5주차
RECURSION, DICTIONARIES

서울대학교 전기정보공학부
명예교수 성원용

MUTABLE, IMMUTABLE

- Mutable – 값이 변할 수 있는 것. 이유 call by reference
- Immutable – 값이 변하면 변수 id 가 달라지는 것. 이유 call by value
- 변경가능한 객체에는 리스트(list)와 딕셔너리(dict) 이 있습니다.
- 변경불가능한 객체에는 일반적인 자료형 int, string 등 과 튜플(tuple) 등

```
a = 1                    #1
>>> id(a)
140729439544144
>>> a += 1               #2
>>> id(a)
140729439544176
```

String

```
a = 'a'                  #a
>>> id(a)
1985199052704
>>> a += 'b'             #ab
>>> id(a)
1985209620944
```

Tuple

```
>>> a = (1,2)            #(1,2)
>>> id(a)
1985209935624
>>> a += (3,)            #(1,2,3)
>>> id(a)
1985209858448
```

```
>>> a = [1,2]
>>> b = a              #b = [1,2]
>>> a == b             #a = [1,2] , b = [1,2]
True
>>> b.append(3)        #b = [1,2,3]
>>> a == b             #a = [1,2,3] , b = [1,2,3]
True
```

Dict

```
>>> a = {1:'a'}
>>> b = a              #b = {1:'a'}
>>> a == b             #a = {1:'a'} , b = {1:'a'}
True
>>> b[2] = 'b'         #b = {1:'a',2:'b'}
>>> a == b             #a = {1:'a',2:'b'} , b = {1:'a',2:'b'}
True
```

- b=a+[4,]
- c.append(5)

```
a=[1, 2, 3]
b=a
c=a
print(id(a), id(b), id(c))
b=a+[4,]
print(a, b, c, id(b))
c.append(5)
print(a, b, c)

2155290353928 2155290353928 2155290353928
[1, 2, 3] [1, 2, 3, 4] [1, 2, 3] 2155290320712
[1, 2, 3, 5] [1, 2, 3, 4] [1, 2, 3, 5]
```

- List는 function 에 reference 만 pass
- Function 에서 list의 element를 바꾸어도 calling function 의 그 list 값도 변한다.

```
def list_print2(m):
    m[0]=4
    print("in the function", id(m))

m_list = [1, 2, 3]
print("in the main", id(m_list))
list_print2(m_list)
print(m_list)
--------
in the main 1816429127688
in the function 1816429127688
[4, 2, 3]
```

```
def list_print1(l):
    l=l*2
    print("in the function", id(l))
    print(l)

l_list = [1, 2, 3]
print("in the main", id(l_list))
list_print1(l_list)
print(l_list)
-----
in the main 1816450451400
in the function 1816450457864
[1, 2, 3, 1, 2, 3]
[1, 2, 3]
```

```python
def string_print(ls):
#    ls = ls*2
    print("in the function", ls, id(ls))


ls = "test"
print("in the main", id(ls))
string_print(ls)
print(ls)
```

```
in the main 1816387087600
in the function test 1816387087600
test
```

```python
def string_print(ls):
    ls = ls*2
    print("in the function", ls, id(ls))


ls = "test"
print("in the main", id(ls))
string_print(ls)
print(ls)
```

```
in the main 1816387087600
in the function testtest 1816451378544
test
```

- Dictionary도 list와 마찬가지로 function 에 reference 만 pass
- Function 에서 dictionary 의 element를 바꾸어도 calling function 의 해당 dictionary 값도 변한다.

```
def test1(d_dic):
    d_dic["Kim"] = 100
    print("in the function", d_dic, id(d_dic))

score = {"Sung": 90, "Choi": 85}
print("in the main1", score, id(score))
test1(score)
print("in the main2", score, id(score))
------------------
in the main1 {'Sung': 90, 'Choi': 85} 1816451393000
in the function {'Sung': 90, 'Choi': 85, 'Kim': 100}
1816451393000
in the main2 {'Sung': 90, 'Choi': 85, 'Kim': 100}
```

```
import copy

a = [1, [1, 2, 3]]
b = copy.copy(a)    # shallow copy 발생
print(b)    # [1, [1, 2, 3]] 출력
b[0] = 100
print(b)    # [100, [1, 2, 3]] 출력,
print(a)    # [1, [1, 2, 3]] 출력, shallow copy 가 발생해 복사된 리스트는 별도의 객체이므로 item을 수정하면 복사본만 수정된다. (immutable 객체의 경우)

c = copy.copy(a)
c[1].append(4)    # 리스트의 두번째 item(내부리스트)에 4를 추가
print(c)    # [1, [1, 2, 3, 4]] 출력
print(a)    # [1, [1, 2, 3, 4]] 출력, a가 c와 똑같이 수정된 이유는 리스트의 item 내부의 객체는 동일한 객체이므로 mutable한 리스트를 수정할때는 둘다 값이 변경됨
```

