Iterating over data with Python

Jordi Cortadella and Jordi Petit
Department of Computer Science

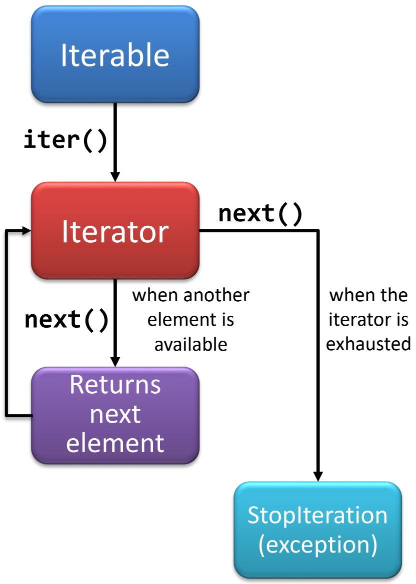
Outline

- Iterables and iterators
- Generators
- Comprehensions
- Enumerate and Zip
- Map, Filter and Reduce
- λ-functions

ITERABLES AND ITERATORS

Iterables

- ***Iterables*** are containers of data in which we can iterate to obtain elements one by one
- Lists, tuples, sets, dictionaries, strings, etc. are iterables
- ***Iterators*** are objects used to iterate over iterables
- Two important functions:
  - **`iter()`**: creates an iterator from an iterable
  - **`next()`**: returns the next item



Iterable

`iter()`

Iterator

`next()`

when another element is available

Returns next element

`next()`

when the iterator is exhausted

StopIteration (exception)

Iterables: example

```
>>> lst = [1, 2, 3]  # lst is an iterable
>>> it = iter(lst)  # it is an iterator
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

`next()` raises a StopIteration exception when no more items are available

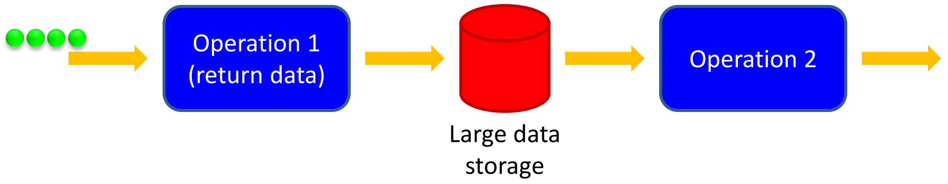
`next(it, default)`

```
it = iter(some_iterable)  # Creates an iterable

# next(it, default) does not raise any exception.
# Instead, it returns the default value.
v = next(it, None)
while v is not None:
    do_something(v)
    v = next(it, None)


# Equivalent code
for v in it:
    do_something(v)
```

GENERATORS

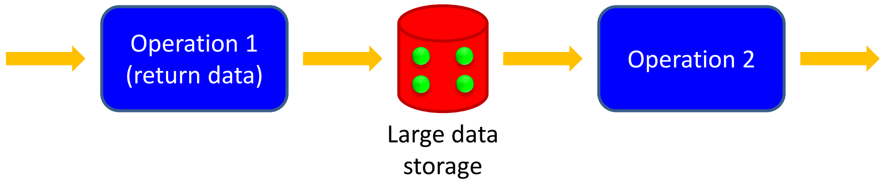
Designing data pipelines

- Big data systems often have to process long streams of data with pipelines chaining different operations.
- How to store the data between operations?



Designing data pipelines

- Big data systems often have to process long streams of data with pipelines chaining different operations.
- How to store the data between operations?



Generators

- A mechanism to avoid storage of large amounts of data.
- Generators are lazy iterators that do not store the whole data structures in memory.

Generators: example

