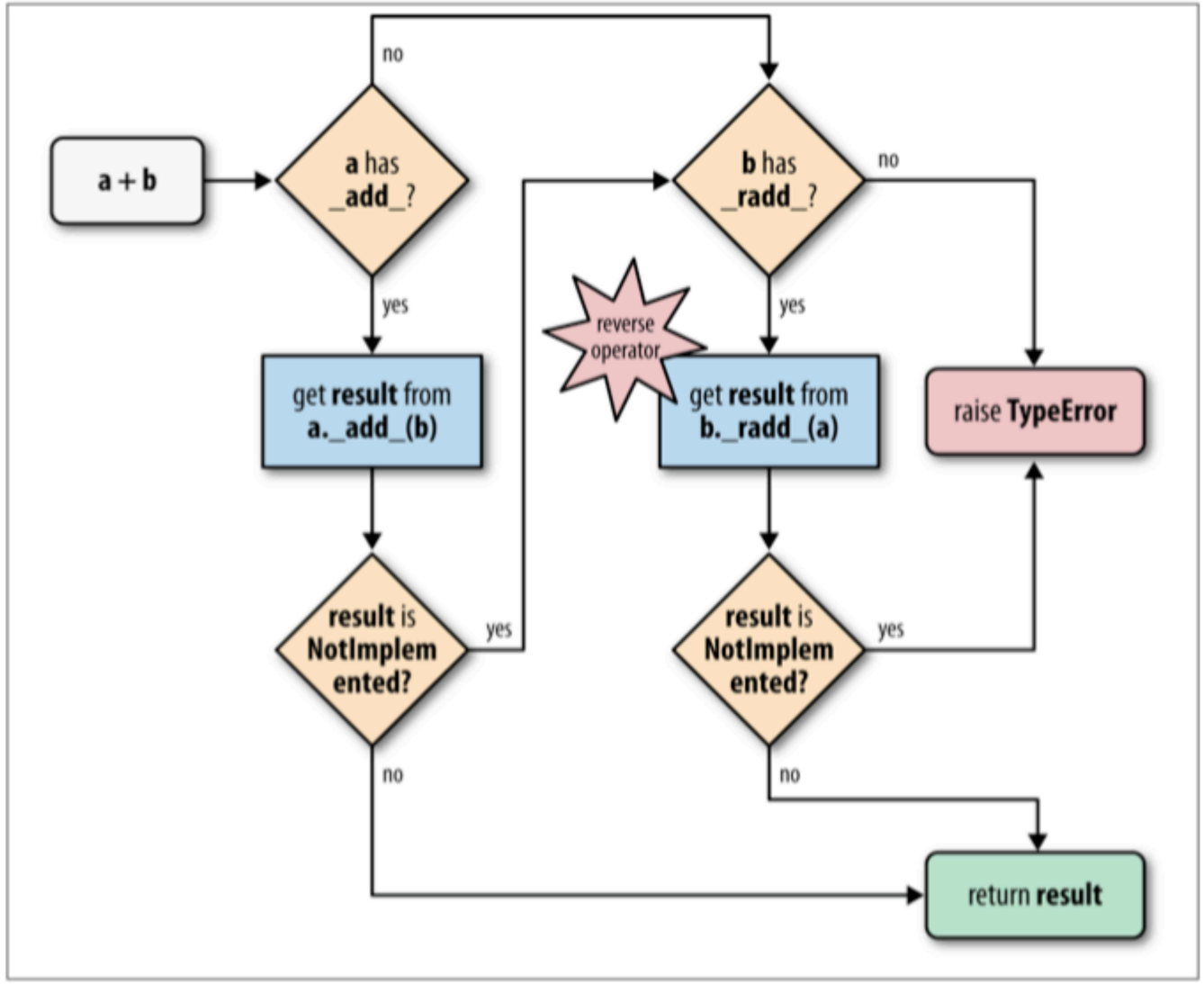


Figure 13-1. Flowchart for computing a + b with __add__ and __radd__

FAIL FAST

ThoughtWorks®

Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious w
Although that way may not be obvious at first unless you'
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea
If the implementation is easy to explain, it may be a goo
Namespaces are one honking great idea -- let's do more of

Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious w
Although that way may not be obvious at first unless you'
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea
If the implementation is easy to explain, it may be a goo
Namespaces are one honking great idea -- let's do more of

NO "UNDEFINED" VALUES

```
In [1]:  1 a = b + 8
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-f446e2577de9> in <module>
----> 1 a = b + 8

NameError: name 'b' is not defined
```

```
In [2]:  1 a = '上海'
         2 a[0]
```

```
Out[2]: '上'
```

```
In [3]:  1 a[3]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-3-f75b6be7d8e3> in <module>
----> 1 a[3]

IndexError: string index out of range
```

NO SURPRISING NULL VALUES FROM DICTS

```
In [5]:
1 countries = {'BR' : 'Brasil', 'CN' : '中国'}
2 countries['CN']
```

```
Out[5]: '中国'
```

```
In [6]:
1 countries['XY']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-6-3133f7e5f3af> in <module>
----> 1 countries['XY']

KeyError: 'XY'
```

Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious w
Although that way may not be obvious at first unless you'
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea
If the implementation is easy to explain, it may be a goo
Namespaces are one honking great idea -- let's do more of

NO SURPRISING NULL VALUES FROM DICTS

```
In [5]:
1 countries = {'BR' : 'Brasil', 'CN' : '中国'}
2 countries['CN']
```

```
Out[5]: '中国'
```

```
In [6]:
1 countries['XY']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-6-3133f7e5f3af> in <module>
----> 1 countries['XY']

KeyError: 'XY'
```

```
In [7]:
1 countries.get('XY', '<not found>')
```

```
Out[7]: '<not found>'
```

NO SURPRISING NULL VALUES WHEN UNPACKING

```
In [8]:  1  a, b, c = [10, 20, 30]
```

```
In [9]:  1  a, b, c = [10, 30, 30, 40, 50]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-2e6e70608432> in <module>
----> 1 a, b, c = [10, 30, 30, 40, 50]

ValueError: too many values to unpack (expected 3)
```

NO SURPRISING NULL VALUES WHEN UNPACKING

```
In [8]:  1 a, b, c = [10, 20, 30]
```

```
In [9]:  1 a, b, c = [10, 30, 30, 40, 50]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-2e6e70608432> in <module>
----> 1 a, b, c = [10, 30, 30, 40, 50]

ValueError: too many values to unpack (expected 3)
```

```
In [10]:  1 a, b, c, *rest = [10, 20, 30, 40, 50]
          2 a, b, c, rest
```

```
Out[10]: (10, 20, 30, [40, 50])
```

Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious w
Although that way may not be obvious at first unless you'
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea
If the implementation is easy to explain, it may be a goo
Namespaces are one honking great idea -- let's do more of

NO SURPRISING RESULTS FOR "2" + 8

```
In [4]: 1 a + 8
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-5589b03f6f40> in <module>
----> 1 a + 8

TypeError: can only concatenate str (not "int") to str
```

THE LAST PARAGRAPH OF FLUENT PYTHON

I haven't yet found a language that manages to be easy for beginners, practical for professionals, and exciting for hackers in the way that Python is. Thanks, Guido van Rossum and everybody else who makes it so.

COMMUNITY

ThoughtWorks®


CAME FOR THE
LANGUAGE
STAYED FOR THE
COMMUNITY
- Brett Cannon

PYTHON BRASIL 2018 BY JULIO MELANDA (INSTAGRAM @JULIOMELANDA)

PYCON KOREA 2018

Committing to **diversity** and **inclusion**

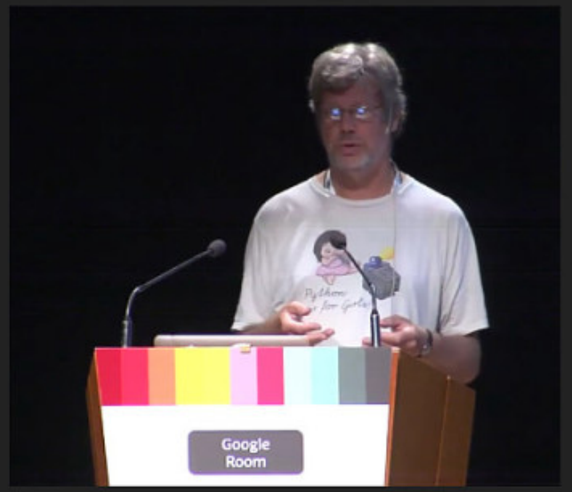
다양성과 포용에 기여하기

Sujin Lee / 이수진
@sujinleeme

Guido loves to wear a “Python is for Girls.” T-shirt.

