

파이썬 기초

박진형

AI

교육 일정

환경구축 / 자료형과 제어문	1일
클래스와 함수	2일
반복 가능 객체와 예외처리	3일

파이썬이란?

- › 1990년 암스테르담의 귀도 반 로섬(Guido Van Rossum)이 개발한 인터프리터 언어
- › 사전적 의미로는 고대 신화에 나오는 파르나소스 산의 동굴에 살던 큰 뱀을 뜻함
- › 구글에서 만든 많은 소프트웨어가 파이썬으로 작성
- › 인스타그램, 드롭박스 또한 파이썬으로 작성
- › 공동 작업과 유지 보수가 매우 쉽고 편함
- › 이미 다른 언어로 작성된 많은 프로그램과 모듈이 파이썬으로 재구성 되고 있음
- › AI 프레임워크인 Pytorch, Tensorflow 또한 파이썬 사용

파이썬의 특징

- › 문법이 쉬우며 빠르게 배울 수 있음
 - 문법 자체가 아주 쉽고 간결하며 사람의 사고 체계와 매우 닮음
- › 무료 언어이지만 강력함
 - 오픈 소스로써 사용료 걱정 없이 언제 어디서든 사용 가능
- › 간결하며 개발속도가 빠름
 - 다른 프로그래밍 언어가 100가지 방법으로 하나의 일을 처리 할 수 있다면 파이썬은 가장 좋은 방법 1가지만 사용하는 것을 선호
 - 다른 사람이 작업한 소스 코드도 한눈에 들어와 이해하기 쉽기 때문에 공동 작업과 유지 보수가 아주 쉽고 편함

파이썬으로 무엇을 할 수 있을까?

- › 시스템 유틸리티 제작
- › GUI 프로그래밍
- › C/C++ 와의 결합
- › 웹 프로그래밍
- › 수치 연산 프로그래밍
- › 데이터 베이스 프로그래밍
- › 데이터 분석, 사물 인터넷
- › 인공지능 프로그래밍



개발 환경 구축

Anaconda, python, jupyter notebook

개발 / 운용 환경 구축 가이드



Python을 사용하는 이유

1. 비교적 읽고 쓰기 쉬운 프로그래밍 언어
2. 효율적인 메모리 관리 기능
3. 풍부한 머신러닝 프레임워크와 라이브러리



ANACONDA®

ANACONDA

1. 핵심적인 과학 및 수학 Python 라이브러리 포함한 패키지이며, 머신러닝에 필수적인 툴
2. 개발환경을 Anaconda Navigator 라는 하나의 통합 공간에서 편리하게 설치할 수 있다.

