What are iterable objects?

PRACTICING CODING INTERVIEW QUESTIONS IN PYTHON

Kirill Smirnov
Data Science Consultant, Altran

datacamp

Definition

iterable objects / Iterables - any object that can be used in a `for` loop

- list
- tuple
- set
- dictionary
- string

Iterating through a list or tuple

list:

```python
droids = ['R2-D2', 'TC-16', 'C-3PO']

for droid in droids:
    print(droid)
```

```
R2-D2
TC-16
C-3PO
```

tuple:

```python
droids = ('R2-D2', 'TC-16', 'C-3PO')

for droid in droids:
    print(droid)
```

```
R2-D2
TC-16
C-3PO
```

Iterating through a set

```python
battleships = {'X-Wing Fighter', 'Millennium Falcon', 'TIE Fighter'}

for battleship in battleships:
    print(battleship)
```

```
TIE Fighter
X-Wing Fighter
Millennium Falcon
```

Iterating through a string

```python
title = 'Star Wars'

for char in title:
    print(char)
```

```
S
t
a
r
 
W
a
r
s
```

Iterating through a dictionary

```python
episodes = {
    'Episode I': 'The Phantom Menace',
    'Episode II': 'Attack of the Clones',
    'Episode III': 'Revenge of the Sith',
    'Episode IV': 'A New Hope',
    'Episode V': 'The Empire Strikes Back',
    'Episode VI': 'Return of the Jedi'
}
```

```python
for episode in episodes:
    print(episode)
```

```
Episode I
Episode II
Episode III
Episode IV
Episode V
Episode VI
```

Getting key-value pairs

```
episodes = {
    'Episode I': 'The Phantom Menace',
    'Episode II': 'Attack of the Clones',
    'Episode III': 'Revenge of the Sith',
    'Episode IV': 'A New Hope',
    'Episode V': 'The Empire Strikes Back',
    'Episode VI': 'Return of the Jedi'
}
```

```
for item in episodes.items():
    print(item)
```

```
('Episode I', 'The Phantom Menace')
('Episode II', 'Attack of the Clones')
('Episode III', 'Revenge of the Sith')
('Episode IV', 'A New Hope')
('Episode V', 'The Empire Strikes Back')
('Episode VI', 'Return of the Jedi')
```

Getting key-value pairs

```python
episodes = {
    'Episode I': 'The Phantom Menace',
    'Episode II': 'Attack of the Clones',
    'Episode III': 'Revenge of the Sith',
    'Episode IV': 'A New Hope',
    'Episode V': 'The Empire Strikes Back',
    'Episode VI': 'Return of the Jedi'
}
```

```python
for title, subtitle in episodes.items():
    print(title + ': ' + subtitle)
```

```
'Episode I': 'The Phantom Menace'
'Episode II': 'Attack of the Clones'
'Episode III': 'Revenge of the Sith'
'Episode IV': 'A New Hope'
'Episode V': 'The Empire Strikes Back'
'Episode VI': 'Return of the Jedi'
```

Less visual objects: range

```python
interval = range(0, 10)
print(interval)
```

```
range(0, 10)
```

```python
for num in interval:
    print(num)
```

```
0
1
2
...
9
```

Less visual objects: enumerate

```python
villains = ['Darth Maul', 'Palpatine', 'Darth Vader']
enum_villains = enumerate(villains)
```

```python
for item in enum_villains:
    print(item)
```

```
(0, 'Darth Maul')
(1, 'Palpatine')
(2, 'Darth Vader')
```

Less visual objects: enumerate

```python
villains = ['Darth Maul', 'Palpatine', 'Darth Vader']
enum_villains = enumerate(villains)
```

```python
for idx, name in enum_villains:
    print(str(idx) + ' - ' + name)
```

```
0 - Darth Maul
1 - Palpatine
2 - Darth Vader
```

Iterables as arguments

`list()` , `tuple()` , `set()` , *etc.*

```
villains = [
    'Darth Maul',
    'Palpatine',
    'Darth Vader'
]
```

```
list(enumerate(villains))
```

```
[
    (0, 'Darth Maul'),
    (1, 'Palpatine'),
    (2, 'Darth Vader')
]
```

Iterables as arguments

`list()` , `tuple()` , `set()` , *etc.*

```
villains = [
    'Darth Maul',
    'Palpatine',
    'Darth Vader'
]
```

```
list(enumerate(villains))
```

```
[
    (0, 'Darth Maul'),
    ...
```

```
set(enumerate(villains))
```

```
{
    (0, 'Darth Maul'),
    (1, 'Palpatine'),
    (2, 'Darth Vader')
}
```

How to know if we deal with an Iterable

```
interval = range(0, 5)
```

```
interval_iter = iter(interval)
```

```
print(interval_iter)
```

```
<range_iterator object at 0x7f3bdf8ad300>
```

