

# LECTURE 3

Python Basics Part 2

(원본 자료) <http://www.cs.fsu.edu/~carnahan/cis4930/>

# FUNCTIONAL PROGRAMMING TOOLS

지난 시간에 함수의 개념에 대해서 살펴보았다. 또한, 특별한 종류인 lambda함수도 살펴보았다.

Lambda 함수는 작지만 functional 언어에서 활용도가 많다.

이에 대해 간단히 살펴보자.

# LAMBDA FUNCTIONS

## Lambda 함수

- 함수 선언시 사용하는 `def`가 아니라 `lambda` 키워드를 이용
- 하나의 표현식으로만 제약적으로 사용가능

```
>>> def f(x):  
...     return x**2  
...  
>>> print f(8)  
64  
>>> g = lambda x: x**2  
>>> print g(8)  
64
```

# FUNCTIONAL PROGRAMMING TOOLS

## Filter

- `filter(function, sequence)`
  - 필터는 `sequence`의 아이템을 필터링 한다.  
각 `item`에 대해 `function(item)`을 호출하여 `true`인 `item`만 `return`해 준다
- `string`이나 `tuple` 타입이면 해당 타입으로, 그외의 경우에는 `list` 타입으로 결과를 `return`한다.

```
def even(x):  
    if x % 2 == 0:  
        return True  
    else:  
        return False
```

```
print(filter(even, range(0,30)))
```

---

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

# FUNCTIONAL PROGRAMMING TOOLS

## Map

- `map(function, sequence)`

맵(map)은 각 sequence에 함수를 적용하여 그 결과를 list로 출력한다.

- arguments 개수가 여러 개인 함수의 경우도 지원함

```
def square(x):  
    return x**2
```

```
print(map(square, range(0,11)))
```

---

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

# FUNCTIONAL PROGRAMMING TOOLS

## Map

- `map(function, sequence)`

맵(map)은 각 sequence에 함수를 적용하여 그 결과를 list로 출력한다.

- arguments개수가 여러 개인 함수의 경우도 지원함

```
def expo(x, y):  
    return x**y
```

```
print(map(expo, range(0,5), range(0,5)))
```

---

```
[1, 1, 4, 27, 256]
```

# FUNCTIONAL PROGRAMMING TOOLS

## Reduce

- `reduce(function, sequence)` 는 하나의 값을 return한다. `sequence`에 대해 `sequence`의 처음, 두번째 값을 함수 초기에 제공하고, 그 계산 결과를 다음번 함수 호출시 제공하는 형식으로 계산이 된다.

- `range(1,5)=[1,2,3,4]`

- `> fact(fact(fact(1,2),3),4)=1*2*3*4=24`

- 옵션으로 세번째 `argument`를 줄 수도 있는데 이 경우는 초기값을 주는 경우이다.

```
def fact(x, y):  
    return x*y
```

```
print(reduce(fact, range(1,5)))
```

# FUNCTIONAL PROGRAMMING TOOLS

lambda 함수와 map 함수를 같이 사용하면, 다음과 같이 따로 함수의 정의 없이 inline 으로 처리할 수 있다.

```
>>> print(map(lambda x: x**2, range(0,11)))  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```



# 자료구조(DATA STRUCTURES) 측면에서 더 살펴보기

- Lists (리스트)
  - Slicing
  - Stacks and Queues (스택과 큐)
- Tuples (튜플)
- Sets (셋) and FrozenSets
- Dictionaries (딕셔너리)
- 어떻게 자료구조(data structure)를 선택하는가?
- Collections (컬렉션)
  - Deques and OrderedDicts

# WHEN TO USE LISTS

- 동일 또는 다른 타입들의 원소들을 관리할 때
- 원소들을 순서를 줄 수 있음
- 원소를 추가 변경 가능
- index로 접근이 가능하지만, 원소로도 접근 가능
- 스택이나 큐가 필요할 때
- 원소들은 리스트에 여러 번 나올 수 있다.

# CREATING LISTS

리스트의 생성은 간단히 `empty list []`나 초기화를 이용하여 만듦

