

# 코스콤 교육 강의 교재

신승우

KAIST, School of Computing

# Contents

<b>1</b>	<b>Introduction and Environment Settings</b>	<b>2</b>
1.1	환경 설정 . . . . .	2
1.1.1	Installing Python . . . . .	2
1.1.2	IDE . . . . .	2
1.1.3	Git . . . . .	3
1.2	Introduction to Python Language . . . . .	3
1.2.1	파이썬 데이터 타입 . . . . .	3
1.2.2	파이썬 문법 : loops, conditionals . . . . .	6
1.2.3	파이썬 문법 : 함수, 클래스 . . . . .	6
1.3	파이썬 인터프리터의 이해 . . . . .	13
<b>2</b>	<b>Computation과 데이터구조의 이해</b>	<b>16</b>
2.1	데이터구조 vs Abstract Data Type . . . . .	16
2.2	Concept of Algorithm . . . . .	20
<b>3</b>	<b>List of ADTs</b>	<b>22</b>
3.1	List . . . . .	22
3.1.1	ADT List Specification . . . . .	22
3.2	Queue . . . . .	23
3.2.1	ADT Queue Specification . . . . .	23
3.2.2	ADT Priority Queue Specification . . . . .	23
3.2.3	Problem Solving Using Queue . . . . .	24
3.3	Tree . . . . .	24
3.3.1	ADT Specification of Tree . . . . .	24
3.4	Graph . . . . .	24
3.4.1	ADT Specification of Graph . . . . .	24

# Chapter 1

## Introduction and Environment Settings

### 1.1 환경 설정

#### 1.1.1 Installing Python

본 수업에서의 실습은 다른 라이브러리를 필요로 하지 않으므로 파이썬 개발에 필요한 최소한의 환경만 설정해주면 충분합니다. 본 환경설정법은 장고걸스 튜토리얼<sup>1</sup>의 해당 부분을 참고하여 작성되었습니다.

**Windows** 먼저, 사용 중인 컴퓨터 윈도우 운영체제가 32비트인지 64비트인지 확인해야 합니다. 확인법은 [마이크로소프트 링크](#)에서 찾아보실 수 있습니다. 이제 윈도우 용 파이썬 설치파일을 [파이썬 공식 다운로드 링크](#)에서 다운로드 할 수 있습니다. 본 교재에서는 파이썬 3 버전을 사용할 것이므로, Latest Python 3 Release - Python x.x.x 를 찾아서 다운로드받으면 됩니다. 64 비트 버전의 Windows인 경우 Windows x86-64 executable installer를 다운로드하시고, 이외에는 Windows x86 executable installer 을 다운로드하면 됩니다. 설치 프로그램을 다운로드 한 후에 실행하고 지시 사항을 따르세요.

설치하는 동안 "Setup(설치하기)"이라고 표시된 창이 나타납니다. 다음과 같이 "Add Python 3.x to PATH(python3.x를 경로에 추가)"체크 박스를 체크하고 "Install Now(지금 설치하기)"를 클릭하세요.

**OS X** [파이썬 공식 사이트](#)로 가서 파이썬 설치 파일을 다운 받으세요.

- Mac OS X 64-bit/32-bit installer 파일을 다운받습니다.
- python-3.6.1-macosx10.6.pkg을 더블클릭해 설치합니다.

**Linux** 이미 파이썬 3이 설치되어 있을 것입니다.

#### 1.1.2 IDE

IDE나 코드 에디터는 개인 취향에 따라서 아래의 옵션 중 골라서 설치하시면 됩니다.

- PyCharm : 가장 범용적으로 쓰이는 파이썬 IDE입니다.
- Spyder
- Notepad++
- Atom

---

<sup>1</sup>[장고걸스 튜토리얼 : 파이썬 설치](#)

- Sublime Text
- Eclipse : PyDev 플러그인 사용
- Visual Studio Code : 파이썬 플러그인 사용

### 1.1.3 Git

Git은 버전 관리 도구로써, 본 수업에서 필수적으로 사용되는 것은 아니지만 사용하는 것에 익숙해지면 쓰일 곳이 많습니다. 특히, 개발자에게 자신이 작성한 코드를 관리하는 것에 이만한 틀이 없습니다. 여기서는 따로 Git의 사용법을 다루지는 않습니다. 본 단락 역시 장고걸스 튜토리얼의 해당 부분<sup>2</sup>을 참고하였습니다.

Windows, OS X git-scm를 [링크](#)에서 다운로드하면 됩니다.

Linux sudo apt-get install git 으로 git을 다운로드할 수 있습니다.

## 1.2 Introduction to Python Language

---

### Implementation 1.1: Hello World! (hello.py)

---

```
1 def main():
2     print('Hello World!')
3
4 main()
```

---

본 단락에서는 파이썬 언어에 대해서 간단히 짚고 넘어갑니다.

### 1.2.1 파이썬 데이터 타입

파이썬은 기본적으로 다음의 벨트인 데이터 타입을 지원<sup>34</sup>합니다.

- Boolean Type : 참/거짓 값을 나타내는 타입입니다.
- Numeric Types : 일반적으로 쓰이는 숫자를 나타내는 타입입니다.
- Sequential Types : 배열 형태의 타입입니다.
- Mapping Types : key-value 순서쌍 형태의 타입입니다.<sup>5</sup>

아래에서는 각 타입의 종류와 다양한 연산법에 대해서 알아볼 것입니다.

**Boolean Type** 참(True), 거짓(False)값을 나타냅니다. Boolean 값들은 and, or, not 연산이 가능합니다. 연산의 결과는 아래와 같습니다.

---

### Implementation 1.2: Boolean Example (example\_bool1.py)

---

```
1 assert (True and True) == True
2 assert (True and False) == False
3 assert (False and True) == False
4 assert (False and False) == False
```

---

<sup>2</sup>장고걸스 튜토리얼 : [배포](#)

<sup>3</sup>참조 링크 1(위키북스), 참조 링크 2(공식 문서)

<sup>4</sup>여기 리스트된 데이터형이 전부는 아니지만, 중요한 데이터형들이니 잘 알아두시길 권장합니다.

<sup>5</sup>프로그래밍 지식이 있으신 분은 파이썬에서의 dict가 해쉬라고 생각하시면 좋습니다.

```
5 assert (True or True) == True
6 assert (True or False) == True
7 assert (False or True) == True
8 assert (False or False) == False
9 assert (not True) == False
10 assert (not False) == True
```

---

**Numeric Types** int, float, complex가 있습니다. 각각 정수, 실수<sup>6</sup>, 그리고 복소수를 나타냅니다.

---

**Implementation 1.3:** Numeric Types (*example\_numeric1.py*)

---

```
1 a = 1
2 print(type(a)) # <type 'int'>
3 b = 1.0
4 print(type(b)) # <type 'float'>
5 c = 1.0 + 1j # j 알파벳은 그냥 변수지만, 숫자 뒤에 붙어 오면 허수로
      인식합니다 .
6 print(type(c)) # <type 'complex'>
7 print('hello world!')
```

---

파이썬은 일반적인 사칙연산을 지원합니다. 아래에서 어떤 식으로 사칙연산이 사용되는지 볼 수 있습니다.

