Core Python: Implementing Collections

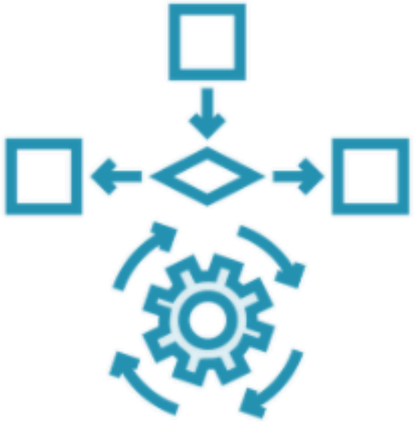
ITERATORS AND ITERABLES

Robert Smallshire
COFOUNDER - SIXTY NORTH
@robsmallshire

Austin Bingham
COFOUNDER - SIXTY NORTH
@austin_bingham

Iterators Decouple Retrieval from Structure

Collection



Generator

File

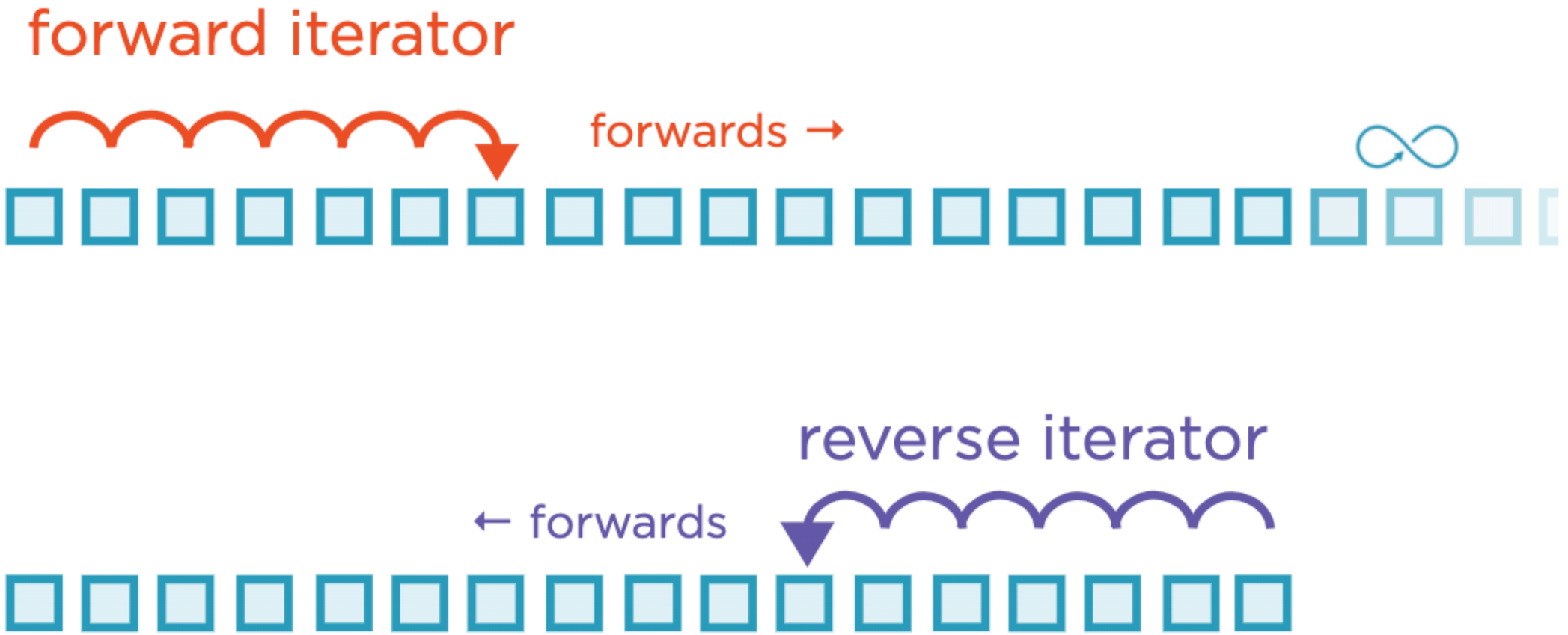
Sensor

The Iterator Pattern

Iterator

Iterable

Traversal Order



Obtaining and Advancing an Iterator

```
iterator = iter(iterable)


try:
    item = next(iterator)
    print(item)
except StopIteration:
    print("No more items")
```

Review: Creating Iterables

```
iterable_list = [2, 4, 6, 8, 10]
iterable_tuple = ("orange", "apple", "banana")
iterable_dict = dict(a="alpha", b="bravo", c="charlie")

def iterable_oceans():
    yield "Arctic"
    yield "Atlantic"
    yield "Indian"
    yield "Pacific"
    yield "Southern"

iterable_squares = (x*x for x in range(10))
```

Core Python: Getting Started

on

PLURALSIGHT

Naming Special Functions

__feature__

dunder

Our way of pronouncing special names

A portmanteau of ‘double underscore’

Instead of `__next__` we'll say “dunder next”

Iteration Protocols

Iterable

```
class MyIterable:

    ...

    def __iter__(self):
        # Must return a new iterator
        # for this iterable
        iterator = MyIterator(self)
        return iterator

iterator = iter(iterable)

item = next(iterator)
```

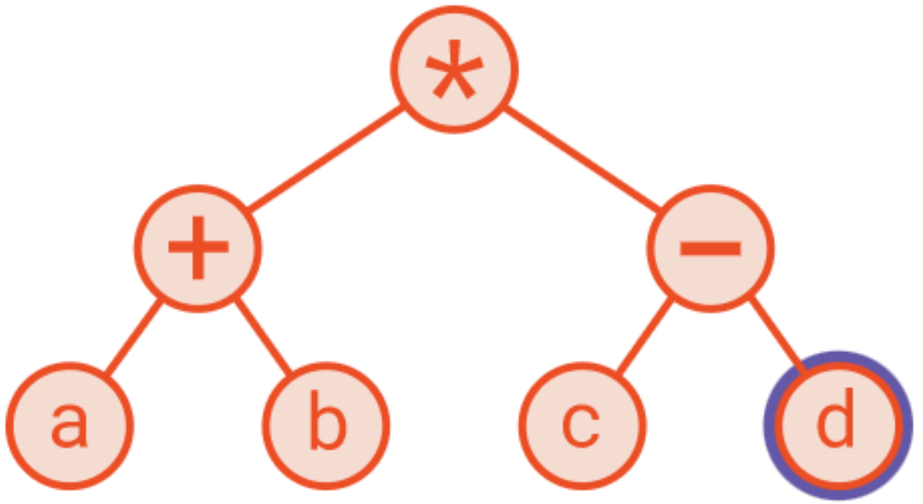
Iterator

```
class MyIterator:

    ...

    def __iter__(self):
        # All iterators are also iterables
        return self

    def __next__(self):
        if not has_more_items():
            raise StopIteration
        item = get_the_next_item()
        return item
```

Motivating Iterators: Tree Traversals

infix notation `(a + b) * (c - d)`

binary tree



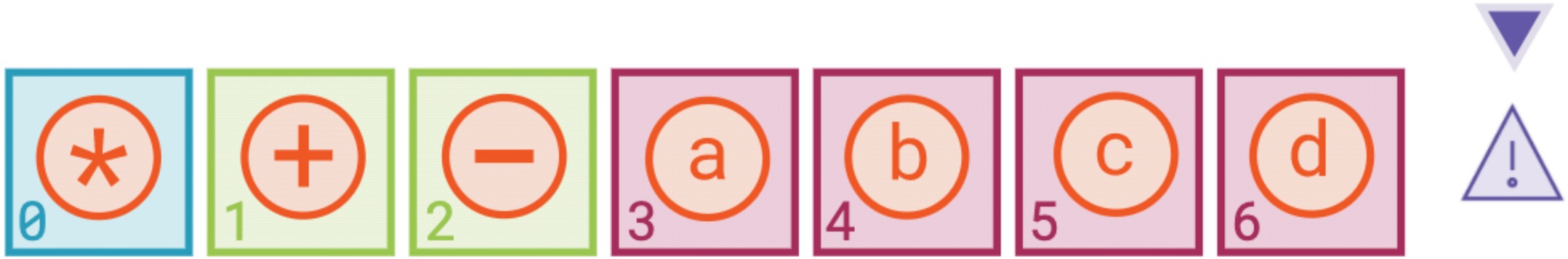
sequence


```
expr_tree = ["*", "+", "-", "a", "b", "c", "d"]
```

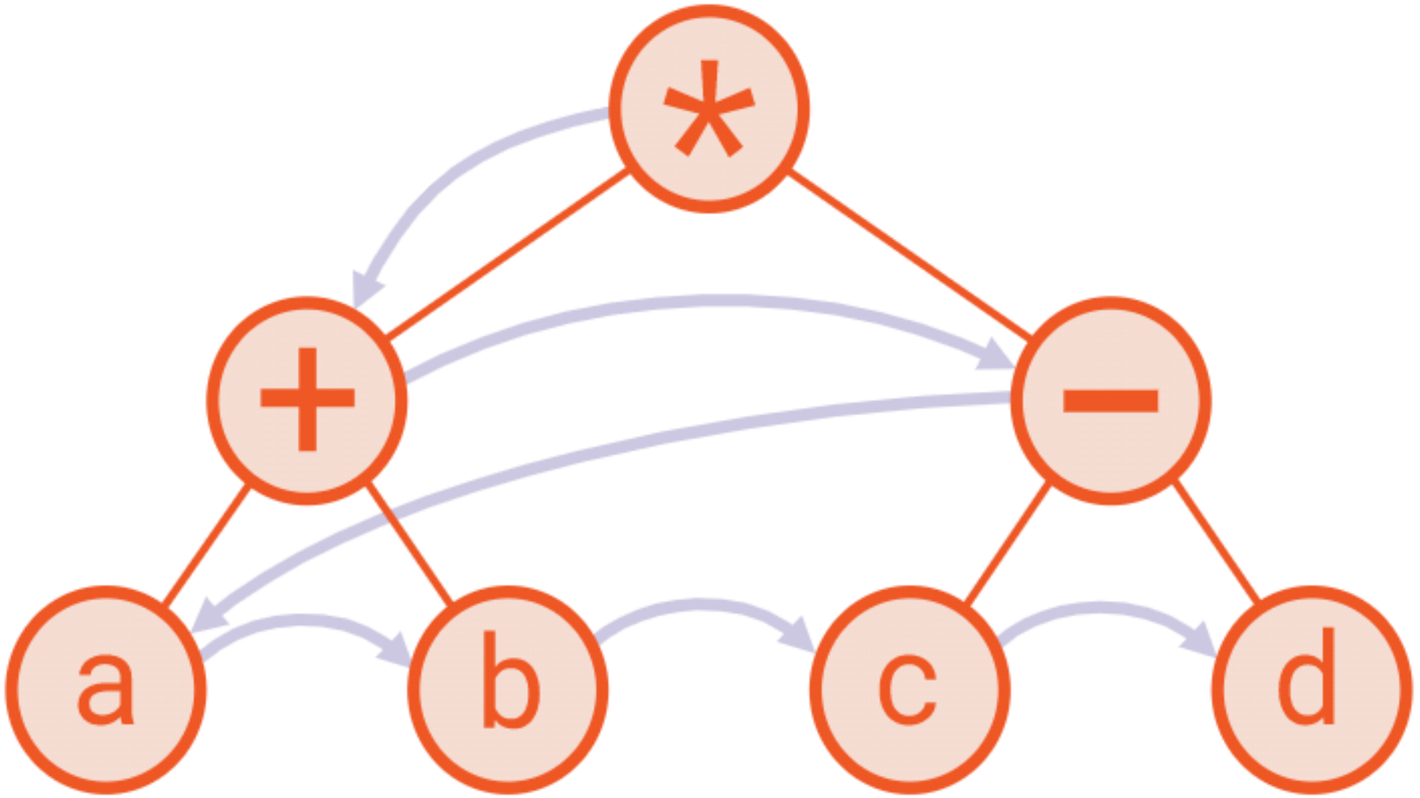
Sequence Iteration

```
>>> iterator = iter(expr_tree)
>>> for item in iterator:
...     print(item)
...
*
+
-
a
b
c
d
>>>
```

