Programming with Python 101

이재호

목차

머리말

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."

— Lewis Carroll, *Alice in Wonderland*

파이썬 *Python*은 문법이 단순하지만 활용 가능성이 무궁무진한 언어입니다. 단순히 학교에서 필수적으로 가르치기 때문에 파이썬을 배운다기보다는 앞으로 다양한 분야에서 오랫동안 활용할 수 있는 유용한 도구를 배운다는 마음가짐을 가지면 좋겠습니다. 파이썬은 R이나 MATLAB 등과 함께 학문적인 용도로도 쓰임이 많은 언어입니다. 특히 NumPy, SciPy, Matplotlib, Pandas 등의 패키지를 활용한다면 MATLAB이나 Mathematica와 같은 유료 프로그래밍 언어가 할 수 있는 다양한 작업들을 대등하게 할 수 있습니다. 나아가 파이썬은 오픈 소스에 무료인데다가, 여러 분야에서 활용할 수 있는 범용 프로그래밍 언어입니다.

저는 물리학, 천문학, 화학, 생물학 등의 학문 분야에서 데이터 분석 및 시각화를 위해 파이썬을 사용하기도 하고, Raspberry Pi에 연결된 다양한 센서와 카메라 모듈 등을 제어하기 위해 파이썬을 사용한 프로젝트를 진행하는데에도 사용했습니다. 이 외에도 사이트에서 여러 자료를 동시에 다운로드 받는 파이썬 스크립트를 작성하거나, 간단한 계산을 수행하기 위해서 사용하는 등 파이썬은 제 일상에 자연스럽게 녹아든 든든한 도구가 되었습니다.

웹 서버 개발에 관심이 있다면, 파이썬을 사용해 손쉽게 서버를 만들 수 있습니다. 특히 Django, Flask, FastAPI 등의 프레임워크를 사용한다면 큰 규모의 서버도 무리 없이 개발할 수 있습니다. YouTube, Dropbox, Facebook, Netflix, Google, Instagram, Spotify 등의 대형 서비스들은 이미 여러 부품에 파이썬을 사용하고 있습니다.

딥러닝에 관심이 있다면, 파이썬은 필수로 배워야 하는 언어입니다. 2022년 현재 PyTorch, Tensorflow, Hugging Face 등 거의 모든 딥러닝 프레임워크는 파이썬을 통한 API를 제공하고 있습니다.

나아가 파이썬은 스크립트 언어로 쉘 스크립트 대신 시스템 수준의 작업을 수행할 때 사용할 수 있습니다. 과거에는 awk, 펄 *Perl* 등의 언어를 사용하는 작업을

파이썬으로 작성하는 경우가 증가하고 있습니다.

이처럼 파이썬은 한 번 배워두면 수 많은 곳에서 유용하게 사용할 수 있습니다.

구성 및 독자층

본 교재는 파이썬 3의 문법과 알고리즘을 기초부터 익힐 수 있도록 구성되어 있습니다. 특히 파이썬의 기본적인 문법 숙지와, 자주 사용되는 패턴에 익숙해지는 것에 초점을 맞추고 있습니다.

전체적으로는 프로그래밍 언어를 처음 접하더라도 익힐 수 있도록 구성되어 있으나, 일부 내용은 어느 정도의 수학적 배경 지식을 가정합니다.

Chapter 1

시작하기

```
printf("hello, world\n");
```

— Brian Kernighan, *The C Programming Language*

1.1 파이썬의 특징

파이썬은 1991년에 귀도 반 로섬 *Guido van Rossum*이 발표한 프로그래밍 언어로, 문법이 굉장히 단순하면서 높은 생산성을 가지고 있는 언어입니다.[1] 특히 파이썬으로 (잘) 작성된 코드는 의사코드 *pseudocode*처럼 쉽게 읽히기 때문에 진입 장벽이 낮고 비교적 배우기가 쉽습니다. 이러한 특징 덕분에 많은 학교에서는 프로그래밍 입문 수업을 파이썬으로 진행하고 있습니다.[2]

하지만 뭐니뭐니해도 파이썬의 가장 큰 장점은, 파이썬 패키지 인덱스 *Python Package Index, PyPI*에 포함된 방대한 양의 패키지 생태계입니다. 이 때문에 파이썬은 흔히 "배터리가 포함된 *batteries included*" 언어라고 불립니다 (그림 1.1). 따라서 파이썬을 사용하면 기존에 만들어진 도구들을 조립하여 원하는 프로그램을 빠르게 만들어낼 수 있습니다.

프로그래밍에 익숙하지 않은 사람도 컴퓨터에 관심이 있다면 C, C++, 자바 *Java* 등의 언어와 함께 파이썬도 한 번 쯤 들어보았을 정도로, 파이썬은 점유율 상위 다섯 언어 안에 드는 주류 언어입니다. 파이썬은 1995년에 등장한 자바보다도 오래 전에 만들어진 언어로, 짧지 않은 역사를 가지고 있습니다.[3]

파이썬은 실행 *inpterpreted* 언어입니다. 실행 언어란 실행기 *interpreter*가 코드를 한 줄씩 읽으면서 그때그때 "동시 통역"하는 방식의 언어입니다. 이와 대조되는 방식은 번역 *compiled*입니다. 번역 언어는 번역기 *compiler*가 코드 전체를 먼저 기계어로 "통번역"이 되어야 실행할 수 있습니다. 이때 기계어는 주어진 CPU와

[1] 귀도 반 로섬은 1989년 크리스마스 연휴에 취미로 프로그래밍 언어를 만들었는데, 이것이 파이썬의 시초가 되었습니다.

[2] 과거에 BASIC, C, 자바 등으로 교육을 한 학교들도 점차 파이썬으로 커리큘럼을 바꾸어 나가고 있는 추세입니다.

[3] 이 글을 작성하는 현 시점의 파이썬 최신 버전은 3.11.1입니다.

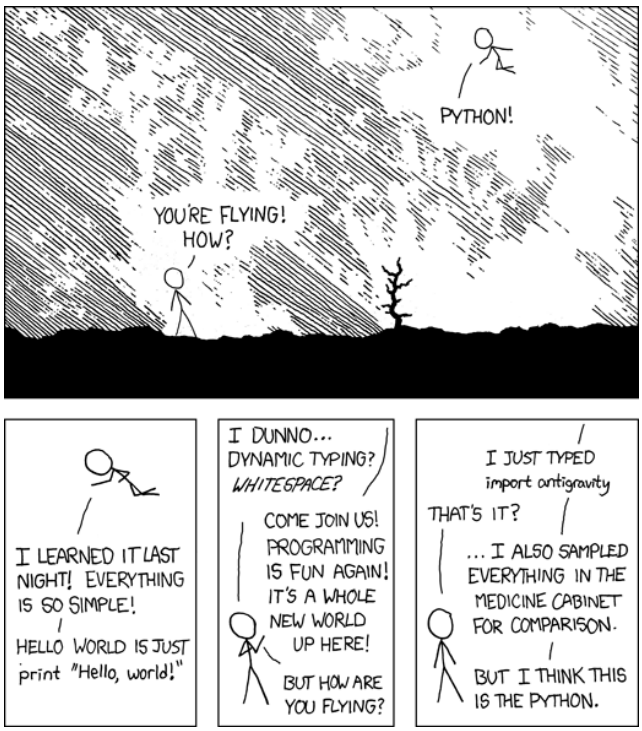


그림 1.1: 파이썬은 배터리가 포함된 언어입니다. 출처: https://xkcd.com/353/

운영체제 *operating system, OS*가 바로 이해할 수 있는 형태의 저수준 언어입니다. C와 같은 언어가 이에 해당됩니다. 이 때문에 번역 언어는 번역된 결과물의 실행 속도가 빠르다는 장점이 있지만, 프로그램을 수정하기 위해서 아무리 작은 변화를 주더라도 전체 코드를 다시 한 번 번역하는 과정이 필요합니다.[4]

이제 파이썬이 얼마나 쉽고 직관적인 언어인지 알아봅시다. 아래는 `A+`가 `F`, `C+`, `B0`, `A-`, `A+`의 목록에 포함되어 있으면 `A+가 있습니다.`를 출력하는 파이썬 코드입니다:

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

프로그래밍을 할 줄 모르더라도 위 코드가 어떤 일을 하는지는 대강 이해할 수 있습니다. 사람이 구사하는 문장과 크게 다르지 않기 때문입니다. 이처럼 파이썬으로는 하고 싶은 일을 직관적으로 코드로 옮길 수 있습니다.

반면 C 언어에서는 위와 같은 작업을 하기 위해서 아래와 같은 코드를 작성해야 합니다:

[4] 엄밀하게 말하면 언어 자체가 실행되는지 번역되는지를 결정하는 것이 아니라, 언어의 구현 *implementation*이 실행기인지 번역기인지 나뉘게 됩니다. 주어진 언어에 대해서 여러 구현이 존재할 수 있기 때문입니다. 파이썬의 경우, CPython은 실행기 방식으로 만들어져 있습니다.

```c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
    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");
            break;
        }
    }
    return 0;
}
```

단 두 줄의 파이썬 코드로 될 일을 C 언어로는 10줄 이상으로 작성해야 하는 것입니다.[5]

1.2 파이썬의 기본 요소

1.2.1 안녕, 세상!

```
#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.

그림 1.2: K&R *The C Programming Language*의 첫 프로그램

1978년에 출판된 *The C Programming Language*[6]에서 예시로 사용된 이후 첫 프로그램은 보통 `Hello, World!`를 출력하는 것으로 시작됩니다 (그림 1.2). 먼저, 통합 개발 환경 *integrated development environment, IDE*[7]이나 텍스트 편집기 *text editor*[8]에 다음과 같은 코드를 입력해봅시다:

[5] 이는 C가 나쁘다는 것을 보여주는 것이 아니라, 파이썬으로 얼마나 간결하고 빠르게 코드를 작성할 수 있는지를 보여주는 예시입니다. C는 시스템 수준의 작업을 할 때 꼭 필요한 매력적인 언어입니다.

[6] 저자인 Brian Kernighan과 Dennis Ritchie의 초성을 딴 K&R로 흔히 불리웁니다.

[7] PyCharm, Wing IDE 등이 있습니다.

[8] Visual Studio Code, vi(m), GNU Emacs 등이 있습니다. 저자는 vim을 8년 넘게 사용하다가 GNU Emacs로 이주하였습니다.

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

이를 실행하면 `Hello, World!`가 출력될 것입니다. 이제 파이썬의 문법에 대해서 본격적으로 알아봅시다.

1.2.2 값과 변수

파이썬의 Integrated Development and Learning Environment, IDLE이나 쉘 *shell*에서 직접 파이썬 실행기를 실행하여 다음을 입력해 봅시다.[9] (>>>는 직접 입력하는 것이 아니라 IDLE 혹은 쉘이 새로운 입력을 받기 위해 표시하는 것입니다.)

```
>>> a = (1 + 2 + 3 + 4) // 2
>>> b = a - 1
>>> c = 3
>>> print(b + c)
7
>>> print(b, c)
4  3
>>> d = c
>>> a = d
>>> print(a)
3
```

`a = (1 + 2 + 3 + 4) // 2`는 등호 왼쪽에 있는 a에 등호 오른쪽에 있는 $\frac{1+2+3+4}{2}$를 계산한 결과를 대입 *assign*한다는 뜻입니다. 즉, 수학에서 말하는 등호와는 의미가 다른 것이지요. 줄 2의 `b = a - 1`은 현재 a에 배정된 $\frac{1+2+3+4}{2} = 5$의 값에서 1을 뺀 4를 b에 배정한다는 뜻입니다. 이 때 좌변에 있는 a, b, c를 변수 *variable*, 우변의 `(1 + 2 + 3 + 4) // 2`를 식 *expression*이라고 합니다. 또한 이러한 변수에 배정되는 결과를 값 *value*이라고 합니다. 다시 위의 예시로 돌아가서, 줄 2에 있는 좌변의 b는 변수, 우변의 `a - 1`는 식, 그 결과인 4는 값인 것이죠. 영어 문장으로 "let {variable} be {value}."가 말이 되는지 대입하여 보면 쉽게 알 수 있습니다.

변수와 값은 문자인지 숫자인지의 여부가 아니라 대입이 되는 대상인지 대입이 되어지는 대상인지의 여부가 결정짓습니다.[10] 식의 구성 요소가 변수 하나만 있는 경우, 그 "결과"는 값이 될 수 있습니다. 줄 8의 같은 경우, 우변의 c는 변수이고, d라는 변수에 대입될 값을 만드는 식입니다.

그렇다면, 다음과 같은 표현을 어떨까요?

[9] 다시 언급하자면, 파이썬은 실행 언어이기 때문에 명령을 한 줄씩 입력하여 명령을 수행할 수 있는 것입니다.

[10] 실제로 이렇게 간단하지는 않지만, 지금의 시점에서는 이렇게 이해하셔도 좋습니다.

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

지금까지 잘 따라왔다면, 변수가 위치해야 할 좌변에 값인 리터럴 *literal*[11]이 위치한 잘못된 표현인 것을 알 수 있습니다. 실제로 이를 실행하면 `SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?`와 같은 문구를 확인할 수 있습니다. 이처럼 파이썬은 우리가 잘못된 표현을 실행하려고 하면 최대한 그 잘못을 알려주려고 합니다. 따라서 앞으로 이와 같은 에러를 마주치면 반드시 뭐라고 써있는지 꼼꼼히 확인하는 습관을 들여야 합니다.

값은 여러 종류로 분류할 수 있습니다. 이러한 분류 기준을 자료형 *data type* 혹은 간단히 타입 *type*이라고 합니다. `-1`, `0`, `76` 등의 값은 `int`[12] 타입, `3.14159`, `0.`, `6.626e-34`, `6.022E23` 등의 값은 `float`[13] 타입, `"Hello, world!"`, `'python'`, `""` 등의 값은 `str`[14] 타입입니다. 타입에 대해서는 조금 뒤에 더 자세히 살펴봅시다.

상태: 기계로 보는 관점

변수에는 어떤 값이 대입된다고 하였습니다. 이렇게 변수에 대입된 값은 읽을 수 있는데, 첫 예시의 줄 2처럼 `a`에 저장된 값 5를 불러오는 것이 이에 해당합니다.

나중에 다른 값을 변수에 대입할 수도 있습니다. 첫 예시에서 줄 9를 보면, 5가 저장되었던 `a`에 `d`에 저장된 값 3이 다시 `a`에 쓰여지는 것을 볼 수 있습니다. 줄 10에서 이 사실을 재확인할 수 있고요.

이처럼 변수는 재활용될 수 있고, 개수를 세거나 특정 사건을 기록하기 위한 목적[15]으로 사용하는데 도움이 됩니다:

```
>>> summ = 0
>>> summ = summ + 1
>>> summ = summ + 1
>>> print(summ)
2
```

이런 방식으로 "시간이 흐름에 따라" 변수의 값이 바뀔 수 있는 것은 파이썬이 프로그램을 기계로 보는 관점을 채택하였기 때문입니다. 이에 따라서 프로그램을 작성할 때는 현재 지점에서의 상태 *state*를 잘 생각해야 합니다. 실제로 방대한 규모의 프로그램을 작성할 때는, 상태를 어떻게 잘 관리할 것인지가 매우 중요한

[11] `0`은 정수 *integer* 리터럴, `"python"`은 문자열 *string* 리터럴입니다. 정수와 문자열은 (뒤에 알아볼) 타입 *type*입니다.

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

[13] 부동 소수점 *floating point*에서 따온 명칭입니다. 실수가 아니라 근삿값이라는 의미에 더 가깝습니다.

[14] 나열이라는 뜻의 string에서 따온 명칭입니다.

[15] 특히 어떤 사건을 기록하는 목적으로 만들어진 변수를 깃발 *flag*이라고 합니다.

문제로, 셀 수 없이 많은 소프트웨어 아키텍처 *software architecture*가 이 문제를 해결하기 위해 만들어졌습니다. 흔히 절차형 프로그래밍 *procedural programming*이 이와 같은 관점으로 프로그래밍을 바라보는 패러다임 *paradigm*입니다.

시간이 흘러도 변수의 값이 바뀌지 않도록 프로그래밍을 하는 패러다임도 있는데요, 이를 함수형 프로그래밍 *functional programming*이라고 합니다. 함수형 프로그래밍이라고 부르는 이유는, 이러한 방식의 프로그래밍 언어에서는 (우리가 아직 배우지 않은) 함수가 프로그래밍의 기본 요소로 필요하기 때문입니다. 이에 대해서는 뒤에서 살펴볼 기회가 있을 것입니다.

파이썬에서는 여러 변수에 여러 값을 동시에 배정할 수 있습니다. 이를 통해 두 변수에 담긴 값들을 쉽게 바꿔치기 *swap*할 수 있습니다. 다음 예시를 통해 함께 확인합시다.

```
>>> a, b, c = 2, 7, 12
>>> a, b, c = b, c, a
>>> print(a, b, c)
7 12 2
>>> a, b = b % a, a
>>> print(a, b)
5 7
```

여기서 `%` 연산자는 나머지 연산자로, `b % a`는 `b`를 `a`로 나눈 나머지를 의미합니다.

이러한 다중 대입 *multiple assignment*을 사용하지 않는다면 아래와 같이 임시 변수를 도입해서 바꿔치기를 해야 합니다.

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

마치 하노이의 탑에서 한 원판을 다른 원판으로 옮기기 위해서 기존의 원판을 다른 막대에 잠시 옮겨 놓는 것처럼 말이죠.

변수의 이름을 정할 때에는 지켜야 할 규칙이 있습니다. 파이썬이 내부적으로 사용하는 `if`, `else`, `for`, `try`, ...등이 이에 해당합니다. 이들을 변수명으로 사용하려 하면 파이썬은 문법 오류로 인식하게 됩니다. 추가적으로, `int`와 같은 타입 이름도 변수 이름으로 사용하지 않는 것이 좋습니다. 이는 `int`, `str`와 같은 타입 변환을 위한 함수가 이미 있기 때문에, 변수로 배정하면 기존의 값을 덮어씌우게 됩니다. 나아가 `sum`과 같은 내장 함수들도 덮어씌우지 않는 것이 좋습니다. 위에서 `summ`이라고 쓴 변수 이름도 `sum`이 이미 존재하기 때문에 일부러 겹치지 않게 피해서 정하였습니다.

변수명은 문자, 숫자, 그리고 _로만 이뤄져 있어야 합니다. 문자라고 하면 알파벳 대소문자, 한글 등으로, $, @ 등의 특수 문자는 포함하지 않습니다. 여기서 주의해야 할 점은, 변수명은 1st_name처럼 숫자로 시작할 수는 없다는 것입니다.

1.2.3 식

식 *expression*은 변수, 값, 그리고 연산자들의 "적절한" 조합입니다. 연산자에는 우리가 흔히 부르는 사칙 연산 +, -, *, /와, 나머지를 구해주는 %, 지수를 뜻하는 ** 등이 포함됩니다. ^이 아니라 **이 지수 연산자인 점에 유의해야 합니다. 그리고 /은 실수 연산이고 //이 정수 연산으로, 후자가 몫을 구해줍니다.[16]

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
```

참고로 ^는 비트 XOR 연산자(eXclusive OR)로, $12 = 1100_2$, $5 = (0)101_2$ 이므로 각 자릿수별로 숫자가 다른 2^3, 2^2, 2^1 의 자릿수만 1을 취한 $1001_2 = 9$가 12 ^ 5의 값입니다.