주석에서 잘 설명하고 있지만, 리스트 내에 리스트가 있는 경우에 얕은 복사(b = copy.copy(a))가 이뤄지더라도 리스트 내의 내부 리스트까지 별도의 객체로 복사가 되는것은 아닙니다.

LAST TIME

- tuples - immutable
- lists - mutable
- aliasing, cloning
- mutability side effects

TODAY

- recursion – divide/decrease and conquer
- dictionaries – another mutable object type

RECURSION

- Recursion is the process of repeating items in a self-similar way.

WHAT IS RECURSION?

- Algorithmically: a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
  - reduce a problem to simpler versions of the same problem
- Semantically: a programming technique where a **function calls itself**
  - in programming, goal is to NOT have infinite recursion
    - must have **1 or more base cases** that are easy to solve
    - must solve the same problem on **some other input** with the goal of simplifying the larger problem input

ITERATIVE ALGORITHMS SO FAR

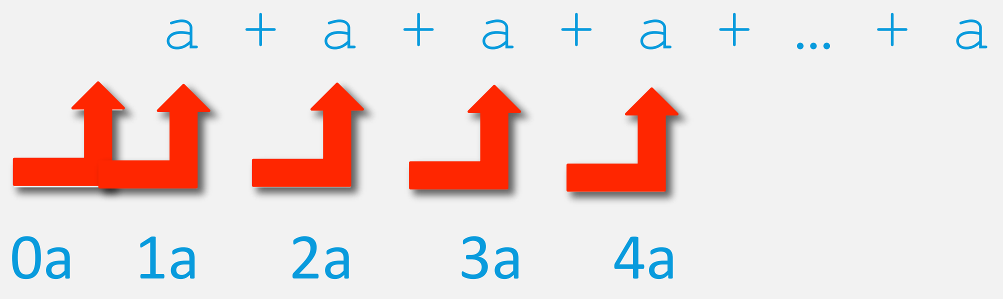
- looping constructs (`while` and `for` loops) lead to **iterative** algorithms
- can capture computation in a set of **state variables** that update on each iteration through loop

MULTIPLICATION – ITERATIVE SOLUTION

- "multiply `a * b`" is equivalent to "add `a` to itself `b` times"
- capture **state** by
  - an **iteration** number (`i`) starts at b

    `i` ← `i-1` and stop when 0
  - a current **value of computation** (`result`)

    `result` ← `result + a`




```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

iteration

current value of computation, a running sum

current value of iteration variable

MULTIPLICATION – BY RECURSION

- **recursive step**
  - think how to reduce problem to a **simpler/ smaller version** of same problem
- **base case**
  - keep reducing problem until reach a simple case that can be **solved directly**
  - when b = 1, a*b = a

$$a*b = \underbrace{a + a + a + a + \ldots + a}_{\text{b times}}$$

$$= a + \underbrace{a + a + a + \ldots + a}_{\text{b-1 times}}$$

$$= a + \boxed{a * (b-1)}$$

recursive reduction

```
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

base case

recursive step

FACTORIAL

서 울 대 학 교 SEOUL NATIONAL UNIVERSITY

```
n! = n*(n-1)*(n-2)*(n-3)* … * 1
```

- for what `n` do we know the factorial?

