

이것이 취업을 위한 코딩 테스트다 with 파이썬

코딩 테스트 개요 및 파이썬 문법

나동빈(dongbinna@postech.ac.kr)

Pohang University of Science and Technology

코딩 테스트 개요

코딩 테스트란?

- 기업/기관에서 직원이나 연수생을 선발하기 위한 목적으로 시행되는 일종의 문제 풀이 시험
- 공개채용을 하는 기업에서는 코딩 테스트를 주로 이용합니다.
 - 문제 해결 역량을 평가할 수 있습니다.
 - 채점 시스템을 통해 응시자의 수를 효과적으로 줄일 수 있습니다.

코딩 테스트의 유형

- **온라인 코딩 테스트**
 - 인터넷을 활용해 프로그래밍 역량을 평가하여 응시자를 선별합니다.
 - 대체로 타인과 문제풀이를 공유하지 않는 선에서 인터넷 검색을 허용합니다.
- **오프라인 코딩 테스트**
 - 시험장에 방문하여 치르는 시험입니다.
 - 대체로 인터넷 검색이 허용되지 않으며 회사에서 제공하는 컴퓨터 환경을 이용하도록 합니다.

온라인 저지(Online Judge)란?

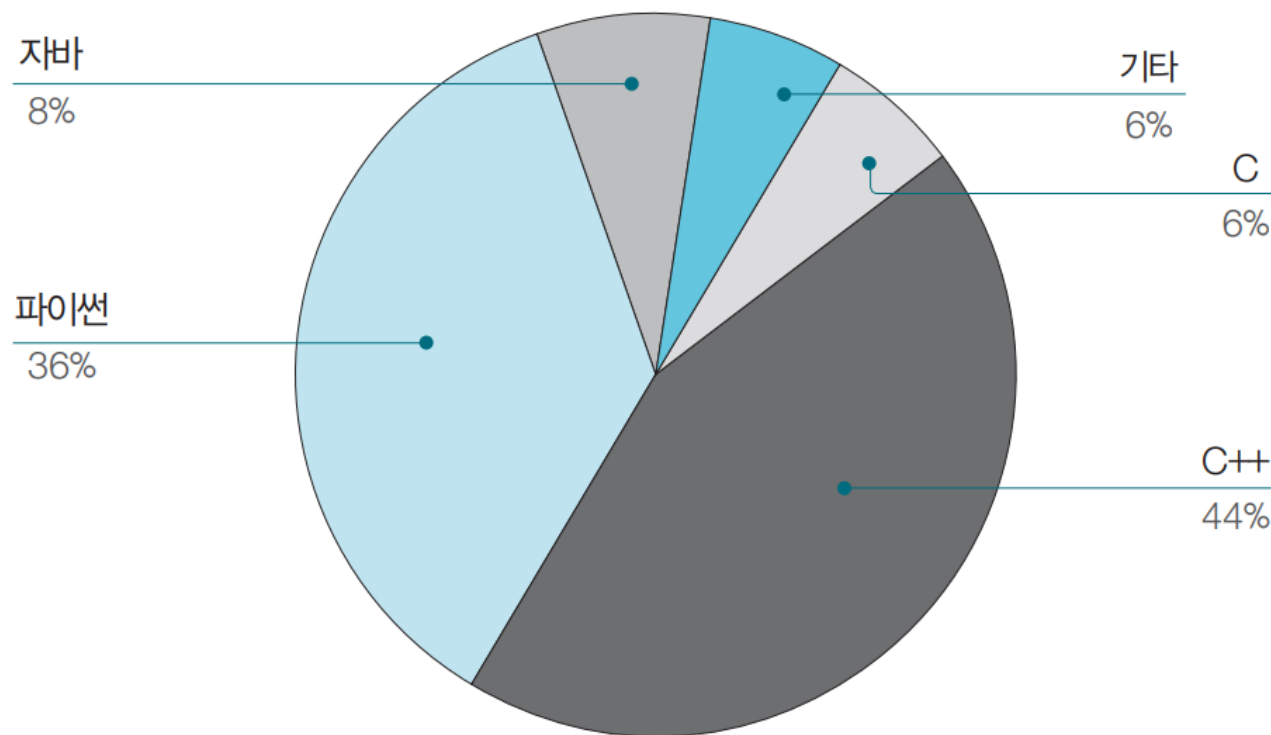
- 프로그래밍 대회나 코딩 테스트에서 나올 법한 문제를 시험해보는 온라인 시스템을 말합니다.

해외	코드포스(Codeforces)	http://www.codeforces.com
	탑코더(TopCoder)	https://www.topcoder.com
	릿코드(LeetCode)	https://leetcode.com
	코드셰프(CODECHEF)	https://www.codechef.com
국내	백준 온라인 저지(BOJ)	https://www.acmicpc.net
	코드업(CodeUp)	https://codeup.kr
	프로그래머스(Programmers)	https://programmers.co.kr
	SW Expert Academy	https://swexpertacademy.com

[표] 다양한 온라인 저지 사이트

코딩 테스트 응시자 설문

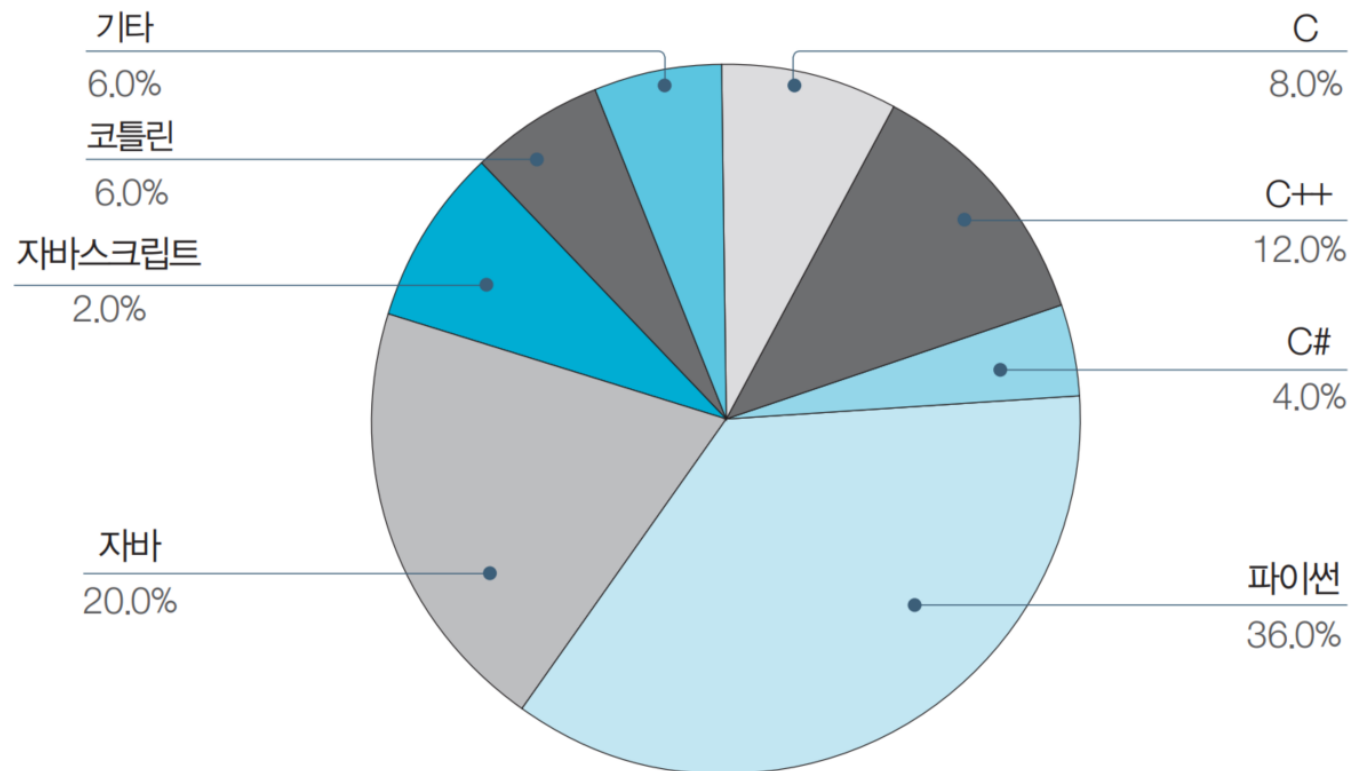
알고리즘 문제 풀이 방식의 코딩 테스트에서 가장 유리한
프로그래밍 언어는 무엇이라고 생각하시나요?



