Effective Python Functional Programming

Techniques, Tools and Practices

丁来强

About me

- Splunk Lab @Shanghai
- 10+ years working experiences
- Mainly used C++/Python and JS

Contact me

- Email: wjo1212 at 163.com
- Wechat: LaiQiangDing

Outline

- FP Overview
  - Definition, Benefit, Python Support
- Pure first-class function
  - First-Class, No side effect, Persistence
- High Order Functions
  - Map, Reduce, Group By, Compose, Partial, Currying
- Recursion Technology
  - TR, TRO, Trampoline, Memorize
  - Overloading, Pattern Matching
- Laziness: Stream, Iterators
- Parallel Related

Functional Programming Overview

Functional Languages

- 1930: **lambda**
- 1950: **LISP**
- Nowadays:
  - **Clojure**: morden LISP for FP
  - **Scala**: with FP for JVM
  - **F#**: for .net
  - mainly others: **Haskell**, **Erlang**

What is Functional Programming

- A declarative paradigm
- Procedural/OO programming (C++/Java etc.)

Procedural Style

```
L = []
for x in xrange(10):
    L.append(x*x)
```

← OOP

FP Style

```
L = [x*x for x in xrange(10)]
or
L = map(lambda x: x*x, xrange(10))
```

A more complex example

- Calculate partially invalid string with operations:

- expr = "28+32+++32++39" ==> 28+32+32+39

Imperative style

- Actions that change state from initial state to result

```python
res = 0
for x in expr.split("+"):
    if x != "":
        res += int(x)

print(res)
```

```
"28+32+++32++39", 0
"28", 0
"32", 28
"", 60
"", 60
"32", 60
"", 92
"39", 92
131
```

Functional style

- Apply transformation (and compositions)

```
from operator import add
from functools import reduce, map, filter
reduce(add, map(int, filter(bool, expr.split("+"))))
```

```
"28+32+++32++39"
["28","32","","","32","","39"]
["28","32","32","39"]
[28,32,32,39]
131
```

FP Characters

- Function as Data, First-class
- Higher-order functions
- Purity: Avoid state, no side effect
- Immutable data structures
- Strong compliers
- Tail Recursion eliminations
- laziness, pattern matching, monads…

Python is IP with FP features…

FP Support in Python (or Libs)

- ✓ Function as Data, First-class
- ✓ Higher-order functions
- ✓ Purity: Avoid state, no side effect
- ✓ Immutable data structures
- ❑ Strong compliers
- ✓ Tail Recursion eliminations (Trampolines)
- ✓ laziness, pattern matching, monads…

No Strong Complier?

Strong Compiler

- Many features of pure functional programming languages are designed for compile optimization:
  - strong assumptions
  - no apply to Python

The BDFL said…

- I have never considered Python to be heavily influenced by functional languages, no matter what people say or think. I was much more familiar with imperative languages such as C and Algol 68 and although I had made functions first-class objects, I didn't view Python as a functional programming language. However, earlier on, it was clear that users wanted to do much more with lists and functions.

---- **Guido van Rossum (2009.4)**

http://python-history.blogspot.com/2009/04/origins-of-pythons-functional-features.html

Even though…FP style is still awesome!

Why FP

- More testable and reliable
- Better at streaming and parallel processing
- Maybe subjective points:
  - A totally different way to approach problems
    - Often provides new, clean, elegant solutions
  - Functional programmers report productivity increases
  - Less Code, Readable?

Your partial codes are probably already FP style...

Pure first-class function

Function: First Class Citizen

```python
funs = [len, max, min, map]
```

```python
sorted(funs, key=lambda f: f.__name__)
```

```python
def outer():
    def inner(data):
        pass
    return inner
```

```python
f = lambda: 10
f.data = 20
```

```python
f2 = f
```

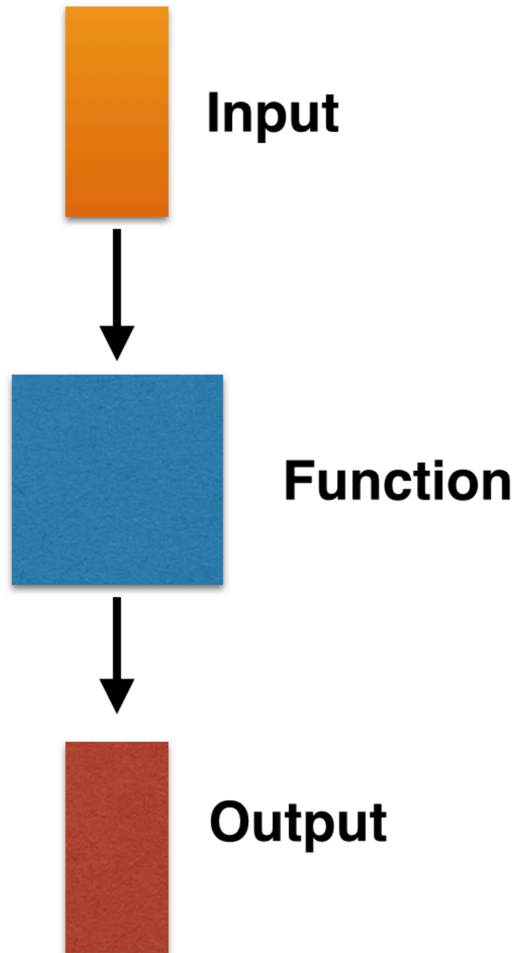
```python
from operator import *
1 + 2 == add(1, 2)
"abc"[1] == itemgetter(1)("abc")
```

Scala/Boost style lambda

```
from fn import _

map(_ * 2, xrange(10))

reduce(_ + _, xrange(10))

sorted([('p', 10), ('y', 5), ('c', 8)],
    key=_[1])
```

It's a object of a Function which implements all kinds of Python Object Protocols and return new function object

Pure Function



- No State: A ==> B
- No Side Effect
- Immutability

Side effect and sate

```
def process_list(lst):
    del lst[-1]
    lst.reverse()
    return lst
```

Input is changed

```
call_count = 0
def process_list(lst):
    call_count += 1
    return lst[-2::-1]
```

outside state is changed

Side effect and state

```
def process_list(lst):
    return lst[-2::-1]
```

No side effect?

```
input = ([1],[2],[3])
output = process_list(input)
output[0].append(2)
print(input)
```

input is changed

```
import copy
def process_list2(lst):
    return copy.deepcopy(lst[-2::-1])
```

Immutable Data != Persistent Data

- Immutable type in Python: tuple, frozenset
- Persistent data structure :
  - always preserves the previous version of itself when it is modified.

```
from pyrsistent import freeze

v1 = freeze([1, 2, 3])
v2 = v1.append(4)
v3 = v2.append(5)
```

```
v1:  [1,2,3]
v2:  [1,2,3,4]
v3:  [1,2,3,4,5]
```

Persistent Data in Pyrsistent.py

- Change data inside a persistent dict

```
news_paper = freeze(
    {'articles': [{'author': 'Sara',
                   'content': 'A short article'},
                  {'author': 'Steve',
                   'content': 'A slightly longer article'}],
     'weather': {'temperature': '11C', 'wind': '5m/s'}})

short_news = news_paper.transform(
    ['articles', ny, 'content'],
    lambda c: c[:10] + '...' if len(c) > 10 else c)
```

```
{
'articles': [ {'content': 'A short ar...',
               'author': 'Sara'},
              {'content': 'A slightly...',
               'author': 'Steve'}],
'weather' : {'temperature': '11C', 'wind': '5m/s'}
}
```