Anaconda Installer 설치

1

<https://www.anaconda.com/products/individual>

01. Anaconda 주소로 접속



Individual Edition

Your data science toolkit

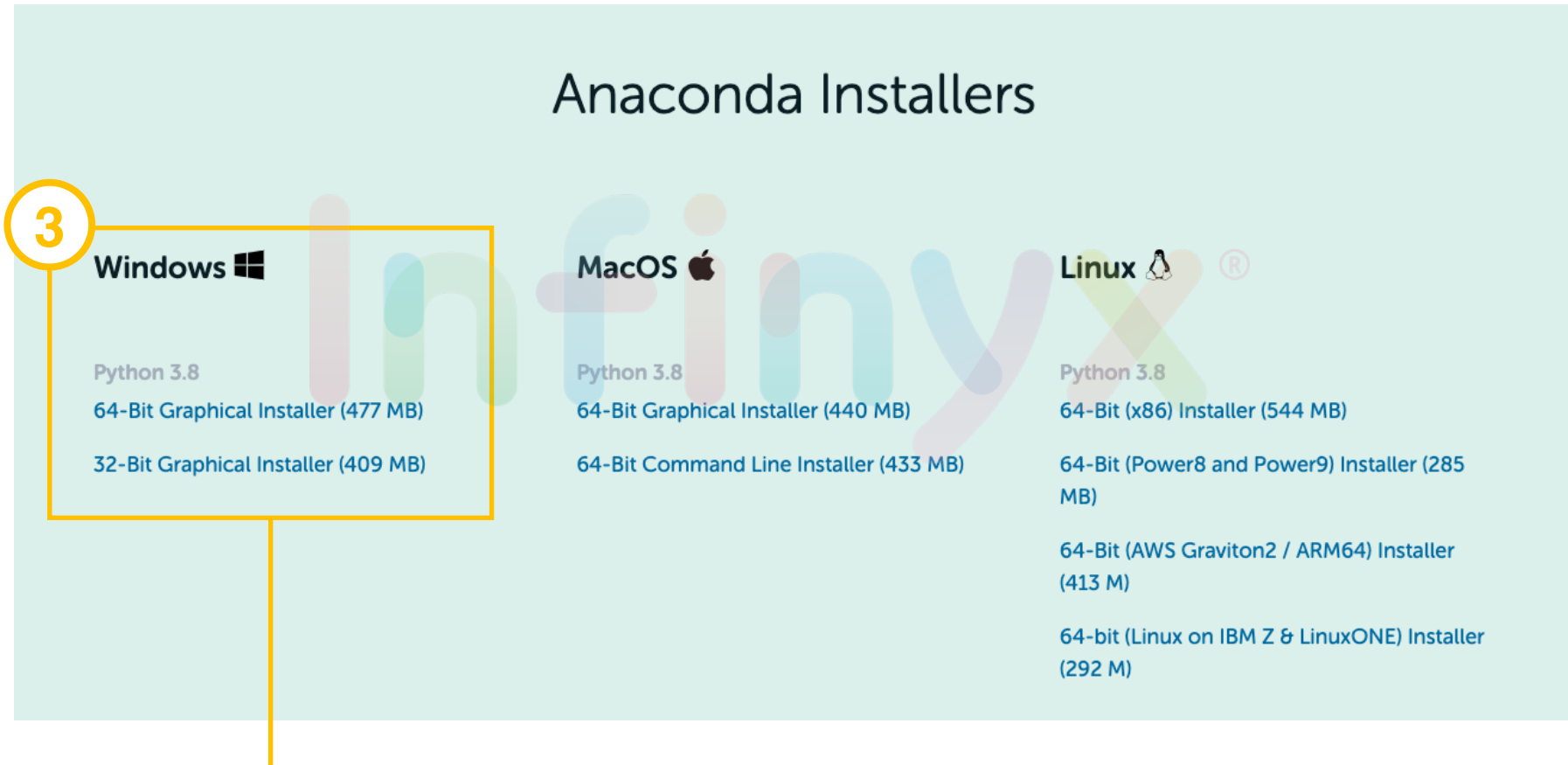
With over 25 million users worldwide, the open-source Individual Edition (Distribution) is the easiest way to perform Python/R data science and machine learning on a single machine. Developed for solo practitioners, it is the toolkit that equips you to work with thousands of open-source packages and libraries.