- 설문 대상: IT 계열 직군의 취업 준비생 및 경력자 50명 (2016 ~ 2019년 사이에 평균 3회 이상의 코딩 테스트 응시) / 설문 진행자: 나동빈

코딩 테스트 응시자 설문

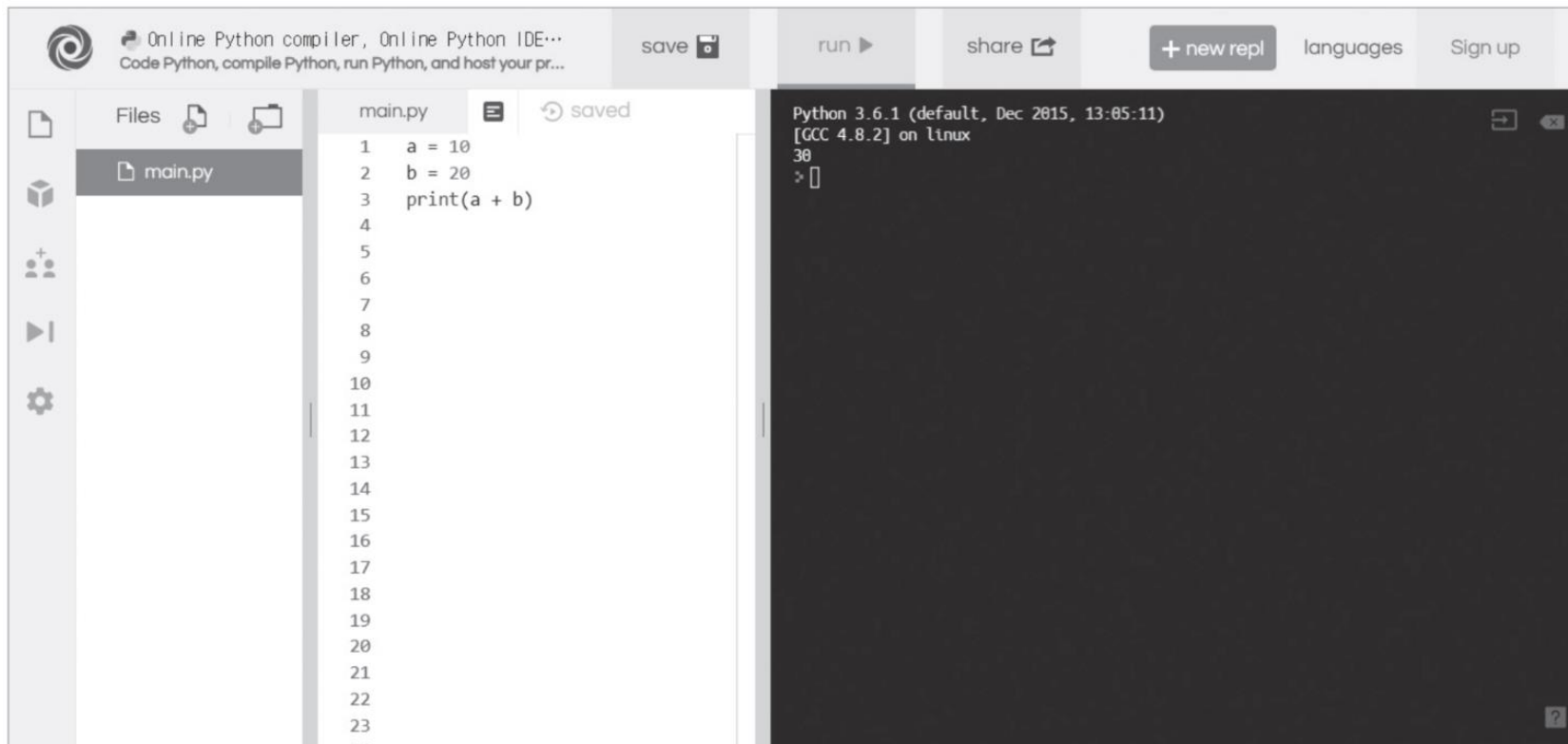
프로그램 개발 방식의 코딩 테스트에서 가장 유리한
프로그래밍 언어는 무엇이라고 생각하시나요?



- 설문 대상: IT 계열 직군의 취업 준비생 및 경력자 50명 (2016 ~ 2019년 사이에 평균 3회 이상의 코딩 테스트 응시) / 설문 진행자: 나동빈

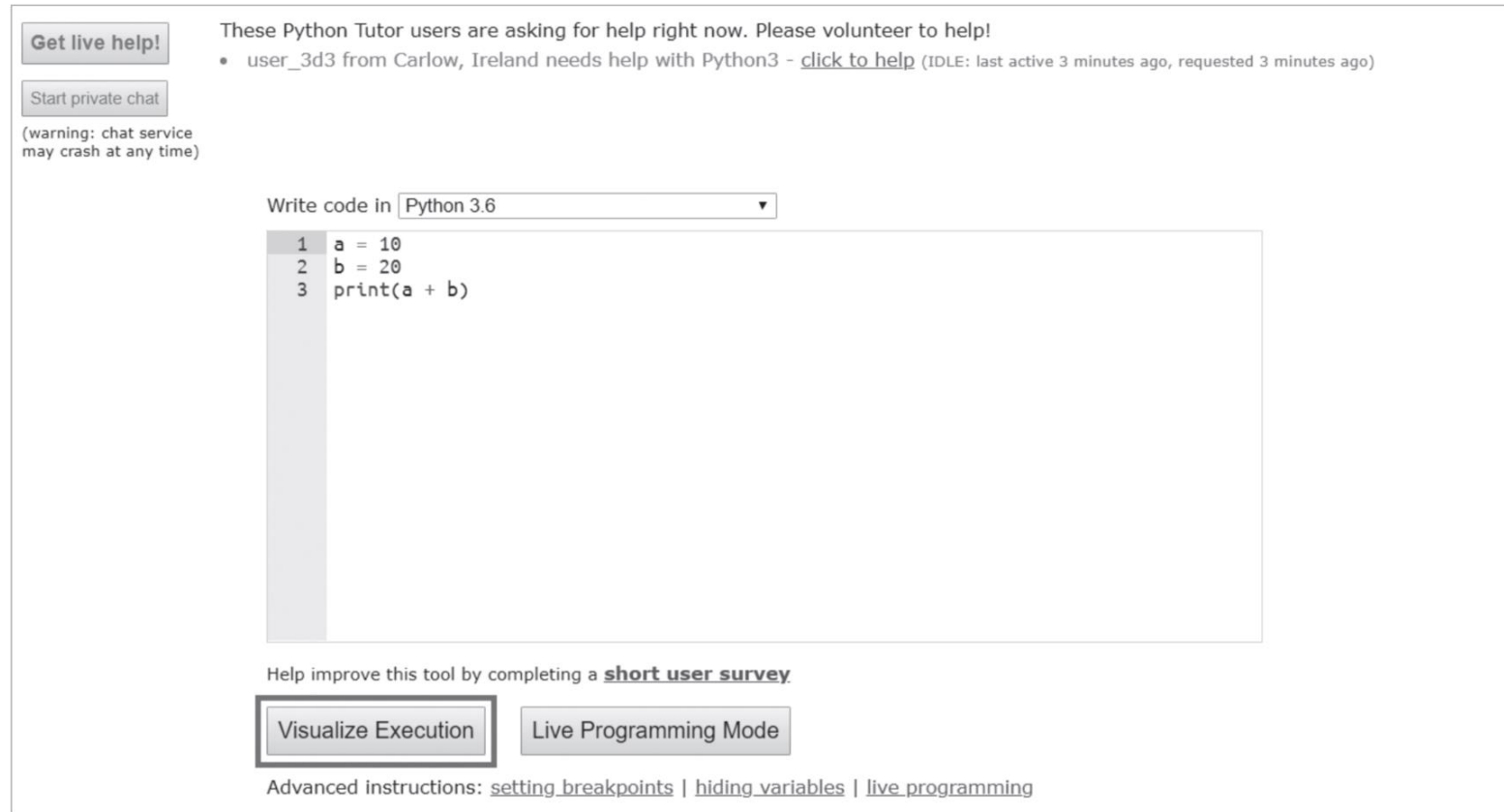
온라인 개발 환경 (Python)

- 리플릿: <https://repl.it/languages/python3>



온라인 개발 환경 (Python)

- 파이썬 튜터: <http://pythontutor.com/visualize.html>



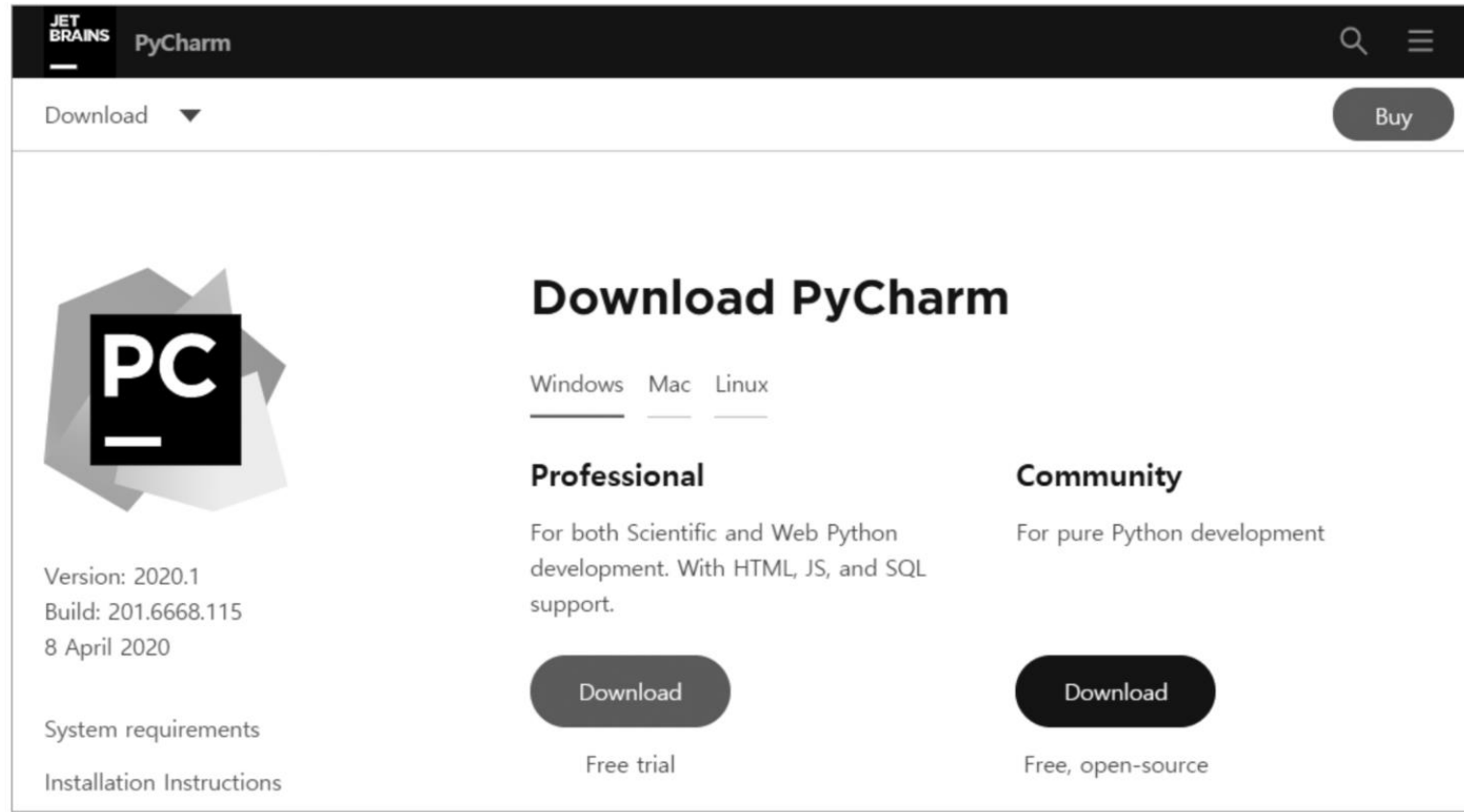
The screenshot displays the Python Tutor web application interface. At the top left, there are buttons for 'Get live help!' and 'Start private chat', with a warning below the chat button stating '(warning: chat service may crash at any time)'. To the right, a message indicates that other users are seeking help, with a link to 'click to help'. The main area features a code editor with a dropdown menu set to 'Python 3.6'. The code entered is:

```
1 a = 10
2 b = 20
3 print(a + b)
```