```
>>> def natural_numbers():
...     n = 0
...     while True:
...         yield n
...         n += 1
...
>>> gen = natural_numbers()
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
3
...
```

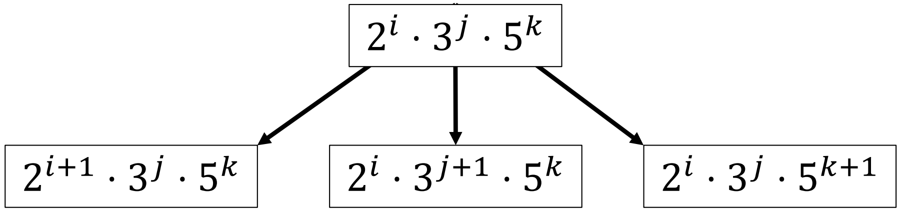
```
>>> for i in natural_numbers():
...     if is_prime(i):
...         print(i)
...
2
3
5
7
11
...
672131
672137
672143
^CTraceback (...)
...
KeyboardInterrupt
>>>
```

Hamming numbers

- Hamming numbers are those numbers whose only prime divisors are 2, 3, and 5.
- Examples:
  - 20 is a Hamming number ($2^2 \cdot 3^0 \cdot 5^1$)
  - 21 is not a Hamming number ($3 \cdot 7$)
- Exercise: design a program that prints the n smallest Hamming numbers.
- Strategy: for every Hamming number $2^i \cdot 3^j \cdot 5^k$, we can generate three new numbers by increasing each one of the exponents.
- How to generate them in ascending order?

Hamming numbers

- Generating the next Hamming numbers:



$2^i \cdot 3^j \cdot 5^k$

$2^{i+1} \cdot 3^j \cdot 5^k$ $2^i \cdot 3^{j+1} \cdot 5^k$ $2^i \cdot 3^j \cdot 5^{k+1}$

- How to generate them in ascending order?
  - Use a priority queue to store the pending numbers
- How to avoid repetitions?
  - Remember the last delivered number

Hamming numbers: simulation

Value	Priority Queue
	1
1	**2** **3** **5**
2	3 **4** 5 **6** **10**
3	4 5 6 **6** **9** 10 **15**
4	5 6 6 **8** 9 10 **12** 15 **20**
5	6 6 8 9 10 **10** 12 15 **15** 20 **25**
6	6 8 9 10 10 12 **12** 15 **18** 20 25 **30**
6	8 9 10 10 12 12 15 18 20 25 30
8	9 10 10 12 12 15 **16** 18 20 **24** 25 30 **40**
9	10 10 12 12 15 16 18 **18** 20 24 25 **27** 30 40 **45**
10	10 12 12 15 16 18 18 20 **20** 24 25 27 30 **30** 40 45 **50**
⋮	⋮

Hamming numbers

```python
import heapq
from typing import Iterator


def hamming_numbers() -> Iterator[int]:
    """Generates all Hamming numbers in ascending order"""
    pq = [1]  # priority queue storing Hamming numbers
    prev = 0  # the last delivered number
    while True:
        value = heapq.heappop(pq) # get the smallest number
        if value != prev:  # avoid repetitions
            yield value  # deliver the value and wait (lazy)
            prev = value
            for x in 2, 3, 5: # generate new numbers
                nxt = x*value
                heapq.heappush(pq, nxt)
```

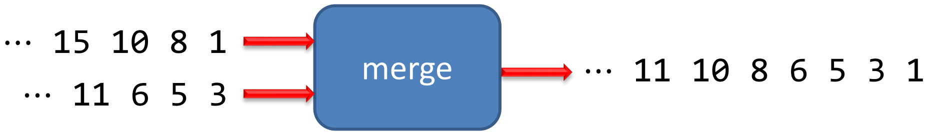
Hamming numbers

```python
def main(n: int) -> None:
    """Test to print first n hamming numbers"""
    hammings = hamming_numbers() # the generator
    for _ in range(n):
        print(next(hammings))


if __name__ == '__main__':
    main(20)
```

```
1
2
3
4
5
6
8
9
10
12
15
16
18
20
24
25
27
30
32
36
```

Merging sequences

- Functions can receive iterators as parameters and generate iterators as results
- Let us design a function that merges two sorted iterators and generated a sorted iterator



Merging sequences: typing

```
from typing import Iterator, TypeVar, Protocol

# This is an abstract class that contains the __lt__
# operator (<). No need to implement it.
class Comparable(Protocol):
    def __lt__(self: 'T', other: 'T') -> bool: ...

# This is a generic type. The bound attribute indicates
# that the type must contain the operators of Comparable.
T = TypeVar('T', bound=Comparable)

# The merge function dealing with sequences of elements
# that are "Comparable" (i.e., the type contains the
# the operator <).
def merge(a: Iterator[T], b: Iterator[T]) -> Iterator[T]:
    """reads two sorted iterators and generates a sorted
       iterator by merging them"""
    ...
```

Merging sequences: code

```
def merge(a: Iterator[T], b: Iterator[T]) -> Iterator[T]:
    """reads two sorted iterators and generates a sorted
       iterator by merging them"""
    x, y = next(a, None), next(b, None)
    while x is not None and y is not None:
        if x < y:
            yield x
            x = next(a, None)
        else:
            yield y
            y = next(b, None)

    if x is not None:
        yield x
        yield from a  # delivers values from another iterator

    if y is not None:
        yield y
        yield from b
```

COMPREHENSIONS

Comprehensions

- Set builder notation. Example:

$$S = \{x^2 \mid x < 1000, x \text{ is prime}\}$$

- Conventional Python using a `for` loop:

```
s = {}
for x in range(1000):
    if is_prime(x):
        s.add(x**2)
```

- Using comprehensions:

```
s = {x**2 for x in range(1000) if is_prime(x)}
```

Nested comprehensions

```
# Given a list of words, create a dictionary with the key-value
# pairs <word: number of vowels>
words = ['cat', 'kangaroo', 'lion', 'dog', 'hippopotamus']

# We can use s.count(x) to count the number of occurrences
# of x in the string s, e.g., 'kangaroo'.count('o') is 2
vowels = {w: sum(w.count(x) for x in 'aeiou') for w in words}
print(vowels)
{'cat': 1, 'kangaroo': 4, 'lion': 2, 'dog': 1, 'hippopotamus': 5}

# Let us print a list of the words with more than 3 vowels
print([w for w in vowels.keys() if vowels[w] > 3])
['kangaroo', 'hippopotamus']
```

Creating matrices with comprehensions

```
# Let us create a 4x4 identity matrix
matrix = [[0]*4]*4
for i in range(4):
    matrix[i][i] = 1

# Surprise! What’s wrong?
print(matrix)
[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]

# Let us use comprehensions
matrix = [[1 if i==j else 0 for j in range(4)] for i in range(4)]
print(matrix)
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]

# How to create a zero matrix with n rows and m columns
matrix = [[0]*m for _ in range(n)]
```

Generator expressions: example

```
>>> import sys
>>> # A list comprehension
>>> squares_lc = [i**2 for i in range(10**6)]
>>> # It generates a long list (larger than 8Mb)
>>> sys.getsizeof(squares_lc)
8448728
>>>
>>> # But we can also create a generator using (…)
>>> squares_gc = (i**2 for i in range(10**6))
>>> sys.getsizeof(squares_gc)
104
>>> # and we can iterate over the generator
>>> for n in squares_gc:
...     if is_prime(n+1):
...         print(n+1)
...
```

ENUMERATE AND ZIP

enumerate

```
# Different ways of printing indices and values
lst = [x**2 for x in range(100)]

for i in range(len(lst)):
    print(i, lst[i])

i = 0
for v in lst:
    print(i, v)
    i += 1

for i, v in enumerate(lst):
    print(i, v)

# It also works for generators!
lst_gen = (x**2 for x in range(100))
for i, v in enumerate(lst_gen):
    print(i, v)
```

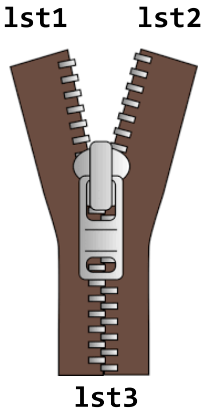
Hamming numbers

```
def main(n: int) -> None:
    """Test to print first n hamming numbers"""
    for i, x in enumerate(hamming_numbers()):
        if i == n:
            break
        print(x)


if __name__ == '__main__':
    main(20)
```

```
1
2
3
4
5
6
8
9
10
12
15
16
18
20
24
25
27
30
32
36
```

zip

```
>>> # Zipping lists
>>> lst1 = [x**2 for x in range(100)]
>>> lst2 = [2*x for x in range(100)]
>>> lst3 = zip(lst1, lst2)
>>> # lst3 is an iterator!
>>> print(lst3)
<zip object at 0x7fd8d8beacc0>
>>> for x, y in lst3:
...     print(x, y)
...
0 0
1 2
4 4
9 6
16 8
25 10
36 12
49 14
```



zipping and unzipping

```
>>> # Let us zip two lists
>>> letters = ['a', 'b', 'c', 'd']
>>> numbers = [1, 2, 3, 4]
>>> ln_zip = zip(letters, numbers)
>>> list_ln = list(ln_zip)
>>> print(list_ln)
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
# Now we can unzip the list of tuples
>>> lett, numb = zip(*list_ln)
>>> print('letters =', lett)
letters = ('a', 'b', 'c', 'd')
>>> print('numbers =', numb)
numbers = (1, 2, 3, 4)
```

MAP, FILTER AND REDUCE

map, filter and reduce

- Loops, comprehensions and generators are techniques used to process data in iterable objects.
- The functions **`map()`**, **`filter()`** and **`reduce()`** provide a functional programming approach to achieve similar goals.
- They can be applied to any iterable object (list, tuple, set, …)
- These functions can provide a very elegant solution to compute expressions like this:

$$\sum_{0 \le i < n,\ \mathrm{is\_prime}(i)} i^2$$

map, filter, reduce: auxiliary functions

