

# Coding the Mathematics 강의 교재

신승우

KAIST, School of Computing

# **Contents**

# 서문

## 강의 소개

본 강의는 프로그래머를 위한 수학 강의용 교재입니다. 학습자가 프로그래머이므로, 수학을 프로그래밍을 이용하여 배울 것입니다. 더 자세하게는, 수학에의 다양한 개념 - 집합, 함수, 식 등 - 을 파이썬으로 구현하는 것을 목표로 합니다.

본 강의는 다음에 대한 내용들을 다룹니다.

- 수학
  - 집합의 정의와 그 연산
  - 함수의 정의와 표현, 식의 계산
  - 행렬과 벡터의 정의, 연산 및 응용
  - 미분, 적분의 정의와 그 응용
  - 확률 및 통계에 대한 소개
- 프로그래밍
  - 객체지향적 프로그램 연습
  - generator, lambda expression 등 다양한 프로그래밍 기법 연습
  - 간단한 파싱과 인터프리터 작성 연습

## 교재의 구성

교재에서는 새로 나온 개념을 우선 수학적인 언어로 설명하고, 그에 해당하는 구현을 같이 소개합니다. 그리고 덧붙여 생소할 수 있는 프로그래밍 개념도 설명합니다.

## 사전지식

이 강의를 듣기 위해서는 중학교 수준의 수학에 대한 지식과 더불어, 재귀나 객체지향에 대한 기본적인 이해가 필요합니다. 만약 대학교 공업수학 정도의 수학에 익숙하면서 동시에 프로그래밍에 대한 지식이 있다면 이 강의를 들을 필요가 없습니다.

## 강의 자료

본 교재는 패스트캠퍼스의 Coding the Mathematics 강의<sup>1</sup>를 위해서 제작되었습니다. 본 강의에서 구현할 코드들과 그 스켈레톤은 깃헙<sup>2</sup>에서 찾아볼 수 있습니다. 제공된 소스 코드는 파이썬 3.5로 윈도우 환경에서 테스트되었습니다.

---

<sup>1</sup>강의 웹페이지

<sup>2</sup>강의자료 깃헙 repo

# Chapter 1

## 실습환경 설정 및 프로그래밍 복습

본 단원에서는 실습환경 설정과 파이썬 문법 소개, 그리고 기초적인 프로그래밍의 개념들을 복습하는데 도움을 주기 위하여 작성되었습니다. 본 단원은 처음으로 프로그래밍을 배우는 사람을 위한 것이 아니라 복습을 위한 것이므로, 처음 프로그래밍을 배우는 분은 다른 교재를 보기로 권장드립니다.

본 단원에서 예시로 작성된 코드는 `src/examples/` 폴더에 들어 있습니다.

### 1.1 실습환경 설정

#### 1.1.1 파이썬

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. 확인 법은 [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.x`를 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.pkg`을 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.

---

<sup>1</sup>[장고걸스 튜토리얼 : 파이썬 설치](#)

- Spyder
- Notepad++
- Atom
- 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 툴이 없습니다. 여기서는 바로 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 파이썬 소개

---

### 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

#### 1.2.1.1 Built-in Data Types

파이썬은 기본적으로 다음의 빌트인 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.

---

<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.

**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
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 헤수로
               인식합니다.
6 print(type(c)) # <type 'complex'>

```

---

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

```

---

<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.

```

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

```

---

Sequence 데이터형들은 다음의 연산들을 지원합니다.

- 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 elem이 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     }
7     return switch_options[a]
```

---

**for loop** 파이썬에서의 for loop는 임의의 Sequential Type 변수에 대해서, 그 변수 안의 원소를 한번씩 둘게 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})
```

---

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

**람다 함수** 람다 함수란 익명함수를 뜻합니다. 이는 람다함수가 변수명을 가질 수 없음을 의미하는 것 이 아닙니다. 예컨대, 아래의 코드에서 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)**

---

```

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 사용 가능한지 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

```

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

```

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 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.

**Implementation 1.20: Defining a Class (example\_class1.py)**

---

```
1 class MyClass(object):
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5         self.z = x+y
```

---

**Magic Methods** Magic Method<sup>8</sup>

**Implementation 1.21: Inspecting Function Calls (example\_class2.py)**

---

```
1 class BiCycle:
2     def __init__(self, brand_name, model_name,
3                  wheel, frame, seat, owner, ):
4         self.brand_name = brand_name
5         self.model_name = model_name
6         self.wheel = wheel
7         self.frame = frame
8         self.seat = seat
9         self.owner = owner
10
11    def __str__(self):
12        return brand_name + ', ' + model_name + ' owned by ' + owner
13
14    def __eq__(self, other):
15        return self.model_name == other.model_name
16
17    def __add__(self, other):
18        return Bicycle(self.brand_name + other.brand_name,
19                      self.model_name + other.model_name,
20                      self.wheel + other.wheel,
21                      self.frame + other.frame,
22                      self.seat + other.seat,
23                      self.owner + other.owner,)
```

---

**클래스 상속** 클래스 상속<sup>9</sup>

**Implementation 1.22: Inspecting Function Calls (example\_class3.py)**

---

```
1 class Person:
2
3     def __init__(self, first, last):
4         self.firstname = first
5         self.lastname = last
6
7     def Name(self):
8         return self.firstname + " " + self.lastname
9
10    class Employee(Person):
```

---

<sup>9</sup> 링크 참조

```

11
12     def __init__(self, first, last, staffnum):
13         Person.__init__(self, first, last)
14         self.staffnumber = staffnum
15
16     def GetEmployee(self):
17         return self.Name() + ", " + self.staffnumber
18
19 x = Person("Marge", "Simpson")
20 y = Employee("Homer", "Simpson", "1007")
21
22 print(x.Name())
23 print(y.GetEmployee())

```

---

#### 1.2.4 파이썬 인터프리터의 이해

본 단락에서는 주어진 소스 코드를 파이썬이 계산하는 법을 알아볼 것<sup>10</sup>입니다. 본격적인 설명에 앞서, 변수의 종류에 대해서 설명하겠습니다. 일반적으로 변수에는 다음의 세 종류가 있습니다.

- 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>11</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 에러를 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 1.23: 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)

```

<sup>10</sup>파이썬 공식 Documentation 참조

<sup>11</sup>사실 많은 인터프리터가 대부분 이렇게 동작합니다.

```

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 1.24: Types of Variables (variables.py)**

---

```

1 a = 1
2 b = a
3 c

```

---

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

**Implementation 1.25: 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 1,2,3이 bind됨)

## 1.3 프로그래밍 복습

### 1.3.1 시간복잡도

Big-O 표기법

계산 예시

### 1.3.2 데이터구조와 추상 데이터 타입(Abstract Data Type, ADT)

데이터구조 vs ADT

#### 1.3.2.1 추상 데이터 타입

스택 ADT

큐 ADT

트리 ADT

그래프 ADT

#### 1.3.2.2 데이터구조

링크드리스트 데이터구조

이진트리 데이터구조

### 1.3.3 알고리즘

#### 1.3.3.1 재귀

예시 : 하노이의 탑

1 pass

Implementation 1.26: Inspecting Function Calls (*example.py*)

---

#### 1.3.3.2 동적 프로그래밍

예시 : Longest Common Subsequence(LCS)

1 pass

Implementation 1.27: Inspecting Function Calls (*example.py*)

---

# Chapter 2

## 집합과 함수

본 단원에서는 집합과 그 표현, 그리고 집합의 구현에 대해서 다룹니다.

### 2.1 집합

#### 2.1.1 집합의 정의

**Definition 1 (집합)** 특정 조건에 맞는 원소들의 모임. 임의의 한 원소가 그 모임에 속하는지를 알 수 있고, 그 모임에 속하는 임의의 두 원소가 다른가 같은가를 구별할 수 있는 명확한 표준이 있는 것을 이른다.<sup>1 2</sup>

---

Implementation 2.1: Set 구현 (*PySet.py*)

---

```
3 class PySet:  
4     def __init__(self,  
5                  membership, # membership checking 아리랑
```

---

---

<sup>1</sup>국립국어원의 정의 2번 참조.

<sup>2</sup>사실 이 정의만 국립국어원이 정의를 따르는데, 이는 집합 자체가 정의하기 매우 어렵기 때문입니다. 여기서는 간단하게 위 정의를 따르고 넘어갑니다.

2.1.2 집합의 표기

2.1.3 집합의 연산

2.1.4 집합의 종류

2.1.5 집합의 예시 : 수 체계

2.2 좌표계

2.3 함수

2.3.1 함수의 정의

2.3.2 함수의 종류

2.3.3 합성함수와 역함수

2.3.4 다양한 함수들

# Chapter 3

## 식과 방정식

### 3.1 식의 계산

- 3.1.1 미지수와 식
- 3.1.2 곱셈공식과 인수분해
- 3.1.3 방정식의 풀이

## Chapter 4

# 함수의 정의와 다양한 함수의 성질

## Chapter 5

### 벡터의 정의와 응용

## Chapter 6

### 행렬의 정의와 계산

## Chapter 7

### 함수의 극한과 미분

## Chapter 8

### 적분의 정의와 적분법

## Chapter 9

# 다변수 미적분학과 그래디언트

## Chapter 10

### 확률과 조건부확률

## Chapter 11

### 통계적 추론

# Appendix