연산의 순서는 기본적으로 괄호((...)), 단항 연산(+x, -x), 지수(**), 곱셈/나눗셈/나머지 연산(*, /, %), 덧뺄셈(+, -)의 순으로 진행됩니다. 헷갈리는 경우나 혼동을 불러올 수 있는 경우에는 괄호((...))를 사용하여 순서를 명시할 수 있습니다. 이를 연산자 우선순위 *operator precedence*라고 하고, 표 1.1에 (아직 배우지 않은 연산자들까지) 정리되어 있습니다. 지수 연산자와 조건문을 제외하고는 모두

[16] 참고로, 파이썬 2에서는 나눗셈을 할 때 정수끼리 행하면 몫만이 구해집니다. 12 / 5 = 2와 같이 말입니다. 그리고 파이썬 2에서는 12.0 / 5 = 2.0와 같이 제수나 피제수 중 하나라도 float 형이면 결과도 실제 float의 나눗셈의 결과로 나옵니다.

표 1.1: 연산자의 우선 순위입니다. 지수 연산자와 조건문을 제외하고는 왼쪽에서 오른쪽으로 결합됩니다.

연산자	설명
`(...)`, `[...]`, `{key: value...}`, `{...}`	괄호, 리스트, 사전, 집합
`x[...]`, `x(...)`, `x.a`	인덱스, 호출, 속성
`**`	지수
`+x`, `-x`	단항 연산
`*`, `/`, `//`, `%`	곱셈, 나눗셈, 나머지
`+`, `-`	덧뺄셈
`in`, `not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	비교 연산
`not x`	NOT (¬)
`and`	AND (∧)
`or`	OR (∨)
`if ... else ...`	조건문
`lambda`	람다

왼쪽에서 오른쪽으로 결합됩니다. 즉, `a + b + c`는 `(a + b) + c`로 해석 *parse* 됩니다.

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

줄 1의 표현보다는 줄 4의 표현이 가독성이 높을 뿐만이 아니라, 더 일반적이어서 값을 바꾸기 위해서는 줄 2의 숫자 부분만 변경을 하면 됩니다. 나아가 줄 1의 표현의 경우 `+`와 `-`의 부호 구분에서 실수를 할 가능성이 매우 높습니다.[17]

마지막으로 소개할 문법은 변수 뒤에 산술 연산자(`+`, `-`, `/`, `//`, `%`, `**` 등) 뒤에 바로 `=`를 붙이는 연산입니다. `x += 1`과 같이 말입니다. 이는 해당 변수에 저장된 값을 읽은 후 등호 뒤에 쓰인 값을 더하여 바로 다시 변수에 쓴다는 의미로, `x = x + 1`과 동일한 의미를 가지고 있습니다. 이는 덧셈 말고 다른 연산자들에도 동일하게 적용됩니다:

[17] 눈치채셨나요? 실제로 위의 식에는 잘못된 부분이 있습니다. 마지막 항에서 `(3 + 4 - 5)`가 아니라 `(3 - 4 - 5)`라고 오타가 있습니다.

```
>>> x = 4
>>> x += 2
>>> x
6
>>> x -= 1
>>> x
5
>>> x *= 2
>>> x
10
>>> x //= 5
>>> 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.[18] a가 3의 값을 가지고 있었을 때 a++를 하면 4가 되는 것이지요. 그러나 ++a와 a++에 따라서 연산 후 a에 담기는 값은 같지만, 1을 더한 후 값을 읽어오는지 값을 읽은 후 1을 더하는지의 미묘한 차이가 있어서 찾기 힘든 버그의 원인이 되기도 합니다. 물론 파이썬에는 이와 같은 문법이 없기 때문에 안심하셔도 됩니다.

1.2.4 타입

앞서 살펴보았듯이, 값의 종류를 타입 *type*이라고 합니다. 지금까지 정수(int), 실수(float), 그리고 문자열(str) 세 종류의 타입을 알아보았습니다.

유의해야할 점은, 3.과 3은 서로 다른 타입의 값이라는 것입니다. 컴퓨터는 내부적으로 3.과 3을 완전히 다른 방식으로 표현하고 저장합니다.[19]

파이썬에서는 타입 이름의 함수를 사용하여 여러 타입끼리 손쉽게 변환을 할 수 있습니다. 다음의 예시를 살펴봅시다.

[18] C++의 ++이 해당 연산자에서 따온 것입니다. C에 무언가 가미되었다는 의미이죠.

[19] 실수는 IEEE Standard for Floating-Point Arithmetic (IEEE 754)라는 부동소수점 *floating-point*으로, 정수는 2의 보수 *two's complement*로 표현합니다.

```
>>> 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
```

나아가 type 함수를 통해 직접 타입을 확인할 수 있습니다.

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

1.2.5 입출력

지금까지는 파이썬 쉘에서만 명령을 실행했습니다. 그렇기 때문에 a에 담긴 값을 알기 위해서는 굳이 print(a)가 아니라 a를 치는 것만으로 충분했습니다. 그러나 어느 정도 길이가 있는 프로그램을 작성할 때에 이런 방식으로는 한계가 있습니다. 작성된 코드를 저장하기도 힘들고, 위에 실수가 있었을 때 이후의 코드를 모두 다시 직접 입력해 넣어야 하기 때문입니다.

출력

쉘에서 변수 이름을 입력했을 때 값을 보여주는 것은 마치 어떤 사람이 무엇을 먹었는지 알아보기 위해 X-레이로 들여다보는 것과 마찬가지입니다. "평상시"에는 이런 방식이 아니라, 그 사람에게 무엇을 먹었냐고 물어본 후 그 사람이 대답하는 방식으로 일을 진행해야 합니다. 이는 `print` 함수를 통해서 수행할 수 있습니다.

이제 쉘이 아니라 직접 파일을 만들어서 다음의 코드를 작성해봅시다.

```
today = "Thursday"

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 literal:")
print(f"Today is {today}")
```

위 예시는 파이썬에서 `Today is Thursday`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 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. 좀 더 간편한 방식의 문자열 포맷팅은 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 literal*입니다. 문자열 앞에 `f`를 붙인 후 단순히 중괄호 안에 원하는 변수 이름을 넣으면 대입이 됩니다.

입력

이제는 input 함수를 사용하여 파이썬 프로그램이 실행 중에 사용자로부터 값을 입력 받는 법에 대해 알아봅시다.

```
today = input("What day is it today? ")
print(f"Today is {today}.")

s = input("Enter the number you want to know the square root of: ")
n = float(s)
print(n ** 0.5)
```

위 코드를 실행시키면 창에 What day is it today? 가 출력된 후 입력이 될 때까지 기다립니다. 키보드로 값을 입력한 후–Thursday를 입력했다고 합시다–리턴 키를 치면 값이 입력되고, Today is Thursday.가 출력될 것입니다. 또한 수를 입력 받은 후 산술 연산을 취하기 위해서는 int 혹은 float로 적절히 타입 변환을 해야 합니다. input 함수는 항상 문자열로 값을 읽어오기 때문입니다.

1.3 예제

1. 1부터 n까지 자연수의 제곱의 합과 세제곱의 합의 차이를 구하는 코드를 작성하세요. 힌트:

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

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

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

2. 원탁 주위에 앨런 튜링 *Alan Turing*, 존 폰노이만 *John von Neumann*, 하스켈 커리 *Haskell Curry*, 도널드 커누스 *Donald Knuth*, 알론조 처치 *Alonzo Church*가 앉아서 "좋아하는 정수 놀이"를 하고 있습니다. 좋아하는 정수 놀이는, 각자 자신이 좋아하는 정수를 종이에 적은 다음 양 옆에 앉은 사람의 정수를 더하여 진행됩니다. 최종적으로 각자 얻은 수를 출력하도록 다음 코드를 완성하세요:

```
alan = int(input("Alan: "))
john = int(input("John: "))
```

```
haskell = int(input("Haskell: "))
donald = int(input("Donald: "))
alonzo = int(input("Alonzo: "))
print("Favorite integers: ", alan, john, haskell, donald,
↪  alonzo)
# Add here!
print("Final integers: ", alan, john, haskell, donald, alonzo)
```

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

```
a = int(input("a: "))
b = int(input("b: "))
c = int(input("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. (파이썬 3에서) 가능한 변수명을 모두 고르세요.

   - `if`
   - `sum`
   - `max`

- `1st_var`
- `Var_1`
- `this+that`
- `_self`
- **변수**

6. 다음 코드는 여섯 개의 실수 `x1`, `x2`, `x3`, `y1`, `y2`, `y3`을 입력받습니다. `i`가 1, 2, 3을 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.

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

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

```
a = int(input("a:"))
b = int(input("b:"))
# Add here!
# r = ...
print(f"Remainder is {r}")
```

8. 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. 힌트: `int(·)` 함수의 그래프를 그려보세요.

```
x = float(input("x:"))
# Add here!
# ceil_x = ...
print(f"Ceil(x) = {ceil_x}")
```

Chapter 2

함수와 조건문

> *When a certain task is to be performed at several different places in a program, it is usually undesirable to repeat the coding in each place. To avoid this situation, the coding (called a subroutine) can be put into one place only, and a few extra instructions can be added to restart the outer program properly after the subroutine is finished.*
>
> — Donald Knuth, *The Art of Computer Programming*

지금까지 파이썬에서

- 기본적인 타입과 이들에 대해 수행할 수 있는 연산자들,
- 식들을 조립하여 복잡한 식을 구성하는 방법, 그리고
- 변수를 정의하여 값을 담을 수 있는 방법을

알아보았습니다. 이러한 기능은 대부분의 언어에서 지원하는 프로그래밍 언어의 기본 요소입니다.

이번 단원에서는 일련의 연산을 하나의 단위로 구성하여 함수 *function*를 작성하고, 경우에 따라 프로그램 실행의 순서를 다르게 변화할 수 있는 구성 요소인 조건문 *conditional statements*에 대해서 알아봅니다.

2.1 함수

함수를 사용하면 논리적으로 하나의 역할을 하는 코드를 감싸서 하나의 블럭으로 만들 수 있습니다. 다음의 간단한 예시를 봅시다:

```
def square(x):
    return x * x
```

```
print(square(9))  # 81
```

위 코드는 어떤 값을 받아 제곱을 수행하는 함수를 파이썬에서 어떻게 작성하는지 보여줍니다.

우선 모든 함수[1]의 정의는 def로 시작합니다. 그 후 함수에 붙일 이름(square)이 이어지고, 괄호 안에 함수가 받을 인자들 *arguments*의 이름이 들어갑니다. 이러한 이름을 매개변수 *parameters*라고 합니다. 매개변수와 인자의 관계는, 변수와 저장되는 값의 관계와 대응됩니다. 이 괄호가 끝나면 반드시 콜론 *colon*을 사용해 함수 이름과 매개변수의 나열이 끝났으며, 함수를 구성하는 코드가 시작된다는 표시를 해야 합니다.

여기서 주목해야할 것은, (콜론 뒤에 이어지는) 함수를 구성하는 코드가 반드시 들여쓰기되어야 한다는 것입니다. 파이썬에서는 기본적으로 네 칸 띄어쓰기로 들여쓰기 *indentation*를 권장합니다.[2] 이에 대해서는 뒤에서 다시 살펴볼 것입니다.

마지막으로 함수를 구성하는 논리적인 코드 조각이 계산된 결과를 돌려주는 반환 *return*입니다. `return x * x`는 `x * x`를 계산한 결과가 함수 square의 계산 결과라는 의미입니다. 반환이라고 이름 붙여진 이유는, `square(9)`와 같이 함수를 사용할 때, square의 안에서 계산된 결과를 `square(9)`가 사용된 위치로 "돌려주어야"하기 때문입니다.

이미 보았지만 이렇게 정의된 함수는 `square(9)`와 같이 괄호 안에 값을 넘겨주어 사용할 수 있습니다. 이를 함수 호출 *function call*이라고 말합니다. 함수를 불러서 사용하는 것이기 때문입니다. 이렇게 호출된 함수는 값을 호출된 장소로 반환합니다.

이번에는 여러 식이 사용된 코드를 통해서 다시 한 번 살펴봅시다[3]:

```
import math

LIGHT_SPEED = 299792458

def square(x):
    return x * x

def gamma(v):
    b = v / LIGHT_SPEED
    return 1 / math.sqrt(1 - square(b))
```

[1] 뒤에서 배우는 람다 함수 *lambda function*가 있지만 여기서는 무시합니다.

[2] 특별한 사유가 있다면 일정한 개수의 띄어쓰기나 탭 문자를 사용할 수도 있습니다.

[3] 변수 이름들과 식을 보면 알 수 있겠지만 특수 상대성 이론 *special relativity theory*에서 상대 시간을 계산하는 코드입니다.

```
def relative_time(x, t, v):
    k = t - v * x / square(LIGHT_SPEED)
    return gamma(v) * k

position = 10
time = 13
velocity = LIGHT_SPEED / 2

print(gamma(velocity))
print(relative_time(position, time, velocity))
```

위 예시에는 앞선 square과 다르게 둘 이상의 매개변수를 사용한 함수 relative_time이 정의되어 있습니다. 코드가 길어졌지만, 구성 방식은 앞서 설명한 것과 동일합니다.

2.1.1 구조적 프로그래밍

기계어로 프로그래밍하는 방식에서 탈피하기 위한 첫 걸음마 중 하나가 함수라고 말할 수 있습니다.[4] 함수를 전혀 사용하지 않고 프로그램을 작성할 수는 있지만, 함수를 사용하는데에는 다양한 이점이 있기 때문입니다.

함수를 사용하면 프로그램을 논리적인 역할에 따라 여러 조각으로 나누는 구조적 프로그래밍 *structural programming*[5]이 가능해집니다. 또한 함수를 사용하면 같은 코드를 반복하지 않음으로써 프로그램의 용량을 줄이고 범용성을 늘릴 수 있습니다. 자주 반복하여 사용하는 식을 하나의 변수에 대입하여 사용하면 효율적인 것과 비슷한 맥락입니다.

이렇게 코드 조각을 함수로 감싸는 것을 캡슐화 *encapsulation*라고 부릅니다. 뒤에서 객체 지향 프로그래밍 *object-oriented programming*을 배우면, 연관된 일을 수행하는 여러 함수들과 변수들까지도 캡슐화를 할 수 있는 방법에 대해 알 수 있습니다.

한편, 함수로 논리를 감싸는 것을 추상화 *abstraction*라고도 부를 수 있습니다. 위에서 square 함수를 생각하면, 함수를 호출하는 입장에서는 square이 x * x 처럼 구현이 되었는지, 아니면 (비효율적으로) x를 x번 더해서 구현이 되었는지는 알 필요가 없습니다. square이 약속한 것처럼, 값을 제곱해서 돌려주기만 하면 그만이기 때문입니다. 이러한 관점으로 함수를 바라보면, 함수는 기능의 구현을 추상화하여 그 역할만 남겨놓은 블랙박스라고 생각할 수 있습니다.

[4] 반대로 번역기 *compiler*가 하는 일에는 이렇게 정의된 함수를 기계어로 녹여내는 일이 포함됩니다.

[5] 구조적 프로그래밍은 비구조적 프로그래밍과 대비되는 패러다임입니다. 비구조적 프로그래밍은 프로그램 전체를 하나의 큰 덩어리로 작성하는 방식입니다. 이러한 프로그램은 GOTO문과 같은 제어문에 의존할 수 밖에 없는데, 이는 프로그램의 디버깅을 어렵게 가독성도 해칩니다. 특히 구조화되지 않은 코드는 프로그램이 복잡해지면 실행 흐름이 복잡하게 얽히게 되는데, 이렇게 읽기 힘들고 잘 구조화되지 않은 코드를 스파게티 코드 *spaghetti code*라고 부릅니다. 사실상 기계어를 제외한 현대의 모든 프로그래밍 언어는 구조적 프로그래밍을 지원합니다.

2.1.2 들여쓰기

파이썬에서는 들여쓰기가 논리적인 단위를 나누는 중요한 문법입니다. C, C++ 등의 언어에서는 들여쓰기에 대한 문법 규칙 없이 중괄호({})와 세미콜론(;)으로 단위를 구분하며, 들여쓰기는 프로그램을 읽고 쓰는 인간을 위한 강제되지 않는 (중요한) 약속 정도입니다. 그러나 중괄호나 세미콜론을 사용하지 않는 파이썬에서는 들여쓰기가 문법적으로 반드시 필요합니다.[6] 참고로, 띄어쓰기 네 번 써서 들여쓴다고 해서 키보드의 스페이스 바를 네 번 누를 필요는 없습니다. (제대로된) IDE나 텍스트 편집기라면 환경설정에서 키보드의 탭 키를 소프트 탭 *soft tab* (띄어쓰기 여러 개를 사용하여 들여쓰는 것)으로 설정하면 됩니다.

함수뿐만이 아니라 이번 단원 뒤에서 배울 조건문, 그리고 나중에 배울 반복문 등에서 들여쓰기는 논리적 블럭을 구분하고 포함 관계를 나타냅니다. 위의 예시처럼 함수에서는 함수 이름과 매개변수 나열 후 함수의 내용물을 적을 때에는 함수의 끝까지 한 단계 들여써야 합니다. 들여쓰기를 실수로 의도한 바와 다르게 한다면 코드의 수행 결과가 완전히 달라지거나 실행조차 안 될 수 있습니다. 더 무서운 점은, 이렇게 잘못 작성된 코드가 문법적으로는 맞지만 논리적으로는 틀려서 에러 메시지 없이 원하지 않는 값만 나올 수도 있습니다. 이렇게 되면 디버깅을 하기가 굉장히 까다로워집니다.[7]

2.1.3 내장 함수와 `import`

사실 지난 단원에서도 여러 함수를 보았는데요, `type`이나 타입 변환을 위해 사용한 `int` 등이 바로 함수입니다. 이들 함수는 프로그래머가 직접 정의하지 않아도 사용할 수 있는 내장 함수 *built-in functions*입니다.

어떤 내장 함수들은 기본적으로 바로 사용할 수 있는 것이 아니라, "이러이러한 묶음 안의 함수들을 사용할 것이다"라는 것을 코드에 명시해야 쓸 수 있습니다. 이는 `import`문을 사용해서 불러오면 됩니다. 수식 계산을 위해 사용하는 패키지인 `math`로 예를 들자면, 제곱근 함수 `sqrt`, 사인 함수 `sin`와 같은 삼각 함수, 로그 함수 `log` 등을 사용하기 위해 `import math`를 해야 합니다.

지난 단원의 마지막 문제인 올림 함수는 `math` 패키지에 들어있는 올림 함수 *ceiling function* `ceil`로 이미 구현되어 있습니다. 마찬가지로 `int`와 같은 "가짜"[8] 내림 함수 *floor function*가 아니라 진짜 내림 함수인 `floor`도 `math`에 들어 있습니다. 참고로 반올림 함수 `round`는 파이썬 2에서 `math` 패키지에서 제공되었지만, 파이썬 3부터는 `math` 없이 사용할 수 있습니다.

다음의 코드로 확인해봅시다:

---

[6] 띄어쓰기를 하여 들여쓸 것인지, 탭 문자를 사용하여 들여쓸 것인지는 프로그래머들 사이에서 펼쳐지는 오랜 논쟁입니다. 다만 파이썬 개발자들은 띄어쓰기 4회를 주로 사용합니다. 이는 파이썬의 스타일 가이드인 PEP-8에 명시되어 있습니다.

[7] 들여쓰기를 명시적으로 나타내주는 IDE의 기능을 활용하면 생산성을 높일 수 있습니다. 대부분의 IDE나 프로그래밍용 텍스트 에디터들은 들여쓰기를 도와주는 기능이 포함되어 있습니다.