Purify function with Pysistent.py

```
from pyrsistent import mutant

@mutant
def process_list(lst):
    return lst.append(10)
```

```
lst1 = [1, 2, 3]
lst2 = process_list(lst1)
lst3 = lst2.append(4)
```

```
lst1: [1,2,3]
lst2: [1,2,3,10]
lst3: [1,2,3,10,4]
```

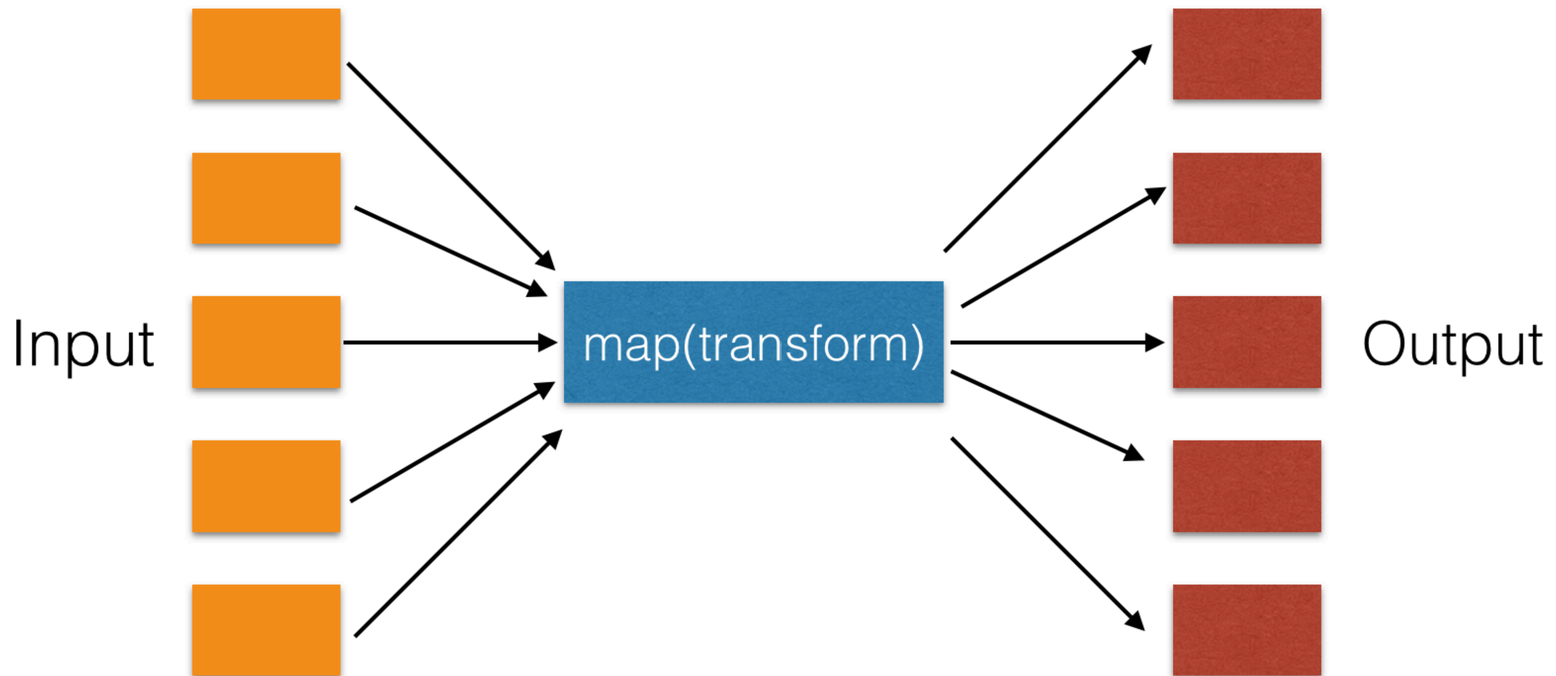
High Order Functions

Higher-order functions

- Higher-order functions:
  - Functions accept or return functions
- Python Support:
  - map, filter, groupby, reduce
  - functools module
  - list comprehension
  - decorators

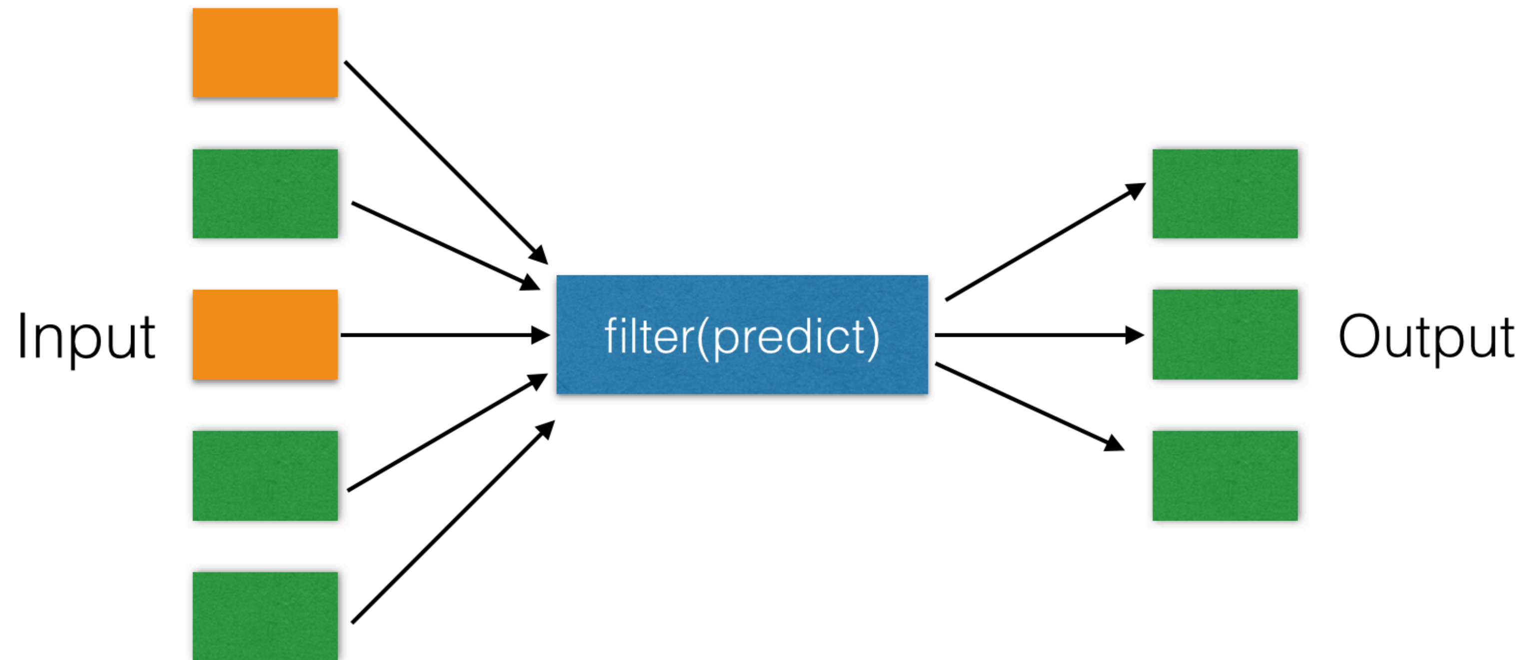
Map

```
map(_*2, xrange(4)) ==> [1, 2, 6, 8]
```