Below the code editor, there is a prompt to 'Help improve this tool by completing a [short user survey](#)'. At the bottom, two buttons are visible: 'Visualize Execution' (which is highlighted with a red border) and 'Live Programming Mode'. Finally, a link for 'Advanced instructions' points to [setting breakpoints](#), [hiding variables](#), and [live programming](#).

오프라인 개발 환경 (Python)

- 파이참(PyCharm): <https://www.jetbrains.com/pycharm/download/>



오프라인 개발 환경 (C/C++)

- Dev C++: <https://sourceforge.net/projects/orwelldevcpp/>



The image shows a project card for Dev-C++ on SourceForge. It features the Dev-C++ logo, the project name, a description, the creator's name, a star rating with the number of reviews, the download count for the week, and three buttons: Download, Get Updates, and Share This.

Dev-C++
A free, portable, fast and simple C/C++ IDE
Brought to you by: [orwelldevcpp](#)

★★★★★ 146 Reviews Downloads: 60,547 This Week

 **Download** Get Updates Share This

자신만의 소스코드 관리하기

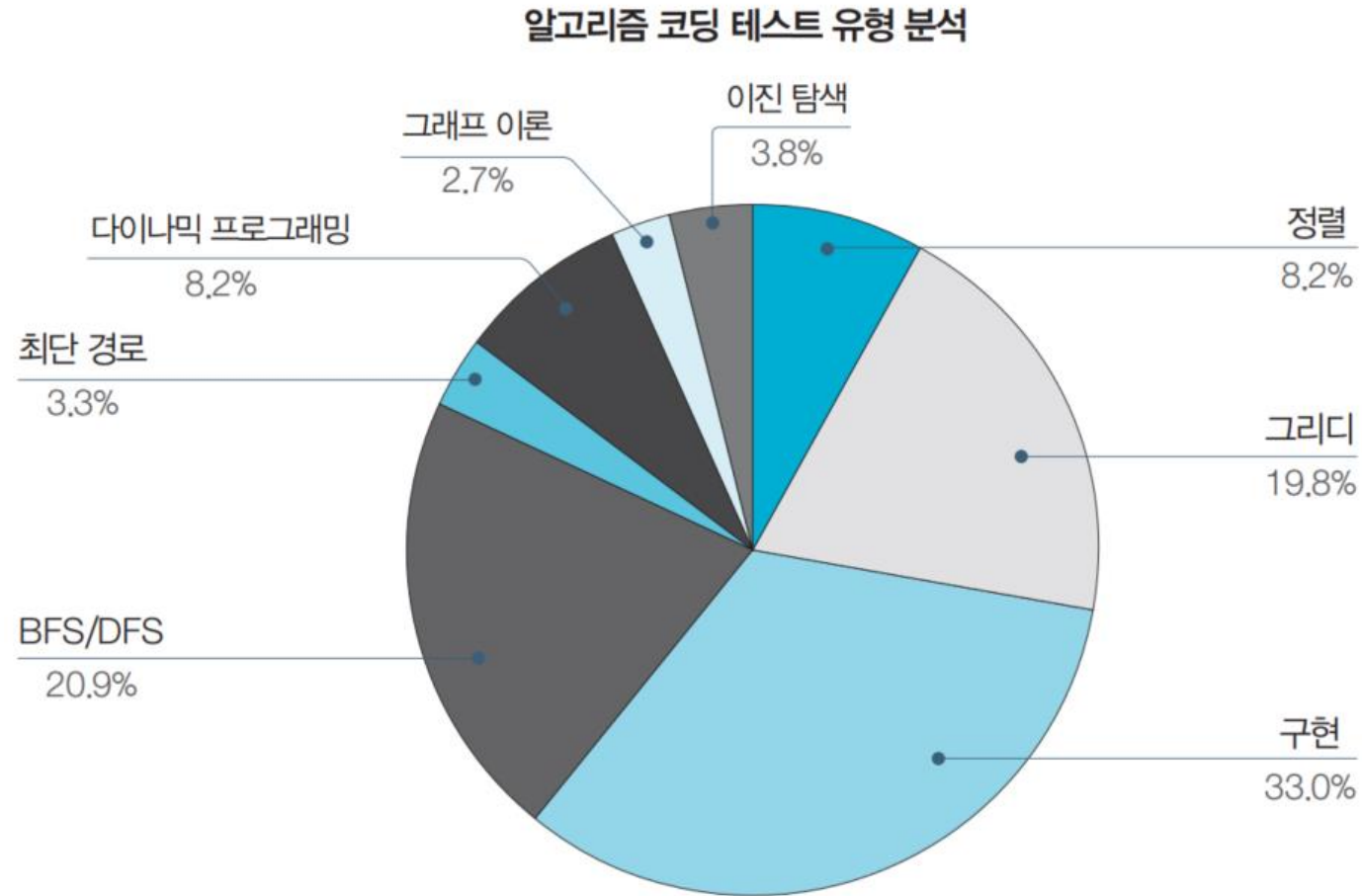
- 알고리즘 코딩 테스트를 준비하는 과정에서 자신만의 소스코드를 관리하는 습관을 들이면 좋습니다.
- 자신이 자주 사용하는 알고리즘 코드를 라이브러리화 하면 좋습니다.
- 팀 노트 예시) <https://github.com/ndb796/Python-Competitive-Programming-Team-Notes>

IT 기업 코딩 테스트 최신 출제 경향

IT 기업 코딩 테스트 최신 출제 경향

- 대부분의 대기업 (삼성전자, 카카오, 라인 포함)은 알고리즘 코딩 테스트를 시행하고 있습니다.
- 응시생들에게 2 ~ 5시간 가량의 시간을 주어 여러 개의 정해진 알고리즘 문제들을 풀도록 합니다.
- **가장 출제 빈도가 높은 알고리즘 유형은 다음과 같습니다.**
 - 그리디 (쉬운 난이도)
 - 구현
 - DFS/BFS를 활용한 탐색

IT 기업 코딩 테스트 최신 출제 경향



- 2016 ~ 2019년에 출제되었던 국내외 주요 기업들의 공채에 등장한 알고리즘 유형

2019년 주요 기업 코딩 테스트 유형 분석

	날짜	풀이 시간	문제 개수	커트라인	주요 문제 유형	시험 유형
삼성전자	상반기 (2019-04-14)	3시간	2문제	2문제	완전 탐색, 시뮬레이션, 구현, DFS/BFS	오프라인
	하반기 (2019-10-20)					
카카오	1차 (2019-09-07)	5시간	7문제	4문제 (예상)	구현, 이진 탐색, 자료구조	온라인
	2차 (2019-09-21)	5시간	1문제	—	추천 시스템 개발	오프라인
라인	상반기 (2019-03-16)	3시간	5문제	3문제 (예상)	탐색, 구현, 문자열, 다이나믹 프로그래밍	온라인
	하반기 (2019-09-22)	3시간	6문제	4문제	자료구조, 완전 탐색, 구현	온라인

- 위 표는 다양한 후기 및 복원된 문제를 참고하여 작성된 것으로, 100% 일치하지는 않을 수 있습니다.

2018년 주요 기업 코딩 테스트 유형 분석

	날짜	풀이 시간	문제 개수	컷라인	주요 문제 유형	시험 유형
삼성전자	상반기 (2018-04-15)	3시간	2문제	1문제	완전 탐색, 구현, DFS/BFS, 시뮬레이션	오프라인
	하반기 (2018-10-21)					
카카오	1차 (2018-09-15)	5시간	7문제	3문제	그리디, 구현, 자료구조	온라인
	2차 (2018-10-06)	5시간	1문제	—	시뮬레이션 개발	오프라인
라인	상반기 (2018-04-05)	2시간	5문제	2문제	탐색, 그리디, 다이나믹 프로그래밍, 구현	온라인
	하반기 (2018-10-13)	2시간	4문제	2문제 (예상)	탐색, 그리디, 구현, 문자열	온라인

- 위 표는 다양한 후기 및 복원된 문제를 참고하여 작성된 것으로, 100% 일치하지는 않을 수 있습니다.

알고리즘 성능 평가

복잡도(Complexity)

- 복잡도는 알고리즘의 성능을 나타내는 척도입니다.
 - **시간 복잡도**: 특정한 크기의 입력에 대하여 알고리즘의 수행 시간 분석
 - **공간 복잡도**: 특정한 크기의 입력에 대하여 알고리즘의 메모리 사용량 분석
- 동일한 기능을 수행하는 알고리즘이 있다면, 일반적으로 복잡도가 낮을수록 좋은 알고리즘입니다.

빅오 표기법(Big-O Notation)

- 가장 빠르게 증가하는 항만을 고려하는 표기법입니다.
 - 함수의 상한만을 나타내게 됩니다.
- 예를 들어 연산 횟수가 $3N^3 + 5N^2 + 1,000,000$ 인 알고리즘이 있다고 합시다.
 - 빅오 표기법에서는 차수가 가장 큰 항만 남기므로 $O(N^3)$ 으로 표현됩니다.