[8] $\texttt{int(-1.3)} = -1 \neq \lfloor -1.3 \rfloor = -2$

```
>>> 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.`을 반복하기 귀찮다면, `from package import *`로 package 안의 모든 함수를 밖으로 끄집어내어 가져올 수 있습니다:

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

여기서 뭔가 무시무시한 것을 발견하였나요? 이제 자연 상수 e를 `math.e`가 아니라 그대로 `e`와 같이 사용해야 합니다. (짧은 변수 이름을 사용하는 것이 좋은 습관은 아니지만) `e`를 다른 곳에서 부주의로 인해 정의한다면 자연 상수에 해당하는 e의 값이 아니라 다른 값이 `e`에 담기게 됩니다. 다음과 같이 말이죠.

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

또한 같은 함수 이름을 가진 서로 다른 두 함수—fn이라고 합시다—를 지닌 두 패키지 pack1과 pack2에서 `from pack1 import *`를 한 후 `from pack2 import *`를 한다면 뒤에서 불러온 pack2의 fn이 먼저 불러온 pack1의 fn을 덮어 씌울 것입니다. 예컨대 numpy라는 패키지를 컴퓨터에 설치한 후, `from numpy import *`와 `from math import *`를 한다면 상당수의 함수가 겹치게 되어 두 패키지 중 하나의 것만 사용할 수 있게 됩니다. 따라서 불가피한 경우에만 `from package import *`를 사용하는 것이 권장됩니다.

나아가 자신이 math 패키지에서 sin 함수처럼 몇 개의 함수들만 필요하다면, 매번 math.을 붙이지 않고 직접 표시한 함수들을 사용할 수도 있습니다:

```
>>> from math import sqrt, sin
>>> sqrt(4)
2.0
>>> sin(0)
0.0
```

그리고 패키지를 불러올 때 이름을 원하는대로 바꿀 수도 있는데요, `import numpy as np`와 같이 패키지를 불러온다면, numpy.sin 대신 np.sin과 같이 사용할 수 있습니다.

2.1.4 함수와 부작용

함수는 반환 값이 있는 것과 그렇지 않은 것으로 나눌 수 있습니다. 그렇지 않은 것은 (함수가 어떤 식으로든지 유용한 일을 한다면) 무조건 부작용 *side effects*을 일으킵니다.

아래는 원의 반지름을 받아서 면적을 반환하는 함수 circ_area입니다.

```
import math

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

return 다음에 반환할 값이 나타나 있습니다. 아래는 원의 면적을 반환하지 않고 단순히 출력하는 함수 print_circ_area입니다.

```
import math

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

circ_area의 경우에는 원의 면적을 출력하기 위해 print를 사용해야 합니다. 그렇지만 다른 값을 계산하는 등의 작업에서 함수를 다른 식에 넣을 수 있습니다. print_circ_area는 자신이 직접 원의 면적을 출력하지만, 그 값을 다른 식에서 활용할 수는 없습니다. 반환하는 값이 없기 때문입니다.

함수를 정의하여 사용할 때에는, 호출하는 시점 이전에 정의가 되어 있으면 됩니다. 이 조건만 만족한다면 함수의 정의는 다른 코드들 사이에 끼워두어도 되고, 함수들 간의 정의 순서도 상관이 없습니다. 또한 함수를 정의하기 위해서 다른 함수를 (당연히) 사용할 수 있습니다.

```
import math

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

def circ_area(cx, cy, px, py):
    r = get_dist(cx, cy, px, py)
    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 정의 순서는 바뀌어도 상관 없습니다.

2.1.5 매개 변수와 인자

함수를 정의할 때 사용되는 매개 변수는 함수 정의 밖에서 사용될 수 없습니다. 일종의 블랙 박스로 생각하면 되는데, 함수 안에서 정의되는 지역 변수들 *local variables* 은 그 안에서만 사용할 수 있습니다. 예를 들어 직전 예시의 get_dist 함수의 dx나 dy는 함수 안에서 만들어졌다가 사라지는 변수입니다. 매개 변수도 지역 변수입니다. 나아가 함수 밖에서 dx라는 이름의 새로운 변수를 만들더라도 이는 함수 안에서 정의되었던 지역 변수 dx와는 전혀 다른 변수입니다. 우연히 이름이 겹친 것일 뿐입니다.

반면 함수 밖에서 먼저 정의되었으면서 함수 안에서 정의되지 않은 변수는 전역 변수 *global variable*라고 부릅니다. 이 경우에는 함수 밖에서 정의된 값이 함수 안에서 그대로 사용됩니다.

아래의 예시를 통해 지역 변수와 전역 변수를 나누는 기준을 완전히 이해했는지 확인해봅시다. 이해를 위해 (이름이 같더라도) 다른 변수들은 다른 색상으로 표시하였습니다.

```
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 `a`와 `b`는 아래에서 인자로 사용된 `a = 1` 및 `b = 2`와 무관합니다. 줄 7과 줄 8의 출력 값이 같다는 사실을 통해, 줄 7의 `fn1(a, b)`는 단지 밖에서 정의된 `a`의 값인 1과 `b`의 값인 2를 대입한 `fn1(1, 2)`와 완전히 동일한 역할을 한다는 것을 확인할 수 있습니다. `fn2`의 경우는 약간 다릅니다. 함수 내부에서 `c`가 지역 변수로 새로 정의되지 않은 상태로, 줄 14에서 `c`를 사용하였습니다. 여기에서 쓰인 `c`는 `c = 3`로 정의된 전역 변수입니다. 반면 `fn2` 내부에서 사용된 `d`는 전역 변수 `d`가 아니라 새로 `a - b`의 값을 가지는 지역 변수입니다.

이와 같이 어떤 변수가 유효한 영역을 범위 *scope*라고 합니다. 마지막 예시를 통해 변수 범위의 의미를 확실히 파악할 수 있을 것입니다.

```
def mult(a, b):
    x = a * b
    return x

def div(a, b):
    x = a / b
    return x
```

```
a, b = 1, 2
print(mult(b, a))  # is mult(2, 1)
print(div(a, b))   # is div(1, 2)
print(x)  # NameError: name 'x' is not defined
```

함수 `mult`와 `div`, 그리고 줄 9 이후의 `a`, `b`, `x`는 서로 무관하며, 줄 12에서는 함수 `mult`와 `div`의 변수 영역 밖에서 `x`가 정의된 적이 없으므로 오류가 발생합니다.

2.2 조건문

절댓값 함수

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

를 앞서 배운 함수로 구현해봅시다.[9]

```
def abs(x):
    return (x ** 2) ** 0.5
```

와 같이 정의할 수도 있습니다. 다만 이는 불필요하게 제곱과 제곱근을 사용한 $|x| = \sqrt{x^2}$의 정의를 사용한 것으로, 위에 작성한 정의를 따르지 않았습니다. 나아가, 인자가 정수 타입이더라도 반환값은 실수 타입으로 바뀌게 됩니다.

이러한 함수를 자연스럽게 구현하고 싶을 때 조건문 *conditional statement*을 사용할 수 있습니다:

```
def abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

절댓값 함수처럼 조각적으로 정의된 *piecewise-defined* 함수는 보통 조건문을 통해 구현할 수 있습니다. 나아가 조건문은 경우를 나눠서 작동하는 수많은 상황들에 적용됩니다. 예를 들어, 사용자가 로그인하였을 때와 그렇지 않았을 때 보여줘야 하는 정보가 다른 웹페이지는 조건문을 사용하여 이를 구현합니다. 낮과 밤에 따라서 화면의 밝기를 조절하는 프로그램을 작성할 때에도 조건문을 쓸 것입니다.

[9] 이미 abs 내장 함수가 있지만 직접 구현해봅시다.

장애물을 맞닥뜨렸을 때 로봇이 해야 하는 행동을 지정하기 위해서도 조건문이 필요합니다. 사실상 조건문 없이는 여러 경우에 대처하는 프로그램을 작성하기 힘듭니다.

로봇 예시를 사용하자면,

```
def check_obstacle(distance):
    if distance < 10:
        print("앞에 장애물이 있습니다.")
    else:
        print("앞에 장애물이 없습니다.")
```

와 같은 예시를 작성할 수 있습니다. 여기서 `distance < 10`는 `distance`가 10 미만의 값을 가졌는지 확인하는 조건 *condition*입니다. 이렇게 `if` 뒤에 오는 조건은 `True` 혹은 `False` 두 종류의 값을 가질 수 있는데요[10], 이는 불리언 *Boolean*이라는 타입의 값입니다.

2.2.1 불리언 타입과 식

지금까지 소개한 `int`, `float` 등의 타입 이외에도 자주 쓰게 될 불리언[11] 타입은 참과 거짓을 나타냅니다. 특이한 점으로는, 불리언 타입은 유한한 개수의 값을 가진다는 점입니다. 그냥 유한한 것도 아니고, `True`와 `False` 단 두 값만 가집니다.

위 예시의 `x >= 0`은 `True`일수도 `False`일수도 있는 불리언 타입의 식입니다. 불리언 식을 구성하기 위해서는 대소 관계, 일치 여부, 포함 관계 등을 판단하는 관계 연산자와 `and`, `or`, `not` 등의 논리 연산자로 구성됩니다. `x >= 0 or (y < 0 and z < 0)`이 불리언 식의 예시입니다.

관계 연산자는 다음의 여섯 개가 있습니다.

- `x == y`: $x = y$이면 `True`
- `x != y`: $x \neq y$이면 `True`
- `x > y`: $x > y$이면 `True`
- `x < y`: $x < y$이면 `True`
- `x >= y`: $x \geq y$이면 `True`
- `x <= y`: $x \neq y$이면 `True`

[10] 혹은 참과 거짓을 판단할 수 있는 기준이 정해진 다른 타입의 값들을 쓸 수도 있는데요, 지금은 무시합니다.

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

지금까지 어떤 변수 x에 값을 대입할 때에는 =를 사용하였습니다. `x = 10`과 같이 말입니다. 수학에서 두 값의 관계에 따라 다른 논리를 풀어나가려 할 때 "$x = 10$인 경우,"와 같은 표현을 사용하는데, 파이썬에서는 값의 일치를 비교할 때 등호 하나가 아니라 두 개를 연달아 쓴 ==를 사용합니다. `if x == 10`와 같이 말입니다.

표 2.1: 논리 연산자 and, or, not에 대한 진리표입니다.

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

논리 연산자는 표 2.1에 볼 수 있듯 and, or, not 세 종류가 있는데요, 각 식에 대해서 어떤 불리언 값을 가지는지에 대한 진리표가 제시되어 있습니다. 위 표에는 각 표현에 대해서 어떤 불리언 값을 가지는지 진리표가 제시되어 있습니다. and, or, not은 각각 수학의 $\wedge$, $\vee$, 그리고 $\neg$에 대응됩니다.

이들 연산자의 우선순위 *operator precedence*는 지난 단원의 표 1.1에 정리되어 있습니다. 모든 관계 연산자들은 논리 연산자들보다 순위가 높고, 산술 연산자들보다는 낮습니다.

지금까지 불리언 자료형에 대한 논리 연산자와 그 우선순위를 살펴보았는데요, 이제 조건문과 함께 사용한 예시를 살펴봅시다.

2.2.2 if와 else

if와 else를 사용한 조건문의 기본적인 형태는 다음과 같습니다.

```
if expression:
    ...
else:
    ...
```

위에서 expression에는 `x // 10 == 0 or x < 0` 등 임의의 불리언 식이 올 수 있습니다. 만약 제시된 불리언 식의 계산된 값이 True라면 줄 2의 내용이 실행되고, 그렇지 않다면 줄 4의 내용이 실행됩니다. 계산된 값이 False일 때 실행할 내용이 없다면 else 부분을 빠뜨릴 수 있습니다:

```
if expression:
    ...
```

다음의 예시를 봅시다.

```
x = 76
if x % 3 == 0 and x % 2 == 0:
    print(f"{x}은(는) 6의 배수입니다.")
else:
    print(f"{x}은(는) 6의 배수가 아닙니다.")
```

실행해보면 76은 6의 배수가 아니기에 76은(는) 6의 배수가 아닙니다.가 출력됩니다.

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

```
x = 76
if x % 3 == 0 and x % 2 == 0:
    print(f"{x}은(는) 6의 배수입니다.")
else:
    if x % 3 == 0:
        print(f"{x}은(는) 3의 배수입니다.")
        else:
            if x % 2 == 0:
                print(f"{x}은(는) 2의 배수입니다.")
            else:
                print(f"{x}은(는) 2의 배수도 3의 배수도 아닙니다.")
```

6의 배수인지 체크한 후, 그렇지 않다면 다시 3의 배수인지 체크한 후, 그렇지 않다면 2의 배수인지 체크하는 코드입니다. 작동은 하겠지만, 이렇게 들여쓰기가 많다면 코드를 읽기 힘들 것입니다. 이런 상황에서는 `elif`를 쓸 수 있습니다. 조건문에 세 개 이상의 가지 *branch*가 있을 때, `if`로 시작하여 두 번째부터 마지막 바로 직전 경우까지는 `elif`를, 마지막 예외의 경우에는 `else`로 마무리하는 방식입니다. 여기서 주의해야 할 것은, `if`와 `elif` 뒤에는 항상 조건문을 써야 하고, `else` 뒤에는 조건문을 절대 쓸 수 없다는 것입니다. 조건문을 처음 배운다면 `elif` 뒤에 조건문을 안 쓰거나 `else` 뒤에 조건문을 작성하는 실수를 흔히 합니다.

`else`는 어떠한 경우의 조건에도 해당하지 않을 때 실행할 코드를 쓰는 조각입니다. 모든 경우에 해당하지 않을 때 시행할 것이 없으면 마지막 `else`는 생략할 수 있습니다.

`elif`를 활용하면 위의 코드를 아래와 같이 간결하게 나타낼 수 있습니다.

```
x = 76
if x % 3 == 0 and x % 2 == 0:
    print(f"{x}은(는) 6의 배수입니다.")
elif x % 3 == 0:
    print(f"{x}은(는) 3의 배수입니다.")
elif x % 2 == 0:
    print(f"{x}은(는) 2의 배수입니다.")
else:
    print(f"{x}은(는) 2의 배수도 3의 배수도 아닙니다.")
```

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

참고로, `if-else`문에서 불리언 식 대신 `int` 등의 타입을 가진 변수를 넣어도 됩니다. 예컨대 `0`은 거짓, `0` 이외의 모든 수는 참의 값을 가집니다. 그렇다고 이러한 값들이 항상 `True` 혹은 `False`의 값을 가지는 것은 아닙니다.[12] 예컨대 불리언 식 `2 == True`의 값은 `False`입니다. 또한 문자열의 경우 `""` 외의 하나 이상의 문자를 담고 있는 문자열은 참입니다. `""`는 거짓에 해당합니다. 그렇다고 `"" == False`인 것은 아닙니다. 이를 통해 특정 경우에 좀 더 파이써닉 *Pythonic*한 코드를 작성할 수도 있습니다. 하지만 이것도 가독성을 해치지 않는 선에서 써야 합니다.[13]

위에서 `True`와 `False`가 각각 `1`과 `0`의 값을 가진다는 사실을 사용해 다음과 같은 장난도 칠 수는 있습니다:

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

어디까지나 이것이 가능하다는 사실을 알려주는 코드이고, 실제로는 이렇게 쓰면 큰 봉변을 당할 수도 있으니 주의하세요.

다만 위에서 `0`이 아닌 `int`가 참인 것을 사용해 다음과 같이 간결한 코드를 작성할 수 있습니다:

```
money = 100
if money:
```

[12] 실제로 `0 == False`, `1 == True`입니다.

[13] 커누스는 "Programs are meant to be read by humans and only incidentally for computers to execute"라고 말한 적이 있습니다.

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

2.2.3 다중 반환점

한 함수에 `if-else`문을 넣어 경우에 따라 다른 값을 반환하도록 할 수 있습니다. 반환한다는 것은 함수에서 호출한 곳으로 값을 돌려주는 것이었습니다. 즉, `if-else`문을 사용해 함수 안 여러 곳에서 값을 돌려줄 수 있습니다. 위에서 작성된 abs 함수가 그 예시로, `x >= 0`일 경우 `x`를 반환하고 그렇지 않은 경우 `-x`를 반환하였습니다. 이러한 함수에는 여러 `return`을 사용하였기 때문에 다중 반환점 *multiple exit points*이 있다고 표현합니다. 만약 `abs` 함수를 하나의 반환점만 사용한다면 아래와 같게 다시 쓸 수 있습니다.

```
def abs(x):
    if x >= 0:
        result = x
    else:
        result = -x
    return result
```

혹은 `else` 가지 없이 아래와 같이 다시 쓸 수도 있습니다.

```
def abs(x):
    if x >= 0:
        return x
    return -x
```

위 코드의 경우에는 가능한 경우를 모두 "걸러서" 마지막 경우에는 `-x`를 반환하라고 읽을 수 있습니다. 함수를 실행할 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 표시로 볼 수 있습니다.

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

```
def foo(x):
    if ...:
```

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

2.3 예제

1. 이름 name을 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 로섬입니
↪  다.
print(introduce("앨런 튜링"))     # 제 이름은 앨런 튜링입니다.
print(introduce("폰 노이만"))     # 제 이름은 폰 노이만입니다.
```

2. 두 수 a와 b를 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
```

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

3. 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.

```
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. 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 떨어져있다고 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.

```
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. $a \neq 0$일 경우, 함수 $f(x) = ax^2 + bx + c$는 극값을 가집니다. 함수 $f(x) = ax^2 + bx + c$의 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 direction이 w일 경우 **전진**, a일 경우 **좌회전**, s일 경우 **후진**, d일 경우 **우회전**을 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 $x > 0 \wedge y > 0$
   - "제2사분면" if $x < 0 \wedge y > 0$
   - "제3사분면" if $x < 0 \wedge y < 0$
   - "제4사분면" if $x > 0 \wedge y < 0$
   - "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 n에 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 n이 주어졌을 때, 10, 50, 100, 500원 동전을 최소로 사용해서 n원을 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. 함수 ceil(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. 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. 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!
```

15. `"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
```

16. `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
```

Chapter 3

불리언 함수와 간단한 반복문

Formula of my happiness: a Yes, a No, a straight line, a goal.

— Friedrich Nietzsche, *Twilight of the Idols*

3.1 불리언 함수

지난 단원에서는 조건문과 함께 불리언 타입에 대해 알아보았습니다. 또한 코드를 논리적 단위로 추상화한 함수에 대해서도 배웠습니다. 이번 절에서는 이러한 불리언 값인 `True`와 `False`를 반환하는 불리언 함수들을 살펴볼 것입니다.

3.1.1 예시

불리언 함수는 `True`혹은 `False`를 반환하는 함수입니다. 이러한 불리언 함수는 길고 복잡한 조건문을 간결하게 표현할 때 유용합니다.

다음은 길이 `a`, `b`, `c`를 선분의 길이로 가지는 삼각형이 존재하는지 판별하는 조건문입니다:

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

이는 다음의 함수 `can_form_triangle`로 대신할 수 있습니다:

```
def can_form_triangle(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 can_form_triangle(a, b, c):
    ...
```

조건문이 복잡해질수록, 불리언 함수를 사용해 조건문을 쪼개는 것이 도움이 됩니다. "두 수 n_1과 n_2가 서로소라면"과 같은 표현을 한 조건문으로 표현하는 것보다는, 서로소이면 `True`를, 아니면 `False`를 반환하는 함수 `coprime`을 정의한 후 `if coprime(n1, n2): ...`와 같이 작성하는 것이 바람직합니다.

나아가, 불리언 함수는 (모든 함수가 그렇지만) 재사용성을 높혀줍니다. 예를 들어, 로봇이 거리를 측정한 후 장애물이 앞에 있는지 확인하는 함수 `check_obstacle`을 생각해봅시다[1]:

```
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.

3.1.2 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'>
```

[1] `check_obstacle`은 곧바로 `return distance < 10`으로 쓸 수도 있습니다. 이러한 패턴은 뒤에서 다시 살펴볼 것입니다.

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

`True`는 문자열이 아니라 그 자체로 불리언 자료형을 지닌 값입니다.[2] 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 ...:
    return True  # or False
```

같은 형태는 단순히 `return ...`으로 쓸 수 있습니다. 아래는 $ax^2 + bx + c = 0$의 실근이 존재하는지를 판별하는 함수 `has_real_solution(a, b, c)`입니다:

```
def has_real_solution(a, b, c):
    if b ** 2 - 4 * a * c >= 0:
        return True
    return False
```

이는 다시 (간략히)

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

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

나아가 불리언 함수를 조건문에 사용할 때는 `func(...) == True`보다는 바로 `func(...)`를 쓰는 것이 바람직합니다. 즉

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

보다는

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

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

이 더 파이써닉한 표현입니다. (`if func(...) == True:`을 쓰려다가 `==` 대신 `=`을 사용하는 실수를 미연에 방지할 수 있는 것은 덤입니다.)

