IPA 주관 인공지능센터 기본(fundamental) 과정

- GitHub link: here
- E-Mail: windkyle7@gmail.com

High-Order Function (orthogonal)

FP Principles

- Treat computation as the evaluation of math functions
- Pure Function
- High-order Function
- Avoid Mutation

iterator를 받아 오름차순으로 정렬해주는 함수를 정의한다.

In [1]:
```
def x(nums):
    nums.sort()
    return nums
```

In [2]:
```
x([1, 3, 2])
```

Out[2]:
```
[1, 2, 3]
```

x 함수를 호출하면 t의 값이 변하게 된다.

In [3]:
```
t = [1, 3, 2]
```

In [4]:
```
x(t)
```

Out[4]:
```
[1, 2, 3]
```

In [5]:
```
t
```

Out[5]:
```
[1, 2, 3]
```

위와 같이 함수 호출 시 input값 자체가 변한다면 `pure function`이 아니다.

dis 패키지

`dis`는 기계어로 변환되는 것을 볼 수 있는 패키지 라이브러리다.

In [6]:
```
import dis
```

아래는 accumulation 패턴으로 구현한 함수 a를 정의하였다. `dis.dis` 함수를 이용하면 a라는 함수가 실행이 될 때 기계어 코드로 변환되는 과정을 확인해 볼 수 있다.

In [7]:
```
def a():
    for i in [1, 2, 3, 4, 5]:
        print(i)
```

In [8]:
```
dis.dis(a)
```

```
  2           0 SETUP_LOOP              20 (to 22)
              2 LOAD_CONST               6 ((1, 2, 3, 4, 5))
              4 GET_ITER
        >>    6 FOR_ITER                12 (to 20)
              8 STORE_FAST               0 (i)

  3          10 LOAD_GLOBAL              0 (print)
             12 LOAD_FAST                0 (i)
             14 CALL_FUNCTION            1
             16 POP_TOP
             18 JUMP_ABSOLUTE            6
        >>   20 POP_BLOCK
        >>   22 LOAD_CONST               0 (None)
             24 RETURN_VALUE
```

위의 x라는 함수와는 달리 밑의 xx 함수처럼 임시로 값을 받은 뒤에 임시로 값을 받은 식별자를 변환하면 input 값은 변하지 않는다.

In [9]:
```
def xx(nums):
    nums2 = nums[:]
    nums2.sort()
    return nums2
```

In [10]:
```
t = [1, 3, 2]
u = xx(t)
```

In [11]:
```
t
```

```
Out[11]: [1, 3, 2]
```

```
In [12]: u
Out[12]: [1, 2, 3]
```

Iteration - Procedural Style

```
In [13]: def sum1(nums):
             total = 0
             for num in nums:
                 total += num
             return total
```

Recursion - Functional Style

```
In [14]: def sum2(nums):
             return 0 if len(nums) == 0 else nums[0] + sum2(nums[1:])
```

- sum1 방식과 sum2 방식을 비교하였을 때, sum2 함수가 더 간결하다.
- 그러나 python에서는 속도가 느려지기 때문에 보통 sum2처럼 Recursion 방식으로 작성하지 않는다.

FP - 반복문을 줄이기 위한 다양한 기법들

map(function_apply, iterator)

- map을 이용하면 반복문을 줄일 수 있다는 강점이 있다.
- map은 iterator의 각 요소를 function_apply에 대한 결과를 반환한다.

```
In [15]: a = [1, 2, 3, 4, 5]
```

map 함수를 이용하여 제곱해본다.

```
In [16]: list(map(lambda x: x * x, a))
Out[16]: [1, 4, 9, 16, 25]
```

- 아래와 같이 comprehension을 이용하면 더 간단해진다.
- 그러나 comprehension은 복잡한 식이 들어갈 수 없다는 한계점이 존재하기에 복잡한 식을 표현할 때는 map 함수를 사용하면 편리하다.

```
In [17]: [x * x for x in a]
Out[17]: [1, 4, 9, 16, 25]
```

filter(function_apply, iterator)

- filter도 map과 마찬가지로 조건식에 만족하는 결과값을 반환한다.
- 다만 filter는 식에 만족하는 결과(True)에 대한 값만을 반환한다.

```
In [18]: list(filter(lambda x: x > 5, [2, 3, 5, 6, 7, 8]))
Out[18]: [6, 7, 8]
```

reduce

- functools 패키지에서 제공하는 함수
- 주로 어떤 함수를 반복 수행하고 그 결과를 반환할 때 유용하다.

```
In [19]: from functools import reduce
```

reduce를 사용하여 range(10)에 5를 더한다.

```
In [20]: reduce(lambda x, y: x + [y + 5], range(10), [])
Out[20]: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

위의 코드를 comprehension을 사용하여 더 간단히 표현할 수 있다.

짚고 넘어가기

- 패러다임 = 생각의 방식
- 존재하는 모든 방법들 중 가장 효율적인 방식을 선택하는 것이 중요하다.