Storage-order Iterator for a Sequence

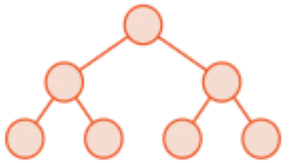
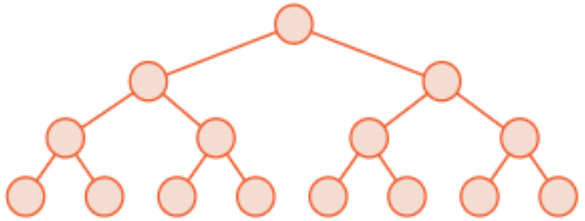
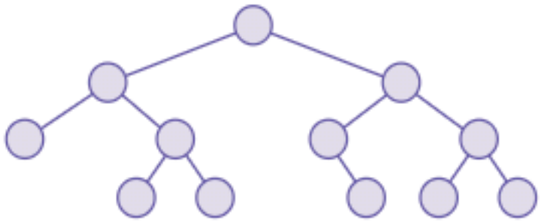
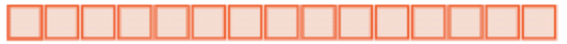


Breadth-first, Level-order Traversal

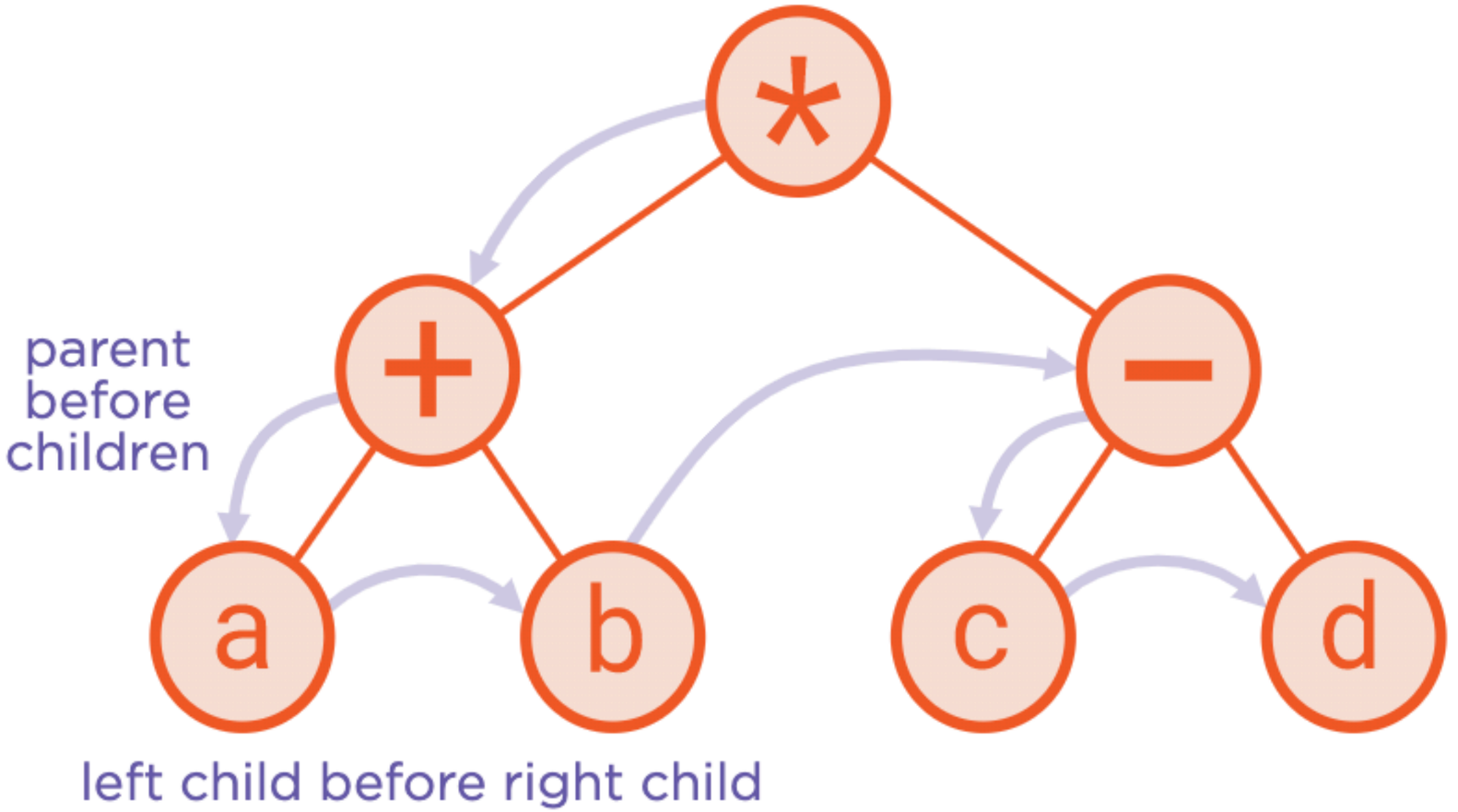



tree-iterators
iterators.py
iterators
iterators.py
6
7
8 class LevelOrderIterator:
9
10 def __init__(self, sequence):
11 if not _is_perfect_length(sequence):
12 raise ValueError(
13 f"Sequence of length {len(sequence)} does not represent "
14 "a perfect binary tree with length 2ⁿ - 1"
15)
LevelOrderIterator › __init__() › if not _is_perfect_length(seque...
Python Console
>>> from iterators import *
>>> non_tree = "+ 24 12 -".split()
>>> iterator = LevelOrderIterator(non_tree)
Traceback (most recent call last):
File "<input>", line 1, in <module>
File "/var/folders/0k/58g36_tx22xcxqd9mwqzg_h00000gp/T/tmpxqa2bgg1/build/tree-iterators/iterators.py", line 12, in __init__
raise ValueError(
ValueError: Sequence of length 4 does not represent a perfect binary tree with length 2ⁿ - 1
>>>
1:1 LF UTF-8 4 spaces Python 3.8 (tree-iterators)

Perfect Binary Trees

1	3	7	15	12
$2^1 - 1$	$2^2 - 1$	$2^3 - 1$	$2^4 - 1$	$2^4 - 3$
				
perfect	perfect	perfect	perfect	imperfect
				

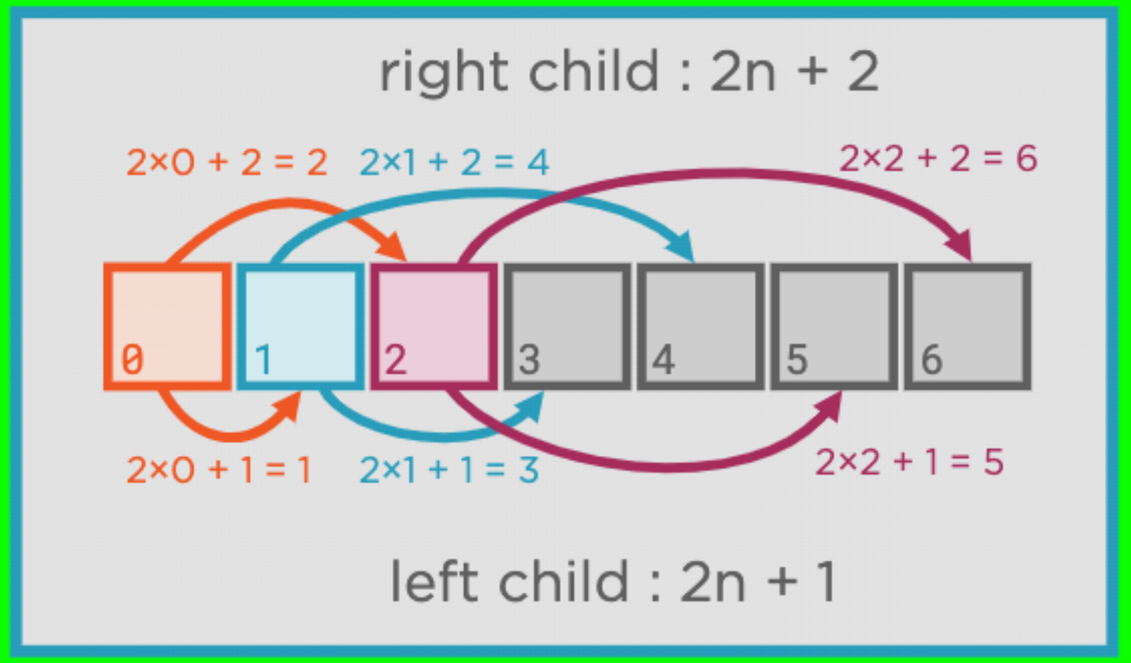
Depth-first, Pre-order Traversal



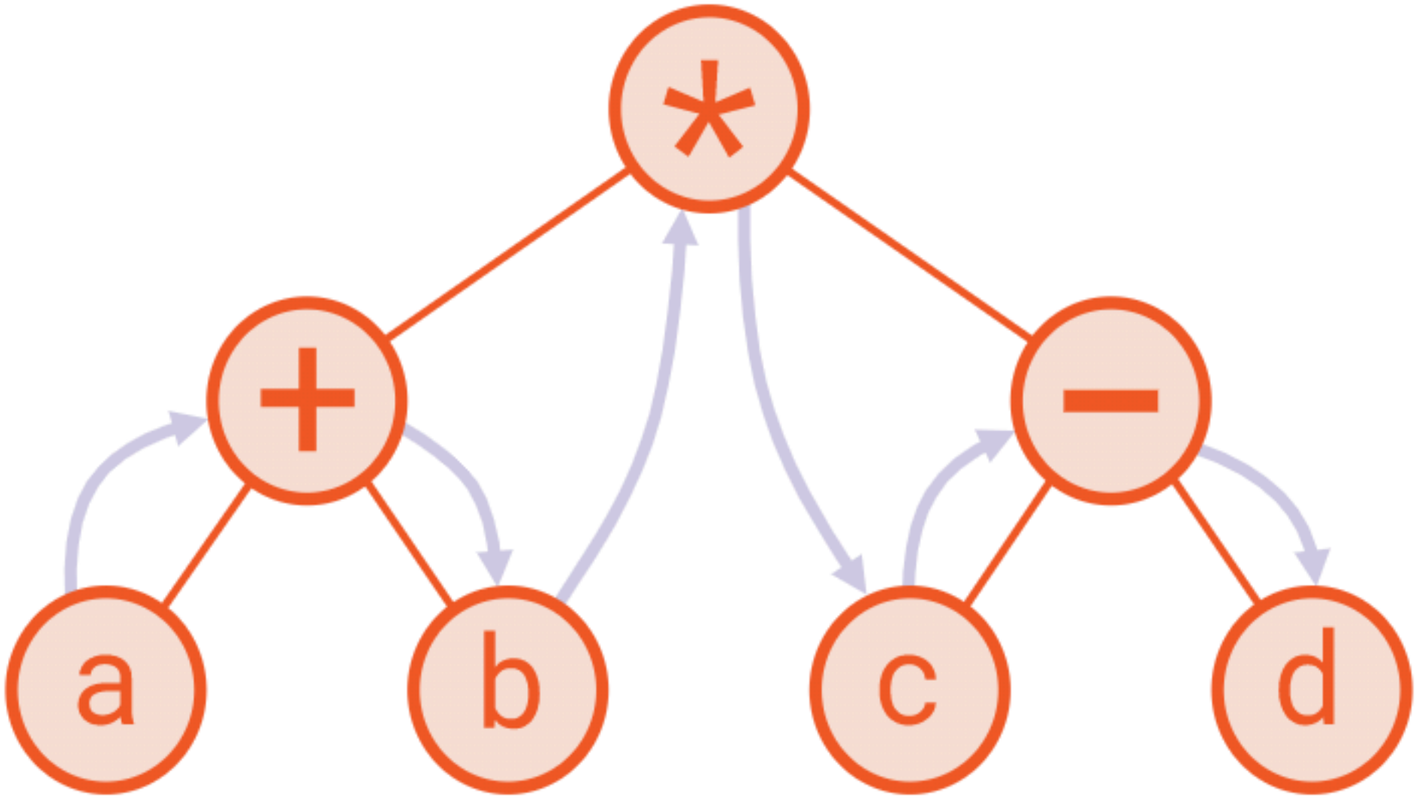
```
38  class PreOrderIterator:
39
40      def __init__(self, sequence):
41          if not _is_perfect_length(sequence):
42              raise ValueError(
43                  f"Sequence of length {len(sequence)} does not represent "
44                  "a perfect binary tree with length 2ⁿ - 1"
45              )
46          self._sequence = sequence
47          self._stack = [0]
```