언제 강조하여도 부족하지 않은 것은, 복잡한 표현을 끊어서 표현하는 것이 미연의 실수를 방지하고 코드의 가독성을 높인다는 점입니다. 다음의

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

보다는

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

이 훨씬 읽기 편한 것을 볼 수 있습니다.

마지막으로, `float`끼리는 등호 연산 `==`을 하면 안 됩니다.

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

이러한 결과는 앞서 잠시 언급된 컴퓨터의 부동 소수점 처리 방식의 반올림 오차 *rounding error* 때문입니다. 간단히는 이진법으로 십진법 소수(小數)를 정확하게 나타낼 수 없어 발생한 오차가 누적된 것이라 생각하면 됩니다. 따라서 `float`끼리는 `>`와 `<`를 사용해 오차 범위 내의 비교만 신뢰할 수 있습니다. 등호 비교는 되도록 `int` 상태에서 비교해야 합니다. 예를 들어, 두 정수 격자점 사이의 거리는 제곱근을 씌우기 전에 비교를 할 수 있습니다. 수학적으로는 $\|\mathbf{v}\| < \|\mathbf{u}\| \Leftrightarrow \|\mathbf{v}\|^2 < \|\mathbf{u}\|^2$ 이기 때문입니다.

3.2 간단한 반복문

컴퓨터는 인간보다 단순 반복 계산을 빠르게 수행할 수 있습니다.[3] 이번 절에서 배울 반복문 *loops*을 사용하면 손으로 할 수 없는 수많은 계산을 파이썬으로 수행할 수 있게 됩니다. 지금까지 배운 조건문과 이번 절의 반복문을 배우면 이론적으로 어떠한 종류의 계산이든지 수행할 수 있게 됩니다.[4]

간단한 예시로는 로봇에게 반복적인 작업을 수행하도록 반복문을 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 반복문을 사용해 다음과 같이 쓸 수 있습니다.

```
for i in range(4):
    move_forward(1)
    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 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.

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

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

[3] 다만, 최근에는 AlphaGo, ChatGPT 등의 등장으로 상황이 급변하고 있습니다.

[4] 물론 자기 자신(의 간단한 버전)을 이용해 스스로 정의되는 재귀 함수 *recursive function*를 사용하면 "반복문" 없이도 모든 계산을 수행할 수 있습니다.

3.2.1 range 함수를 활용한 반복문

range 함수는 인자를 한 개에서 세 개까지 받을 수 있습니다. 일단 인자가 하나인 경우를 봅니다.

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

이는

```
0
1
2
...
n - 1
```

를 출력합니다. 즉, range(n)을 사용하여 0에서 n이 되기 직전까지 i를 1씩 증가시키며 print(i)을 실행하는 코드입니다. 참고로 반복문이 끝난 뒤 i는 n - 1로 남아 있습니다.

range에 두 개의 인자가 전달되는 경우는 아래에 나타나 있습니다.

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

이는

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

의 값들을 출력합니다. range(m, n)은 m에서 n이 되기 직전까지 i를 1씩 증가시키며 print(i)를 실행하는데, m ≥ n이라면 반복문이 수행되지 않습니다.

마지막으로 세 개의 인자가 range에 전달되는 경우를 살펴봅시다.

```
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 띄어쓰기를 하기 위함입니다. (파이썬의 내장 `print` 함수에는 `end`에 값을 넣어 값을 출력한 후 뒤에 붙일 문자를 직접 지정할 수 있습니다.)

3.2.2 예시

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

```
def factorial(x):
    prod = 1
```

```
    for i in range(1, x + 1):
        prod *= i
    return prod
```

줄 2에서 product의 값을 1로 초기화를 한 후, 이후 반복하여 product에 i를 1부터 x까지 반복하며 곱해주었습니다.

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

```
for i in range(1, 10):
    for j in range(1, 10):
        print(i * j, end=" ")
    print()
```

i를 고정시킨 후 j를 1부터 9까지 변화시키며 곱을 출력한 후, 줄을 바꾼 후 i를 1 증가시킨후 j를 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.

```
def f(x):
    return (x - 2) ** 2

def find_min_value():
    m = f(0)

    for i in range(1, 5):
        if f(i) < m:
            m = f(i)

    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. 홀수일 때 `True`를 반환하고 짝수일 때 `False`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. 위의 `is_odd`를 사용해 `exactly_two_even(n1, n2, n3)`를 작성하세요. `exactly_two_even(n1, n2, n3)`은 `n1`, `n2` 와 `n3` 중 정확히 두 개가 짝수이면 `True`, 아니면 `False`를 반환합니다.

```
def exactly_two_even(n1, n2, n3):
    cnt = 0
    # Add here!
    return cnt == 2

print(exactly_two_even(1, 2, 3))   # False
print(exactly_two_even(4, 2, 3))   # True
print(exactly_two_even(7, 3, 4))   # False
print(exactly_two_even(2, 5, 8))   # True
print(exactly_two_even(2, 18, 4))  # False
```

3. `a`, `b`, `c`를 세 변으로 가지는 삼각형이 있는지 판별하는 함수 `triangle(a, b, c)`를 작성하세요. 이때 `a`, `b`, `c`는 양의 정수입니다.

```
# Add here!

print(triangle(3, 4, 5))  # True
print(triangle(1, 5, 2))  # False
print(triangle(1, 1, 1))  # True
print(triangle(3, 1, 1))  # False
```

4. 세 점 `(x1, y1)`, `(x2, y2)`, `(x3, y3)`가 예각 삼각형을 이룰 수 있는지 판별하는 함수 `acute(x1, y1, x2, y2, x3, y3)`를 작성하세요.

```
# Add here!

print(acute(1, 2, 4, 3, 2, 7))  # True
print(acute(1, 2, 4, 2, 5, 4))  # False
print(acute(1, 2, 4, 2, 4, 3))  # False
```

5. 중심이 각각 (x1, y1)과 (x2, y2)이며 반지름은 각각 r1과 r2인 두 원이 있습니다. 이때 두 원이 두 점에서 만나는지 판별하는 함수 intersect(x1, y1, r1, x2, y2, r2)를 작성하세요.

```
# Add here!

print(intersect(1, 1, 3, 5, 4, 2))  # False
print(intersect(1, 1, 3, 4, 3, 2))  # True
print(intersect(1, 1, 3, 2, 1, 2))  # False
```

6. 어떤 수가 4의 배수이면 기본적으로는 윤년입니다. 그러나 해당 연도가 100으로 나누어지면서 400으로는 안 나누어진다면, 윤년이 아닙니다. 윤년일 경우에 True를 반환하고 아닐 경우에는 False를 반환하는 함수 leap_year을 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
```

7. 위에서 작성한 leap_year 함수를 사용해, year년의 month달에 며칠이 있는지 계산하는 함수 num_days(year, month)를 작성하세요.

```
def num_days(year, month):
    assert 1 <= month <= 12
    if month == 1 or month == 3 or month == 5 or month == 7 or \
       month == 8 or month == 10 or month == 12:
        return 31
    # Add here!
```

```
print(num_days(2000, 1), num_days(2001, 4), num_days(2004, 8))
↪  # 31 30 31
print(num_days(2004, 9), num_days(2005, 3), num_days(2005, 7))
↪  # 30 31 31
print(num_days(2008, 2), num_days(2011, 2), num_days(2012, 2))
↪  # 29 28 29
print(num_days(2000, 2), num_days(2100, 2), num_days(2200, 2))
↪  # 29 28 28
print(num_days(2300, 2), num_days(2400, 2), num_days(3200, 2))
↪  # 28 29 29
```

8. 수혈은 다음과 같은 혈액형 조합에서 가능합니다:

   - O형에서 O형
   - A형에서 A 또는 O형
   - B형에서 B 또는 O형
   - AB형에서 A, B, AB 또는 O형.

   함수 `blood(supply_O, supply_A, supply_B, supply_AB, demand_O, demand_A, demand_B, demand_AB)`는 각 혈액형별 수요와 공급을 받아 혈액이 충분한지를 판별해야 합니다. 이를 작성하세요.

```
def blood(supply_O, supply_A, supply_B, supply_AB,
          demand_O, demand_A, demand_B, demand_AB):
    if supply_O < demand_O:
        return False
    # Add here!

print(blood(50, 36, 11, 8, 45, 42, 10, 3))  # False
print(blood(50, 36, 11, 3, 45, 38, 10, 7))  # True
```

9. 아래의 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
```

10. 주어진 수 n에 따라 **좌회전**, **직진**을 줄마다 출력하는 `turn_left_and_move(n)`를 작성하세요.

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

turn_left_and_move(3)
# 좌회전
# 직전
# 좌회전
# 직전
# 좌회전
# 직전
```

11. 다음과 같은 값들을 출력해야 합니다.

```
1 2 3 4 5 6 7 8 9
3 6 9 12 15 18 21 24 27
5 10 15 20 25 30 35 40 45
7 14 21 28 35 42 49 56 63
9 18 27 36 45 54 63 72 81

1
3 6 9
5 10 15 20 25
7 14 21 28 35 42 49
9 18 27 36 45 54 63 72 81
```

이런 일을 하는 함수 `print_tables()`를 작성하세요.

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

print_tables()
```

12. 함수 `sum_interval(a, b)`는 두 정수 a와 b를 받아 $a + (a + 1) + \cdots + b$를 계산합니다. 이를 작성하세요.

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

print(sum_interval(5, 10))  # 45
print(sum_interval(15, 100))  # 4945
```

13. 원주율에 수렴하는 아래의 관계식이 알려져 있습니다[5]:

$$\pi = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots\right)$$

위 식의 첫 n항들을 더해 π의 근사값을 구하는 함수 `calculate_pi(n)`를 작성하세요.

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

print(calculate_pi(1000))
```

14. 원점을 중심으로 하고 반지름이 r인 원 내부의 정수 격자점을 세는 함수 `within_circ(r)`를 먼저 작성하세요.

몬테 카를로 기법 *Monte Carlo method*를 통해 다음과 같이 원주율을 계산할 수 있습니다:

$$\pi = \lim_{r \to \infty} \frac{\texttt{within\_circle(r)}}{r^2}$$

이를 통해 원주율을 근사하는 `calculate_pi(r)`를 작성하세요.

```
def within_circ(r):
    cnt = 0
    # Add here!
    return cnt

def calculate_pi(r):
    # Add here!

print(calculate_pi(1000))
```

[5] 라이프니츠 *Gottfried Wilhelm Leibniz*가 발견하였습니다.

15. 날짜가 주어졌을 때 요일을 `"Mon"`, `"Tue"`, `"Wed"`, `"Thu"`, `"Fri"`, `"Sat"`, 혹은 `"Sun"`로 반환하는 함수 `day_of_week(year, month, day)`를 작성하세요. 1970년 1월 1일 목요일 이후의 날짜만 고려합니다. 위에서 작성한 `leap_year`과 `num_days`를 활용하세요.

```
def day_of_week(year, month, day):
    # Add here!

print(day_of_week(2023, 1, 19))  # Thu
```

16. 함수 $f : \{0, 1, \ldots, 99\}^2 \to \mathbb{Z}$는 다음과 같이 정의됩니다:

$$f(i, j) = \left(i^5 + 2i^3j^2 + 5i^2 + j + 5000\right) \pmod{10000}$$

f의 최댓값을 구하는 함수 `find_max()`를 작성하세요.

```
def f(i, j):
    return (i**5 + 2* i**3 * j**2 + 5 * i**2 + j + 5000) % 10000

def find_max():
    # Add here!

print("Maximum is", find_max())  # 9997
```

17. `n` 미만의 3 혹은 5의 배수의 합을 계산하는 함수 `sum_mlt35(n)`을 작성하세요.

```
def sum_mlt35(n):
    # Add here!

print(sum_mlt35(10))    # 23
print(sum_mlt35(100))   # 2318
print(sum_mlt35(1000))  # 233168
```

18. 피보나치 수열의 첫 `n` 항 중 짝수인 것들을 더한 값을 계산하는 함수 `sum_even_fib(n)`를 작성하세요. 피보나치 수열의 첫 두 항은 1과 2입니다.

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

```
print(sum_even_fib(300)) == ...   # Too large to write here
print(sum_even_fib(400)) == ...   # Too large to write here
print(sum_even_fib(1000)) == ...  # Too large to write here
```

19. 먼저 n의 진약수의 합을 구하는 함수 d(n)을 작성하세요.

    만약 $b = \mathtt{d}(a) = \mathtt{d}(\mathtt{d}(b))$ $(a \neq b)$이면, a와 b는 우호적인 쌍이고, 각각을 우호적인 수 *amicable number*라고 부릅니다. n이 우호적인 수인지 판별하는 함수 is_amicable(n)을 작성하세요.

    이제 n 미만의 모든 우호적인 수들의 합을 구하는 함수 sum_amicable(n)을 작성하세요.

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

def is_amicable(n):
    # Add here!

def sum_amicable(n):
    # Add here!

print(sum_amicable(10000))  # 31626
```

Chapter 4

리스트, 문자열, 카운터

What we have to learn to do we learn by doing.

— Aristotle, *Ethica Nicomachea II*

4.1 리스트

지난 단원에서는 반복문으로 단순 반복 작업을 수행하는 법에 대해 알아보았습니다. 하지만 지난 시간에 맛 본 `for`문은 단순히 `range`에서 받아온 값을 그대로 사용하는 것에 그쳤습니다. 만약 `i`에 4를 대입하는 경우, 4를 그대로 사용해 일반항을 계산한 것입니다.

리스트 *list*를 사용하면 임의의 수열에 대해 일반적인 연산을 할 수 있습니다. 리스트는 일종의 (유한) 수열로 생각할 수 있는데, `for`문과 리스트를 적절히 사용하면 수열의 원소를 접근해 일반적인 계산을 수행할 수 있습니다.

지금까지는 10개의 변수를 받아 합을 반환하는 함수를 작성하기 위해 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 `ni`는 `ni`라는 이름을 가진 변수에 불과합니다. 하지만 이런 이름의 변수는 정의된 적이 없으므로 오류를 뿜어내게 됩니다.

리스트를 사용하면 임의의 개수의 변수를 (담는 리스트를) 받아 계산을 할 수 있습니다:

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

혹은 더 간단히,

```
def sum_arb(numbers):
    summ = 0
    for n in numbers:
        summ += n
    return summ
```

와 같이 바로 `numbers`에서 원소를 `n`이라는 이름으로 하나씩 꺼내올 수도 있습니다.

4.1.1 리스트의 생성

리스트는 보통 세 가지 방법으로 생성할 수 있습니다:

1. 원소들이 이미 정해진 경우,
2. 원소들의 개수가 이미 정해진 경우,
3. 임의의 원소들이 들어갈 수 있는 경우

입니다. 각 경우를 살펴봅시다.

원소들이 이미 정해진 경우

원소들이 이미 정해진 경우 리스트에 직접 원소를 입력하여 생성할 수 있습니다.

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

리스트의 값은 나중에 수정할 수 있기 때문에 처음에 채워 넣은 값들은 초기값입니다. 하지만 이렇게 값을 입력하는 것은 리스트가 담고 있는 원소의 개수가 적을 경우입니다.

원소들의 개수가 이미 정해진 경우

리스트에 담을 값이 규칙적이면서 크기가 정해져 있다면, 값을 `None`으로 초기화한 후 정해진 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 정수인 `-1`이나 문자열인 `""` 등으로도 채워넣을 수 있습니다. 단, `0`과 같은 값은 실제로 값을 채워 넣은 유의미한 `0`과 혼동될 수 있으니 유의해야 합니다.

혹은, 처음에 적응하는데 조금 시간이 걸릴 수도 있지만 리스트 표현 *list comprehension*을 사용할 수도 있습니다:

```
numbers = [None for i in range(10)]
```

위 코드에서 `i`는 사용되지 않으므로 `[None for _ in range(10)]`와 같이 다시 쓸 수도 있습니다. 이 같은 리스트 표현은 다차원 리스트를 초기화할 때 필요한데요, 이는 해당 절에서 더 자세히 확인할 수 있습니다.[1]

이렇게 만든 리스트 안의 `None`들을 대신할 값을 채워야 합니다. 바로 `for`문으로 해결할 수 있습니다. 다음은 첫 열 개의 3의 배수를 넣는 과정입니다.

```
numbers = [None] * 10

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

[1] 리스트의 `*`는 얕은 복사 *shallow copy*를 하는 방면 리스트 표현을 쓰면 복사가 아니라 각 값이 새로 만들어지기 때문입니다.

`len` 함수는 리스트의 길이를 반환합니다. 예컨대 `len(numbers)`는 `10`을 반환합니다. 보다 일반적이고 재사용 가능한 명료한 코드를 작성하기 위해서는 리스트의 길이를 숫자 `10`과 같이 직접 치는 것보다는 `len`을 사용하는 것이 좋습니다.

리스트의 원소를 접근할 때는 `lst[index]`와 같이 쓸 수 있습니다. `index`는 `0`에서 `len(lst) - 1`까지의 값을 가집니다. 즉, 리스트의 첫 번째 원소를 접근하려면 `lst[1]`이 아닌 `lst[0]`을, `n`번째 원소를 접근하려면 `lst[n]`이 아닌 `lst[n - 1]`을, 마지막 원소를 접근하려면 `lst[len(lst)]`이 아닌 `lst[len(lst) - 1]`을 사용해야 합니다. 정리하자면, `lst[n]`은 `lst`의 $n+1$번째 원소를 가리킵니다.

임의의 원소들이 들어갈 수 있는 경우

마지막으로는 리스트의 크기도 정해지지 않은 경우입니다. 물론, 이 방법은 원소들이나 개수가 정해진 경우에도 사용할 수 있는 가장 일반적인 방법입니다. 리스트의 append 메소드 *method*[2]를 사용하는 방법입니다.

```
numbers = []

for i in range(10):
    numbers.append((i + 1) * 3)
```

처음에 아무것도 담지 않은 리스트 `[]`에다가, append를 통해 새로운 원소를 뒤에 하나씩 추가하는 방법입니다. 예를 들어, `a = [1, 2, 3]`일 때 `a.append(4)`를 수행하면 `a`는 `[1, 2, 3, 4]`가 됩니다.

4.1.2 리스트의 원소와 인덱스

리스트 `lst`의 원소 `lst[index]`에서 `index`는 인덱스 *index*[3]라고 합니다. 인덱스는 `0`부터 `len(lst) - 1`까지의 값 이외에도, `-len(lst)`에서부터 `-1`까지 사용할 수 있습니다. `lst[-1]`은 `lst[len(lst) - 1]`과 같으며, `lst[-len(lst)]`은 `lst[0]`과 같습니다. 이때 `len(lst)` 이상의 값이나 `-len(lst)` 미만의 값을 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 컨테이너 *container*입니다. 리스트는 정수 혹은 실수뿐만이 아니라 문자열, 불리언, 나아가 다른 리스트들도 담을 수 있습니다. 또한 한 가지 타입의 원소들만이 아니라 여러 타입이 섞인 원소들도 한 리스트에 담을 수 있습니다. 즉, 아래와 같은 예시 모두 가능합니다.

[2] 객체에 딸린 함수를 뜻하는 말입니다. 객체 지향 프로그래밍을 배울 때 알아볼 것입니다.
[3] Index의 복수형은 indices입니다.

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

이런 리스트를 불균일 *heterogeneous*하다고 부릅니다. 하지만 이렇게 인덱스별로 다양한 타입의 값을 담고 있는 리스트는 다루기 힘들 뿐만이 아니라 (잘 짜여진 코드에서는) 많이 보기도 어렵습니다.

인덱스를 통해 리스트에 담긴 원소를 다룰 때에는 해당 원소에 대한 연산들을 그대로 사용할 수 있습니다:[4]

```
>>> 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 정수 타입이고 적절한 범위에 들어가는 값으로 계산된다면 어떤 식이든 사용할 수 있습니다:

[4] 인덱스로 접근한 이후에는 그 원소에 해당하는 변수를 사용하는 것과 다름이 없기 때문에 당연한 일입니다.