n = 1 →

```
if n == 1:
    return 1
```

base case

- how to reduce problem? Rewrite in terms of something simpler to reach base case

n*(n-1)! →

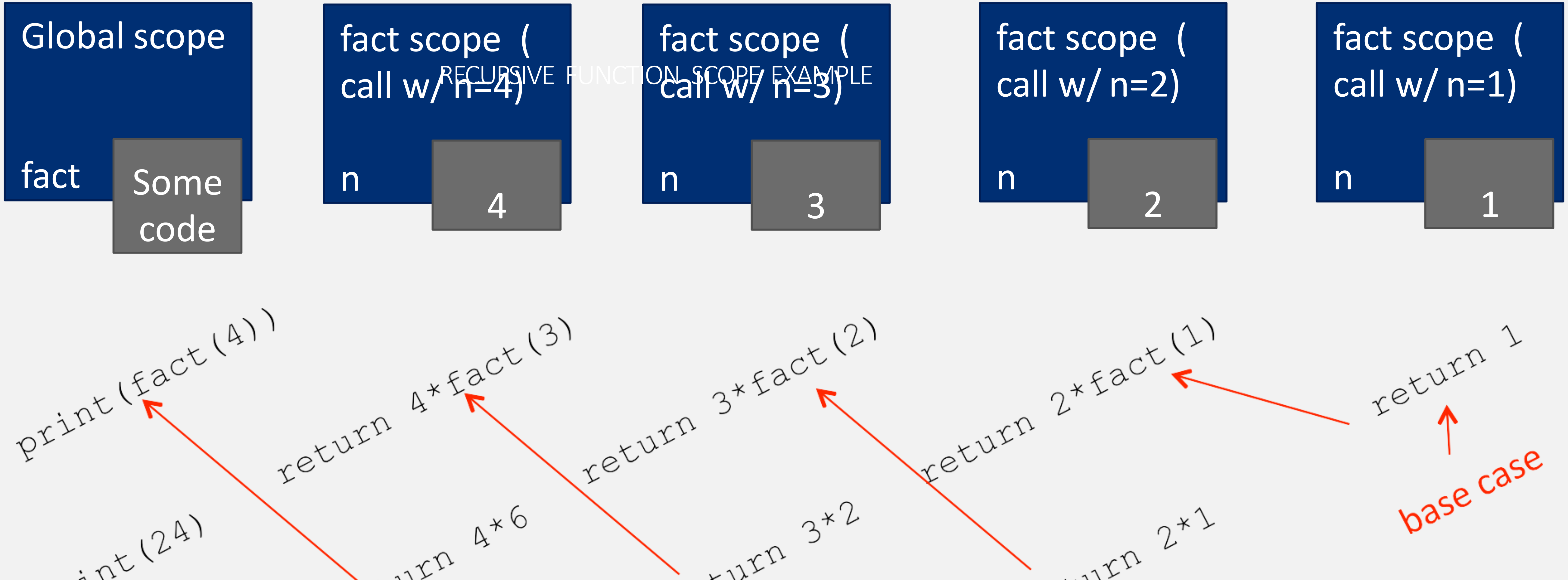
```
else:
    return n*factorial(n-1)
```

recursive step

RECURSIVE FUNCTION SCOPE EXAMPLE

```
def fact(n):
    if n >1:
        return n*fact(n-1)
    else:
        return 1

print(fact(4))
------------------
24
```



SOME OBSERVATIONS

- each recursive call to a function creates its **own scope/environment**
- **bindings of variables** in a scope are not changed by recursive call
- flow of control passes back to **previous scope** once function call returns value

using the same variable names but they are different objects in separate scopes

ITERATION VS FACTORIAL


```
def factorial_iter(n):
    prod = 1
    for i in range(1,n+1):
        prod *= i
    return prod
```

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```

- recursion may be simpler, more intuitive
- recursion may be efficient from programmer POV
- recursion may not be efficient from computer POV

MATHEMATICAL INDUCTION

- To prove a statement indexed on integers is true for all values of n:
  - Prove it is true when n is smallest value (e.g. n = 0 or n = 1)
  - Then prove that if it is true for an arbitrary value of n, one can show that it must be true for n+1

EXAMPLE OF INDUCTION

- 0 + 1 + 2 + 3 + … + n = (n(n+1))/2
- Proof:
  - If n = 0, then LHS is 0 and RHS is 0*1/2 = 0, so true
  - Assume true for some k, then need to show that

    0 + 1 + 2 + … + k + (k+1) = ((k+1)(k+2))/2
    - LHS is k(k+1)/2 + (k+1) by assumption that property holds for problem of size k
    - This becomes, by algebra, ((k+1)(k+2))/2
  - Hence expression holds for all n >= 0

RELEVANCE TO CODE?