Python Console

```
import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['/var/folders/0k/58g36_tx22xcxqd9mwqzg_h00000gp/T/tmpcqi2sl43/build/tree-iterators'])

Python Console
>>> from iterators import *
>>> expr_tree  = "* + - a b c d".split()
>>> iterator = PreOrderIterator(expr_tree)
>>> " ".join(iterator)
'* + a b - c d'

>>>
```


right child : 2n + 2
2×0 + 2 = 2
2×1 + 2 = 4
2×2 + 2 = 6
0
1
2
3
4
5
6
2×0 + 1 = 1
2×1 + 1 = 3
2×2 + 1 = 5
left child : 2n + 1

Depth-first, in-order Traversal



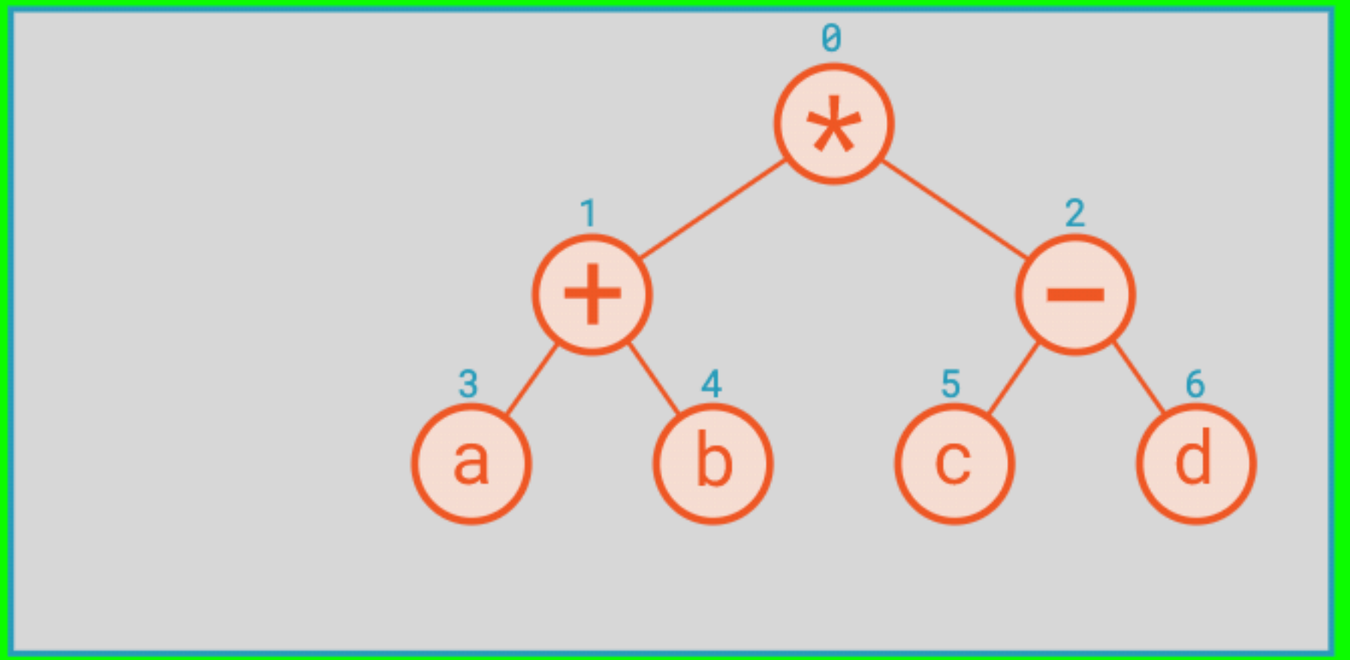
left subtree before current node before right subtree

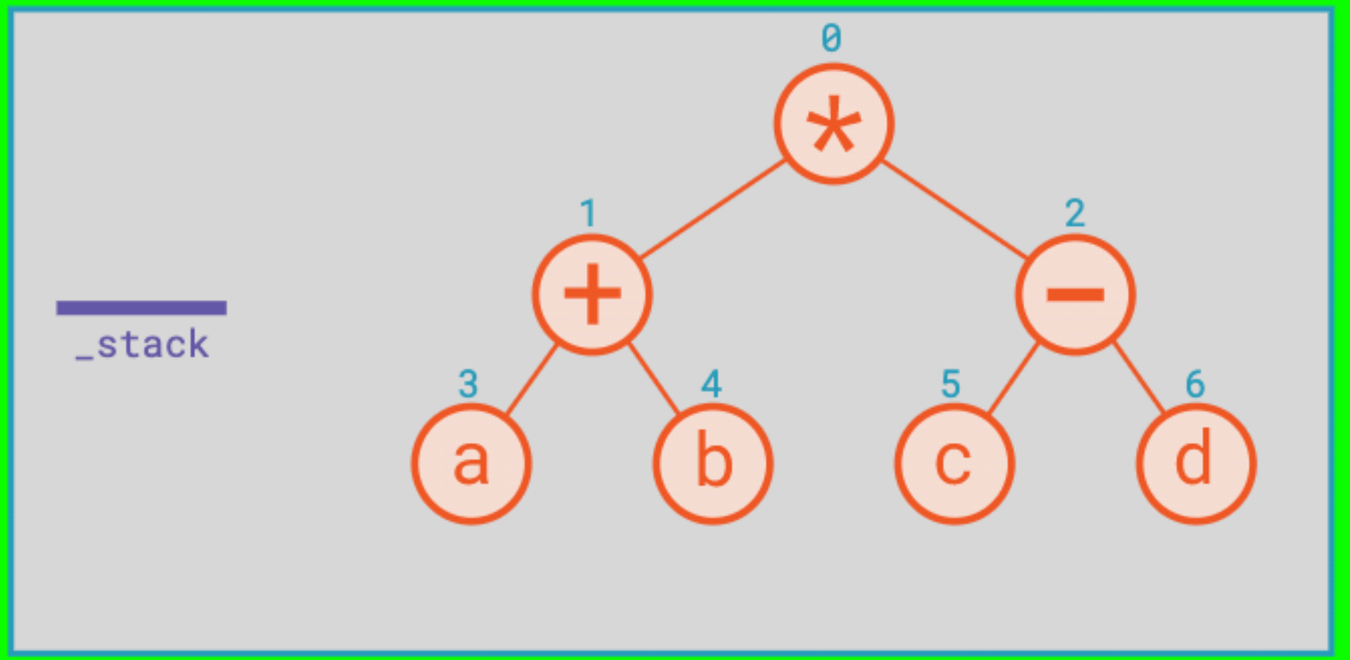
```
85      def __next__(self):
86          if (len(self._stack) == 0) and (self._index >= len(self._sequence)):
87              raise StopIteration
88
89          # Push left children onto the stack while possible
90          while self._index < len(self._sequence):
91              self._stack.append(self._index)
92              self._index = _left_child(self._index)
93
94          # Pop from stack and process, before moving to the right child
```

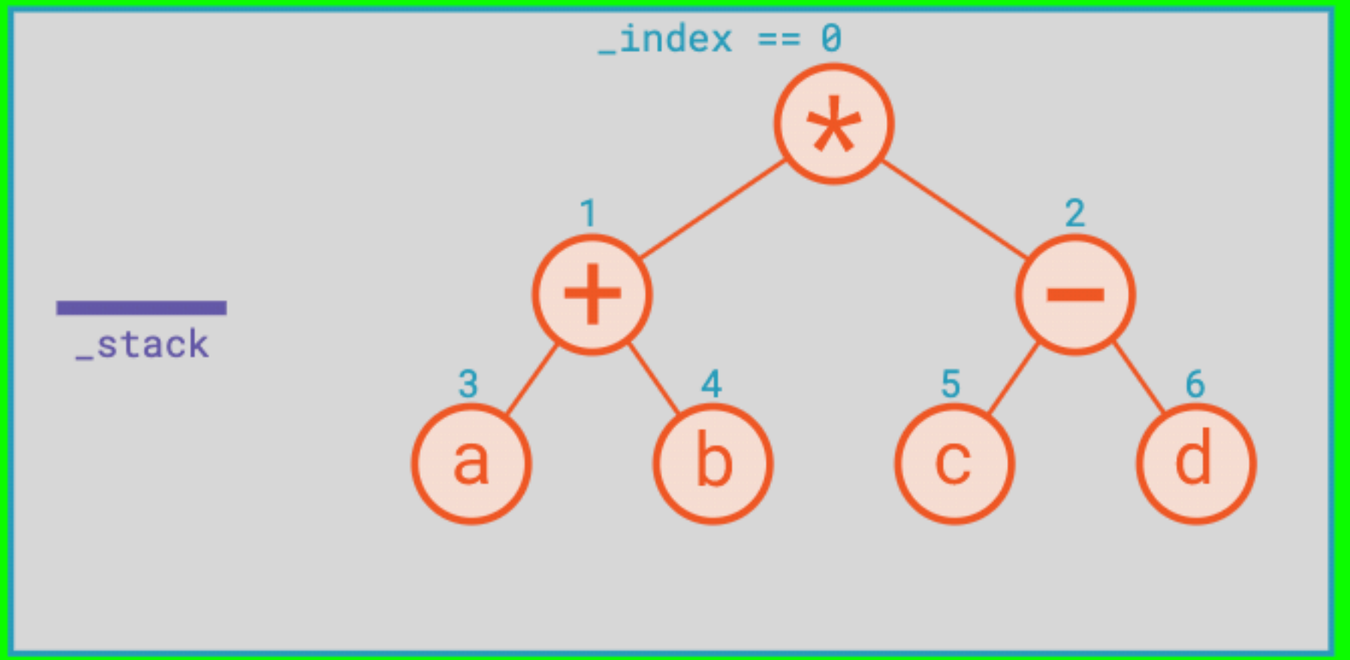
_is_perfect_length()