---

**Implementation 1.4:** Operations for Numeric Types (*example\_numeric2.py*)

---

```
1 print(1 + 2) # sum of x and y
2 print(3 - 1) # difference of x and y
3 print(2 * 4) # product of x and y
4 print(3 / 2) # quotient of x and y
5 print(3 // 2) # floored quotient of x and y
6 print(3 % 2) # remainder of x / y
7 print(-1) # x negated
8 print(+1) # x unchanged
9 print(abs(-2)) # absolute value or magnitude of x
10 print(int(3.2)) # x converted to integer
11 print(float(2)) # x converted to floating point
```

---

**Sequential Types** 파이썬에서는 list, tuple, range, string 등의 배열 형태의 데이터형을 지원합니다. 여기서는 문자열 데이터형 string에 대해서 따로 다루지는 않으며, 더 자세한 정보는 [공식 Documentation](#)을 참고하시면 됩니다.

---

**Implementation 1.5:** Sequential Types (*example\_sequence1.py*)

---

```
1 a = [1,2,3,4] # list
2 b = (1,2,3,4) # tuple
3 c = range(10) # range
4 d = 'hello world!' # string
```

---

Sequence 데이터형들은 다음의 연산들을 지원합니다.

---

<sup>6</sup>차후에 다루겠지만, 정확하게 실수를 나타내는 것은 불가능합니다. 더 정확하게는, 모든 실수를 정확하게 나타내는 것은 불가능합니다. 여기서의 float은 C언어에서의 double과 같다고 보는 것이 정확합니다.

- `a in d` : 배열(d) 안에 특정 원소(a)가 있는지를 검사합니다.
- `+` : 배열 두 개를 이어서 새로운 배열을 만듭니다. 같은 데이터형이여야 합니다.
- `d[i]` : d의 i번째 원소를 반환합니다.
- `d[i:j:k]` : d의 i번째 원소부터 j번째 원소까지, k번째 원소마다 선택하여 리스트를 만들어 반환합니다.
- `d.index(elem)` : d에서 elem 이 처음으로 나오는 위치를 반환합니다.

---

**Implementation 1.6:** Operations for Sequential Types (example\_sequence2.py)

---

```
1 from example_sequence1 import *
2
3 print('e' in d) # True
4 print([1,2,3] + [4,5,6]) # [1,2,3,4,5,6]
5 print(d[1]) # 'e'
6 print(d[1:3]) # 'el'
7 print(d[1:6:2]) # 'el '
8 print(d[::-1]) # !dlrow olleh'
9 print(len(d)) # 12
10 for idx, elem in enumerate(d):
11     print(idx, elem)
```

---

**Mapping Types** key-value 쌍을 저장하는 데이터형으로, 파이썬에서는 dict가 있습니다.

---

**Implementation 1.7:** Mapping Types (example\_dict1.py, line 7-26 omitted)

---

```
1 num2alphabet = \
2     { 1 : 'a',
3      2 : 'b',
4      3 : 'c',
5      4 : 'd',
6      5 : 'e',
7      26 : 'z', }
```

---

dict는 다음의 연산을 지원합니다.

- `d[key]` : dict에서 key에 해당되는 value를 반환합니다.
- `d[key] = val` : dict에서 key에 해당되는 value를 val로 업데이트합니다.
- `d.keys()` : dict의 key들을 반환합니다.

---

**Implementation 1.8:** Operations for Mapping Types (example\_dict2.py)

---

```
1 from example_dict1 import *
2
3 a[3] # 'c'
4 a.keys() # [1,2,3,...,26]
5 len(a) # 26
6 a[27]='A'
7 1 in a # True
8 del d[1]
9 1 in a # False
```

---

### 1.2.2 파이썬 문법 : loops, conditionals

**if-elif-else** 다른 모든 언어와 비슷하게, 파이썬에서도 if-else 문을 지원합니다. 아래와 같은 문법으로 사용됩니다.

---

```
1 if cond1:  
2     # when cond1 is True  
3 elif cond2:  
4     # when cond1 is False and cond2 is True  
5 else:  
6     # when cond1 is False and cond2 is False
```

---

**switch** 파이썬에서는 switch문을 지원하지 않습니다. 하지만, 아래와 같이 switch문을 대체하여 사용할 수는 있습니다.

**Implementation 1.9:** Switch using dict (*example\_switch1.py*)

---

```
1 def func(a):  
2     switch_options = {\  
3         '1' : '1st',  
4         '2' : '2nd',  
5         '3' : '3rd', }  
6     return switch_options[a]
```

---

**for loop** 파이썬에서의 for loop는 임의의 Sequential Type 변수에 대해서, 그 변수 안의 원소를 한번씩 돌게 됩니다. 예를 들어서 아래 코드를 살펴봅시다.

**Implementation 1.10:** For Loop Example (*example\_for1.py*)

---

```
1 for elem in ['a', 'b', 'c']:  
2     print(elem)
```

---

**while loop** 파이썬에서의 while문은 다른 언어에서의 while문과 크게 다르지 않습니다. 아래의 소스 코드를 살펴보면 알 수 있을 것입니다.

**Implementation 1.11:** While Loop Example (*example\_while1.py*)

---

```
1 i = 0  
2 while i<10:  
3     print(i)  
4     i += 1
```

---

### 1.2.3 파이썬 문법 : 함수, 클래스

#### 1.2.3.1 함수

파이썬에서 함수는 다음과 같이 정의합니다.

**Implementation 1.12:** Function Syntax (*example\_function1.py*)

---

```

1 def function(args):
2     return None

```

---

위 소스코드에서 각 항목은 아래와 같은 의미를 가집니다.

- def : 함수 정의 키워드입니다.
- function : 함수 이름을 나타냅니다.
- args : 함수 인자입니다. 아래와 같은 옵션이 있습니다.
  - arg
  - arg\_default : 함수 인자의 기본값을 정해줄 때, =을 이용하여 기본값을 지정해줄 수 있습니다.
  - \*arg\_list : 정해지지 않은 수의 인자를 받고자 할 때, \*을 하나 붙여서 들어온 인자를 배열로 받을 수 있습니다.
  - \*\*arg\_dict : 정해지지 않은 수의 이름이 명시된 인자를 받고자 할 때, \*를 두개 붙여서 들어온 인자들을 dict 형태로 받을 수 있습니다.
- return None : 함수의 결과값으로 return 뒤의 구문을 반환합니다.

아래의 코드<sup>7</sup>를 보면 조금 더 명백해집니다.

#### **Implementation 1.13: Function Argument Options (example\_function2.py)**

---

```

1 def f(a = 0, *args, **kwargs):
2     print("Received by f(a, *args, **kwargs)")
3     print("=> f(a=%s, args=%s, kwargs=%s)" % (a, args, kwargs))
4     print("Calling g(10, 11, 12, *args, d = 13, e = 14, **kwargs)")
5     g(10, 11, 12, *args, d = 13, e = 14, **kwargs)
6
7 def g(f, g = 0, *args, **kwargs):
8     print("Received by g(f, g = 0, *args, **kwargs)")
9     print("=> g(f=%s, g=%s, args=%s, kwargs=%s)" % (f, g, args, kwargs))
10
11 print("Calling f(1, 2, 3, 4, b = 5, c = 6)")
12 f(1, 2, 3, 4, b = 5, c = 6)

```

---

위 프로그램의 실행 결과는 다음과 같습니다.