2013

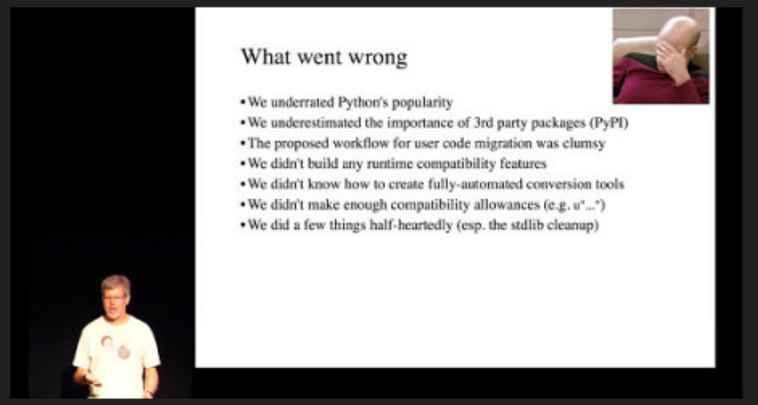
2014



2015

2016

2017



2018

[11]

[10]

During PyCon US 2014 keynote,

Guido answered questions only from women.

"Through out the conference, I've been attacked by by people with questions, and they were almost all men, so **I think the women [...] are a little behind and they can catch up here,**"

[10] Why Guido van Rossum supports the Ada Initiative, wears a "Python is for girls" shirt, and answered questions from only women at PyCon 2014 | adainitiative.org
https://adainitiative.org/2014/09/22/why-guido-van-rossum-supports-the-ada-initiative-wears-a-python-is-for-girls-shirt-and-answered-questions-from-only-women-at-pycon-2014

[11] Guido's Talk at PyConUS 2014 https://www.youtube.com/watch?v=G-uKNd5TSBw&feature=youtu.be&t=13m40s

@mariatta [15]

Finally! In Jan, 2017,

Mariatta Wijaya joins a Python core dev team.
She is the first female a Python core developer.

[15] Mariatta's profile http://pycaribbean.com/speaker/mariatta-wijaya.html

PYTHON BRASIL 2019

Ribeirão Preto
23 a 28 de outubro

adquira seu ingresso

O Evento

A **PythonBrasil** é o maior evento sobre linguagem de programação Python do Brasil. Feito pela comunidade para a comunidade, tem o objetivo de difundir a linguagem, promover a troca de experiências e manter a comunidade crescendo

veja o local

Keynotes

Carol Willing

Diretora da PSF e Projeto Jupyter, desenvolvedora core do CPython e JupyterHub

Carol Naslund Willing atua no Conselho Diretor do Projeto Jupyter e trabalha como Core Developer no JupyterHub e mybinder.org. Ela também atua como coeditora do “The Journal of Open Source Education” (JOSE) e é coautora de um livro open source, o “Teaching and Learning with Jupyter”.
Ela é membro do primeiro Conselho Diretor do Python e faz parte da equipe principal de desenvolvimento do CPython. Ela também é ex-diretora da Python Software Foundation e atual Fellow Member. Em 2019, recebeu o Prêmio Frank Willison por suas contribuições técnicas e para com a comunidade Python. Fortemente comprometida com a divulgação da comunidade, Carol coorganiza o PyLadies San Diego e o San Diego Python User Group.
Carol tem mestrado em Gestão pelo MIT e Bacharel em Engenharia Elétrica pela Duke University.