빅오 표기법(Big-O Notation)

좋음(Better)



나쁨(Worse)

순위	명칭
$O(1)$	상수 시간(Constant time)
$O(\log N)$	로그 시간(Log time)
$O(N)$	선형 시간
$O(N \log N)$	로그 선형 시간
$O(N^2)$	이차 시간
$O(N^3)$	삼차 시간
$O(2^n)$	지수 시간

시간 복잡도 계산해보기 1)

- N개의 데이터의 합을 계산하는 프로그램 예제

```
array = [3, 5, 1, 2, 4] # 5개의 데이터(N = 5)
summary = 0 # 합계를 저장할 변수

# 모든 데이터를 하나씩 확인하며 합계를 계산
for x in array:
    summary += x

# 결과를 출력
print(summary)
```

- 수행 시간은 데이터의 개수 N에 비례할 것임을 예측할 수 있습니다.
 - 시간 복잡도: $O(N)$

시간 복잡도 계산해보기 2)

- 2중 반복문을 이용하는 프로그램 예제

```
array = [3, 5, 1, 2, 4] # 5개의 데이터(N = 5)

for i in array:
    for j in array:
        temp = i * j
        print(temp)
```

- 시간 복잡도: $O(N^2)$
- 참고로 모든 2중 반복문의 시간 복잡도가 $O(N^2)$ 인 것은 아닙니다.
 - 소스코드가 내부적으로 다른 함수를 호출한다면 그 함수의 시간 복잡도까지 고려해야 합니다.

알고리즘 설계 Tip

- 일반적으로 CPU 기반의 개인 컴퓨터나 채점용 컴퓨터에서 연산 횟수가 5억을 넘어가는 경우:
 - C언어를 기준으로 통상 1 ~ 3초 가량의 시간이 소요됩니다.
 - Python을 기준으로 통상 5 ~ 15초 가량의 시간이 소요됩니다.
 - PyPy의 경우 때때로 C언어보다도 빠르게 동작하기도 합니다.
- $O(N^3)$ 의 알고리즘을 설계한 경우, N의 값이 5,000이 넘는다면 얼마나 걸릴까요?
- **코딩 테스트 문제에서 시간제한은 통상 1 ~ 5초가량이라는 점에 유의하세요.**
 - 혹여 문제에 명시되어 있지 않은 경우 대략 5초 정도라고 생각하고 문제를 푸는 것이 합리적입니다.

요구사항에 따라 적절한 알고리즘 설계하기

- 문제에서 가장 먼저 확인해야 하는 내용은 시간제한(수행시간 요구사항)입니다.
- 시간제한이 1초인 문제를 만났을 때, 일반적인 기준은 다음과 같습니다.
 - N의 범위가 500인 경우: 시간 복잡도가 $O(N^3)$ 인 알고리즘을 설계하면 문제를 풀 수 있습니다.
 - N의 범위가 2,000인 경우: 시간 복잡도가 $O(N^2)$ 인 알고리즘을 설계하면 문제를 풀 수 있습니다.
 - N의 범위가 100,000인 경우: 시간 복잡도가 $O(N\log N)$ 인 알고리즘을 설계하면 문제를 풀 수 있습니다.
 - N의 범위가 10,000,000인 경우: 시간 복잡도가 $O(N)$ 인 알고리즘을 설계하면 문제를 풀 수 있습니다.

알고리즘 문제 해결 과정

- 일반적인 알고리즘 문제 해결 과정은 다음과 같습니다.
 1. 지문 읽기 및 컴퓨터적 사고
 - 2. 요구사항(복잡도) 분석**
 3. 문제 해결을 위한 아이디어 찾기
 4. 소스코드 설계 및 코딩
- 일반적으로 대부분의 문제 출제자들은 핵심 아이디어를 캐치한다면, 간결하게 소스코드를 작성할 수 있는 형태로 문제를 출제합니다.

수행 시간 측정 소스코드 예제

- 일반적인 알고리즘 문제 해결 과정은 다음과 같습니다.

```
import time
start_time = time.time() # 측정 시작

# 프로그램 소스코드

end_time = time.time() # 측정 종료
print("time:", end_time - start_time) # 수행 시간 출력
```

자료형

자료형

- 모든 프로그래밍은 결국 데이터를 다루는 행위입니다.
 - 자료형에 대한 이해는 프로그래밍의 길에 있어서의 첫걸음이라고 할 수 있습니다.
- 파이썬의 자료형으로는 정수형, 실수형, 복소수형, 문자열, 리스트, 튜플, 사전 등이 있습니다.
 - 파이썬의 자료형은 필수적으로 알아 두어야 합니다.

정수형

- 정수형(Integer)은 정수를 다루는 자료형입니다.
 - 양의 정수, 음의 정수, 0이 포함됩니다.
- 코딩 테스트에서 출제되는 많은 문제들은 정수형을 주로 다루게 됩니다.

정수형

양의 정수

a = 1000

print(a)

음의 정수

a = -7

print(a)

0

a = 0

print(a)

실행 결과

1000

-7

0

실수형

- 실수형(Real Number)은 소수점 아래의 데이터를 포함하는 수 자료형입니다.
 - 파이썬에서는 변수에 소수점을 붙인 수를 대입하면 실수형 변수로 처리됩니다.
 - 소수부가 0이거나, 정수부가 0인 소수는 0을 생략하고 작성할 수 있습니다.

실수형

```
# 양의 실수  
a = 157.93  
print(a)
```

```
# 음의 실수  
a = -1837.2  
print(a)
```

```
# 소수부가 0일 때 0을 생략  
a = 5.  
print(a)
```

```
# 정수부가 0일 때 0을 생략  
a = -.7  
print(a)
```

실행 결과

```
157.93  
-1837.2  
5.0  
-0.7
```

지수 표현 방식

- 파이썬에서는 e나 E를 이용한 **지수 표현 방식을** 이용할 수 있습니다.
 - e나 E 다음에 오는 수는 10의 지수부를 의미합니다.
 - 예를 들어 1e9라고 입력하게 되면, 10의 9제곱(1,000,000,000)이 됩니다.

$$\text{유효숫자}e^{\text{지수}} = \text{유효숫자} \times 10^{\text{지수}}$$

- 지수 표현 방식은 임의의 큰 수를 표현하기 위해 자주 사용됩니다.
- 최단 경로 알고리즘에서는 도달할 수 없는 노드에 대하여 최단 거리를 **무한(INF)**로 설정하곤 합니다.
- 이때 가능한 최댓값이 10억 미만이라면 무한(INF)의 값으로 1e9를 이용할 수 있습니다.

지수 표현 방식

```
# 1,000,000,000의 지수 표현 방식
```

```
a = 1e9  
print(a)
```

```
# 752.5
```

```
a = 75.25e1  
print(a)
```

```
# 3.954
```

```
a = 3954e-3  
print(a)
```

실행 결과

```
1000000000.0  
752.5  
3.954
```

실수형 더 알아보기

- 오늘날 가장 널리 쓰이는 IEEE754 표준에서는 실수형을 저장하기 위해 4바이트, 혹은 8바이트의 고정된 크기의 메모리를 할당하므로, **컴퓨터 시스템은 실수 정보를 표현하는 정확도에 한계를 가집니다.**
- 예를 들어 10진수 체계에서는 0.3과 0.6을 더한 값이 0.9로 정확히 떨어집니다.
 - 하지만 2진수에서는 0.9를 정확히 표현할 수 있는 방법이 없습니다.
 - 컴퓨터는 최대한 0.9와 가깝게 표현하지만, 미세한 오차가 발생하게 됩니다.

실수형 더 알아보기

```
a = 0.3 + 0.6
print(a)

if a == 0.9:
    print(True)
else:
    print(False)
```

실행 결과

```
0.8999999999999999
False
```

실수형 더 알아보기

- 개발 과정에서 실수 값을 제대로 비교하지 못해서 원하는 결과를 얻지 못할 수 있습니다.
- 이럴 때는 `round()` 함수를 이용할 수 있으며, 이러한 방법이 권장됩니다.
- 예를 들어 123.456을 소수 셋째 자리에서 반올림하려면 `round(123.456, 2)`라고 작성합니다.
 - 결과는 123.46이 됩니다.

실수형 더 알아보기

```
a = 0.3 + 0.6
print(round(a, 4))

if round(a, 4) == 0.9:
    print(True)
else:
    print(False)
```

실행 결과

```
0.9
True
```

수 자료형의 연산

- 수 자료형에 대하여 사칙연산과 나머지 연산자가 많이 사용됩니다.
- 단 나누기 연산자(/)를 주의해서 사용해야 합니다.
 - 파이썬에서 나누기 연산자(/)는 나뉘진 결과를 실수형으로 반환합니다.
- 다양한 로직을 설계할 때 나머지 연산자(%)를 이용해야 할 때가 많습니다.
 - 예시: a가 홀수인지 체크해야 하는 경우
- 파이썬에서는 몫을 얻기 위해 몫 연산자(//)를 사용합니다.
- 이외에도 거듭 제곱 연산자(**)를 비롯해 다양한 연산자들이 존재합니다.

수 자료형의 연산

```
a = 7
b = 3

# 나누기
print(a / b)

# 나머지
print(a % b)

# 몫
print(a // b)
```

실행 결과

```
2.3333333333333335
1
2
```

수 자료형의 연산

```
a = 5
b = 3

# 거듭 제곱
print(a ** b)

# 제곱근
print(a ** 0.5)
```

실행 결과

```
125
2.23606797749979
```

리스트 자료형

- 여러 개의 데이터를 연속적으로 담아 처리하기 위해 사용하는 자료형입니다.
 - 사용자 입장에서 C나 자바에서의 배열(Array)의 기능 및 연결 리스트와 유사한 기능을 지원합니다.
 - C++의 STL vector와 기능적으로 유사합니다.
 - 리스트 대신에 배열 혹은 테이블이라고 부르기도 합니다.