2

Download

02. Download 버튼 클릭

Anaconda Installer 설치



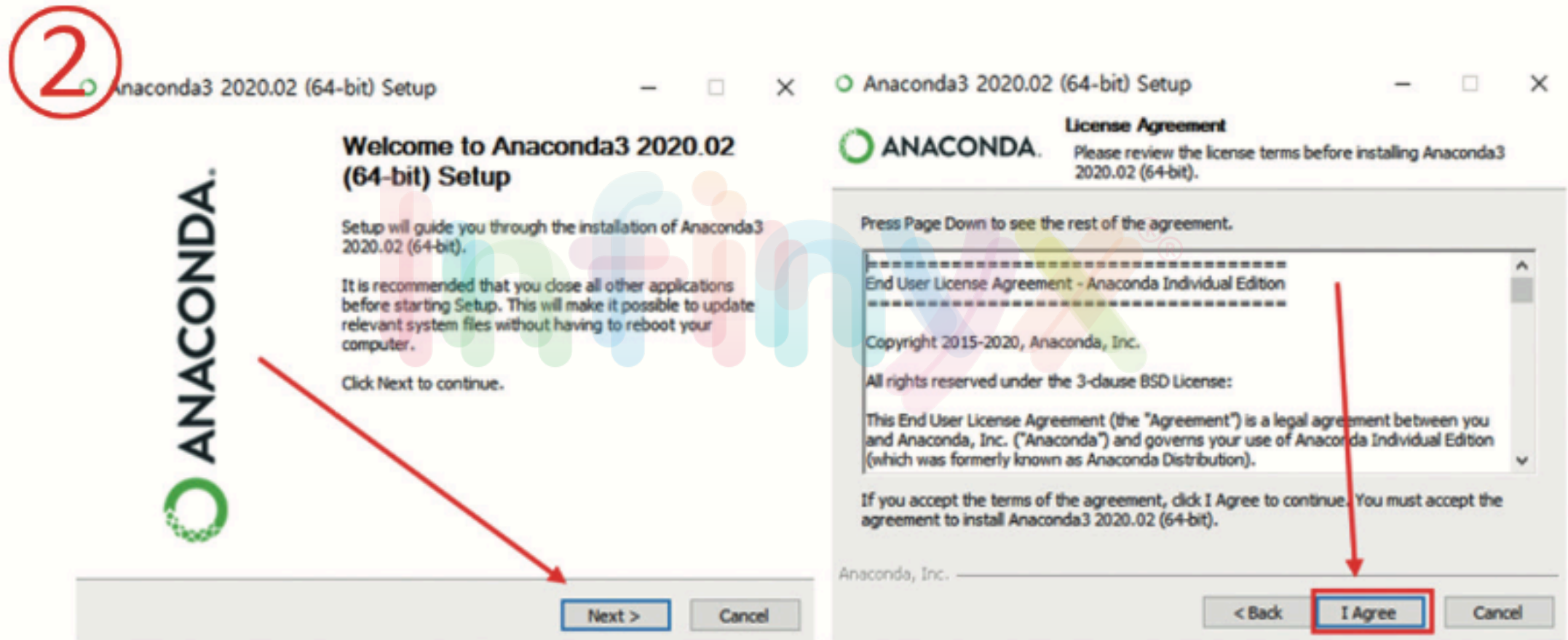
03. Windows, MacOS, Linux 버전 중 자신의 PC 운영체제와 일치하는 버전을 선택하여 Installer를 다운로드

Anaconda 실행 파일 설치 (Windows 64-bit 기준)