Filter

```
filter(_ % 2 == 0, xrange(6)) ==> [0, 2, 4]
```



List Comprehension

- Python’s answers to “map”, “filter”

```
map(_*2, xrange(10))
==>
[x*2 for x in xrange(10)]   # list
{x*2 for x in xrange(10)}   # set
{x:x*2 for x in xrange(10)} # dict
(x*2 for x in xrange(10))   # generator
```

```
filter(_ % 2 == 0, xrange(10))
==>
[x for x in xrange(10) if x%2==0]
{x for x in xrange(10) if x%2==0}
```

Another case

```
map(_*_, xrange(10), xrange(10))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- How to do this in list comprehension?

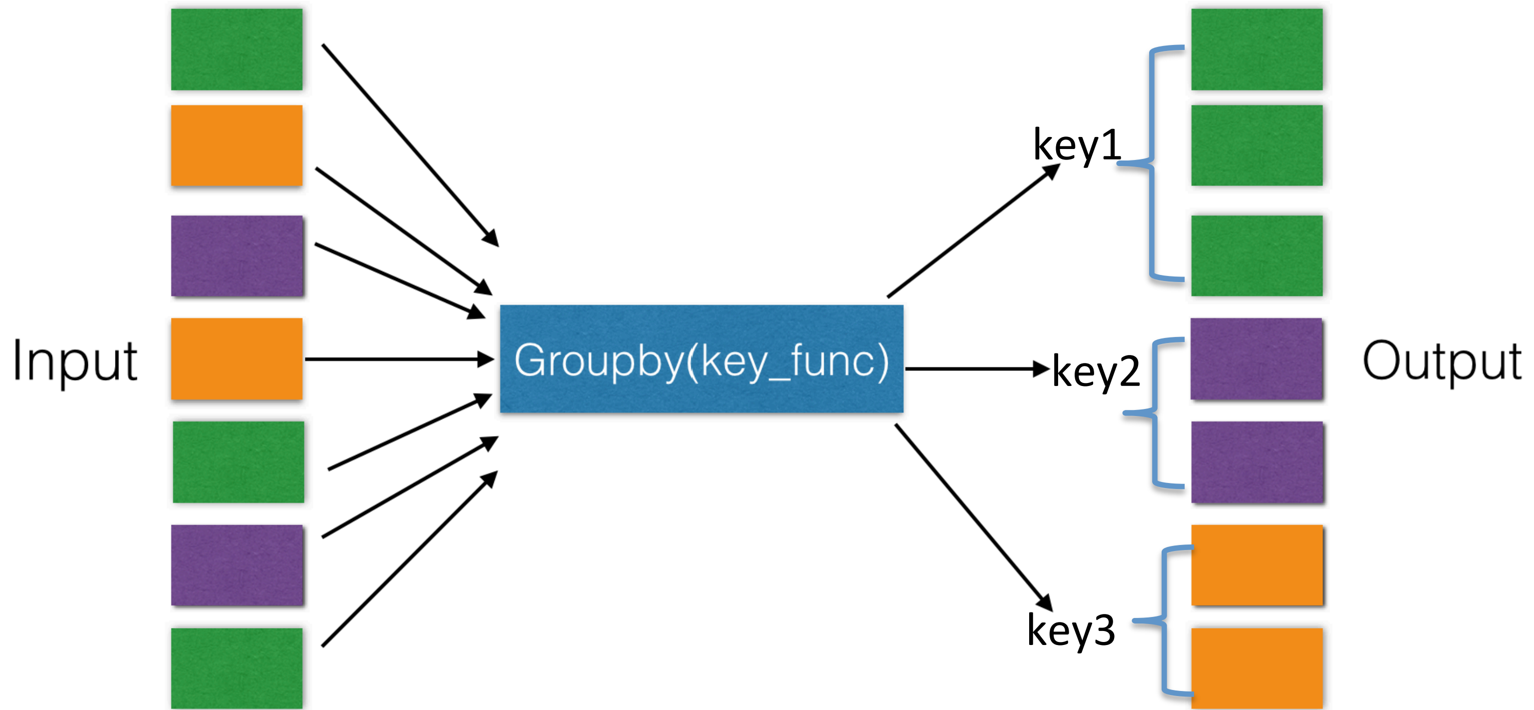
```
[x*y for x, y in zip(xrange(10), xrange(10))]
```

list comprehension characters

- Pros:
  - Hide the logic of generation for different kinds of structures: tuple, list, set, dict
  - Handy and powerful when combing with iterators
  - Immutable for first layer
- Cons:
  - Strange for new learners: action-loop-if
  - Imperative style inside: break the modularity

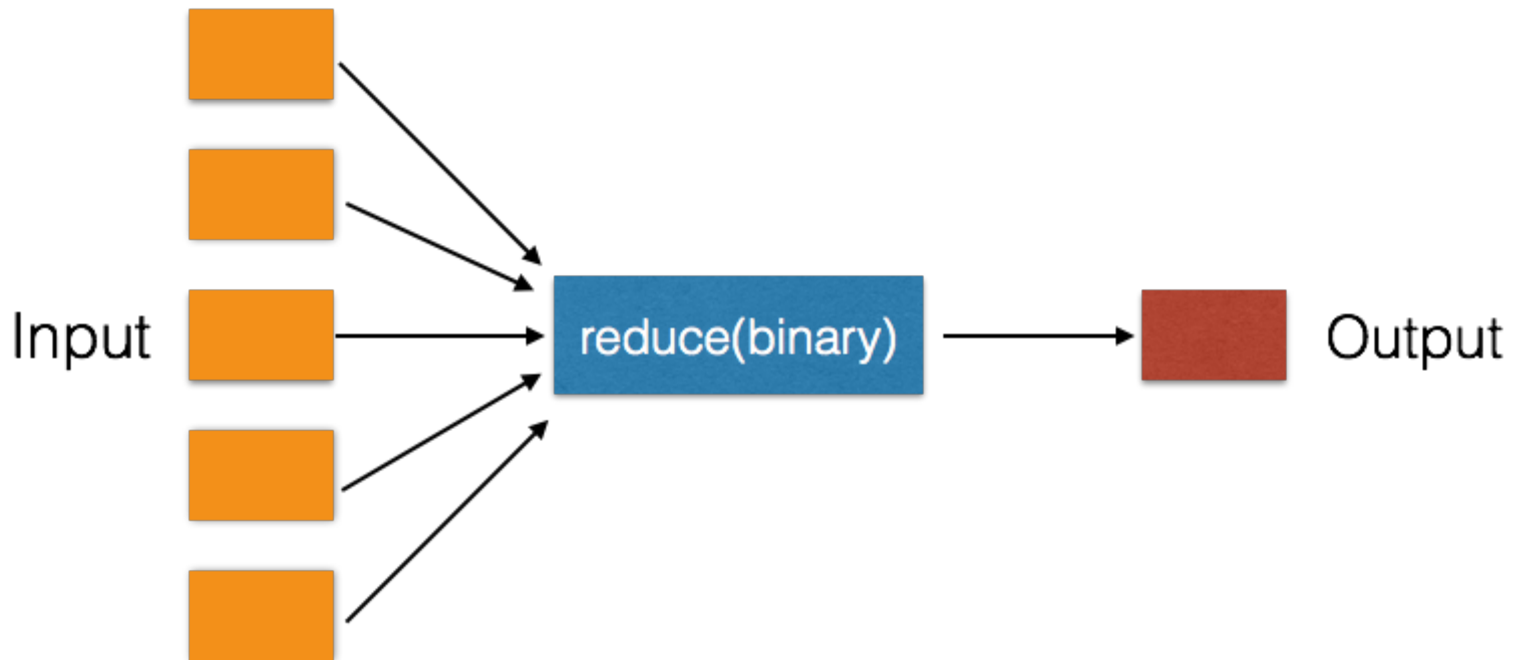
Groupby

