# Pipes and “then” notation: Functional Programming in Python

# Remark

- Techniques from F#.
- This is an update of my previous talk on “then” notation in Python

# About speaker

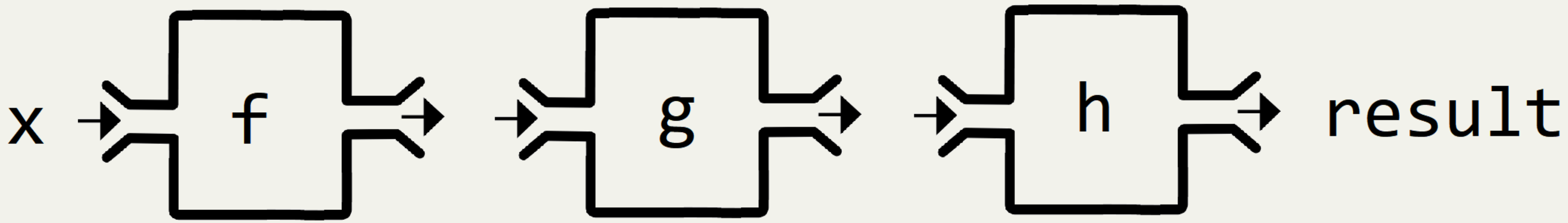
- Chang HaiBin
- Data Engineer (Hedge Fund)
- M.Sc. Uni. of Michigan (Math)

- Financial Engineer (2017-2018)
- Business Analyst (2015-2016)

Previously:

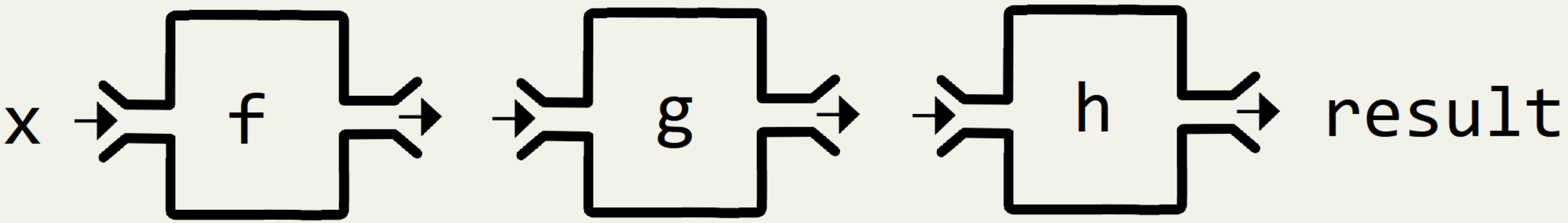
then (pipe-forward)