01. 설치된 실행 파일을 열 때는 [마우스 오른쪽] - [관리자 권한으로 실행]을 클릭

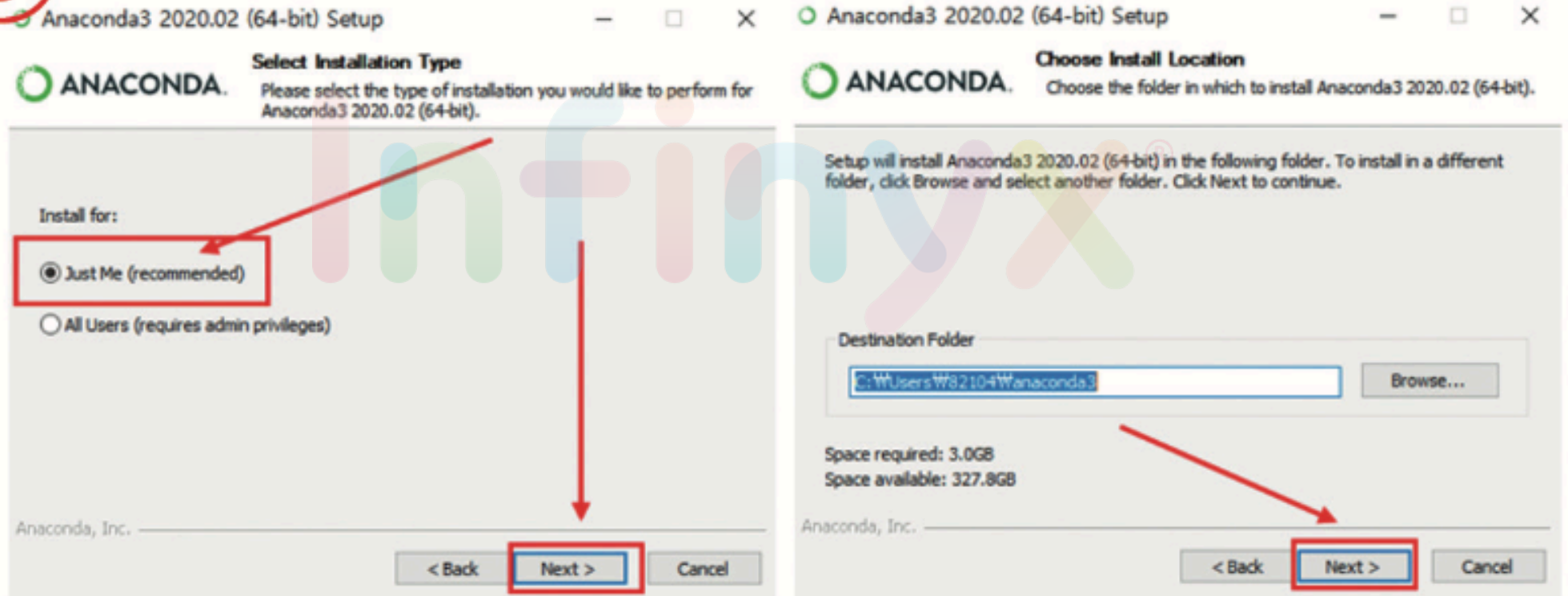
Anaconda 실행 파일 설치 (Windows 64-bit 기준)



02. 별도의 설정없이 Next 버튼을 눌러 다음 단계로 진행

Anaconda 실행 파일 설치 (Windows 64-bit 기준)

