Functional Programming - 2

Recap

In [ ]:
```
#CodeA
@f
def f1():
    print("Hello")

#CodeB
def f1():
    print("Hello")
f1 = f(f1)
```

@f
f1 = f(f1)

@pretty
def foo():
    ___
    ___
    ___

foo = pretty(foo)

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Why functional programming?

In [3]:
```
x = 5 # initial data

# hidden mutations => changing the state, but can't track them
x = 3*x
x += 2

print(x)
# print(previous_x?)
# can you rollback (go back to a previous state) of the variable x?
```

```
17
```

all mutations in fn call.

In [6]:
```
x = 5

# Mutation 1
x1 = 3*x

# Mutation 2
x2 = x1 + 2

# Can you get back to the previous states now?

print(x, x1, x2)

# PROBLEM : TOO MANY VARIABLES
```

```
5 15 17
```

```
5 15 17
```

Best of both worlds => tracking but with less no of variables

In [8]:
```
x = 5

# what are the pros of making changes to data via calling functions
# 1. reusability of the changes will increase
# 2. tracking the data
def mutation_1(x): # x is a parameter, different from the global x
    x = 3*x
    x += 1
    return x

x1 = mutation_1(x)

print(x, x1)
```

```
5 16
```

In [ ]:

In [ ]:

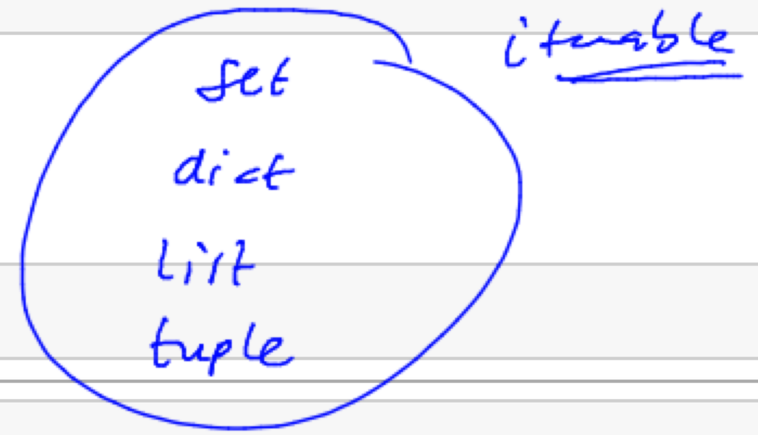
In [ ]:

```
[1, 4, 9, 16, 25]
```

In [18]:
```
# __iter__
# __next__

# they are actually behaviors in python => dunder method
```

In [ ]:



Functional way

In [19]:
```
map?
```

In [ ]:

iter((1,2,3))

In [ ]:

```
Init signature: map(self, /, *args, **kwargs)
Docstring:
map(func, *iterables) --> map object

Make an iterator that computes the function using arguments from
each of the iterables.  Stops when the shortest iterable is exhausted.
Type:           type
Subclasses:
```

Functional way

In [19]:
```
map?
```

In [20]:
```
a = [1, 2, 3, 4, 5]
```

In [21]:
```
res = map(lambda x: x**2, a)
```

In [22]:
```
print(res)
```

```
<map object at 0x7fb768d769a0>
```

In [23]:
```
res_list = list(res)
```

In [24]:
```
print(res_list)
```

```
[1, 4, 9, 16, 25]
```

In [27]:
```
# doing in a single line => very elegant
res_direct = list(map(lambda x: x**2, a))
print(res_direct)
```

```
[1, 4, 9, 16, 25]
```

Anonymous fn

In [ ]:

In [ ]:

```
heights = [144, 167, 189, 170, 190, 150, 165, 178, 200, 130]
```

In [34]:
```
def complex_logic(height):
    if height < 150:
        return "M"
    else:
        return "L"
```

In [35]:
```
sizes = list(map(complex_logic, heights))
```

In [36]:
```
print(sizes)
```

```
['M', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'M']
```

In [37]:
```
# can we do it in one line?
```

In [41]:
```
def complex_logic_one_liner(height):
    # shorthand if else, TERNARY IF-ELSE
    return "M" if height < 150 else "L"
```