7	1	5	3	2	6	7	5
---	---	---	---	---	---	---	---

리스트 초기화

- 리스트는 대괄호([])안에 원소를 넣어 초기화하며, 쉼표(,)로 원소를 구분합니다.
- 비어 있는 리스트를 선언하고자 할 때는 list() 혹은 간단히 []를 이용할 수 있습니다.
- 리스트의 원소에 접근할 때는 인덱스(Index) 값을 괄호에 넣습니다.
 - 인덱스는 0부터 시작합니다.

리스트 초기화

```
# 직접 데이터를 넣어 초기화
```

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print(a)
```

```
# 네 번째 원소만 출력
```

```
print(a[3])
```

```
# 크기가 N이고, 모든 값이 0인 1차원 리스트 초기화
```

```
n = 10
```

```
a = [0] * n
```

```
print(a)
```

실행 결과

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
4
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

리스트의 인덱싱과 슬라이싱

- 인덱스 값을 입력하여 리스트의 특정한 원소에 접근하는 것을 인덱싱(Indexing)이라고 합니다.
 - 파이썬의 인덱스 값은 양의 정수와 음의 정수를 모두 사용할 수 있습니다.
 - 음의 정수를 넣으면 원소를 거꾸로 탐색하게 됩니다.

리스트의 인덱싱과 슬라이싱

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# 여덟 번째 원소만 출력
```

```
print(a[7])
```

```
# 뒤에서 첫 번째 원소 출력
```

```
print(a[-1])
```

```
# 뒤에서 세 번째 원소 출력
```

```
print(a[-3])
```

```
# 네 번째 원소 값 변경
```

```
a[3] = 7
```

```
print(a)
```

실행 결과

8

9

7

[1, 2, 3, 7, 5, 6, 7, 8, 9]

리스트의 인덱싱과 슬라이싱

- 리스트에서 연속적인 위치를 갖는 원소들을 가져와야 할 때는 슬라이싱(Slicing)을 이용합니다.
 - 대괄호 안에 콜론(:)을 넣어서 시작 인덱스와 끝 인덱스를 설정할 수 있습니다.
 - 끝 인덱스는 실제 인덱스보다 1을 더 크게 설정합니다.

리스트의 인덱싱과 슬라이싱

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# 네 번째 원소만 출력
```

```
print(a[3])
```

```
# 두 번째 원소부터 네 번째 원소까지
```

```
print(a[1 : 4])
```

실행 결과

4

[2, 3, 4]

리스트 컴프리헨션

- 리스트를 초기화하는 방법 중 하나입니다.
 - 대괄호 안에 조건문과 반복문을 적용하여 리스트를 초기화 할 수 있습니다.
- 주의: 아직 조건문과 반복문에 대한 내용을 배우기 전이라면 대략적인 쓰임새만 이해하고 넘어가세요.

리스트 컴프리헨션

```
# 0부터 9까지의 수를 포함하는 리스트  
array = [i for i in range(10)]  
  
print(array)
```

실행 결과

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

리스트 컴프리헨션

```
# 0부터 19까지의 수 중에서 홀수만 포함하는 리스트  
array = [i for i in range(20) if i % 2 == 1]
```

```
print(array)
```

```
# 1부터 9까지의 수들의 제곱 값을 포함하는 리스트  
array = [i * i for i in range(1, 10)]
```

```
print(array)
```

실행 결과

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]  
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

리스트 컴프리헨션과 일반적인 코드 비교하기

코드 1: 리스트 컴프리헨션

```
# 0부터 19까지의 수 중에서 홀수만 포함하는 리스트  
array = [i for i in range(20) if i % 2 == 1]  
  
print(array)
```

코드 2: 일반적인 코드

```
# 0부터 19까지의 수 중에서 홀수만 포함하는 리스트  
array = []  
for i in range(20):  
    if i % 2 == 1:  
        array.append(i)  
  
print(array)
```

실행 결과 (두 코드 모두 동일)

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

리스트 컴프리헨션

- 리스트 컴프리헨션은 2차원 리스트를 초기화할 때 효과적으로 사용될 수 있습니다.
- 특히 $N \times M$ 크기의 2차원 리스트를 한 번에 초기화 해야 할 때 매우 유용합니다.
 - 좋은 예시: `array = [[0] * m for _ in range(n)]`
- 만약 2차원 리스트를 초기화할 때 다음과 같이 작성하면 예기치 않은 결과가 나올 수 있습니다.
 - 잘못된 예시: `array = [[0] * m] * n`
 - 위 코드는 전체 리스트 안에 포함된 각 리스트가 모두 같은 객체로 인식됩니다.

리스트 컴프리헨션 (좋은 예시)

```
# N X M 크기의 2차원 리스트 초기화
n = 4
m = 3
array = [[0] * m for _ in range(n)]
print(array)

array[2][1] = 5
print(array)
```

(1)

0	0	0
0	0	0
0	0	0
0	0	0

(2)

0	0	0
0	0	0
0	5	0
0	0	0

실행 결과

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
[[0, 0, 0], [0, 0, 0], [0, 5, 0], [0, 0, 0]]
```

리스트 컴프리헨션 (잘못된 예시)

N X M 크기의 2차원 리스트 초기화 (잘못된 방법)

n = 4

m = 3

array = [[0] * m] * n

print(array)

array[2][1] = 5

print(array)

(1)

0	0	0
0	0	0
0	0	0
0	0	0

(2)

0	5	0
0	5	0
0	5	0
0	5	0

실행 결과

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]  
[[0, 5, 0], [0, 5, 0], [0, 5, 0], [0, 5, 0]]
```


언더바는 언제 사용하나요?

- 파이썬에서는 반복을 수행하되 반복을 위한 변수의 값을 무시하고자 할 때 언더바(_)를 자주 사용합니다.

코드 1: 1부터 9까지의 자연수를 더하기

```
summary = 0
for i in range(1, 10):
    summary += i
print(summary)
```

코드 2: "Hello World"를 5번 출력하기

```
for _ in range(5):
    print("Hello World")
```

리스트 관련 기타 메서드

함수명	사용법	설명	시간 복잡도
append()	변수명.append()	리스트에 원소를 하나 삽입할 때 사용한다.	$O(1)$
sort()	변수명.sort()	기본 정렬 기능으로 오름차순으로 정렬한다.	$O(N\log N)$
	변수명.sort(reverse = True)	내림차순으로 정렬한다.	
reverse()	변수명.reverse()	리스트의 원소의 순서를 모두 뒤집어 놓는다.	$O(N)$
insert()	insert(삽입할 위치 인덱스, 삽입할 값)	특정한 인덱스 위치에 원소를 삽입할 때 사용한다.	$O(N)$
count()	변수명.count(특정 값)	리스트에서 특정한 값을 가지는 데이터의 개수를 셀 때 사용한다.	$O(N)$
remove()	변수명.remove(특정 값)	특정한 값을 갖는 원소를 제거하는데, 값을 가진 원소가 여러 개면 하나만 제거한다.	$O(N)$

리스트 관련 기타 메서드

```
a = [1, 4, 3]
print("기본 리스트: ", a)

# 리스트에 원소 삽입
a.append(2)
print("삽입: ", a)

# 오름차순 정렬
a.sort()
print("오름차순 정렬: ", a)

# 내림차순 정렬
a.sort(reverse = True)
print("내림차순 정렬: ", a)
```

실행 결과

```
기본 리스트: [1, 4, 3]
삽입: [1, 4, 3, 2]
오름차순 정렬: [1, 2, 3, 4]
내림차순 정렬: [4, 3, 2, 1]
```

리스트 관련 기타 메서드

```
a = [4, 3, 2, 1]

# 리스트 원소 뒤집기
a.reverse()
print("원소 뒤집기: ", a)

# 특정 인덱스에 데이터 추가
a.insert(2, 3)
print("인덱스 2에 3 추가: ", a)

# 특정 값인 데이터 개수 세기
print("값이 3인 데이터 개수: ", a.count(3))

# 특정 값 데이터 삭제
a.remove(1)
print("값이 1인 데이터 삭제: ", a)
```

실행 결과

```
원소 뒤집기: [1, 2, 3, 4]
인덱스 2에 3 추가: [1, 2, 3, 3, 4]
값이 3인 데이터 개수: 2
값이 1인 데이터 삭제: [2, 3, 3, 4]
```

리스트에서 특정 값을 가지는 원소를 모두 제거하기

```
a = [1, 2, 3, 4, 5, 5, 5]
remove_set = {3, 5} # 집합 자료형 (집합 자료형은 추후에 다시 다룹니다.)