# then



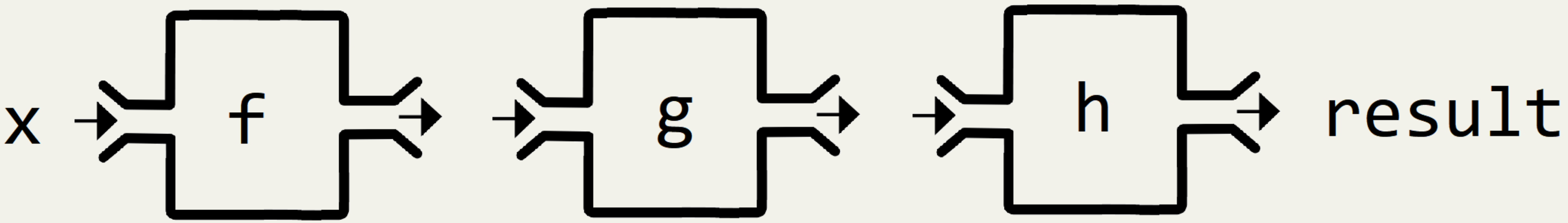
# then

- `h(g(f(x)))`



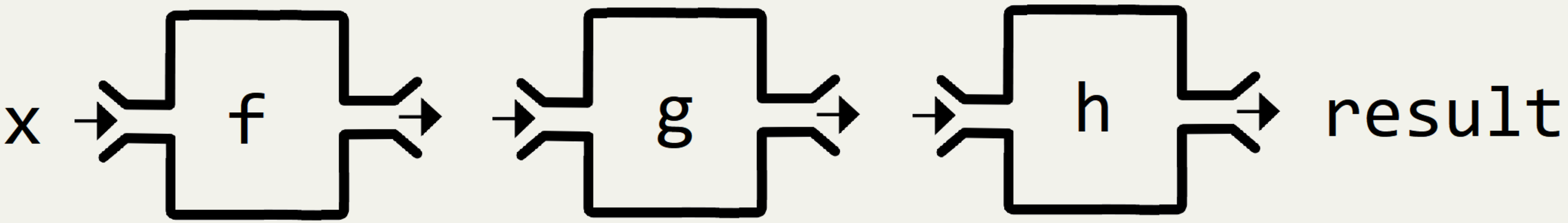
# then

- 0. Use x
- 1. Do f
- 2. Do g
- 3. Do h



# then

-
  ```
  x   \
  | then | f \
  | then | g \
  | then | h
  ```



# (Demo) Project Euler

- Math/Programming Challenge problems.

# Question 1

- From 1 to 999, find the sum of all numbers that are either multiples of 3, or multiples of 5.

# Solution Q1

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print
```

# Solution Q1

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print
```

Start from 1 to 999

Then keep the numbers you want (multiples of 3 or 5)

Then sum up the remaining numbers

Then print the result

```
range(1,1000)
```

```
[1, 2, 3, 4, 5, ......, 999]
```

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0)
```

```
[3, 5, 6, 9, 10, 12, 15, 18, 20, ......]
```

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum
```

```
233168
```

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print
```

Print 233168 to console

# Original Code

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print
```

# If we allow symbols...

```
range(1,1000) \
-> keep(lambda x : x % 3 == 0 or x % 5 == 0) \
-> sum \
-> print
```

Remark: The symbol used in F# is |>

- Cannot create new symbols in Python
- Overwrite the | operator to support the “`| then |`” notation

- Cannot create new symbols in Python
- Overwrite the | operator to support the “`| then |`” notation
- But will cause trouble if other package also uses the vertical | operator.

# How to define “then”

```
from functools import partial
class Infix(object):
    def __init__(self, func):
        self.func = func
    def __or__(self, other):
        return self.func(other)
    def __ror__(self, other):
        return Infix(partial(self.func, other))
    def __call__(self, v1, v2):
        return self.func(v1, v2)

then = Infix(lambda x,f: f(x))
```

# Pythonic way?

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print
```

# *args

# Function input

```
def add(a,b,c):
    return a + b + c
```

# Function input

```
def add(a,b,c):
    return a + b + c

x = add(1,2,3)
# x = 6
```

# Function input

```
def add(a,b,c):
    return a + b + c

x = add(1,2,3)
# x = 6

y = add(1,2)
# ERROR!

z = add(1,2,3,4,5)
# ERROR!
```

# Variable number of inputs

```
def add(*args):
    result = 0
    for x in args:
        result = result + x
    return result
```

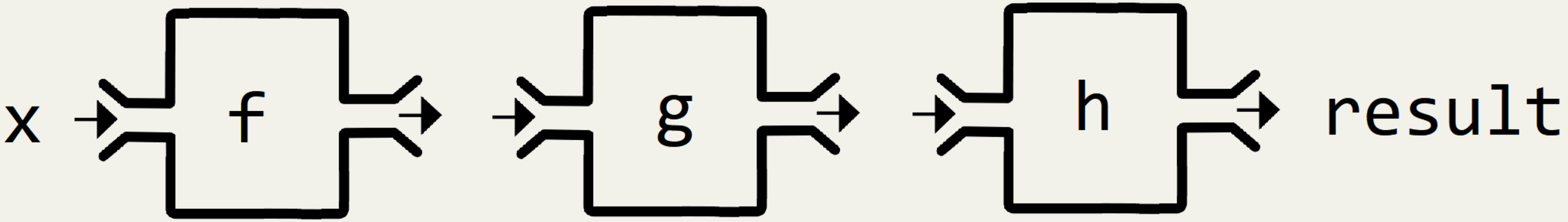
# Variable number of inputs