In [39]:
```
complex_logic_one_liner(150)
```

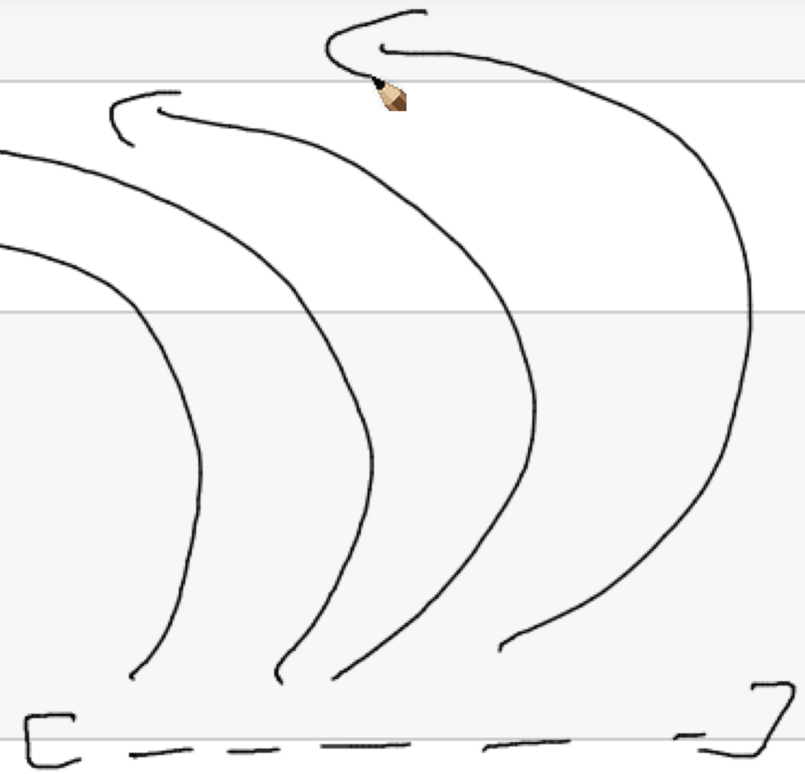
Out[39]:
```
'L'
```

In [42]:
```
complex_logic_one_liner(149)
```

Out[42]:
```
'M'
```

```
# if > 180 -> L
```

SOLVE



In [51]:
```
def complex_logic(height):
    if height < 150:
        return "S"
    elif height >= 150 and height < 180:
        return "M"
    else:
        return "L"
```

In [52]:
```
sizes = list(map(complex_logic, heights))
```

In [53]:
```
print(sizes)
```

```
['S', 'M', 'L', 'M', 'L', 'M', 'M', 'M', 'L', 'S']
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [51]:
```
def complex_logic(height):
    if height < 150:
        return "S"
    elif height >= 150 and height < 180:
        return "M"
    else:
        return "L"
```

In [52]:
```
sizes = list(map(complex_logic, heights))
```

In [53]:
```
print(sizes)
```

```
['S', 'M', 'L', 'M', 'L', 'M', 'M', 'M', 'L', 'S']
```

In [57]:
```
sizes2 = list(map(lambda x: "S" if x < 150 else "M"
                  if x >= 150 and x < 180 else "L", heights))
```

In [58]:
```
print(sizes2)
```

```
['S', 'M', 'L', 'M', 'L', 'M', 'M', 'M', 'L', 'S']
```

In [ ]:

In [ ]:

In [51]:
```
def complex_logic(height):
    if height < 150:
        return "S"
    elif height >= 150 and height < 180:
        return "M"
    else:
        return "L"
```

In [52]:
```
sizes = list(map(complex_logic, heights))
```

In [53]:
```
print(sizes)
```

```
['S', 'M', 'L', 'M', 'L', 'M', 'M', 'M', 'L', 'S']
```

In [57]:
```
sizes2 = list(map(lambda x: "S" if x < 150 else "M"
                  if x >= 150 and x < 180 else "L", heights))
```

In [58]:
```
print(sizes2)
```

```
['S', 'M', 'L', 'M', 'L', 'M', 'M', 'M', 'L', 'S']
```

In [ ]:

In [ ]:

In [57]:
```
sizes2 = list(map(lambda x: "S" if x < 150 else "M"
                  if x >= 150 and x < 180 else "L", heights))
```

In [58]:
```
print(sizes2)
```

```
['S', 'M', 'L', 'M', 'L', 'M', 'M', 'M', 'L', 'S']
```

Problem coming up

In [61]:
```
random1 = [1, 2, 3]
random2 = [4, 5, 6]

# Add the 2 lists using map function
# res = [5, 7, 9]
```

In [62]:
```
map?
```

```
Init signature: map(self, /, *args, **kwargs)
Docstring:
map(func, *iterables) --> map object