# remove_list에 포함되지 않은 값만을 저장
result = [i for i in a if i not in remove_set]
print(result)
```

실행 결과

[1, 2, 4]

문자열 자료형

- 문자열 변수를 초기화할 때는 큰따옴표(“)나 작은 따옴표(‘)를 이용합니다.
- 문자열 안에 큰따옴표나 작은따옴표가 포함되어야 하는 경우가 있습니다.
 - 전체 문자열을 큰따옴표로 구성하는 경우, 내부적으로 작은따옴표를 포함할 수 있습니다.
 - 전체 문자열을 작은따옴표로 구성하는 경우, 내부적으로 큰따옴표를 포함할 수 있습니다.
 - 혹은 백슬래시(\)를 사용하면, 큰따옴표나 작은따옴표를 원하는 만큼 포함시킬 수 있습니다.

문자열 자료형

```
data = 'Hello World'
print(data)

data = "Don't you know \"Python\"?"
print(data)
```

실행 결과

```
Hello World
Don't you know "Python"?
```

문자열 연산

- 문자열 변수에 덧셈(+)을 이용하면 문자열이 더해져서 연결(Concatenate)됩니다.
- 문자열 변수를 특정한 양의 정수와 곱하는 경우, 문자열이 그 값만큼 여러 번 더해집니다.
- 문자열에 대해서도 마찬가지로 인덱싱과 슬라이싱을 이용할 수 있습니다.
 - 다만 문자열은 특정 인덱스의 값을 변경할 수는 없습니다. (Immutable)

문자열 연산

```
a = "Hello"  
b = "World"  
print(a + " " + b)
```

```
a = "String"  
print(a * 3)
```

```
a = "ABCDEF"  
print(a[2 : 4])
```

실행 결과

```
Hello World  
StringStringString  
CD
```

튜플 자료형

- 튜플 자료형은 리스트와 유사하지만 다음과 같은 문법적 차이가 있습니다.
 - 튜플은 한 번 선언된 값을 변경할 수 없습니다.
 - 리스트는 대괄호([])를 이용하지만, 튜플은 소괄호(())를 이용합니다.
- 튜플은 리스트에 비해 상대적으로 공간 효율적입니다.

튜플 사용 예제

```
a = (1, 2, 3, 4, 5, 6, 7, 8, 9)

# 네 번째 원소만 출력
print(a[3])

# 두 번째 원소부터 네 번째 원소까지
print(a[1 : 4])
```

실행 결과

```
4
(2, 3, 4)
```

튜플 사용 예제 (오류가 발생하는 예제)

```
a = (1, 2, 3, 4)
print(a)

a[2] = 7
```

실행 결과

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    a[2] = 7
TypeError: 'tuple' object does not support item assignment
```

튜플을 사용하면 좋은 경우

- 서로 다른 성질의 데이터를 묶어서 관리해야 할 때
 - 최단 경로 알고리즘에서는 (비용, 노드 번호)의 형태로 튜플 자료형을 자주 사용합니다.
- 데이터의 나열을 해싱(Hashing)의 키 값으로 사용해야 할 때
 - 튜플은 변경이 불가능하므로 리스트와 다르게 키 값으로 사용될 수 있습니다.
- 리스트보다 메모리를 효율적으로 사용해야 할 때

사전 자료형

- 사전 자료형은 키(Key)와 값(Value)의 쌍을 데이터로 가지는 자료형입니다.
 - 앞서 다루었던 리스트나 튜플이 값을 순차적으로 저장하는 것과는 대비됩니다.
- 사전 자료형은 키와 값의 쌍을 데이터로 가지며, 원하는 '변경 불가능한(Immutable) 자료형'을 키로 사용할 수 있습니다.
- 파이썬의 사전 자료형은 해시 테이블(Hash Table)을 이용하므로 데이터의 조회 및 수정에 있어서 $O(1)$ 의 시간에 처리할 수 있습니다.

사전 자료형

```
data = dict()
data['사과'] = 'Apple'
data['바나나'] = 'Banana'
data['코코넛'] = 'Coconut'
```

```
print(data)
```

```
if '사과' in data:
    print("'사과'를 키로 가지는 데이터가 존재합니다.")
```

키(Key)	값(Value)
사과	Apple
바나나	Banana
코코넛	Coconut

실행 결과

```
{'사과': 'Apple', '바나나': 'Banana', '코코넛': 'Coconut'}
'사과'를 키로 가지는 데이터가 존재합니다.
```

사전 자료형 관련 메서드

- 사전 자료형에서는 키와 값을 별도로 뽑아내기 위한 메서드를 지원합니다.
 - 키 데이터만 뽑아서 리스트로 이용할 때는 `keys()` 함수를 이용합니다.
 - 값 데이터만을 뽑아서 리스트로 이용할 때는 `values()` 함수를 이용합니다.

사전 자료형 관련 함수

```
data = dict()
data['사과'] = 'Apple'
data['바나나'] = 'Banana'
data['코코넛'] = 'Coconut'

# 키 데이터만 담은 리스트
key_list = data.keys()
# 값 데이터만 담은 리스트
value_list = data.values()
print(key_list)
print(value_list)

# 각 키에 따른 값을 하나씩 출력
for key in key_list:
    print(data[key])
```

실행 결과

```
dict_keys(['사과', '바나나', '코코넛'])
dict_values(['Apple', 'Banana', 'Coconut'])
Apple
Banana
Coconut
```

집합 자료형

- 집합은 다음과 같은 특징이 있습니다.
 - 중복을 허용하지 않습니다.
 - 순서가 없습니다.
- 집합은 **리스트** 혹은 **문자열**을 이용해서 초기화할 수 있습니다.
 - 이때 `set()` 함수를 이용합니다.
- 혹은 **중괄호** (`{}`)안에 각 원소를 콤마(,)를 기준으로 구분하여 삽입함으로써 초기화 할 수 있습니다.
- 데이터의 조회 및 수정에 있어서 $O(1)$ 의 시간에 처리할 수 있습니다.

집합 자료형

```
# 집합 자료형 초기화 방법 1  
data = set([1, 1, 2, 3, 4, 4, 5])  
print(data)
```

```
# 집합 자료형 초기화 방법 2  
data = {1, 1, 2, 3, 4, 4, 5}  
print(data)
```

실행 결과

```
{1, 2, 3, 4, 5}  
{1, 2, 3, 4, 5}
```

집합 자료형의 연산

- 기본적인 집합 연산으로는 합집합, 교집합, 차집합 연산 등이 있습니다.
 - **합집합**: 집합 A에 속하거나 B에 속하는 원소로 이루어진 집합 ($A \cup B$)
 - **교집합**: 집합 A에도 속하고 B에도 속하는 원소로 이루어진 집합 ($A \cap B$)
 - **차집합**: 집합 A의 원소 중에서 B에 속하지 않는 원소들로 이루어진 집합 ($A - B$)

집합 자료형의 연산

```
a = set([1, 2, 3, 4, 5])  
b = set([3, 4, 5, 6, 7])
```

```
# 합집합  
print(a | b)
```

```
# 교집합  
print(a & b)
```

```
# 차집합  
print(a - b)
```

실행 결과

```
{1, 2, 3, 4, 5, 6, 7}  
{3, 4, 5}  
{1, 2}
```

집합 자료형 관련 함수

```
data = set([1, 2, 3])  
print(data)  
  
# 새로운 원소 추가  
data.add(4)  
print(data)  
  
# 새로운 원소 여러 개 추가  
data.update([5, 6])  
print(data)  
  
# 특정한 값을 갖는 원소 삭제  
data.remove(3)  
print(data)
```

실행 결과

```
{1, 2, 3}  
{1, 2, 3, 4}  
{1, 2, 3, 4, 5, 6}  
{1, 2, 4, 5, 6}
```

사전 자료형과 집합 자료형의 특징

- 리스트나 튜플은 순서가 있기 때문에 인덱싱을 통해 자료형의 값을 얻을 수 있습니다.
- 사전 자료형과 집합 자료형은 **순서가 없기 때문에** 인덱싱으로 값을 얻을 수 없습니다.
 - 사전의 키(Key) 혹은 집합의 원소(Element)를 이용해 **$O(1)$** 의 시간 복잡도로 조회합니다.

기본 입출력

기본 입출력

- 모든 프로그램은 적절한 (약속된) 입출력 양식을 가지고 있습니다.
- 프로그램 동작의 첫 번째 단계는 데이터를 입력 받거나 생성하는 것입니다.
- **예시)** 학생의 성적 데이터가 주어지고, 이를 내림차순으로 정렬한 결과를 출력하는 프로그램

입력 예시

```
5
65 90 75 34 99
```

출력 예시

```
99 90 75 65 34
```

자주 사용되는 표준 입력 방법

- `input()` 함수는 한 줄의 문자열을 입력 받는 함수입니다.
- `map()` 함수는 리스트의 모든 원소에 각각 특정한 함수를 적용할 때 사용합니다.
- **예시)** 공백을 기준으로 구분된 데이터를 입력 받을 때는 다음과 같이 사용합니다.
 - `list(map(int, input().split()))`
- **예시)** 공백을 기준으로 구분된 데이터의 개수가 많지 않다면, 단순히 다음과 같이 사용합니다.
 - `a, b, c = map(int, input().split())`

입력을 위한 전형적인 소스코드 1)

```
# 데이터의 개수 입력
n = int(input())
# 각 데이터를 공백을 기준으로 구분하여 입력
data = list(map(int, input().split()))

data.sort(reverse=True)
print(data)
```

실행 결과

```
5 ↵
65 90 75 34 99 ↵
[99, 90, 75, 65, 34]
```

입력을 위한 전형적인 소스코드 2)

```
# n, m, k를 공백을 기준으로 구분하여 입력  
n, m, k = map(int, input().split())  
  
print(n, m, k)
```

실행 결과

```
3 5 7 ↵  
3 5 7
```

빠르게 입력 받기

- 사용자로부터 입력을 최대한 빠르게 받아야 하는 경우가 있습니다.
- 파이썬의 경우 `sys` 라이브러리에 정의되어 있는 `sys.stdin.readline()` 메서드를 이용합니다.
 - 단, 입력 후 엔터(Enter)가 줄 바꿈 기호로 입력되므로 `rstrip()` 메서드를 함께 사용합니다.

빠르게 입력 받기

```
import sys

# 문자열 입력 받기
data = sys.stdin.readline().rstrip()
print(data)
```

자주 사용되는 표준 출력 방법

- 파이썬에서 기본 출력은 `print()` 함수를 이용합니다.
 - 각 변수를 콤마(,)를 이용하여 띄어쓰기로 구분하여 출력할 수 있습니다.
- `print()`는 기본적으로 출력 이후에 줄 바꿈을 수행합니다.
 - 줄 바꿈을 원치 않는 경우 'end' 속성을 이용할 수 있습니다.

출력을 위한 전형적인 소스코드

```
# 출력할 변수들
a = 1
b = 2
print(a, b)
print(7, end=" ")
print(8, end=" ")

# 출력할 변수
answer = 7
print("정답은 " + str(answer) + "입니다.")
```

실행 결과

```
1 2
7 8 정답은 7입니다.
```


f-string 예제

- 파이썬 3.6부터 사용 가능하며, 문자열 앞에 접두사 'f'를 붙여 사용합니다.
- 중괄호 안에 변수명을 기입하여 간단히 문자열과 정수를 함께 넣을 수 있습니다.

```
answer = 7  
print(f"정답은 {answer}입니다.")
```

실행 결과

정답은 7입니다.