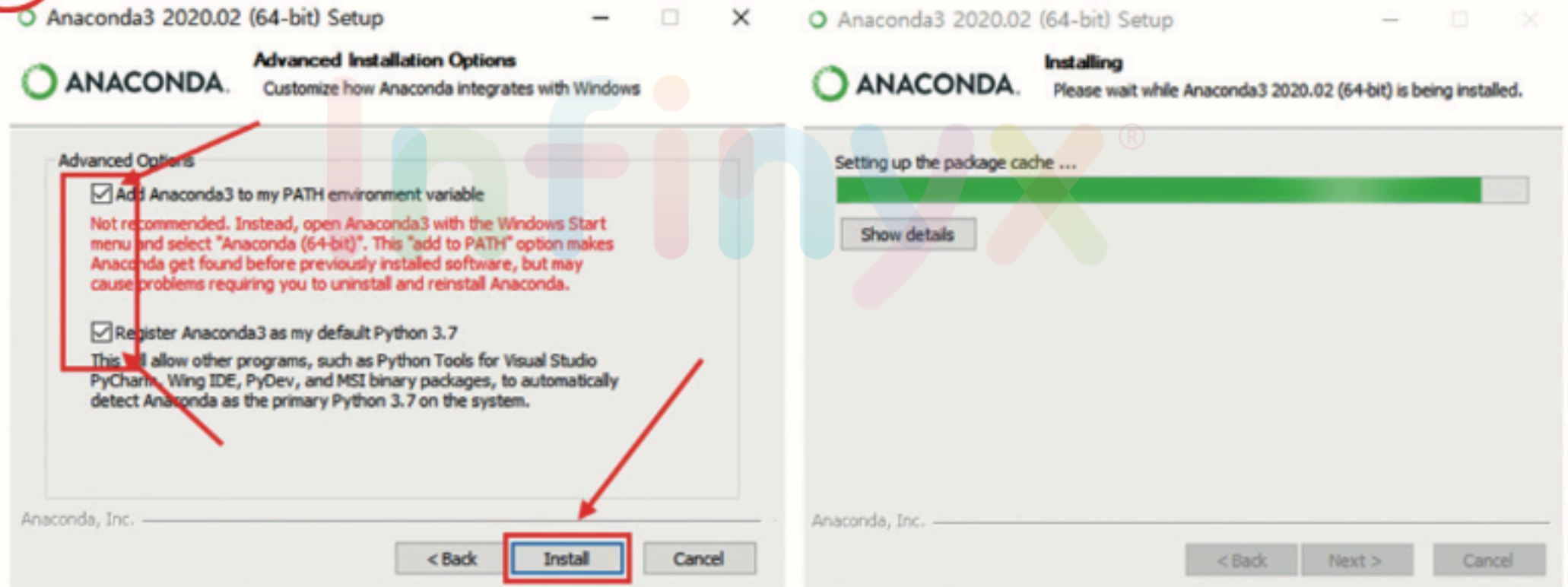
③



03. 'Just me(recommended)'를 체크하고 넘어가서 다운로드 경로는 별도로 설정하지 않고 바로 다음단계로 진행

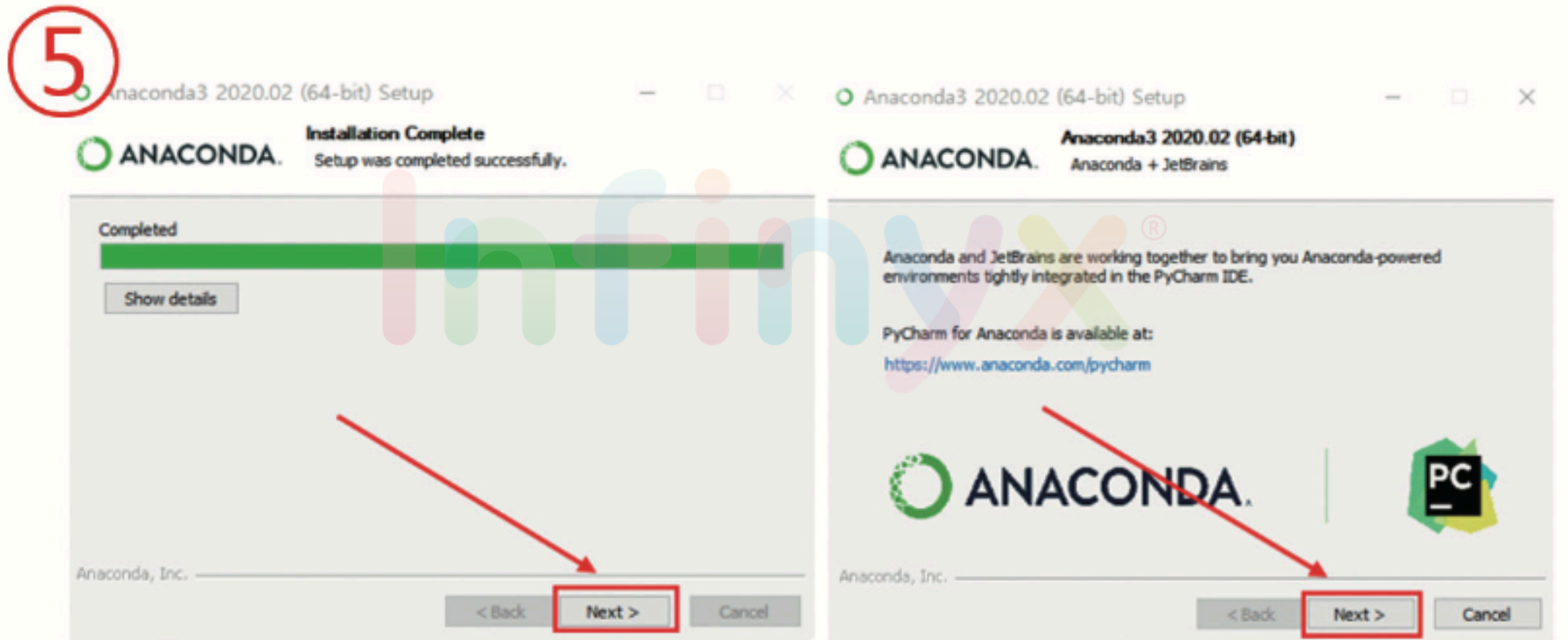
Anaconda 실행 파일 설치 (Windows 64-bit 기준)