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

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

4.1.3 리스트의 조작

슬라이싱

리스트에서 유용한 연산 중에는 슬라이싱 *slicing*이 있습니다. 슬라이싱이란 리스트의 부분 리스트를 만드는 연산으로, 다음과 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]
```

l이라는 리스트가 주어졌을 때 l[i:j]는 l[i]부터 l[j - 1]까지의 원소를 뽑아 새로운 리스트를 만듭니다. 만약 i가 생략된다면 첫 원소—인덱스 0—부터 시작하며, j가 생략된다면 마지막 원소—인덱스 -1—까지 포함합니다. 둘 다 생략하는 경우 같은 원소들을 담고 있는 리스트의 복사본을 만들게 됩니다.[5]

원소들을 일정 간격으로 건너뛰면서 부분 리스트를 만들 때에도 슬라이싱을 사용할 수 있습니다. 이 경우 l[i:j:k]와 같이 슬라이싱을 하는데, k만큼 건너뛰면서 원소들을 추출합니다. range(i, j, k)에 대응된다고 생각하면 됩니다. 사실 range는 리스트처럼 값을 순회할 수 있게 하는 값을 반환하는 메소드이기 때문입니다.

[5]하지만 한 단계만 복사하는 얕은 복사입니다.

```
>>> 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]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

위에서 볼 수 있듯 `l[i:j:k]`의 `i`, `j`, `k` 중 `i`와 `j`, 혹은 둘 다 생략할 수 있습니다. `k`가 음수일 경우, 원소를 뒤에서부터 골라서 추출하게 됩니다. 특히 `k = -1`인 경우 `l[::-1]`과 같이 사용하면 `l`의 원소를 거꾸로 나열한 새로운 리스트를 만들게 됩니다.

리스트의 비교와 포함 관계 확인

두 리스트를 비교할 때는 다른 타입과 마찬가지로 `==`를 사용해 비교할 수 있습니다. 또한 특정 원소가 리스트에 해당하는지를 알아보려면 `in`을 사용하면 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
```

리스트의 연결

두 리스트를 연결 *concatenate*하고 싶은 경우 +를 사용하여 연결하고, n번 연결한 리스트를 만드려면 *를 사용하면 됩니다[6]:

```
>>> 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]
```

4.1.4 다차원 리스트

다차원 리스트는 리스트들의 리스트입니다. 특히 행렬 *matrix*을 만들 때 유용하게 사용할 수 있습니다.

행렬 M이

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

와 같을 때, 이는 다음과 같은 리스트 l로 표현할 수 있습니다.

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

만약 l의 행 크기를 알고 싶다면 len(l[0])을, 열 크기를 알고 싶다면 len(l)을 사용하면 됩니다.

각 원소를 접근할 때는, 1행 1열의 원소라면 l[0][0], 1행 2열의 원소라면 l[0][1], 2행 3열의 원소를 접근하고 싶다면 l[1][2]를 사용하면 됩니다. 일반적으로 i행 j열의 원소는 l[i - 1][j - 1]로 접근할 수 있습니다.

다차원 리스트의 경우에도 미리 원소를 지정하지 않고 None으로 초기화하여 리스트를 생성할 수 있습니다. 행 크기 height, 열 크기 width인 None으로 초기화된 리스트는 다음과 같이 생성할 수 있습니다. 아래 예시에는 i행 j열의 값을 i + 2*j + 1로 지정하는 과정까지 담고 있습니다.

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

[6] 얕은 복사를 하므로 주의해야 합니다.

```
# or table = [[None for j in range(width)] for i in range(height)]
# or just table = [[None for _ in range(width)] for _ in
↪  range(height)]

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

위 과정을 통해

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

에 대응되는 리스트 `[[1, 3, 5, 7], [2, 4, 6, 8], [3, 5, 7, 9]]`를 얻을 수 있습니다.

나아가 임의의 차원을 가지는 리스트도 만들 수 있습니다. 3차까지는 머리 속에서 시각화할 수 있는데요, 길이, 너비, 높이를 가지는 격자에 값이 놓여 있다고 생각하시면 됩니다.[7] 이 경우 삼중 루프를 써서 값을 채워 넣을 수 있습니다:

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

4.1.5 알아두면 유용한 리스트 내장 함수

파이썬에서는 리스트를 조작할 수 있는 다양한 내장 함수들을 미리 준비해 두었는데요, 다음의 연산들이 있습니다:

- `sum(l)`: `l`의 모든 원소의 합을 반환합니다.

[7] 4차도 시간에 따라 3차 격자에서 값들이 바뀌는 것으로 시각화할 수 있습니다.

- `min(l)`: `l`의 최소원을 반환합니다.
- `max(l)`: `l`의 최대원을 반환합니다.
- `l.append(x)`: `l`에 `x`를 맨 뒤에 추가합니다. `None`을 반환합니다.
- `l.count(x)`: `l`에 포함된 `x`의 개수를 반환합니다.
- `l.insert(i, x)`: `l`의 `i`번째 인덱스에 `x`를 추가합니다. `None`을 반환합니다.
- `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 뒤집습니다.
- `l.sort()`: `l`의 원소들을 정렬하며 `None`을 반환합니다.
- `reversed(l)`: `l`의 뒤집힌 버전을 반환합니다.[8] `l`은 그대로 남아 있습니다.
- `sorted(l)`: `l`의 정렬된 버전을 반환합니다. `l`은 그대로 남아 있습니다.

반환값이 `None`인 메소드와 그렇지 않은 것들을 잘 구분해야 합니다. 예컨대 `l.sort()`를 하면 `l` 자체가 정렬이 되지만, `m = sorted(l)`을 하면 `m`에 `l`의 정렬된 버전이 대입되고 `l`은 변화가 없습니다.

아래 예시를 통해 확인해봅시다:

```
>>> 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)
```

[8] 이렇게 반환된 값은 사실 리스트가 아니라 이터레이터 *iterator*인데, `list(reversed(l))`과 같이 `list` 함수로 변환하여 리스트로 변환할 수 있습니다.

```
>>> 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 = list(reversed(l))
>>> m
[3, 1, 'hi']
>>> m = l.sort()
>>> m
None
>>> l
[1, 3, 'hi']
```

4.2 문자열

지금까지 문자열은 단순히 어떤 문구를 출력하기 위해 사용하는 용도였다면, 이제는 문자열을 가지고 연산하는 법을 알아볼 것입니다.

문자열은 리스트와 비슷하게 다룰 수 있습니다. 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 있습니다.

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. in과 not in을 통해 어떤 문자열이 다른 문자열의 부분 문자열이 되는지도 판단할 수 있습니다.

```
>>> s = "Python"
>>> s[2]
't'
>>> s[1:4]
'yth'
>>> s * 3
'PythonPythonPython'
>>> s = s + "PYTHON"
>>> s
'PythonPYTHON'
>>> "Python" > "PYTHON"
True
>>> 'thon' in s
True
```

4.2.1 문자열의 불변성

리스트와의 차이라면 리스트는 가변 *mutable* 객체이고, 문자열은 불변 *immutable* 객체라는 것입니다. 이에 따라 한 문자열을 "조작"하고 싶다면 새로운 문자열을 만들어야지, 기존의 문자열을 수정하는 것은 불가능합니다.

그렇지만 +=이나 *=을 사용하는 것은 가능합니다. 이는 s += "a"는 단순히 s = s + "a"를 축약한 것으로, s의 값을 수정하는 것이 아니라 s에 새로운 값을 대입하는 것이기 때문입니다.

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

4.2.2 알아두면 유용한 문자열 내장 함수들

문자열을 더 편리하게 다루기 위한 다양한 내장 함수들도 준비되어 있습니다.

문자열 s와 t가 주어졌을 때,

- s.upper(): s의 모든 문자를 대문자로 바꾸어 반환합니다.
- s.lower(): s의 모든 문자를 소문자로 바꾸어 반환합니다.
- s.capitalize(): s의 첫 문자는 대문자로, 나머지는 소문자로 바꾸어 반환합니다.
- s.isupper(): s의 모든 문자가 대문자라면 True를, 아니라면 False를 반환합니다.
- s.islower(): s의 모든 문자가 소문자라면 True를, 아니라면 False를 반환합니다.
- s.isdigit(): s의 모든 문자가 숫자라면 True를, 아니라면 False를 반환합니다.
- s.split(): s에서 띄어쓰기 문자로 구분된 구절들을 원소로 가지는 리스트를 반환합니다.
- s.find(t): s에 t가 포함되는 경우, t가 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 = "pYtHon".upper()
>>> s
'PYTHON'
>>> s = "pYtHon".lower()
>>> s
'python'
>>> s = "pYtHon".capitalize()
>>> s
'Python'
>>> "PYTHON12".isupper()
True
>>> "pyThon12".islower()
False
>>> "1242".isdigit()
```

```
True
>>> l = "Hello World!".split()
>>> l
['Hello', 'World!']
>>> s = "Python"
>>> s.find("thon")
2
```

4.3 카운터 패턴과 휴보

리스트를 배웠기 때문에 조금 더 자유로운 반복문의 활용이 가능합니다. 이번 절에서는 카운터 패턴에 대해 집중적으로 살펴볼 것이며, hubo.cs1robots 패키지를 통해 "휴보"를 조작할 것입니다.[9]

카운터 패턴은 리스트에서 어떤 조건을 만족하는 원소의 개수를 구하는데 사용합니다. 아래는 numbers에서 7의 배수를 세는 함수 count_seven입니다.

```
def count_seven(numbers):
    cnt = 0
    for i in range(len(numbers)):
        if numbers[i] % 7 == 0:
            cnt += 1
    return cnt

num = [1, 7, 2, 4, 3, 5, 34, 12, 42, 26]
print(count_seven(num))  # 2
```

for문을 통해 각 원소에 대해 반복적인 실행을 표현하고, 내부에 if문을 두어 특정 조건을 만족하면 개수를 세는 패턴이 카운터 패턴입니다.

또한 for문으로 휴보의 움직임을 제어해볼 것입니다 (그림 4.1).

[9] 이는 카이스트에서 프로그래밍을 가르치기 위해 만들어진 패키지입니다.

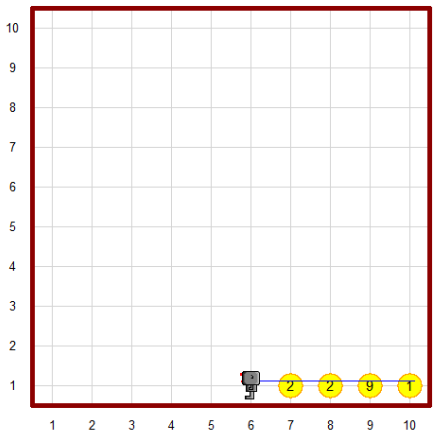


그림 4.1: 휴보를 원하는대로 조작하는 법을 알아볼 것입니다.

휴보는 전진, 좌회전, 우회전에 해당하는 `move()`, `turn_left()`, `turn_right()` 등의 동작을 취할 수 있습니다.

`hubo` 패키지는 pip으로 설치할 수 있습니다. pip을 사용하면 PyPI에 있는 패키지들을 손쉽게 설치할 수 있습니다:

```
pip install hubo
```

4.3.1 카운터 패턴

카운터 패턴은 다음의 같은 형식을 따릅니다.

```
cnt = 0
for i in range(len(lst)):
    if cond(lst[i]):
        cnt += 1
```

리스트 `lst`가 주어졌을 때, 해당 리스트의 모든 원소에 대해(줄 2) 어떤 조건 `cond`을 `lst[i]`가 만족한다면(줄 3) `cnt`의 값을 1씩 더하는(줄 4) 것입니다.

위에서 보았던 7의 배수의 개수를 세는 예시도 마찬가지입니다. `numbers[i] % 7 == 0`의 조건이었는데, 이를 사용해 7의 배수이면 `True`를, 아니면 `False`를 반환하는 함수 `is_mult_7`로 만든다면 정의한다면 `is_mult_7(lst[i])`의 형태로 쓸 수 있습니다. 불리언 함수를 배울 때 언급하였듯, 조건이 복잡한 경우에는 함수를 정의하는 방식으로 코드를 작성해야 합니다. 소수의 개수를 세는 카운터 패턴의 경우, 소수를 판별하는 불리언 함수 `is_prime`으로 `if is_prime(lst[i]):`

와 같이 사용하면 됩니다. 주어진 리스트에서 소수의 개수를 세는 함수가 아래 예시에 나와 있습니다.

```
def is_prime(p):
    """A naive implementation"""
    for i in range(2, p // 2):
        if p % i == 0:
            return False
    return True

def count_primes(numbers):
    cnt = 0
    for i in range(len(numbers)):
        if is_prime(numbers[i]):
            cnt += 1
    return cnt

num = [217, 287, 181, 143, 163, 319, 233, 399, 203]
print(count_primes(num))  # 3
```

4.3.2 휴보

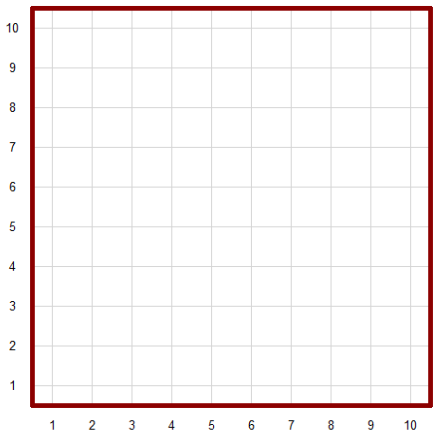


그림 4.2: 휴보가 움직일 빈 세상입니다.

지금까지 배운 함수, 조건문, 루프, 리스트 등의 개념을 모두 활용하여 휴보를 움직이게 할 것입니다. 시작하기 위해 다음과 같은 코드를 작성해봅시다:

```
from hubo.cs1robots import *
create_world()
```

이를 실행시키면 그림 4.2와 같은 창이 뜹니다. 이것이 앞으로 휴보가 움직일 세상입니다. 세상은 기본적으로 가로 세로로 10칸의 격자입니다.

이제 `hubo = Robot()`을 실행하면, 그림 4.3과 같이 휴보가 위치 $(1, 1)$에 생성됩니다.

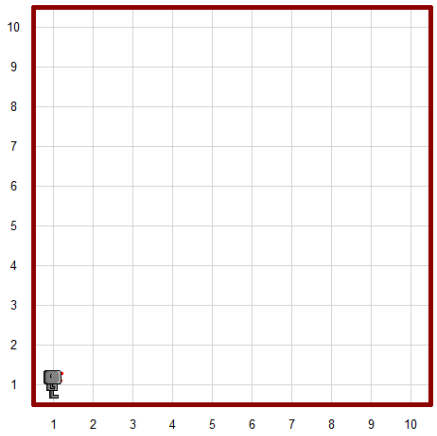


그림 4.3: 휴보가 생성된 모습입니다.

휴보가 한 칸 전진하기 위해서는 `hubo.move()`를 실행하고, 90도 좌회전하기 위해서는 `hubo.turn_left()`를, 90도 우회전하기 위해서는 `hubo.turn_right()`를 실행하면 됩니다. 그림 4.4에 휴보를 좌회전시킨 후 한 칸 전진한 후의 모습이 나와 있습니다.

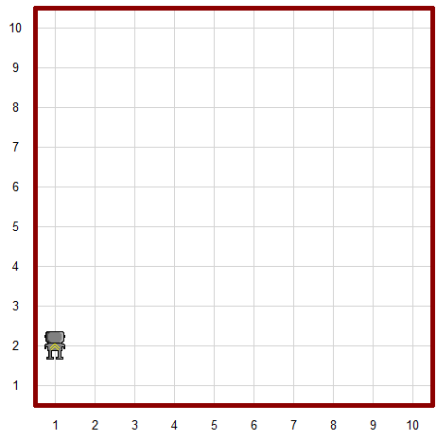


그림 4.4: 휴보가 좌회전한 후 한 칸 전진한 모습입니다.

휴보가 이동한 경로를 가시화하고 싶다면 hubo.set_trace("blue")와 같이 set_trace 메소드에 인자로 색상을 문자열로 입력하면 됩니다. 또한 휴보가 느리게 움직이도록 하려면 hubo.set_pause(0.5)와 같이 매 명령마다 정지할 시간을 초 단위로 넘겨주면 됩니다.

휴보가 한 변의 길이가 2인 정사각형의 경로를 따라 이동하는 코드가 아래에 나타나 있습니다.

```
from hubo.cs1robots import *
create_world()

hubo = Robot()
hubo.set_trace("blue")
hubo.set_pause(0.5)

def move_square():
    for i in range(4):
        hubo.move()
        hubo.move()
        hubo.turn_left()

move_square()
```

이를 실행한 결과는 그림 4.5입니다.

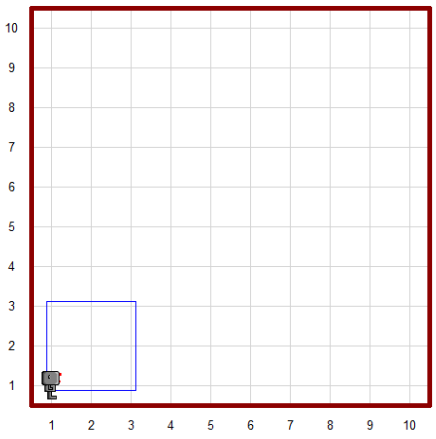


그림 4.5: 휴보가 한 변의 길이가 2인 정사각형의 경로를 따라 이동한 모습입니다.

세상에는 벽도 추가할 수 있습니다. 그림 4.6에 계단형 지형을 만든 세상을 볼 수 있습니다. 여기서 나타난 움직임을 구현하는 코드는 아래와 같습니다.

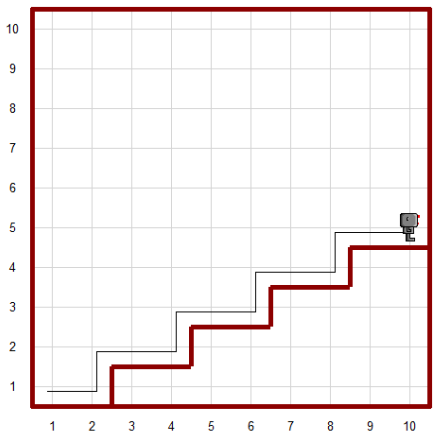


그림 4.6: 계단형 지형입니다.

```
hubo.move()
for _ in range(4):
    hubo.turn_left()
    hubo.move()
    hubo.turn_right()
    hubo.move()
    hubo.move()
```

fot문에서 사용한 변수 이름이 _인 것은, 해당 값을 실제로 사용하지 않기 때문에 저렇게 놓은 것입니다. 기억한다면 _는 가능한 변수 이름에 해당하고, 파이썬 프로그래머들은 보통 사용되지 않는 변수 이름으로 _를 자주 사용합니다.

4.4 예제

1. 리스트의 원소들의 제곱의 합을 반환하는 함수 sum_squares를 작성하세요.

```
def sum_squares(a):
    # Add here!

print(sum_squares([3, 5, 4]))  # 50
print(sum_squares([2, 5, 4, 0, 1, -1, 5, 1]))  # 73
```

2. 길이 n인 정수 리스트 a와 정수 x를 받아

$$\sum_{i=0}^{n-1} a[i] \cdot x^i$$

을 계산하는 함수 `compute_polynomial(a, x)`를 작성하세요.