Iterator - an object knowing how to retrieve consecutive elements from an Iterable one by one

```
next(interval_iter)
```

```
0
```

```
next(interval_iter)
```

```
1
```

```
next(interval_iter)
```

```
2
```

StopIteration

```
next(interval_iter)
```

```
3
```

```
next(interval_iter)
```

```
4
```

```
next(interval_iter)
```

```
StopIteration
```

Describing a for loop

```python
droids = ['R2-D2', 'TC-16', 'C-3PO']

for droid in droids:
    print(droid)
```

```
R2-D2
TC-16
C-3PO
```

```python
iter_droids = iter(droids)
while True:
    try:



    except StopIteration:
        break
```

Describing a for loop

```python
droids = ['R2-D2', 'TC-16', 'C-3PO']

for droid in droids:
    print(droid)
```

```
R2-D2
TC-16
C-3PO
```

```python
iter_droids = iter(droids)
while True:
    try:
        droid = next(iter_droid)
        print(droid)
    except StopIteration:
        break
```

```
R2-D2
TC-16
C-3PO
```

Many Iterables are Iterators

- iter()
- next()

e.g. **enumerate, finditer** *etc.*

```python
import re
pattern = re.compile(r'[\w\.]+@[a-z]+\.[a-z]+')

text = 'john.smith@mailbox.com is the e-mail of John. He often writes to his boss '\
'at boss@company.com. But the messages get forwarded to his secretary at info@company.com.'

result = re.finditer(pattern, text)
```

iter() or next()

`iter()`

```python
result = re.finditer(pattern, text)

for item in result:
    print(item)
```

```
<_sre.SRE_Match object; span=(0, 22), match='john.smith@mailbox.com'>
<_sre.SRE_Match object; span=(77, 93), match='boss@company.com'>
<_sre.SRE_Match object; span=(146, 162), match='info@company.com'>
```

iter() or next()

`next()`

```
result = re.finditer(pattern, text)
```

```
next(result)
```

```
<_sre.SRE_Match object; span=(0, 22),
match='john.smith@mailbox.com'>
```

```
next(result)
```

```
<_sre.SRE_Match object; span=(77, 93),
match='boss@company.com'>
```

```
next(result)
```

```
<_sre.SRE_Match object; span=(146, 162),
match='info@company.com'>
```

Expendable Iterables

```python
result = re.finditer(pattern, text)
for item in result:
    print(item)
```

```
<_sre.SRE_Match object; ...
<_sre.SRE_Match object; ...
<_sre.SRE_Match object; ...
```

```python
for item in result:
    print(item)
```

```
# nothing
```

```python
short_list = [2, 4]
for item in short_list:
    print(item)
```

```
2
4
```

```python
for item in short_list:
    print(item)
```

```
2
4
```

Traversing a DataFrame

```
pars = {'weight': [168, 183, 198], 'height': [77, 79, 135]}
characters = pd.DataFrame(pars, index=['Luke Skywalker', 'Han Solo', 'Darth Vader'])
print(characters)
```

```
                weight  height
Luke Skywalker     168      77
Han Solo           183      79
Darth Vader        198     135
```

Direct approach

```python
for item in characters:
    print(item)
```

```
weight
height
```

.iterrows()

```
result = characters.iterrows()
```

```
print(result)
```

```
<generator object DataFrame.iterrows at 0x7f5dff6b9c50>
```

.iterrows()

```python
result = characters.iterrows()
```

```python
for item in result:
    print(item)
```

`item` → `(index name, Series)`

```
('Luke Skywalker',
weight    168
height     77
Name: Luke Skywalker, dtype: int64)
('Han Solo',
weight    183
height     79
Name: Han Solo, dtype: int64)
('Darth Vader',
weight    198
height    135
Name: Darth Vader, dtype: int64)
```

.iterrows()

```
result = characters.iterrows()
```

```
for index, series in result:
    print(index)
    print(series)
```

```
Luke Skywalker
weight    168
height     77
Name: Luke Skywalker, dtype: int64)
Han Solo
weight    183
height     79
Name: Han Solo, dtype: int64)
Darth Vader
weight    198
height    135
Name: Darth Vader, dtype: int64)
```

.iteritems()

```
result = characters.iteritems()
```

```
print(result)
```

```
<generator object DataFrame.iteritems at 0x7f5dff69f938>
```

.iteritems()

```python
result = characters.iteritems()
```

```python
for item in result:
    print(item)
```

`item` → `(column name, Series)`

```
('weight',
Luke Skywalker    168
Han Solo          183
Darth Vader       198
Name: weight, dtype: int64)
('height',
Luke Skywalker     77
Han Solo           79
Darth Vader       135
Name: height, dtype: int64)
```

.iteritems()

```
result = characters.iteritems()
```

```
for name, series in result:
    print(name)
    print(series)
```

```
weight
Luke Skywalker    168
Han Solo          183
Darth Vader       198
Name: weight, dtype: int64
height
Luke Skywalker     77
Han Solo           79
Darth Vader       135
Name: height, dtype: int64
```

Let's practice!