4



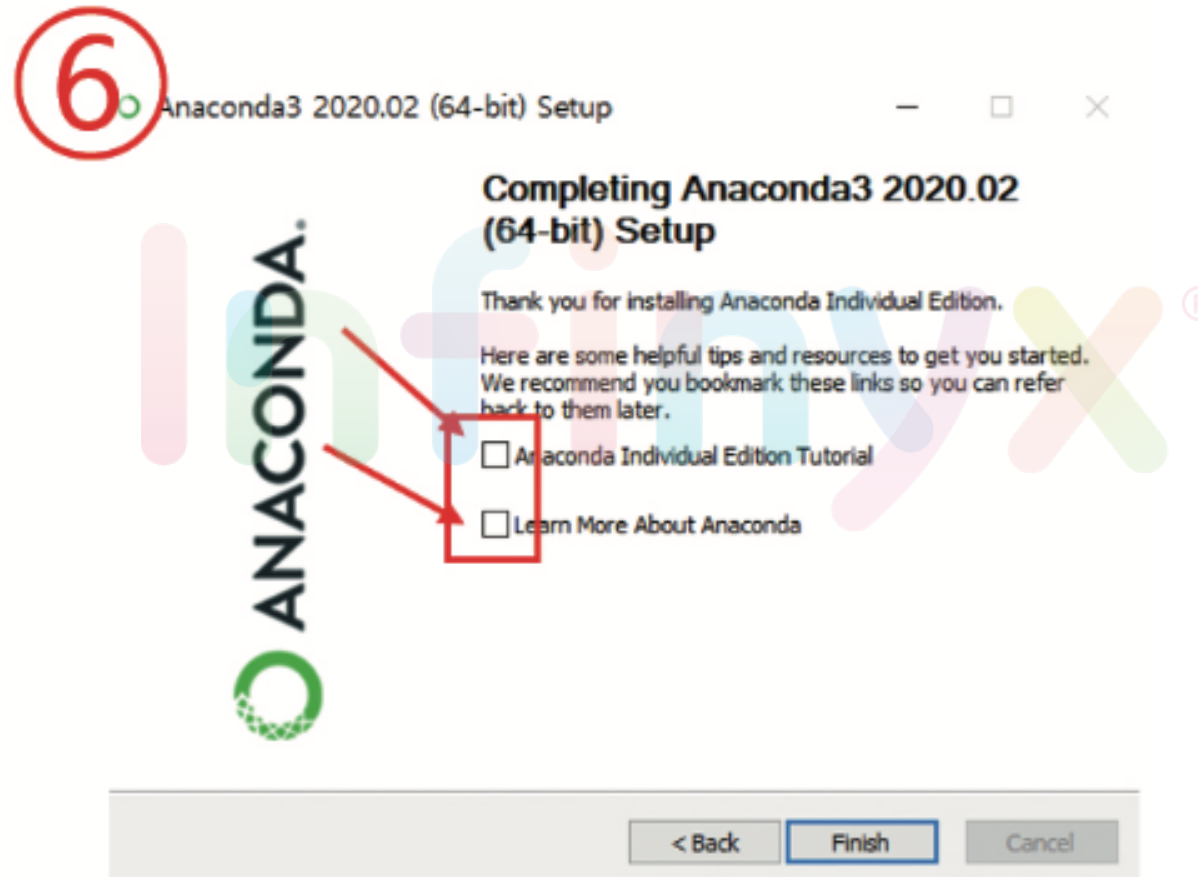
04. 체크박스를 모두 체크한 후 Install을 클릭

Anaconda 실행 파일 설치 (Windows 64-bit 기준)