```
def square(x: int) -> int:
    return x*x

def add(x: int, y: int) -> int:
    return x + y

def is_prime(n: int) -> bool:
    if n <= 1:
        return False
    d = 2
    while d*d <= n:
        if n%d == 0:
            return False
        d += 1
    return True
```

map

```
lst = [1, 2, 3, 4, 5, 6]

# map creates an iterator that applies a function
# to all elements of the iterable object
result = map(square, lst)

list(result)

# Output: [1, 4, 9, 16, 25, 36]
```

---

filter

```
# filter creates an iterator that selects the
# elements that satisfy the filtering condition
result = filter(is_prime, range(30))

list(result)

# Output: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

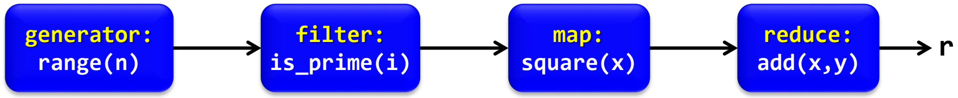
---

reduce

```
from functools import reduce

# reduce visits all elements and executes a
# function that "accumulates" their values
result = reduce(add, range(10))
result
# Output: 45

# An initial value can also be specified
result = reduce(add, range(10), 5)
result
# Output: 50
```

Back to our problem

$$\sum_{0\le i<n,\ \text{is\_prime}(i)} i^2$$



generator: range(n)

filter: is_prime(i)

map: square(x)

reduce: add(x,y)

r

```
n = 10**7
r = reduce(add, map(square, filter(is_prime, range(n))))
print(r)
# Output: 21113978675102768574
```

Important: no intermediate lists are generated. Very low storage is required (< 1000 bytes). By using lists to store the intermediate results, about 400Mb of storage would be required.

Pythonic Boolean reductions

`any()` and `all()` are particular cases of reduce functions with Boolean results.

```
>>> numbers = [2, 3, 7, 11, 13, 23]
>>> all(is_prime(x) for x in numbers)
True
>>> all(x%2 == 1 for x in numbers)
False
>>> any(6 < x < 12 for x in numbers)
True
>>> words = ['cat', 'kangaroo', 'lion', 'dog', 'hippopotamus']
>>> all(len(w) > 10 for w in words)
False
>>> any(len(w) > 10 for w in words)
True
>>> all(len(w) < 15 for w in words)
True
>>> any(w[0] == 'h' for w in words)
True
```

Pythonic numerical reductions

`sum()`, `min()` and `max()` are particular cases of reduce functions with numerical parameters.

```
>>> numbers = [5, 8, -2, 6, 0]
>>> sum(numbers)
17
>>> sum(numbers, start=10)  # with an initial value of the sum
27
>>> min(numbers)
-2
>>> max(numbers)
8
>>> sum(numbers)/len(numbers)  # mean
3.4
>>> a, b = [3, -4, 2], [1, 3, -1]  # two vectors
>>> sum([x*y for x, y in zip(a, b)])  # dot product of a and b
-11
>>> # a more memory-efficient solution, why?
>>> sum(x*y for x, y in zip(a, b))
-11
```

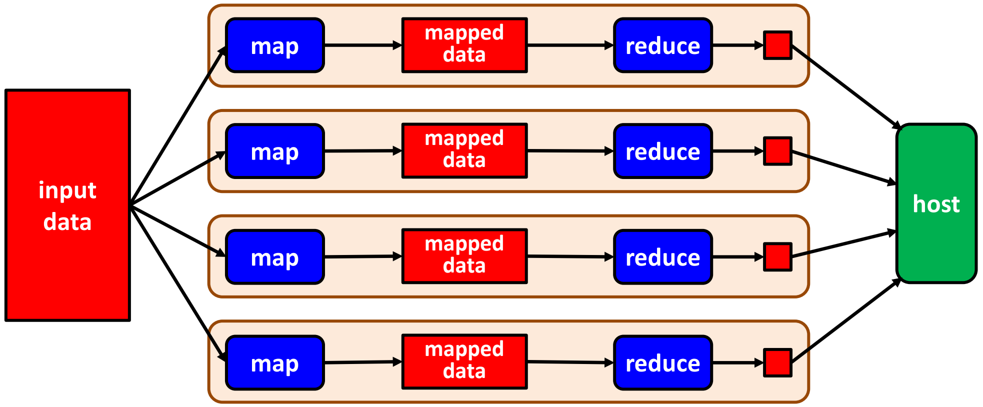
Generate or list?

```
import random, sys, time

n = 200_000_000
a = [random.random() for _ in range(n)]
b = [random.random() for _ in range(n)]

tinit = time.perf_counter()
gen = (x*y for x,y in zip(a,b))
sum_gen = sum(gen)
time_gen = time.perf_counter() - tinit
size_gen = sys.getsizeof(gen)

tinit = time.perf_counter()
lst = [x*y for x,y in zip(a,b)]
sum_lst = sum(lst)
time_lst = time.perf_counter() - tinit
size_lst = sys.getsizeof(lst)

print(f'time: gen={time_gen:.2f} secs, list={time_lst:.2f} secs')
print(f'memory: gen={size_gen} bytes, list={size_lst:,} bytes')
```

```
time: gen=13.51 secs, list=12.32 secs
memory: gen=104 bytes, list=1,693,045,240 bytes
```

