

Coding the Mathematics 강의 교재

신승우

KAIST, School of Computing

Contents

1	실습환경 설정 및 프로그래밍 복습	4
1.1	실습환경 설정	4
1.1.1	파이썬	4
1.1.2	IDE	4
1.1.3	Git	5
1.2	파이썬 소개	5
1.2.1	파이썬 데이터 타입	5
1.2.2	파이썬 문법 : loops, conditionals	8
1.2.3	파이썬 문법 : 함수, 클래스	9
1.2.4	파이썬 인터프리터의 이해	15
2	집합과 함수	18
2.1	집합	18
2.1.1	집합의 정의	18
2.1.2	집합의 연산	19
2.1.3	집합의 종류	19
2.1.4	집합의 예시 : 수 채계	20
2.2	좌표계	20
2.3	함수	20
2.3.1	함수의 정의	20
2.3.2	함수의 종류	20
2.3.3	합성함수와 역함수	20
2.3.4	다양한 함수들	20
3	식과 방정식	21
3.1	파싱	21
3.1.1	What is Parsing?	21
3.1.2	Grammar	21
3.1.3	Abstract Syntax Tree	21
3.1.4	Tokenizer/Parser	21
3.2	식의 계산	21
3.2.1	미지수와 식	21
4	함수의 정의와 다양한 함수의 성질	22
5	벡터의 정의와 응용	23
6	행렬의 정의와 계산	24

7 함수의 극한과 미분	25
8 적분의 정의와 적분법	26
9 다변수 미적분학과 그래디언트	27
10 확률과 조건부확률	28
11 통계적 추론	29

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 강의¹를 위해서 제작되었습니다. 본 강의에서 구현할 코드들과 그 스켈레톤은 깃헙²에서 찾아볼 수 있습니다. 제공된 소스 코드는 파이썬 3.5로 윈도우 환경에서 테스트되었습니다.

¹강의 웹페이지

²강의자료 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 튜토리얼¹의 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.

¹[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 부분²을 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 지원³⁴합니다.

- 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.⁵

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.

²장고걸스 튜토리얼 : 배포

³참조 링크 1(위키북스), 참조 링크 2(공식 문서)

⁴여기 리스트된 데이터형이 전부는 아니지만, 중요한 데이터형들이니 잘 알아두시길 권장합니다.

⁵프로그래밍 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, 실수⁶, 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'>
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)

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

```

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.

- 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 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 코드⁷를 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})
```

⁷[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.⁸

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

```

⁸물론 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.

클래스란 클래스는 흔히 과자들과 그 과자들로 찍어낸 과자로 비유됩니다. 클래스는 객체를 만들기 위한 코드 템플릿⁹을 말합니다. 인스턴스는 이 템플릿을 이용하여 만들어진 코드 덩어리로 생각할 수 있습니다. 예¹⁰를 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):
```

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

¹⁰본 예시는 <https://wikidocs.net/28점프> 투파이썬에서 참고하였습니다.

```

6     self.result += num
7     return self.result
8
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.¹¹

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.

¹¹파이썬의 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.

여기서 클래스(a)와 인스턴스(b)의 차이를 살펴보기 위해서 다음 코드를 실행해 보겠습니다.

Implementation 1.24: Making a MyString Class (example_class1.py)

```

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 쿠키틀의 예시를 들면, 쿠키 하나를 바꾼다고 쿠키틀이 바뀌지는 않는 것과 같습니다. 반대로, 쿠키틀을 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())
```

1.2.4 파이썬 인터프리터의 이해

본 단락에서는 주어진 소스 코드를 파이썬이 계산하는 법을 알아볼 것¹²입니다. 본격적인 설명에 앞서, 변수의 종류에 대해서 설명하겠습니다. 일반적으로 변수에는 다음의 세 종류가 있습니다.

- Bound variable : 어떤 값이나 다른 변수에 의해서 값이 결정되는지 정해진 변수

¹²파이썬 공식 Documentation 참조

- Binding variable : Bound variable의 값을 결정하는 변수
- Free variable : Bound되지 않은 변수

이 때, 파이썬 인터프리터¹³는 **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 해더는 루프 코드블럭 안에서 for loop 해더에서 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

```

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

¹³사실 많은 인터프리터가 대부분 이렇게 동작합니다.

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

집합과 함수

본 단원에서는 집합과 그 표현, 그리고 집합의 구현과 함수에 대해서 다룹니다.

2.1 집합

2.1.1 집합의 정의

집합의 정의는 무엇일까요? 국립국어원에 따르면, 집합의 정의는 아래와 같습니다.

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.^{1 2}

우리의 목적은 집합을 구현하는 것이므로, 위 정의를 조금 더 생각해 보겠습니다. 먼저, 집합을 정의하기 위해서는 다음의 두 가지 표준이 필요합니다.

- 임의의 한 원소가 그 모임에 속하는지를 알 수 있어야 함
- 모임에 속하는 임의의 두 원소가 다른가 같은가를 구별할 수 있어야 함

이를 프로그래밍의 관점에서 생각해 보면, 결국 우리는 다음과 같은 두 개의 함수가 필요하다는 결론을 내릴 수 있습니다.

- membership 함수 : 임의의 한 원소가 그 모임에 속하는지를 알 수 있어야 함 -> 어떤 object를 입력받아, True 혹은 False를 리턴하는 함수³

이 경우, 어떤 원소에 대해서 membership 함수가 True를 리턴하면 그 원소는 집합의 구성원으로 생각할 수 있습니다. 예를 들어서, 짝수들의 집합이라면 먼저 원소가 정수인지 체크한 다음, 2로 나눈 나머지가 0인지 체크하는 함수를 membership 함수로 생각할 수 있을 것입니다. 이는 파이썬 클래스로 생각하면 `isinstance` 내장함수와 같은 역할을 합니다.

- equivalence 함수 : 모임에 속하는 임의의 두 원소가 다른가 같은가를 구별할 수 있어야 함 -> 어떤 두 object를 입력받아 True 혹은 False를 리턴하는 함수

파이썬의 경우, 클래스의 magic method 중 하나로 `__eq__`를 제공합니다. 물론 `__eq__` 함수의 기본값은 두 인스턴스가 담겨있는 주소가 같은지를 체크하지만, 필요한 경우 `__eq__` 함수를 오버라이드해서 우리가 원하는 형태로 구현하여 사용할 수 있으므로 충분히 집합이라고 할 수 있습니다.

¹국립국어원의 정의 2번 참조.

²사실 이 정의만 국립국어원의 정의를 따르는데, 이는 집합 자체가 정의하기 매우 어렵기 때문입니다. 여기서는 간단하게 위 정의를 따르고 넘어갑니다.

³이러한 함수를 predicate라고도 합니다.

따라서, 사실 위 두 함수는 이미 파이썬의 클래스에서 이미 구현되어 있습니다. 즉, 파이썬에서 말하는 클래스는 수학적으로 볼 때, 이미 좋은 집합의 예시라고 할 수 있습니다. 본 강의에서는 편의를 위해서 PySet을 만들어, 집합의 클래스로 생각할 것입니다.

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

```
3  
4 class PySet:  
5     def __init__(self,  
6                  membership,): # membership checking  
7         self.membership = membership
```

즉, membership 함수와 eq_func을 가지고 있는 객체를 집합으로 생각할 것입니다.

2.1.2 집합의 연산

집합의 연산에는 합집합, 교집합, 여집합과 카테시안 곱이 있습니다.

2.1.3 집합의 종류

본 수업에서는 집합을 두 가지 기준에 따라서 나눌 것입니다. 첫 번째 기준은 집합의 원소의 갯수에 따라서 3가지 - 유한집합, 가산집합, 비가산집합 - 으로 나눌 것입니다. 두 번째 기준은 집합의 원소들 간의 순서 여부에 따라서 ordered set, partially ordered set으로 나눕니다.

2.1.3.1 집합의 크기에 따른 구분

유한집합

가산집합

비가산집합 비가산집합은 컴퓨터로는 구현할 수 없기에, 여기서는 구현하지 않습니다⁴

집합의 원소간 순서 여부에 따른 구분

Totally Ordered Set

Partially Ordered Set

⁴비가산집합이 구현 불가능한 것에 대한 증명은 튜링 머신의 계산가능성 부분을 참고하면 됩니다. 여기서는 다루지 않습니다.

2.1.4 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

본 단원에서는 수식을 계산하는 법을 간단하게 복습하고 넘어가고, 이를 파싱하는 클래스 PyFormula 를 구현하여 문자열 형태의 수식을 트리 형태로 만들어 앞으로의 진행 과정에 계속 사용할 것이다.

3.1 파싱

3.1.1 What is Parsing?

3.1.2 Grammar

3.1.3 Abstract Syntax Tree

3.1.4 Tokenizer/Parser

Tokenizer Implementation

Parser Implementation

Extending Parser

3.2 식의 계산

3.2.1 미지수와 식

Chapter 4

벡터와 행렬

4.1 벡터

4.2 행렬

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