```
In [21]: print([x + 5 for x in range(10)])
         [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

map, filter, reduce, comprehension을 이용하여 홀수만 출력하는 방법

In [22]:
```
def isOdd(n): return n % 2
```

In [23]:
```
# map
print(list(map(lambda x: x if isOdd(x) else False, range(10))))

# filter
print(list(filter(isOdd, range(10))))

# reduce
print(reduce(lambda l, x: l + [x] if isOdd(x) else l, range(10), []))

# comprehension
print([x for x in range(10) if isOdd(x)])
```

```
[False, 1, False, 3, False, 5, False, 7, False, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
```

Tip

toolz python

- https://toolz.readthedocs.io/en/latest/api.html
- FP 기법으로 구현된, FP 기법을 손쉽게 구현할 수 있도록 만들어진 오픈 API
- 다만 오픈 API의 한계상 버전이 업데이트되면 유지보수 하기가 힘들 수 있다.

다양한 패키지들

operator

In [24]:
```
import operator
```

In [25]:
```
type(operator)
```

Out[25]:
```
module
```

In [26]:
```
dir(operator)
```

Out[26]:
```
['__abs__',
 '__add__',
 '__all__',
 '__and__',
 '__builtins__',
 '__cached__',
 '__concat__',
 '__contains__',
 '__delitem__',
 '__doc__',
 '__eq__',
 '__file__',
 '__floordiv__',
 '__ge__',
 '__getitem__',
 '__gt__',
 '__iadd__',
 '__iand__',
 '__iconcat__',
 '__ifloordiv__',
 '__ilshift__',
 '__imatmul__',
 '__imod__',
 '__imul__',
 '__index__',
 '__inv__',
 '__invert__',
 '__ior__',
 '__ipow__',
 '__irshift__',
 '__isub__',
 '__itruediv__',
 '__ixor__',
 '__le__',
 '__loader__',
 '__lshift__',
 '__lt__',
 '__matmul__',
 '__mod__',
 '__mul__',
 '__name__',
 '__ne__',
 '__neg__',
 '__not__',
 '__or__',
 '__package__',
 '__pos__',
 '__pow__',
 '__rshift__',
```

```
'__setitem__',
'__spec__',
'__sub__',
'__truediv__',
'__xor__',
'_abs',
'abs',
'add',
'and_',
'attrgetter',
'concat',
'contains',
'countOf',
'delitem',
'eq',
'floordiv',
'ge',
'getitem',
'gt',
'iadd',
'iand',
'iconcat',
'ifloordiv',
'ilshift',
'imatmul',
'imod',
'imul',
'index',
'indexOf',
'inv',
'invert',
'ior',
'ipow',
'irshift',
'is_',
'is_not',
'isub',
'itemgetter',
'itruediv',
'ixor',
'le',
'length_hint',
'lshift',
'lt',
'matmul',
'methodcaller',
'mod',
'mul',
'ne',
'neg',
'not_',
'or_',
'pos',
'pow',
'rshift',
'setitem',
'sub',
'truediv',
'truth',
'xor']
```

In [27]:
```
# 함수 호출을 통해 연산
print('3 + 4 =', operator.add(3, 4))
print('3 - 4 =', operator.sub(3, 4))
```

```
3 + 4 = 7
3 - 4 = -1
```

In [28]:
```
# Same as a<=b
print(operator.le(2, 3))
```

```
True
```

python 기본 연산자를 사용하면 되는데 굳이 operator 패키지를 사용하는 이유가 무엇일까?

- 아래의 partial 패키지를 통해 operator 패키지를 어떤 식으로 활용하는지 확인해볼 수 있다.

partial

- partial은 closur 함수처럼 사용할 수 있도록 막강한 기능을 제공해준다.

In [29]:
```
from functools import partial
```

In [30]:
```
# Higher-order function
x = partial(operator.add, 2)
```

아래처럼 closur 함수처럼 사용이 가능하다.

In [31]:
```
x(3)
```

Out[31]:
```
5
```

각각 1~10을 더해주는 함수들이 반환된 것을 확인할 수 있다

In [32]:
```
y = [partial(operator.add, i) for i in range(1, 11)]
```

In [33]:
```
y
```

Out[33]:
```
[functools.partial(<built-in function add>, 1),
 functools.partial(<built-in function add>, 2),
 functools.partial(<built-in function add>, 3),
 functools.partial(<built-in function add>, 4),
 functools.partial(<built-in function add>, 5),
 functools.partial(<built-in function add>, 6),
 functools.partial(<built-in function add>, 7),
 functools.partial(<built-in function add>, 8),
 functools.partial(<built-in function add>, 9),
 functools.partial(<built-in function add>, 10)]
```

각각의 함수를 호출하여 출력해본다.

In [34]:
```
y[0](1)
```

Out[34]:
```
2
```

In [35]:
```
y[0](2)
```

Out[35]:
```
3
```

In [36]:
```
y[1](100)
```

Out[36]:
```
102
```

In [37]:
```
y[9](3)
```

Out[37]:
```
13
```