#### **Implementation 1.14: Output for Function Argument Options**

---

```

1 Calling f(1, 2, 3, 4, b = 5, c = 6)
2 Received by f(a, *args, **kwargs)
3 => f(a=1, args=(2, 3, 4), kwargs={'c': 6, 'b': 5})
4 Calling g(10, 11, 12, *args, d = 13, e = 14, **kwargs)
5 Received by g(f, g = 0, *args, **kwargs)
6 => g(f=10, g=11, args=(12, 2, 3, 4), kwargs={'c': 6, 'b': 5, 'e': 14, 'd': 13})

```

---

람다 함수 람다 함수란 익명함수를 뜻합니다. 이는 람다함수가 변수명을 가질 수 없음을 의미하는 것 이 아닙니다. 예컨대, 아래의 코드에서의 func1, func2는 둘 다 같은 함수(주어진 수에 2를 더하는)이며, func1은 람다식으로 작성되었지만 엄연히 func1이라는 이름을 가지고 있습니다.

#### **Implementation 1.15: Lambda Function Example (example\_lambda1.py)**

---

<sup>7</sup>[stackoverflow](#) 질문 : Understanding kwargs in Python 참조

---

```

1 func1 = lambda x: x+2
2
3 def func2(x):
4     return x+2
5
6 func3 = lambda x,y,z : x+y+z
7 func4 = lambda *args : sum(args)

```

---

람다식의 문법은 위 소스 코드에서 볼 수 있듯이 다음과 같이 이루어집니다.

- `lambda` : 람다함수 키워드. 람다함수 뒤의 구문 중 콜론 전에 있는 구문은 함수의 인자를, 뒤는 반환하는 값을 나타낸다.
- `x,y,z(func3)/*args(func4)` : 람다함수의 인자. 쉼표로 구분되며, 상기된 `*args`등도 똑같이 사용 가능함을 `func4`에서 확인할 수 있다.
- `x+y+z(func3)/sum(args)(func4)` : 람다함수의 반환값.

익명함수가 가지는 이점 중 하나는, 우리가 정수나 문자열을 다루듯이 함수 또한 하나의 변수로 다루고 싶을 때 편리하다는 점입니다. 본 단락에서는 일반화된 정렬 문제에서 어떤 식으로 람다식이 사용가능한지 보여드리고자 합니다.

어떤 배열을 정렬하는 문제를 생각해 봅시다. 이 때, 어떤 배열의 원소들이 정수라면 정렬 결과에는 이의가 없을 것입니다. 예컨대, 아래의 코드의 마지막 라인에서 `AssertionError`가 나지 않는다면 충분할 것입니다.<sup>8</sup>

---

#### Implementation 1.16: Inspecting Function Calls (example\_lambda\_sort1.py)

---

```

1 def mysort(lst): # insertion sort
2     if len(lst) == 1:
3         return lst
4     else:
5         head, tail = lst[0], lst[1:]
6         tail = mysort(tail)
7         for idx, elem in enumerate(tail):
8             if head <= elem:
9                 return tail[:idx] + [head] + tail[idx:]
10            return tail + [head]
11
12 assert mysort([2,1,3]) == [1,2,3]

```

---

하지만 주어진 리스트가 비교하기 어려운 것들로 되어있는 경우 - 예를 들어서, 숫자 3개짜리 튜플로 되어있는 경우 -에는 어떤 식으로 배열할 수 있을까요? 이를 위해서는 우선 배열의 원소를 서로 비교하기 위한 기준이 필요할 것입니다. 위 코드의 경우 원소간의 비교 기준은 대소관계이며, 8번째 라인 (`head>=elem`)에 이것이 반영되었다고 볼 수 있습니다. 여기서는 이 기준을 세 숫자의 합으로 생각해 봅시다. 그렇다면, 새로운 기준(세 숫자의 합)을 아래와 같이 반영할 수 있을 것입니다.

---

#### Implementation 1.17: Inspecting Function Calls (example\_lambda\_sort2.py)

---

```

1 def mysort(lst): # insertion sort
2     if len(lst) == 1:
3         return lst
4     else:
5         head, tail = lst[0], lst[1:]
6         tail = mysort(tail)

```

---

<sup>8</sup>물론 실전에서는 더 많은 테스트를 하시는 것을 권장드립니다.

---

```

7     for idx, elem in enumerate(tail):
8         if sum(head) >= sum(elem):
9             return tail[:idx] + [head] + tail[idx:]
10 assert mysort([(1,2,3), (2,-4,2), (1,3,1)]) == [(1, 2, 3), (1, 3, 1), (2, -4, 2)]

```

---

이제 여기서 조금 더 나아가서, 다음과 같은 소스를 생각해 봅시다.

---

**Implementation 1.18: Inspecting Function Calls (example\_lambda\_sort3.py)**

---

```

1 def compare(l, r): # (1)
2     return sum(l) >= sum(r)
3
4 def mysort(lst, cmp): # (2)
5     if len(lst) == 1:
6         return lst
7     else:
8         head, tail = lst[0], lst[1:]
9         tail = mysort(tail)
10        for idx, elem in enumerate(tail):
11            if cmp(head, elem): # (3)
12                return tail[:idx] + [head] + tail[idx:]
13
14 assert mysort([(1,2,3), (2,-4,2), (1,3,1)],
15               cmp = compare) == [(1, 2, 3), (1, 3, 1), (2, -4, 2)] # (4)

```

---

여기서, 위 소스를 조금 더 간소화해서 애초에 compare 함수를 def를 쓰지 않고 람다식을 이용하여 저렇게 쓸 수 있습니다.

---

**Implementation 1.19: Inspecting Function Calls (example\_lambda\_sort4.py)**

---

```

1 def mysort(lst, cmp = lambda x,y: sum(x) >= sum(y)): # (2)
2     if len(lst) == 1:
3         return lst
4     else:
5         head, tail = lst[0], lst[1:]
6         tail = mysort(tail)
7         for idx, elem in enumerate(tail):
8             if cmp(head, elem):
9                 return tail[:idx] + [head] + tail[idx:]
10
11 assert mysort([(1,2,3), (2,-4,2), (1,3,1)],
12               cmp = compare) == [(1, 2, 3), (1, 3, 1), (2, -4, 2)] # (3)

```

---

### 1.2.3.2 클래스

본 단락에서는 파이썬에서 클래스를 어떻게 정의하는지를 살펴보고자 합니다.

**클래스란** 클래스는 흔히 과자들과 그 과자들로 찍어낸 과자로 비유됩니다. 클래스는 객체를 만들기 위한 코드 템플릿<sup>9</sup>을 말합니다. 인스턴스는 이 템플릿을 이용하여 만들어진 코드 덩어리로 생각할 수

---

<sup>9</sup><https://www.hackerearth.com/practice/python/object-oriented-programming/classes-and-objects-i/tutorial/>

있습니다. 예<sup>10</sup>를 들어서, 더하기만 가능한 간단한 계산기 코드를 만든다고 가정합시다. 다음과 같이 전역변수를 사용하면 어렵지 않게 구현할 수 있을 것입니다.