```
def compute_polynomial(a, x):
    # Add here!

print(compute_polynomial([3, 5, 4], 5))  # 128
print(compute_polynomial([2, 0, 4, 0, 1, -1, 5, 1], 3))  # 5708
```

3. 리스트 a의 조화 평균을 계산하는 함수 harmonic_mean을 작성하세요. $a_1, a_2, \ldots, a_n$의 조화 평균은

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{a_i}}$$

와 같이 계산됩니다.

또, a의 기하 평균을 계산하는 함수 geometric_mean을 작성하세요. $a_1, a_2, \ldots, a_n$의 기하 평균은

$$\sqrt[n]{\prod_{i=1}^{n} a_i}$$

입니다.

```
def harmonic_mean(a):
    # Add here!

def geometric_mean(a):
    # Add here!

numbers = [2, 4, 3, 10, 7, 2, 5, 6]
print(harmonic_mean(numbers))  # 3.64820846906
print(geometric_mean(numbers))  # 4.22116731332
```

4. 반복문을 사용해서 주어진 리스트 a의 뒤집힌 리스트를 반환하는 함수 `reversed(a)`을 작성하세요.

```
def reversed(a):
    n = len(a)
    b = [None] * n
    # Add here!

print(reversed([3, 1, 5, 2, 4]))  # [4, 2, 5, 1, 3]
print(reversed([7, 6, 3, 1, 5, 8, 2, 4]))  # [4, 2, 8, 1, 3, 6,
↪  7]
```

5. 첫 n개의 피보나치 수들의 리스트를 반환하는 fibonacci(n)을 작성하세요. n은 양의 정수라고 가정하고, 피보나치 수열의 첫 두 값은 1입니다.

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

print(fibonacci(10))  # [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

6. 좌표 평면 상의 점들의 리스트 points가 주어졌을 때, 이 점들이 구성하는 다각형의 면적을 계산하는 함수 area(points)을 작성하세요. points는 [3, 4]처럼 x 좌표와 y 좌표의 성분이 리스트로 표현된 원소들의 리스트입니다. 예를 들어 그림 4.7을 나타내는 points는 [[3, 1], [6, 3], [4, 4], [7, 6], [2, 7], [0, 5], [2, 3], [1, 2]]입니다.

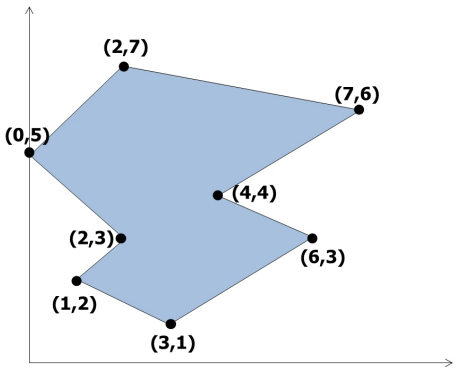


그림 4.7: 문제 6의 예시 다각형입니다.

일반적으로 점 (x_i, y_i) $(i = 0, 1, \ldots, n-1)$들이 구성하는 다각형의 면적은 다음의 "신발끈" 공식으로 계산할 수 있습니다:

$$\frac{1}{2}\left|\sum_{i=0}^{n-1} x_i \left(y_{(i+1) \mod n} - y_{(i-1) \mod n}\right)\right|$$

또, 이 점들이 구성하는 다각형의 둘레를 계산하는 `perimeter(points)`도 작성하세요.

`area`와 `perimeter`를 작성하기 위해 다른 함수들을 작성할 수 있습니다.

```
# Add here!

def area(points):
    # Add here!

def perimeter(points):
    # Add here!

points = [[3, 1], [6, 3], [4, 4], [7, 6], [2, 7], [0, 5], [2,
↪  3], [1, 2]]
print(area(points))  # 22.0
print(perimeter(points))  # 23.8533258314
```

7. 리스트 `a`의 최소원을 구하는 함수 `find_min(a)`을 작성하세요.

```
def find_min(a):
    # Add here!

print(find_min([7.2, 5, 21, -1, 4, 0.4]))  # -1
```

8. `points`의 가장 가까운 두 점의 길이를 반환하는 함수 `closest_pair(points)`을 작성하세요.

```
# Add here!

print(closest_pair([[4, -4], [7, 5], [2, 1], [-2, -1], [-3,
↪  5]]))  # 4.472135955
```

9. 리스트 `numbers`가 주어졌을 때, `lower` 이상 `upper` 이하의 값들의 개수를 세는 함수 `count_range(numbers, lower, upper)`를 작성하세요.

```
def count_range(numbers, lower, upper):
    # Add here!
```

```
print(count_range([8, 9, 10, 2, 4, 5, 9, 7, 2, 3, 7], 3, 7))  #
↪ 5
```

10. $x^2+y^2 \leq \text{radius}^2$을 만족하는 점 (x, y)들의 개수를 세는 함수 count_within_circle(points, radius)를 작성하세요.

```
def count_within_circle(points, radius):
    # Add here!

points = [[2, 1], [7, 5], [-5, 2], [-3, 5], [-7, 4], [-2, -1],
↪ [-2, -4], [-4, -2], [-6, -4], [4, -4], [6, -2]]
print(count_within_circle(points, 3))  # 2
print(count_within_circle(points, 5))  # 4
print(count_within_circle(points, 8))  # 9
```

11. top, bottom, left, right으로 표현되는 직사각형의 내부 혹은 경계 위에 (x, y)가 포함되는지 확인하는 함수 within_rect(top, bottom, left, right, x, y)를 작성하세요. 예를 들어서 (top, bottom, left, right) = $(2, -4, -5, 6)$으로 표현되는 직사각형은 그림 4.8과 같습니다.

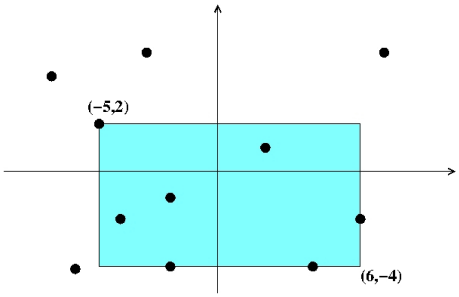


그림 4.8: 문제 11의 직사각형입니다.

```
def within_rect(top, bottom, left, right, x, y):
    # Add here!

print(within_rect(2, -4, -5, 6, -5, 2))  # True
print(within_rect(2, -4, -5, 6, 6, -1))  # True
print(within_rect(2, -4, -5, 6, 0, 1))  # True
print(within_rect(2, -4, -5, 6, -6, 0))  # False
print(within_rect(2, -4, -5, 6, 0, 3))  # False
```

이제 위의 within_rect 함수를 사용해서 points의 점들 중 직사각형에 포함되는 (경계 포함) 것들의 개수를 세는 함수 count_within_rect(top, bottom, left, right, points)를 작성하세요.

```
def count_within_rect(top, bottom, left, right, points):
    # Add here!

points = [[2, 1], [7, 5], [-5, 2], [-3, 5], [-7, 4], [-2, -1],
↪  [-2, -4], [-4, -2], [-6, -4], [4, -4], [6, -2]]
print(count_within_rect(2,-4,-5,6, points))  # 7
```

---

12. years에서 윤년의 개수를 세는 함수 count_leap_year(numbers)를 작성하세요.

```
def count_leap_years(numbers):
    # Add here!

print(count_leap_years([2008, 2011, 2012, 2000]))  # 3
print(count_leap_years([2100, 2300, 2400, 2200]))  # 1
```

---

13. numbers에서 합성수의 개수를 세는 함수 count_composite(numbers)를 작성하세요.

```
# Add here!

def count_composite(numbers):
    # Add here!

numbers = [217, 287, 181, 143, 163, 319, 233, 399, 203]
print(count_composite(numbers))  # 6
```

Chapter 5

한정자와 While문

There exists, for everyone, a sentence—a series of words—that has the power to destroy you. Another sentence exists, another series of words, that could heal you. If you're lucky you will get the second, but you can be certain of getting the first.

— Philip K. Dick, *VALIS*

5.1 한정자

지금까지 개수 세기, 최소/최대값 구하기 등의 기본적인 반복문 패턴을 알아보았습니다. 이번에는 한정자 *quantifier* 패턴을 알아봅니다.

수학에서 사용하는 전체 한정자 *universal quantifier* ∀와 존재 한정자 *existential quantifier* ∃를 프로그램으로 작성한 것을 한정자 패턴이라고 합니다. 한정자 패턴을 통해 어떤 집합 S가 주어졌을 때,

- $\forall x \in S.\ p(x)$
- $\exists x \in S.\ p(x)$

의 참/거짓을 판별할 수 있습니다.

또한 지난 단원에서 `hubo.cs1robots` 패키지를 통해 휴보를 원하는대로 움직이도록 코드를 작성했습니다. 이번에는 휴보가 줍고 놓을 수 있는 비퍼 *beeper*를 다룰 것입니다. 비퍼는 격자마다 배치할 수 있는데, 휴보는 해당 격자로 이동해야만 비퍼를 세거나 놓고 주울 수 있습니다. 그림 5.1처럼 생긴 지형에서 놓고 주울 수 있는 `drop_beeper()`와 `pick_beeper()` 메소드 등에 대해서 알아봅니다.

표 5.1에서 리스트`numbers`의 원소가 모두 양인 경우, 일부만 양인 경우, 그리고 양의 원소가 없는 경우에 대해, 한정자가 적용된 식의 값들이 정리되어 있습니다. 전체 한정자의 경우 `numbers`의 모든 원소가 양수일 경우에만 `True` 값을 가지게

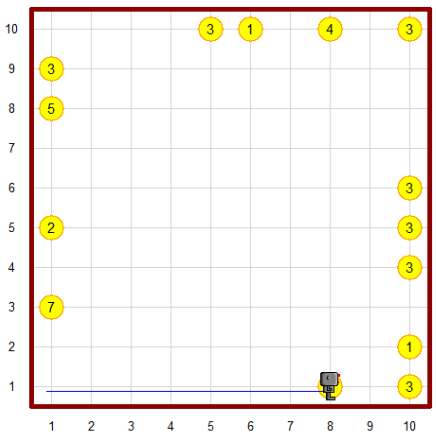


그림 5.1: 비퍼가 놓인 지형입니다.

표 5.1: 한정자가 사용된 예시입니다.

numbers	∀i. numbers[i] > 0	∃i. numbers[i] > 0
[1, 3, 2, 5, 2]	True	True
[-1, -3, -2, -5, -2]	False	False
[-1, -3, -2, 5, 2]	False	True

하며, 음의 원소가 하나라도 있을 경우에는 `False` 값을 가지게 합니다. 반면 존재 한정자의 경우 `numbers`의 원소 가운데 양인 것이 하나라도 있다면 `True`입니다.

한정자를 다루는 것은 곧 명제의 진리값을 구하는 것이기 때문에 불리언 함수로 계산할 수 있습니다. 또한 리스트의 각 원소를 확인해야 하므로 `for` 반복문을 사용해야 합니다. 전체 한정자의 경우 모든 원소가 어떤 조건을 만족하는지를 확인해야 합니다. 이는 거꾸로 조건을 만족시키지 못하는 단 하나의 원소라도 있다면 `False`를 반환하는 불리언 함수를 만들면 됩니다. `n = len(numbers)`이면

$$\texttt{∀i ∈ range(n)}.\ p(\texttt{numbers[i]}) \Leftrightarrow \neg\,(\texttt{∃i ∈ range(n)}.\ \neg p(\texttt{numbers[i]}))$$

이기 때문입니다. 즉, 모든 원소를 조건문으로 확인하는 `for` 문에서 조건을 만족하지 못하면 바로 `False`를 반환하고, 루프를 통과하면 `True`를 반환하면 됩니다.

아래는 모든 원소가 양수인지를 확인하는 함수 `all_positive`입니다.

```
def all_positive(numbers):
    for i in range(len(numbers)):
        if not numbers[i] > 0:
            return False
    return True
```

소수를 판별하는 함수의 구현도 전체 한정자를 사용한 경우입니다. 1과 자기 자신이 아닌 약수가 단 하나라도 존재한다면 `False`를 반환하는 경우이기 때문입니다. 산술 기하 평균에 의해 해당 수의 절반까지의 루프를 구성했는데, 제곱근까지의 범위를 구성하여도 됩니다.

존재 한정자의 경우 조건을 만족시키는 단 하나의 원소라도 찾으면 됩니다. 이에 따라, 모든 원소를 조건문으로 확인하는 `for` 문에서 조건을 만족하는 원소를 발견하는 즉시 `True`를 반환하고, 반복문을 통과하면 `False`를 반환하면 됩니다.

아래는 하나의 원소라도 양수인지를 확인하는 함수 `some_positive`입니다.

```
def some_positive(numbers):
    for i in range(len(numbers)):
        if numbers[i] > 0:
            return True
    return False
```

5.1.1 예시

다음은 어떤 정수 `n`이 두 정수의 제곱의 합인지의 여부를 알려주는 함수 `sum_squares`입니다.

```
def sum_squares(n):
    sqrt_n = int(n ** .5)
    for x in range(1, sqrt_n + 1):
        for y in range(x, sqrt_n + 1):
            if n == x ** 2 + y ** 2:
                return True
    return False
```

만약 $x^2 + y^2 = n^2$이라면 $x, y \leq \sqrt{n}$일 것이므로 `sqrt_n = int(n ** .5)`로 정의하여 줄 3, 4에서 `for` 반복문의 범위를 제한하였습니다. 나아가 줄 4에서는 $x \leq y$의 조건을 주어 계산할 값들의 범위를 줄였습니다.

아래는 주어진 리스트 `numbers`의 모든 원소가 두 정수의 제곱의 합인지를 판단하는 함수 `all_sum_squares`입니다. 위에서 구한 불리언 함수 `sum_squares`를 이용한 전체 한정자 패턴입니다.

```
def all_sum_squares(numbers):
    for i in range(len(numbers)):
        if not sum_squares(numbers[i]):
```

```
            return False
    return True
```

마지막으로 주어진 수열이 증가 수열인지를 확인하는 함수 inc_seq입니다.

```
def inc_seq(numbers):
    for i in range(len(numbers) - 1):
        if not numbers[i] <= numbers[i + 1]:
            return False
    return True
```

전체 한정자 패턴과 동일하지만, 주의할 점은 줄 2에서 range의 범위를 len(numbers) - 1로 해야 IndexError 예외가 나지 않습니다. 이는 줄 3에서 다음 인덱스의 원소와 현재 인덱스의 원소를 비교하기 때문입니다.

지금까지 배운 for 반복문의 패턴 세 가지를 아래에 정리합니다.

1. 최대/최소 구하기

```
def maximum(numbers):  # def minimum(numbers):
    m = numbers[0]
    for i in range(len(numbers)):
        if numbers[i] > m:  # if numbers[i] < m:
            m = numbers[i]
    return m
```

2. 카운터 패턴

```
def cnt_odd(numbers):
    cnt = 0
    for i in range(len(numbers)):
        if numbers[i] % 2:
            cnt += 1
    return cnt
```

3. 한정자 패턴

```
def all_odd(numbers):  # def some_odd(numbers):
    for i in range(len(numbers)):
```

```
        if not numbers[i] % 2:  # if numbers[i] % 2:
            return False  # return True
    return True  # return False
```

5.1.2 비퍼

비퍼는 휴보가 놓고 주울 수 있는 물체로, 격자마다 위치할 수 있습니다. 휴보가 비퍼를 놓기 위해서는 drop_beeper() 메소드를 사용하면 됩니다. 이때 휴보는 on_beeper() 메소드로 비퍼를 감지할 수 있습니다. 만약 격자점에 비퍼가 놓여 있었다면, pick_beeper()로 주우면 됩니다. 이렇게 주운 비퍼는 휴보가 가지고 있다가 다른 격자에 다시 놓을 수 있습니다. 만약 비퍼가 놓이지 않은 격자에서 비퍼를 주우려 하면 에러가 발생합니다. 이에 따라 줍기 전에 항상 on_beeper()를 통해 현재 휴보가 위치한 격자에 비퍼가 있는지 확인을 해야 합니다. 아래는 on_beeper()를 통해 비퍼가 있는지 확인한 후, 있다면 pick_beeper()를 통해 비퍼를 줍는 코드입니다. 그림 5.2에 실행 전후 결과가 나타나 있습니다.

```
for i in range(9):
    hubo.move()
    if hubo.on_beeper():
        hubo.pick_beeper()
```

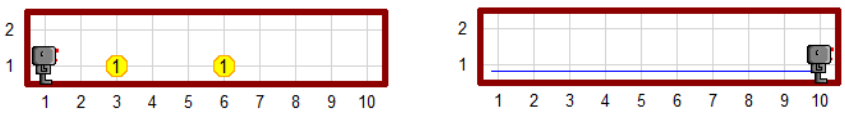


그림 5.2: 휴보가 비퍼를 반복해서 줍는 모습입니다.

휴보가 처음에 가지고 있는 비퍼의 개수를 설정할 수도 있습니다. 지금까지는 단순히 Robot()으로 휴보를 생성하였지만, Robot(beepers = 1)과 같이 인자를 전달해주어 초기에 가지고 있는 비퍼의 개수를 지정할 수 있습니다.

drop_beeper()를 통해 비퍼를 가지고 있는 것보다 많이 놓으려 하면 에러가 발생하기 때문에, carries_beeper()로 비퍼를 가지고 있는지 확인을 한 후 놓을 수 있습니다. carries_beeper() 메소드는 휴보가 현재 비퍼를 가지고 있는지 여부를 확인하는 불리언 함수입니다. 아래는 beepers = 6 인자를 통해 휴보가 처음에 여섯 개의 비퍼를 가지고 시작하도록 한 뒤, carries_beeper()를 통해 비퍼가 남아 있다면 하나씩 떨어뜨리면서 전진하도록 하는 코드입니다. 그림 5.3에 실행 전후 모습이 나타나 있습니다.

```
hubo = Robot(beepers = 6)

for i in range(9):
    hubo.move()
    if hubo.carries_beeper():
        hubo.drop_beeper()
```

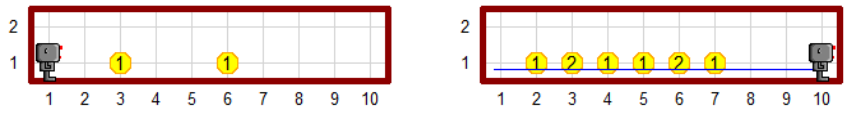


그림 5.3: 휴보가 비퍼를 하나씩 놓으면서 이동하는 모습입니다.

5.2 While문

그런데 어떤 격자에 있는 비퍼를 모두 주우려면 어떻게 해야 할까요? 반복적인 행동을 취해야 하므로 for문을 활용해야 할 것 같은데, 매 격자점마다 몇 번 반복할지 정해지지 않았기 때문입니다. 이런 경우에 사용하는 것이 바로 while문입니다.

while문은 for문과 같은 반복문의 한 종류입니다. for문은 명시적으로 몇 회 반복해야 하는지가 주어지지만, while문은 어떤 조건을 만족한다면 계속 명령을 반복하도록 하는 반복문입니다.

5.2.1 While문과 휴보

아래는 지난 절에서 beeper를 줍는 pick_beeper() 메소드를 소개할 때 사용하였던 코드입니다.

```
for i in range(9):
    hubo.move()
    if hubo.on_beeper():
        hubo.pick_beeper()
```

그림 5.2에 실행 결과가 담겨 있었습니다.

각 격자점에 하나 이상의 비퍼가 놓여있는 경우에는, 아래와 같이 if를 while로 바꾸면 됩니다.