```
mylist1 = [] # Creates an empty list
mylist2 = [expression1, expression2, ...]
mylist3 = [expression for variable in sequence]
```

# CREATING LISTS

`list()` 생성자를 통해서도 만들수 있음

```
mylist1 = list()
```

```
mylist2 = list(sequence)
```

```
mylist3 = list(expression for variable in sequence)
```

두번째 예제의 `sequence` 나 `iterable`한 객체이면 되고  
다른 리스트를 집어넣는다면, 해당 `list`를 복제합니다.

# CREATING LISTS

assignment (=) 를 통해서 새로운 list를 생성하지 못한다.

```
# mylist1와 mylist2는 같은 list임  
mylist1 = mylist2 = []
```

```
# mylist3과 mylist4는 같은 list임  
mylist3 = []  
mylist4 = mylist3
```

```
mylist5 = []; mylist6 = [] # different lists
```

# ACCESSING LIST ELEMENTS

인덱스를 이용해서 접근

```
>>> mylist = [34, 67, 45, 29]
>>> mylist[2]
45
```

값(value)를 통한 접근

\* 만약 리스트에 해당 값이 없다면 exception 발생

```
>>> mylist = [34, 67, 45, 29]
>>> mylist.index(67)
1
```

# SLICING AND SLIDING

- 리스트의 길이(원소의 개수): `len(mylist)`.
- 슬라이스(Slicing): 리스트의 서브리스트를 획득

```
mylist[start:end] # items start to end-1
```

```
mylist[start:]    # items start to end of the array
```

```
mylist[:end]      # items from beginning to end-1
```

```
mylist[:]         # a copy of the whole array
```

- step만큼 띄어서 서브리스트 생성 가능

```
mylist[start:end:step] # start to end-1, by step
```

# SLICING AND SLIDING

- 인덱스에 음수를 주는 경우가 있음.
- 양수는 앞에서 부터 라면, 음수는 뒤에서 부터 를 의미함

```
mylist[-1]      # last item in the array
mylist[-2:]     # last two items in the array
mylist[:-2]     # everything except the last two items
```

- 예:

```
mylist = [34, 56, 29, 73, 19, 62]
mylist[-2]      # yields 19
mylist[-4::2]   # yields [29, 19]
```



# INSERTING/REMOVING ELEMENTS

- 기존리스트에 원소를 추가하고 싶으면, `append()` 사용

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.append(47)
>>> mylist
[34, 56, 29, 73, 19, 62, 47]
```

- `extend()`에 원소가 아니라 리스트를 주면 그 리스트 전부 추가>

```
>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.extend([47, 81])
>>> mylist
[34, 56, 29, 73, 19, 62, 47, 81]
```

# INSERTING/REMOVING ELEMENTS

- `insert(pos, item)`는 주어진 위치(`pos`)에 원소 추가  
위치에 `negative indexing`(음수) 사용 가능

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.insert(2, 47)
>>> mylist
[34, 56, 47, 29, 73, 19, 62]
```

- `remove()`는 주어진 원소를 리스트에서 삭제. 원소가 다수일 때는 맨 처음 원소만 삭제  
리스트에 매칭되는 원소가 없는 경우에는 `exception` 발생

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.remove(29)
>>> mylist
[34, 56, 73, 19, 62]
```

# LISTS AS STACKS

- 자료구조인 스택으로 사용가능
- 스택은 push와 pop연산이 있는데, 리스트의 append()와 pop()을 이용해서 구현 가능
- pop()은 일반적으로 마지막 원소를 삭제 후 리턴하는데, pop(index)처럼 인덱스를 주는 경우 해당 index에 해당하는 원소를 삭제 후 리턴함

```
>>> stack = [34, 56, 29, 73, 19, 62]
>>> stack.append(47)
>>> stack
[34, 56, 29, 73, 19, 62, 47]
>>> stack.pop()
47
>>> stack
[34, 56, 29, 73, 19, 62]
```