---

**Implementation 1.20:** 더하기만 가능한 계산기 (*example\_cal1.py*)

---

```
1 result = 0
2
3 def adder(num):
4     global result
5     result += num
6     return result
7
8 print(adder(3))
9 print(adder(4))
```

---

이 때, 각자 다른 계산기를 2개를 만들어서 사용하고자 하면, 다음과 같이 2개의 계산기를 만들어야 할 것입니다.

---

**Implementation 1.21:** 더하기만 가능한 계산기 2개 (*example\_cal2.py*)

---

```
1 result1 = 0
2 result2 = 0
3
4 def adder1(num):
5     global result1
6     result1 += num
7     return result1
8
9 def adder2(num):
10    global result2
11    result2 += num
12    return result2
13
14 print(adder1(3))
15 print(adder1(4))
16 print(adder2(3))
17 print(adder2(7))
```

---

이렇게 코드를 작성할 경우, 완벽하게 똑같은 코드를 두 번 작성해야 함을 알 수 있습니다. 따라서 이러한 중복을 피하기 위해서, 파이썬에서는 - 그리고 객체지향적 언어에서는 - 클래스를 제공합니다. 클래스는 다음과 같이 계산기 코드를 템플릿화하여, 재사용이 용이하게 만듭니다.

---

**Implementation 1.22:** 더하기 계산기 클래스 (*example\_cal3.py*)

---

```
1 class Calculator:
2     def __init__(self):
3         self.result = 0
4
5     def adder(self, num):
6         self.result += num
7         return self.result
8
```

---

<sup>10</sup>본 예시는 <https://wikidocs.net/28점프> 투 파이썬에서 참고하였습니다.

---

```

9  cal1 = Calculator()
10 cal2 = Calculator()
11
12 print(cal1.adder(3))
13 print(cal1.adder(4))
14 print(cal2.adder(3))
15 print(cal2.adder(7))

```

---

이 때, cal1, cal2는 클래스 Calculator의 인스턴스입니다.

클래스를 사용하는 이유는 개발의 속도와 유지보수의 편의성, 그리고 코드 디자인의 간결성 때문입니다. 잘 구성된 클래스의 경우, 구현하고자 하는 대상과 구현하는 프로그램 소스 코드간 대응이 직관적입니다. 예컨대 회계사용 프로그램을 작성할 경우, 고객/법인/재무재표 등의 클래스를 사용하게 될 것입니다. 이런 경우, 단순 코드 블럭으로 되어있는 것보다 더 직관적으로 프로그램 전체의 구조를 이해할 수 있으며, 이는 개발 속도와 정확도에 긍정적인 영향을 끼칠 것입니다. 프로그램의 유지보수 측면에서도 이와 비슷한 이유로 추후 유지보수가 간결해집니다.

이제 조금 더 자세하게 클래스(특히, 파이썬에서의 클래스에 대해서) 알아보겠습니다.

**파이썬에서의 클래스** 위 단락에서 클래스가 무엇인지, 그리고 왜 필요한지에 대해서 간단하게 살펴보았습니다. 본 단락에서는 클래스를 구성하는 요소를 살펴보고, 이를 정의하는 파이썬 문법에 대해서 알아보고자 합니다. 이해를 돋기 위해서 여기서는 문자열을 다루는 간단한 클래스를 만들어보고자 합니다. 먼저,

클래스는 속성(attribute)과 메소드(method)로 구성됩니다. 속성은 클래스 속성과 인스턴스 속성으로 분류되며, 메소드는 인스턴스 메소드, 클래스 메소드, 스태틱 메소드로 분류됩니다.<sup>11</sup>

각각의 예시에 대해서 아래 코드에서 살펴보겠습니다. 먼저, 가장 기본적인 부분만을 작성한 코드를 살펴보겠습니다.

---

**Implementation 1.23: Making a MyString Class (example\_class1.py)**

---

```

1 class MyString:
2     class_name = 'MyString'
3     maker = 'Schin'
4
5     datum = ''
6
7 a = MyString # a is a class
8 b = MyString() # b is an instance

```

---

위 코드에서 파이썬 클래스의 각 요소들을 설명하면 다음과 같습니다.

- class : 클래스를 정의하는 키워드입니다. 이 키워드 뒤의 단어가 클래스의 이름이 됩니다. 여기서는 MyString입니다.
  - maker = 'Schin' : 클래스의 속성을 지정합니다. 여기서는 class\_name, maker, datum의 3개의 속성이 있습니다. 편의상 datum을 우리가 다루고자 하는 문자열로 생각하겠습니다.
  - a = MyString : a라는 변수의 값으로 MyString이라는 클래스를 지정합니다.
  - b = MyString() : b라는 변수의 값으로 MyString이라는 인스턴스를 지정합니다.
- 여기서 클래스(a)와 인스턴스(b)의 차이를 살펴보기 위해서 다음 코드를 실행해 보겠습니다.

**Implementation 1.24: Making a MyString Class (example\_class1.py)**

---

<sup>11</sup>파이썬의 경우, private method를 지원하기는 하나 완벽하게 private하지는 않습니다. \_\_로 시작하는 속성이나 메소드는 private로 분류되지만, 이것이 완벽하게 클래스 밖에서 은닉되어 있지는 않습니다. 예컨대, 클래스 이름이 Mycls이고 속성이 \_\_attr1일 경우, Mycls\_\_attr1로 접근할 수 있습니다. 더 자세한 내용은 [참고 링크](#)를 참고하세요.

---

```
1 print(a.class_name) # prints 'MyString'
2 print(b.class_name) # prints 'MyString'
3 a.class_name = 'new MyString'
4 print(a.class_name) # prints 'new MyString'
5 print(b.class_name) # prints 'new MyString'
6 b.class_name = 'MyString again'
7 print(a.class_name) # prints 'new MyString'
8 print(b.class_name) # prints 'MyString again'
```

---

여기서 볼 수 있듯이, 클래스 자체의 변화는 인스턴스에 영향을 주지만 그 역은 성립하지 않습니다. 이는 모든 인스턴스는 클래스의 틀을 따르기 때문입니다. 위에서의 쿠키클의 예시를 들면, 쿠키 하나를 바꾼다고 쿠키클이 바뀌지는 않는 것과 같습니다. 반대로, 쿠키클을 바꾸면 모든 쿠키는 영향을 받게 됩니다.

본격적으로 문자열을 다루기 위해서 b에 우리가 다루고자 하는 문자열을 저장하고자 합니다. 이는 다음과 같은 방식으로 할 수 있습니다.

---

#### Implementation 1.25: MyClass Attribute (*example\_class1.py*)

---

```
1 b.datum = 'hello world!'
2 print(b.datum)
```

---

이제 이 후에는 어떻게 해야 할까요? 문자열을 저장하는 것만으로는 충분하지 않습니다. 이제 조금 더 복잡한 클래스 문법을 살펴보겠습니다.