- Same logic applies

```
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

- Base case, we can show that `mult` must return correct answer
- For recursive case, we can assume that `mult` correctly returns an answer for problems of size smaller than b, then by the addition step, it must also return a correct answer for problem of size b
- Thus by induction, code correctly returns answer

RECURSION WITH MULTIPLE BASE CASES

- Fibonacci numbers
  - Leonardo of Pisa (aka Fibonacci) modeled the following challenge
    - Newborn pair of rabbits (one female, one male) are put in a pen
    - Rabbits mate at age of one month
    - Rabbits have a one month gestation period
    - Assume rabbits never die, that female always produces one new pair (one male, one female) every month from its second month on.
    - How many female rabbits are there at the end of one year?

FIBONACCI

Ayer one month (call it 0) – 1 female

Ayer second month – still 1 female (now pregnant)

Ayer third month – two females, one pregnant, one not

In general, females(n) = females(n-1) + females(n-2)

- Every female alive at month n-2 will produce one female in month n;
- These can be added those alive in month n-1 to get total alive in month n

Month	Females
0	1

FIBONACCI

- Base cases:
  - Females(0) = 1
  - Females(1) = 1
- Recursive case
  - Females(n) = Females(n-1) + Females(n-2)

```python
def fib(x):
    """assumes x an int >= 0  re
       turns Fibonacci of x"""”
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)
```

- an example of a “divide and conquer” algorithm
- solve a hard problem by breaking it into a set of sub-problems such that:
  - sub-problems are easier to solve than the original
  - solutions of the sub-problems can be combined to solve the original
- Divide conquer algorithm 의 예
  - Fast Fourier Transform

DICTIONARY – CUSTOM INDEXING

STUDENT INFO

- so far, can store using separate lists for every info

```
names = ['Ana', 'John', 'Denise', 'Katy']
grade = ['B', 'A+', 'A', 'A']
course = [2.00, 6.0001, 20.002, 9.01]
```

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person

HOW TO UPDATE/RETRIEVE STUDENT INFO

```
def get_grade(student, name_list, grade_list, course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list[i]
    return (course, grade)
```

- **messy** if have a lot of different info to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists

A BETTER AND CLEANER WAY– A DICTIONARY

- nice to **index item of interest directly** (not always int)
- nice to use **one data structure**, no separate lists

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index
element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index by label
element

A PYTHON DICTIONARY

SEOUL NATIONAL UNIVERSITY

- store pairs of data
  - key
  - value

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

custom index by label

element

empty dictionary

```
my_dict = {}

grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

DICTIONARY LOOKUP

- similar to indexing into a list
- **looks up** the **key**
- **returns** the **value** associated with the key
- if key isn't found, get an error

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

`grades['John']` → evaluates to `'A+'`

`grades['Sylvan']` → gives a `KeyError`

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- **add** an entry

```
grades['Sylvan'] = 'A'
```

- **test** if key in dictionary

```
'John' in grades    → returns True
'Daniel' in grades  → returns False
```

- **delete** entry

```
del(grades['Ana'])
```

```
#This work, but not the best

if 'key1' in dict.keys():
  print "blah"
else:
  print "boo"
```

```
d = {"key1": 10, "key2": 23}

if "key1" in d:
    print("this will execute")

if "nonexistent key" in d:
    print("this will not")
------
this will execute
```

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- get an **iterable that acts like a tuple of all keys** no guaranteed order

`grades.keys()` → returns `['Denise','Katy','John','Ana']`

- get an **iterable that acts like a tuple of all values** no guaranteed order

`grades.values()` → returns `['A', 'A', 'A+', 'B']`

DICTIONARY KEYS AND VALUES

- values
  - any type (**immutable and mutable**)
  - can be **duplicates**
  - dictionary values can be lists, even other dictionaries!
- keys
  - must be **unique**
  - **immutable** type (`int`, `float`, `string`, `tuple`, `bool`)
    - actually need an object that is **hashable,** but think of as immutable as all immutable types are hashable
  - careful with `float` type as a key
- **no order** to keys or values!

```
d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}
```

LIST VS dict

- **ordered** sequence of elements
- look up elements by an integer index
- indices have an **order**
- index is an **integer**