```
for i in range(9):
    hubo.move()
```

```
    while hubo.on_beeper():
        hubo.pick_beeper()
```

그림 5.4에 결과가 나타나 있습니다. 즉, while문은 if문을 여러 번 반복한 것과

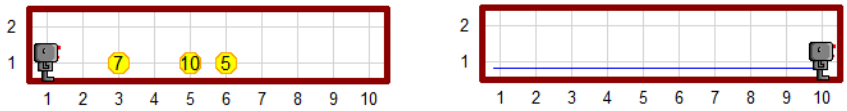


그림 5.4: 비퍼가 한 격자에 여러 개 있는 경우 휴보가 비퍼를 반복해서 줍는 모습입니다.

동일한 것을 알 수 있습니다.

마찬가지로 비퍼를 놓을 때도 놓을 수 있는 만큼 최대한 놓으라는 명령을 while문으로 작성할 수 있습니다.

```
hubo = Robot(beepers = 50)

while hubo.carries_beepers():
    hubo.drop_beeper()
hubo.move()
```

if문에서 조건이 False이면 아예 실행이 되지 않는 것처럼 while문의 조건도 False로 시작될 경우 반복문이 실행되지 않고 넘어가게 됩니다. 따라서 휴보가 비퍼를 가지고 있지 않은 경우에는 drop_beeper()가 실행되지 않습니다.

while 문을 활용해서 비퍼가 있는 곳까지 이동한 후 모두 주워서 돌아오라는 코드는 지금까지 배운 반복문을 활용하여 아래와 같이 작성할 수 있습니다.

```
dist = 0

while not hubo.on_beeper():
    hubo.move()
    dist += 1

while hubo.on_beeper():
    hubo.pick_beeper()

hubo.turn_right()
hubo.turn_right()
```

```
for i in range(dist):
    hubo.move()
```

줄 1에서 4까지는 while 문에서 카운터 패턴이 적용된 것을 볼 수 있습니다. 이는 나중에 휴보가 돌아올 때 몇 칸을 move()해야 할지 세야 하기 때문입니다. 이렇게 센 dist 값은 줄 13의 for 문에서 사용되었습니다. 줄 3의 while 문 조건의 경우, 휴보가 비퍼 위에 있지 않다면 앞으로 계속 이동하라는 줄 4의 명령을 수행하게 합니다. 그리고 해당 위치까지 이동해서 반복문을 탈출했을 때, 줄 7의 반복문이 수행되어 비퍼가 더 이상 없을 때까지 줍게 됩니다(줄 8). 마지막으로는 줄 13에서 이동한 칸 수만큼 move()하여 되돌아옵니다.

5.3 예제

1. 정수 n이 세 양의 정수의 제곱의 합으로 나타내어질 수 있을 때 True를 반환하는 함수 sum_three_squares(n)를 작성하세요. 예를 들어, 38은 $2^2+3^2+5^2$ 이므로 True를 반환해야 합니다.

```
def sum_three_squares(n):
    m = int(n ** .5)
    # Add here!

for n in range(20, 31):
    print(n, sum_three_squares(n))

"""
20 False
21 True
22 True
23 False
24 True
25 False
26 True
27 True
28 False
29 True
30 True
"""
```

2. 정수 n이 서로 다른 세 양의 정수의 제곱의 합으로 나타내어질 수 있을 때 True를 반환하는 함수 sum_three_squares(n)를 작성하세요.

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

for n in range(20, 31):
    print n, sum_three_dist_squares(n)

"""
20 False
21 True
22 False
23 False
24 False
25 False
26 True
27 False
28 False
29 True
30 True
"""
```

3. 정수 n이 두 소수의 합으로 나타내어질 수 있을 때 True를 반환하는 함수 sum_two_primes(n)를 작성하세요.

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

for n in range(20, 31):
    print(n, sum_two_primes(n))

"""
20 True
21 True
22 True
23 False
24 True
25 True
```

```
26 True
27 False
28 True
29 False
30 True
"""
```

4. 정수 n이 두 소수의 제곱의 합으로 나타내어질 수 있을 때 True를 반환하는 함수 sum_two_prime_squares(n)를 작성하세요.

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

for n in range(50, 61):
    print(n, sum_two_prime_squares(n))

"""
50 True
51 False
52 False
53 True
54 False
55 False
56 False
57 False
58 True
59 False
60 False
"""
```

5. numbers에 소수가 포함되어 있을 때만 True를 반환하는 함수 some_prime(numbers)를 작성하세요. 또한 numbers의 모든 수가 소수일 때만 True를 반환하는 함수 all_primes(numbers)를 작성하세요.

```
def some_prime(numbers):
    # Add here!

def all_prime(numbers):
    # Add here!
```

```
num1 = [217, 287, 143, 163, 319]
num2 = [217, 287, 143, 169, 319]
num3 = [223, 281, 227, 151, 149]
print(some_prime(num1), all_prime(num1))  # True False
print(some_prime(num2), all_prime(num2))  # False False
print(some_prime(num3), all_prime(num3))  # True True
```

6. numbers에 속한 수들이 모두 다를 때에만 True를 반환하는 함수 all_distinct(numbers)를 작성하세요.

```
def all_distinct(numbers):
    # Add here!

print(all_distinct([1, 3, 2, 5, 2, 1]))  # False
print(all_distinct([1, 0, 2, 5, 3, 4]))  # True
```

7. numbers에 속한 모든 수들이 lower $\leq n \leq$ upper를 만족할 때 True를 반환하는 함수 all_within_range(numbers, lower, upper)를 작성하세요.

```
def all_within_range(numbers, lower, upper):
    # Add here!

print(all_within_range([1, 0, 2, 6, 3, 4], 0, 5))  # False
print(all_within_range([1, 0, 2, 5, 3, 4], 0, 5))  # True
```

8. 위 두 문제에서 작성한 all_distinct와 all_within_range를 사용해서 numbers가 순열 *permutation*일 때 True를 반환하는 함수 is_permutation(numbers)를 작성하세요. 정수 수열 $a_0, a_1, \ldots, a_{n-1}$에 속한 모든 수가 다르고 $i = 0, 1, \ldots, n-1$에 대해 $0 \leq n_i \leq n-1$이면 수열 a_i는 순열입니다..

```
def is_permutation(numbers, lower, upper):
    # Add here!

print(is_permutation([1, 3, 2, 5, 2, 1]))  # False
print(is_permutation([1, 0, 2, 5, 3, 4]))  # True
print(is_permutation([1, 0, 2, 6, 3, 4]))  # False
```

9. 십진수로 나타낸 정수 n의 표현에서 7을 세는 함수 count_sevens(n)를 작성하세요.

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

print(count_sevens(1723))  # 1
print(count_sevens(1357924770))  # 3
```

10. 두 양의 정수 a와 b의 최대 공약수를 계산하는 gcd(a, b)를 작성하세요.

```
def gcd(a, b):
    if a < b:
        a, b = b, a

    while b > 0:
        # Add here!

    return # Add here!

print(gcd(36, 20))  # 4
print(gcd(2408208, 2790876))  # 132
```

Chapter 6

반복문의 응용과 파일 입출력

> *A program without a loop and a structured variable isn't worth writing.*
>
> — Alan Perlis, *Epigrams in Programming*

지금까지 반복문의 다양한 패턴을 알아보았으니, 이를 정리하고 `for`문을 구성하는 새로운 방법과 `break`, `continue`으로 반복문을 직접 제어하는 방법에 대해서 알아볼 것입니다.

또한 텍스트 파일을 읽고 출력하는 파일 입출력에 대해서 간단히 살펴볼 것이며, 이를 통해 많은 양의 데이터를 처리할 수 있는 길이 열리게 됩니다.

6.1 반복문의 응용

6.1.1 리스트 원소의 직접 접근

지금까지는 리스트 `l`이 주어졌을 때 `range(len(l))`을 사용하여 다음과 같이 `for` 문을 구성하였습니다.

```
l = [2, 4, 1, 7, 3]
for i in range(len(l)):
    print(l[i])
```

그러나 리스트의 원소를 인덱스를 통해 접근하지 않고, 직접 접근할 수도 있습니다.

```
l = [2, 4, 1, 7, 3]
for e in l:
    print(e)
```

수학적으로 표현하자면, 전자는 $\sum_{0 \le i < |l|}$ `l[i]`에, 후자는 $\sum_{e \in l}$ `e`에 대응됩니다. 이러한 두 가지 방식은 각각 장단점이 있습니다. 리스트의 원소를 인덱스로 접근할 때에는 해당 원소의 메모리 상 위치를 참조하기 때문에 리스트의 원소를 수정할 수 있습니다. 또한, 리스트의 단조증가 여부 등을 알기 위해서는 `l[i]`, `l[i + 1]`와 같이 전후 원소를 비교해야 하며, 이처럼 순서가 중요한 경우 인덱스로 접근하는 것이 바람직합니다.

리스트의 원소를 직접 접근할 때에는 해당 원소의 값을 가지는 변수—위의 예시에서는 `e`—를 새로 생성하기 때문에, 리스트를 수정하는 것이 아니라 해당 지역 변수의 `e` 값만 수정이 됩니다. 따라서 다음에 나오는 두 코드는 다른 결과를 내놓게 됩니다.

```
l = [2, 4, 1, 7, 3]
for i in range(len(l)):
    l[i] += 1

print(l)  # [3, 5, 2, 8, 4]
```

```
l = [2, 4, 1, 7, 3]
for e in l:
    e += 1

print(l)  # [2, 4, 1, 7, 3]
```

파이썬에서는 이 둘을 모두 사용할 수 있게 `enumerate` 함수를 제공합니다:

```
l = [2, 4, 1, 7, 3]
for i, e in enumerate(l):
    l[i] += e

print(l)  # [4, 8, 2, 14, 6]
```

6.1.2 break와 continue

특정 상황에서 반복문을 중단하거나, 바로 다음 반복문 실행으로 넘어가고 싶다면 `break`와 `continue`를 활용할 수 있습니다. 아래에 `break`와 `continue`을 사용한 예시가 나와 있습니다.

```
for i in range(8):
    if i == 5:
        break
    print(i, end=" ")

# 0 1 2 3 4
```

```
for i in range(8):
    if i == [3, 5]:
        continue
    print(i, end=" ")

# 0 1 2 4 6 7
```

break와 continue는 while문과 결합하면 효과적으로 사용할 수 있습니다. 일반적으로 while문은 어떤 조건을 만족할 때 종료가 되는데, 반복문 내부에서도 마치고 싶을 때가 있기 때문입니다.

다중 반복문

다중 반복문에 break와 continue는 가장 가까이 있는 반복문만을 종료하거나 넘기게 됩니다. 다중 반복문에서의 break와 continue을 나타낸 예시입니다:

```
i = 0
while i < 5:
    j = 0
    while j < 5:
        if i > 0:
            break
        print(j, end=" ")
        j += 1
    print(i, end=" ")
    i += 1

# 0 1 2 3 4 0 1 2 3 4
```

```
i = 0
while i < 5:
    j = 0
    while j < 5:
        if i > 0:
            continue
        print(j, end=" ")
        j += 1
    print(i, end=" ")
    i += 1

# infinite loop
```

```
for i in range(5):
    if i > 0:
        continue
    j = 0
    while j < 5:
        if i > 0:
            break
        print(j, end=" ")
        j += 1
    print(i, end=" ")

# 0 1 2 3 4 0
```

6.2 파일 입출력

많은 양의 자료를 다뤄야 하는 경우, 외부 파일에서 데이터를 읽고 쓰는 것이 필요합니다.

6.2.1 파일 입력

아래와 같이 파일에서 문자열을 읽어올 수 있습니다.

```
fin = open("input.txt", "r")
content = fin.read()
```

```
words = content.split()
fin.close()
```

첫 번째 줄에서 `open("input.txt", "r")`을 통하여 `input.txt` 파일을 읽기 전용(`"r"`)으로 열어 `fin`에 할당하게 됩니다. 두 번째 줄에서 이 파일을 문자열로 읽어 `content`에 저장한 후, 세 번째 줄에서는 `content`의 문자열을 단어별로 끊어 `words`라는 리스트에 저장을 하게 됩니다. 마지막 줄에서는 이렇게 열린 파일을 닫습니다. 한 번 연 파일을 닫지 않는 것은 리소스 낭비로 이어지기 때문에 반드시 닫아주어야 합니다.[1]

이를 활용하여 `dictionary.txt` 파일의 가장 긴 단어를 최대/최소 패턴을 통해 다음과 같이 찾을 수 있습니다.

```
def longest_word(filename):
    f = open(filename, "r")
    words = f.read().split()
    f.close()

    max_word = None
    max_len = -1

    for word in words:
        if max_len < len(word):
            max_word = word
            max_len = len(word)

    return maxword
```

이를 쓰면 `dictionary.txt`에서 pneumonoultramicroscopicsilicovolcanoconiosis라는 단어를 찾아낼 수 있습니다.

이번에는 파일에서 문자열을 읽어들여 처리를 하는 다양한 방법에 대해서 알아보고, 파일에 문자열을 입력하는 법도 알아볼 것입니다. 어떤 문자열에 `strip()` 메소드를 취하면 문자열 양 끝에 있는 공백 문자들을 없앤 새로운 문자열을 반환합니다. 또한, `split()` 메소드는 공백으로 구분된 단어들의 리스트를 반환합니다.

```
f = open("./input.txt", "r")
print(type(f))  # <type 'file'>
```

[1] `with` 문법을 통해 자동으로 파일을 닫게 할 수도 있습니다.

```
for i in range(5):
    s = f.readline().strip()
    print(s)

print()

for line in f:
    s = line.strip()
    print(s)

f.close()
```

위 코드에서 for문을 통해 바로 f을 줄 단위로 읽어올 수 있다는 것을 확인할 수 있습니다.

파일 출력

아래는 구구단을 파일로 작성하는 코드입니다.

```
f = open("./gugu.txt", "w")

for i in range(1, 10):
    for j in range(1, 10):
        f.write(f"{i * j} ")
    f.write("\n")

f.close()
```

유의할 점은, write은 print와는 다르게 개행을 자동으로 하지 않는다는 것입니다. 따라서 직접 .write("\textbackslash n")을 통해 개행 문자를 넣어야 합니다.

이외에도 다음과 같은 메소드를 활용할 수 있습니다.

- .read(): 전체 내용을 한 문자열로 읽어옵니다.
- .readlines(): 각 줄의 내용을 문자열로 가지는 리스트를 반환합니다.
- .writelines(): 리스트의 각 원소를 줄로 가지도록 파일에 작성합니다.

Chapter 7

재귀, 다양한 자료구조, 람다 함수

> *Sometimes recursion seems to brush paradox very closely. For example, there are* recursive definitions. *Such a definition may give the casual viewer the impression that something is being defined in terms of* itself. *That would be circular and lead to infinite regress, if not to paradox proper. Actually, a recursive definition* (*when properly formulated*) *never leads to infinite regress or paradox. This is because a recursive definition never defines something in terms of itself, but always in terms of* simpler versions *of itself.*
>
> — Douglas Hofstadter, *Gödel, Escher, Bach*

7.1 재귀

다음과 같은 수열을 피보나치 수열이라고 부릅니다:

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

바로 이전 두 항의 합이 그 다음 항을 결정합니다. 피보나치 수열은 처음 두 항이 1로 정해져 있고, 이런 첫 두 값들을 초항이라고 합니다. 이러한 두 값을 초항이라고 부릅니다. 따라서 피보나치 수열은 두 초항과 다음 항을 생성하는 규칙을 통해 다음과 같이 정의할 수 있습니다:

$$f_n = \begin{cases} 1 & \text{for } n \in \{1, 2\} \\ f_{n-1} + f_{n-2} & \text{for } n \in \mathbb{N} \setminus \{1, 2\} \end{cases}$$

위와 같이 수열을 귀납적으로 정의하면, 이를 계산하는 코드를 바로 쓸 수 있습니다.

```
def f(n):
    if n <= 2:
        return 1
    return f(n - 1) + f(n - 2)
```

귀납적으로 정의되는 수열들은 손쉽게 코드로 옮겨쓸 수 있습니다.

이렇게 작성한 함수는 자기 자신을 호출하는데요, 이와 같은 함수를 재귀 함수 *recursive function*이라고 합니다. 이러한 예시들을 하나씩 살펴보도록 합시다.

7.1.1 유클리드 호제법

유클리드 호제법은 두 수의 최대공약수를 구하는 알고리즘입니다. 두 수 a와 b가 정해졌을 때, 유클리드 호제법은 이 둘의 최대공약수를 다음의 규칙에 따라 귀납적으로 구합니다:

$$\gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

이는 다음과 같이 바로 코드로 옮겨 적을 수 있습니다.

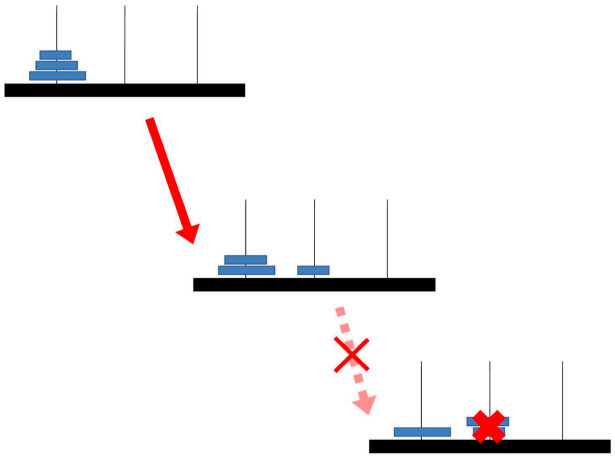
```
def f(n):
    if b == 0:
        return a
    return gcd(b, a % b)
```

기존에 반복문으로 작성한 것보다 우아하지 않나요?

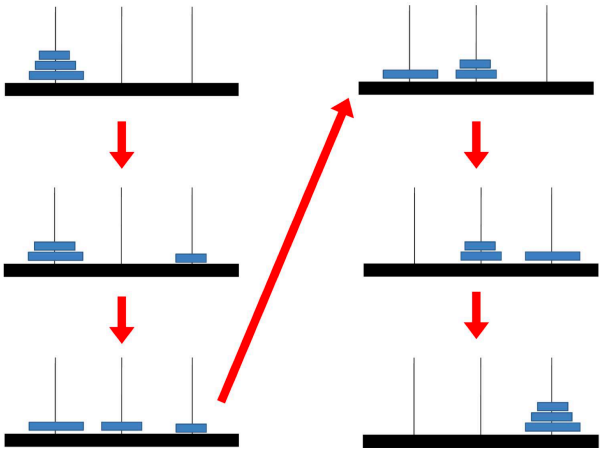
이는 피보나치 수열을 코드로 구현한 것과 동일한 패턴을 가집니다. 조건문으로 초기 조건을 처리해주고, 귀납적으로 정의된 식을 그대로 반환해주는 형태로 구성되어 있습니다.

7.1.2 하노이의 탑

하노이의 탑은 세 개의 봉에 크기 순으로 쌓여진 원판들을 규칙에 맞게 다른 봉으로 옮기는 문제를 말합니다. 이 때, 모든 원판들은 항상 크기 순으로 쌓여져 있어야 하고, 한 번에 하나씩만 옮길 수 있습니다. 즉, 다음과 같은 움직임은 허용되지 않습니다.



만약 세 개의 원판이 있는 경우, 최단 횟수로 원판을 모두 옮기는 해법은 다음과 같습니다.



n개의 원판을 봉 1에서 봉 3으로 옮기는 상황을 가정합시다. 가장 큰 n 번째 원판은 항상 맨 밑에 위치해야 하므로, 해당 원판을 봉 1에서 봉 3으로 옮기기 위해서는 나머지 $(n-1)$개의 원판 모두가 봉 2에 위치해 있어야 합니다. 따라서 n개의 원판을 봉 1에서 봉 3으로 옮기기 위해서는 일단 위의 $(n-1)$개의 원판을 봉 1에서 봉 2로 옮기고, 가장 큰 원판을 봉 1에서 봉 3으로 옮긴 후, 다시 $(n-1)$개의 원판을 봉 2에서 봉 3으로 옮겨야 합니다. 이 때 $(n-1)$개의 원판을 옮기는 상황은 n개의 원판을 옮기는 경우를 귀납적으로 정의한다는 사실을 알 수 있습니다.

하노이 탑 문제는 다음과 같이 수학적으로 정의할 수 있습니다. 함수 $f : \mathbb{N} \times \{1,2,3\} \times \{1,2,3\} \to \{\text{list of moves}\}$를 귀납적으로 정의합니다:

$$\begin{cases} f(0, \text{from}, \text{to}) & = [] \\ f(n, \text{from}, \text{to}) & = f(n-1, \text{from}, \text{to}) + [(n, \text{from}, \text{to})] + f(n-1, \text{other}, \text{to}) \end{cases}$$

이를 코드로 옮기면 아래와 같습니다.