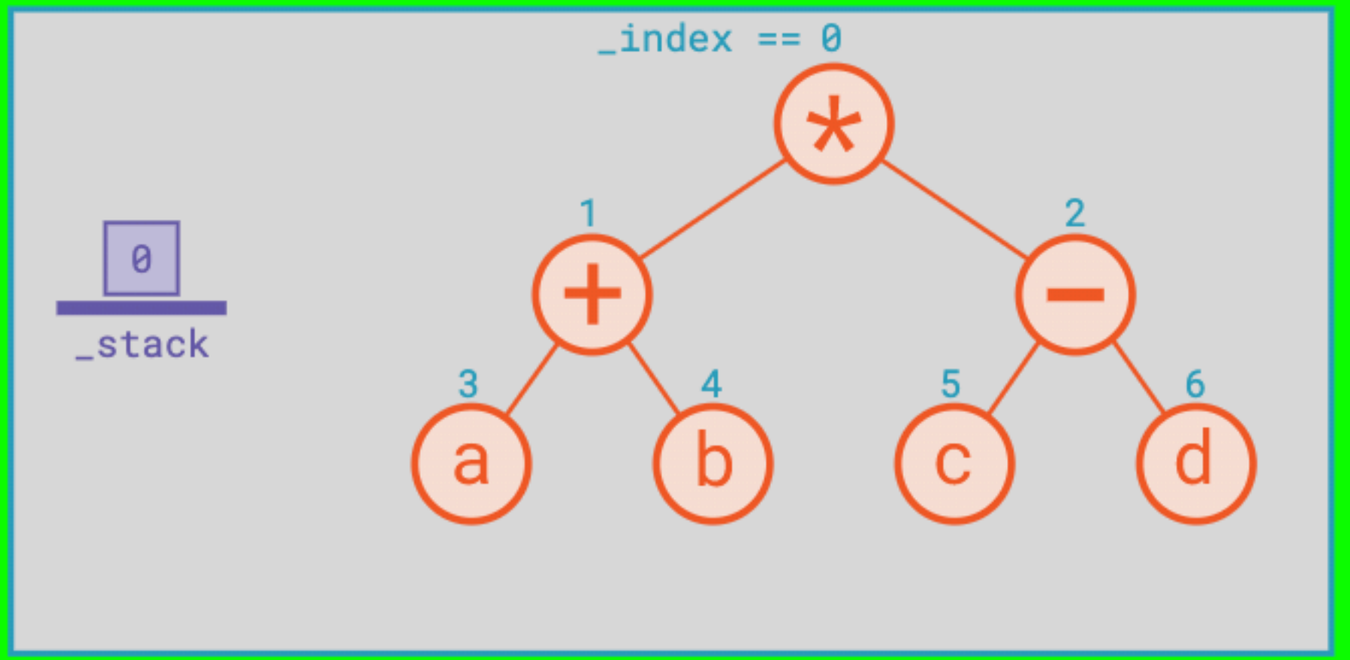
Python Console

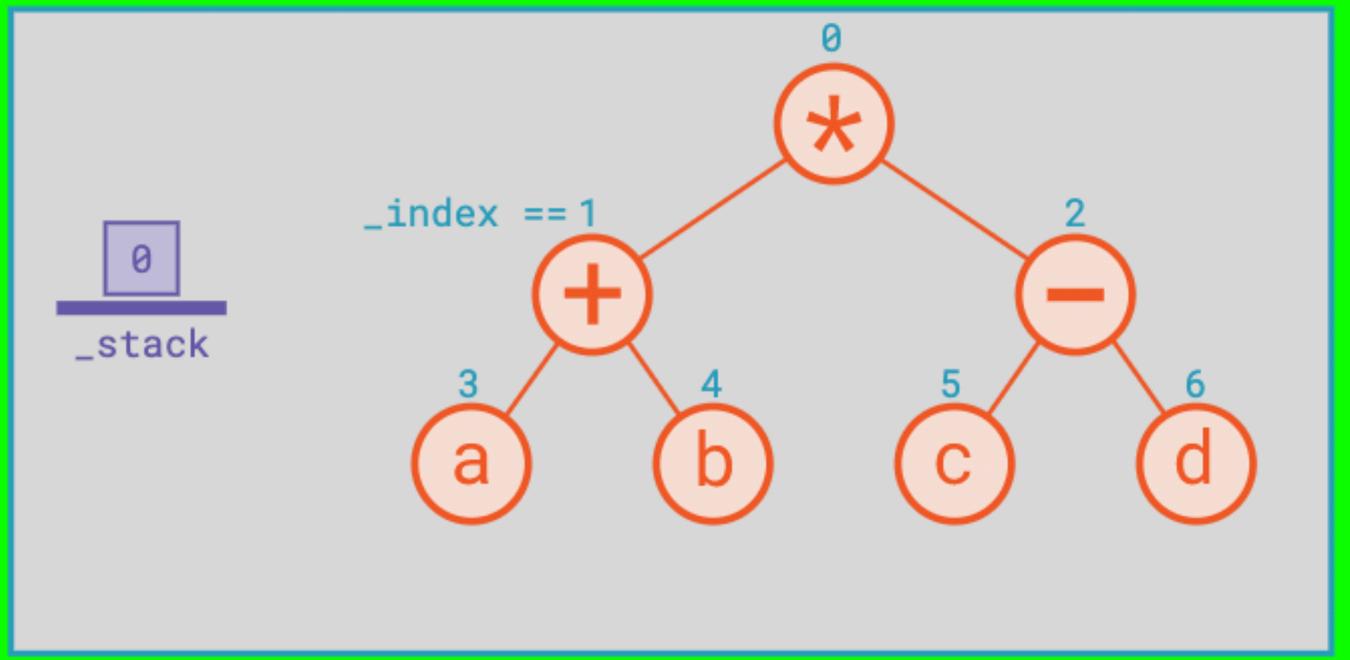
```
import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['/var/folders/0k/58g36_tx22xcxqd9mwqzg_h00000gp/T/tmpgfwut6bj/build/tree-iterators'])

Python Console
>>> from iterators import *
>>> expr_tree  = "* + - a b c d".split()
>>> iterator = InOrderIterator(expr_tree)
>>> " ".join(iterator)
'a + b * c - d'

>>>
```