PRACTICING CODING INTERVIEW QUESTIONS IN PYTHON

What is a list comprehension?

PRACTICING CODING INTERVIEW QUESTIONS IN PYTHON

Kirill Smirnov
Data Science Consultant, Altran

datacamp

List comprehension

```
nums = [2, 4, 6, 8, 10]
print(nums)
```

```
[2, 4, 6, 8, 10]
```

```
nums_new = []
for i in range(1, 6):
    nums_new.append(2*i)

print(nums_new)
```

```
[2, 4, 6, 8, 10]
```

List comprehension

```python
nums = [2, 4, 6, 8, 10]
print(nums)
```

```
[2, 4, 6, 8, 10]
```

```python
                      for num in range(1, 6)
```

List comprehension

```python
nums = [2, 4, 6, 8, 10]
print(nums)
```

```
[2, 4, 6, 8, 10]
```

```python
            [          for num in range(1, 6)]
```

List comprehension

```
nums = [2, 4, 6, 8, 10]
print(nums)
```

```
[2, 4, 6, 8, 10]
```

```
            [(2 * num) for num in range(1, 6)]
```

List comprehension

```python
nums = [2, 4, 6, 8, 10]
print(nums)
```

```
[2, 4, 6, 8, 10]
```

```python
nums_new = [(2 * num) for num in range(1, 6)]
```

```python
print(nums_new)
```

```
[2, 4, 6, 8, 10]
```

Summing up

List comprehension is defined by:

-
-

Summing up

List comprehension is defined by:

- an iterable object (*e.g.* list, tuple, set)
-

```
[             for num in range(1, 6)]
```

Summing up

List comprehension is defined by:

- an iterable object (*e.g.* list, tuple, set)
- an operation on an element

```
[(2 * num) for num in range(1, 6)]
```

- (optional) conditions

List comprehension with condition

```python
nums = [2, 4, 6, 8, 10]
print(nums)
```

```
[2, 4, 6, 8, 10]
```

1 2 3 4 5 6 7 8 9 10

List comprehension with condition

```python
nums = [2, 4, 6, 8, 10]
print(nums)
```

```
[2, 4, 6, 8, 10]
```

1 **2** 3 **4** 5 **6** 7 **8** 9 **10** $\rightarrow$ **2 4 6 8 10**

Adding a condition

```
nums_new = [num for num in range(1, 11)]
```

```
print(nums_new)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Adding a condition

```python
nums_new = [num for num in range(1, 11) if num % 2 == 0]
```

```python
print(nums_new)
```

```
[2, 4, 6, 8, 10]
```

More examples

```
text = 'list COMPREHENSION is A way TO create LISTS'
```

Task:

Create a list that contains the length of each lowercased word.

`list` , `is` , `way` , `create` $\longrightarrow$ `[4, 2, 3, 6]`

More examples

```python
text = 'list COMPREHENSION is A way TO create LISTS'
```

Task:

Create a list that contains the length of each lowercased word.

`list` , `is` , `way` , `create` → `[4, 2, 3, 6]`

```python
output = [           for word in text.split()                  ]
```

More examples

```
text = 'list COMPREHENSION is A way TO create LISTS'
```

Task:

Create a list that contains the length of each lowercased word.

`list`, `is`, `way`, `create` $\rightarrow$ `[4, 2, 3, 6]`

```
output = [            for word in text.split() if word.islower()]
```

More examples

```
text = 'list COMPREHENSION is A way TO create LISTS'
```

Task:

Create a list that contains the length of each lowercased word.

`list` , `is` , `way` , `create` $\rightarrow$ `[4, 2, 3, 6]`

```
output = [len(word) for word in text.split() if word.islower()]
```

```
print(output)
```

```
[4, 2, 3, 6]
```

Multiple loops

```
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
```

Create all the possible pairs between `numbers` and `letters` :

```
[
    (1, 'a'), (1, 'b'), (1, 'c'),
    (2, 'a'), (2, 'b'), (2, 'c'),
    (3, 'a'), (3, 'b'), (3, 'c'),
]
```

Iterating through multiple loops

```
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
```

```
pairs = [        for i in numbers                 ]
```

Iterating through multiple loops

```
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
```

```
pairs = [        for i in numbers for j in letters]
```

Iterating through multiple loops

```python
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
```

```python
pairs = [(i, j) for i in numbers for j in letters]
```

```python
print(pairs)
```

```
[
    (1, 'a'), (1, 'b'), (1, 'c'),
    (2, 'a'), (2, 'b'), (2, 'c'),
    (3, 'a'), (3, 'b'), (3, 'c'),
]
```

Deeper look