```
groupby(xrange(6), _%3) ==> {0: [0,3], 1:[1,4], 2:[2,5]}
```



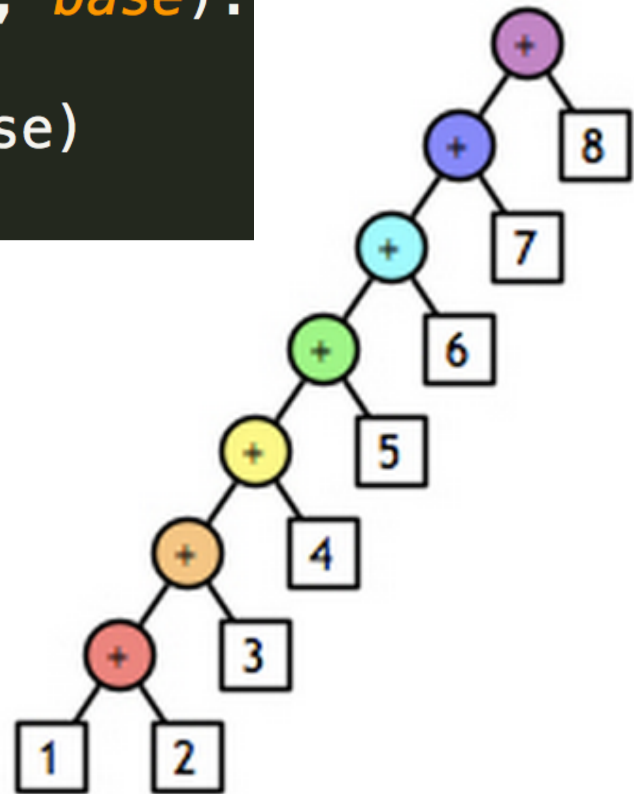
Reduce

```
reduce(_+_, xrange(10)) == 1+2+3+... = 45
```



Reduce

```python
def reduce(func, iterable, base):
    for x in iterable:
        base = func(x, base)
    return base
```



Sum, Multiple

```python
def sum(iterable):
    acc = 0
    for x in iterable:
        acc = acc + x
    return acc
```

```python
def muli(iterable):
    acc = 1
    for x in iterable:
        acc = acc * x
    return acc
```

```python
sum(xrange(10))
==> reduce(operator.add, xrange(10), 0)
```

```python
multi(xrange(1, 10))
==> reduce(operator.mul, xrange(1, 10), 1)
```

A simple problem

- Write a sorted2 with default reverse sorting

```
[3, 2, 1] == sorted2([2, 1, 3])
[1, 2, 3] == sorted2([2, 1, 3], reverse=False)
['000', '12', 'x'] == sorted2(['12', 'x', '000'],
                              key=len)
```

```
def sorted2(*args, **kwargs):
    if "reverse" not in kwargs:
        kwargs["reverse"] = True
    return sorted(*args, **kwargs)
```

Better way using Partial

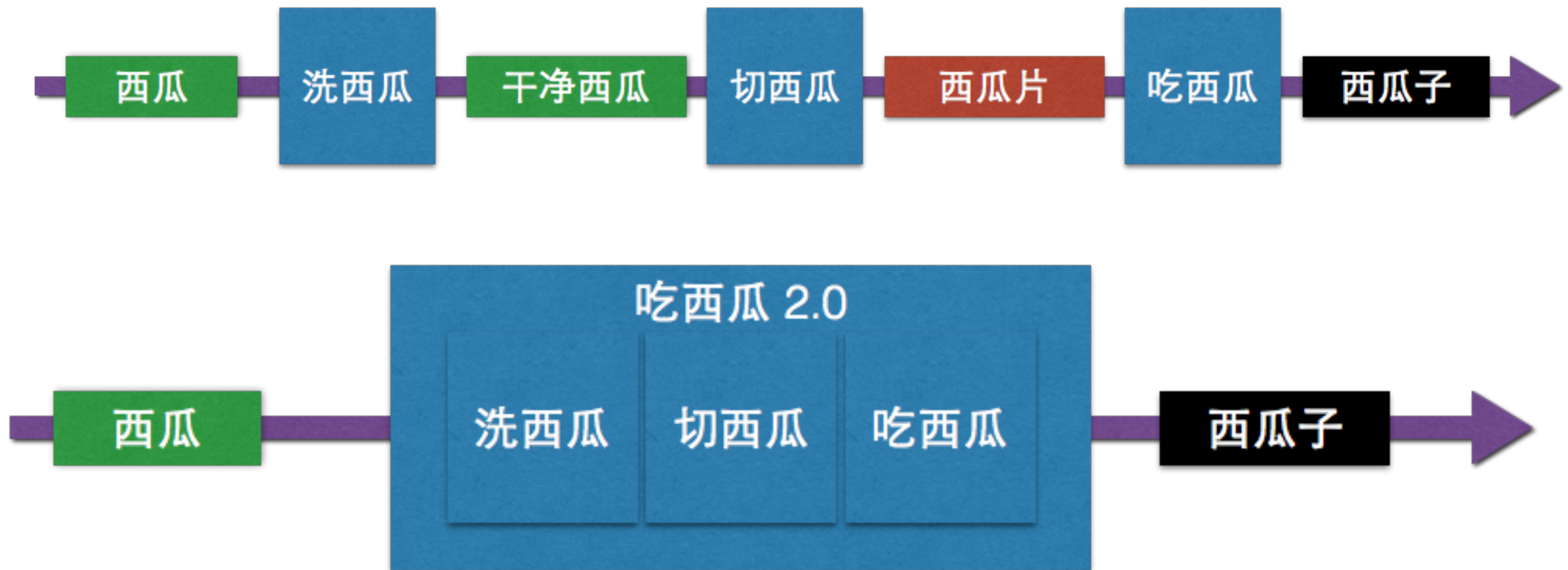
```
from functools import partial
sorted2 = partial(sorted, reverse=True)
```