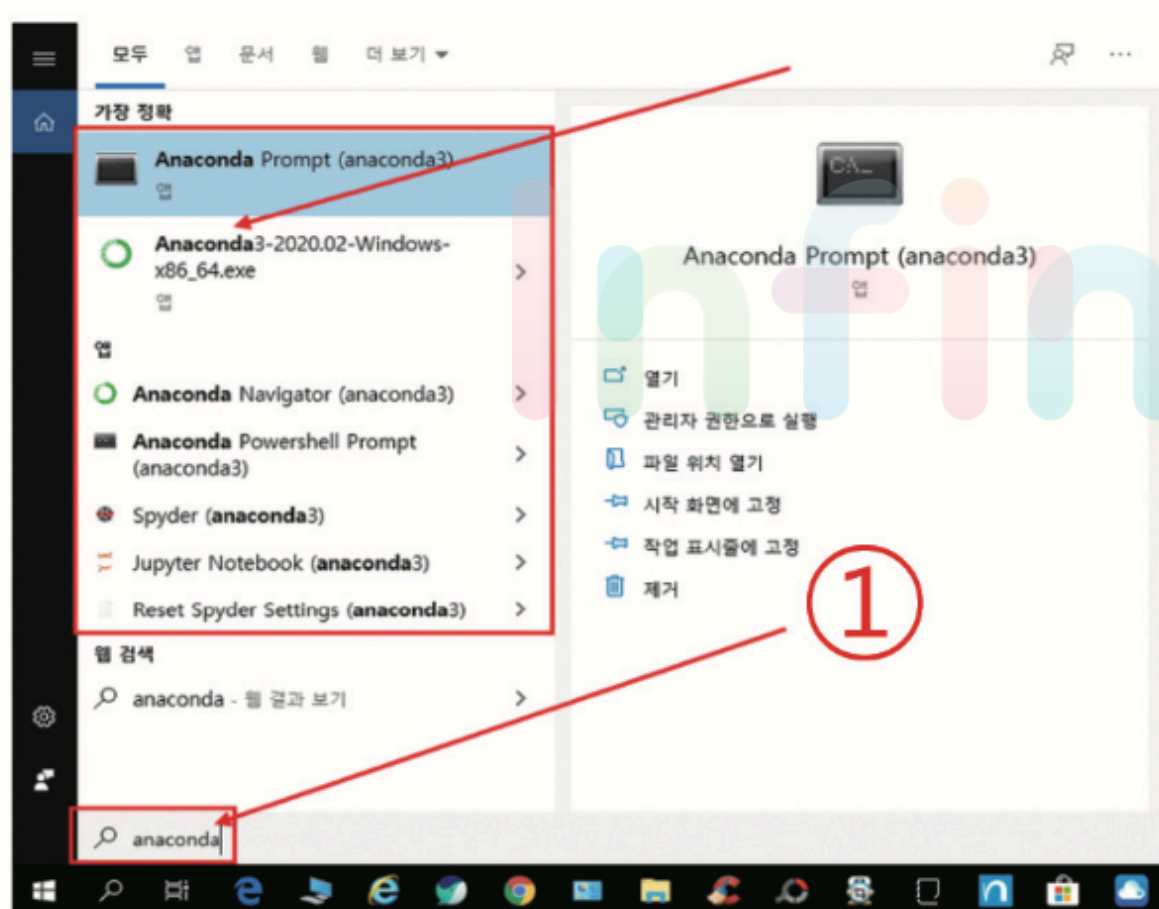
05. 별도의 설정 없이 Next 버튼을 눌러 진행한다.