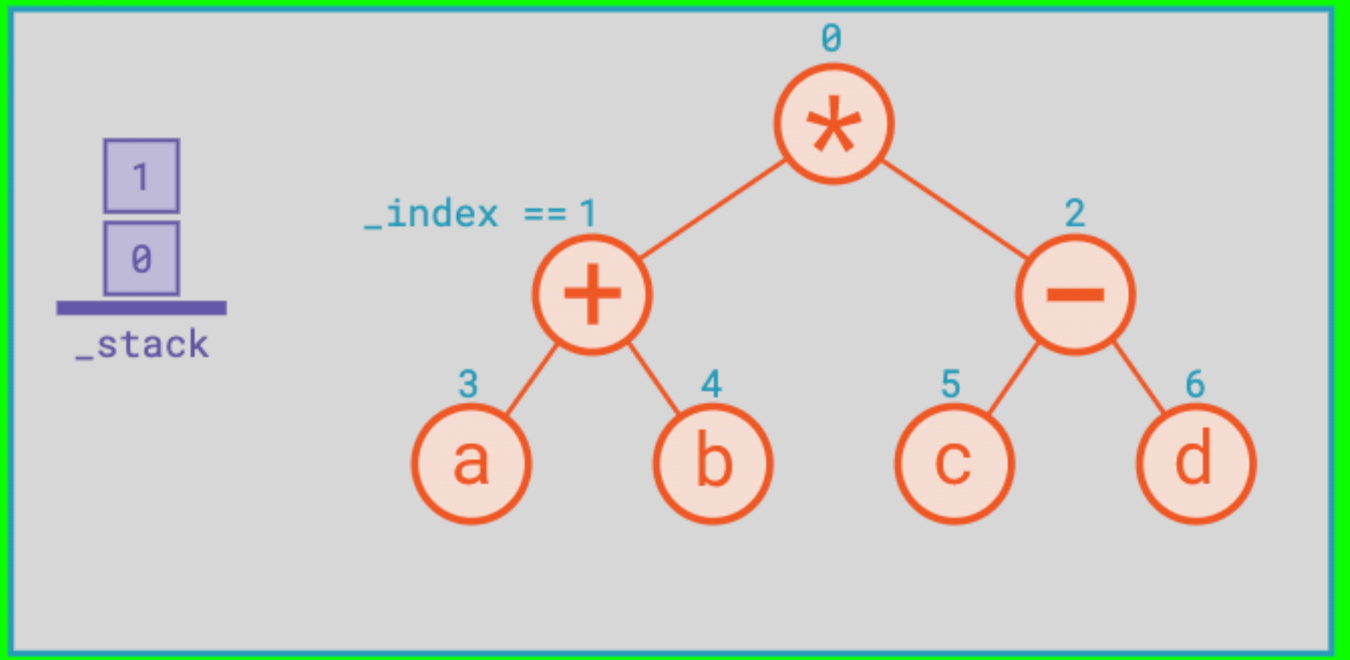

0
*
1
+
2
−
3
a
4
b
5
c
6
d

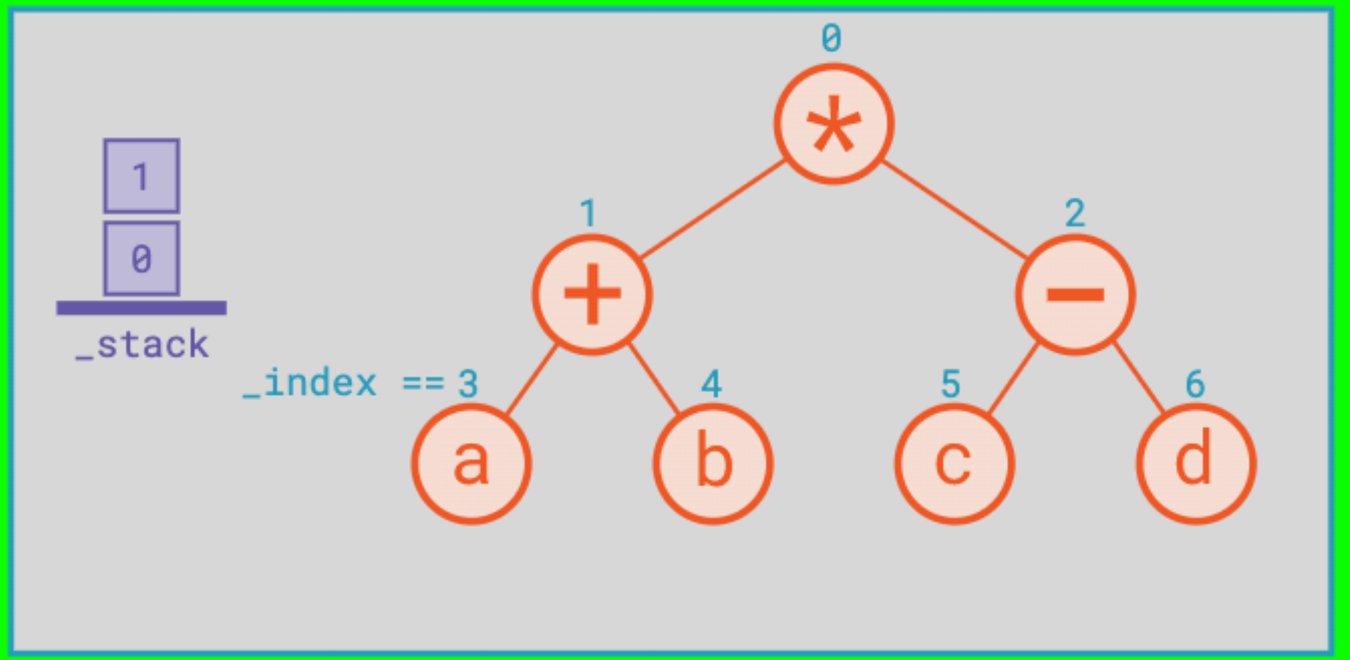

_stack
0
*
1
+
2
−
3
a
4
b
5
c
6
d

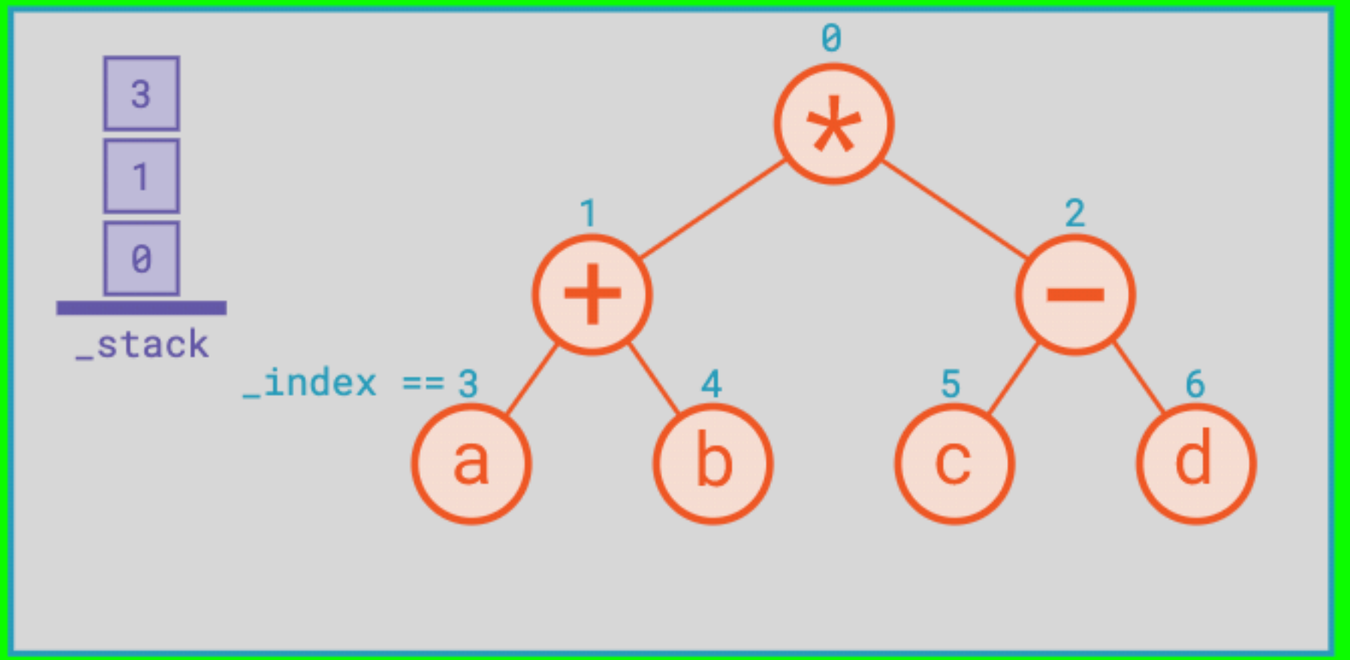

_index == 0
*
1
2
+
−
3
4
5
6
a
b
c
d
_stack

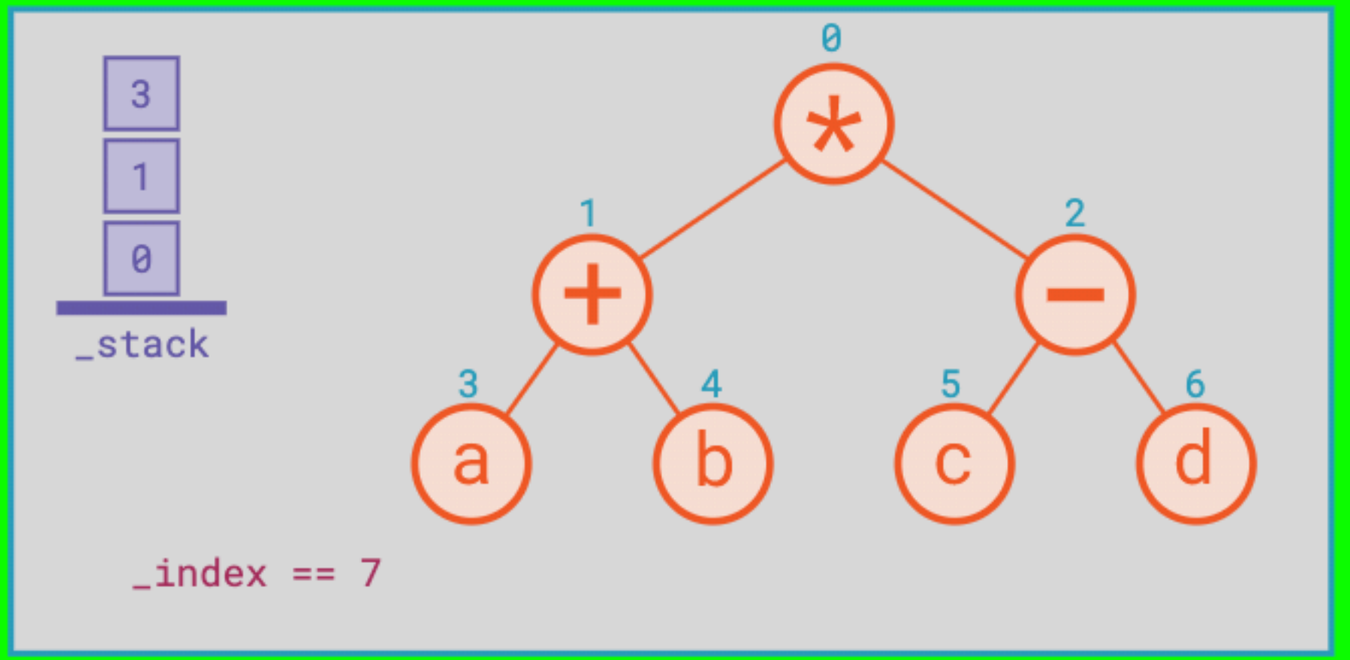

_index == 0
0
_stack
*
1
2
+
−
3
4
5
6
a
b
c
d

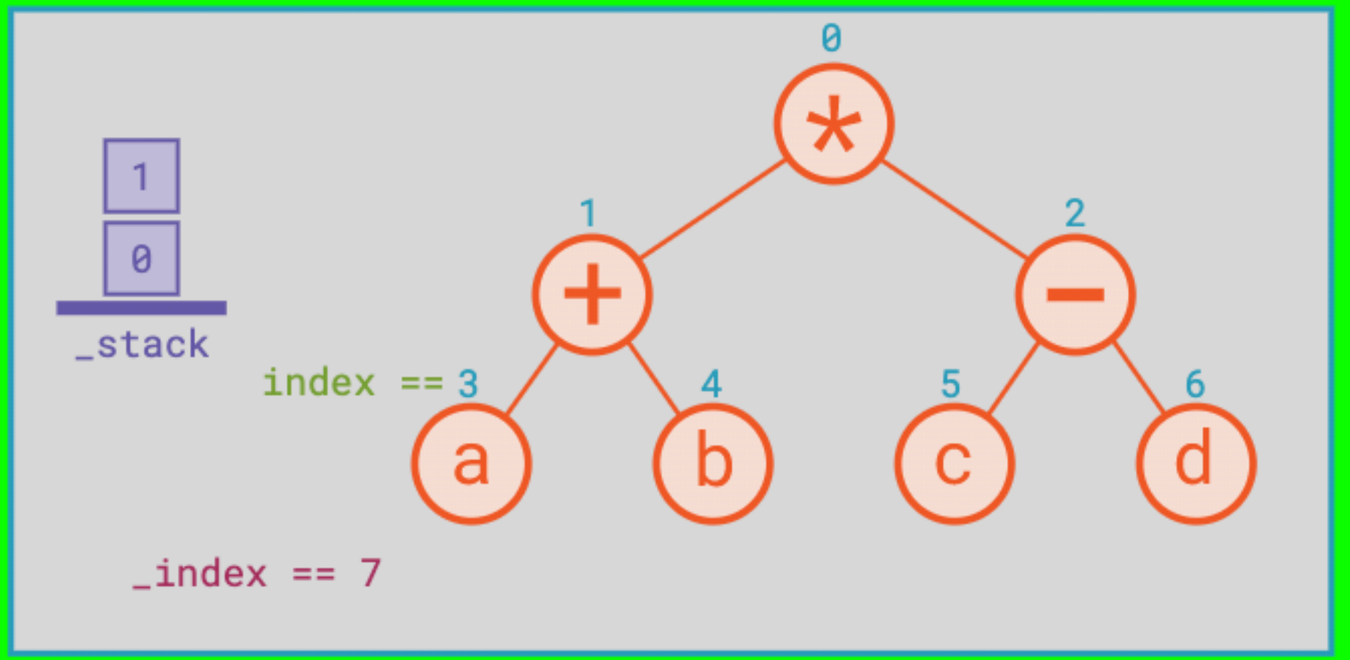

0
*
_index == 1
2
0
+
−
_stack
3
4
5
6
a
b
c
d

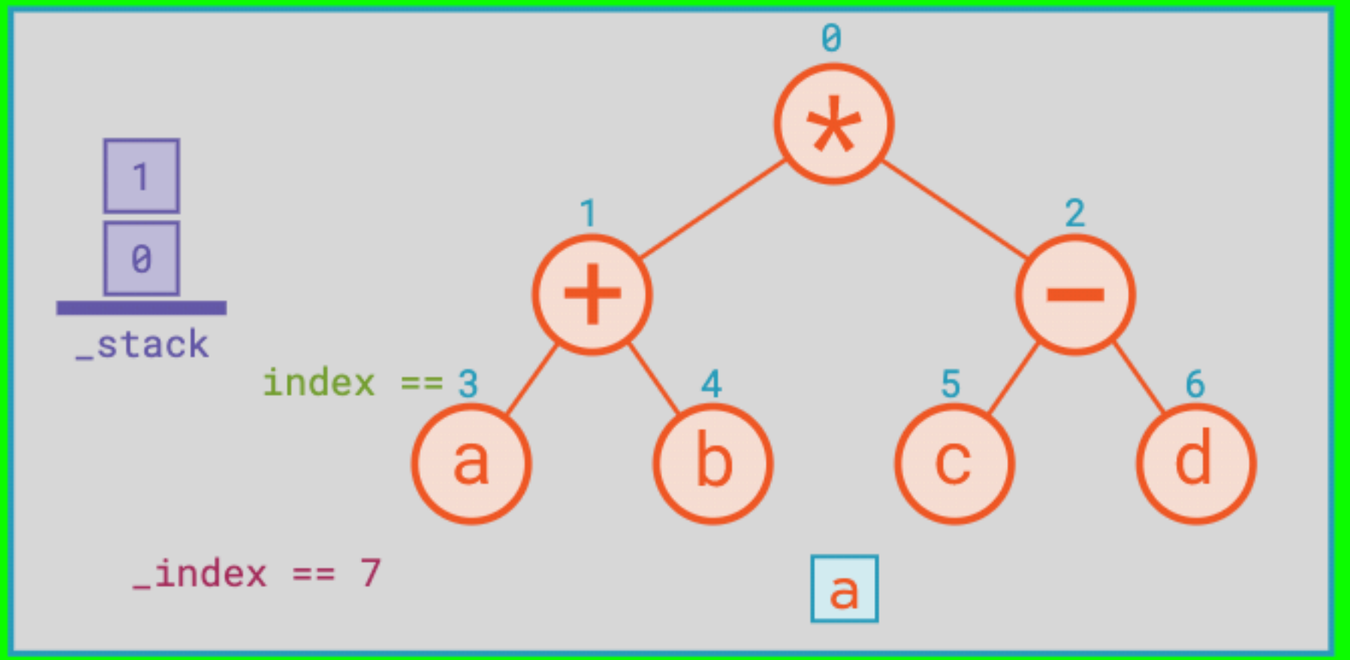

1
0
_stack
_index == 1
0
*
2
+
−
3
4
5
6
a
b
c
d

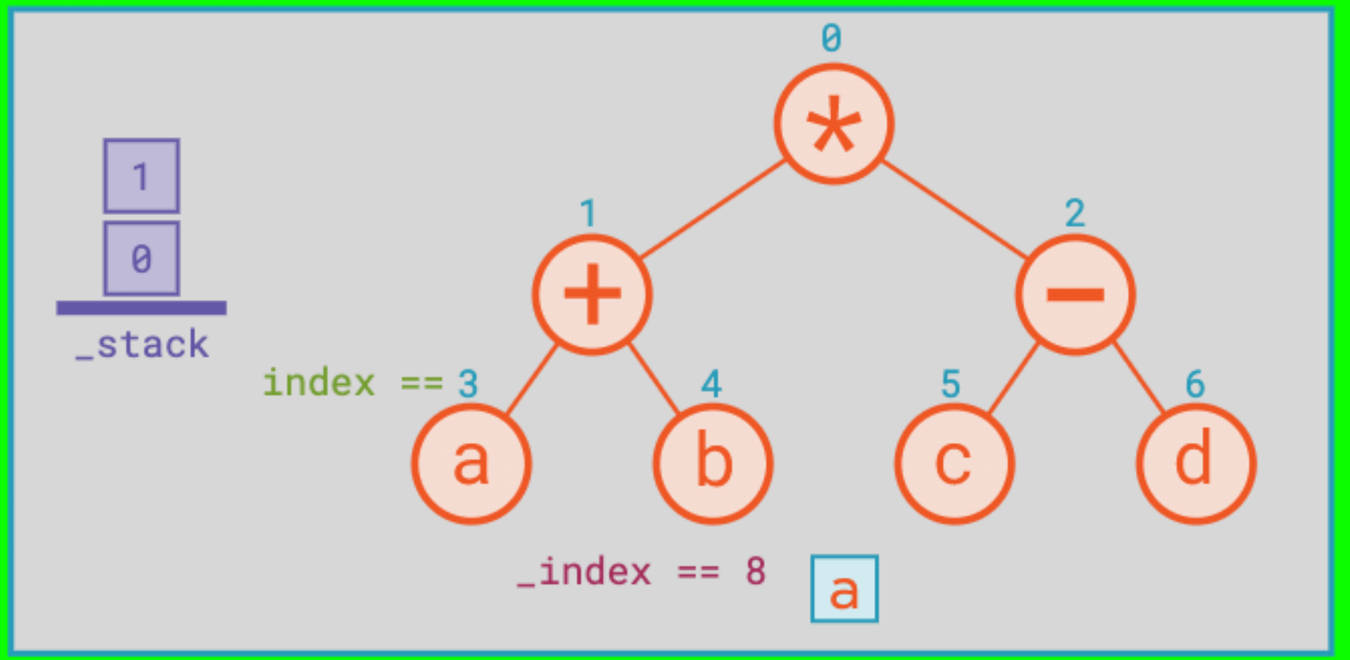

0
*
1
+
2
−
3
a
4
b
5
c
6
d
_index ==
1
0
_stack

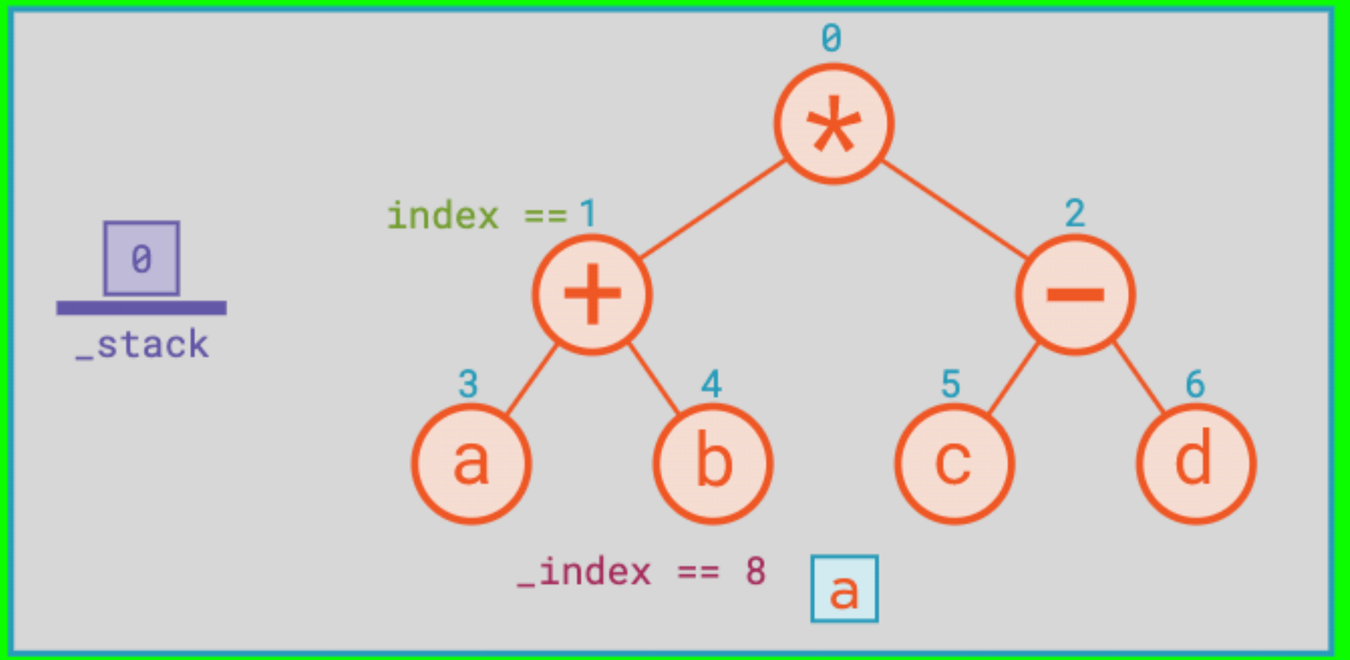

3
1
0
_stack
_index == 3
0
*
1
+
2
−
a
b
c
d
4
5
6

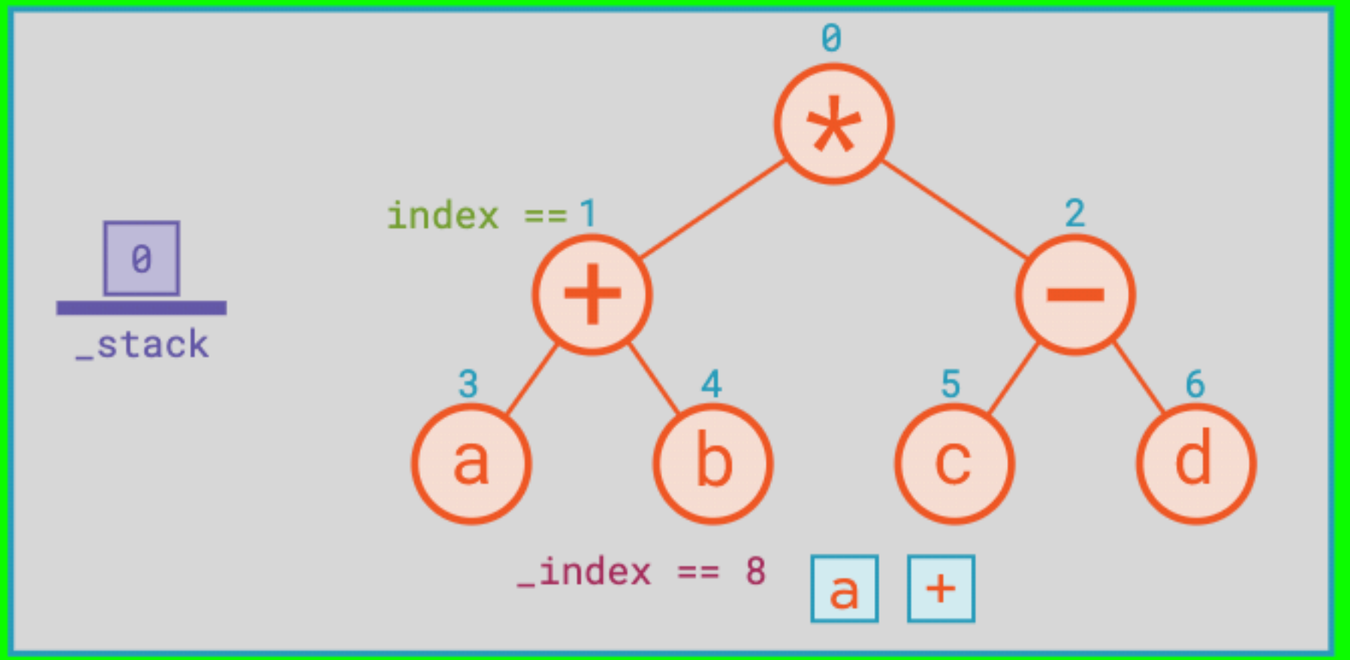

3
1
0
_stack
0
*
1
+
2
−
3
a
4
b
5
c
6
d
_index == 7

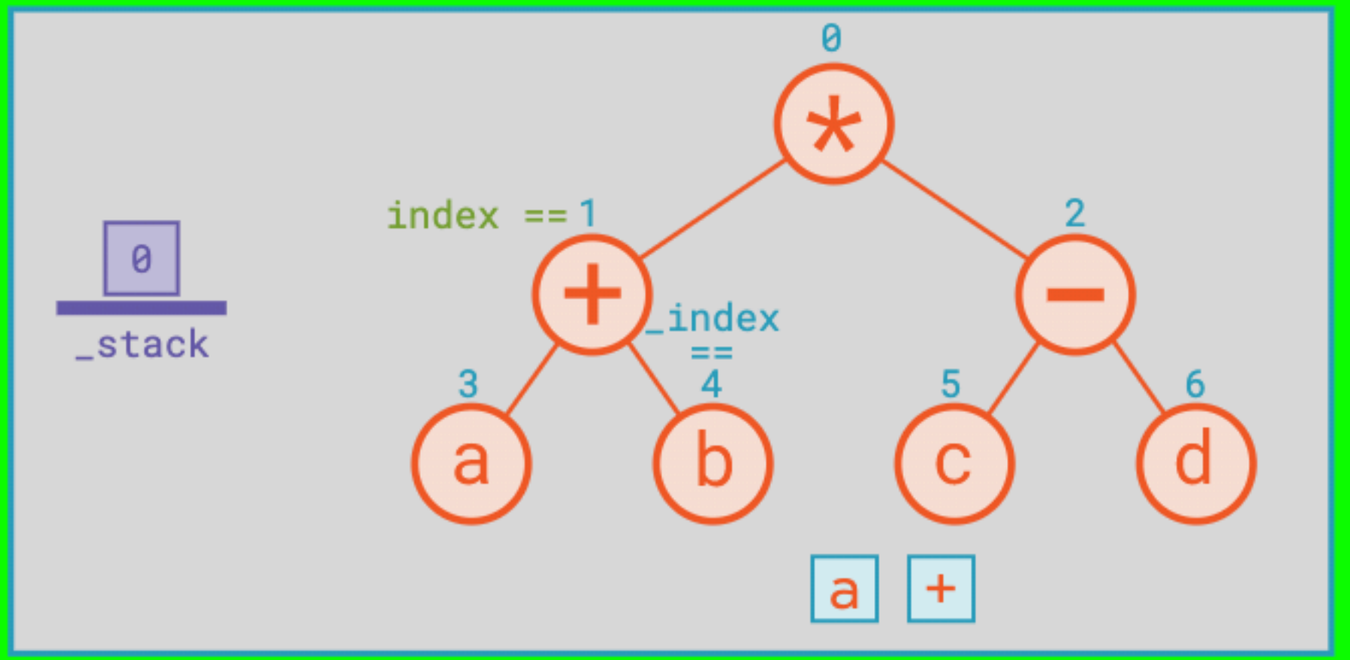

0
*
1
+
2
−
1
0
_stack
index == 3
4
5
6
a
b
c
d
_index == 7


0
*
1
+
2
−
3
a
4
b
5
c
6
d
1
0
_stack
index == 3
_index == 7
a


0
*
1
+
2
−
index == 3
4
5
6
a
b
c
d
1
0
_stack
_index == 8
a


0
*
index == 1
2
+
−
3
4
5
6
a
b
c
d
0
_stack
_index == 8
a


0
*
index == 1
2
+
−
3
4
5
6
a
b
c
d
0
_stack
_index == 8
a
+

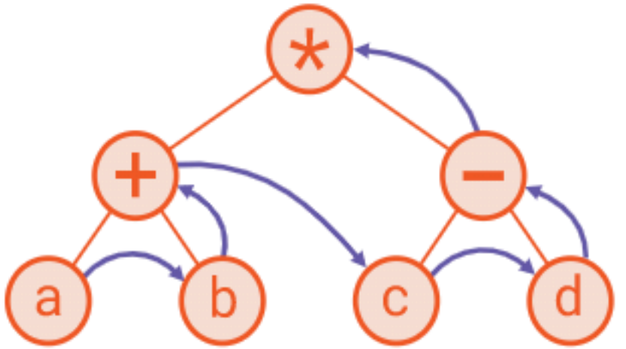

0
_stack
0
*
index == 1
2
+
_index
==
4
−
3
5
6
a
b
c
d
a
+

Tree Traversals

level-order

in-order

pre-order



post-order

Loose Coupling

Collection

`list` doesn't know about tree iterators

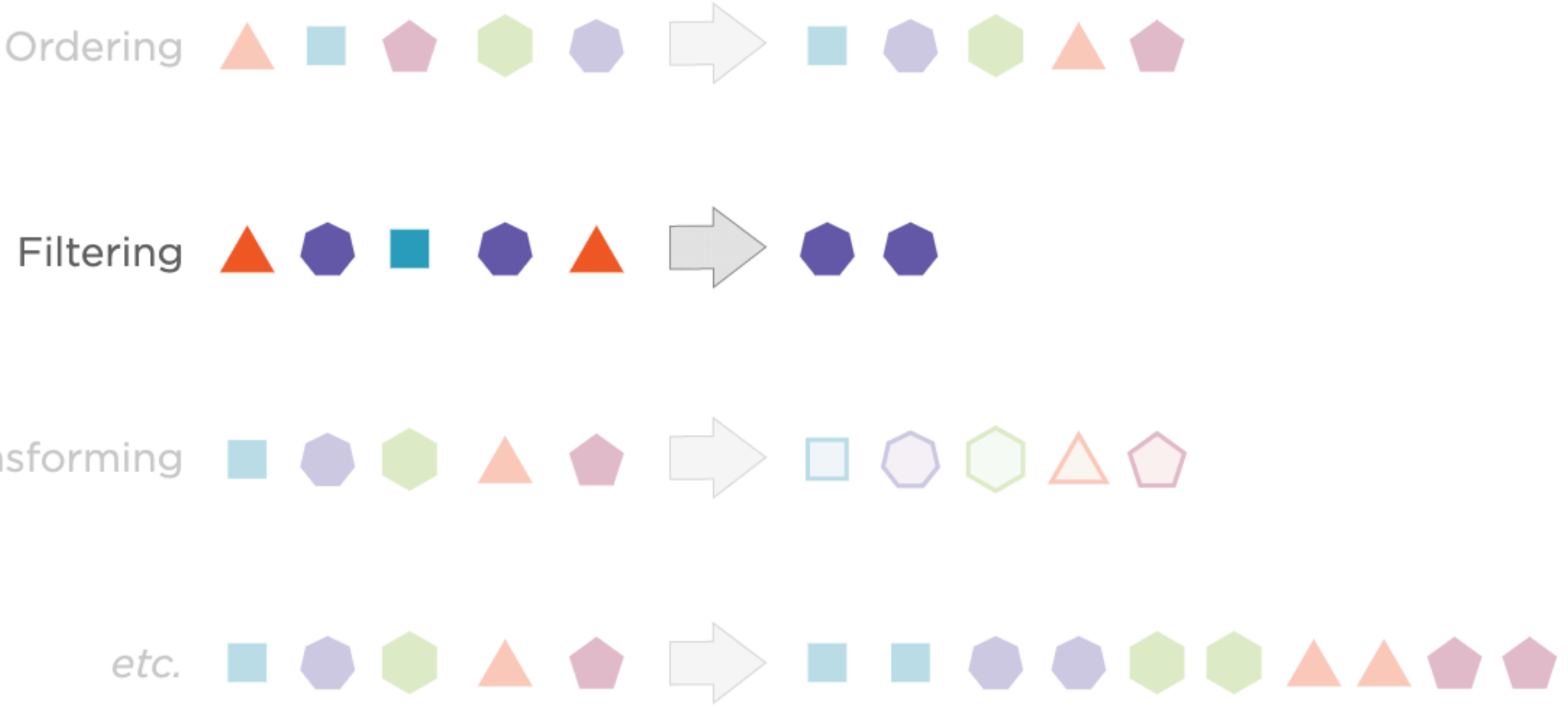
Tree Iterators

Don't know directly about `list`

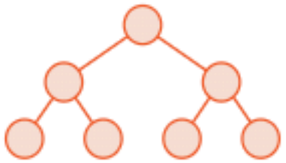
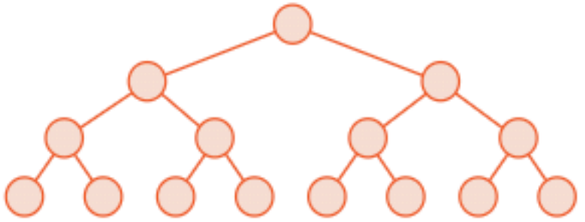
Sequence Protocol

Used by tree iterators to access list

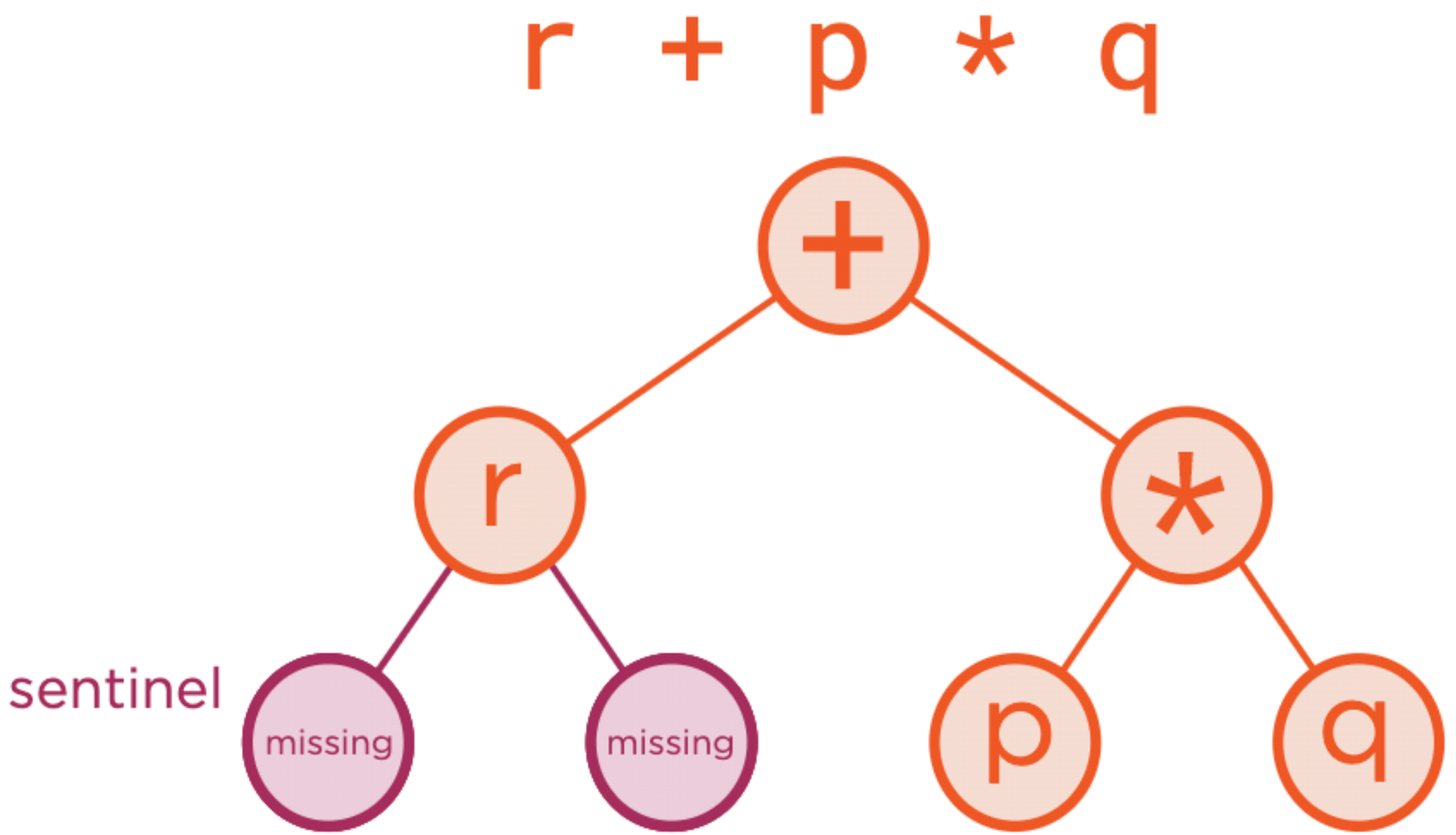
Processing with Iterators



Perfect Binary Trees

1	3	7	15
$2^1 - 1$	$2^2 - 1$	$2^3 - 1$	$2^4 - 1$
			
perfect	perfect	perfect	perfect
			

Representing Imperfect Binary Trees



iterators.py

```
107 class SkipMissingIterator:
108
109     def __init__(self, iterable):
110         self._iterator = iter(iterable)
111
112     def __next__(self):
113         while True:
114             item = next(self._iterator)
115             if item is not missing:
116                 return item
```

Python Console

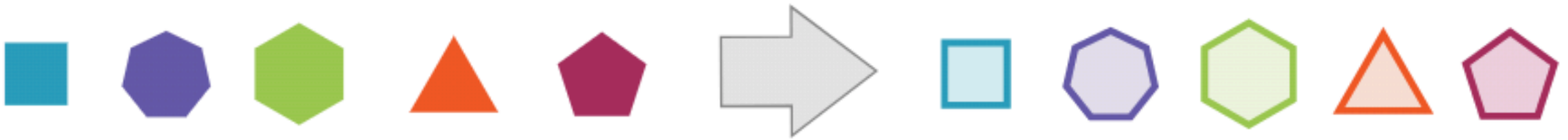
```
Python Console
>>> from iterators import *
>>> expr_tree = ["+", "r", "*", missing, missing, "p", "q"]
>>> iterator = SkipMissingIterator(expr_tree)
>>> list(iterator)
['+', 'r', '*', 'p', 'q']
>>> iterator = SkipMissingIterator(InOrderIterator(expr_tree))
>>> " ".join(iterator)
'r + p * q'

>>>
```

Transforming Iterators

Ordering

Filtering

Transforming



Arithmetic Symbol Translation

p * q - r / s + t

p × q − r ÷ s + t

Symbol Translation Dictionary

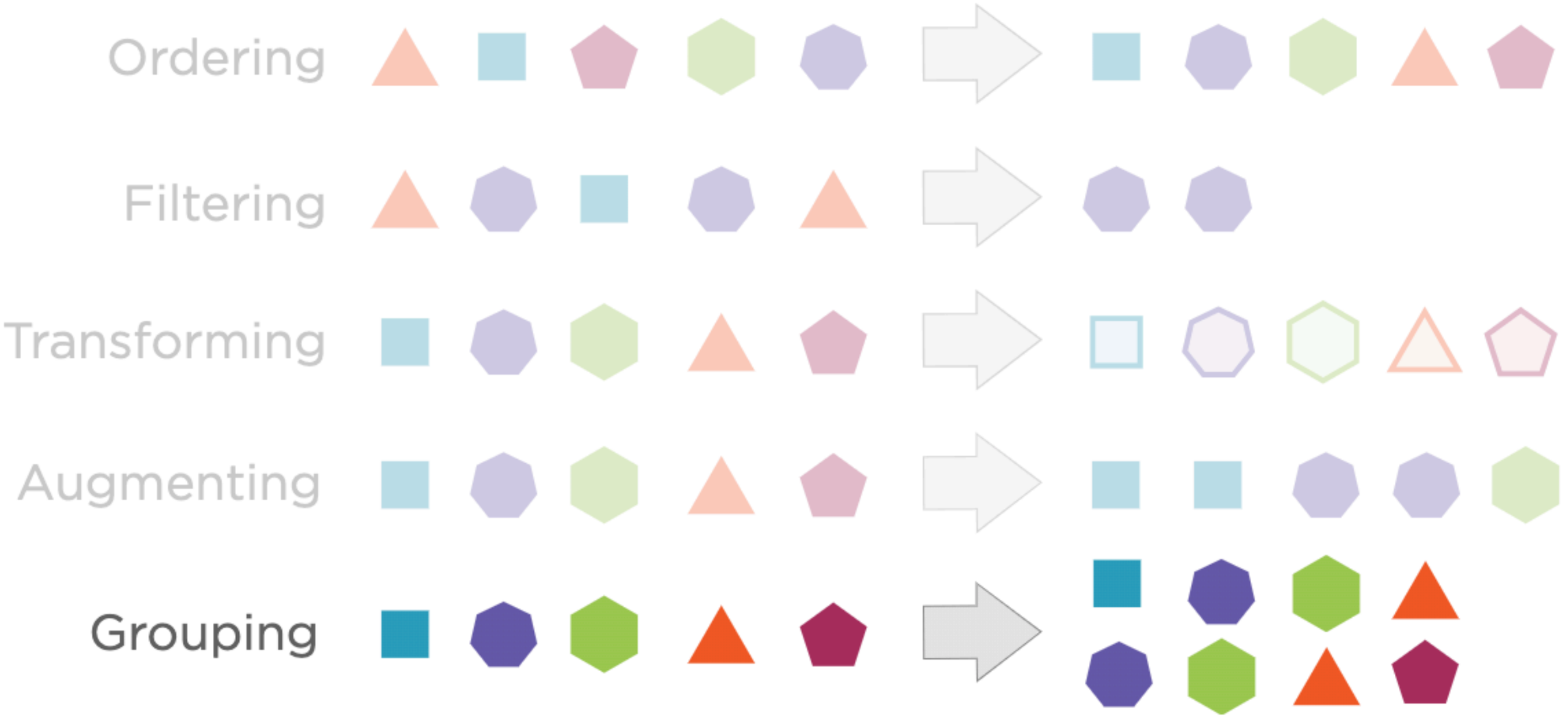
```
typesetting_table = {
    "-": "\u2212",   # Minus sign
    "*": "\u00D7",   # Multiplication sign
    "/": "\u00F7",   # Division sign
}
```

```
111
112     def __next__(self):
113         while True:
114             item = next(self._iterator)
115             if item is not missing:
116                 return item
117
118     def __iter__(self):
119         return self
```

Python Console

```
...         "*",          "/",
...      "p",   "q",   "r",    "+",
...     m, m,   m, m,  m, m, "s", "t"
... ]
>>> iterator = TranslationIterator(
...     typesetting_table,
...     SkipMissingIterator(InOrderIterator(expr_tree))
... )
>>> " ".join(iterator)
'p × q − r ÷ s + t'

>>>
```

Flexible Iterators



itertools — Functions creating iterators for efficient looping

This module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an "iterator algebra" making it possible to construct specialized tools succinctly and efficiently in pure Python.

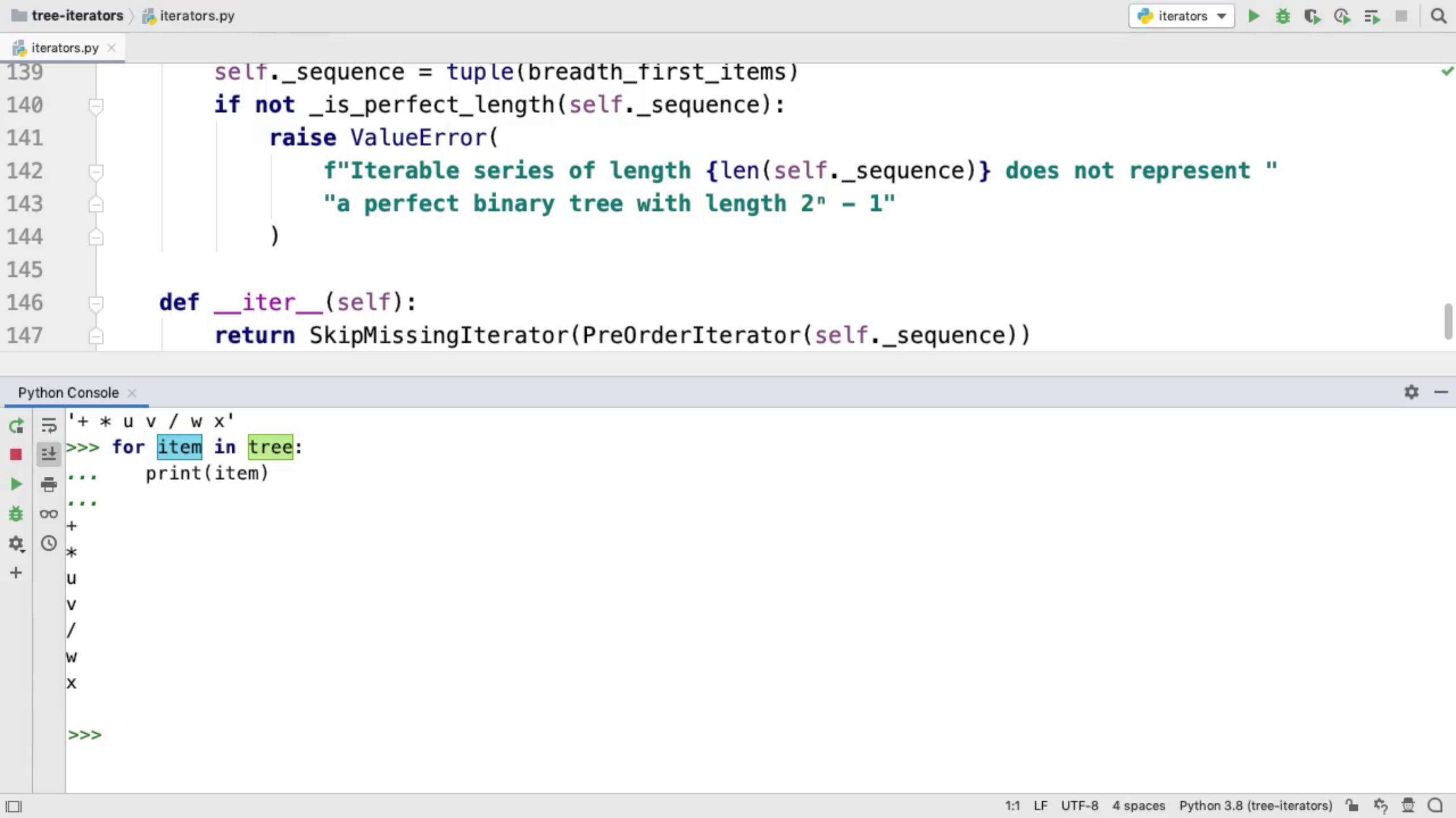
For instance, SML provides a tabulation tool: `tabulate(f)` which produces a sequence `f(0), f(1), ...`. The same effect can be achieved in Python by combining `map()` and `count()` to form `map(f, count())`.

These tools and their built-in counterparts also work well with the high-speed functions in the `operator` module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: `sum(map(operator.mul, vector1, vector2))`.

Infinite iterators:

Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up	`repeat(10, 3) --> 10 10 10`

Iterables


tree-iterators
iterators.py
iterators
iterators.py
139 self._sequence = tuple(breadth_first_items)
140 if not _is_perfect_length(self._sequence):
141 raise ValueError(
142 f"Iterable series of length {len(self._sequence)} does not represent "
143 "a perfect binary tree with length 2ⁿ - 1"
144)
145
146 def __iter__(self):
147 return SkipMissingIterator(PreOrderIterator(self._sequence))
Python Console
'+ * u v / w x'
>>> for item in tree:
... print(item)
...
+
*
u
v
/
w
x
>>>
1:1 LF UTF-8 4 spaces Python 3.8 (tree-iterators)

Iterators are usually used indirectly, through iterable-consuming functions and for-loops

The Alternative Iterable Protocol

`__getitem__`

Called to implement evaluation of `sequence[index]`

...the accepted keys should be integers...

...if a value outside the set of indexes for the sequence `IndexError` should be raised...

Python 3 data model documentation

The Alternative Iterable Protocol

```
class MyAlternativeIterable:

    ...

    def __getitem__(self, index):
        if index >= self.number_of_items():
            raise IndexError
        return self.get_element_at(index)
```

An Evenly and Exactly Divided Range

rational_range.py

```
        raise ValueError(f"num_steps {num_steps} is not positive")
    self._start = Fraction(start)
    self._num_steps = num_steps
    self._step = Fraction(stop - start, num_steps)

def __getitem__(self, index):
    if not (0 <= index < self._num_steps):
        raise IndexError
    return self._start + index * self._step
```

Python Console

```
5
19/3
23/3
9
31/3
35/3
>>> [float(item) for item in r]
[5.0, 6.333333333333333, 7.666666666666667, 9.0, 10.333333333333334, 11.666666666666666]
>>> sum(r)
Fraction(50, 1)

>>>
```

Core Python: Numeric Types, Dates and Times

on

PLURALSIGHT

Two Forms of `iter()`

```
iterator = iter(iterable)
```

```
iterator = iter(

    callable,

    sentinel

)
```

◀ zero-argument callable - invoked once per iteration

◀ iteration ends when the callable produces this value

```
291
149
17
31
547
2069
END
```

```
>>> with open("end_terminated_file.txt", 'rt') as f:
...     lines = iter(lambda: f.readline().strip(), "END")
...     readings = [int(line) for line in lines]
...
>>> readings
[291, 149, 17, 31, 547, 2069]
>>>
```

Timestamp Iterator

```
>>> import datetime
>>> timestamps = iter(datetime.datetime.now, None)
>>> next(timestamps)
2020-07-27 18:36:15.329098
>>> next(timestamps)
2020-07-27 18:36:22.429098
>>> next(timestamps)
2020-07-27 18:36:29.629098
>>> next(timestamps)
2020-07-27 18:36:37.629098
>>> next(timestamps)
2020-07-27 18:36:45.129098
>>> next(timestamps)
2020-07-27 18:36:53.729098
>>>
```

Realtime Data Iterator

```
...      time.sleep(1.0)
...
2020-07-27 18:38:31.774068 910868606976
2020-07-27 18:38:32.773330 910868588707
2020-07-27 18:38:33.773438 910868570438
2020-07-27 18:38:34.772775 910868552169
2020-07-27 18:38:35.773281 910868533900
2020-07-27 18:38:36.774501 910868515631
2020-07-27 18:38:37.774541 910868497362
2020-07-27 18:38:38.773032 910868479093
2020-07-27 18:38:39.773038 910868460824
2020-07-27 18:38:40.774318 910868442555
2020-07-27 18:38:41.774103 910868424286
```

Summary

`next(iterator)` delegates to `iterator.__next__()`

Iterators must support `__next__()`

`__next__()` should return the next item in the series

If there are no more items, `__next__` should raise `StopIteration`

Summary

`iter(iterable)` delegates to `iterable.__iter__()`

Iterable objects must support `__iter__()`

`__iter__()` should return an iterator

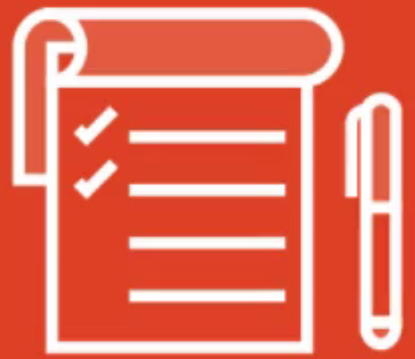
Iterators must also be iterable, so implement both `__iter__()` and `__next__()`

Summary

Objects with a `__getitem__()` method that accepts consecutive integers from zero are also iterable

Iterables implemented via `__getitem__()` must raise `IndexError` when exhausted

Summary



The two-argument form of `iter()` accepts a zero-argument callable and a sentinel

On each iteration the callable is invoked, until the sentinel is returned

The iterator yields values from the callable

Convert simple functions into iterators