```
sum = partial(reduce, operator.add)
mul = partial(reduce, operator.mul)
```

```
add_10 = partial(add, 10)
add_10(20) == add(10, 20)
```

Compose

- Chain multiple functions into one



Compose

```
from toolz.functoolz import compose
```

```
吃西瓜v2 = compose(吃西瓜, 切西瓜, 洗西瓜)
吃西瓜v2(西瓜) == 吃西瓜(切西瓜(洗西瓜(西瓜)))
```

```
add_2_multi_4 = compose(partial(mul, 4),
                        partial(add, 2))
```

```
add_2_multi_4(1) == 12    # (1+2)*4
add_2_multi_4(4) == 24    # (4+2)*4
```

<u>Compose</u> promotes scalable and modular software

A little bit more complex example

- How to sum of all even number inside the string?

- s1 = "12x3y45z67t89” => 2+4+6+8 => 20

Solution

```
sum(int(x) for x in s1 if x.isdigit() and int(x) % 2 == 0)
```

```
from fn import _
sum(filter(_%2==0, map(int, filter(str.isdigit, s1))))
```

```
sum_even = compose(sum, partial(filter, _%2==0),
                        partial(map, int),
                        partial(filter, str.isdigit))
```

```
sum_even(s1)
sum_even("as12a234sdf45z67t89")
```

Currying

- An extension to partial

```
from toolz.functoolz import curry
         from fn.func import curried as curry
@curry
def sum5(a, b, c, d, e):
    return a + b + c + d + e

assert 15 == sum5(1)(2)(3)(4)(5)
assert 15 == sum5(1, 2, 3)(4, 5)
assert 15 == sum5(1, 2, 3, 4, 5)
```

Refine the solution

```
sum_even = compose(sum, partial(filter, _%2==0),
                        partial(map, int),
                        partial(filter, str.isdigit))
```

Lots of “partial” is a indicator of currying

```
from toolz.curried import *

sum_even = compose(sum,
                   filter(_%2==0),
                   map(int),
                   filter(str.isdigit))
```

All those functions are already curried

Refine the solution (2)

```
sum_even = compose(sum, partial(filter, _%2==0),
                        partial(map, int),
                        partial(filter, str.isdigit))
```

Another way to compose a high order function

```
from fn import F, _
sum_even = F() << sum << F(filter, _%2==0)
               << F(map, int)
               << F(filter, str.iisdigit)
```

Also work for >>

Parameterized Decorator

- A powerful tool to make high order function

```
@login_check(must_login=False)
def login_page(login):
    pass
```

```
@login_check(must_login=True)
def pay_page():
    pass
```

Parameterized Decorator

- Common coding idioms…

```
def login_check(must_login=True):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            if must_login and not is_login():
                raise ValueError("...")
            return function(*args, **kwargs)
        return wrapper
    return decorator
```

Annoying 3 levels' nesting

Currying free decorators

- Directly add parameters to decorators

```
@curry
def login_check(function, must_login=True):
    def wrapper(*args, **kwargs):
        if must_login and not is_login():
            raise ValueError("...")
        return function(*args, **kwargs)
    return wrapper
```

Recursion Technology

Recursion in Pure Functional Language

- Some pure functional language, has no loop
- Recursion is used instead
- Tail Recursion eliminations is supported

Why?

- Clearly show what to do, rather than how

```
def fib(n):
    if n <= 1:
        return 1
    elif n == 2:
        return 1
    else:
        return fib(n-1)
        + fib(n-2)
```

$$F_n = F_{n-1} + F_{n-2}$$

Recursion makes logic clearer

- How to flatten a list?

```
[1, 2, 3] ==> [1, 2, 3]
[1, [2], 3] ==> [1, 2, 3]
[1,[2],[[3,[4]]]] ==> [1,2,3,4]
```

```
def flatten(L):
    if isinstance(L, Iterable):
        return sum(map(flatten, L), [])
    else:
        return [L]
```

Why Not?

1. Stack overflow

```
import sys
# max recursion depth: 1000
fib(sys.getrecursionlimit())
```

2. duplicated computing

```
 return fib(n-1)
 + fib(n-2)
```

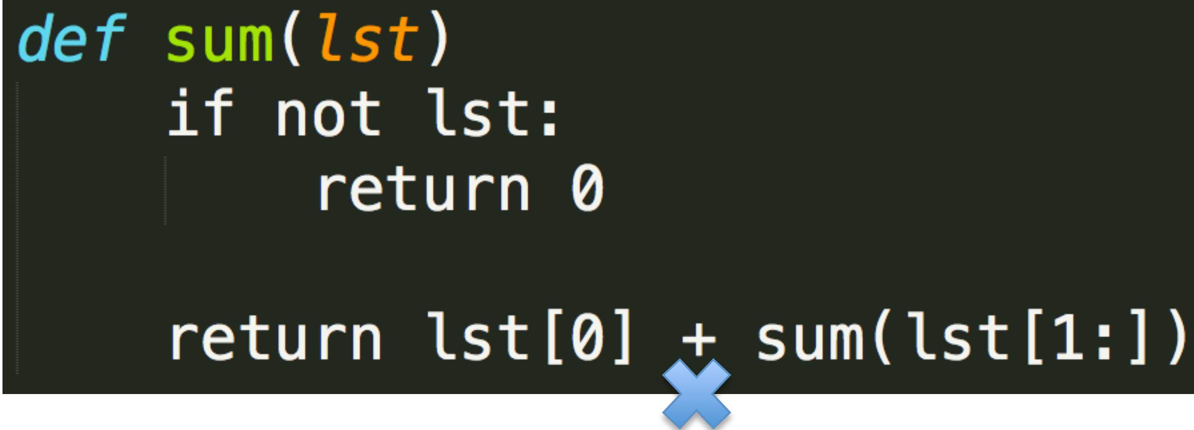
3. Lack of pattern matching or overloading

Tail Recursion Optimization

- Some functional languages can eliminate the recursion by using loop internally if it's tail recursion

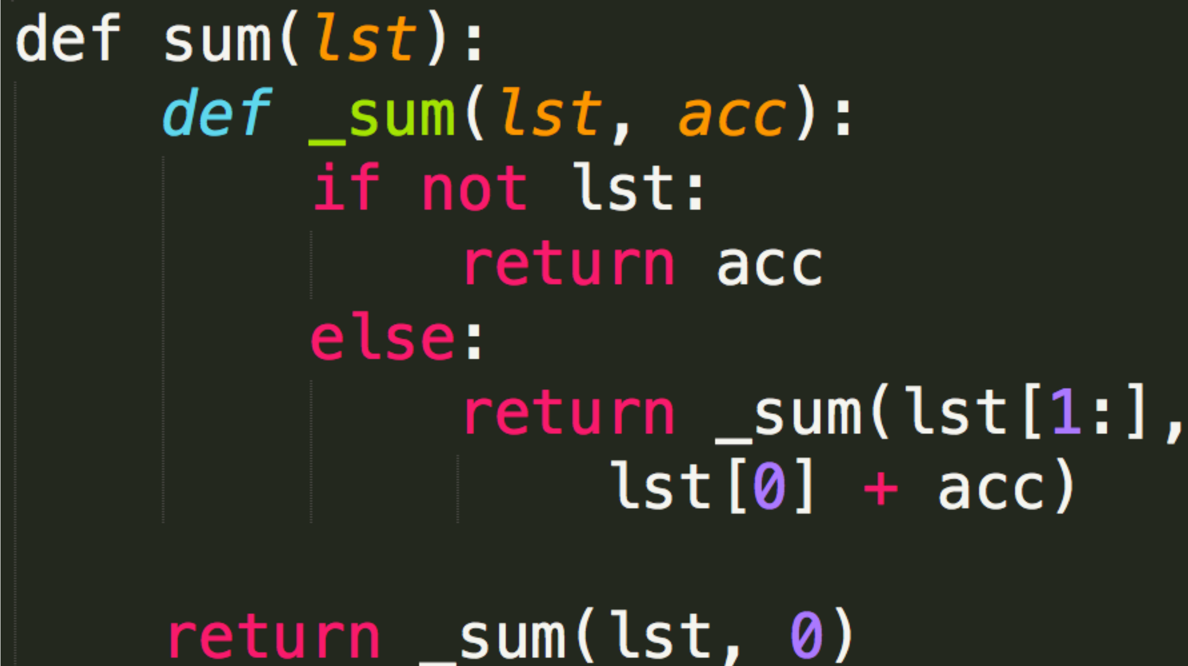
Tail Recursion

- Last call recursively without any evaluation



Change to Tail Recursion

- Normally use a helper function to do so

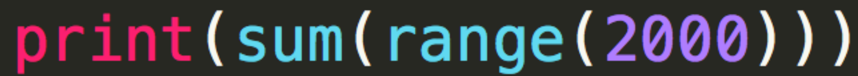


But Python doesn't support TR Optimization…

Trampoline