```
def add(*args):
    result = 0
    for x in args:
        result = result + x
    return result

x = add(1,2,3)
y = add(1,2)
z = add(1,2,3,4,5)
# x = 6, y = 3, z = 15
```

# pipe

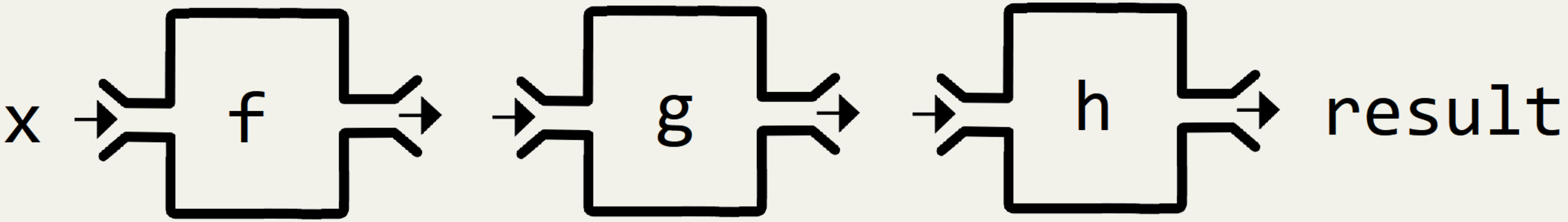
# then

- `h(g(f(x)))`



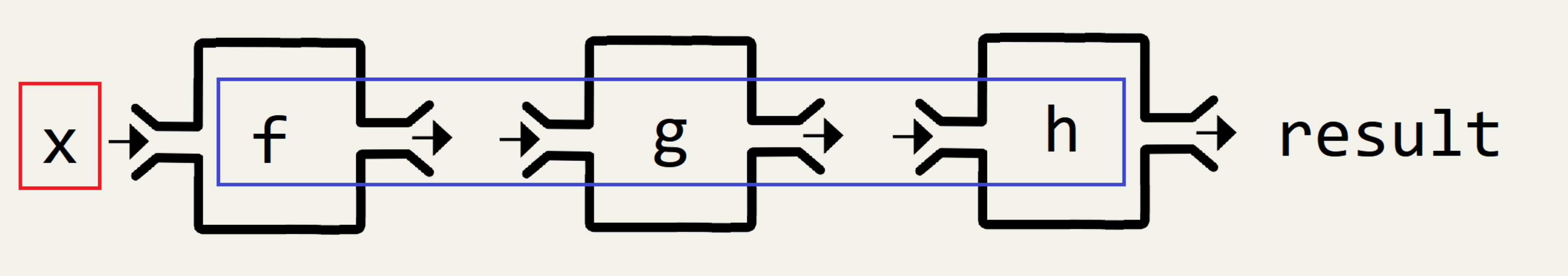
# then

- 0. Use x
- 1. Do f
- 2. Do g
- 3. Do h



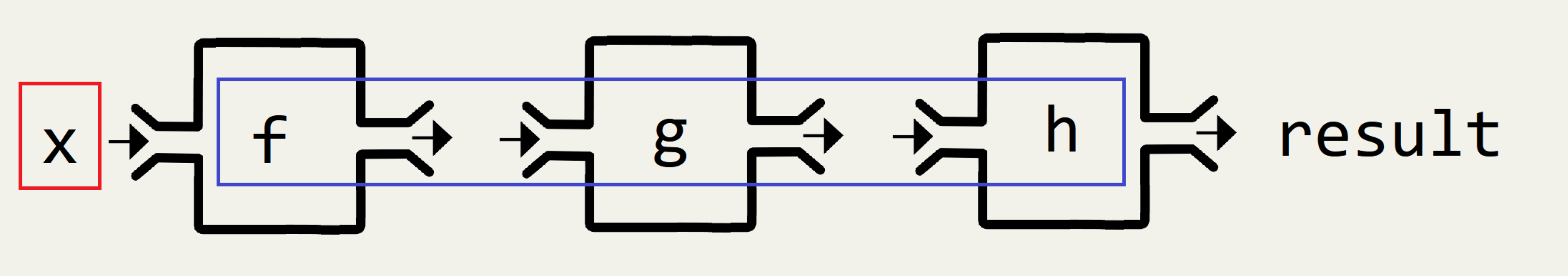
# then

- 0. Use x
- 1. Do f
- 2. Do g
- 3. Do h



# pipe