```
def hanoi(n, fr, to):
    if n == 0:
        return []
    ot = 1 + 2 + 3 - fr - to
    return hanoi(n - 1, fr, ot) + [(n, fr, to)] + hanoi(n - 1, ot, to)

n = 5
print(hanoi(n, 1, 3))
```

위에서는 원판이 총 5개일 때 봉 1에서 봉 3으로 옮기는 경우를 출력했습니다. 약간 형태가 복잡해졌을 뿐, 위에서 보았던 코드와 동일한 패턴입니다.

7.1.3 이진 탐색

이진 탐색은 정렬된 리스트가 주어졌을 때 원소를 효율적으로 찾을 수 있는 알고리즘입니다. 이는 분할 정복의 대표적인 예시로, $O(\log n)$의 시간복잡도를 가지고 있습니다. 일반적으로 정렬되지 않은 리스트에서 어떤 원소를 찾기 위해서는 모든 원소를 순서대로 찾아보는 $O(n)$의 작업이 필요하지만, 정렬이 되어 있다는 사실이 주어지면 이진 탐색을 통해 훨씬 효율적으로 원소를 찾을 수 있습니다. 이진 탐색은 가운데 원소를 본 후 만약 해당 원소가 찾고자 하는 값보다 큰 경우 오른쪽을 버리고, 작은 경우 왼쪽을 버리면서 탐색을 진행합니다 (물론 원소는 오름차순 정렬이라고 가정합니다).

이진 탐색을 함수 f : {(sorted list, value, left, right)} → {index}라고 할 때, 다음과 같이 귀납적으로 정의할 수 있습니다:

$$f(a,v,l,r) = \begin{cases} -1 & \text{if } l = r \text{ and } a[r] \neq v \\ r & \text{if } l = r \text{ and } a[r] = v \\ f\left(a,v,l,\frac{l+r}{2}-1\right) & \text{if } a\left[\frac{l+r}{2}\right] > v \\ f\left(a,v,\frac{l+r}{2}+1,r\right) & \text{if } a\left[\frac{l+r}{2}\right] < v \end{cases}$$

이는 아래와 같이 코드로 옮길 수 있습니다.

```
def search(a, value, left, right):
    if not a:
        return -1
    if left == right:
        if a[left] == value:
```

```
            return left
        return -1
    mid = (left + right) / 2
    if a[mid] > value:
        return f(a, value, left, mid - 1)
    if a[mid] < value:
        return f(a, value, mid + 1, right)
    return mid
```

7.1.4 지수 계산

암호학 등의 분야에서는 매우 큰 지수를 다룰 일이 많기 때문에, 효율적인 지수 계산 알고리즘이 필요합니다. 일반적으로 지수 계산을 함수로 구현하라고 하면, 아래와 같은 단순한 코드를 쉽게 떠올릴 수 있습니다.

```
def power_simple(x, n):
    p = 1
    for _ in range(n):
        p *= x
    return p
```

이 경우, 시간복잡도는 $O(n)$입니다. 하지만 다음과 같이 귀납적으로 지수를 계산한다면 $O(\log n)$만에도 지수를 계산할 수 있습니다. 지수 계산 함수 $x^n = f(x,n)$이라고 할 때,

$$f(x,n) = \begin{cases} 1 & \text{if } b = 0 \\ f\left(x, \frac{x}{2}\right)^2 & \text{if } b \text{ is even} \\ x f\left(x, \frac{b-1}{2}\right)^2 & \text{if } b \text{ is odd} \end{cases}$$

으로 정의할 수 있습니다. 이는 바로 코드로 옮길 수 있습니다.

```
def power(x, n):
    if n == 0:
        return 1
    p = power(x, n // 2)
    if n % 2:
        return p * p * x
    return p * p
```

이러한 지수 계산은 아래와 같이 피보나치 수열의 계산에도 사용할 수 있습니다.

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

이를 통해 피보나치 수열의 계산을 $O(\log n)$으로 수행할 수 있다는 사실도 알 수 있습니다.

재귀법은 작성하는데 간편하다는 장점이 있지만, 함수를 여러번 호출한다는 점에서 큰 부하가 걸릴 수 있습니다. 따라서 필요한 경우 반복문을 사용해 수행 시간을 단축시킬 수도 있습니다. 아래에는 각각 재귀법과 반복법으로 피보나치 수열을 계산하는 방법이 나와 있습니다.

```
def f_rec(n):
    if n <= 2:
        return 1
    return f_rec(n - 1) + f_rec(n - 2)

def f_iter(n):
    f = [None] * (n + 1)
    f[0], f[1] = 0, 1
    for i in range(2, n + 1):
        f[i] = f[i - 1] + f[i - 2]
    return f[n]
```

이렇게 재귀법을 반복법으로 바꾸어 해결하는 전략을 동적 계획법이라고 하는데, 나중에 더 자세히 알아보기로 합니다.

7.2 다양한 자료구조

7.2.1 튜플

지금까지 우리는 어떤 데이터를 담는 자료형으로써 리스트만을 사용했습니다. 이러한 리스트는 가변적으로, 어떤 원소를 새로 추가하거나 없앨 수 있었습니다. 나아가 리스트는 원소들의 순서를 보존하고, 같은 값을 여러 번 추가할 수 있었습니다. 예컨대, `[1, 2, 1]`과 `[1, 1, 2]`는 다른 리스트이며, `[1, 1, 1]`은 `[1]`과는 다른 리스트입니다. 또한 append 메소드를 통해 새로운 원소를 더하고, pop 등을 통해 원소를 제거할 수 있었습니다. 파이썬에서는 이러한 가변 리스트 뿐만이 아니라, 불변하느 리스트라고 볼 수 있는 튜플 *tuple* 자료형을 제공합니다. 아래 코드에서는 리스트와 튜플의 유사점과 차이점을 알아볼 수 있습니다.

```
a = [2, 4, 2, 9, 5]
print(type(a))  # <type 'list'>

sum = 0
for i in range(len(a)):
    sum += a[i]

for x in a:
    sum += x

a[1] = 10  # legal

b = (2, 4, 2, 9, 5)
print(type(b))  # <type 'tuple'>

sum = 0
for i in range(len(b)):
    sum += b[i]

for x in b:
    sum += x

b[1] = 10  # illegal
```

또한 튜플 자료형의 변수를 선언할 때에는 괄호를 뺄 수 있습니다. 위에서 `b = (2, 4, 2, 9, 5)` 대신 `b = 2, 4, 2, 9, 5`로 정의하여도 무방합니다. 단, 원소가 하나인 튜플을 정의할 때는 `(1)`의 형태가 아닌 `(1,)`으로 쉼표를 붙여주어야 합니다. 이 때도 괄호는 생략하고 `b = 1,`와 같이 작성하여도 길이가 1인 튜플로 선언됩니다. 나아가 리스트처럼 튜플도 수, 문자열, 튜플, 나아가 리스트까지 임의의 변수를 다 원소로 담을 수 있습니다. 즉, `(1, (2, 4), "string", ["Python", "KOI", (3, 1), None], True)`의 형태의 튜플을 사용할 수 있습니다.

지금까지 변수 뒤바꾸기(swapping)등에서 보았던 다중 할당도 사실 튜플의 문법을 사용한 것입니다. 파이썬에서는 `(a, b) = (1, 2)`와 같이 작성하는 것은 `a`와 `b`에 각각 1과 2를 대입하라는 것과 동일한 의미를 가집니다. 나아가 위에서 튜플을 사용할 때 괄호를 생략해도 된다고 언급한 것을 상기하면 `a, b = 1, 2`와 동일한 표현임을 알 수 있습니다. 즉, `a, b = b, a`를 통해 두 변수의 값을 뒤바꾸는 것은 `(a, b) = (b, a)`와 완전히 동일한 과정입니다.

정리하자면, 아래의 표현들은 모두 동일한 의미를 가집니다.

- `a, b, c = 1, 2, 3`
- `a, b, c = (1, 2, 3)`
- `(a, b, c) = 1, 2, 3`
- `(a, b, c) = (1, 2, 3)`
- `t = 1, 2, 3; a, b, c = t`
- `t = (1, 2, 3); a, b, c = t`
- `t = 1, 2, 3; (a, b, c) = t`
- `t = (1, 2, 3); (a, b, c) = t`

튜플에서 사용할 수 이용할 수 있는 연산자와 메소드는 리스트의 연산자 혹은 메소드 중 불변성을 벗어나지 않는 것들을 모두 적용할 수 있습니다. 가령, 튜플에 아래의 연산을 모두 시행할 수 있습니다.

- `+`, `*`
- `a[i:j]`
- `in`, `for`
- `==`, `is`
- `len`, `min`, `max`, `sum`

반면 아래의 메소드들은 모두 사용할 수 없습니다:

- `a[i] = x`
- `.append`
- `.reverse`
- `.sort`
- `.remove`
- `.pop`

튜플과 `for`문을 활용해 다음과 같은 작업을 할 수 있습니다.

```
a = [(1, 2, "abc"), (3, 4, (5, 6)), (7, True, [8, 9])
for x, y, z in a:
    print(z)
```

위 코드를 실행하면 "abc", (5, 6), [8, 9]가 출력될 것입니다.

튜플은 리스트와 자유롭게 형변환을 할 수 있습니다. 어떤 튜플 (1, 4, 2)가 주어졌을 때, 이를 정렬하고 싶다면 불변 객체인 튜플을 가변 객체인 리스트로 형변환하여 정렬한 후, 다시 튜플로 형 변환할 수 있습니다.

```
a = (1, 4, 2)
a = list(a)
a.sort()
a = tuple(a)
```

이러한 튜플은 값을 변형하지 말아야 하는 경우나, 불변 객체가 반드시 필요한 경우에 사용할 수 있습니다. 곧이어 설명할 집합과 사전은 원소로 가변 객체를 허용하지 않기 때문에 이를 사용할 수 있습니다.

7.2.2 집합

파이썬에서 집합 *set*은 리스트, 튜플, 혹은 문자열에 set를 취하여 얻을 수 있습니다. 예컨대, set([1, 2, 2])는 set([1, 2])를, set("ABBBC")는 set(['A', 'C', 'B']) 를 얻을 수 있습니다. 만약 공집합을 얻고 싶다면 set(), set(()), set([]), set("") 등을 사용하면 됩니다. 유의할 점은, set([()])는 빈순서쌍을 원소로 가지는 집합이라는 것입니다 ({}를 생각해보면 이해가 될 것입니다.).

집합 자료형에는 다음과 같은 연산자들이 있습니다.

- .add(e): 원소를 e를 없다면 새로 추가합니다.
- .remove(e): 원소 e를 제거하고, 없다면 에러를 일으킵니다.
- .clear(): 모든 원소를 제거합니다.
- s1 | s2: 두 집합의 합집합을 구합니다.
- s1 \& s2: 두 집합의 교집합을 구합니다.
- s1 - s2: 두 집합의 차집합을 구합니다.
- s1 < s2, s1 <= s2: 두 집합의 포함 관계의 참 거짓을 구합니다.
- e in s: e가 s의 원소인지의 참 거짓을 구합니다.
- e not in s: not (e in s)의 값과 동일합니다.
- s1 == s2: 두 집합이 동일한지의 참 거짓을 구합니다.

어떤 집합을 복사하기 위해서는 리스트와 동일하게 .copy() 연산자를 사용해야 합니다. 아래 예시를 살펴봅시다.

```
s1 = set([1, 2, 3])
s2 = s1
s3 = s1.copy()
s1.add(4)
print s2  # aliased
print s3  # copied
```

이는 리스트와 동일한 모습을 보인다는 것을 알 수 있습니다. (대신, 리스트의 경우에는 `[:]`를 통해서도 복사를 할 수 있었습니다.)

리스트와 마찬가지로 집합도 `for` 문에 사용할 수 있습니다. 그러나, 그 순서는 정해져 있지 않고, `[i]`와 같은 인덱스로의 접근도 불가능합니다. 그리고 위에서 리스트와 튜플이 형 변환이 가능했던 것과 같이, 집합 또한 이들과의 형 변환이 가능합니다.

마지막으로는, 집합의 원소로는 앞서 간략히 언급했듯이 불변 객체만 올 수 있다는 것입니다. 예컨대, 가변 객체인 리스트나 집합은 집합의 원소가 될 수 없습니다. `set([set([1, 2]), 3])`이나 `set([[1, 2], 3])`은 불가능한 것입니다. 반면 튜플이나 문자열과 같은 불변 객체는 집합의 원소가 될 수 있습니다. 따라서 어떤 객체들의 모임을 집합의 원소로 지정하고 싶을 때에는 튜플로 형 변환을 한 후 넣어야 합니다.

7.2.3 문자열

이미 여러 차례 문자열에 대해서 다룬 적이 있지만, 문자열의 포맷 지정과 추가적인 메소드에 대해서 언급할 내용이 있습니다.

- `.startswith(prefix)`: 문자열이 `prefix`로 시작하는지 확인합니다.
- `.endswith(suffix)`: 문자열이 `suffix`로 끝나는지 확인합니다.
- `.find(str)`: 문자열이 `str`을 포함하는 첫 번째 인덱스를 반환하고, 없다면 -1을 반환합니다.
- `.replace(old, new)`: 문자열의 `old`를 `new`로 치환한 문자열을 반환합니다.
- `.strip()`: 문자열의 시작과 끝에 있는 공백 문자를 모두 제거한 값을 반환합니다.
- `.lstrip()`: 문자열의 시작에 있는 공백 문자를 모두 제거한 값을 반환합니다.
- `.rstrip()`: 문자열의 끝에 있는 공백 문자를 모두 제거한 값을 반환합니다.
- `.split()`: 문자열의 공백으로 구분된 단어들의 리스트를 반환합니다.

- .join(list): 주어진 문자열을 사이로 list의 문자열들을 잇습니다.

문자열의 포맷 지정은 다음과 같이 할 수 있습니다.

```
print(f"Max between {x0:d} and {x1:d} is {val:g}")
```

이 때, 다음과 같은 서식 지정자가 있습니다.

- d: 정수
- g: 실수
- .nf: 소수점 아래 n 자리수까지 나타내는 실수
- s: 문자열

서식 지정자를 활용해 좌측 혹은 우측 정렬을 할 수도 있습니다. nf와 같이 우측 정렬을, <nf와 같이 좌측 정렬을 할 수 있습니다.

```
print(f"{x0:3d} - {x1:3d} : {val:10g}")
print(f"{x0:<3d} - {x1:<3d} : {val:<10g}")
```

또한, 이와 소수점 아래 자리수에 대한 형식 지정자를 결합하여 3.2g와 같은 표현도 가능합니다.

7.2.4 사전

사전 *dictionary*은 일반화된 인덱스를 가지는 리스트로 생각할 수 있습니다. 사전 자료형에서 이러한 인덱스를 키 *key*라고 부르고, 해당 키에 대응되는 값 *value*이 존재합니다. 사전에서는 키를 통해서 값을 효율적으로 찾을 수 있게 구현이 되어 있습니다. 다음은 사전 자료형의 사용 예시입니다.

```
d = {2: ["a", "bc"], (2, 4): 27, "xy": {1:2.5, "a":3}}
print(d[2])
print(d[(2, 4)])
print(d["xy"])
print(d["xy"][1])
```

위에서 볼 수 있듯이, 사전에서 키는 불변 객체이어야 하며, 값은 어떠한 객체이든지 올 수 있습니다.

사전에 어떤 키와 값을 추가하는 것은 단순히 d[key] = value를 실행하는 것으로 충분합니다. 또한 이미 있는 키의 값을 바꾸는 것도 동일하게 수행할 수 있습니다. 반면 어떤 키와 값을 없애는 것은 del d[key]를 통해 실시할 수 있습니다. 아래는 사용 예시입니다.

```
d = {2: ["a", "bc"], (2, 4): 27, "xy": {1:2.5, "a":3}}
d[(1, "p)] = "q"
d[(2, 4)] += 1
del d["xy"]
print(d)
```

사전의 원소의 개수는 리스트에서처럼 len을 사용할 수 있으며, 어떤 키가 존재하는지 파악할 때는 key in d를 쓸 수 있습니다. 값이 존재하는지 파악하기 위해서는 .values()를 통해 모든 값들의 리스트를 불러온 후 확인해야 합니다. 또한 .keys()를 통해 모든 키들의 리스트를 불러올 수 있고, 키와 값의 쌍들의 리스트는 .items()를 통해 불러올 수 있습니다.

집합을 출력하는 것과 같이 사전을 출력하는 것도 모두 무작위의 순서를 가지고 있습니다. 만약 키를 정렬된 순서대로 값을 출력하고 싶다면, .keys()를 통해 키들의 리스트를 얻은 후, 정렬한 후 for문을 통해 출력을 하는 방법을 생각할 수 있습니다.

7.3 람다 함수

람다 함수 *lambda function*는 간단한 함수를 한 줄에 작성하도록 하는 문법입니다. 람다 함수는 다음과 같이 작성할 수 있습니다.

```
lambda arguments: expression
```

예컨대 다음과 같은 함수

```
def add(x, y):
    return x + y

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

는 아래와 같이 람다 함수로 간결하게 표현할 수 있습니다.

```
>>> (lambda x, y: x + y)(10, 20)
30
```

위에서 보다시피 람다 함수는 이름이 명시되어 있지 않기 때문에, 익명 함수 *anonymous function*라고도 불리웁니다.

람다 함수는 간단한 함수를 사용할 때 편리하게 사용할 수 있으며, `map`이나 `sort`등와 함께 사용할 때 빛을 발합니다. 먼저 `map` 함수를 살펴봅시다. `map` 함수는 어떤 함수와 리스트, 튜플, 문자열과 같은 순서형 자료를 인자로 받으며, 리스트의 각 원소에 해당 함수를 적용시킨 새로운 리스트를 반환합니다. 이를 사용해 0부터 4까지의 정수의 제곱을 담은 리스트를 다음과 같이 만들 수 있습니다.

```
>>> list(map(lambda x: x ** 2, range(5)))
[0, 1, 4, 9, 16]
```

`map`은 주어진 리스트의 각 원소를 형 변환할 때도 아래와 같은 방법으로 `for` 문을 사용하지 않고 간단하게 사용할 수 있습니다.

```
>>> a = ['3', '5', '1', '8']
>>> list(map(int, a))
[3, 5, 1, 8]
```

정렬을 할 때 원소의 절댓값으로 정렬하고 싶다면 어떻게 해야 할까요? 혹은 다른 가중치를 줘서 주고 싶다면요? `sort` 혹은 `sorted`의 `key` 인자로 값을 변환할 함수를 넣어주면 됩니다:

```
>>> sorted([-5, 3, 5, 7, 1], key=lambda x: abs(x))
[1, 3, -5, 5, 7]
```

찾아보기

Important works such as histories, biographies, scientific and technical text-books, etc., should contain indexes. Indeed, such works are scarcely to be considered complete without indexes. An index is almost invariably placed at the end of a volume and is set in smaller type than the text-matter. Its subjects should be thoroughly alphabetized. The compiling of an index is interesting work, though some authors are apt to find it tedious and delegate the work to others. The proofreader who undertakes it will find that it is splendid mental exercise and brings out his latent editorial capability.

— Albert H. Highton, *Practical Proofreading*