Débora Azevedo

Professora na Secretaria Estadual de Educação e Cultura do Rio Grande do Norte

Débora é professora na rede estadual de ensino no Rio Grande do Norte, e atualmente aluna do Mestrado em Inovação com Tecnologias Educacionais no Instituto Metrópole Digital (UFRN). Com uma formação mista, graduação em Letras - Inglês e bacharelado em TI ainda no meio do caminho, é cofundadora do PyLadies Brasil, estando em sua formação inicial até agora, além de organizadora do Django Girls Natal, sua cidade. Ano passado organizou a primeira PyLadiesBR Conf, primeira conferência do Brasil de PyLadies para PyLadies, e tem atuado ultimamente na comunidade de mulheres Python a nível nacional. Acredita que é com a educação que conseguiremos mudar alguma coisa, em nós, na comunidade, no país. Fã de Harry Potter e de literatura policial, pode chamá-la para conversar sobre isso ou sobre algum livro da Agatha Christie.

Débora Azevedo

Professora na Secretaria Estadual de Educação e Cultura do Rio Grande do Norte

Débora é professora na rede estadual de ensino no Rio Grande do Norte, e atualmente aluna do Mestrado em Inovação com Tecnologias Educacionais no Instituto Metrópole Digital (UFRN). Com uma formação mista, graduação em Letras - Inglês e bacharelado em TI ainda no meio do caminho, é cofundadora do PyLadies Brasil, estando em sua formação inicial até agora, além de organizadora do Django Girls Natal, sua cidade. Ano passado organizou a primeira PyLadiesBR Conf, primeira conferência do Brasil de PyLadies para PyLadies, e tem atuado ultimamente na comunidade de mulheres Python a nível nacional. Acredita que é com a educação que conseguiremos mudar alguma coisa, em nós, na comunidade, no país. Fã de Harry Potter e de literatura policial, pode chamá-la para conversar sobre isso ou sobre algum livro da Agatha Christie.

Felipe de Morais

Comunidade AfroPython

Felipe de Morais é um Desenvolvedor de Software envolvido há alguns anos com a comunidade Python. Entre as contribuições mais relevantes para a Comunidade estão o AfroPython, iniciativa para empoderar pessoas negras através da tecnologia e colaborações em projetos abertos como a Operação Serenata de Amor. Tudo que ele faz são manifestações de seus 2 valores mais básicos: Ajudar pessoas e aprender com e de pessoas incríveis.

Laís Varejão

Desenvolvedora Full-Stack e Gerente de Projetos @ Vinta Software

É desenvolvedora full-stack e gerente de projetos na Vinta Software. PyLady e organizadora do Django Girls Recife, vislumbra um futuro com mais mulheres na computação. Alumna do Recurse Center, um retiro para programadores em Nova York. É engenheira de software por formação, tem 7 anos de experiência no mercado e atualmente lidera uma equipe de desenvolvimento onde aplica metodologias ágeis.

Laís Varejão

Desenvolvedora Full-Stack e Gerente de Projetos @ Vinta Software

É desenvolvedora full-stack e gerente de projetos na Vinta Software. PyLady e organizadora do Django Girls Recife, vislumbra um futuro com mais mulheres na computação. Alumna do Recurse Center, um retiro para programadores em Nova York. É engenheira de software por formação, tem 7 anos de experiência no mercado e atualmente lidera uma equipe de desenvolvimento onde aplica metodologias ágeis.

Lorena Mesa

Engenheira de Dados @ Github

Cientista política que virou programadora, Lorena Mesa é engenheira de dados na equipe de sistemas de inteligência de software do GitHub, diretora da Python Software Foundation e coorganizadora do PyLadies Chicago. Durante o seu período na campanha "Obama for America" e, em seguida, na pesquisa de pós-graduação, Lorena teve que aprender como transformar dados confusos e incompletos em análises inteligíveis, como por exemplo a previsão do comportamento dos eleitores latinos. Foi esse histórico em pesquisa e matemática aplicada que levou Lorena a seguir uma carreira em engenharia e ciência de dados. Uma parte ativista, uma parte fanática por Star Wars e outra Trekkie, Lorena segue o lema "live long and prosper".