```
def pipe(x,*fs):
```

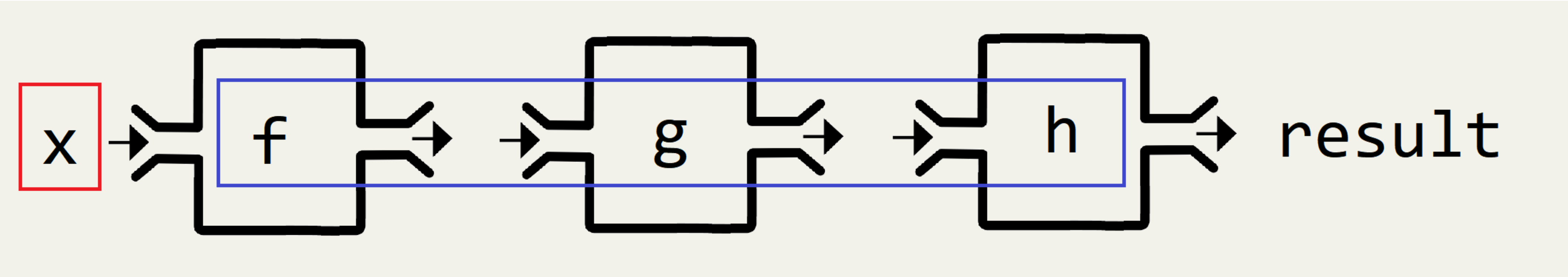


# pipe

```
def pipe(x,*fs):
```

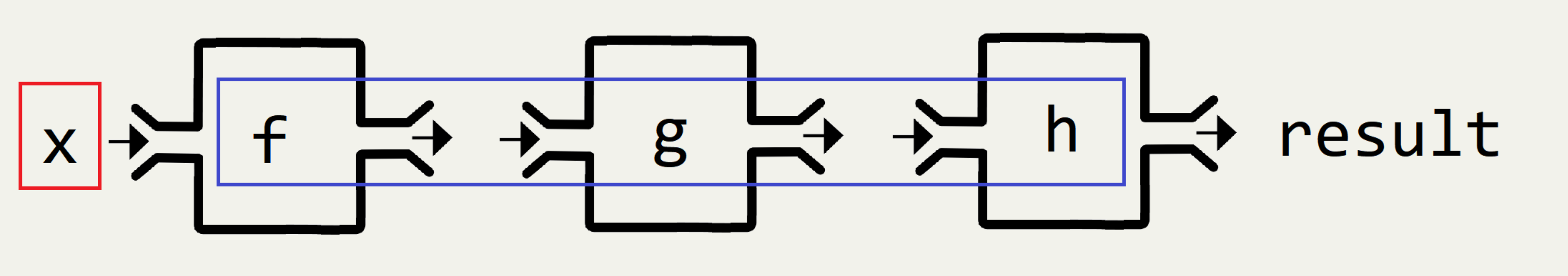
More than 1 apple -> apples
More than 1 computer -> computers

More than 1 function (functions are usually denoted “`f`” in math) -> `fs`



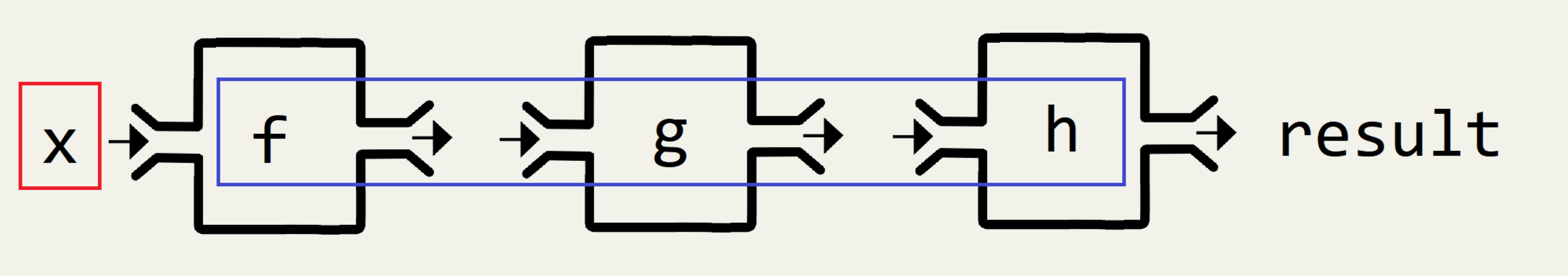
# pipe

```
def pipe(x,*fs):
    temp = x
    for f in fs:
        temp = f(temp)
    return temp
```



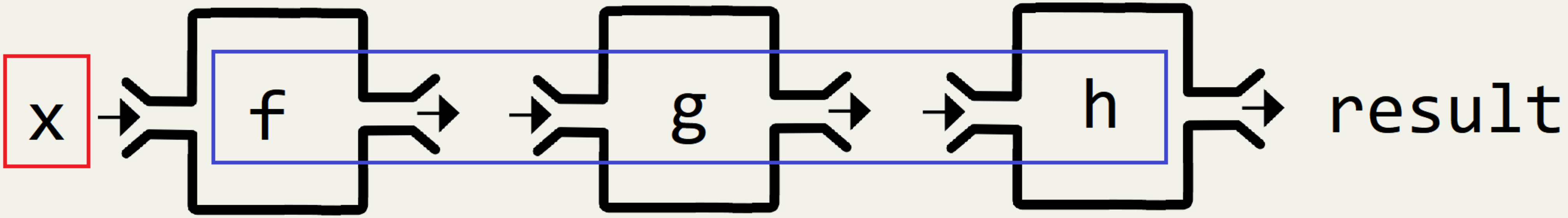
# How to use it?

```
def pipe(x,*fs):

result = pipe(x,f,g,h)
```



# How to use it?

```
def pipe(x,*fs):

result = pipe(x,
              f,
              g,
              h)
```



Compare with “then” notation

# Original “then” Code

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print
```

# pipe-notation

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print

pipe(range(1,1000),
    keep(lambda x : x % 3 == 0 or x % 5 == 0),
    sum,
    print
)
```

# pipe-notation

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print

pipe(range(1,1000),
    keep(lambda x : x % 3 == 0 or x % 5 == 0),
    sum,
    print
)
```

# pipe-notation

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print

pipe(range(1,1000),
    keep(lambda x : x % 3 == 0 or x % 5 == 0),
    sum,
    print
)
```

Demo

# Pros and Cons

# Reminder: Symbols