조건문과 반복문

조건문

- 조건문은 프로그램의 흐름을 제어하는 문법입니다.
- 조건문을 이용해 조건에 따라서 프로그램의 로직을 설정할 수 있습니다.

조건문 예제

```
x = 15

if x >= 10:
    print("x >= 10")

if x >= 0:
    print("x >= 0")

if x >= 30:
    print("x >= 30")
```

실행 결과

```
x >= 10
x >= 0
```

들여쓰기

- 파이썬에서는 코드의 블록(Block)을 들여쓰기(Indent)로 지정합니다.
- 다음의 코드에서 ②번 라인은 무조건 실행됩니다.

```
score = 85
```

```
if score >= 70:
```

```
    print('성적이 70점 이상입니다.')
```

```
    if score >= 90:
```

```
        print('우수한 성적입니다.')
```

```
else:
```

```
    print('성적이 70점 미만입니다.')
```

```
    print('조금 더 분발하세요.')
```

← ①

성적이 70점 이상입니다.
프로그램을 종료합니다.

```
print('프로그램을 종료합니다.') ← ②
```

들여쓰기

- 탭을 사용하는 쪽과 공백 문자(space)를 여러 번 사용하는 쪽으로 두 진영이 있습니다.
 - 이에 대한 논쟁은 지금까지도 활발합니다.
- 파이썬 스타일 가이드라인에서는 **4개의 공백 문자를 사용하는 것을 표준으로 설정하고** 있습니다.

조건문의 기본 형태

- 조건문의 기본적인 형태는 **if ~ elif ~ else**입니다.
 - 조건문을 사용할 때 **elif** 혹은 **else** 부분은 경우에 따라서 사용하지 않아도 됩니다.

if 조건문 1:

조건문 1이 True일 때 실행되는 코드

elif 조건문 2:

조건문 1에 해당하지 않고, 조건문 2가 True일 때 실행되는 코드

else:

위의 모든 조건문이 모두 True 값이 아닐 때 실행되는 코드

성적 구간에 따른 학점 출력 예제

```
score = 85

if score >= 90:
    print("학점: A")
elif score >= 80:
    print("학점: B")
elif score >= 70:
    print("학점: C")
else:
    print("학점: F")
```

- 성적이 90점 이상일 때: A
- 성적이 90점 미만, 80점 이상일 때: B
- 성적이 80점 미만, 70점 이상일 때: C
- 성적이 70점 미만일 때: F

실행 결과

학점: B

비교 연산자

- 비교 연산자는 특정한 두 값을 비교할 때 이용할 수 있습니다.
 - 대입 연산자(=)와 같음 연산자(==)의 차이점에 유의하세요.

비교 연산자	설명
$X == Y$	X와 Y가 서로 같을 때 참(True)이다.
$X != Y$	X와 Y가 서로 다를 때 참(True)이다.
$X > Y$	X가 Y보다 클 때 참(True)이다.
$X < Y$	X가 Y보다 작을 때 참(True)이다.
$X >= Y$	X가 Y보다 크거나 같을 때 참(True)이다.
$X <= Y$	X가 Y보다 작거나 같을 때 참(True)이다.

논리 연산자

- 논리 연산자는 논리 값 (True/False) 사이의 연산을 수행할 때 사용합니다.

논리 연산자	설명
X and Y	X와 Y가 모두 참(True)일 때 참(True)이다.
X or Y	X와 Y 중에 하나만 참(True)이어도 참(True)이다.
not X	X가 거짓(False)일 때 참(True)이다.

파이썬의 기타 연산자

- 다수의 데이터를 담는 자료형을 위해 **in 연산자**와 **not in 연산자**가 제공됩니다.
 - 리스트, 튜플, 문자열, 딕셔너리 모두에서 사용이 가능합니다.

in 연산자와 not in 연산자	설명
x in 리스트	리스트 안에 x가 들어가 있을 때 참(True)이다.
x not in 문자열	문자열 안에 x가 들어가 있지 않을 때 참(True)이다.

파이썬의 pass 키워드

- 아무것도 처리하고 싶지 않을 때 pass 키워드를 사용합니다.
- 예시) 디버깅 과정에서 일단 조건문의 형태만 만들어 놓고 조건문을 처리하는 부분은 비워놓고 싶은 경우

```
score = 85

if score >= 80:
    pass # 나중에 작성할 소스코드
else:
    print('성적이 80점 미만입니다.')

print('프로그램을 종료합니다.')
```

프로그램을 종료합니다.

조건문의 간소화

- 조건문에서 실행될 소스코드가 한 줄인 경우, 굳이 줄 바꿈을 하지 않고도 간략하게 표현할 수 있습니다.

```
score = 85
```

```
if score >= 80: result = "Success"  
else: result = "Fail"
```

Success

- 조건부 표현식(Conditional Expression)은 if ~ else문을 한 줄에 작성할 수 있도록 해줍니다.

```
score = 85  
result = "Success" if score >= 80 else "Fail"  
  
print(result)
```

Success

파이썬 조건문 내에서의 부등식

- 다른 프로그래밍 언어와 다르게 파이썬은 조건문 안에서 수학의 부등식을 그대로 사용할 수 있습니다.
- 예를 들어 $x > 0$ *and* $x < 20$ 과 $0 < x < 20$ 은 같은 결과를 반환합니다.

코드 스타일 1

```
x = 15
if x > 0 and x < 20:
    print("x는 0 이상 20 미만의 수입니다.")
```

코드 스타일 2

```
x = 15
if 0 < x < 20:
    print("x는 0 이상 20 미만의 수입니다.")
```

- 본 책에서는 다른 언어를 다룰 때 헷갈리지 않도록 $x > 0$ *and* $x < 20$ 와 같이 비교 연산자 사이에 and, or 등의 연산자가 들어가는 형태의 코드를 이용합니다.

반복문

- 특정한 소스코드를 반복적으로 실행하고자 할 때 사용하는 문법입니다.
- 파이썬에서는 while문과 for문이 있는데, 어떤 것을 사용해도 상관 없습니다.
 - 다만 코딩 테스트에서의 실제 사용 예시를 확인해 보면, for문이 더 간결한 경우가 많습니다.

1부터 9까지 모든 정수의 합 구하기 예제 (while문)

```
i = 1
result = 0

# i가 9보다 작거나 같을 때 아래 코드를 반복적으로 실행
while i <= 9:
    result += i
    i += 1

print(result)
```

실행 결과

45

1부터 9까지 홀수의 합 구하기 예제 (while문)

```
i = 1
result = 0

# i가 9보다 작거나 같을 때 아래 코드를 반복적으로 실행
while i <= 9:
    if i % 2 == 1:
        result += i
    i += 1

print(result)
```

실행 결과

25

반복문에서의 무한 루프

- 무한 루프(Infinite Loop)란 끊임없이 반복되는 반복 구문을 의미합니다.
 - 코딩 테스트에서 무한 루프를 구현할 일은 거의 없으니 유의해야 합니다.
 - 반복문을 작성한 뒤에는 항상 반복문을 탈출할 수 있는지 확인합니다.

```
x = 10  
  
while x > 5:  
    print(x)
```

실행 결과

```
10  
10  
10  
10  
10  
10  
10  
...  
(중략)
```

반복문: for문

- 반복문으로 for문을 이용할 수도 있습니다.
- for문의 구조는 다음과 같은데, 특정한 변수를 이용하여 'in' 뒤에 오는 데이터(리스트, 튜플 등)에 포함되어 있는 원소를 첫 번째 인덱스부터 차례대로 하나씩 방문합니다.

```
for 변수 in 리스트:  
    실행할 소스코드
```

반복문: for문

```
array = [9, 8, 7, 6, 5]
```

```
for x in array:  
    print(x)
```

실행 결과

9
8
7
6
5

반복문: for문

- for문에서 연속적인 값을 차례대로 순회할 때는 `range()`를 주로 사용합니다.
 - 이때 `range(시작 값, 끝 값 + 1)` 형태로 사용합니다.
 - 인자를 하나만 넣으면 자동으로 시작 값은 0이 됩니다.

```
result = 0
```

```
# i는 1부터 9까지의 모든 값을 순회
```

```
for i in range(1, 10):
```

```
    result += i
```

```
print(result)
```

45

1부터 30까지 모든 정수의 합 구하기 예제 (for문)

```
result = 0

for i in range(1, 31):
    result += i

print(result)
```

실행 결과

465

파이썬의 continue 키워드

- 반복문에서 남은 코드의 실행을 건너뛰고, 다음 반복을 진행하고자 할 때 **continue**를 사용합니다.
- 1부터 9까지의 홀수의 합을 구할 때 다음과 같이 작성할 수 있습니다.

```
result = 0

for i in range(1, 10):
    if i % 2 == 0:
        continue
    result += i

print(result)
```

실행 결과

25

파이썬의 break 키워드

- 반복문을 즉시 탈출하고자 할 때 **break**를 사용합니다.
- 1부터 5까지의 정수를 차례대로 출력하고자 할 때 다음과 같이 작성할 수 있습니다.

```
i = 1

while True:
    print("현재 i의 값:", i)
    if i == 5:
        break
    i += 1
```

실행 결과

```
현재 i의 값: 1
현재 i의 값: 2
현재 i의 값: 3
현재 i의 값: 4
현재 i의 값: 5
```


학생들의 합격 여부 판단 예제 1) 점수가 80점만 넘으면 합격

```
scores = [90, 85, 77, 65, 97]

for i in range(5):
    if scores[i] >= 80:
        print(i + 1, "번 학생은 합격입니다.")
```

실행 결과

1 번 학생은 합격입니다.
2 번 학생은 합격입니다.
5 번 학생은 합격입니다.