Make an iterator that computes the function using arguments from
each of the iterables.  Stops when the shortest iterable is exhausted.
Type:           type
Subclasses:
```

In [58]:
```
print(sizes2)
```

```
['S', 'M', 'L', 'M', 'L', 'M', 'M', 'M', 'L', 'S']
```

Problem coming up

In [61]:
```
random1 = [1, 2, 3]
random2 = [4, 5, 6]

# Add the 2 lists using map function
# res = [5, 7, 9]
```

In [62]:
```
map?
```

In [63]:
```
res = list(map(lambda x, y: x + y, random1, random2))
print(res)
```

```
[5, 7, 9]
```

In [ ]:

```
[5, 7, 9]
```

Quiz

In [ ]:
```
list1 = [1, 0, 1]
list2 = [0, 0, 1]

res = list(map(lambda x, y:  x == y, list1, list2))
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

```
res = list(map(lambda x, y:  x == y, list1, list2))
```

In [69]:
```
print(res)
```

```
[False, True, True, False]
```

Filter function

In [70]:
```
a = [1, 2, 3, 4, 5]
```

In [71]:
```
filter?
```

In [ ]:

```
Init signature: filter(self, /, *args, **kwargs)
Docstring:
filter(function or None, iterable) --> filter object

Return an iterator yielding those items of iterable for which function(item)
is true. If function is None, return the items that are true.
Type:           type
Subclasses:
```

map.

In [72]:
```
res = filter(lambda x: x % 2 == 0, a)
```

In [73]:
```
print(res)
```

```
<filter object at 0x7fb748ff1760>
```

In [74]:
```
res_list = list(res)
```

In [75]:
```
print(res_list)
```

```
[2, 4]
```

In [77]:
```
# single line with typecasting
f_direct = list(filter(lambda x: x % 2 == 0, a))
```

In [78]:
```
print(f_direct)
```

```
[2, 4]
```

In [ ]:

In [ ]:

Quizzes

In [79]:
```
l = [1, 10, -3, 4, 15]

def f1(x):
    return x<6

# m1 = ????

m1 = filter(f1, l)

print(list(m1))

# Replace ??? to print all numbers less than 6 in the list.
```

[1, -3, 4]

```
---------------------------------------------------------------------
--
NameError                                 Traceback (most recent call las
t)
Input In [79], in <cell line: 8>()
      4     return x<6
      6 # m1 = ????
----> 8 print(list(m1))

NameError: name 'm1' is not defined
```

In [ ]:

```
# first argument should be the function
# second argument should be the iterable
```

```
----------------------------------------------------------------------
--
TypeError                                 Traceback (most recent call las
t)
Input In [82], in <cell line: 1>()
----> 1 m1 = filter(l, f1)

TypeError: 'function' object is not iterable
```

In [86]:
```
s = {1, 2, 2, 5}
print(s)

def f1(x):
    return x<6

m1 = filter(f1, s)
print(list(m1))
```

```
{1, 2, 5}
[1, 2, 5]
```

{1,2,5}

||

list(1 2 5)

In [ ]:

In [ ]:

In [ ]:

```
# first argument should be the function
# second argument should be the iterable
```

```
----------------------------------------------------------------------
--
TypeError                                 Traceback (most recent call las
t)
Input In [82], in <cell line: 1>()
----> 1 m1 = filter(l, f1)

TypeError: 'function' object is not iterable
```

In [86]:
```
s = {1, 2, 2, 5}
print(s)

def f1(x):
    return x<6

m1 = filter(f1, s)
print(list(m1))
```

```
{1, 2, 5}
[1, 2, 5]
```

In [ ]:

In [ ]:

In [ ]:

Reduce

In [87]:
```
reduce?
```

```
Object `reduce` not found.
```

In [88]:
```
from functools import reduce

# this line will bring the reduce function into our current code
```

In [89]:
```
reduce?
```

In [ ]:

```
Docstring:
reduce(function, sequence[, initial]) -> value

Apply a function of two arguments cumulatively to the items of a sequence,
from left to right, so as to reduce the sequence to a single value.
For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates
((((1+2)+3)+4)+5).  If initial is present, it is placed before the items
of the sequence in the calculation, and serves as a default when the
sequence is empty.
Type:      builtin_function_or_method
```

```
# 2nd arg: the iterable
res = reduce(lambda x, y: x + y, a)
```