```
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
```

```
pairs = [(i, j) for i in numbers for j in letters]
```

```
pairs = []
for i in numbers:
    for j in letters:
        pairs.append((i, j))
```

Deeper look

```python
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
```

```python
pairs = [(i, j)                  for j in letters]
```

```python
pairs = []

    for j in letters:
        pairs.append((i, j))
```

Deeper look

```python
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
```

```python
pairs = [(i, j) for i in numbers                  ]
```

```python
pairs = []
for i in numbers:


        pairs.append((i, j))
```

Adding square brackets

```
pairs = [ (i, j) for i in numbers  for j in letters]
```

Adding square brackets

```python
pairs = [[(i, j) for i in numbers] for j in letters]
```

```python
print(pairs)
```

```
[
    [(1, 'a'), (2, 'a'), (3, 'a')],
    [(1, 'b'), (2, 'b'), (3, 'b')],
    [(1, 'c'), (2, 'c'), (3, 'c')]
]
```

Adding square brackets

```
pairs = [[(i, j) for i in numbers] for j in letters]
```

```
pairs = []
for j in letters:
    temp = []
    for i in numbers:
        temp.append((i, j))
    pairs.append(temp)
```

Adding square brackets

```
pairs = [[(i, j) for i in numbers]                  ]
```

```
pairs = []


    temp = []
    for i in numbers:
        temp.append((i, j))
    pairs.append(temp)
```

Adding square brackets

```
pairs = [[(i, j)                         ] for j in letters]
```

```
pairs = []
for j in letters:
    temp = []


        temp.append((i, j))
    pairs.append(temp)
```

Swap numbers and letters

```python
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
```

```python
pairs = [[(i, j) for i in numbers] for j in letters]
print(pairs)
```

```
[
    [(1, 'a'), (2, 'a'), (3, 'a')],
    [(1, 'b'), (2, 'b'), (3, 'b')],
    [(1, 'c'), (2, 'c'), (3, 'c')]
]
```

Swap numbers and letters

```python
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
```

```python
pairs = [[(i, j) for i in letters] for j in numbers]
print(pairs)
```

```
[
    [('a', 1), ('b', 1), ('c', 1)],
    [('a', 2), ('b', 2), ('c', 2)],
    [('a', 3), ('b', 3), ('c', 3)]
]
```

Difference between list comprehensions