```python
from fn import recur
```

```python
def sum(lst):
    @recur.tco
    def _sum(lst, acc):
        if not lst:
            return False, acc
        else:
            return True, (lst[1:],
                          lst[0] + acc)

    return _sum(lst, 0)
```



Any solution to solve duplicated computing?

Function Result Cache

```python
from functools import lru_cache
```

```python
@lru_cache(maxsize=100)
def get_pep(num):
    res = '.../pep-%04d/' % num
    with urllib.request.urlopen(res) as s:
        return s.read()
```

```python
get_pep(8)
get_pep(8)
```

Second call will directly use cache

memoize in toolz

- Support cache parameters and function keys

```python
from toolz.functoolz import memoize

@memoize(cache={0: 1, 1: 1})
def fib(n):
    return fib(n-1) + fib(n-2)
```

How about overloading and pattern matching?

Overloading

- Python doesn't support overloading
- However, it has powerful function arguments

```
def func(data1, data2,
         opt1=False, opt2=True,
         *args, **kwargs):
    ...
```

Add overloading to Python

```python
from functools import partial
from multipledispatch import dispatch
dispatch = partial(dispatch, namespace=dict())
```

```python
@dispatch(object, object)
def f(x, y):
    return x + y

@dispatch(str, int)
def f(x, y):
    return int(x) + y

@dispatch(int, str)
def f(x, y):
    return x + int(y)
```

```python
f("1", "2") # f(object, object)
f("1", 1)   # f(str, int)
f(1, "1")   # f(int, str)
```

Overloading makes function modular

```
def flatten(L):
    if isinstance(L, Iterable):
        return sum(map(flatten, L), [])
    else:
        return [L]
```

```
@dispatch(Iterable)
def flatten(L):
    return sum(map(flatten, L), [])

@dispatch(object)
def flatten(x):
    return [x]
```

Pattern Matching

```
from patterns import patterns, Mismatch
```

```
@patterns
def factorial():
    if 0: 1
    if n is int: n * factorial(n-1)
    if []: []
    if [x] + xs: [factorial(x)] + factorial(xs)
    if {'n': n, 'f': f}: f(factorial(n))
```

Haskell style

```
assert factorial(0) == 1
assert factorial(5) == 120
assert factorial([3,4,2]) == [6, 24, 2]
assert factorial({'n': [5, 1], 'f': sum}) == 121
factorial('hello') # raises Mismatch
```

Laziness

Support in Python

- Iterators and laziness are core conventions
  - xrange
  - tuple comprehension
  - itertools module
  - dict.iter* methods
  - generator
  - for-loop protocol

fn.Stream

- fn.Stream provide syntax sugar for iterators

```
from fn import Stream, _

def gen():
    yield 1
    yield 2

s = Stream() << gen
             << (1, 2)
             << xrange(10)

map(print, s[1::2])
```

*generator function,
*Iterable data
*iterators

Support slicing for iterators

Infinite iterator

- Using Generator to make infinite iterator

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

Lazy and infinite stream

- Different way to solve the problem.

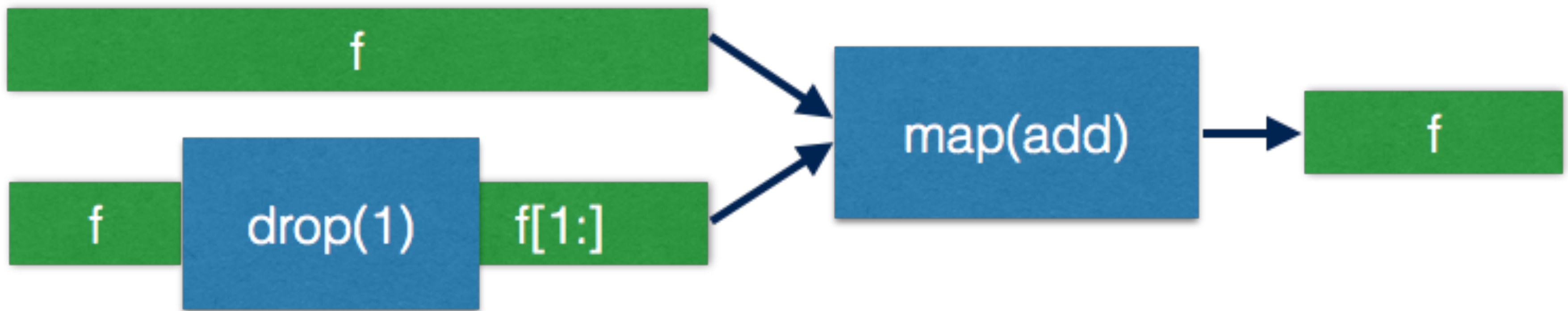
```
from fn import Stream
from fn.iters import take, drop, map
```

```
f = Stream()
fib = f << [0, 1] << map(add, f, drop(1, f))
```

```
list(take(8, fib)) == [0,1,1,2,3,5,8,13]
fib[20] == 6765
```

Infinite stream