# LISTS AS QUEUES

- 또한 리스트는 큐(queue)로 활용이 가능한데, 이는 `insert()` 와 `pop()`이 모두 `index` 파라미터를 지원하기 때문. 하지만, 리스트이 맨 앞의 원소를 `insert`하거나 `pop`하는 것은 상대적으로 마지막 원소를 `append`하거나 `pop`하는 거 대비 속도가 느림
- 그래서, `collection` 모듈에 있는 `deque`를 사용함

```
>>> from collections import deque
>>> queue = deque([35, 19, 67])
>>> queue.append(42)
>>> queue.append(23)
>>> queue.popleft()
35
>>> queue.popleft()
19
>>> queue
deque([67, 42, 23])
```

# OTHER OPERATIONS

- 리스트의 `count(x)`는 주어진 원소 `x`가 리스트 안에서 몇 개나 있는지를 리턴함

```
>>> mylist = ['a', 'b', 'c', 'd', 'a', 'f', 'c']
>>> mylist.count('a')
2
```

- 정렬 방식에는 `sort()`와 `reverse()`가 있으며, 이는 in place 방식이다.
- 기존 리스트를 유지하고, 정렬된 리스트를 따로 얻고 싶은 경우, `sorted(mylist)`나 `reversed(mylist)` 함수를 이용한다.

```
>>> mylist = [5, 2, 3, 4, 1]
>>> mylist.sort()
>>> mylist
[1, 2, 3, 4, 5]
>>> mylist.reverse()
>>> mylist
[5, 4, 3, 2, 1]
```

# CUSTOM SORTING

- built-in 함수인 `sorted()`와 리스트의 `sort()`는 다음과 같이 추가적인 argument를 가질 수 있다.

```
sorted(iterable[, cmp[, key[, reverse]])
```

- *cmp 파라미터는 argume*파라미터는 두 개의 파라미터가 주어졌을 경우, 두 개를 비교하여 첫번째 파라미터가 두번째 파라미터보다 작으면 `negative`, 같으면 `zero`, 크면 `postive`을 `reture`한다.
- *key 파라미터는 리스트의 원소로 부터 비교를 위한 key를 추출하는데 사용되는 함수 이다.*
- *reverse* 파라미터는 Boolean 값으로 `true`면 비교 결과를 `reverse`하며 정렬을 수행한다.

# CUSTOM SORTING

```
>>> mylist = ['b', 'A', 'D', 'c']
>>> mylist.sort(cmp = lambda x,y: cmp(x.lower(), y.lower()))
>>> mylist
['A', 'b', 'c', 'D']
```

또는,

```
>>> mylist = ['b', 'A', 'D', 'c']
>>> mylist.sort(key = str.lower)
>>> mylist
['A', 'b', 'c', 'D']
```

str.lower() 는 string의 built-in함수

# WHEN TO USE SETS

- 원소(elements)가 유닉해야함(unique).
- 추가/변경이 가능
- 수학적인 set 연산 가능 (교집합, 합집합, 차집합 등)
- 원소로써 nested list나 set이나 dictionary를 가질 수 없음



# CREATING SETS

- set생성자를 이용하여 empty set생성

```
myset = set()  
myset2 = set([]) # both are empty sets
```

- 초기화된 set을 생성할 때는 set생성자나 {}을 사용함. (주의사항: empty curly braces {} 만으로 정의할 경우 set이 아니라 dictionary가 생성됨)

```
myset = set(sequence)  
myset2 = {expression for variable in sequence}
```

# HASHABLE ITEMS

set에서 중복된 원소를 찾는 내부적인 방법은 hash를 이용함. 따라서 모든 set안의 원소들은 *hashable*해야함

모든 불변한 objects들은 hashable하다. 하지만, 변경가능한 리스트나 딕셔너리의 경우에는 hashable하지 않습니다. 일반적인 사용자 정의 object들도 기본적으로 hashable합니다.