```
range(1,1000) \
-> keep(lambda x : x % 3 == 0 or x % 5 == 0) \
-> sum \
-> print

range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print
```

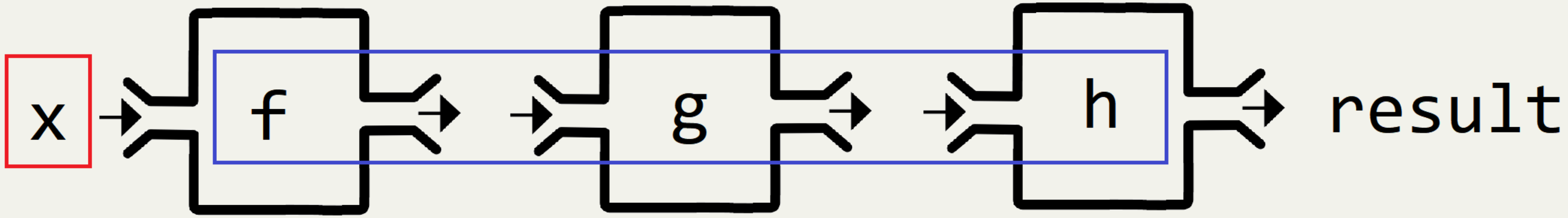
# comparison

“then” arrow notation:

```
x -> f -> g -> h
```

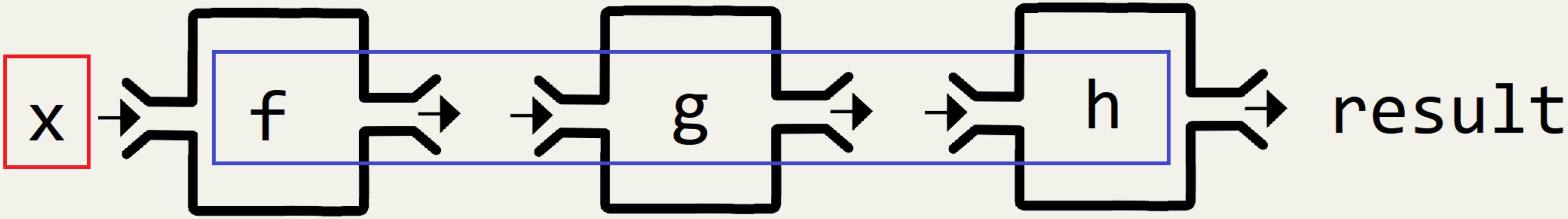
Pipe notation:

```
pipe(x,f,g,h)
```



# analogy

```
“then” arrow notation:
x -> f -> g -> h
1 +  2 +  3 +  4

Pipe notation:
pipe(x,f,g,h)
add (1,2,3,4)
```



# “then” feels natural

- “then”: 1 + 2 + 3 + 4
- Pipe: add(1,2,3,4)

# “then” feels natural

- “then”: 1 + 2 + 3 + 4
- Pipe: add(1,2,3,4)

- “then” is easier to chain:
- 1 + 2 + 3 + 4 + 5   = (1 + 2 + 3 + 4) + 5

# “then” feels natural

- “then”: 1 + 2 + 3 + 4
- Pipe: add(1,2,3,4)

- “then” is easier to chain:
- 1 + 2 + 3 + 4 + 5   = (1 + 2 + 3 + 4) + 5

- add(1,2,3,4,5) = add(add(1,2,3,4),5)
- Infix notation “+” feels natural, e.g. like primary school math

# “then” has clear direction

- “then”: `x -> f -> g -> h`
- Pipe: `add(x,f,g,h)`

# “then” has clear direction

- “then”: `x -> f -> g -> h`
- Pipe: `add(x,f,g,h)`

- `Visually x -> f -> g -> h looks clearer`

# “then” has clear direction

- “then”: `x -> f -> g -> h`
- Pipe: `add(x,f,g,h)`

- `Visually x -> f -> g -> h looks clearer`

- `add(x,f,g,h) does not convey the order visually`

# Pipe is more Pythonic

- “then”: `x -> f -> g -> h`
- Pipe: `add(x,f,g,h)`

# Pipe is more Pythonic

- “then”: `x -> f -> g -> h`
- Pipe: `add(x,f,g,h)`

- `Python does not allow new symbol.`
- `Implemented in this way:`
- `x |then| f |then| g |then| h`

# Pipe is more Pythonic

- “then”: `x -> f -> g -> h`
- Pipe: `add(x,f,g,h)`

- Python does not allow new symbol.
- Implemented in this way:
- `x |then| f |then| g |then| h`

- Leads to conflict if there are other uses of vertical | as well.

# Pipe easy implementation

```
def pipe(x,*fs):
    temp = x
    for f in fs:
        temp = f(temp)
    return temp
```

Compare this to implementation of then (see next slide)

# How to define “then”

```
from functools import partial
class Infix(object):
    def __init__(self, func):
        self.func = func
    def __or__(self, other):
        return self.func(other)
    def __ror__(self, other):
        return Infix(partial(self.func, other))
    def __call__(self, v1, v2):
        return self.func(v1, v2)

then = Infix(lambda x,f: f(x))
```

# List of instructions

# using the * notation

# * notation

- The star * notation allows more flexibility.

- (Double-edged sword)

# Example

```
def product(xs):
    ......
```

Create a function that takes a list of numbers, and calculate the product.

# Example

```
def product(xs):
    temp = 1
    for x in xs:
        temp = temp * x
    return temp
```

# Example

```
def product(xs):
    return pipe(
        1,
        *[mult_with(x) for x in xs]
    )



def mult_with(x):
    return lambda y: x * y
```

# Example

```
def product(xs):
    return pipe(
        1,
        *[mult_with(x) for x in xs]
    )
```

Arbitrary number
of steps/instructions

```
def mult_with(x):
    return lambda y: x * y
```

# Example: Project Euler Q5

# Project Euler Q5

```
What is the smallest integer that can be
divided (no remainder) by all numbers from 1
to 20?
```

# Remark

Assume that you know how to calculate the LCM (lowest common multiple) of two numbers,

Which you can calculate with HCF/GCD (Highest Common Factor/ Greatest Common Divisor) of two numbers

Which you can calculate using Euclid’s algorithm.

# Project Euler Q5

```
What is the smallest integer that can be
divided (no remainder) by all numbers from 1
to 20?
```

# Project Euler Q5

What is the smallest integer that can be divided (no remainder) by all numbers from 1 to 20?

A

Step A: 1,2

# Project Euler Q5

```
What is the smallest integer that can be
divided (no remainder) by all numbers from 1
to 20?
```

```
A -> B
```

```
Step A: 1,2

Step B: 1,2,3
```

# Project Euler Q5

What is the smallest integer that can be divided (no remainder) by all numbers from 1 to 20?

A -> B -> C

Step A: 1,2

Step B: 1,2,3

Step C: 1,2,3,4

etc.

# Solution 1

```
temp = 1
for x in range(1,21):
    temp = lcm(temp,x)
return temp
```

# Solution 2

```
pipe(
    1,
    *[lcm_with(x) for x in range(1,21)]
)
```

# How does it work?

# Example

```
def f1(x): return x + 1
def f2(x): return x + 2
def f3(x): return x + 3

pipe(
    100,
    f1,
    f2,
    f3
)
```

# Example

```
def f1(x): return x + 1
def f2(x): return x + 2
def f3(x): return x + 3
fs = [f1, f2, f3]

pipe(
    100,
    *fs
)
```

# Conclusion

# pipe-notation

```
range(1,1000) \
| then | keep(lambda x : x % 3 == 0 or x % 5 == 0) \
| then | sum \
| then | print

pipe(range(1,1000),
    keep(lambda x : x % 3 == 0 or x % 5 == 0),
    sum,
    print
)
```

# * notation

```
pipe(
    1,
    *[lcm_with(x) for x in range(1,21)]
)
```

# Implementation

```
def pipe(x,*fs):
    temp = x
    for f in fs:
        temp = f(temp)
    return temp
```

Q&A