```
f = Stream()
fib = f << [0, 1] << map(add, f, drop(1, f))
```



More iterators

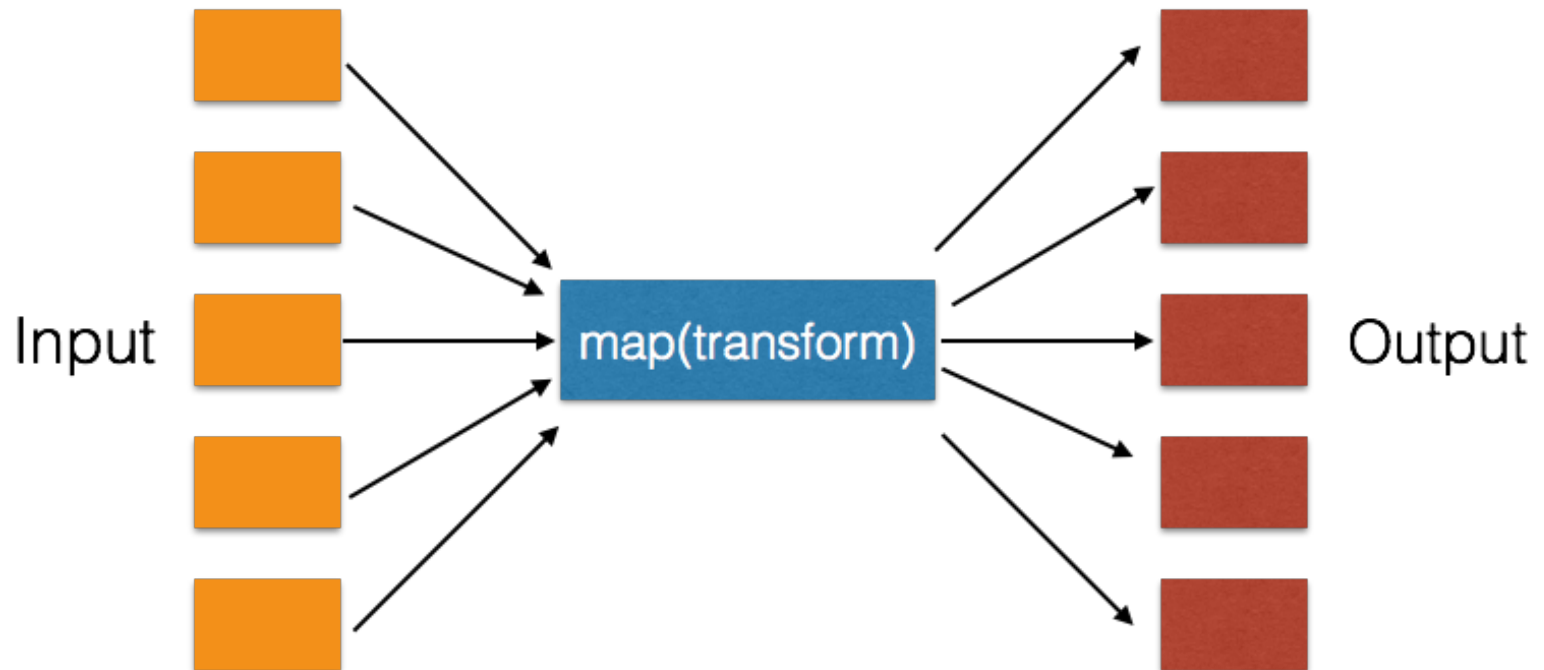
- **toolz.itertoolz** expose itertools and provide certain counts of algorithm basing on stream
  - stats: count, groupby, frequency
  - filter: unique, partition
  - picking: take, drop, first, last, n_th etc.
  - merge_sorted

Parallel

Pure FP is by nature for parallel

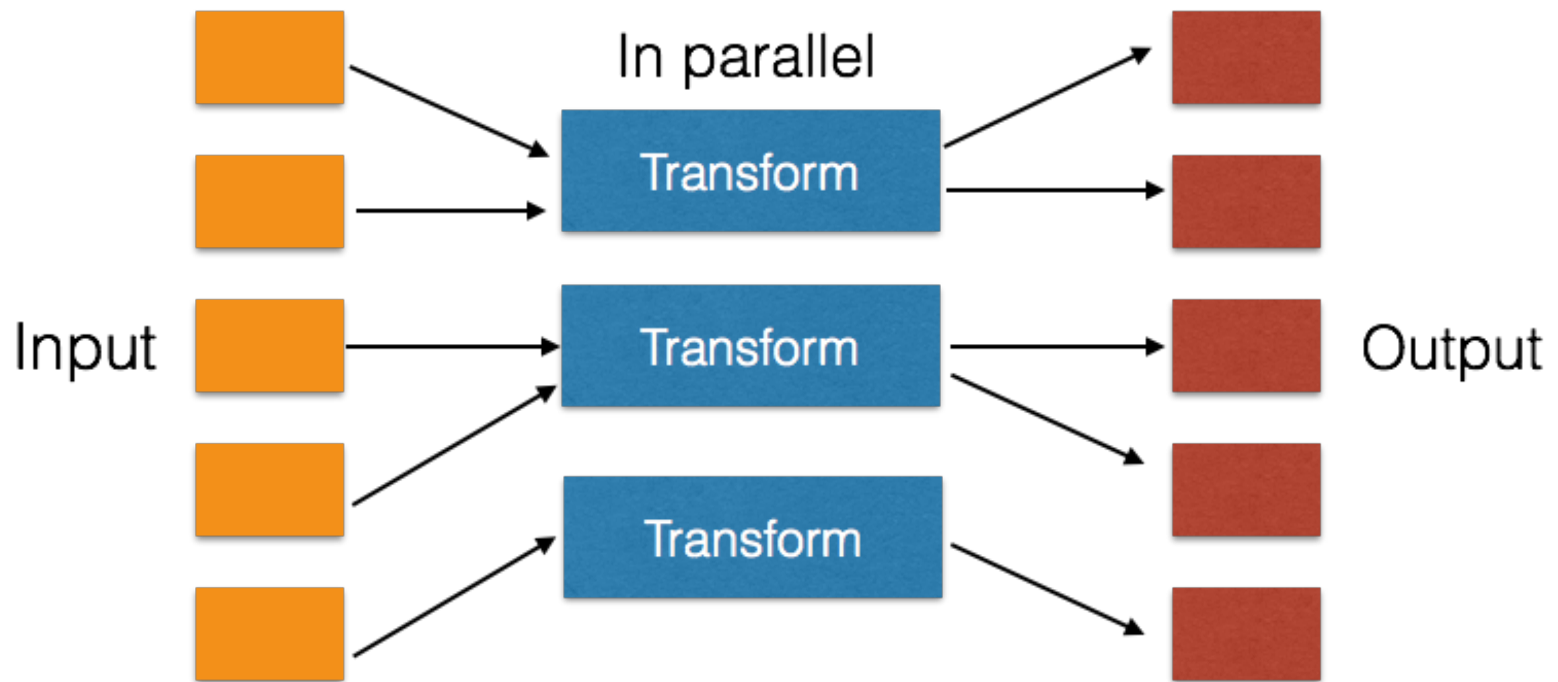
Map

- because function has no side effect in FP



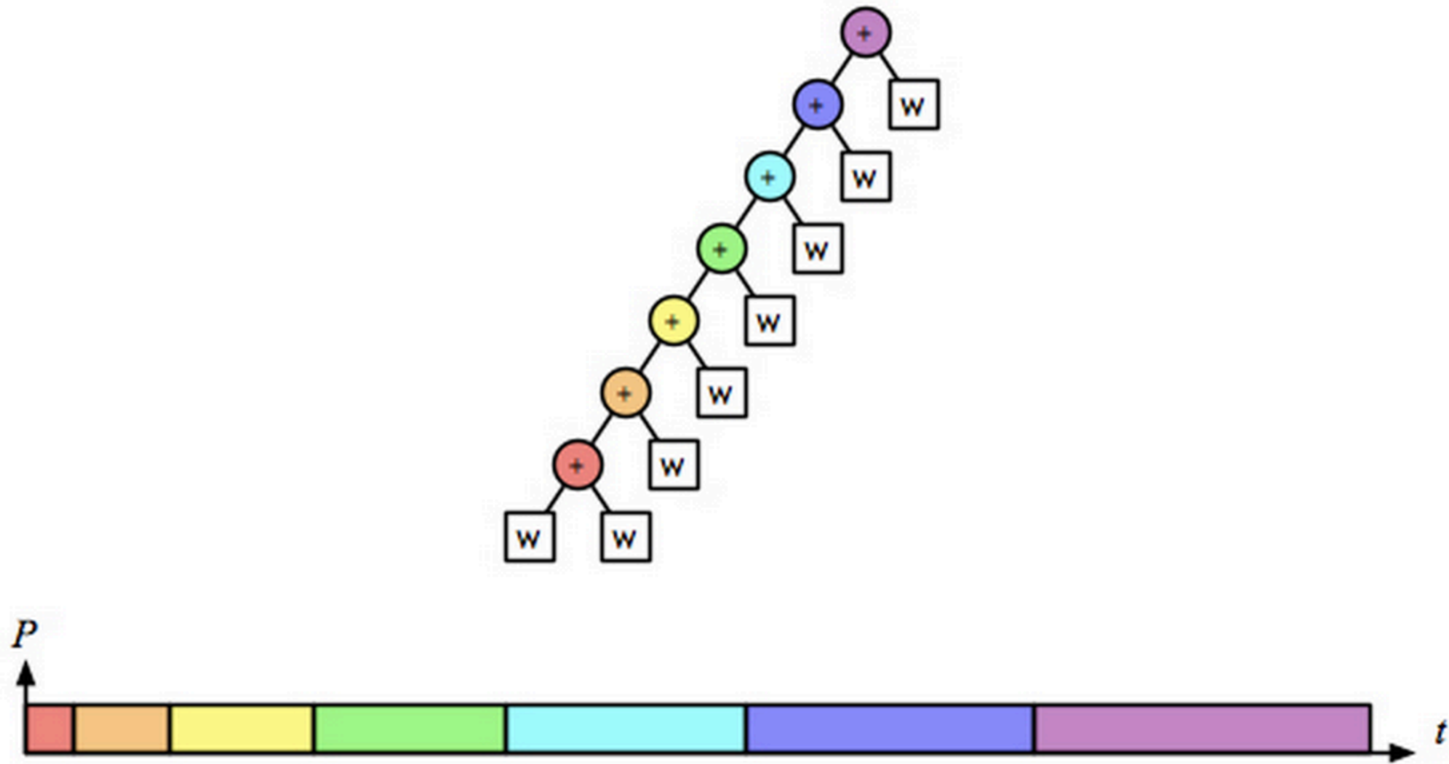
Map

- Scalable by increasing parallel transform unit



Reduce

- There seems a temporary state, however…



Reduce

- Due to no-side effect function, tasks can be done in logarithmic time

MP or Distribution is the only way?

- **Premature optimization is the root of all evil.**
- Besides parallel solution, evaluate different interpreter or JIT: Cython, PyPy, numba etc.
- In some cases, Single Node computing make sense because there're more cores on one node and less overhead comparing to cluster.
- Another trend for Python is using GPUs (e.g. OpenCL, CUDA).
- IO-Bound solution are well discussed a lot.

Python Parallel Overview

- **GIL** makes multi-threading for CPU-bound tasks makes no sense
- built-in **multi-processing** module provides quite high level utilities
- Distribution for common domains, Clustering using **Spark, Hadoop or Celery** are also available.
- For **data science area**, anaconda and blaze based, distributed numpy and Global Array based solution are also quickly developed.

Python Parallel Options

- Built-in MP/RPC
- IPython Cluster
- scikit-learn parallel algorithm
- Python Parallel (lack of Python3 version)
- Celery
- More: joblib, …

Spreading Data

- Closure cannot be pickled by default
- Use dill
- or copy_reg