---

#### Implementation 1.26: Making a MyString Class (*example\_class2.py*)

---

```
1 class MyString:
2     class_name = 'MyString'
3     maker = 'Schin'
4
5     def __init__(self, datum = ''):
6         self.datum = datum
7
8     # instance method
9     def get_first_letter(self):
10         return self.datum[0]
11
12     def most_frequent_word(self):
13         bow = MyString._create_bow(self.datum)
14         res = ''
15         tmp = 0
16         for k in bow.keys():
17             if bow[k] > tmp:
18                 res = k
19                 tmp = bow[k]
20         return res
21
22     # magic method
23     def __add__(self, other):
24         if isinstance(other, self.__class__):
25             return MyString(datum = self.datum + other.datum)
26         return NotImplemented
```

---

```

27
28     @classmethod
29     def assign_author(cls, author):
30         cls.author = author
31
32     @staticmethod
33     def _create_bow(input_str):

```

---

- 속성(Attribute) : 클래스나 인스턴스가 가지고 있는 정보를 말합니다. 예를 들어서, 문자열의 경우라면 문자열의 내용이나 저자 등이 있을 것입니다.
    - 클래스 속성 : 클래스 자체가 가지고 있는 속성을 말합니다. 예를 들어서, 문자열의 경우라면 클래스 이름 MyString 등이 있습니다.
    - 인스턴스 속성 : 인스턴스 각각이 가지고 있는 속성을 말합니다. 예를 들어서, 문자열의 경우라면 문자열의 내용 등이 있습니다.
  - 메소드(method) : 클래스나 인스턴스의 상태를 변화시키거나, 새로운 클래스를 만드는 등의 작업을 하는 함수입니다.
    - 인스턴스 메소드 : 인스턴스에 대해서 작동하는 함수입니다. 인스턴스를 변형시키거나 인스턴스를 이용하여 어떠한 작업을 수행합니다.
    - 클래스 메소드 : 클래스에 대해서 작동하는 함수입니다. 클래스 자체를 변형시킬 수 있습니다.
    - 스태틱 메소드 : 클래스나 인스턴스와 상관없는 함수입니다. 보통 클래스 내에서만 쓰이지만, 딱히 인스턴스나 클래스의 정보를 필요로 하지 않는 경우에 쓰입니다.
- 위 소스 코드를 통해서 우리는 다음과 같은 일들을 할 수 있습니다.

**Implementation 1.27:** MyClass Attribute (*example\_class2.py*)

---

```

1 a = MyString('hello')
2 b = MyString('world')
3
4 print(a.get_first_letter())
5
6 c = a+b # MyString('helloworld')
7 print(c.datum)
8
9 c.assign_author('Shin')
10 print(c.author)
11 print(a.author)
12
13 d = MyString('hello world it is so nice to meet you how are you doing world')
14 print(d.most_frequent_word())

```

---

**Magic Methods** Magic Method는 대표적인 syntactic sugar로써, 일반적으로 `__method_name__` 형태로 메소드를 지칭합니다.

### 1.3 파이썬 인터프리터의 이해

본 단락에서는 주어진 소스 코드를 파이썬이 계산하는 법을 알아볼 것<sup>12</sup>입니다. 본격적인 설명에 앞서, 변수의 종류에 대해서 설명하겠습니다. 일반적으로 변수에는 다음의 세 종류가 있습니다.

<sup>12</sup>[파이썬 공식 Documentation](#) 참조

- Bound variable : 어떤 값이나 다른 변수에 의해서 값이 결정되는지 정해진 변수
- Binding variable : Bound variable의 값을 결정하는 변수
- Free variable : Bound되지 않은 변수

이 때, 파이썬 인터프리터<sup>13</sup>는 **bound variable**을 **binding variable**로 대체(substitute) 하여 계산합니다. 만약 계산해야 하는 모든 변수들의 값이 결국 어떤 값(숫자, 문자열 등등)으로 환원되면 그 값을 계산하여 반환하고, 그렇지 않다면 예외를 반환합니다. 따라서 파이썬 인터프리터의 동작을 이해하는 것은 곧 어떤 식으로 변수들이 서로를 bind/bound 하는지를 이해하고, 이를 기반으로 **기계적으로** 변수를 적절한 값으로 대체하여 계산을 수행함을 의미합니다.

Binding이 일어나는 경우는 아래와 같습니다.

**import문의 사용** import문을 사용할 경우, import된 모듈에서의 모든 namespace가 bind됩니다.

**for loop** for loop에서 헤더는 루프 코드블럭 안에서 for loop 헤더에서 선언된 변수를 bind 합니다.

**함수, 클래스의 정의** 함수나 클래스의 정의는 함수나 클래스 이름을 bind하게 됩니다. 예를 들어서 아래 소스코드를 보면, compare이라는 변수는 1번 라인에 의해서 2번 라인에 binding되어, 14번 라인을 거쳐 11번 라인에서 쓰이게 됩니다.

---

**Implementation 1.28: Binding in Function definition (example\_lambda\_sort3.py)**

---

```

1 def compare(l, r): # (1)
2     return sum(l) >= sum(r)
3
4 def mysort(lst, cmp): # (2)
5     if len(lst) == 1:
6         return lst
7     else:
8         head, tail = lst[0], lst[1:]
9         tail = mysort(tail)
10        for idx, elem in enumerate(tail):
11            if cmp(head, elem): # (3)
12                return tail[:idx] + [head] + tail[idx:]
13
14 assert mysort([(1,2,3), (2,-4,2), (1,3,1)],
15               cmp = compare) == [(1, 2, 3), (1, 3, 1), (2, -4, 2)] # (4)

```

---

**=의 사용** 예를 들어서, 아래의 소스코드에서는 각각 a,b가 bound variable, a가 binding variable, c가 free variable입니다.

---

**Implementation 1.29: Types of Variables (variables.py)**

---

```

1 a = 1
2 b = a
3 c

```

---

<sup>13</sup>사실 많은 인터프리터가 대부분 이렇게 동작합니다.

함수 인자의 binding 함수 인자 역시 함수가 정의된 곳 내에서의 binding을 야기합니다. 예를 들어서 아래 소스코드를 살펴봅시다.

---

**Implementation 1.30: Inspecting Function Calls (example\_interpreter1.py)**

---

```
1 def func1(input_num):
2     a = int(input_num[0])
3     b = int(input_num[1])
4     c = int(input_num[2])
5
6
7     return func2(a,b,c,)
8
9 def func2(a,b,c):
10    return 100*c + 10*b + a
11
12 print(func1('123'))
```

---

이 때 출력될 값은 321일 것입니다. 이를 이해하기 위해서 파이썬 인터프리터 안에서 어떤 일이 벌어지는지 한 단계씩 살펴보도록 하겠습니다.

1. line 12 : func1('123')을 호출, 반환된 값을 출력함
2. line 1 : func1 정의된 부분으로 감
3. line 2-7 : 이 부분의 식을 계산하되, input\_num을 '123'으로 대체하여 계산함 (=binding이 일어남: input\_num 이 '123'에 bind됨)
  - (a) line 2-4 : int('123'[0]) == 1 과 같은 계산을 반복
  - (b) line 7 : func2(1,2,3)을 호출, 반환된 값을 반환함
4. line 9 : func2 정의된 부분으로 감
5. line 10 : 100c + 10b + a 계산하되, a,b,c를 각각 1,2,3으로 대체하여 계산함 (=binding이 일어남: a,b,c가 각각 1,2,3에 bind됨)

## Chapter 2

# Computation과 데이터구조의 이해

### 2.1 데이터구조 vs Abstract Data Type

컴퓨터를 이용하여 어떤 계산을 할 때, 우리는 정해진 operation들을 이용하여 원하는 결과를 얻습니다. 예를 들어서, 우리가 어떤 숫자의 배열을 정렬하고 싶다면 숫자간의 비교를 하는 것이 우리가 사용할 operation이 될 것입니다. 이러한 operation들은 상황에 따라서 필요한 operation들을 정의하여 사용하게 됩니다. 이 때 정의된 operation들의 집합을 **Abstract Data Type(ADT)**이라고 합니다. 그리고 이것을 구현하기 위한 구조체를 **데이터구조**라고 합니다. ADT와 데이터구조는 헷갈리기 매우 좋은데, 이는 이름이 같은 경우가 많기 때문입니다. 예를 들어서, 트리는 맵에 따라서 ADT일 수도, 데이터구조일 수도 있습니다. 이 둘을 구분하는 방법은 구현에 대한 언급이 있는지를 살펴보는 것입니다.

ADT와 데이터구조의 관계는 자바에서 interface와 클래스의 관계나, C++에서 추상 클래스와 클래스의 관계와 매우 비슷합니다. 이 둘을 구분하는 것은 구현과 구조의 분리라는 객체지향의 원리를 지킨다는 측면에서 중요합니다. 또한, 다른 데이터구조를 사용할 때 같은 ADT더라도 성능이 달라질 수 있으므로 상황에 따라서 적절한 구현을 고르기 위해서는 ADT와 데이터구조를 분리하여 생각하는 것이 좋습니다.

구체적인 예를 들기 위해서 list ADT를 정의하고, 이를 두 가지의 다른 데이터구조(array, linked list)로 구현했을 때의 차이점을 살펴보겠습니다.

**List ADT Specification** 개념적으로 list는 head, tail로 정의되며, head는 임의의 데이터, tail은 리스트입니다. 이 때 tail이 아무런 element도 없는 리스트일 경우 Nil<sup>1</sup>이라고 합니다.

List ADT는 다음과 같은 operation들을 말합니다<sup>2</sup>.

- `__init__` : 빈 리스트를 만드는 constructor
- `is_empty` : 리스트가 비어 있으면 True, 아니면 False를 리턴.
- `prepend / append` : 리스트의 앞/뒤에 어떤 element를 넣는 함수.
- `head` : 리스트의 첫 번째 원소를 리턴.
- `tail` : 리스트의 첫 번째 원소를 제외한 나머지 원소들을 리턴.

이를 파이썬 코드로 나타내면 다음과 같습니다.

---

<sup>1</sup>언어에 따라 다르지만, None이라고도 하고 Null이라고도 합니다. 파이썬의 경우 None을 사용합니다.

<sup>2</sup>이는 필수적으로 있어야 하는 것들만 포함하는 것이고, 대개의 경우 이것 외에도 많은 것들을 포함시킬 수 있습니다.

**Implementation 2.1: List ADT (List.py)**


---

```

1  from abc import *
2
3  class List(metaclass = ABCMeta):
4      @abstractmethod
5      def __init__(self):
6          pass
7
8      @abstractmethod
9      def is_empty(self):
10         pass
11
12     @abstractmethod
13     def prepend(self, item):
14         pass
15
16     @abstractmethod
17     def append(self, item):
18         pass
19
20     @abstractmethod
21     def head(self):
22         pass
23
24     @abstractmethod
25     def tail(self):
26         pass
27
28     @abstractmethod
29     def iter(self, option):
30         pass

```

---

여기서 1번째, 3번째 라인은 파이썬에서 추상 클래스를 만들기 위한 장치이며, `@abstractmethod` 는 이 메소드가 추상 메소드임을 의미합니다. `List`를 상속받는 클래스는 `@abstractmethod` 데코레이터가 있는 메소드들을 구현하여야 합니다. 여기서는 파이썬 언어 강의가 아니므로, 그냥 자바에서의 인터페이스로 받아들이면 충분합니다.

**Implementation Using Array Data Structure** `Array`는 인접한 메모리에 데이터를 저장하는 데이터구조를 의미합니다. 데이터가 인접한 메모리에 저장되어 있으므로, 임의의 `element`에 접근하는 것이 상수 시간에 이루어집니다. <sup>3</sup> <sup>4</sup> 이 때, 각 operation은 다음과 같이 구현될 수 있습니다.

- `__init__` : 빈 리스트를 만드는 constructor  
빈 배열을 반환합니다.
- `is_empty` : 리스트가 비어 있으면 `True`, 아니면 `False`를 리턴.
- `prepend / append` : 리스트의 앞/뒤에 어떤 `element`를 넣는 함수.

---

<sup>3</sup> 예를 들어서, 크기가 10인 데이터를 담고 있는 `array`의 시작 주소를 알고 있다면, `i`번째 데이터는 시작점에  $10*i$ 만큼을 더함으로써 주소값을 바로 얻을 수 있으며, 따라서 `array` 내 임의의 원소에 상수 시간에 접근이 가능합니다.

<sup>4</sup> 파이썬에서는 C처럼 메모리를 명시적으로 다루기 어렵기 때문에, 여기서는 `array`를 이용한 구현은 생략합니다.

append의 경우, array에 빈 공간이 있고 array의 원소 갯수를 알고 있으면 넣어야 할 주소값에 element를 넣으면 되며, 만약 빈 공간이 없으면 array의 크기를 늘린 후 넣으면 됩니다. 이런 경우, amortized cost<sup>5</sup>는 상수시간으로 볼 수 있습니다.

하지만 prepend의 경우는 array의 내용들을 memcpy를 이용하여 복사하여 element의 크기만큼 shift 해야 하므로 array에 있는 내용에 비례하는 시간이 걸리게 됩니다.

- head : 리스트의 첫 번째 원소를 리턴.  
Array의 시작부터 element의 크기만큼의 비트를 리턴해주면 됩니다.
- tail : 리스트의 첫 번째 원소를 제외한 나머지 원소들을 리턴.  
Array에서 head를 제외하고 나머지를 리턴해주면 됩니다.

## Linked List

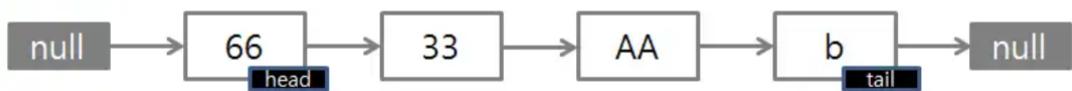


Figure 2.1: Linked List의 구조

**Implementation Using Linked List** Linked list는 위와 같이, 데이터와 그 다음 데이터로의 주소값으로 이루어진 데이터구조입니다. 맨 처음과 끝에 Null값을 붙일 수도, 안 붙일 수도 있습니다. Linked list는 list의 개념적인 정의에 매우 충실했던 구조인데, 이는 head 뒤에 다음 element로의 포인터를 tail로 보면 tail 또한 linked list이기 때문입니다.

Linked List에서는 각 operation을 다음과 같이 구현할 수 있습니다.

- `__init__` : 빈 리스트를 만드는 constructor

Implementation 2.2: LinkedList Constructor (LinkedList.py)

---

```

1 def __init__(self):
2     self.head = None
3     self.tail = None
  
```

---

여기서는 빈 list를 head, tail 모두 None인 리스트로 생각합니다.

- `is_empty` : 리스트가 비어 있으면 True, 아니면 False를 리턴.

Implementation 2.3: LinkedList is\_empty (LinkedList.py)

---

```

1 def is_empty(self):
2     return self.head is None and self.tail is None
  
```

---

위 empty list의 정의에 따라, head와 tail이 None인지 체크합니다.

- `prepend / append` : 리스트의 앞/뒤에 어떤 element를 넣는 함수.  
`prepend`는 리스트의 앞에 element를 추가하는 것으로, 다음의 과정을 통해서 이루어집니다.

---

<sup>5</sup>간단하게 말하면, array의 크기를 늘리는 malloc 함수는 비싼 함수지만 이를 부를 일이 많지 않으므로, 이를 고려하여 계산하는 것.

**Implementation 2.4: LinkedList prepend (LinkedList.py)**

```

1  def prepend(self, item):
2      if self.is_empty():
3          self.head = item
4      else:
5          self.set_tail(deepcopy(self))
6          self.set_head(item)

```

먼저, 새 LinkedList를 만든 후, head에 item을 첨가합니다. 이후 새로 만든 LinkedList의 tail을 기준 LinkedList로 만든 후, 새로 만든 LinkedList를 반환합니다. 이 과정에서 tail은 언제나 None 아니면 LinkedList일 것입니다. 또한, 이 operation은 지금 리스트 안에 들어 있는 element의 갯수와 상관 없이 상수 시간이 걸립니다.

이와는 달리, append는 다음의 과정을 통해서 이루어집니다.

**Implementation 2.5: LinkedList append (LinkedList.py)**

```

1  def append(self, item):
2      if self.is_empty():
3          self.head = item
4      else:
5          tmp = LinkedList()
6          tmp.head = item
7          cur = self
8          while cur.tail is not None:
9              cur = cur.tail
10             cur.tail = tmp

```

만약 리스트가 비어 있는 경우 prepend와 똑같이 시행합니다. 그렇지 않은 경우, 빈 LinkedList를 만들어 head에 item을 넣습니다. 그 후, 원 LinkedList의 tail에서 None을 발견할 때까지 LinkedList를 traverse한 후, tail이 None인 LinkedList를 발견하면 tail을 새로 만든 LinkedList로 대체합니다. 이 과정에서는 LinkedList를 한 번 끝까지 traverse 해야 하므로, 리스트 내의 element 갯수에 선형적으로 비례하는 시간이 걸립니다.<sup>6</sup>

- head : 리스트의 첫 번째 원소를 리턴.

**Implementation 2.6: List ADT (LinkedList.py)**

```

1  def head(self):
2      return self.head

```

- tail : 리스트의 첫 번째 원소를 제외한 나머지 원소들을 리턴.

**Implementation 2.7: List ADT (LinkedList.py)**

```

1  def tail(self):
2      return self.tail

```

**Comparison** 각 operation의 복잡도는 다음과 같이 정리할 수 있습니다.

<sup>6</sup> 이를 방지하기 위해서, tail 뿐만 아니라 앞쪽 방향의 list도 미리 저장하는 doubly linked list를 사용하기도 합니다.

	ArrayList	LinkedList
<code>__init__</code>	$O(1)$	$O(1)$
<code>is_empty</code>	$O(1)$	$O(1)$
<code>prepend</code>	$O(n)$	$O(1)$
<code>append</code>	$O(1)$	$O(n)$ ( $O(1)$ if doubly linked list)
<code>head/tail</code>	$O(1)$	$O(1)$

만약 Binary Tree나 B+ 트리 같은 데이터구조를 사용한다면 복잡도는 또 달라지게 됩니다. 이에 대해서는 뒤 단원에서 더 자세하게 다뤄 보도록 하겠습니다.

## 2.2 Concept of Algorithm

위와 같이 ADT에서 operation들을 정의하였으면, 이를 레고 블럭을 조립하듯이 적절히 조합하여 원하는 계산을 행하는 것을 알고리즘이라고 합니다. 따라서, 알고리즘이 적절히 정의되기 위해서는 사용할 ADT를 먼저 정의해야 합니다. 그 후, ADT를 어떤 데이터구조를 이용하여 구현할지를 결정해야 합니다. 데이터구조를 고를 때는 만들어진 알고리즘이 ADT의 어떤 operation을 많이 사용하는지, 그리고 어떤 데이터구조가 많이 쓰이는 operation을 효율적으로 구현하는지를 고려하여 결정을 내리게 됩니다. 여기서는 Dijkstra's algorithm을 예시로 위 과정을 따라가보자 합니다.

Dijkstra의 알고리즘은 그래프 위에서 최단경로를 찾기 위해서 만들어진 알고리즘입니다. 이 알고리즘은 priority queue라는 ADT를 사용하는데, 이 priority queue의 구현 방법에 따라서 operation의 복잡도가 달라집니다. 본 단락에서 언급되는 ADT나 데이터구조의 구현에 대해서는 추후 단원에서 다를 것입니다.

**Priority Queue** Priority queue는 큐 중에서 특수한 형태의 큐로, element들이 우선순위와 같이 큐 속에 저장되어 있습니다. Dequeue시에는 가장 낮은<sup>7</sup> priority를 가진 원소가 큐에서 제거됩니다. Priority queue는 다음의 4개의 operation으로 구성되어 있습니다.

- `__init__` : 빈 priority queue를 만듭니다.
- `push` : 한 원소와 priority를 queue에 추가합니다.
- `decrease_key` : 한 원소의 priority를 낮춥니다.
- `pop` : 가장 priority가 낮은 원소를 큐에서 제거하고, 그 원소를 반환합니다.

**Implementation of Priority Queue** 일반적으로 priority queue를 구현하는 방법은 크게 3가지가 있습니다. Linked list, binaray heap, fibonacci heap입니다. 각 데이터구조당 operation의 시간복잡도는 아래와 같습니다.

	push	decrease_key	pop
linked list	$O(1)$	$O(n)$	$O(n)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log n)$

<sup>7</sup>가장 높아도 상관없으며, priority를 기반으로 dequeue되는 원소가 정해지는 것이 중요합니다.

**Dijkstra's Algorithm** Dijkstra의 알고리즘에서는 각각 V<sup>8</sup>번의 push, pop, 그리고 E<sup>9</sup>번의 decrease\_key가 수행됩니다. 따라서, 복잡도는 다음과 같습니다.

	Complexity
Linked List	$O(V^2)$
Binary Heap	$O((E+V) \log V)$
Fibonacci Heap	$O(E + V \log V)$

즉, 같은 알고리즘과 같은 ADT더라도 어떠한 데이터구조를 이용하는지에 따라 복잡도가 달라집니다. 따라서 적절한 데이터구조를 선택해야 할 것입니다. 예컨대, push가 많고 상대적으로 pop이 매우 적은 경우에는 binary heap보다 linked list가 더 효과적일 수도 있습니다.