In [92]:
```
print(res)
```

```
15
```

$n\frac{(n+1)}{2}$

Quizzes

$1+2+\cdots+10$

In [93]:
```
a = list(range(1, 11))
```

$=\frac{10(11)}{2}$

In [94]:
```
print(a)
```

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

$=\frac{110}{2}=55$

In [95]:
```
res = reduce(lambda x, y: x + y, a)
print(res)
```

```
55
```

In [ ]:
```

```

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

In [93]:
```
a = list(range(1, 11))
```

In [94]:
```
print(a)
```

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

In [95]:
```
res = reduce(lambda x, y: x + y, a)
print(res)
```

```
55
```

In [96]:
```
b = a[::-1]
print(b)
```

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

In [98]:
```
res = reduce(lambda x, y: x + y, b)
print(res)
```

```
55
```

In [ ]:

order of addition is diff, but result is same.

all elements are in total same

$x+y = y+x$

Commutative.

20 / 20

```
55
```

Tricky Quizzes

In [99]:
```
from functools import reduce

list = {1, 2, 33, 2}
print(list)

print(reduce(lambda x, y: x+y, list))
```

1+2+33 = 36

```
{1, 2, 33}
36
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [99]:
```
from functools import reduce

list = {1, 2, 33, 2}
print(list)

print(reduce(lambda x, y: x+y, list))
```

```
{1, 2, 33}
36
```

{1, 2, 33}

In [ ]:
```
from functools import reduce

list = {1, 2, 33, 2}
print(list)
```

22 / 22

```
Docstring:
reduce(function, sequence[, initial]) -> value

Apply a function of two arguments cumulatively to the items of a sequence,
from left to right, so as to reduce the sequence to a single value.
For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates
((((1+2)+3)+4)+5).  If initial is present, it is placed before the items
of the sequence in the calculation, and serves as a default when the
sequence is empty.
Type:      builtin_function_or_method
```

5

```
print(reduce(lambda x, y: x+y, list))
```

```
{1, 2, 33}
36
```

In [ ]:
```
from functools import reduce

list = {1, 2, 33, 2}
print(list)

print(reduce(lambda x, y: x+y, list))
```

{5, 1, 2, 33}

In [100]:
```
reduce?
```

In [101]:
```
# initial value => might be useful when carry over from some
# previous operations
```

In [102]:
```
print(reduce(lambda x, y: x+y, list, 5))
```

41

fn it. initial value

In [ ]:

In [ ]:

In [ ]:

```
print(reduce(lambda x, y: x+y, list))
```

[100]:
```
reduce?
```

[101]:
```
# initial value => might be useful when carry over from some
# previous operations
```

[102]:
```
print(reduce(lambda x, y: x+y, list, 5))
```

```
41
```

[103]:
```
from functools import reduce
list = [1, 2, 33, 2]
print(reduce(lambda x, y: x+y, list, 10))
```

```
48
```

$10 + [1, 2, 33, 2]$

$10 + 38$

$= 48$

$y + 0 + x = x$

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

[103]:
```
from functools import reduce
list = [1, 2, 33, 2]
print(reduce(lambda x, y: x+y, list, 10))
```

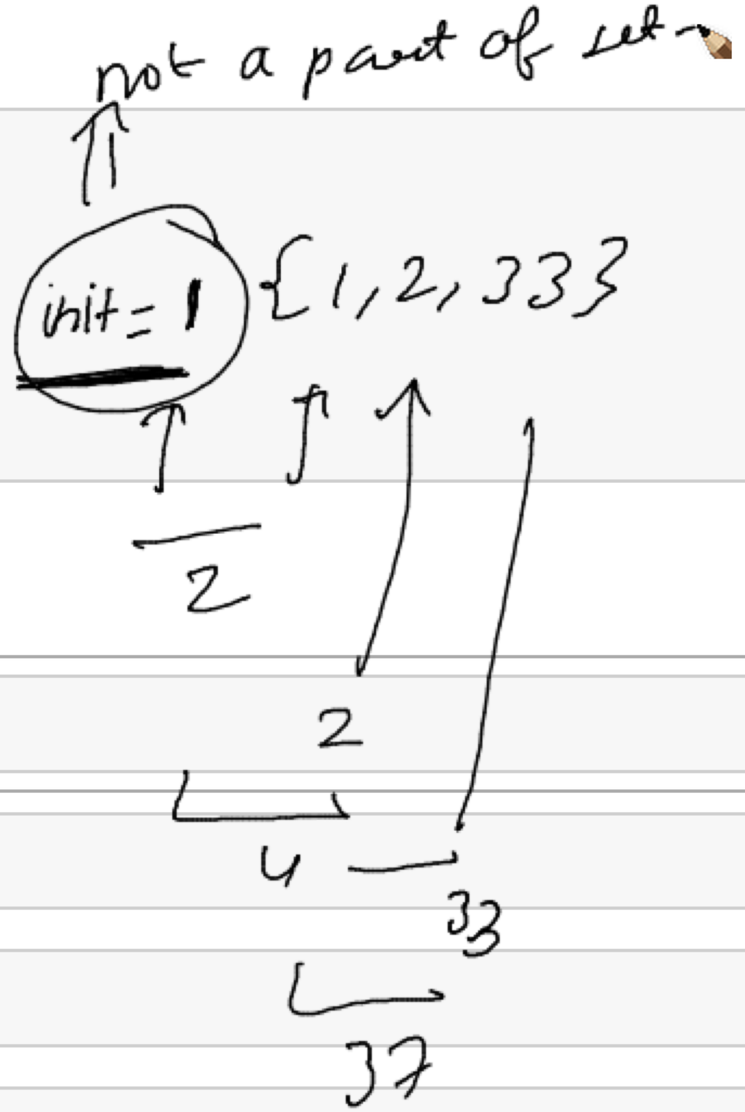
```
48
```

[104]:
```
from functools import reduce