```
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
```

```
pairs = [(i, j) for i in numbers for j in letters]
```

```
pairs = [[(i, j) for i in numbers] for j in letters]
```

```
pairs = [[(i, j) for i in letters] for j in numbers]
```

Let's practice!

PRACTICING CODING INTERVIEW QUESTIONS IN PYTHON

What is a zip object?

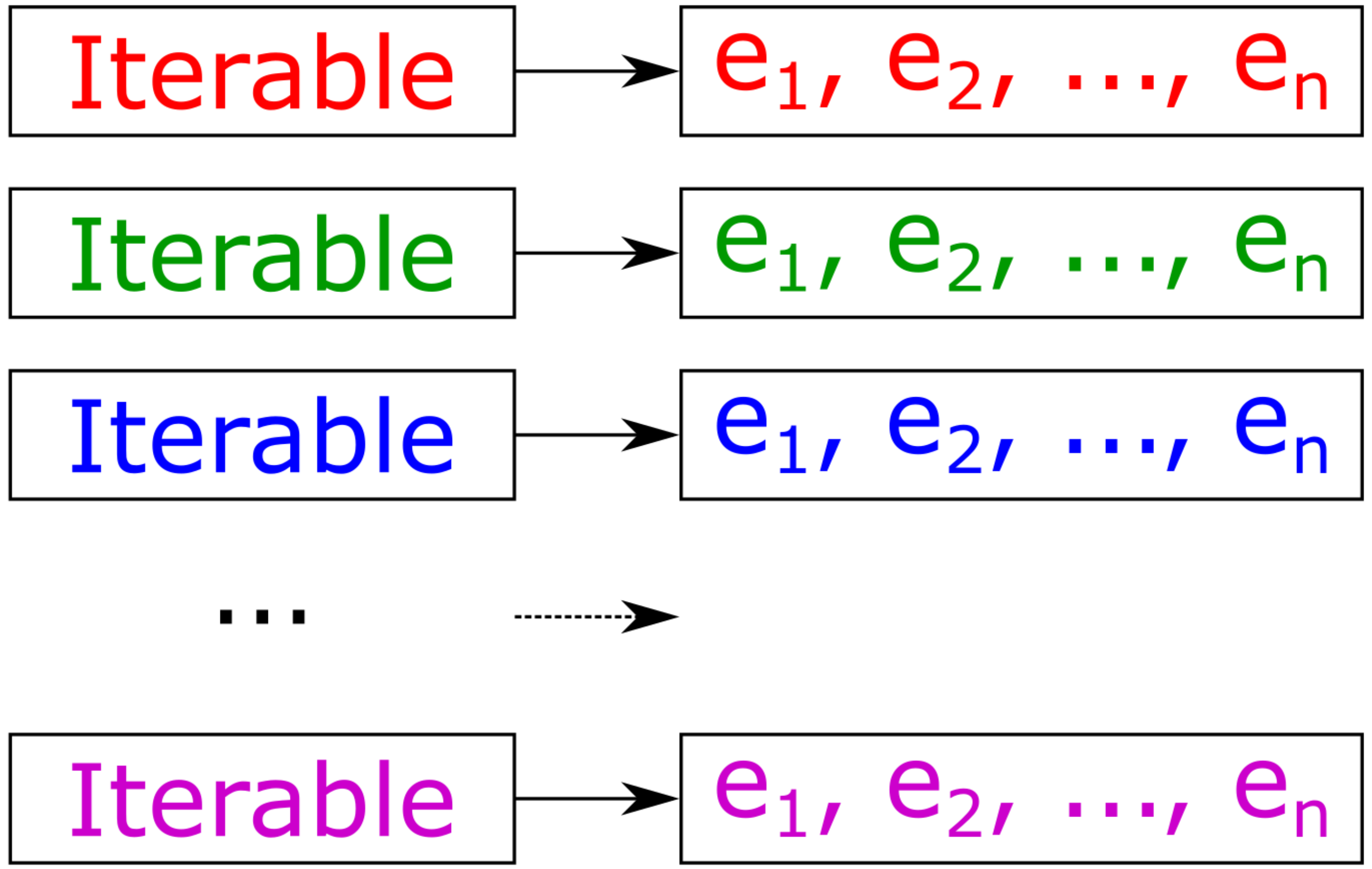
PRACTICING CODING INTERVIEW QUESTIONS IN PYTHON

Kirill Smirnov
Data Science Consultant, Altran

datacamp