# MUTABLE OPERATIONS

- `add(x)`는 set에 원소 `x`가 없다면 추가한다.
- `remove(x)`, `discard(x)`는 set에서 원소 `x`를 삭제한다.
- `pop()`은 임의의 원소 하나를 삭제한다. 만약 set이 비어있으면, `error`를 낸다
- `clear()`는 set의 모든 원소를 삭제한다.
- 단 위 연산은 `frozen sets`에서는 사용불가

```
>>> myset = {x for x in 'abracadabra'}
>>> myset
set(['a', 'b', 'r', 'c', 'd'])
>>> myset.add('y')
>>> myset
set(['a', 'b', 'r', 'c', 'd', 'y'])
>>> myset.remove('a')
>>> myset
set(['b', 'r', 'c', 'd', 'y'])
>>> myset.pop()
'b'
>>> myset
set(['r', 'c', 'd', 'y'])
```

# MUTABLE OPERATIONS CONTINUED

*set |= other*

합집합 : Update the set, adding elements from all others.

*set &= other*

교집합 : Update the set, keeping only elements found in it and all others.

*set -= other*

차집합 : Update the set, removing elements found in others.

*set ^= other*

합집합 – 교집합 : Update the set, keeping only elements found in either set, but not in both.

# MUTABLE OPERATIONS CONTINUED

```
>>> s1 = set('abracadabra')
>>> s2 = set('alacazam')
>>> s1
set(['a', 'b', 'r', 'c', 'd'])
>>> s2
set(['a', 'l', 'c', 'z', 'm'])
>>> s1 |= s2
>>> s1
set(['a', 'b', 'r', 'c', 'd', 'l', 'z', 'm'])
>>> s1 = set('abracadabra')
>>> s1 &= s2
>>> s1
set(['a', 'c'])
```

# SET OPERATIONS

- 다음 연산은 set 과 frozenset 모두에 적용 가능하다.
- 비교 연산자 ( $\geq$ ,  $\leq$ ) 는 test하나의 set이 다른 set의 superset인지 subset인지를 판별한다. 비교 연산자 ( $>$ ,  $<$ ) 는 온전한 supersets/subsets 인지를 판별한다.

```
>>> s1 = set('abracadabra') # {a, b, r, c, d}
>>> s2 = set('bard')       # {a, b, r, d}
>>> s1 >= s2
True
>>> s1 > s2
True
>>> s1 <= s2
False
```

# SET OPERATIONS

```
>>> s1 = set('abracadabra')
>>> s1
set(['a', 'b', 'r', 'c', 'd'])
>>> s2 = set('alacazam')
>>> s2
set(['a', 'l', 'c', 'z', 'm'])
>>> s1 | s2
set(['a', 'b', 'r', 'c', 'd', 'l', 'z', 'm'])
>>> s1 & s2
set(['a', 'c'])
>>> s1 - s2
set(['b', 'r', 'd'])
>>> s1 ^ s2
set(['b', 'r', 'd', 'l', 'z', 'm'])
```

# OTHER OPERATIONS

- `s.copy()` set의 shallow copy를 리턴한다.
- `s.isdisjoint(other)` 두 set에 중복되는 원소가 없으면 True
- `s.issubset(other)` s가 other의 subset이면 True
- `len()`, `in`, `not` 연산을 지원



# WHEN TO USE TUPLES

- 저장된 원소들을 변경시킬 필요가 없는 경우
- 성능에 신경 쓸 필요가 있을 때
- 논리적으로 불변한 pairs나 triples를 저장하기를 원할 때(?)

# CONSTRUCTING TUPLES

- () 을 통해서 Tuple을 생성
- comma로 구분된 list로 부터 Tuple을 생성할 수 있으며, 이 때 괄호()는 있어도 되고 없어도 된다.
- 기이한 점 : 원소 하나로 Tuple을 정의할 때는 마지막에 comma (trailing comma) 추가

```
>>> t1 = (1, 2, 3, 4)
>>> t2 = "a", "b", "c", "d"
>>> t3 = ()
>>> t4 = ("red", )
```

# TUPLE OPERATIONS

Tuples 은 리스트와 비슷하며, 대부분의 연산자를 공통으로 지원한다.