```python
def f(x):            # x
    def g(y):
        return x+y
    return g

f1 = f(10)
print f1(1) # using x
```

```python
p = Pool(5)
p.map(f1, range(10))
```

IPython Cluster

- Support more data serialization (using dill)

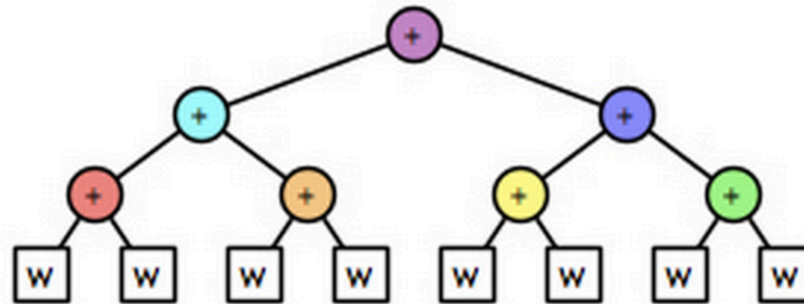
```
from IPython.parallel import Client

def f(x):          # x
    def g(y):
        return x+y
    return g

f1 = f(10)
Client()[:].map(f1, range(10))
```

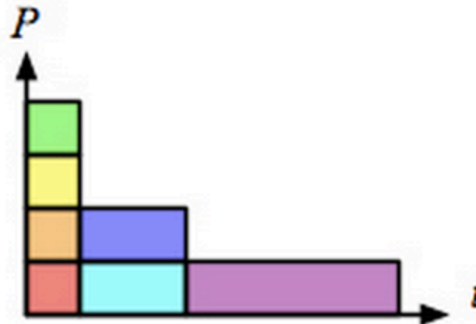
Comparison

	Built-in	IPython Cluster	Celery
High Level APIs	✔	✔	✔
Spreading Data	(✔)	✔	(✔)
Interactive Mode	✘	✔	✔
Scale cross machines	✘	✔	✔
Add-hoc workers add/drop	✘	(✔)	✔
Overhead	S	M	L

A question?

- Implement a parallel reduce using parallel map





Summary

Now you know something cool

- FP Overview
  - Definition, Benefit, Python Support
- Pure first-class function
  - First-Class, No side effect, Persistence
- High Order Functions
  - Map, Reduce, Group By, Compose, Partial, Currying
- Recursion Technology
  - TR, TRO, Trampoline, Memorize
  - Overloading, Pattern Matching
- Laziness: Stream, Iterators
- Parallel Related

Tools Used

- multipledispatch
- pyrsistent
- fn
- (cy)toolz
- pyrthon
- patterns
- ‘ipython [all]’
- pp
- IPython

Thank you!

人生苦短

python当歌

Contact me

- Email: wjo1212 at 163.com
- Wechat: LaiQiangDing