Definition

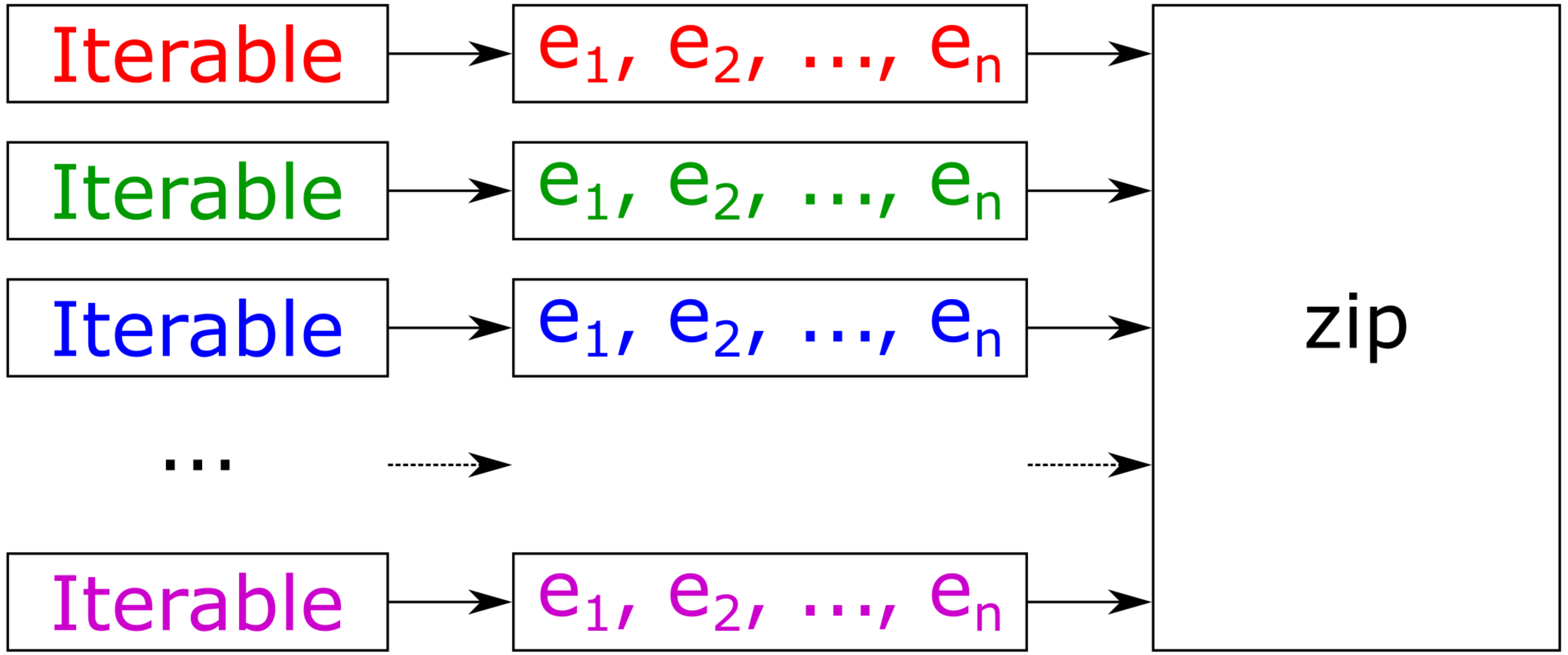
`zip` - object that combines several iterable objects into one iterable object.



e_i - an element from an Iterable

Definition

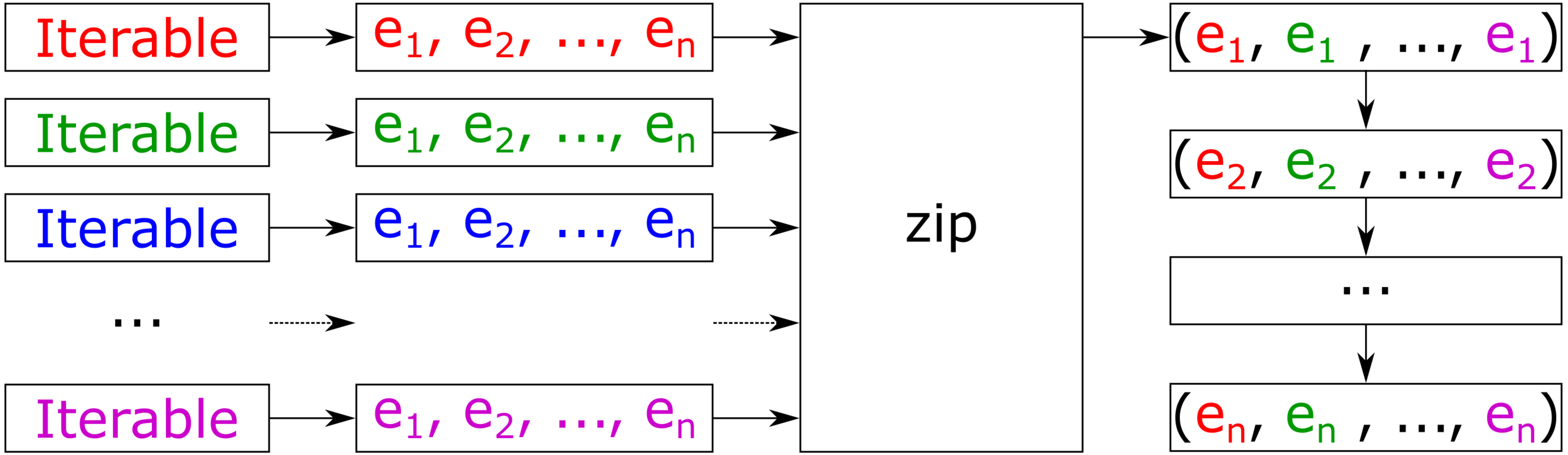
`zip` - object that combines several iterable objects into one iterable object.



e_i - an element from an Iterable

Definition

`zip` - object that combines several iterable objects into one iterable object.



e_i - an element from an Iterable

Example