list = {1, 2, 33, 2}
print(list)

print(reduce(lambda x, y: x+y, list, 1))
```

```
{1, 2, 33}
37
```



In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

```
{1, 2, 33}
37
```

zip function

will be covered again in next class problem solving beginning

In [106]:
```
a = ["sachin", "shreyash", "needa", "harshita", "anwesh"]
```

In [107]:
```
for i in enumerate(a):
    print(i)
```

```
(0, 'sachin')
(1, 'shreyash')
(2, 'needa')
(3, 'harshita')
(4, 'anwesh')
```

In [ ]:

In [ ]:

In [ ]:

```
(0, 'sachin')
(1, 'shreyash')
(2, 'needa')
(3, 'harshita')
(4, 'anwesh')
```

In [1]:
```
states = ['Haryana', 'Uttarakhand', 'Punjab', 'Uttar Pradesh']
capitals = ['Chandigarh', 'Dehradun', 'Chandigarh', 'Lucknow']

res = zip(states, capitals)
```

In [2]:
```
print(res)
```

```
<zip object at 0x7fde080925c0>
```

In [3]:
```
res_list = list(res)
```

In [4]:
```
print(res_list)
```

```
[('Haryana', 'Chandigarh'), ('Uttarakhand', 'Dehradun'), ('Punjab', 'Chan
digarh'), ('Uttar Pradesh', 'Lucknow')]
```

In [ ]:

Iterate on list of tuples

In [5]:
```
res = list(zip(states, capitals))

for state_capital_pair in res:
    print(state_capital_pair)
```

```
('Haryana', 'Chandigarh')
('Uttarakhand', 'Dehradun')
('Punjab', 'Chandigarh')
('Uttar Pradesh', 'Lucknow')
```

In [6]:
```
res = dict(zip(states, capitals))

print(res)
```

```
{'Haryana': 'Chandigarh', 'Uttarakhand': 'Dehradun', 'Punjab': 'Chandigarh', 'Uttar Pradesh': 'Lucknow'}
```

In [ ]:

Iterate on list of tuples

In [5]:
```
res = list(zip(states, capitals))

for state_capital_pair in res:
    print(state_capital_pair)
```

```
('Haryana', 'Chandigarh')
('Uttarakhand', 'Dehradun')
('Punjab', 'Chandigarh')
('Uttar Pradesh', 'Lucknow')
```

In [6]:
```
res = dict(zip(states, capitals))

print(res)
```

```
{'Haryana': 'Chandigarh', 'Uttarakhand': 'Dehradun', 'Punjab': 'Chandigar
h', 'Uttar Pradesh': 'Lucknow'}
```

In [ ]:

```
('Uttarakhand', 'Dehradun')
('Punjab', 'Chandigarh')
('Uttar Pradesh', 'Lucknow')
```

In [8]:
```
res = dict(zip(states, capitals))

print(res)
```

```
{'Haryana': 'Chandigarh', 'Uttarakhand': 'Dehradun', 'Punjab': 'Chandigar
h', 'Uttar Pradesh': 'Lucknow'}
```

In [10]:
```
for i in res: # only get the keys while iterating on a dict
    print(i)
```