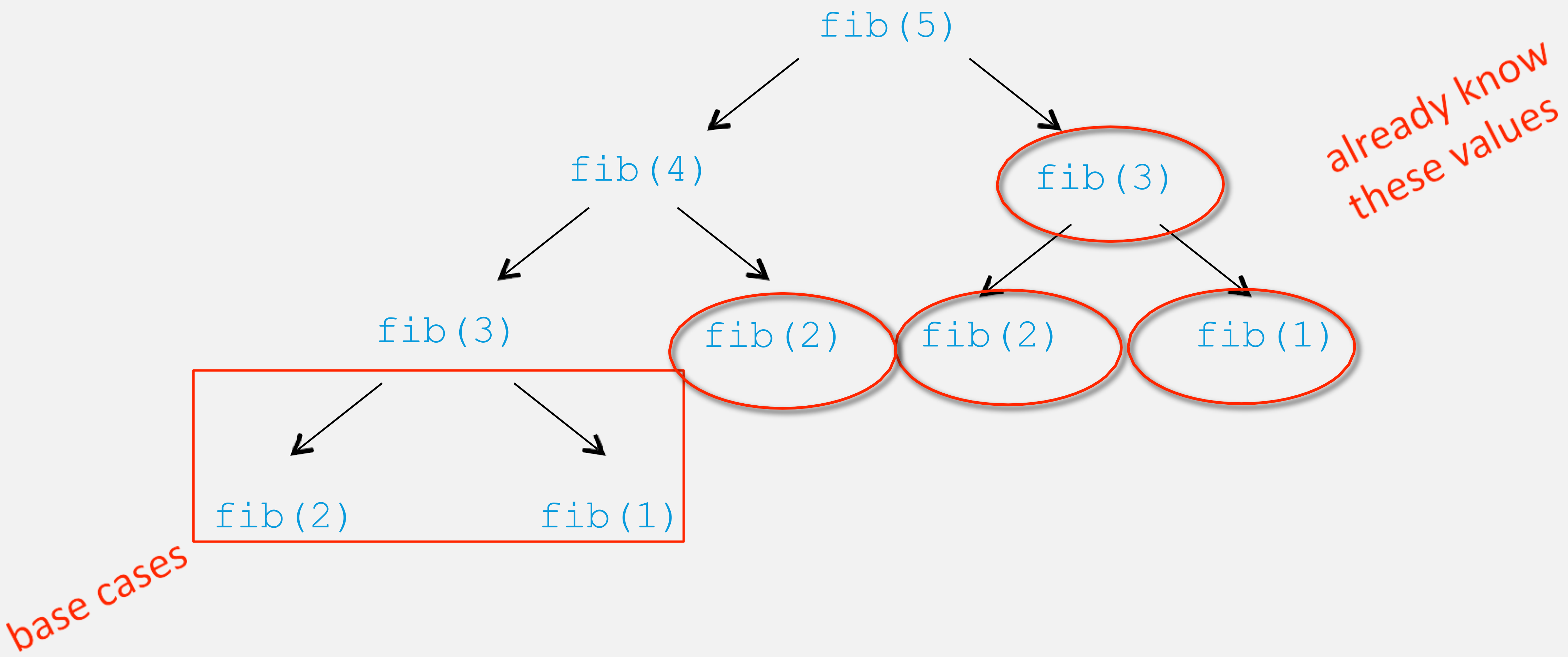
- **matches** “keys” to “values”
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

FIBONACCI RECURSIVE CODE

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

- two base cases
- calls itself twice
- this code is inefficient

INEFFICIENT FIBONACCI



- **recalculating** the same values many times!
- could keep **track** of already calculated values

FIBONACCI WITH A DICTIONARY - MEMORIZATION

```
def fib_efficient(n, d):
    if n in d:
        return d[n]
    else:
        ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
        d[n] = ans
    return ans

d = {1:1, 2:2}
print(fib_efficient(6, d))
--------------------
13
```

Method sometimes called " memoization"

Initialize dictionary with base cases

- do a **lookup first** in case already calculated the value
- **modify dictionary** as progress through function calls

EFFICIENCY GAINS

- Calling fib(34) results in 11,405,773 recursive calls to the procedure
- Calling fib_efficient(34) results in 65 recursive calls to the procedure
- Using dictionaries to capture intermediate results can be very efficient
- But note that this only works for procedures without side effects (i.e., the procedure will always produce the same result for a specific argument independent of any other computations between calls)

결론

- Python의 recursion 을 이용한 programming – 연역법을 이용하는데, 대체로 시간은 더 걸린다 (function call 의 overhead)
- Dictionary 구조 – custom indexing 이라 할 수 있다. 내부에서 큰 dictionary 에서 원하는 값을 빨리 찾는 hashing 등의 알고리즘이 구현되어 있다.