```python
title = 'TMNT'
villains = ['Shredder', 'Krang', 'Bebop', 'Rocksteady']
turtles = {
    'Raphael': 'Sai', 'Michelangelo': 'Nunchaku',
    'Leonardo': 'Twin katana', 'Donatello': 'Bo'
}
```

```python
result = zip(title, villains, turtles)
print(result)
```

```
<zip object at 0x7f37bab6e608>
```

Traversing through a zip object

```
result = zip(title, villains, turtles)
```

```
for item in result:
    print(item)
```

```
('T', 'Shredder', 'Raphael')
('M', 'Krang', 'Michelangelo')
('N', 'Bebop', 'Leonardo')
('T', 'Rocksteady', 'Donatello')
```

Returning a list of tuples

```
result = zip(title, villains, turtles)
```

```
tuples = list(result)
print(tuples)
```

```
[
    ('T', 'Shredder', 'Raphael'), ('M', 'Krang', 'Michelangelo'),
    ('N', 'Bebop', 'Leonardo'), ('T', 'Rocksteady', 'Donatello')
]
```

zip object as Iterator

```python
result = zip(title, villains, turtles)
```

```python
next(result)
```

```
('T', 'Shredder', 'Raphael')
```

```python
next(result)
```

```
('M', 'Krang', 'Michelangelo')
```

```python
next(result)
```

```
('N', 'Bebop', 'Leonardo')
```

```python
next(result)
```

```
('T', 'Rocksteady', 'Donatello')
```

```python
next(result)
```

```
StopIteration
```

zip object is expendable

```
result = zip(title, villains, turtles)
```

```
for item in result:
    print(item)
```

```
('T', 'Shredder', 'Raphael')
('M', 'Krang', 'Michelangelo')
('N', 'Bebop', 'Leonardo')
('T', 'Rocksteady', 'Donatello')
```

zip object is expendable

```python
result = zip(title, villains, turtles)
```

```python
for item in result:
    print(item)
```

```
('T', 'Shredder', 'Raphael')
('M', 'Krang', 'Michelangelo')
...
```

```python
for item in result:
    print(item)
```

```
# nothing
```

```python
result = zip(title, villains, turtles)
```

```python
tuples = list(result)
print(tuples)
```

```
[
    ('T', 'Shredder', 'Raphael'),
    ('M', 'Krang', 'Michelangelo'),
    ('N', 'Bebop', 'Leonardo'),
    ('T', 'Rocksteady', 'Donatello')
]
```

'zip' object is expendable

```python
result = zip(title, villains, turtles)
```

```python
for item in result:
    print(item)
```

```
('T', 'Shredder', 'Raphael')
('M', 'Krang', 'Michelangelo')
...
```

```python
for item in result:
    print(item)
```

```
# nothing
```

```python
result = zip(title, villains, turtles)
```

```python
tuples = list(result)
print(tuples)
```

```
[
    ('T', 'Shredder', 'Raphael'),
    ...
```

```python
tuples = list(result)
print(tuples)
```

```
[]
```

Unequal Iterable sizes

```
title = 'TMNT'
villains = ['Shredder', 'Krang', 'Bebop', 'Rocksteady']
turtles = {
    'Raphael': 'Sai', 'Michelangelo': 'Nunchaku',
    'Leonardo': 'Twin katana', 'Donatello': 'Bo'
}
```

Unequal Iterable sizes

```python
title = 'Teenage Mutant Ninja Turtles'
villains = ['Shredder', 'Krang', 'Bebop', 'Rocksteady']
turtles = {
    'Raphael': 'Sai', 'Michelangelo': 'Nunchaku',
    'Leonardo': 'Twin katana', 'Donatello': 'Bo'
}
```

```python
result = zip(title, villains, turtles)
```

Traversing through the 'zip' object

```
result = zip(title, villains, turtles)
```

```
for item in result:
    print(item)
```

```
('T', 'Shredder', 'Raphael')
('e', 'Krang', 'Michelangelo')
('e', 'Bebop', 'Leonardo')
('n', 'Rocksteady', 'Donatello')
```

Reverse operation

```python
turtle_masks = [
    ('Raphael', 'red'), ('Michelangelo', 'orange'),
    ('Leonardo', 'blue'), ('Donatello', 'purple')
]
```

```python
result = zip(*turtle_masks)
print(result)
```

```
[
    ('Raphael', 'Michelangelo', 'Leonardo', 'Donatello'),
    ('red', 'orange', 'blue', 'purple')
]
```

Unequal tuple sizes

```python
turtle_masks = [
    ('Raphael', 'red'), ('Michelangelo', 'orange'),
    ('Leonardo', 'blue', 'cyan'), ('Donatello', 'purple', 'magenta')
]
```

```python
result = zip(*turtle_masks)
print(result)
```

```
[
    ('Raphael', 'Michelangelo', 'Leonardo', 'Donatello'),
    ('red', 'orange', 'blue', 'purple')
]
```

Relation to a dictionary

A `zip` object can be used to create a dictionary

```
keys = ['movie', 'year', 'director']
values = [
    ['Forest Gump', 'Goodfellas', 'Se7en'],
    [1994, 1990, 1995],
    ['R.Zemeckis', 'M.Scorsese', 'D.Fincher']
]
```

```
movies = dict(zip(keys, values))
```

```
print(movies)
```

```
{
    'director': [
        'R.Zemeckis',
        'M.Scorsese',
        'D.Fincher'
    ],
    'movie': [
        'Forest Gump',
        'Goodfellas',
        'Se7en'
    ],
    'year': [1994, 1990, 1995]
}
```

Creating a DataFrame

```python
import pandas as pd


df_movies = pd.DataFrame(movies)
```

```python
print(df_movies)
```

```
          director        movie  year
0  Robert Zemeckis  Forest Gump  1994
1  Martin Scorsese   Goodfellas  1990
2    David Fincher        Se7en  1995
```

`list()`

Creating a DataFrame

```
import pandas as pd


df_movies = pd.DataFrame(movies)
```

```
print(df_movies)
```

```
           director        movie  year
0  Robert Zemeckis  Forest Gump  1994
1  Martin Scorsese   Goodfellas  1990
2    David Fincher        Se7en  1995
```

`list()` → `zip()`

Creating a DataFrame