```
Haryana
Uttarakhand
Punjab
Uttar Pradesh
```

In [7]:
```
res = list(dict(zip(states, capitals)))

print(res)
```

```
['Haryana', 'Uttarakhand', 'Punjab', 'Uttar Pradesh']
```

In [ ]:

In [7]:
```
res = list(dict(zip(states, capitals)))

print(res)
```

```
['Haryana', 'Uttarakhand', 'Punjab', 'Uttar Pradesh']
```

In [20]:
```
from functools import reduce

a = [9, 2, 3, 5, 6]

res1 = reduce(lambda x, y: x + y, a)
print(res1)

b = [1, 3, 5, 6, 7]

total_res = reduce(lambda x, y: x + y, b, res1)
print(total_res)
```

```
25
47
```

In [ ]:

In [25]:
```
print(type(len))
```

```
<class 'builtin_function_or_method'>
```

In [26]:
```
a = ["hello", "world", "this", "is", "sorting"]
```

In [27]:
```
res = sorted(a, key = len)
```

In [28]:
```
print(res)
```

```
['is', 'this', 'hello', 'world', 'sorting']
```

In [31]:
```
summ = 0
def f(val):
    global summ
    summ=summ+val

x = [1,2,3,4,5]
final_list = list(map(f,x))

print(x)
```

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

In [ ]:
```

```

In [25]:
```
print(type(len))
```

```
<class 'builtin_function_or_method'>
```

In [26]:
```
a = ["hello", "world", "this", "is", "sorting"]
```

In [27]:
```
res = sorted(a, key = len)
```

In [28]:
```
print(res)
```

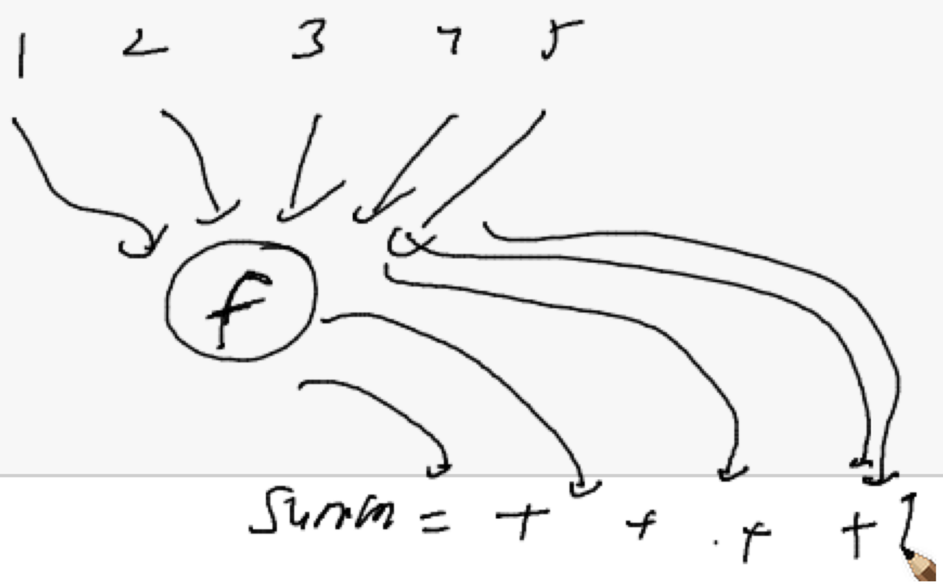
```
['is', 'this', 'hello', 'world', 'sorting']
```

In [31]:
```
summ = 0
def f(val):
    global summ
    summ=summ+val

x = [1,2,3,4,5]
final_list = list(map(f,x))

print(x)
```

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



In [ ]:
```

```

```
----> 1 x = dict(map(lambda x: x**2, {1, 2, 3, 3, 7, 9, 9}))
      2 print(x)

TypeError: cannot convert dictionary update sequence element #0 to a sequ
ence
```

In [39]:
```
x = dict(map(lambda x: x, [(1, 2), (2, 4), (3, 6)]))
print(x)
```

```
{1: 2, 2: 4, 3: 6}
```

In [40]:
```
x = dict(filter(None, [(1, 2), (2, 4), (3, 6)]))
print(x)
```

```
{1: 2, 2: 4, 3: 6}
```

In [44]:
```
x = reduce(lambda x, y: (x[0] + y[0], x[1] + y[1]), [(1, 2), (2, 4), (3, 6
print(x)
```

```
(6, 12)
```

In [ ]: