Python 및 정보과학 입문 교재

Python 3.7 및 기본적인 알고리즘에 대한 이해

교재 제작 v3.3
이재호 2019-08-09

목차

1 Python 소개 및 기본 요소

1.1 강의 계획

본 교재는 Python 3.7의 기본적인 문법을 익힐 수 있는 내용과 예제들이 수록되어 있습니다. 난이도는 프로그래밍 언어를 처음 접하더라도 익힐 수 있도록 구성되어 있으며, 상황에 맞추어 유동적으로 진행할 예정입니다. 본 수업은 algorithm알고리즘[1.1]을 배우는 것보다는 Python 3 언어의 기본적인 문법 숙지와 더불어 자주 사용되는 패턴에 익숙해지는 것에 초점을 맞추고 있습니다. 또한, 정보과학적인 사고(순차적으로 논리적인 비약이 없이 문제 해결을 할 수 있는 능력)를 배우고 간단한 알고리즈믹 문제의 해결법을 배우는 것이 목적입니다. 아래의 진도표는 회차 당 세 시간 수업을 기준으로 한 것으로, 가변적일 수 있습니다.

1회차 Python 소개 및 기본 요소

2회차 Functions함수와 Conditionals조건문

3회차 Boolean Functions불리언 함수와 Loops반복문 기본

4회차 Lists리스트, Strings문자열, Counters카운터

5회차 Quantifiers한정자와 While 문

6회차 Loops반복문 응용과 파일 입출력

~~**7회차** Toy Robot~~

7회차 재귀법Recursion과 Python의 다양한 객체

8회차 람다Lambda 함수

기본적으로 알고리즘보다는 언어 자체의 문법과 활용에 초점을 맞춘 커리큘럼이므로, 알고리즘을 본격적으로 다루기 위해서는 자료 구조와 알고리즘에 대해서 깊게 다루는 서적을 읽어보시는 것을 추천드립니다. Python과 같이 언어를 익히고 난 다음에는 알고리즘을 배우고 실제로 구현할 수 있는 준비가 되어 있을 것입니다.

알고리즘에 대해서 더 배우고 싶으시다면 흔히 CLRS라고 불리우는 *Introduction to Algorithms 3/e* (*MIT Press, 2009*)를 추천드립니다. 혹시 더 깊은 이해를 원하신다면, 저도 아직 읽어보지는 못했지만 커누스 교수의 TAOCP로 불리우는 *The Art of Computer Programming* 시리즈[1.2]를 읽어보시면 됩니다.

[1.1] 간단히 말해서, 어떤 값들을 입력받아 값들을 출력하는 잘 정의된 과정을 말합니다.

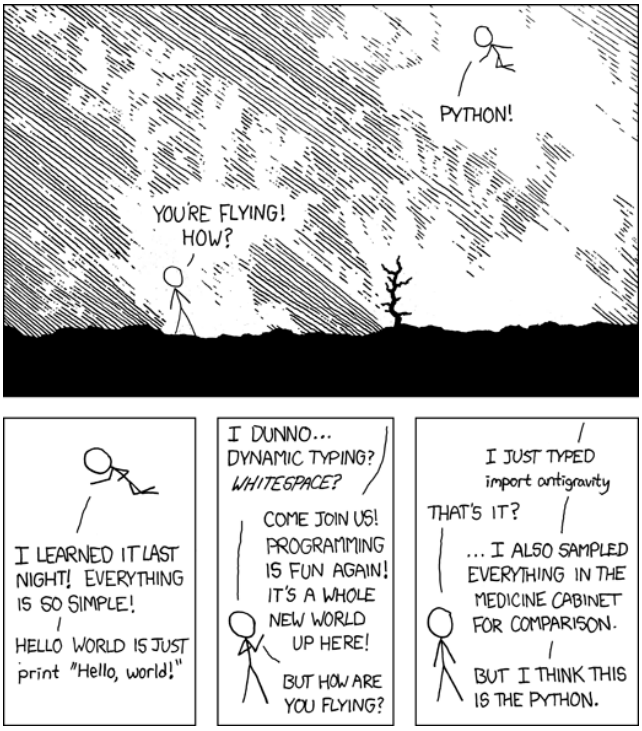
[1.2] 커누스 교수가 1968년부터 집필을 시작해 현재는 4권의 일부분까지 완성되어 있으며, 현재 7권까지 계획이 되어 있습니다. 빌 게이츠는 본인이 훌륭한 프로그래머라고 생각하면 TAOCP를 읽어보고, 다 읽을 수 있다면 자신에게 이력서를 보내라고 하였습니다. 여담으로, 커누스 교수는 이 책의 조판을 위해서 본 교재를 위해서 사용된 TEX을 개발하였습니다.

1.2 들어가기 전에

Python은 문법이 단순하면서도 활용 가능성이 무궁무진한 언어입니다. 단순히 '정보과학'만을 위해서 Python을 배운다기보다는, 평생 활용할 수 있는 도구를 배우는 것입니다. Python은 R이나 Matlab 등과 함께 학문적인 용도로도 쓰임이 많은 언어입니다. 특히 Matlab과 다르게 Python은 오픈 소스에 무료인데다가, 단순히 데이터 처리만을 위한 언어가 아닙니다. NumPy와 Matplotlib 등의 패키지를 활용한다면 Matlab이나 Mathematica와 같은 상용 프로그래밍 언어가 할 수 있는 다양한 작업들을 대등하게 할 수 있기도 합니다.

실제로 저는 학업 중 물리학, 천문학, 화학, 생물학 등의 분야에서 데이터 분석 및 차트 제작을 위해 Python을 활용하거나, Raspberry Pi와 연동하여 다양한 센서를 사용해 프로젝트를 진행하는데에도 사용했습니다. 또한 Django 등의 프레임워크를 사용한다면 웹서버를 구축할 수도 있는데, YouTube, Dropbox, Facebook, Netflix, Google, Instagram, Spotify 등의 유명 사이트들이 Python을 활용하여 서비스를 제공하고 있습니다. 이처럼 Python은 배워두면 무궁무진한 방면에서 활용할 수 있는 가능성을 가지고 있는 언어입니다.

1.3 Python이란?



흔히 Python을 "batteries included"(배터리가 들어간) 언어라고 합니다.

Python은 1991년에 Guido van Rossum이 발표한 프로그래밍 언어로, 문법이 굉장히 쉬우면서도 높은 생산성을 가지고 있는 언어입니다.[1.3] 특히 pseudocode의사코드와 유사하기 때문에 비교적 배우기 쉬워, 많은 학교에서 프로그래밍 입문 수업 언어로 Python을 채택하고

[1.3] 그가 1989년 크리스마스 연휴에 취미로 Python을 제작하였던 것이 그 시작입니다.

있습니다. 또, **프로그래밍 언어로 할 수 있을 법한 거의 모든 기능들이 이미 Python을 위한 패키지로 구현되어 있습니다.** 이 때문에 Python을 흔히 "batteries included"라고 부릅니다. 따라서 속도가 중요한 작업이 아니라면 C/C++보다 Python을 쓰는 것이 효율적입니다. (연산이 아니라 업무의 효율을 말하는 것입니다.)

프로그래밍에 익숙치 않더라도 컴퓨터에 관심이 많다면 C, C++, Java 등의 언어와 함께 Python도 한 번 쯤 들어보았을 정도로, Python은 점유율 상위 다섯 언어 안에 드는 주류 언어입니다. Python은 1995년에 등장한 Java보다도 오래 전에 만들어진 언어로, 긴 역사를 가지고 있습니다. 그 만큼 버전의 숫자도 큰데, 이 글을 작성하는 현 시점에서 Python 2의 최신 버전은 2.7.16, Python 3의 최신 버전은 3.7.3입니다. 한동안은 Python 2와 Python 3가 동시에 개발되기도 하였고, Python 3보다는 Python 2를 사용하는 것이 낫다고 말하던 때가 있었습니다.[1.4] 그러나 현재는 Python 3를 배우고 사용하는 것이 권장됩니다. Python 2는 2.7이 최종 버전이고, 이후에는 bugfix 릴리즈만 있을 예정인데다가 2020년까지만 지원을 할 예정입니다. 나아가 최신 버전의 Ubuntu(Linux 배포판의 한 버전)는 Python 3를 기본 Python 버전으로 설정하고 있습니다.

Python은 interpreter인터프리터 언어입니다. Interpreter 언어란 소스 코드를 한 줄씩 기계어[1.5]로 번역하는 방식의 언어입니다. 조금 더 가독성을 높인, 기계어와 일대일 대응이 되는 (마찬가지로 저급 언어인) 어셈블리어가 있기도 합니다. 고급 언어를 기계어로 변환시키는 방법은 크게 두 가지가 있습니다. Compiler컴파일러와 interpreter입니다. C와 같은 언어는 compiler 언어로, 코드 전체를 한꺼번에 기계어로 변환시킵니다. 이 때문에 실행 속도는 빠르다는 장점이 있지만, 코드를 수정하기 위해서는 다시 한 번 전체를 기계어로 변환시키는 과정이 필요합니다. 반면 Python은 한 줄씩 기계어로 번역하기 때문에 실행 속도는 다소 느리지만, debug디버그[1.6]에 유리합니다. 한 줄씩 실행하기 때문에 애초에 compile을 거부하는 compiler 언어와는 다르게 버그가 있는 해당 줄까지 코드를 실행시켜주기 때문입니다. 이러한 장단점 때문에 프로그램의 골격을 Python으로 만들고, 빠른 연산이 필요한 부분만 C로 만드는 것도 가능합니다.

마지막으로, Python이 얼마나 쉽고 직관적인 언어인지 알아봅시다. 만약 A+가 F, C+, B0, A-, A+에 있으면 "A+가 있습니다."를 실행해주는 Python 코드를 봅시다.

```
if "A+" in ["F", "C+", "B0", "A-", "A+"]: print("A+가 있습니다.")
```

프로그래밍을 할 줄 모르더라도 저 코드를 이해할 수 있을 것입니다. 이처럼 Python은 인간이 사고하는 방식을 그대로 옮겨 놓았다고 해도 과언이 아닐만큼이나 직관적입니다. 익숙해진다면 자신이 하고 싶은 일을 코드로 옮기려고 끙끙댈 필요 없이, 생각하는 그대로 코드를 작성할 수 있을 것입니다. 참고로, 이와 같은 일을 하려면 C언어에서는 다음과 같이 해야 합니다.

```
1 #include <stdio.h>
```

[1.4] 1, 2년 전까지만해도 Python 2 vs Python 3의 논쟁이 비일비재하게 일어났었습니다.

[1.5] 기계가 바로 이해할 수 있는 저급 언어로, 0과 1의 이진수로만 구성되어 있는 언어로, 모든 CPU를 구동시키기 위해서는 C와 같은 고급 언어를 저급 언어로 변환시키는 과정이 필요합니다.

[1.6] 코드에서 bug버그, 즉 오류를 제거하는 것을 의미합니다.

```c
#include <string.h>
int main(void) {
    char grades[][3] = {"F", "C+", "B0", "A-", "A+"};
    for (int i = 0; i < 5; ++i) {
        if (strncmp(grades[i], "A+", 2) == 0) {
            printf("A+가 있습니다.\n");
        }
    }
    return 0;
}
```

단 한 줄의 Python 코드로 될 일을 C 언어로는 10줄 이상으로 작성해야 하는 것입니다.

1.4 Python의 기본 요소

Hello, World

```
#include <stdio.h>                    include information about standard library

main()                                define a function named main
                                      that receives no argument values
{                                     statements of main are enclosed in braces
    printf("hello, world\n");         main calls library function printf
}                                     to print this sequence of characters;
                                      \n represents the newline character
```

The first C program.

K&R *The C Programming Language*의 첫 프로그램

1978년에 씌여진 K&R이라고 불리우는 *The C Programming Language*에서 예시로 사용된 이후 첫 프로그램은 보통 `Hello, World!`를 출력하는 것으로 시작됩니다. 먼저, IDE에 다음과 같은 코드를 입력해봅시다:

```python
print("Hello, World!")
```

실행시켜보면 `Hello, World!`가 출력될 것입니다. C 코드에서는 `include` 등 이것저것 따로 입력해야 할 것이 있는데, Python에서는 사뭇 간단한 것을 볼 수 있습니다. 이제 Python의 문법에 대해서 본격적으로 알아봅시다.

Values값과 Variables변수

윈도우를 쓴다면 명령 프롬프트나 powershell, 맥이나 리눅스와 같은 유닉스 계열을 쓴다면 터미널, 혹은 그냥 IDE에 내장된 쉘shell에 다음을 입력해 봅시다. Interpreter 언어이기 때문에, 명령을 한 줄씩 입력하여 명령을 수행할 수 있습니다. (>>>는 직접 입력하는 것이 아닙니다.)

```
1 >>> a = (1+2+3+4)/2
2 >>> b = a - 1
3 >>> c = 3
4 >>> print(b + c)
5 7
6 >>> print(b, c)
7 4  3
8 >>> d = c
9 >>> a = d
10 >>> print(a)
11 3
```

`a = (1+2+3+4)/2`는 등호 왼쪽에 있는 `a`에 등호 오른쪽에 있는 $\frac{1+2+3+4}{2}$를 배정한다는 뜻입니다. 즉, 수학에서 말하는 등호와는 의미가 다른 것이지요. 줄 2의 `b = a - 1`은 현재 `a`에 배정된 $\frac{1+2+3+4}{2} = 5$의 값에 1을 뺀 4를 `b`에 배정한다는 뜻입니다. 이 때, 좌변에 있는 `a`, `b` , `c`를 variables변수[1.7], 우측의 `(1+2+3+4)/2`를 expression표현이라고 합니다. 또한 이러한 변수에 배정되는 것을 value값이라고 합니다. 다시 위의 예시로 돌아가서, 줄 2에 있는 좌변의 `b`는 variable, 우변의 `a - 1`는 value인 것이죠. 영어 문장으로 "Let {variable} be {value}."가 말이 되는지 대입하여 보면 쉽게 알 수 있습니다. Variable과 value는 문자인지 숫자인지의 여부가 아니라 대입이 되는 대상인지 대입이 되어지는 대상인지의 여부가 결정짓습니다. 물론, variable이 value 그 자체가 될 수 있습니다. 줄 8의 예시와 같은 경우, 우변의 `c`는 variable이면서 `d`라는 variable에 대입되는 value입니다. 그렇다면, 아래와 같은 표현을 어떨까요?

```
>>> 10 = a
```

지금까지 잘 따라왔다면, value가 위치해야 할 좌변에 value인 literal[1.8]이 위치해 잘못된 syntax 구문이라는 것을 알 수 있습니다. `SyntaxError: can't assign to literal`이라는 오류 log를 볼 수 있을 것입니다. (여담이지만, 앞으로 수 없이 많은 오류 log를 볼 텐데, 이를 꼼꼼히 읽어보는 습관을 들입시다. 오류 log를 무시하고 디버깅하는 것은 마치 백사장에서 바늘을 찾는 것

[1.7] 필요에 따라 영문의 'variable', 국문의 '변수'를 사용하여 서술하겠습니다. 이외의 용어도 처음에는 영문과 국문을 병기한 후, 필요에 따라 둘 중 하나의 표현만 사용하겠습니다.

[1.8] 어떤 객체에 value를 부여할 수 있는 것입니다. `0`은 `int` literal, `"python"`은 `str` literal입니다. `int`, `str`이 무엇인지는 조금 뒤에 type형이라는 것을 배우면 알 수 있습니다.

과도 같습니다.) 혹은, "let 10 be a"라는 표현이 말이 되지 않는다는 것을 통해서도 직관적으로 잘못되었음을 파악할 수 있습니다.

이러한 변수는 여러 가지 종류가 있습니다. 이를 data type자료형, 혹은 간단히 type형이라고 합니다. `-1`, `0`, `76` 등의 값은 `int`[1.9] type, `3.14159`, `0.`, `6.626e-34`, `6.022E23` 등의 값은 `float`[1.10] type, `"Hello, world!"`, `'python'`, `""` 등의 값은 `str`[1.11] type입니다. Type에 대해서는 조금 뒤에 더 자세히 살펴봅시다.

변수는 어떤 값이 배정된 것이라고 했는데, 이것을 assign되었다고 하며 값이 written되었다고도 표현할 수 있습니다. 이렇게 변수에 저장된 값은 read읽을 수 있는데, 첫 예시의 줄 2처럼 `a`에 저장된 값 `5`를 불러오는 것이 이에 해당합니다. 또한 줄 4의 `print`를 통해서도 값을 읽어올 수 있습니다.

변수는 서로 다른 값이 배정될 수 있습니다. 첫 예시에서 줄 9를 보면, `5`가 저장되었던 `a`에 `3`이 저장된 `d`의 value가 다시 `a`에 쓰여지는 것을 볼 수 있습니다. 줄 10에서 이 사실을 재확인할 수 있고요. 이와 같이 변수는 재활용될 수 있고, 이는 개수를 세거나(counting) 특정 사건을 기록하기 위해 flag 등으로 사용하는데 도움이 됩니다.

```
1 >>> summ = 0
2 >>> summ = summ + 1
3 >>> summ = summ + 1
4 >>> print(summ)
5 2
```

더 자세한 활용은 차차 Python을 익혀가면서 알아봅시다.

이 뿐만이 아니라, Python은 여러 변수에 여러 값을 한 번에 배정하는 것(multiple assignment)을 허용합니다. 나아가 **변수의 swapping을 매우 손쉽게 할 수 있습니다.** 이 둘을 다음 예시를 통해 함께 확인합시다.

```
1 >>> a, b, c = 2, 7, 12
2 >>> a, b, c = b, c, a
3 >>> print(a, b, c)
4 7 12 2
5 >>> a, b = b % a, a
6 >>> print(a, b)
7 5 7
```

Multiple assignment를 허용하지 않는 대다수의 언어에서는 임시 변수를 도입해야 합니다. 예컨대 줄 5의 경우 아래와 같은 방법을 사용해야 합니다.

[1.9] 정수라는 뜻의 integer에서 따온 명칭입니다. 단, 수학에서 3.0은 정수이지만 `3.0`은 `float` type입니다.

[1.10] 부동 소수점 혹은 떠돌이 소수점이라는 뜻의 floating point에서 따온 명칭입니다. 실수가 아니라 근삿값이라는 의미에 더 가깝습니다.

[1.11] 나열이라는 뜻의 string에서 따온 명칭입니다.

```
>>> tmp = a
>>> a = b % tmp
>>> b = tmp
```

Python에서는 마치 하노이의 탑을 연상시키는 이러한 과정을 시행하지 않아도 됩니다.

마지막으로, 변수의 이름으로 정할 수 없는 특정 문자열이 있습니다. Python이 내부적으로 사용하는 `int`, `str`, `if`, `else`, `for`, `range`, ...등이 이에 해당합니다.[1.12] 그리고 변수명은 영어 대소문자, 숫자, 그리고 _로만 이뤄져 있어야 합니다. (Python 3부터는 한글로도 이름을 명명할 수 있습니다.) 나아가 숫자로 시작할 수 없는데, `1st_name`과 같은 문자열을 변수명으로 지정할 수 없는 것입니다.

Expressions표현

Expression은 variable, value, 그리고 operator들의 조합입니다. 우리가 흔히 부르는 사칙 연산 `+`, `-`, `*`, `/`와, 나머지를 구해주는 `%`, 지수를 뜻하는 `**` 등이 이에 해당합니다. **주의할 점은, `^`이 지수를 뜻하는 것이 아니라 `**`이라 것입니다.** 또한 `//`는 몫을 구해줍니다. 아래의 예시를 봅시다.

```
>>> 12 + 5
17
>>> 12 - 5
7
>>> 12 * 5
60
>>> 12 / 5
2.4
>>> 12 // 5
2
>>> 12 % 5
2
>>> 12 ** 5
248832
>>> 12 ^ 5
9
```

`^`는 bitwise XOR의 연산자로서, $12 = 1100_2$, $5 = (0)101_2$이므로 digit이 다른 $2^3, 2^2, 2^1$ 자릿수만 1을 취한 $1001_2 = 9$가 `12 ^ 5`의 값입니다. `^`은 지수의 연산이 아닙니다! 연산의 순서는 기

[1.12] `int` 등 type 명은 배정이 가능하지만, 형 변환을 위한 함수로 사용되므로 사용하면 안됩니다. 위에 나왔던 코드 중 `summ`이라고 썼던 변수명도 `sum`으로 쓰지 않은 이유는 `sum`에 해당하는 내장 함수가 있기 때문입니다.

본적으로 괄호(`(...)`), unary 연산(`+x`, `-x`), 지수(`**`), 곱셈/나눗셈/나머지 연산(`*`, `/`, `%`), 덧셈/뺄셈(`+`, `-`)의 순서입니다. 헷갈리는 경우나 혼동을 불러올 수 있는 경우에는 `(...)`를 사용하여 순서를 명시할 수 있습니다. 또는 논리적인 블럭이 되는 경우 괄호를 사용하여 묶어주는 것이 권장됩니다.

참고로, Python 2에서는 나눗셈을 할 때 정수끼리 행하면 몫만이 구해집니다. `12 / 5 = 2`와 같이 말입니다. 반면 `12.0 / 5 = 2.0`와 같이 제수나 피제수 중 하나라도 `float` 형이면 결과도 실제 `float`의 나눗셈의 결과로 나옵니다. 위와 같은 결과를 인터넷이나 서적에서 보신다면 Python 2 코드이므로 유의하시기 바랍니다.[1.13]

코드를 작성할 때에는 특별한 경우를 제외하고는 가독성이 중요합니다. 예컨대, 중복되는 값이나 의미가 있는 값은 특정 변수에 저장하여 해당 변수를 통해 식을 표현하는 것이 바람직합니다. 아래의 예시를 봅시다.

```
>>> S = ((3 + 4 + 5) * (-3 + 4 + 5) * (3 - 4 + 5) * (3 + 4 - 5))**0.5
>>> a, b, c = 3, 4, 5
>>> s = (a + b + c) / 2
>>> S = (s * (s - a) * (s - b) * (s - c))**0.5
```

비록 줄 수는 늘어났지만, 줄 1의 표현보다는 줄 4의 표현이 가독성이 높을 뿐만이 아니라 더 일반적이어서 값을 바꾸기 위해서는 줄 2의 숫자 부분만 변경을 하면 된다. 반면 줄 1의 표현의 경우 `+`와 `-`의 부호 구분에서 실수를 할 수 있고 다른 값을 대입하기 위해서는 12 부분에 수정을 가해야 한다.

마지막으로 소개할 문법은 위에서 잠시 언급한 개수 세기 등에서 유용하게 쓸 수 있습니다. 변수 뒤에 산술 연산자(`+`, `-`, `*`, `/`, `%`, `**`) 뒤에 바로 `=`를 붙인 후 수를 쓰는 syntax입니다.[1.14] `x += 1`과 같이 말입니다. 이는 해당 변수에 저장된 값에 등호 뒤에 쓰인 값을 더한다는 의미로, `x = x + 1`과 동일한 의미를 가지고 있습니다. 아래와 같이 응용할 수 있습니다.

```
>>> x = 4
>>> x += 2
>>> x
6
>>> x -= 1
>>> x
5
>>> x *= 2
>>> x
10
>>> x /= 5
```

[1.13] 또한 이 교재도 원래 Python 2 기준으로 쓰여져 있었는데, 이와 관련해 미처 수정되지 않은 부분이 있다면 알려주시기 바랍니다.

[1.14] `int`나 `float`형에서는 모든 산술 연산자를 쓸 수 있고, `str`형에 대해서는 정의가 되어 있는 `+`만 사용할 수 있습니다.

```
>>> x
2
>>> x %= 3
>>> x
2
>>> x **= 3
>>> x
8
```

참고로, C/C++이나 Java와 같은 언어에는 `++`, `--`와 같이 쉽게 1을 더하거나 뺄 수 있는 연산자가 있습니다. `a`가 3의 값을 가지고 있었을 때 `a++`를 하면 4가 되는 것이지요. C++의 `++`이 해당 연산자에서 따온 것입니다. 그러나 `a++`과 `++a`에 따라서 연산 후 결과는 같지만 실제 코드에서 사용되는 위치에 따라 수행 결과가 달라지는 등 버그의 원인이 되기 때문에 Python이나 모던 언어에서는 해당 연산자를 문법에 넣지 않는 추세입니다.

Types형

위에서 간단히 소개한 바 있는데, variable의 종류를 type형이라고 합니다. 현재로서는 지금까지 언급한 `int`(정수형), `float`(실수형), `str`(문자열) 세 가지 type만 알아두시면 됩니다. 지금까지는 숫자가 실수형임을 명시할 때 `3.`과 같이 온점을 찍어 표현하였는데, type conversion형변환이라는 것을 사용하여도 됩니다. **형 변환은 정수형과 실수형 간에서 자유롭게 가능하고, 문자열의 경우에는 그 자체가 수일 경우에만 변환이 가능합니다.**

```
>>> x = 76
>>> x
76
>>> float(x)
76.0
>>> str(x)
'76'
>>> pi = 3.14
>>> pi
3.14
>>> int(pi)
3
>>> str(pi)
'3.14'
>>> s = "1"
>>> s
'1'
>>> int(s)
```

```
1
>>> float(s)
1.0
```

위 예시를 통해 필요한 모든 경우를 파악하셨을 것입니다. 또한, `type(·)`를 통해 직접 type을 확인할 수 있습니다.

```
>>> print(type(76))
<class 'int'>
>>> print(type(76.))
<class 'float'>
>>> print(type("76"))
<class 'str'>
```

Input/Output입출력

지금까지는 Python shell에서만 명령을 실행했습니다. 그렇기 때문에 a에 담긴 값을 알기 위해서는 굳이 `print(a)`가 아니라 a를 치는 것 만으로도 충분했습니다. 하지만 여러 줄의 코드를 한꺼번에 작성하여 실행할 때에는 이런 방식의 접근이 불가능합니다. 또, 코드를 실행 중일 때 어떤 입력을 받기 위해서는 지금까지와는 다른 방법이 필요합니다. Shell과는 다르게 한 줄씩 직접 입력하는 방식이 아니기 때문입니다. 값을 출력하는 것은 지금까지 해왔던 것처럼 `print`를 사용하면 되는데, 아래에서 `print`에 대해 좀 더 알아보겠습니다. Shell이 아니라 파일을 만들어서 실행시킵니다.

```
today = "Monday"

print("Today is", today)
print("Today is " + today)

print("\nprintf style:")
print("Today is %s" % today)
print("Today is %(day)s" % {"day": today})

print("\nPython 3, back-ported to Python 2:")
print("Today is {}".format(today))
print("Today is {day}".format(day=today))

print("\nPython 3.6+, Formatted String Literals:")
print(f"Today is {today}")
```

위 예시는 Python에서 `Today is Monday`를 출력하는 여러가지 방법을 나열한 것입니다. 첫 번째(줄 3)는 `,`를 사용하여 `print` 함수 내의 여러 인자들을 출력하는 방식입니다. 자동으로 띄어쓰기가 들어간다는 것에 유의합니다. 두 번째(줄 4)는 문자열의 덧셈을 통해 출력한 것으로, 띄어쓰기는 직접 앞 `Today is`에 추가하였습니다. 다음 예시부터는 문자열 포맷팅에 관한 내용입니다. 먼저 줄 7과 8의 예시는 과거 사용되었던 방식으로, 현재에도 사용할 수 있는 방식입니다. C 언어의 `printf`와 유사한 방식입니다. `%s`는 뒤 `%` 뒤의 변수를 문자열 형식으로 넣으라는 뜻입니다. 만약 이름을 붙여서 지정하고 싶다면 줄 8과 같이 사용하면 됩니다. Python 3에서 시작되어 현재는 Python 2로 백포트된 문자열 포맷팅 방식은 줄 11과 12에 나와 있습니다. `{}`로 지정된 부분에 뒤 `.format()` 메소드 내부의 인자가 대입되는 것을 확인할 수 있습니다. 마지막 15 줄의 방식이 가장 최근에 도입된 Formatted String Literals라는 것입니다. 문자열 앞에 `f`를 붙인 후 단순히 중괄호 안에 원하는 변수 이름을 넣으면 대입이 됩니다. 만약 Python 3.6 이상 버전을 쓰고 있다면 사용이 가능하므로 되도록이면 해당 방식을 사용하는 것이 권장됩니다.

이제는 값을 입력하는 법에 대해 알아봅시다. `input(·)` 함수를 사용하면 됩니다. 다음과 같은 예시를 살펴봅시다.

```
1 today = input("What day is it today? ")
2 print("Today is", today)
3
4 s = input("Enter the number you want to know the square root of: ")
5 n = float(s)
6 print(n**.5)
```

위 코드를 실행시키면 창에 `What day is it today?` 가 출력된 후 입력이 될 때까지 기다립니다. 키보드로 값을 입력한 후—`Thursday`를 입력했다고 합시다—리턴 키를 치면 값이 입력되고, `Today is Thursday.`가 출력될 것입니다. 또한, 수를 입력 받을시 `int(·)`나 `float(·)`로 형 변환을 수행해줘야 합니다. **`input(·)`이 넘겨주는 값은 항상 `str` 형이기 때문입니다.**

1.5 예제

1. 1부터 n까지 자연수의 제곱의 합과 세제곱의 합의 차이를 구하는 코드를 작성하세요. 단, 아직 배우지 않은 `for` 문 없이 다음의 공식을 사용하세요:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$
$$\sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2$$

```
1 n = int(input("Enter n: "))
2 # Add here!
```

2. 원탁 주위에 a, b, c, d, e가 앉아 있습니다. 각자는 자신이 좋아하는 정수를 종이에 적은 후, 양 옆에 앉은 사람의 정수를 더합니다. 최종적으로 각자가 얻게된 수를 출력하도록 다음 코드를 완성하세요:

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
c = int(input("Enter c: "))
d = int(input("Enter d: "))
e = int(input("Enter e: "))
print("Favorite integers: ", a, b, c, d, e)
# Add here!
print("Final integers: ", a, b, c, d, e)
```

3. 이차방정식 $ax^2 + bx + c = 0$의 해를 구하는 코드를 작성하세요. 단, 판별식 $b^2 - 4ac > 0$이라고 가정합니다.

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
c = int(input("Enter c: "))
# Add here!
# x1 = ...
# x2 = ...
print("Solutions for the quadratic equation are ", x1, " and ", x2)
```

4. 다음 표현의 값을 예상하세요.

```
>>> 3*2**6+12
???
>>> 32/7+2**3/3
???
>>> 13/3%3+2.0
???
>>> 13//2/4.
???
>>> int(2**(1/2))+1
???
```

5. Python 3에서 가능한 변수명을 모두 고르세요.

- `if`
- `sum`
- `max`
- `1st_var`
- `var_1`
- `this+that`
- `_self`
- 변수
- `r&e`

6. 다음 코드는 여섯 개의 실수 `x1`, `x2`, `x3`, `y1`, `y2`, `y3`을 입력받습니다. `i`가 `1`, `2`, `3`을 취할 때 (`xi`, `yi`)가 좌표평면에서 서로 다른 세 점을 나타내는 좌표라고 가정합니다. 이때 세 점이 이루는 삼각형의 면적을 계산하는 코드를 작성하세요.
   참고로,
   - 세 변의 길이가 a, b, c인 삼각형의 면적은 $s = \frac{a+b+c}{2}$ 일 때 $\sqrt{s(s-a)(s-b)(s-c)}$입니다.
   - 또한, 어떤 두 벡터 $\mathbf{u}$와 $\mathbf{v}$가 이루는 삼각형의 면적은 $\frac{1}{2}\|\mathbf{u}\times\mathbf{v}\| = \frac{1}{2}\|\mathbf{u}\|\|\mathbf{v}\|\cos\theta$입니다.
   - 이때, $\cos\theta = \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{v}\|\|\mathbf{u}\|}$ 입니다.

```python
x1 = int(input("Enter x1: "))
y1 = int(input("Enter y1: "))
x2 = int(input("Enter x2: "))
y2 = int(input("Enter y2: "))
x3 = int(input("Enter x3: "))
y3 = int(input("Enter y3: "))
# Add here!
# area = ...
print("Area of the triangle is", area)
```

7. `a`를 `b`로 나눈 나머지를 `%` 없이 사칙연산과 `//` 만으로 구하세요.

```python
a = int(input("Enter a:"))
b = int(input("Enter b:"))
# Add here!
# r = ...
print("Remainder is", r)
```

참고. 다음 코드는 양의 실수에 대해 올림 함수 $\lceil x \rceil = \min\{n \in \mathbb{Z} \mid n \geq x\}$를 계산합니다. `int(·)`, `+`, `-`만을 사용하여 구현하세요.[1.15]

- 힌트: `int(·)` 함수의 그래프를 그려보세요.

```
x = float(input("Enter x:"))
# Add here!
# ceil_x = ...
print("Ceil(x) =", ceil_x)
```

2 Functions함수와 Conditionals조건문

2.1 Functions함수

오늘날 프로그래밍의 기본 중 기본은 functions함수라고 말할 수 있습니다. 함수를 전혀 사용하지 않고 프로그래밍을 하는 것이 가능은 하지만, 함수를 쓰면 다양한 이점이 있습니다. 일단 프로그램을 역할에 따라 여러 부분으로 나누어 구조적 프로그래밍[2.1]이 가능해집니다. 또한 함수를 사용하면 같은 코드를 반복하지 않음으로써 프로그램의 용량을 줄이고 범용성을 늘릴 수 있습니다. 지난 시간에 자주 반복하여 사용하는 식을 하나의 변수에 대입하여 사용하면 효율적이라고 했던 것과 같은 맥락입니다. 나중에 객체 지향 프로그래밍 패러다임을 배우면 더 자세히 배우겠지만, 함수를 사용하여도 encapsulation캡슐화가 가능해집니다. 캡슐화란 함수를 한 번 구현해두면 내용을 모르더라도 그 기능을 사용할 수 있게 하는 것이라고 생각하면 됩니다.

한 번 Python에서 함수를 어떻게 정의하고 사용하는지, 아래의 예시를 통해서 살펴봅시다.

```
def gamma(v):
    c = 299792458
    b = v / c
    return 1 / (1 - b**2)**0.5

def t_rel(x, t, v):
    c = 299792458
    return gamma(v) * (t - v * x / c**2)

```

[1.15] 정답: `t + (x)tnuɹ + (t - (x)tnuɹ - x)tnuɹ`

[2.1] 구조적 프로그래밍은 비구조적 프로그래밍과 대비되는 프로그래밍 패러다임입니다. 비구조적 프로그래밍은 프로그램 전체를 하나의 연속된 코드로 작성하는 방식입니다. 이러한 프로그래밍 방식에서는 GOTO문과 같은 제어문에 의존할 수 밖에 없는데, 이는 디버깅이 어렵고 가독성이 떨어집니다. 특히 구조화되지 않은 코드는 프로그램이 복잡해지면 수행 방향이 복잡하게 얽히게 되는데, 이를 비유적으로 '스파게티 코드'라고 표현하기도 합니다. machine language기계어와 assembly language어셈블리어를 제외한 현대의 대부분 프로그래밍 언어는 구조적 프로그래밍을 지원합니다.

```
10 pos = 10
11 time = 13
12 vel = 3E7
13
14 print(gamma(vel))
15 print(t_rel(pos, time, vel))
```

일단 함수를 사용하기 위해서는 함수를 정의해야 합니다. 함수의 정의는 `def function(arg1, arg2, ...)`의 형태로 시작해서 `return value`으로 끝납니다. 이때 `function`은 원하는 함수명이고, `arg1`, `arg2`, $\cdots$ 은 함수의 인자에 해당하는 변수명입니다. 함수를 호출(사용)할 때에는 그냥 `function(a, b, c)`의 형태로 사용하시면 됩니다. 또한, 예시처럼 매개 변수의 수는 여러 개가 될 수 있습니다. 수학의 다변수 함수와 유사하게 받아들이면 됩니다. 본격적으로 함수를 어떻게 정의하고 사용해야 하는지 알아봅시다.

Indentation들여쓰기

Python에서는 indentation들여쓰기가 구문을 나누는 중요한 syntax입니다. C, C++ 등의 언어에서는 들여쓰기를 하지 않아도 중괄호`{·}`와 세미콜론 `;`으로 명령을 구분하고, 들여쓰기는 완전히 가독성만을 위한 것입니다. 그러나 중괄호나 세미콜론을 사용하지 않는 Python에서는 들여쓰기가 반드시 필요합니다. 띄여쓰기를 4회 하여 들여쓰기를 하던가, 탭을 통해서 들여쓰기를 할 수도 있습니다.[2.2] 띄어쓰기 4회를 통해 들여쓰기를 한다고 스페이스 키를 네 번 누를 필요는 없습니다. IDE나 텍스트 에디터 설정에서 탭 키를 soft tab (띄어쓰기 여러 개를 사용하여 들여쓰기를 하는 것)으로 설정하던지 hard tab (탭을 사용하여 들여쓰기를 하는 것)으로 설정하면 됩니다. Python의 코딩 스타일 가이드인 PEP-8에서는 soft tab이 권장됩니다.

위에서 언급하였듯이, Python에서 들여쓰기는 필수적인 문법입니다. 함수뿐만이 아니라 이번 시간에 배울 조건문, 나중에 배울 반복문 등에서 들여쓰기는 구문을 구별하고 포함 관계를 나타내는 역할을 합니다. 위의 예시처럼 함수에서는 `def` 다음 줄부터 `return`의 해당 줄까지 한 수준 들여쓰기를 해야 합니다. **들여쓰기를 실수로 의도한 바와 다르게 한다면 코드의 수행 결과가 완전히 달라질 수 있습니다. 나아가 문법적으로는 맞을 수 있어서 에러 메시지는 뜨지 않고 결과만 차이가 생기는 경우도 빈번하니 유의해야 합니다.** 따라서 들여쓰기를 명시적으로 나타내주는 IDE의 기능을 활용하면 생산성을 높일 수 있습니다. IDE의 기능에서 기본 중의 기본이니 대부분의 IDE에서 들여쓰기를 도와주는 기능이 있을 것입니다. Wing IDE에서는 Preferences > Editor > Indentation으로 들어가면 Show Indent Guides 항목에 체크를 하면 됩니다. PyCharm에서는 기본적으로 indentation guide를 보여줄 것입니다.

Built-in Functions내장 함수

사실 이번에 함수를 처음 접하는 것이 아닙니다. 지난 시간에도 여러 함수를 보았는데요, `type(·)`이나 형 변환을 위해 사용한 `int(·)` 등이 바로 함수입니다. 이러한 함수는 사용자가 직접 정의

[2.2] 띄어쓰기를 하여 들여쓰기를 할것인지, 탭을 하여 들여쓰기를 할 것인지는 프로그래머들 사이에서 펼쳐지는 오랜 논쟁입니다.

하지 않아도 사용할 수 있는 built-in functions내장 함수입니다. 또한 어떤 내장 함수들은 바로 사용할 수 있는 것이 아니라, '이러이러한 종류의 함수들을 사용할 것이다'라는 것을 코드에 작성하여야 쓸 수 있는 것들이 있습니다. 바로 `import package`을 (보통) 코드 맨 윗 줄에 적어주는 것입니다. 특히 수식 계산을 위해 자주 사용할 패키지는 `math` 패키지로, 제곱근 함수 `sqrt(·)` 이라든지 사인 함수 `sin(·)`와 같은 삼각 함수, 로그 함수 `log(·)`을 사용하기 위해서는 `math` 패키지를 import해야 합니다. 지난 단원의 마지막 문제인 올림 함수는 `math` 패키지의 `ceil(·)`[2.3] 함수를 사용하면 됩니다. 마찬가지로 `int(·)`와 같은 '가짜' 내림 함수가 아니라 진짜 내림 함수인 `floor(·)`가 제공됩니다. Python 2에서는 `math` 패키지에서 제공되던 반올림 함수`round(·)`는 이제 `math` 없이도 제공됩니다. 아래에서 사용 예시를 봅시다.

```
>>> import math
>>> math.sqrt(4)
2.0
>>> math.sin(30 * math.pi / 180)
0.49999999999999994
>>> math.log(math.e)
1.0
>>> math.ceil(-1.5)
-1
>>> math.floor(-1.5)
-2
>>> round(-1.5)
-2
```

만약 `math....(·)`를 반복하는 것이 귀찮다면, `from 패키지 명 import *`로 import를 하면 아래와 같이 계속 `math.`를 앞에 붙일 필요가 없어집니다.

```
>>> from math import *
>>> sqrt(4)
2.0
>>> sin(30. * pi / 180.)
0.49999999999999994
>>> log(e)
1.0
```

단, `sin` 등의 함수를 사용자가 새로 정의하거나, `pi`나 `e`와 같은 값을 새로 지정한다면 값이 덮어씌워져서 실수할 가능성이 커집니다.

[2.3] Ceiling function올림 함수의 앞에서 따온 명칭입니다.

```
>>> from math import *
>>> e
2.718281828459045
>>> e = 1
>>> e
1
```

또한 같은 함수 명 fn을 지닌 두 개의 패키지 pack1과 pack2에서 `from pack1 import *`를 한 후 `from pack2 import *`를 한다면 뒤에서 불러온 pack2의 fn이 먼저 불러온 pack1의 fn을 덮어씌울 것입니다. 예컨데 numpy라는 패키지를 컴퓨터에 설치한 후, `from numpy import *`와 `from math import *`를 한다면 상당수의 함수가 겹치게 되어 문제가 생길 가능성이 높습니다. 따라서 문제가 생기지 않을 것이라고 확신하거나 불가피할 경우에만 `from ...import *`를 사용하는 것이 권장됩니다. 나아가 자신이 math 패키지에서 sin(·) 함수만 필요하다는 경우에는 `from math import sin`을 하여 바로 math. 없이 sin 함수를 사용할 수 있습니다.[2.4]

User-Defined Functions사용자 정의 함수

본인이 원하는 함수가 없다면 직접 정의를 해서 함수를 만들 수 있습니다. 함수는 반환 값이 있는 것과 그렇지 않은 것 두 종류가 있습니다. 아래는 원의 반지름을 받아서 면적을 반환하는 함수 circ_area(·)입니다.

```
import math

def circ_area(radius):
    return math.pi * radius**2
```

return 다음에 반환할 값이 나타나 있습니다. 반면 아래는 원의 면적을 반환하지 않고 단순히 출력을 하는 함수 print_circ_area(·)입니다.

```
import math

def print_circ_area(radius):
    print(math.pi * radius**2)
```

circ_area(·)의 경우에는 원의 면적을 출력하기 위해 print를 사용해야 합니다. 그렇지만 다른 값을 계산하는 등의 작업에서 함수를 표현에 넣을 수 있습니다. print_circ_area(·)는 직접

[2.4] 보통 `import numpy as np`로 numpy 패키지를 불러옵니다. 이러한 경우, np.sin과 같은 형식으로 패키지를 사용할 수 있습니다.

적으로 원의 면적을 출력하지만, 그 값을 다른 표현에서 활용할 수는 없습니다. 반환하는 값이 없기 때문입니다.

함수를 정의하여 사용할 때에는, 사용하기 위해 호출하기 이전에 정의를 하기만 하면 됩니다. 위 조건만 만족한다면 다른 코드들 사이에 끼워두어도 되고, 함수들 정의 순서도 상관이 없습니다. 또한 함수를 정의하기 위해서 다른 함수를 사용할 수 있습니다.

```python
import math

def get_dist(x1, y1, x2, y2):
    dx = x1 - x2
    dy = y1 - y2
    return (dx**2 + dy**2)**0.5

def circ_area(x1, y1, x2, y2):
    r = get_dist(x1, y1, x2, y2)
    return math.pi * r**2

print(circ_area(0, 0, 3, 4))
```

위 코드의 `circ_area(·)` 함수를 정의하기 위해 `get_dist(·)` 함수를 사용하였는데, `get_dist(·)` 함수의 정의와 `circ_area(·)` 함수 정의의 순서는 바뀌어도 상관은 없습니다. 하지만 코드를 읽는 사람이 위부터 읽어 내려갈 때 모르는 함수가 있으면 코드를 이해하는데 어려움이 생길 수 있으니 가독성을 위해서 순서를 적당히 지켜주는 것이 좋을 것 같습니다.

코드의 뼈대 부분을 `main()` 등의 함수로 묶어두고, 함수 정의 밖에는 `main()` 한 번만 호출되게 할 수도 있습니다. 이는 C/C++, Java 등의 언어에서는 기본적으로 채택되는 방식입니다. 예컨대, C에서는 `Hello, world!`를 출력하는 코드가 다음과 같습니다.

```c
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

Python 코드에서는 함수 없이 가장 윗 수준에 적힌 코드가 실행되는 반면, C에서는 `main()` 함수에 적힌 내용이 실행됩니다. 하지만 Python에서도 다음과 같이 코드를 구성할 수 있습니다.

```python
def main():
    print("Hello, world!")
```

```
3
4 main()
```

방금 정의한 `main()` 함수는 매개 변수를 가지지 않는데, 이러한 종류의 함수는 매개 변수 없이 항상 동일한 기능을 하게 됩니다.

Parameters매개 변수와 Arguments전달 인자

함수를 정의할 때 사용되는 매개 변수들은 함수 정의 밖에서 호출될 수 없습니다. 일종의 블랙박스로 생각하면 되는데, 함수 안에서 정의되는 local variable지역 변수는 그 안에서만 사용을 할 수 있습니다. 예를 들어 예시에서 본 `get_dist(·)` 함수에서 `dx`나 `dy`는 함수 안에서만 생성되었다가 사라지는 변수라고 생각하면 됩니다. 나아가 함수 밖에서 `dx`라는 이름의 새로운 변수를 만들더라도 이는 함수 안에서 정의되었던 지역 변수 `dx`와는 다른 변수입니다. 반면, 함수 밖에서 먼저 정의가 되었으면서 함수 안에서 다시 정의되지 않은 변수는 global variable전역 변수라고 합니다. 이러한 경우에는 함수 밖에서 정의가 된 값이 그대로 함수 안에서 사용됩니다. 또한 함수의 매개 변수도 함수 밖에서 사용될 수 없는 지역 변수입니다. 매개 변수의 이름과 함수의 밖에서 정의된 어떤 변수의 이름이 같더라도 이 둘은 관련이 없습니다. 물론, 해당 변수에 지정된 값이 같은 이름의 매개 변수로 정의된 함수의 arguments전달 인자로 지정될 수는 있습니다. 매개 변수는 항상 전달 인자로 값이 지정되기 때문에 전역 변수와 이름이 겹친다고 해도 영향을 받지 않습니다. 여기서 전달 인자는 `fn(3)`의 3처럼 단순히 함수에 전달되는 값입니다. 아래의 예시를 통해 좀 더 명확히 이해를 해보시길 바랍니다. 이해를 위해 같은 이름을 가졌더라도 가리키는 값이 다르면 다른 색상으로 표시하였습니다.

```python
def fn1(a, b):
    c = a + b
    d = a - b
    return (c + d) / 2.

a, b = 1, 2
print(fn1(a, b))
print(fn1(1, 2))

c, d = 3, 4

def fn2(a, b):
    d = a - b
    return (c + d) / 2.

print(fn2(c, d))
print(fn2(3, 4))
```

정리하자면, fn1(·)의 매개 변수 a와 b는 아래에서 전달 인자로 사용된 a = 1 및 b = 2와 무관합니다. 줄 7과 줄 8의 출력 값이 같다는 사실을 통해, 줄 7의 fn1(a, b)는 단지 밖에서 정의된 a의 값인 1과 b의 값인 2를 대입한 fn1(1, 2)와 완전히 동일한 역할을 한다는 것을 확인할 수 있습니다. fn2(·)의 경우는 약간 다릅니다. 함수 내부에서 c가 지역 변수로 새로 정의되지 않은 상태로, 줄 14에서 c를 사용하였습니다. 여기에서 쓰인 c는 c = 3로 정의된 전역 변수입니다. 반면 fn2(·) 내부에서 사용된 d는 전역 변수 d가 아니라 새로 a - b의 값을 가지는 지역 변수입니다. 이와 같이 어떤 변수가 유효성을 가지는 영역을 scope변수 영역이라고 합니다. 마지막 예시를 통해 변수 영역의 의미를 확실히 파악할 수 있을 것입니다.

```
1  def mult(a, b):
2      x = a * b
3      return x
4
5  def div(a, b):
6      x = a / float(b)
7      return x
8
9  a, b = 1, 2
10 print(mult(b, a))  # mult(2, 1)
11 print(div(a, b))  # div(1, 2)
12 print(x)  # NameError: name 'x' is not defined
```

함수 mult(·)와 div(·), 그리고 줄 9 이후의 a, b, x는 서로 무관하며, 줄 12에서는 함수 mult(·)와 div(·)의 변수 영역 밖에서 x가 정의된 적이 없으므로 오류가 발생합니다.

2.2 Conditionals조건문

그렇다면, 입력값(전달 인자)에 따라서 반환값이 달라지는 함수를 만들 때에는 어떻게 해야 할까요? 단순히, 절댓값 함수를 구현하려고 해도 다음과 같이 경우가 갈리게 됩니다.

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

이러한 함수를 구현하고 싶을 때 활용해야 하는 것이 바로 conditionals조건문입니다. 내장 함수인 abs(·)를 사용할 수도 있지만, 절댓값 함수는 다음과 같이 구현할 수 있습니다.

```
1  def abs(x):
2      if x >= 0:
3          return x
4      else:
5          return -x
```

위의 조각적-정의된 (piecewise-defined) 함수의 예시 같이, 수학에서 정의역에 따라 다르게 정의되는 수식을 코드로 작성하기 위해서는 조건문이 필요합니다. 하지만 조건문은 이와 같이 수식을 계산하는 일을 넘어, 경우를 나눠야 하는 모든 종류의 상황에 적용됩니다. 예를 들어, 어떤 프로그램에서 사용자가 로그인하였을 때와 그렇지 않았을 때 보여줘야 하는 항목이 다를 경우에 조건문을 사용해야 합니다. 낮과 밤에 따라서 화면의 밝기를 조절하는 툴을 작성할 때에도 조건문을 쓸 수 있습니다. 어떤 로봇이 장애물을 맞닥뜨렸을 때 해야 하는 행동과 앞이 트여 있을 때 해야 하는 행동을 조절하기 위해서 조건문이 필요합니다. 이처럼, 컴퓨터가 상황에 따라 다른 연산을 하도록 하는 일에 조건문을 사용할 수 있습니다.

```
1 def check_obstacle(distance):
2     if distance < 10:
3         print("앞에 장애물이 있습니다!")
4     else:
5         print("앞에 장애물이 없습니다.")
```

이 예시에서 `distance < 10`의 부분은 크게 설명하지 않아도 어떤 의미인지 파악하실 수 있을 것입니다. 이렇게 `if` 뒤에 오는 부분은 `True` 혹은 `False` 두 종류의 값을 가질 수 있는데, 이는 이제 소개해드릴 **Boolean불리언**이라는 자료형을 가집니다.

Boolean Type불리언 자료형과 Expressions표현

지금까지 소개한 `int`, `float` 등의 자료형 외에도 자주 쓰게 될 자료형은 Boolean type불리언 자료형[2.5]입니다. 불리언 자료형은 어떤 표현의 참/거짓을 나타내는 형으로, `True`와 `False` 두 값을 가집니다. 위의 예시에서 본 표현인 `x >= 0`은 `True`일수도 `False`일수도 있는 불리언 자료형입니다. 불리언 표현을 구성하기 위해서는 대소 관계와 일치 여부 등을 판단하는 관계 연산자와 `and`, `or`, `not` 등의 논리 연산자로 구성될 수 있습니다. `x >= 0 or (y < 0 and z < 0)`이 불리언 자료형을 가지는 표현의 예시입니다.

관계 연산자는 다음의 여섯 가지 종류가 있습니다.

- `x == y`: x와 y가 일치하면 ($=$) `True`
- `x != y`: x와 y가 일치하지 않으면 ($\neq$) `True`
- `x > y`: x가 y 초과이면 ($>$) `True`
- `x < y`: x가 y 미만이면 ($<$) `True`
- `x >= y`: x가 y 이상이면 ($\geq$) `True`
- `x <= y`: x가 y 이하면 ($\leq$) `True`

[2.5] 영국의 수학자이자 논리학자인 George Boole의 이름을 따온 것입니다.

지금까지 어떤 변수 x에 값을 대입할 때에는 =를 사용하였습니다. `x = 10`과 같이 말입니다. 수학에서도 "let $x = 10$"처럼 $=$를 사용하는 것과 같은 맥락입니다. 나아가 수학에서는 보통 두 값을 비교할 때, "if $x = 10$, "과 같은 표현을 사용하는데, Python에서는 두 값을 비교할 때 단순히 등호 하나가 아니라 두 개를 연달아 쓴 ==를 사용합니다. `if x == 10`과 같이 말입니다.

논리 연산자는 다음 표의 세 가지 종류가 있습니다.

p	q	p and q	p or q	not p
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

위 표에는 각 표현에 대해서 어떤 불리언 값을 가지는지 진리표가 제시되어 있습니다. and, or, not은 각각 수학의 $\wedge$, $\vee$, 그리고 $\neg$에 대응됩니다.

이제 여러가지 관계 연산자와 논리 연산자에 대해서 알아보았으니, 기존에 알았던 연산의 순서에 새로 배운 연산자들을 넣어봅시다. 연산의 순서는 우선 순위에 따라 다음과 같이 나열할 수 있습니다.

1. (·)
2. **
3. *, /, %
4. +, -
5. ==, !=, >, <, >=, <=
6. not
7. and
8. or
9. =

모든 관계 연산자들은 논리 연산자들보다는 순위가 높고, 산술 연산자들보다는 낮습니다. 이전과 같이 우선 순위가 겹친다면 좌에서 우로 순서가 결정됩니다.

지금까지 불리언 자료형에 대한 논리 연산자와 그 우선순위를 살펴보았는데요, 이제 조건문과 함께 사용해서 그 사용 예시를 더 자세히 살펴봅시다.

if-else Conditionals조건문

if-else statements문의 기본적인 형태는 아래와 같습니다.

```
1 if expression:
2     ...
```

```
3 else:
4     ...
```

위에서 expression에는 x%2 == 0 or x%3 == 0 등의 임의의 불리언 표현이 올 수 있습니다. 만약 제시된 불리언 표현이 True라면 줄 2의 내용이 실행되고, 그렇지 않다면 줄 4의 내용이 실행됩니다. 제시된 불리언 표현이 False일 때 실행할 내용이 없다면 else 부분은 빠뜨릴 수 있습니다. 다음과 같은 예시를 통해 확인해 봅시다.

```
1 x = 76
2 if x % 3 == 0 and x % 2 == 0:
3     print(x, "은(는) 6의 배수입니다.")
4 else:
5     print(x, "은(는) 6의 배수가 아닙니다.")
```

실제로 실행해보면 76은 6의 배수가 아니기에 76은(는) 6의 배수가 아닙니다.가 출력되는 것을 볼 수 있습니다.

만약 6의 배수가 아니라면, 적어도 3의 배수인지, 혹은 2의 배수인지까지 판별하고 싶다면 어떻게 해야 할까요? 일단 배운 것만으로는 다음과 같이 할 수 있을 것입니다.

```
1 x = 76
2 if x % 3 == 0 and x % 2 == 0:
3     print(x, "is divisible by 6")
4 else:
5     if x % 3 == 0:
6         print(x, "is divisible by 3")
7         else:
8             if x % 2 == 0:
9                 print(x, "is divisible by 2")
10            else:
11                print(x, "is neither divisible by 3 nor 2")
```

6의 배수인지 체크한 후, 그렇지 않다면 다시 3의 배수인지 체크한 후, 그렇지 않다면 2의 배수인지 체크하는 코드입니다. 분명 작동은 잘 하겠지만, 가독성도 좋지 않고 깔끔하지도 않습니다. 이런 상황에서 쓸 수 있는 것이 elif[2.6]입니다. 세 가지 이상의 경우가 있을 때, if로 시작하여 두 번째부터 마지막 바로 직전 경우까지는 elif를, 마지막 예외의 경우에는 else로 마무리하는 방식입니다. 마찬가지로 모든 경우에 해당하지 않을 때 시행할 것이 없으면 마지막 else는 생략할 수 있습니다. elif를 활용하면 위의 코드를 아래와 같이 간결하게 나타낼 수 있습니다.

[2.6] else if에서 else의 el만 따온 것입니다. C 언어에서는 그대로 else if를 사용합니다.

```
x = 76
if x % 3 == 0 and x % 2 == 0:
    print(x, "is divisible by 6")
elif x % 3 == 0:
    print(x, "is divisible by 3")
elif x % 2 == 0:
    print(x, "is divisible by 2")
else:
    print(x, "is neither divisible by 3 nor 2")
```

만약 `if-elif-...-elif-else`를 훑는 중간에 참인 것이 나오면, 해당 조건에서 실행되는 명령을 수행한 후 이후의 경우는 모두 건너뛰게 됩니다.

참고로, `if-else` 문에서 불리언 표현 대신 `int` 등의 자료형을 가진 변수를 넣어도 됩니다. 예컨대 `0`은 거짓, `0` 이외의 모든 수는 참의 값을 가집니다. 그렇다고 이러한 값들이 정확히 `True` 혹은 `False`의 값을 가지는 것은 아닙니다. (실제로 `0 == False`, `1 == True`이지만) `2 == True`에 대한 결과값은 `False`입니다. 또한 문자열의 경우 `""` 외의 하나 이상의 문자를 담고 있는 문자열은 참입니다. `""`는 거짓에 해당합니다. 이는 Python에서 `if` 문을 조금 더 간편하게 쓸 수 있도록 하지만, 특별한 이유가 있지 않는 이상 불리언 표현을 사용하는 것이 혼란을 줄일 수 있습니다.

사실 `bool`형은 `int`형의 부분으로, `True`는 정확히 `1`과, `False`는 정확히 `0`과 동일합니다.[2.7] 따라서, 다음과 같은 결과가 나옵니다.

```
>>> True * 3
3
>>> True * False
0
>>> (6 % 2 == 0) + False
1
```

어디까지나 **가능하다는** 사실을 알려주는 코드이고, 실제로는 이렇게 쓸 이유가 없습니다.

그러나 논의된 내용을 바탕으로 다음과 같은 (유의미한) 코드를 쓸 수는 있습니다.

```
money = 100
if money:
    print("I have money")
```

[2.7] Python 2에서는 `True`와 `False`가 키워드로 지정이 되어 있지 않아 일반 변수명으로 사용하여 아무 값으로나 재지정할 수 있었습니다. 반면 Python 3에서는 `True`와 `False` 모두 키워드라 값을 지정하려고 하면 syntax 에러가 발생하게 됩니다.

```
4 else:
5     print("I don't have money")
```

Multiple Exit Points 다중 반환점

한 함수에 `if-else` 문을 넣어 경우별로 값을 반환하도록 만들 수 있습니다. 상기하자면, 반환한다는 것은 함수에서 값을 뱉어내는 것이었습니다. 즉, `if-else` 문을 사용해 함수 정의의 여러 곳에서 값을 뱉어낼 수 있습니다. **2.2 Conditionals조건문** 처음에 제시한 `abs(·)` 함수가 그 예시로, `x >= 0`일 경우 `x`를 반환하고 그렇지 않은 경우 `-x`를 반환하였습니다. 이러한 함수에는 여러 개의 `return`을 사용하였기 때문에 multiple exit points다중 반환점이 있다고 표현하기도 합니다.[2.8] 만약 `abs(·)` 함수를 하나의 반환점만을 사용한다면 아래와 같게 나타낼 수 있습니다.

```
1 def abs(x):
2     if x >= 0:
3         y = x
4     else:
5         y = -x
6     return y
```

2.2절 처음에 제시한 코드와 위 코드는 정확하게 같은 방식으로 작동합니다. 또한 2.2절 처음에서 제시한 코드를 변형하여 `else` 없이 아래와 같이 다시 쓸 수 있습니다.

```
1 def abs(x):
2     if x >= 0:
3         return x
4     return -x
```

위 코드의 경우에는 가능한 경우를 모두 걸러서 마지막 경우에는 `-x`를 반환하라는 코드로 읽을 수 있습니다. 함수를 읽어나갈 때, `return`을 만나는 순간 함수가 해당 값을 반환하고 종료되기 때문에 위와 같은 코드를 작성할 수 있습니다. 즉, **`return`은 함수에서 값을 반환하는 동시에, 함수의 수행을 종료하라는 마크로 볼 수 있습니다.**

다중 반환점을 사용할 경우, 일반적으로는 다음과 같이 함수를 만들 수 있습니다.

```
1 def abs(x):
2     if ...:
3         return ...
4     if ...:
```

[2.8] 함수에 여러 개의 반환점이 있으면 경우에 따라 실수를 하여 예기치 못한 버그가 생길 수 있기 때문에, 다중 반환점을 쓰는 것이 바람직한가에 대한 논쟁이 있습니다. 하지만, 적절한 상황에서 사용한다면 분명 편리할 것입니다.

```
        return ...
    ...
    if ...:
        return ...
    return ...
```

2.3 예제

이하 예제에서 [물리] 혹은 [수학]과 같이 표시된 문제는 물리학 혹은 수학에서 등장하는 복잡한 연산을 코딩으로 해결할 수 있다는 것을 보여주기 위한 것으로, 선택적으로 해결하시면 됩니다.

1. 이름 `name`을 문자열로 받아서 `제 이름은 {name}입니다.`와 같이 출력하는 함수 `introduce(name)`를 작성하세요.

```
def introduce(name):
    # Add here!

print(introduce("귀도 반 로섬"))  # 제 이름은 귀도 반 로섬입니다.
print(introduce("앨런 튜링"))     # 제 이름은 앨런 튜링입니다.
print(introduce("폰 노이만"))     # 제 이름은 폰 노이만입니다.
```

2. 두 수 `a`와 `b`를 받아 덧셈, 뺄셈, 곱셈, 나눗셈을 해주는 함수 `add(a, b)`, `subtract(a, b)`, `multiply(a, b)`, `divide(a, b)`를 각각 작성하세요.

```
def add(a, b):
    # Add here!

def subtract(a, b):
    # Add here!

def multiply(a, b):
    # Add here!

def divide(a, b):
    # Add here!


print(add(1, 2))       # 3
print(subtract(1, 2))  # -1
```

```python
print(multiply(1, 2))  # 2
print(divide(1, 2))    # 0.5
```

3. [수학] 정규분포를 표현하기 위해 사용되는 가우시안 함수는

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2},$$

이며, 이때 μ는 기댓값, σ는 표준편차입니다. 가우시안 함수 `gaussian(mu, sigma, x)`를 작성하세요.

```python
import math

def gaussian(mu, sigma, x):
    # Add here!

print(gaussian(0, 1, 0))                # 0.3989422804014327
print(gaussian(-2, math.sqrt(0.5), 0))  # 0.010333492677046037
```

4. [물리] 전하가 각각 q_1과 q_2인 두 물체 사이의 쿨롱힘은 거리 r에 따른 함수

$$F_C(r) = \frac{1}{4\pi\varepsilon_0}\frac{q_1 q_2}{r^2} \text{ where } \varepsilon_0 = 8.854\,187\,817 \times 10^{-12}\ \mathrm{F}\cdot\mathrm{m}^{-1}.$$

로 쓸 수 있습니다. 또한, 질량이 각각 m_1, m_2인 두 물체 사이의 만유인력은 거리 r에 따른 함수

$$F_g(r) = G\frac{m_1 m_2}{r^2} \text{ where } G = 6.674\,08 \times 10^{-11}\ \mathrm{m}^3\cdot\mathrm{kg}^{-1}\mathrm{s}^{-2}.$$

로 표현됩니다. 두 물체가 전하 `q1`과 `q2`, 그리고 질량 `m1`과 `m2`를 가지고, 거리 `r`만큼 떨어져 있다고 가정합니다. 각각 쿨롱힘과 만유인력을 구하여 반환하는 함수 `coulomb(q1, q2, r)`과 `gravity(m1, m2, r)`을 작성하세요. 또한 이 둘 모두를 구하여 더하는 `total_force(q1, q2, m1, m2, r)`도 작성하세요.

```python
import math

def coulomb(q1, q2, r):
    # Add here!

def gravity(m1, m2, r):
    # Add here!

def total_force(q1, q2, m1, m2, r):
```

```
    # Add here!

e_c = -1.6021766208e-19
e_m = 9.10938356e-31
p_c = -e_c
p_m = 1.672621898e-27
a_0 = 5.2917721067e-11

print(coulomb(e_c, p_c, a_0))
print(gravity(e_m, p_m, a_0))
print(total_force(e_c, p_c, e_m, p_m, a_0))
```

5. [수학] $a \neq 0$일 경우, 함수 $f(x) = ax^2 + bx + c$는 극값을 가진다. 함수 $f(x) = \texttt{a}x^2 + \texttt{b}x + c$은 극값을 반환하는 `extremum(a, b, c)`를 작성하세요.

```
def f(a, b, c, x):
    return a*x**2 + b*x + c

def extremum(a, b, c):
    # Add here!

print(extremum(1, 5, 10))   # 3.75
print(extremum(1, -5, 10))  # 3.75
print(extremum(3, 7, 5))    # 0.916666666667
```

6. 세 수 `a`, `b`, `c` 중에서 가장 작은 수를 반환하는 `least(a, b, c)`를 작성하세요.

```
# Add here!

print(least(1, 2, 3))  # 1
print(least(2, 1, 3))  # 1
print(least(1, 1, 2))  # 1
```

7. 입력 받은 수 `n`이 짝수이면 `{n}은 짝수입니다.`, 홀수이면 `{n}은 홀수입니다.`를 출력하는 함수 `check_parity(n)`을 작성하세요.

```
# Add here!

```

```
check_parity(2)    # 2는 짝수입니다.
check_parity(101)  # 101은 홀수입니다.
```

8. 입력 받은 문자 `direction`이 `w`일 경우 전진, `a`일 경우 좌회전, `s`일 경우 후진, `d`일 경우 우회전을 출력하는 함수 `navigate(direction)`을 작성하세요. 단, 이외의 문자가 입력되었을 경우에는 `알 수 없는 명령입니다.`를

```
# Add here!

navigate('w')  # 전진
navigate('a')  # 좌회전
navigate('s')  # 후진
navigate('d')  # 우회전
navigate('z')  # 알 수 없는 명령입니다.
```

9. 다음을 반환하는 함수 `quadrant(x, y)`를 작성하세요:

   - `"제1사분면"` if $x > 0$ & $y > 0$
   - `"제2사분면"` if $x < 0$ & $y > 0$
   - `"제3사분면"` if $x < 0$ & $y < 0$
   - `"제4사분면"` if $x > 0$ & $y < 0$
   - `"경계선"` otherwise

```
# Add here!

print(quadrant(10, 5))   # 제1사분면
print(quadrant(-5, 3))   # 제2사분면
print(quadrant(-5, -7))  # 제3사분면
print(quadrant(3, -5))   # 제4사분면
print(quadrant(0, -3))   # 경계선
```

10. 10000보다 작은 자연수 `n`에 대해서, 각 자릿수를 거꾸로 출력하는 함수 `reverse(n)`를 작성하세요.

```
# Add here!

print(reverse(3702))  # 2073
print(reverse(370))   # 73
```

```
print(reverse(37))    # 73
print(reverse(3))     # 3
```

11. 10 이상 990 이하의 10의 배수 `n`이 주어졌을 때, 10, 50, 100, 500원 동전을 최소로 사용해서 `n`원을 만드는데 사용되는 동전의 수를 반환하는 함수 `count_coins(n)`을 작성하세요.

```
# Add here!

print(count_coins(730))  # 6
print(count_coins(790))  # 8
print(count_coins(260))  # 4
print(count_coins(70))   # 3
```

12. [수학] 함수 `ceiling(x)`이 $\lceil x \rceil$을 반환하도록 작성하세요.

```
# Add here!

print(ceil(4.3))   # 5
print(ceil(-0.3))  # 0
print(ceil(0.01))  # 1
```

13. [수학] Write a function `min_x(a, b, c)` that returns the integer x that minimizes $ax^2 + bx + c$. Assume that `a`, `b`, and `c` are `float` type, where $a > 0$ and $b < 0$. If such x is not unique, return the smallest one.

```
def f(a, b, c, x):
    return a*x**2 + b*x + c

def min_x(a, b, c):
    # Add here!

print(min_x(1, -9, 2))    # 4
print(min_x(9, -5, 0))    # 0
print(min_x(9, -15, 0))   # 1
print(min_x(7, -13, 3))   # 1
```

14. [수학] Write a function `area_triangle(a, b)` that returns the area of a triangle formed by $y = ax + b$, x-axis, and y-axis. Return 0 if no triangle is formed. Assume `a` and `b` are either `int` or `float` type.

```
def area_triangle(a, b):
    # Add here!
```

심화 "S", "R", and "P" represent scissors, rock, and paper, respectively. Write a function rock_paper_scissors(a, b) that returns "a" if a wins, "b" if b wins, or Tie if the game is tied, where a and b are elements of {"S", "R", "P"}.

```
def rock_paper_scissors(a, b):
    if a == b:
        return "Tie"
    # Add here!

print(rock_paper_scissors("R","R"))  # Tie
print(rock_paper_scissors("R","S"))  # a
print(rock_paper_scissors("R","P"))  # b
print(rock_paper_scissors("S","S"))  # Tie
print(rock_paper_scissors("S","P"))  # a
print(rock_paper_scissors("S","R"))  # b
print(rock_paper_scissors("P","P"))  # Tie
print(rock_paper_scissors("P","R"))  # a
print(rock_paper_scissors("P","S"))  # b
```

도전 gold1, silver1, and bronze1 represent the numbers of gold, silver, and bronze medals of the first country, respectively. gold2, silver2, and bronze2 represent the numbers of gold, silver, and bronze medals of the second country, respectively. Write a function better(gold1, silver1, bronze1, gold2, silver2, bronze2) that returns "First" if the first country achieved better than the second, "Second" if the second country achieved better, or "Tie" if they tied. The score is evaluated according to the gold-silver-bronze order, not by the sum of total medals.

```
def better(gold1, silver1, bronze1, gold2, silver2, bronze2):
    if gold1 > gold2:
        return "First"
    # Add here!

print(better(10,4,24, 1,35,25))  # First
print(better(1,35,25, 10,4,24))  # Second
print(better(10,18,0, 10,4,24))  # First
print(better(10,4,24, 10,18,0))  # Second
```

```
print(better(10,20,5, 10,20,4))  # First
print(better(10,20,4, 10,20,5))  # Second
print(better(10,20,5, 10,20,5))  # Tie
```

3 Boolean Functions불리언 함수와 Loops반복문 기본

3.1 Boolean Functions불리언 함수

지난 시간에는 조건문을 배우면서 불리언 자료형에 대해 알아보았습니다. 그렇다면 불리언 자료형인 `True`와 `False`를 반환하는 Boolean functions불리언 함수도 생각해볼 수 있습니다. 이미 함수와 조건문을 배웠기 때문에, 이번 절의 내용은 매우 간단합니다. 지난 시간의 복습이라고 보아도 무방합니다. 이번 절에서는 불리언 함수를 정의하고 사용할 때 무엇을 주의해야 하는지에 대해 간략히 알아봅니다.

Boolean Functions불리언 함수의 예시

불리언 함수는 불리언 자료형 `True`와 `False`를 반환하는 함수입니다. 불리언 함수는 길고 복잡한 조건문을 간략하게 표현할 때 유용합니다. 아래는 길이 `a`, `b`, `c`를 가지는 선분이 삼각형을 이룰 수 있는지 판별하는 조건문입니다.

```
if a < b + c and b < c + a and c < a + b:
    ...
```

이는 아래와 같은 함수로 대신할 수 있습니다.

```
def triangle_formed(a, b, c):
    if not a < b + c:  # if a >= b + c:
        return False
    if not b < c + a:
        return False
    if not c < a + b:
        return False
    return True

if triangle_formed(a, b, c):
    ...
```

복잡한 조건문의 경우에는 불리언 함수의 유용성이 명백해집니다. "두 수 n_1과 n_2가 서로소이라면" 등의 표현을 한 줄로 표현하는 것보다는, 두 수가 서로소이면 `True`를 아니면 `False`를 반환하는 함수를 정의한 뒤–예컨대 `coprime(n1, n2)`–`if coprime(n1, n2):`와 같이 정리하는

것이 바람직합니다. 나아가, 이러한 불리언 함수는 조건 확인의 재사용성을 높여줍니다. 예를 들어, 로봇이 거리를 체크한 후 장애물이 앞에 있는지 확인하여 불리언을 반환하는 함수를 생각해봅시다.

```
def check_obstacle(distance):
    if distance < 10:
        return True
    return False
```

언뜻 보면 거리를 확인해야할 곳에 `distance < 10`을 써넣는 것이 굳이 함수를 작성하는 것보다 간단해 보일 수도 있지만, 로봇을 조종하는 코드에서 장애물 확인은 자주 발생할 것입니다. 그런데 만약 장애물이 있다고 인식하는 거리를 10에서 5로 줄이고 싶다고 합시다. 만약 `distance < 10`로 거리를 확인했다면, 코드의 수십, 아니 수백 곳에서 10을 5로 바꿔줘야 할 것입니다. 반면 함수를 정의한 후 `check_obstacle(distance)`와 같이 사용했다면 단 한 곳만 수정하면 될 것입니다.

유의 사항

불리언 `True`는 문자열 `"True"`가 아닙니다. 마찬가지로 불리언 `False`는 문자열 `"False"`가 아닙니다.

```
>>> type(True)
<type 'bool'>
>>> type("True")
<type 'str'>
>>> type(False)
<type 'bool'>
>>> type("False")
<type 'str'>
```

`True`는 문자열이 아니라 그 자체로 불리언 자료형을 지닌 값입니다.[3.1] 이를 강조하는 이유는, 간혹 `True`를 반환해야할 곳에 실수로 `"True"`라고 적는 경우가 있기 때문입니다. 이 둘은 서로 다른 값입니다.

조건문은 그 자체로 불리언 자료형을 가지기 때문에, `if 조건문: return True(False)`와 같은 형태는 단순히 `return 조건문`으로 줄여 쓸 수 있습니다. 아래는 $ax^2+bx+c=0$의 실근이 존재하는지를 판별하는 함수 `real_solution(a, b, c)`입니다.

```
def real_solution(a, b, c):
    if b**2 - 4*a*c >= 0:
```

[3.1] 물론, 지난 시간에 언급했듯이 `True`는 1, `False`는 0입니다.

```
        return True
    return False
```

이는 간략히

```
def real_solution(a, b, c):
    return b**2 - 4*a*c >= 0
```

으로 나타낼 수 있습니다. 반대로 `if 조건문: return False`는 `return not 조건문`과 같이 정리가 됩니다.

나아가 이런 불리언 함수를 조건문에 사용할 때, `if 불리언_함수 == True:` 대신 `if 불리언_함수:`를 쓰는 것이 바람직합니다. 즉,

```
if real_solution(a, b, c) == True:
    ...
```

보다는

```
if real_solution(a, b, c):
    ...
```

이 더 간결한 표현입니다. 특히 `if 불리언_함수 == True:`을 쓰려다가 `==` 대신 `=`을 사용하는 실수를 미연에 방지할 수 있습니다.

그리고 복잡한 표현은 끊어서 표현하는 것이 미연의 실수를 방지하고 코드의 가독성을 높입니다. 다음 코드

```
if ((x1 - x2)**2 + (y1 - y2)**2)**.5 <= ((x2 - x3)**2 + (y2 - y3)**2)**.5 +
↪  ((x3 - x1)**2 + (y3 - y1)**2)**.5:
    ...
```

보다는

```
a = ((x1 - x2)**2 + (y1 - y2)**2)**.5
b = ((x2 - x3)**2 + (y2 - y3)**2)**.5
c = ((x3 - x1)**2 + (y3 - y1)**2)**.5
if a < b + c:
    ...
```

이 훨씬 읽기 편한 것을 볼 수 있습니다.

마지막으로, **float형 수끼리는 등호 비교 연산==을 하면 안 됩니다.**

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
```

이러한 결과는 컴퓨터의 부동 소수점 처리 방식 때문인데, 부동 소수점 rounding error반올림 오차라고 합니다. 간단히는 이진수 소수로 십진수 소수를 정확하게 나타낼 수 없어서, 오차가 누적되어 생기는 오차라고 생각하시면 됩니다.[3.2] 따라서 `float`끼리의 비교는 `>`와 `<`만 신뢰할 수 있으며, 되도록이면 `int` 상태에서 비교를 하는 것이 좋습니다. 예를 들어 두 정수 격자점 사이의 거리를 비교할 때는 제곱근을 씌우지 않은 상태에서 비교를 해야 정확합니다.

3.2 Loops반복문 기본

컴퓨터는 인간보다 단순 반복 계산을 빠르게 수행할 수 있습니다. 이번 절에서 배울 loops반복문을 익히면 손으로는 할 수 없는 많은 계산을 Python으로 수행할 수 있을 것입니다. 나아가 **지금까지 배운 함수, *if-else* 문과 같이 사용한다면, 원하는 작업을 하는 코드를 설계하고 작성할 수 있을 것입니다.**

간단한 예시로는 로봇에게 반복적인 작업을 수행하도록 반복문을 통해 작성할 수 있습니다. 로봇이 특정 거리만큼 앞으로 가게 하는 함수 `move_forward(distance)`, 특정 각도만큼 우회전하는 함수 `turn_right(angle)`가 주어져 있다고 합시다. 이때 정사각형으로 한 바퀴를 돌게 하는 코드는 Python의 반복문을 사용해 다음과 같이 쓸 수 있습니다.

```
1 for i in range(4):
2     move_forward(1)
3     turn_right(90)
```

줄 2와 줄 3의 내용을 총 4번 반복하도록 하는 `for` 반복문으로, 어떤 일을 하는 코드인지 어렵지 않게 이해할 수 있습니다.

첫 번째 수업에서 1부터 n까지의 수를 더하는 문제를 풀 때에는 공식 $1+2+\cdots+n=\frac{n(n+1)}{2}$을 사용하였습니다. 그렇다면 $1^m+2^m+\cdots+n^m$은 어떻게 계산할 수 있을까요? 임의의 m에 대해서 공식을 유도할 수는 없습니다. 이런 경우에 `for` 반복문을 활용할 수 있습니다. 지금까지는 'n회 무슨 작업을 반복하라'의 명령을 하려면 직접 n회 칠 수 밖에 없었지만, `for` 반복문을 사용하면 임의의 n에 대해서 단 한 번의 일반적인 규칙을 제시하면 명령을 수행할 수 있습니다. $\sum_{i=1}^{n} i^m$을 반환하는 함수는 아래와 같습니다.

[3.2] 자세한 내용은 https://en.wikipedia.org/wiki/IEEE_754와 https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html에 자세히 나와 있습니다.

```
1 def summ(m, n):
2     summ = 0
3     for i in range(1, n + 1):
4         summ += i**m
5     return summ
```

줄 3에서 i를 1 이상 n + 1 미만까지 하나씩 증가하며 밑의 줄 4를 반복한다는 뜻입니다. 이처럼 for 반복문의 기본은 range(·)이라고 볼 수 있습니다. 이번 절에서는 range(·) 함수를 활용하여, 조건문을 활용한 반복문이나 nested loops중첩 반복문에 대해 자세히 알아봅니다.

range(·) 함수를 활용한 for 반복문

range(·) 함수[3.3]는 인자를 한 개에서 세 개까지 넣을 수 있습니다. 일단 하나를 넣는 경우를 봅니다.

```
1 for i in range(n):
2     print(i)
```

이는

```
1 0
2 1
3 2
4 ...
5 n - 1
```

를 출력합니다. 즉, range(n)은 0에서 n이 되기 직전까지 i를 1씩 증가시키며 print(i)을 실행합니다. 참고로, 여기서 함수 밖에서 for 반복문이 정의되었다면 i는 전역 변수이므로, 다시 정의하지 않으면 n - 1로 남아있게 됩니다.

range(·)에 두 개의 인자가 전달되는 경우가 아래에 나타나 있습니다.

```
1 for i in range(m, n):
2     print(i)
```

이는

[3.3] range(·)는 리스트를 반환하는 함수로, 리스트에 대해서는 다음 시간에 자세히 알아볼 것입니다.

```
m
m + 1
m + 2
...
n - 1
```

와 동일합니다. `range(m, n)`은 `m`에서 `n`이 되기 직전까지 `i`를 1씩 증가시키며 `print(i)`를 실행하는데, $m \geq n$이라면 반복문이 수행되지 않습니다.

마지막으로 세 개의 인자가 `range(·)`에 전달되는 경우를 살펴봅니다.

```
for i in range(m, n, k):
    print(i)
```

이는

```
m
m + k
m + 2*k
...
m + l*k
```

와 같습니다. 위에서 `m + l*k`는 `n`이 되기 직전의 값입니다. `range(m, n, k)`은 `m`에서 `n`이 되기 직전까지 `i`를 `k`씩 증가시키며 `print(i)`를 실행하는 것입니다. `k`는 음수가 될 수도 있습니다. 이 경우에는 `i`가 `m`에서부터 `n`이 되기 직전까지 감소됩니다.

정리하자면 `range(n)`은 `range(0, n)`, `range(0, n, 1)`과 같고, `range(m, n)`은 `range(m, n, 1)`과 같습니다. 아래는 인자를 넣는 세가지 경우를 모두 나타낸 예시입니다.

```
for i in range(n):
    print(i, end=' ')  # 0 1 ... n - 1

for i in range(1, n + 1):
    print(i, end=' ')  # 1 2 ... n

for i in range(10, 0, -1):
    print(i, end=' ')  # 10 9 ... 1

for i in range(1, 10, 2):
    print(i, end=' ')  # 1 3 ... 9
```

위 예시에서 `print()`안에 `end=' '`를 써준 것은 줄바꿈을 하지 않고 대신 띄어쓰기를 하기 위함입니다. Python의 내장 `print()` 함수에서는 `end`에 값을 넣어 값을 출력한 후 뒤에 붙일 문자를 직접 지정할 수 있습니다.

예시

이제는 `for` 문을 사용한 다양한 예시를 살펴보겠습니다. 아래는 팩토리얼을 계산하는 함수를 `for` 문을 통해 만든 것입니다.

```
1 def factorial(x):
2     product = 1
3     for i in range(1, x + 1):
4         product *= i
5     return product
```

줄 2에서 `product`의 값을 1로 초기화를 한 후, 이후 반복하여 `product`에 `i`를 1부터 `x`까지 반복하며 곱해주었습니다. *range*에서 마지막 인자는 해당 값이 되기 이전까지 반복문을 수행해줍니다.

구구단표 작성과 같은 규칙적인 작업도 `for` 문을 통해 할 수 있습니다.

```
1 for i in range(1, 10):
2     for j in range(1, 10):
3         print(i * j, end=' ')
4     print()
```

`i`를 고정시킨 후 `j`를 1부터 9까지 변화시키며 곱을 출력한 후, 줄을 바꾼 후 `i`를 1 증가시킨후 `j`를 1부터 9까지 변화시키며 곱을 출력하는 것을 반복하는 것이므로 결과적으로 구구단표를 작성하게 됩니다. 이렇게 `for` 문 안에서 또 다른 `for` 문이 수행되는 것을 중첩 반복문이라고 합니다.

코딩에서 개수 세기 등과 함께 중요한 패턴 중 하나는 최댓값과 최솟값을 찾는 것입니다. $f(n) = (x-2)^2$으로 정의된 함수에서 $f(0), \ldots, f(4)$ 중 가장 작은 값을 찾고 싶다면 다음과 같이 코드를 작성할 수 있습니다.

```
1 def f(x):
2     return (x - 2)**2
3
4 def find_min_value():
5     m = f(0)
6
7     for i in range(1, 5):
8         if f(i) < m:
```

```
9             m = f(i)
10
11    return m
```

위 코드는 첫 번째 함숫값부터 마지막 함숫값까지 하나씩 살펴보는 것과 같습니다. 첫 번째 값을 보았을 때는 해당 값이 제일 작은 것(줄 5)이고 이후로는 이전까지의 최솟값보다 크면(줄 7) 해당 값으로 최솟값이 수정되는 것(줄 8)입니다. 최댓값을 구하는 것도 마찬가지로, 줄 8의 부등호 방향을 반대로 바꾸면 최댓값을 구하는 코드가 됩니다. 어떤 주어진 값에서 제일 작거나 큰 값을 찾을 때 자주 사용하는 패턴입니다.

3.3 예제

1. 짝수일 때 `True`를 반환하는 함수 `is_odd(n)`을 작성하세요.

```
def is_odd(n):
    # Add here!

print(is_odd(12))  # False
print(is_odd(1))   # True
```

2. 어떤 수가 4의 배수이면 기본적으로 윤년입니다. 그러나 해당 연도가 100으로 나누어지면서 400으로는 안 나누어진다면, 윤년이 아닙니다. 윤년일 경우에 `True`를 반환하고 아닐 경우에는 `False`를 반환하는 함수 `leap_year`을 작성하세요.

```
# Add here!

print(leap_year(2008), leap_year(2011))  # True False
print(leap_year(2000), leap_year(2100))  # True False
print(leap_year(2300), leap_year(2400))  # False True
print(leap_year(2012), leap_year(2200))  # True False
```

3. 구구단 표를 작성하는 함수 `print_tables()`를 작성하세요.

```
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
...
9 18 27 36 45 54 63 72 81
```

```
def print_tables():
    # Add here!

print_tables()
```

4. 주어진 수 n에 따라 좌회전, 직진을 줄마다 출력하는 `turn_left_and_move(n)`를 작성하세요.

```
def turn_left_and_move(n):
    # Add here!

turn_left_and_move(3)
# 좌회전
# 직진
# 좌회전
# 직진
# 좌회전
# 직진
```

4 Lists리스트, Strings문자열, Counters카운터

4.1 Lists리스트

지난 시간에 반복문을 통해서 단순 반복 작업을 Python으로 수행하는 법에 대해서 알아보았습니다. 하지만 지난 시간에서 `for` 문의 활용은 단순히 값을 대입하여, 해당 값을 그대로 사용하는 것에 그쳤습니다. 만약 `i`를 4로 대입하는 경우에는 4를 그대로 사용하여 일반항을 계산하는 작업을 하였던 것입니다. 이번 시간에 list리스트 자료형을 배우게 된다면, 임의의 주어진 수열에 대해서 연산을 수행할 수 있습니다. 리스트는 일종의 수열로 생각할 수 있는데, `for` 문과 리스트를 사용하면 수열의 첨자를 통해 일반적인 계산을 수행할 수 있습니다.

지금까지는 10개의 변수를 받아 합을 반환하는 함수를 작성하기 위해서는 아래와 같은 코드를 작성했어야 합니다.

```
def sum_ten(n1, n2, n3, n4, n5, n6, n7, n8, n9, n10):
    summ = 0
    summ += n1
    summ += n2
    ...
    summ += n10
    return summ
```

심지어 for 문을 활용할 수도 없습니다. 다음과 같은 문법은 허용되지 않기 때문입니다.

```
def sum_ten(n1, n2, n3, n4, n5, n6, n7, n8, n9, n10):
    summ = 0
    for i in range(1, 11):
        summ += ni
    return summ
```

위에서 ni는 단지 새로운 변수 ni라는 이름을 가진 변수에 불과합니다. 그러나 리스트를 사용한다면, 단지 10개의 변수가 아니라 임의의 개수의 변수를 받아서–엄밀하게는 임의의 개수의 변수를 담고 있는 하나의 리스트형 변수를 받아서–계산을 할 수 있습니다.

```
def sum_arb(numbers):
    summ = 0
    for i in range(len(numbers)):
        summ += numbers[i]
    return summ
```

요약하자면, 묶음의 데이터를 한 번에 처리할 수 있는 방법이 리스트와 for 문을 활용하는 것입니다. 이제 리스트의 정의와, 이를 어떻게 다룰 수 있을지 자세히 알아봅니다.

리스트의 생성

리스트의 생성은 세 가지 경우로 나눌 수 있습니다. 첫 번째 경우는 리스트에 어떤 원소가 들어갈 것인지 이미 정해진 경우입니다. 이러한 경우, 직접 원소를 입력하여 리스트를 생성할 수 있습니다.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]
```

리스트의 값은 나중에 수정할 수 있기 때문에 처음에 채워 넣은 값을 initial values초기값이라고 합니다. 하지만 이렇게 값을 입력하는 것은 리스트가 담고 있는 원소의 개수가 적을 때만 가능합니다. 그렇지만 값이 불규칙할 경우에는 직접 입력할 수 밖에 없습니다.

리스트에 담을 값이 규칙적이면서, 크기가 정해져 있다면 값을 초기화하지 않은 상태에서 일정 크기를 가진 리스트를 생성할 수 있습니다. 아래는 크기 10의 리스트를 생성한 것입니다.

```
numbers = [None] * 10
```

`[None] * 10`은 `[None, None, None, None, None, None, None, None, None, None]`과 동일합니다. 위에서와 같이 `None` 말고 정수형인 `-1`이나 문자열인 `"empty"` 등으로도 채워넣을 수 있습니다. 단, `0`과 같은 값은 실제로 값을 채워 넣은 유의미한 `0`과 혼동될 수 있으니 사용하지 않는 것이 바람직합니다. 무의미한 값으로 리스트를 채웠으니 값을 채워야 합니다. 이는 `for` 문으로 해결할 수 있습니다. 아래는 첫 10개의 3의 배수를 넣는 과정입니다.

```
numbers = [None] * 10

for i in range(len(numbers)):
    numbers[i] = (i + 1) * 3
```

`len(·)` 함수가 처음 등장했는데, 이는 리스트의 길이를 반환합니다. 위의 예시에서 `len(numbers)`는 `10`을 반환합니다. 코드를 보다 일반적이고 명료하게 나타내기 위하여 리스트의 길이를 숫자 `10`과 같이 직접 치는 것보다는 `len(·)`을 사용하는 것이 좋습니다. 또한 리스트의 원소를 접근하는 방식은 `리스트_명[원소_첨자]`입니다. `원소_첨자`는 `0`에서 `len(리스트_명) - 1`까지의 값을 가집니다. 즉, 리스트의 첫 원소를 호출하려면 `리스트_명[1]`이 아닌 `리스트_명[0]`을, `n`번째 원소를 호출하려면 `리스트_명[n]`이 아닌 `리스트_명[n - 1]`을, 마지막 원소를 호출하려면 `리스트_명[len(리스트_명)]`이 아닌 `리스트_명[len(리스트_명) - 1]`을 사용해야 합니다. 정리하자면, `리스트_명[n]`은 `리스트_명`의 $n+1$번째 원소를 가리킵니다.

마지막으로는 리스트의 크기도 정해지지 않은 경우입니다. 물론, 위 두 경우처럼 개수나 원소가 정해진 경우에도 사용할 수 있는 일반적인 방법입니다. 리스트의 `append(·)` method 메소드[4.1]를 사용하는 것입니다.

```
numbers = []

for i in range(len(numbers)):
    numbers.append((i + 1) * 3)
```

처음에 아무것도 담지 않은 리스트 `[]`에다가, `.append(·)`를 통해 새로운 원소를 뒤에 하나씩 추가하는 방법입니다. 예컨대, `a = [1, 2, 3]`일 때 `a.append(4)`를 수행하면 `a`는 `[1, 2, 3, 4]`가 됩니다. 간단히 "리스트 버전의 `+=` 연산"이라고 생각하면 됩니다.

리스트의 elements원소와 indices인덱스

리스트 `리스트_명`의 element원소 `리스트_명[원소_첨자]`에서 `원소_첨자`는 index인덱스[4.2]라고 합니다. 지금까지는 수열처럼 첨자라고 했는데, 앞으로는 인덱스라고 부릅니다. 인덱스는 `0`부터 `len(리스트_명) - 1`까지의 값 이외에도, `-len(리스트_명)`에서부터 `-1`까지 사용할 수 있습니다.

[4.1] 객체에 관련된 함수를 뜻하는 말입니다. 객체 지향 프로그래밍을 배울 때 알아볼 것입니다. 지금으로는 단순히 어떤 변수에 딸린 함수라고 생각하면 됩니다.

[4.2] 복수형은 indices입니다.

리스트_명[-1]은 리스트_명[len(리스트_명) - 1]과 같으며, 리스트_명[-len(리스트_명)]은 리스트_명[0]과 같습니다. Modulo모듈로 len(리스트_명)로 생각할 수 있는데, len(리스트_명) 이상의 값이나 -len(리스트_명) 미만의 값은 인덱스로 사용하면 IndexError: list index out of range 에러를 볼 수 있습니다. 복잡한 코드에서 주의를 기울이지 않으면 자주 보게 될 에러 메시지인데, 해당 메시지가 뜨면 for 문의 리스트 인덱스를 다시 한 번씩 살펴보아야 합니다.

리스트는 굉장히 범용적인 container용기라고 생각할 수 있습니다. 리스트는 정수형 혹은 실수형 수 뿐만이 아니라 문자열, 불리언, 나아가 리스트들을 담을 수 있습니다. 또한, 한 종류의 자료형이 아니라 여러 종류의 자료형을 한 리스트에 담을 수 있습니다. 즉, 아래와 같은 예시 모두 가능합니다.

```
>>> type(["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"])
<type 'list'>
>>> type([True, None, [3, 2, 1], 3.14, "one"])
<type 'list'>
```

나아가 인덱스를 통해 리스트에 담긴 원소를 다룰 때는 해당 원소의 자료형처럼 그대로 다룰 수 있습니다.

```
>>> numbers = [None] * 4
>>> numbers[0] = 3
>>> numbers[0]
3
>>> numbers[1]
None
>>> numbers[1] = numbers[0] * 2
>>> numbers[1]
6
>>> numbers[2] = numbers[1] - 7
>>> numbers[2]
-1
>>> numbers[3] = numbers[2]**2
>>> numbers[3]
1
>>> numbers[3] /= 2.
>>> numbers[3]
0.5
>>> numbers[4] = 2 * numbers[3]
>>> numbers[4]
1.0
```

리스트의 인덱스는 정수형이라면 어떠한 표현을 사용하여도 문제가 되지 않습니다.

```
for i in range(4):
    print numbers[(i + 2)%4]
```

위 예시는 `numbers[2]`, `numbers[3]`, `numbers[0]`과 `numbers[1]`을 차례대로 출력합니다.

리스트의 연산

리스트에서 유용한 연산 중에는 slicing슬라이싱이 있습니다. 슬라이싱이란 리스트의 부분 리스트를 만드는 연산으로, 아래와 같이 사용할 수 있습니다.

```
>>> l = [0, 1, 2, 3, 4, 5]
>>> l[1:4]
[1, 2, 3]
>>> l[0:3]
[0, 1, 2]
>>> l[:3]
[0, 1, 2]
>>> l[2:6]
[2, 3, 4, 5]
>>> l[2:]
[2, 3, 4, 5]
>>> l[:]
[0, 1, 2, 3, 4, 5]
>>> l[1:-1]
[1, 2, 3, 4]
```

즉, `l`이라는 리스트가 주어졌을 때 `l[i:j]`는 `l[i]`부터 `l[j - 1]`까지의 원소를 뽑아서 새로운 리스트를 만듭니다. 만약 `i`가 생략된다면 첫 문자부터 시작하며, `j`가 생략된다면 마지막 문자까지 포함을 합니다. 둘 다 생략하는 경우 같은 원소들을 담고 있는 리스트의 복사본을 만들게 됩니다.

원소들을 특정 간격으로 띄워서 부분 리스트를 만들 때도 슬라이싱을 사용할 수 있습니다. 이 경우 `l[i:j:k]`와 같이 슬라이싱을 하는데, `k`만큼의 간격을 두고 원소들을 추출하게 됩니다. `range(i, j, k)`와 동일하다고 생각하면 됩니다. 곧 알게 되겠지만 `range(·)`는 리스트를 호출하는 메소드이기 때문입니다.

```
>>> l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[1:9:2]
[1, 3, 5, 7]
```

```
>>> l[:9:2]
[0, 2, 4, 6, 8]
>>> l[1::2]
[1, 3, 5, 7, 9]
>>> l[1:-1:2]
[1, 3, 5, 7]
```

위에서 볼 수 있듯이, `i`, `j`, `k`에서 `i`와 `j`, 혹은 둘 다 생략을 할 수 있습니다. 또한 `k`가 음수일 경우, 원소를 뒤에서부터 골라서 추출하게 됩니다. 특히 `k = -1`로 지정하여 `l[::-1]`과 같이 사용한다면 `l`의 원소를 거꾸로 나열한 새로운 리스트를 생성하게 됩니다.

또한 두 리스트를 비교할 때는 다른 자료형과 마찬가지로 `==`를 사용하여 비교할 수 있습니다. 만약 특정 원소가 리스트에 해당하는지를 알아보려면 `in`을 사용하면 됩니다. 이 두 연산은 불리언을 반환하게 됩니다.

```
>>> l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> m = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l == m
True
>>> 4 in l
True
>>> 10 in l
False
>>> [1, 4] in l
False
>>> [1, 4] in [[1, 4], 2, 3]
True
```

두 리스트를 잇고 싶은 경우 단순히 `+`를 사용하여 연결하고, n 회 반복한 만큼의 리스트를 만들고 싶다면 `*`를 사용하면 됩니다. 아래와 같이 사용할 수 있습니다.

```
>>> l = [0, 1, 2] + [3, 4, 5, 6, 7, 8, 9]
>>> m = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l == m
True
>>> l = [1, 2] * 3
>>> l
[1, 2, 1, 2, 1, 2]
```

Multi-Dimensional 다차원 리스트

다차원 리스트는 리스트들의 리스트입니다. 이는 matrix행렬을 Python으로 구현하고자 할 때 유용하게 사용할 수 있습니다. (일단 행렬은 아래와 같이 행과 열에 맞추어 수들을 나열한 수학적 객체라고 생각하시면 됩니다.)

$$l = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 & 7 \end{pmatrix}$$

위와 같은 행렬은 아래와 같은 리스트로 표현할 수 있습니다.

```
l = [[1, 2, 3, 4, 5], [3, 4, 5, 6, 7]]
```

만약 `l`의 행의 길이를 알고 싶다면 `len(l[0])`을, 열의 길이를 알고 싶다면 `len(l)`을 사용하면 됩니다. 또한 각 원소를 접근할 때에, 1행 1열의 원소를 접근하고 싶다면 `l[0][0]`, 1행 2열의 원소를 접근하고 싶다면 `l[0][1]`, 2행 3열의 원소를 접근하고 싶다면 `l[1][2]`를 사용하면 됩니다. 일반적으로, `i`행 `j`열의 원소는 `l[i - 1][j - 1]`로 접근할 수 있습니다.

다차원 리스트의 경우에도 미리 원소를 지정하지 않고 `None`으로 초기화하여 리스트를 생성할 수도 있습니다. 만약 `height`만큼의 행의 개수, `width`만큼의 열의 개수를 가지는, `None`으로 초기화된 리스트를 생성한다면, 다음과 같이 생성할 수 있습니다. 아래 예시에는 `i`행 `j`열의 값을 `i + 2*j + 1`로 지정하는 과정까지 담고 있습니다.

```
height = 3
width = 4
table = [[None]*width for i in range(height)]

for i in range(height):
    for j in range(width):
        table[i][j] = i + 2*j + 1
```

위 과정을 통해서

$$l = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ 3 & 5 & 7 & 9 \end{pmatrix}$$

에 대응되는 리스트 `[[1, 3, 5, 7], [2, 4, 6, 8], [3, 5, 7, 9]]`를 구성하게 됩니다.

나아가, 임의 차원을 가지는 리스트를 구성할 수 있습니다. 3차까지는 시각적으로 길이, 너비, 높이를 가지는 각 격자점에 값이 놓여 있다고 생각하시면 됩니다. 이 경우, 3중 루프를 구성하여 값을 채워 넣을 수 있습니다.

```
height = 3
width = 4
depth = 2
table = [[[None]*width for j in range(height)] for i in range(depth)]

for i in range(depth):
    for j in range(height):
        for k in range(width):
            table[i][j][k] = i + 2*j + k + 1
```

알아두면 유용한 내장 함수

리스트 `l`가 주어졌을 때,

- `sum(l)`: `l`의 모든 원소의 합을 반환합니다.
- `min(l)`: `l`의 최소원을 반환합니다.
- `max(l)`: `l`의 최대원을 반환합니다.
- `l.append(x)`: `l`에 `x`를 마지막 위치에 추가하며 반환값이 없습니다.
- `l.count(x)`: `l`에 포함된 `x`의 개수를 반환합니다.
- `l.insert(i, x)`: `l`의 `i`번째 인덱스에 `x`를 추가하며 반환값이 없습니다.
- `l.pop(i)`: `l`의 `i`번째 인덱스에 해당하는 원소를 없애고 해당 값을 반환합니다.
- `l.remove(x)`: `l`의 첫 번째 `x`를 없앱니다.
- `l.reverse()`: `l`을 뒤집으며 반환값이 없습니다.
- `l.sort()`: `l`의 원소들을 정렬하며 반환값이 없습니다. 이 때, 수들 먼저 정렬된 후 문자열이 알파벳 순서로 정렬이 됩니다.

이 때 반환값이 없는 메소드에 대해서는 다른 원소를 정의할 때 사용할 수가 없다는 점을 유의해야 합니다. 예컨대, `m = l.append(x)`를 하면 `l`에는 `x`가 추가되지만, `m`에는 `None`이 배정됩니다. 아래 예시를 모아두었습니다.

```
>>> l = [0, 1, 2]
>>> m = sum(l)
>>> m
3
>>> m = min(l)
>>> m
```

```
0
>>> m = max(l)
2
>>> m = l.append('hi')
>>> m
None
>>> l
[0, 1, 2, 'hi']
>>> m = l.count(0)
>>> m
1
>>> m = l.insert(1, 3}
>>> m
None
>>> l
[0, 3, 1, 2, 'hi']
>>> m = l.pop(3)
>>> m
2
>>> l
[0, 3, 1, 'hi']
>>> m = l.remove(0)
>>> m
None
>>> l
[3, 1, 'hi']
>>> m = l.reverse()
>>> m
None
>>> l
['hi', 1, 3]
>>> m = l.sort()
>>> m
None
>>> l
[1, 3, 'hi']
```

4.2 Strings문자열

문자열은 리스트와 비슷하게 다룰 수 있습니다. 지금까지 문자열은 단순히 어떤 문구를 출력하기 위해 사용하는 정도로 그쳤었다면, 이제는 문자열을 가지고 연산을 하는 법을 알아볼 것입니다. 리스트 `l`의 `i` 번째 원소를 접근하기 위하여 `l[i - 1]`을 사용하였다면, 어떠한 문자열 `s`의 `i` 번째 문자를 접근하기 위하여 동일한 방식으로 `s[i - 1]`을 사용할 수 있다. 슬라이싱도 동일한 문법을 따릅니다. 나아가 두 문자열을 연결하기 위해서는 `+`를, 반복하기 위하여 `*`에 수를 곱하는 방식으로 연산하는 것까지 동일합니다. 또한 부등호를 통해 사전식 배열을 하였을 때 어떤 문자열이 먼저 나오는지를 판단할 수 있습니다. `in`과 `not in`을 통해 어떤 문자열이 다른 문자열의 부분 문자열이 되는지도 판단할 수 있습니다.

```
>>> s = "WEIZMANN"
>>> s[2]
'I'
>>> s[1:4]
'EIZ'
>>> s * 3
'WEIZMANNWEIZMANNWEIZMANN'
>>> s = s + "PYTHON"
>>> s
'WEIZMANNPYTHON'
>>> "WEIZMANN" > "PYTHON"
True
>>> 'MANN' in s
True
```

문자열의 연산

리스트와의 차이라면 리스트는 mutable가변 객체이고, 문자열은 immutable불변 객체라는 것입니다. 이에 따라 한 문자열을 통해 새로운 값을 만들고 싶다면 새로운 문자열을 정의하는 방식을 사용해야 하며, 기존의 문자열을 수정하는 것은 불가능합니다. 반면, `+=`이나 `*=`을 사용하는 것은 가능합니다. 이는 `s += 'a'`는 단순히 `s = s + 'a'`를 축약한 것으로써 동일한 이름의 문자에 새로운 값을 배정하는 것을 의미하기 때문입니다.

```
>>> s = "WEIZMANN"
>>> s[0] = "w"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s += "PYTHON"
```

```
>>> s
"WEIZMANNPYTHON"
```

알아두면 유용한 내장 함수들

문자열을 더 편리하게 다루기 위한 다양한 내장 함수들도 준비되어 있습니다. 문자열 `s`와 `t`가 주어졌을 때,

- `s.upper()`: `s`의 모든 문자를 대문자로 바꾸어 반환합니다.
- `s.lower()`: `s`의 모든 문자를 소문자로 바꾸어 반환합니다.
- `s.capitalize()`: `s`의 첫 문자는 대문자로, 나머지는 소문자로 바꾸어 반환합니다.
- `s.isupper()`: `s`의 모든 문자가 대문자라면 `True`를, 아니라면 `False`를 반환합니다.
- `s.islower()`: `s`의 모든 문자가 소문자라면 `True`를, 아니라면 `False`를 반환합니다.
- `s.isdigit()`: `s`의 모든 문자가 숫자라면 `True`를, 아니라면 `False`를 반환합니다.
- `s.split()`: `s`에서 띄어쓰기 문자로 구분된 구절들을 원소로 가지는 리스트를 반환합니다.
- `s.find(t)`: `s`에 `t`가 포함되는 경우, `t`가 나타나는 첫 위치의 인덱스를 반환하며, 포함되지 않는 경우 -1을 반환합니다.

리스트와는 다르게 문자열을 수정하는 메소드는 없는 것을 확인할 수 있습니다. 전에 언급하였던 것처럼 문자열은 불변 객체이기 때문입니다.

```
>>> s = "wEiZmANN".upper()
>>> s
'WEIZMANN'
>>> s = "weiZManN".lower()
>>> s
'weizmann'
>>> s = "weizmAN".capitalize()
>>> s
'Weizmann'
>>> "WEIZMANN12".isupper()
True
>>> "weizmanN12".islower()
False
>>> "1242".isdigit()
True
>>> l = "Hello World!".split()
>>> l
```

```
['Hello', 'World!']
>>> s = "WEIZMANN"
>>> s.find("IZ")
2
```

4.3 Turtle

5 Quantifiers한정자와 While 문

6 Loops반복문 응용과 파일 입출력

7 Recursion재귀법, Python의 다양한 객체, 그리고 Lambda람다 함수

8 Object-Oriented Programming객체 지향 프로그래밍

9 정렬 알고리즘

10 Divide-and-Conquer분할 정복법

11 Dynamic Programming동적 계획법