MapReduce

A programming model for big data sets using parallel, distributed algorithms



λ-FUNCTIONS

λ-functions

- Lambda functions are anonymous functions that receive parameters and return expressions
- Syntax:

```
lambda parameters: expression
```

is equivalent to:

```
def anonymous(parameters):
    return expression
```

- Examples:

```
    lambda x: x*x
    lambda x, y: x+y
```

Using λ-functions in map/filter/reduce

$$\sum_{0 \leq i < n,\ \text{is\_prime}(i)} i^2$$

```
r = reduce(add, map(square, filter(is_prime, (range(n)))))

# Using λ-functions
r = reduce(lambda x, y: x+y,
           map(lambda x: x*x, filter(is_prime, range(n)))
           )

# Using generators
r = sum(i*i for i in range(n) if is_prime(i))
```

Conclusions

- Processing long streams of data is one of the main tasks of big data systems. Memory storage is one of the critical resources
- When designing data pipelines exploit lazy evaluation mechanisms to generate data upon demand and avoid unnecessary data storage

EXERCISES

Comprehensions

- Create a dictionary where the keys are the numbers of a list and the values are the highest one-digit divisor of each number
- Create a list with all positive numbers smaller than n that are divisible by some number included in a list called `divisors`
- Given a rectangular matrix (list of lists), calculate its transpose using list comprehensions

---

Generating the Fibonacci series

- Design a generator of the Fibonacci series
- Given a list of divisors, design a generator that generates the Fibonacci numbers that are divisible by all divisors of the list. Example:

```
divisors = [3, 5, 7, 11]
gen = (…)  # design the generator
for x in gen:
    print(x)

Output:
0
102334155
23416728348467685
5358359254990966640871840
1226132595394188293000174702095995
...
```

Intersection of sequences

Implement the function intersect with the following specification:

```
from typing import Iterator, TypeVar, Protocol

class Comparable(Protocol):
    def __lt__(self: 'T', other: 'T') -> bool: ...

T = TypeVar('T', bound=Comparable)

def intersect(a: Iterator[T], b: Iterator[T]) -> Iterator[T]:
    """reads two sorted iterators and generates a sorted
       iterator with only the common elements"""
    ...

# Example:
#    a = [1, 3, 5, 5, 6, 7, 10, 13, 16, 18]
#    b = [2, 5, 5, 8, 13, 13, 15, 16, 20]
#    output: [5, 13, 16]
```

Farey sequence

- The Farey sequence of order n is the sequence of completely reduced fractions between 0 and 1 with denominators less than or equal to n, arranged in ascending order. Example:

$$F_5 = \left\{\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}\right\}$$

- Design the generator **`farey(n)`** that generates the Farey sequence of order n:

```
def farey(n: int) -> Iterator[tuple[int, int]]:
```

- Write Python expressions to calculate:
  - the sum of the elements of F_n
  - the number of elements of F_n

Hint: The next element of the Farey sequence can be calculated using only the two previous elements (find the rule in Wikipedia!)

Filter/reduce pipeline

Design two versions of the following function using a filter-reduce pipeline:

- One version with auxiliary functions
- One version with lambda functions

```
from dataclasses import dataclass
from typing import Iterable
from functools import reduce

@dataclass
class Person:
    name: str
    age: int
    salary: float

def avg_salary(people: Iterable[Person],
               min_age: int, max_age: int) -> float:
    """Returns the average salary of the people with age
       between min_age and max_age"""
```