Luciano Ramalho

Autor do livro Python Fluente e Consultor Principal na ThoughtWorks Brasil

Depois de 17 anos programando em Python, Ramalho publicou o livro Fluent Python (O'Reilly, 2015), lançado em 9 idiomas, inclusive PT-BR (Python Fluente, Novatec, 2015). Foi o primeiro presidente da Associação Python Brasil, e co-fundador do Garoa Hacker Clube. É fellow da Python Software Foundation e consultor principal na ThoughtWorks Brasil.

ELEGANCE

ThoughtWorks®

FIRST QUOTATION IN CHAPTER 1 OF FLUENT PYTHON

> Guido's sense of the aesthetics of language design is amazing. I've met many fine language designers who could build theoretically beautiful languages that no one would ever use, but Guido is one of those rare people who can build a language that is just slightly less theoretically beautiful but thereby is a joy to write programs in.[1]
>
> —Jim Hugunin, *Creator of Jython, cocreator of AspectJ, architect of the .Net DLR*

1 Story of Jython, written as a Foreword to *Jython Essentials* (O'Reilly, 2002), by Samuele Pedroni and Noel Rappin.

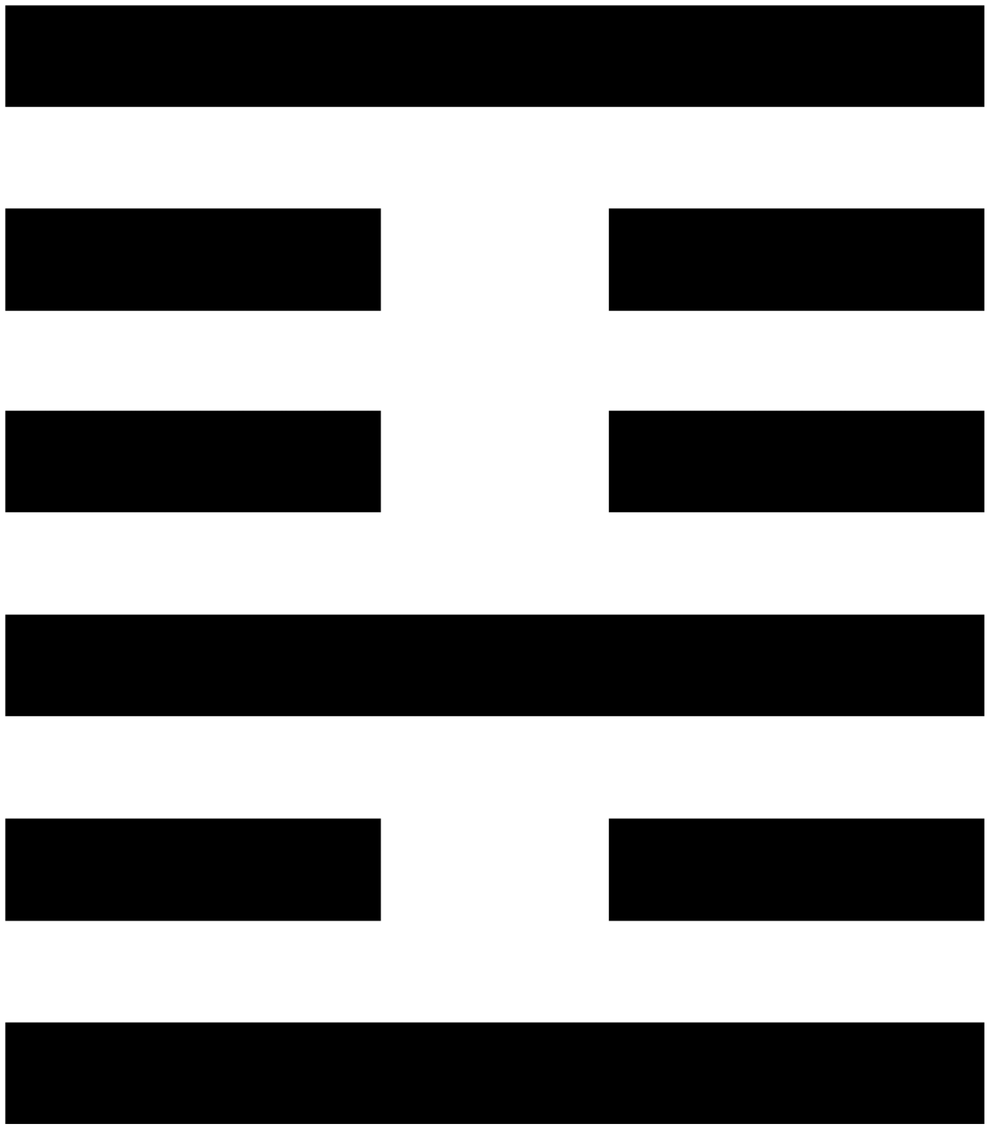
BRUCE ECKEL

Author of:

- On Java 8
- Thinking in Java
- Atomic Scala
- Atomic Kotlin
- and many other books.

Loves Python 😍.

Designed t-shirt for PyCon US 2009.



易經

Yìjīng

周易/賁

< 周易

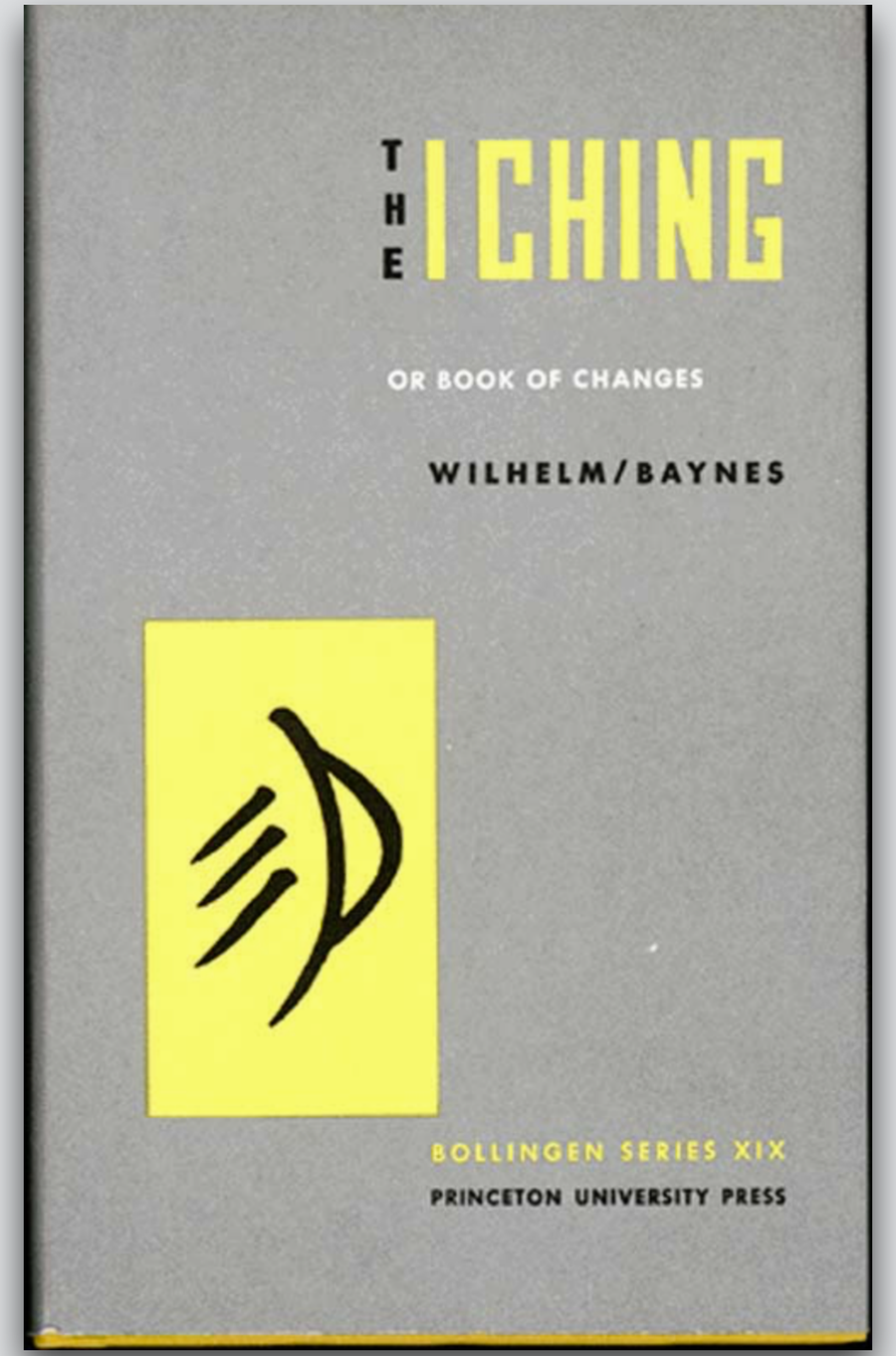
噬嗑 ◄ 周易 賁 ► 剝

周易　第二十二卦

賁

䷕ 離下艮上

- **易經：**
  - **賁**：亨。小利有攸往。
    1. 初九：賁其趾，舍車而徒。
    2. 六二：賁其須。
    3. 九三：賁如濡如，永貞吉。
    4. 六四：賁如皤如，白馬翰如，匪寇婚媾。
    5. 六五：賁於丘園，束帛戔戔，吝，終吉。
    6. 上九：白賁，无咎。
- **彖曰：**



The I Ching or Book of Changes

The Richard Wilhelm Translation rendered into English by Cary F. Baynes

Foreword by C. G. Jung

Preface to the Third Edition by Hellmut Wilhelm

Bollingen Series XIX

Princeton University Press

賁

22. Pi / Grace

above KÊN KEEPING STILL, MOUNTAIN
below LI THE CLINGING, FIRE

This hexagram shows a fire that breaks out of the secret depths of the earth and, blazing up, illuminates and beautifies the mountain, the heavenly heights. Grace—beauty of form—is necessary in any union if it is to be well ordered and pleasing rather than disordered and chaotic.

THE JUDGMENT

GRACE has success.
In small matters
It is favorable to undertake something.

Grace brings success. However, it is not the essential or fundamental thing; it is only the ornament and must therefore be used sparingly and only in little things. In the lower trigram of fire a yielding line comes between two strong lines and makes them beautiful, but the strong lines are the essential content and the weak line is the beautifying form. In the upper trigram of the mountain, the strong line takes the lead, so that here again the strong element must be regarded as the decisive factor. In nature we see in the sky the strong light of the sun; the life of the world depends on it. But this strong, essential thing is changed and given pleasing variety by the moon and the stars. In human affairs, aesthetic form comes into being when traditions exist that, strong and abiding like mountains, are made pleasing by a lucid beauty. By contemplating the forms existing in the heavens we come to understand time and its changing demands. Through contemplation of the forms existing in human society it becomes possible to shape the world.[1]

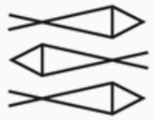
THE IMAGE

Fire at the foot of the mountain:
The image of GRACE.
Thus does the superior man proceed
When clearing up current affairs.
But he dare not decide controversial issues in this way.

I CHING

THE BOOK OF CHANGE

TRANSLATED BY
DAVID HINTON



22

ELEGANCE

Moving with the beauty of cowrie shells, elegance penetrates everywhere. And so, setting out toward a destination brings forth very little bounty.