- 원소 접근 방법 : (e.g. `t1[2]`, `t1[2:5]`)
- Tuple의 길이 : `len(t1)`
- 불변하는 sequence type 연산자들 모두 지원
  - `+`, `*`
  - `in`, `not in`
  - `min(t)`, `max(t)`, `t.index(x)`, `t.count(x)`

# PACKING/UNPACKING

Tuple packing은 아이템들을 “pack” 하여 하나의 tuple을 만드는 것이다. 또한 Python의 multiple assignment 특징을 이용하여 “unpack”이 가능하다.

```
>>> s = ("Susan", 19, "CS") # tuple packing
>>> (name, age, major) = s # tuple unpacking
>>> name
'Susan'
>>> age
19
>>> major
'CS'
```

# WHEN TO USE DICTIONARIES

- key:value pairs의 관계가 필요할 때
- 특정 key를 가지고 데이터에 빠르게 접근하고자 할 때
- key:value pairs의 추가/변경/삭제가 가능

# CONSTRUCTING A DICTIONARY

- 빈 dictionary를 만들 경우 empty curly braces {} 나 dict() 생성자 이용
- dictionary 초기화는 curly braces {} 사이에 key:value pair 를 위치
- 모든 key는 *hashable* objects 이어야 함

```
>>> d1 = {}
>>> d2 = dict() # both empty
>>> d3 = {"Name": "Susan", "Age": 19, "Major": "CS"}
>>> d4 = dict(Name="Susan", Age=19, Major="CS")
>>> d5 = dict(zip(['Name', 'Age', 'Major'], ["Susan", 19, "CS"]))
>>> d6 = dict([('Age', 19), ('Name', "Susan"), ('Major', "CS")])
```

Note: zip 은 두 개의 같은 길이 collections을 하나로 합치는 연산으로, 각 collection의 대응되는 원소를 하나의 tuple로 만들어줌

# ACCESSING THE DICTIONARY

dictionary 데이터에 접근하기 위해 인덱스처럼 key 사용  
만약 key가 dictionary에 없으면 error 발생

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1['Age']
19
>>> d1['Name']
'Susan'
```

# UPDATING A DICTIONARY

key:value pair를 acces하는 형식으로 쉽게 새로운 pair를 삽입하거나 수정 가능  
del 키워드는 하나의 single key:value pair를 삭제하거나 전체 dictionary 삭제 가능  
clear() 함수는 dictionary를 초기화함

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1['Age'] = 21
>>> d1['Year'] = "Junior"
>>> d1
{'Age': 21, 'Name': 'Susan', 'Major': 'CS', 'Year': 'Junior'}
>>> del d1['Major']
>>> d1
{'Age': 21, 'Name': 'Susan', 'Year': 'Junior'}
>>> d1.clear()
>>> d1
{}
```



# BUILT-IN DICTIONARY METHODS

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1.has_key('Age') # True if key exists
True
>>> d1.has_key('Year') # False otherwise
False
>>> d1.keys() # Return a list of keys
['Age', 'Name', 'Major']
>>> d1.items() # Return a list of key:value pairs
[('Age', 19), ('Name', 'Susan'), ('Major', 'CS')]
>>> d1.values() # Returns a list of values
[19, 'Susan', 'CS']
```

**Note:** `in`, `not in`, `pop(key)`, and `popitem()` are also supported.

# ORDERED DICTIONARY

Dictionaries는 key가 입력된 순서를 기억하지 않는다.

Ordered dictionary는 키의 입력 순서를 기억하며, 일반적인 dictionary 명령을 지원한다.

OrderedDict은 추가적인 명령어를 지원한다.

```
OrderedDict.popitem(last=True)    # pops items in LIFO order
```

# ORDERED DICTIONARY

```
>>> # regular unsorted dictionary
```

```
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}
```

```
>>> # dictionary sorted by key
```

```
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
```

```
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])
```

```
>>> # dictionary sorted by value
```

```
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
```

```
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])
```

```
>>> # dictionary sorted by length of the key string
```

```
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
```

```
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```