```python
import pandas as pd


df_movies = pd.DataFrame(movies)
```

```python
print(df_movies)
```

```
          director         movie  year
0  Robert Zemeckis  Forest Gump  1994
1  Martin Scorsese   Goodfellas  1990
2    David Fincher        Se7en  1995
```

`list()` $\rightarrow$ `zip()` $\rightarrow$ `dict()`

Creating a DataFrame

```python
import pandas as pd


df_movies = pd.DataFrame(movies)
```

```python
print(df_movies)
```

```
           director         movie  year
0  Robert Zemeckis  Forest Gump  1994
1  Martin Scorsese   Goodfellas  1990
2    David Fincher        Se7en  1995
```

`list()` → `zip()` → `dict()` → `DataFrame()`

Let's practice!

PRACTICING CODING INTERVIEW QUESTIONS IN PYTHON

What is a generator and how to create one?

PRACTICING CODING INTERVIEW QUESTIONS IN PYTHON

Kirill Smirnov
Data Science Consultant, Altran

datacamp

Definition

Generator - a special iterable object created by a function having a `yield` keyword in its body.

```python
def func():
    # Return a value from super complex calculations
    return 0
```

```python
result = func()
print(result)
```

```
0
```

Definition

Generator - a special iterable object created by a function having a `yield` keyword in its body.

```python
def func():
    # Yield a value from super complex calculations
    yield 0
```

```python
result = func()
print(result)
```

```
<generator object result at 0x105736e10>
```

Generator as Iterable

```python
def func():
    # Yield a value from super complex calculations
    yield 0

result = func()
```

```python
for item in result:
    print(item)
```

```
0
```

More yields!

```python
def func():
    yield 0
    yield 1
    yield 2
```

```python
result = func()
for item in result:
    print(item)
```

```
0
1
2
```

Yield in a loop

```python
def func(n):
    for i in range(0, n):
        yield 2*i
```

```python
result = func(3)
for item in result:
    print(item)
```

```
0
2
4
```

Converting a generator to a list

```
def func(n):
    for i in range(0, n):
        yield 2*i

result = func(5)
```

```
list(result)
```

```
[0, 2, 4, 6, 8]
```

Generator as Iterator

Generator is an Iterable AND an Iterator

```
def func(n):
    for i in range(0, n):
        yield 2*i
```

```
result = func(3)
```

```
next(result)
```

```
0
```

```
next(result)
```

```
2
```

```
next(result)
```

```
4
```

```
next(result)
```

```
StopIteration
```

Generators are expendable

```python
def func(n):
    for i in range(0, n):
        yield 2*i
```

```python
result = func(3)
```

```python
for item in result:
    print(item)
```

```
0
2
4
```

```python
for item in result:
    print(item)
```

```
# nothing
```

```python
result = func(3)
for item in result:
    print(item)
```

```
0
2
4
```

Generators are expendable

```
def func(n):
    for i in range(0, n):
        yield 2*i
```

```
result = func(3)
list(result)
```

```
[0, 2, 4]
```

```
list(result)
```

```
[]
```

```
result = func(3)
list(result)
```

```
[0, 2, 4]
```

Generator comprehension

```python
result = [2*i for i in range(0, 3)]
print(result)
```

```
[0, 2, 4]
```

```python
result = (2*i for i in range(0, 3))
print(result)
```

```
<generator object result at 0x105736e10>
```

Traversal

```
result = (2*i for i in range(0, 3))
```

```
for item in result:
    print(item)
```

```
0
2
4
```

```
next(result)
```

```
StopIteration
```

Why generators?

- simple way to create a custom iterable object

```
[1, 3, 2, 4, 3, 5]
```

```python
def create_jump_sequence(n):
    for i in range(1, n-1):
        yield i
        yield i+2
```

```python
jump_sequence = create_jump_sequence(5)
list(jump_sequence)
```

```
[1, 3, 2, 4, 3, 5]
```

Why generators?

- simple way to create a custom iterable object
- lazy initialization

```
[1, 3, 2, 4, 3, 5, 4, 6, 5, 7, ...]
```

```python
def create_jump_sequence(n):
    for i in range(1, n-1):
        yield i
        yield i+2
```

```python
jump_sequence = create_jump_sequence(500)
next(jump_sequence)
```

```
1
```

Why generators?

- simple way to create a custom iterable object
- lazy initialization
- possibility to create infinite iterable objects

```
def create_inf_generator():
    while True:
        yield 'I am infinite!'
```

```
inf_generator = create_inf_generator()
```

```
next(inf_generator)
```

```
I am infinite
```

Let's practice!

PRACTICING CODING INTERVIEW QUESTIONS IN PYTHON