PRESENTATION

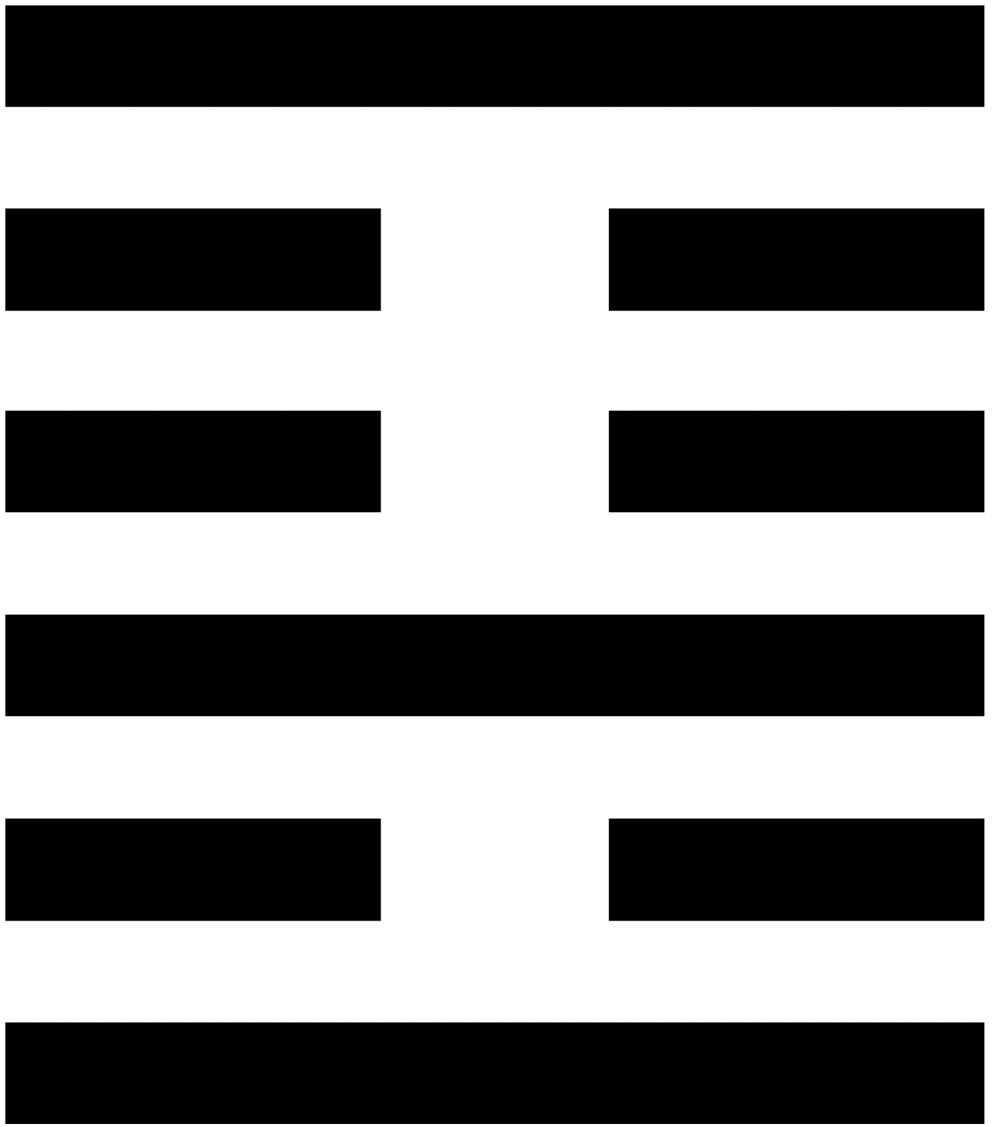
Elegance penetrates everywhere. Coming all tender assent moving with the beauty of cowrie shells, you come steely as a mountain in cloud moving deep into the grain of things: and so, you penetrate everywhere.

Rising steely as a mountain in cloud moving with the beauty of cowrie shells, you rise all tender assent moving deep into the grain of things: and so, *setting out toward a destination brings forth very little bounty*.

It's the deep grain of heaven. To illuminate heaven's deep grain and abide there, that is the deep grain of people.

If you can see into the deep grain of heaven with a heron's-eye gaze, you can fathom the seasons and their transformations. If you can see into the deep grain of humankind with a heron's-eye gaze, you can transform and perfect all beneath heaven.

Elegance Begets Simplicity



```
import sys

for arg in sys.argv:
    print arg
```

谢谢

e-mail: luciano.ramalho@thoughtworks.com

Twitter: @ramalhoorg

WeChat: Luciano_Ramalho

ThoughtWorks®

pycon china

PyConChina2019

中国 PYTHON 开发者大会

主办方 | 钻石赞助 | 白金赞助 | 黄金赞助 | 黄金赞助 | 白银赞助 | 白银赞助 | 会议同传支持

特别支持 | 图书支持 | 战略合作伙伴 | 战略合作伙伴 | 战略合作伙伴 | 战略合作伙伴 | 战略合作伙伴 | 社区合作伙伴 | 社区合作伙伴