학생들의 합격 여부 판단 예제 2) 특정 번호의 학생은 제외하기

```
scores = [90, 85, 77, 65, 97]
cheating_student_list = {2, 4}

for i in range(5):
    if i + 1 in cheating_student_list:
        continue
    if scores[i] >= 80:
        print(i + 1, "번 학생은 합격입니다.")
```

실행 결과

1 번 학생은 합격입니다.
5 번 학생은 합격입니다.

중첩된 반복문: 구구단 예제

```
for i in range(2, 10):  
    for j in range(1, 10):  
        print(i, "X", j, "=", i * j)  
    print()
```

실행 결과

```
2 X 1 = 2  
2 X 2 = 4  
2 X 3 = 6  
2 X 4 = 8  
2 X 5 = 10  
2 X 6 = 12  
2 X 7 = 14  
2 X 8 = 16  
2 X 9 = 18
```

```
3 X 1 = 3  
3 X 2 = 6  
3 X 3 = 9  
3 X 4 = 12  
3 X 5 = 15  
...  
(중략)
```

함수와 람다 표현식

함수

- 함수(Function)란 특정한 작업을 하나의 단위로 묶어 놓은 것을 의미합니다.
- 함수를 사용하면 불필요한 소스코드의 반복을 줄일 수 있습니다.



함수의 종류

- 내장 함수: 파이썬이 기본적으로 제공하는 함수
- 사용자 정의 함수: 개발자가 직접 정의하여 사용할 수 있는 함수

함수 정의하기

- 프로그램에는 똑같은 코드가 반복적으로 사용되어야 할 때가 많습니다.
- 함수를 사용하면 소스코드의 길이를 줄일 수 있습니다.
 - 매개변수: 함수 내부에서 사용할 변수
 - 반환 값: 함수에서 처리 된 결과를 반환

```
def 함수명(매개변수):  
    실행할 소스코드  
    return 반환 값
```

더하기 함수 예시

- 더하기 함수 예시 1)

```
def add(a, b):  
    return a + b  
  
print(add(3, 7))
```

10

- 더하기 함수 예시 2)

```
def add(a, b):  
    print('함수의 결과:', a + b)  
  
add(3, 7)
```

함수의 결과: 10

파라미터 지정하기

- 파라미터의 변수를 직접 지정할 수 있습니다.
 - 이 경우 매개변수의 순서가 달라도 상관이 없습니다.

```
def add(a, b):  
    print('함수의 결과:', a + b)
```

```
add(b = 3, a = 7)
```

함수의 결과: 10

global 키워드

- global 키워드로 변수를 지정하면 해당 함수에서는 지역 변수를 만들지 않고, 함수 바깥에 선언된 변수를 바로 참조하게 됩니다.

```
a = 0

def func():
    global a
    a += 1

for i in range(10):
    func()

print(a)
```

10

여러 개의 반환 값

- 파이썬에서 함수는 여러 개의 반환 값을 가질 수 있습니다.

```
def operator(a, b):  
    add_var = a + b  
    subtract_var = a - b  
    multiply_var = a * b  
    divide_var = a / b  
    return add_var, subtract_var, multiply_var, divide_var  
  
a, b, c, d = operator(7, 3)  
print(a, b, c, d)
```

람다 표현식

- 람다 표현식을 이용하면 함수를 간단하게 작성할 수 있습니다.
 - 특정한 기능을 수행하는 함수를 한 줄에 작성할 수 있다는 점이 특징입니다.

```
def add(a, b):  
    return a + b
```

```
# 일반적인 add() 메서드 사용  
print(add(3, 7))
```

```
# 람다 표현식으로 구현한 add() 메서드  
print((lambda a, b: a + b)(3, 7))
```

10

10

람다 표현식 예시: 내장 함수에서 자주 사용되는 람다 함수

```
array = [('홍길동', 50), ('이순신', 32), ('아무개', 74)]
```

```
def my_key(x):  
    return x[1]
```

```
print(sorted(array, key=my_key))  
print(sorted(array, key=lambda x: x[1]))
```

실행 결과

```
[('이순신', 32), ('홍길동', 50), ('아무개', 74)]  
[('이순신', 32), ('홍길동', 50), ('아무개', 74)]
```

람다 표현식 예시: 여러 개의 리스트에 적용

```
list1 = [1, 2, 3, 4, 5]
list2 = [6, 7, 8, 9, 10]

result = map(lambda a, b: a + b, list1, list2)

print(list(result))
```

실행 결과

```
[7, 9, 11, 13, 15]
```

실전에서 유용한 표준 라이브러리

실전에서 유용한 표준 라이브러리

- **내장 함수:** 기본 입출력 함수부터 정렬 함수까지 기본적인 함수들을 제공합니다.
 - 파이썬 프로그램을 작성할 때 없어서는 안 되는 필수적인 기능을 포함하고 있습니다.
- **itertools:** 파이썬에서 반복되는 형태의 데이터를 처리하기 위한 유용한 기능들을 제공합니다.
 - 특히 순열과 조합 라이브러리는 코딩 테스트에서 자주 사용됩니다.
- **heapq:** 힙(Heap) 자료구조를 제공합니다.
 - 일반적으로 우선순위 큐 기능을 구현하기 위해 사용됩니다.
- **bisect:** 이진 탐색(Binary Search) 기능을 제공합니다.
- **collections:** 덱(deque), 카운터(Counter) 등의 유용한 자료구조를 포함합니다.
- **math:** 필수적인 수학적 기능을 제공합니다.
 - 팩토리얼, 제곱근, 최대공약수(GCD), 삼각함수 관련 함수부터 파이(pi)와 같은 상수를 포함합니다.

자주 사용되는 내장 함수

```
# sum()
result = sum([1, 2, 3, 4, 5])
print(result)

# min(), max()
min_result = min(7, 3, 5, 2)
max_result = max(7, 3, 5, 2)
print(min_result, max_result)

# eval()
result = eval("(3+5)*7")
print(result)
```

실행 결과

15
2 7
56

자주 사용되는 내장 함수

```
# sorted()
result = sorted([9, 1, 8, 5, 4])
reverse_result = sorted([9, 1, 8, 5, 4], reverse=True)
print(result)
print(reverse_result)

# sorted() with key
array = [('홍길동', 35), ('이순신', 75), ('아무개', 50)]
result = sorted(array, key=lambda x: x[1], reverse=True)
print(result)
```

실행 결과

```
[1, 4, 5, 8, 9]
[9, 8, 5, 4, 1]
[('이순신', 75), ('아무개', 50), ('홍길동', 35)]
```

순열과 조합

- 모든 경우의 수를 고려해야 할 때 어떤 라이브러리를 효과적으로 사용할 수 있을까요?
- **순열**: 서로 다른 n 개에서 서로 다른 r 개를 선택하여 일렬로 나열하는 것
 - {'A', 'B', 'C'}에서 세 개를 선택하여 나열하는 경우: 'ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA'
- **조합**: 서로 다른 n 개에서 순서에 상관 없이 서로 다른 r 개를 선택하는 것
 - {'A', 'B', 'C'}에서 순서를 고려하지 않고 두 개를 뽑는 경우: 'AB', 'AC', 'BC'

$$\text{순열의 수: } nPr = n * (n - 1) * (n - 2) * \dots * (n - r + 1)$$

$$\text{조합의 수: } nCr = \frac{n * (n - 1) * (n - 2) * \dots * (n - r + 1)}{r!}$$

순열과 조합

- **순열**: 서로 다른 n 개에서 서로 다른 r 개를 선택하여 일렬로 나열하는 것
 - {'A', 'B', 'C'}에서 두 개를 선택하여 나열하는 경우: 'ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA'

```
from itertools import permutations

data = ['A', 'B', 'C'] # 데이터 준비

result = list(permutations(data, 3)) # 모든 순열 구하기
print(result)
```

실행 결과: [('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'), ('C', 'A', 'B'), ('C', 'B', 'A')]

순열과 조합

- 조합: 서로 다른 n개에서 순서에 상관 없이 서로 다른 r개를 선택하는 것
 - {'A', 'B', 'C'}에서 순서를 고려하지 않고 두 개를 뽑는 경우: 'AB', 'AC', 'BC'

```
from itertools import combinations

data = ['A', 'B', 'C'] # 데이터 준비

result = list(combinations(data, 2)) # 2개를 뽑는 모든 조합 구하기
print(result)
```

실행 결과: `[('A', 'B'), ('A', 'C'), ('B', 'C')]`

중복 순열과 중복 조합

```
from itertools import product

data = ['A', 'B', 'C'] # 데이터 준비

result = list(product(data, repeat=2)) # 2개를 뽑는 모든 순열 구하기 (중복 허용)
print(result)
```

```
from itertools import combinations_with_replacement

data = ['A', 'B', 'C'] # 데이터 준비

result = list(combinations_with_replacement(data, 2)) # 2개를 뽑는 모든 조합 구하기 (중복 허용)
print(result)
```

Counter

- 파이썬 collections 라이브러리의 **Counter**는 등장 횟수를 세는 기능을 제공합니다.
- 리스트와 같은 반복 가능한(iterable) 객체가 주어졌을 때 내부의 원소가 몇 번씩 등장했는지를 알려줍니다.

```
from collections import Counter

counter = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])

print(counter['blue']) # 'blue'가 등장한 횟수 출력
print(counter['green']) # 'green'이 등장한 횟수 출력
print(dict(counter)) # 사전 자료형으로 반환
```

실행 결과:

```
3
1
{'red': 2, 'blue': 3, 'green': 1}
```

최대 공약수와 최소 공배수

- 최대 공약수를 구해야 할 때는 math 라이브러리의 gcd() 함수를 이용할 수 있습니다.

```
import math

# 최소 공배수(LCM)를 구하는 함수
def lcm(a, b):
    return a * b // math.gcd(a, b)

a = 21
b = 14

print(math.gcd(21, 14)) # 최대 공약수(GCD) 계산
print(lcm(21, 14)) # 최소 공배수(LCM) 계산
```

실행 결과:

```
7
42
```