---

<sup>8</sup>그래프에서 vertex의 갯수

<sup>9</sup>그래프에서 edge의 갯수

# Chapter 3

## List of ADTs

본 단원에서는 다양한 ADT의 개념과 정의를 살펴보고 이를 이용한 문제를 제시하고, 그 문제를 푸는 알고리즘을 알아볼 것입니다. 본 수업에서는 이 ADT를 다양한 데이터구조를 이용하여 구현하고 이를 이용하여 제시된 문제를 푸는 알고리즘을 구현해볼 것입니다.

### 3.1 List

리스트는 head, tail의 순서쌍이며, head는 저장하는 데이터, tail은 리스트입니다. 이 때 tail이 빈 리스트인 경우는 None이라고 표기합니다. 리스트 구조 자체가 재귀적이기 때문에, 리스트에서 시행할 operation들 중에는 이를 적극적으로 활용하는 경우가 많습니다. 예를 들어서, 리스트에서 가장 큰 element를 찾는 함수를 짠다면, 다음과 같이 생각할 수 있습니다.

Implementation 3.1: *find\_max (ListAlgorithms.py)*

---

```
1 def find_max(lst):
2     if lst.tail is None:
3         return lst.head
4     else:
5         l = lst.head
6         r = find_max(lst.tail)
7         return max(l, r)
```

---

즉, 리스트의 모든 원소에 대해서 어떤 함수를 시행하고 싶으면, head에서 그 함수를 시행한 후 재귀적으로 tail에서 그 함수를 시행하면 됩니다. 이러한 재귀 구조의 장점은 트리에서도 유감없이 사용할 수 있습니다.

#### 3.1.1 ADT List Specification

위에서는 예시로 필요최소한의 list operation만을 살펴보았지만, 여기서는 조금 더 다양한 operation들을 구현하도록 하겠습니다.

- `__init__` : List Constructor
- `__iter__` : 리스트의 원소를 한번씩 traverse하는 iterator<sup>1</sup>

---

<sup>1</sup>Iterator는 단순 루프와는 다소 차이가 있습니다. 파이썬에서는 yield를 사용해야 하며, 위에서 간단하게 다른 바 있습니다.

- `is_empty` : 리스트가 비어 있으면 True, 아니면 False를 리턴
- `prepend / append` : 리스트의 앞/뒤에 어떤 element를 넣는 함수. 기존 리스트를 변화시킴.
- `add` : 리스트의 주어진 위치에 element를 추가하는 함수. 기존 리스트를 변화시킴.
- `remove` : 리스트의 원소 중 하나를 삭제. 기존 리스트를 변화시킴.
- `change` : 리스트의 원소 중 하나를 바꿈. 기존 리스트를 변화시킴.
- `head` : 리스트의 첫 번째 원소를 리턴.
- `tail` : 리스트의 첫 번째 원소를 제외한 나머지 원소들을 리턴.
- `sort` : 리스트를 주어진 기준에 따라 소팅하여 새 리스트를 반환. 기존 리스트는 변하지 않음.
- `max/min` : 리스트의 원소 중 주어진 기준으로 볼 때 가장 큰/작은 원소 반환. 기존 리스트는 변하지 않음.

## 3.2 Queue

큐는 일반적으로 선입선출(First In, First Out; FIFO)를 만족하는 데이터구조를 말합니다. 소위 말하는 큐 사인에서 쓰는 큐가 이것이며, 선착순으로 어떤 일을 처리해야 할 때 많이 사용됩니다. 가장 기본적인 데이터구조로 다른 데이터구조를 만들 때에도 많이 쓰입니다.

큐의 일반적인 변형으로 priority queue가 있습니다. 일반적인 큐는 들어오는 순서가 우선순위이지만, 우선순위 큐는 원소가 들어올 때 priority를 같이 가지고 들어옵니다. 또, pop 시에는 지금 큐에 있는 원소들 중에 가장 priority가 큰 원소가 먼저 pop됩니다. 우선순위 큐는 가깝게는 dijkstra의 알고리즘을 구현하는 것에서부터 멀리는 게임에서 적절한 mmr을 가지는 사람들끼리 큐를 잡아 줄 때나, os에서 프로세스에게 cpu를 분배할 때 등 다양한 곳에서 사용될 수 있습니다.

### 3.2.1 ADT Queue Specification

- `__init__` : Queue Constructor
- `__iter__` : 큐 안의 원소를 한번씩 traverse하는 iterator
- `is_empty` : 큐가 비어 있으면 True, 아니면 False를 리턴
- `push` : 큐의 맨 뒤에 원소 삽입.
- `pop` : 큐 맨 앞의 원소 반환하고 삭제. 기존 큐를 바꿈.

### 3.2.2 ADT Priority Queue Specification

- `__init__` : Priority Queue Constructor
- `__iter__` : 큐 안의 원소를 한번씩 traverse하는 iterator
- `is_empty` : 큐가 비어 있으면 True, 아니면 False를 리턴
- `push` : 큐의 맨 뒤에 원소 삽입.
- `pop` : 큐 맨 앞의 원소 반환하고 삭제. 기존 큐를 바꿈.
- `decrease_key` : 특정 원소의 priority를 줄임. 기존 큐를 바꿈.

### 3.2.3 Problem Solving Using Queue

**MMR matching**

## 3.3 Tree

트리는 리스트와 매우 비슷합니다. 트리와 리스트가 다른 것은 보통 트리에서는 head를 루트라고 부른다는 것과, tail이 여러 개 있다는 점입니다. 트리는 일반적으로 대부분의 분류 구조를 나타낼 때 폭넓게 쓰일 수 있으며, DB 등의 데이터 저장이나 분류 등에서도 많이 사용됩니다.

### 3.3.1 ADT Specification of Tree

- `__init__` : Tree Constructor
- `iter` : 트리 안의 모든 element를 traverse하는 iterator. 위 리스트나 큐에서의 `__iter__`과는 다르게, 추가 argument인 option에 따라서 iterate하는 순서를 바꿀 수 있다.
- `__iter__` : 편의를 위한 default iterator
- `root` : 트리의 루트 노드를 반환.
- `children` : 트리의 child들을 담고 있는 리스트. child들 각각 역시 트리이다.
- `leaves` : 트리의 노드들 중 `children`이 빈 리스트인 노드들을 traverse하는 iterator.
- `find_subtree` : 트리 중 주어진 element를 루트로 가지는 subtree들의 리스트를 출력.

## 3.4 Graph

그래프는 노드들이 임의의 형태로 연결된 것을 말합니다. 이 때 연결 방식에 따라서 그래프의 종류가 나뉩니다.

- directed / undirected
- weighted
- cyclic / acyclic

### 3.4.1 ADT Specification of Graph

- `__init__` : Graph Constructor
- `__iter__` : Graph iterator.
- `iter` : 그래프에 있는 모든 node를 traverse하는 iterator. option에 따라서 traverse하는 순서가 달라짐.
- `is_adjacent` : 주어진 두 노드가 인접하면 True, 아니면 False를 반환.
- `adjacent` : 주어진 노드와 인접한 모든 노드를 담고 있는 리스트를 반환.
- `add_vertex, remove_vertex` : 주어진 노드를 그래프의 vertex에 추가.
- `add_edge, remove_edge` : 주어진 노드의 순서쌍을 그래프의 edge에 추가.