Anaconda 실행 파일 설치 (Windows 64-bit 기준)



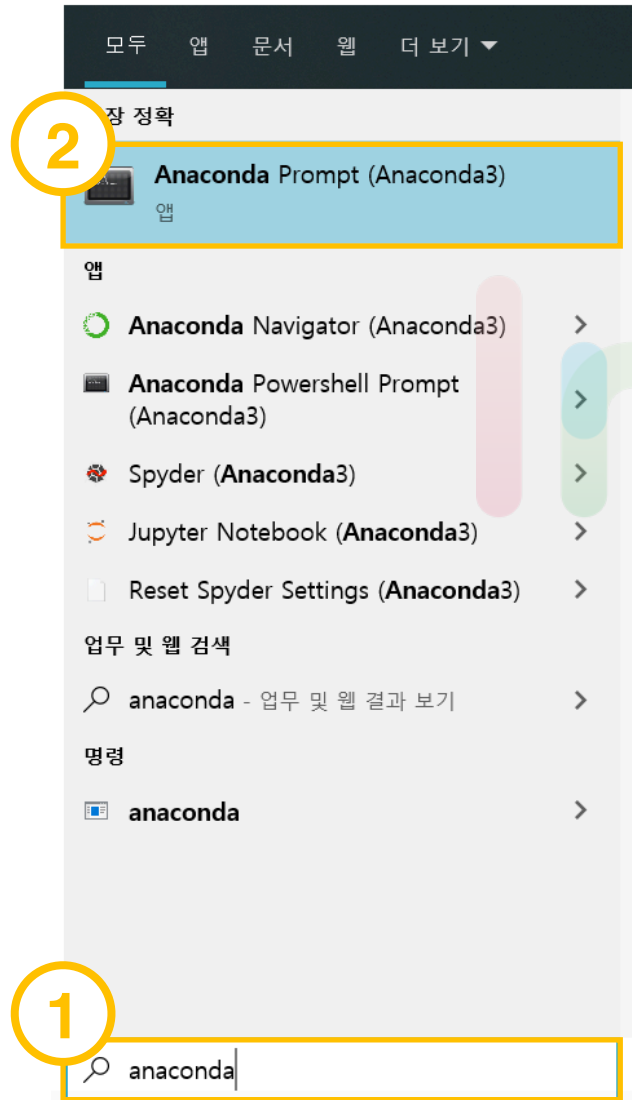
06. 두 가지 체크박스는 Anaconda 와 관련된 튜토리얼과 교육자료에 관한 것을 설치 및 연결할 것인지 묻는 창이므로, 체크를 전부 해제 하고 Finish 클릭

Anaconda 설치 확인



01. 설치가 완료되면 Windows 좌측 하단의 돋보기를 클릭하여 Anaconda를 입력해 설치 여부 확인

Anaconda 가상환경 구축



01. 설치가 완료되면 Windows 좌측 하단의 돋보기를 클릭하여 Anaconda 입력

02. Anaconda Prompt 열 때는 [마우스 오른쪽] - [관리자 권한으로 실행] 클릭

Anaconda env 구축 (생성)

env : python 독립적인 가상의 실행 환경(Environment)

1

관리자: Anaconda Prompt (anaconda3)

```
(base) C:\Windows\system32>conda create -n AI python=3.8
```

01. Command 창에 다음 명령어 입력

```
$ conda create -n AI python=3.8
```

Anaconda env 구축 (리스트 보기)

2

```
(base) C:\Windows\system32>conda env list
# conda environments:
#
base * C:\Users\yugio\anaconda3
AI C:\Users\yugio\anaconda3\envs\AI
```

02. Command 창에 다음 명령어 입력

\$ conda env list

Anaconda env 구축 (활성화)

3

```
(base) C:\Windows\system32>conda activate AI  
(AI) C:\Windows\system32>
```

03. Command 창에 다음 명령어 입력

\$ conda activate AI

Anaconda env에 라이브러리 설치

01. Conda activate 접속 후 다음을 이용하여 교육에 필요한 라이브러리 설치

```
$ conda pip install
```



Anaconda env에 라이브러리 설치

01. Conda activate 접속 후 다음을 이용하여 교육에 필요한 라이브러리 설치

\$ conda pip install

1. Torch , torch vision conda install

→ conda install pytorch==1.7.1 torchvision==0.8.2 torchaudio==0.7.2 cudatoolkit=11.0 -c
pytorch

2. Open-cv install

→ pip install opencv-python

3. Matplotlib

→ pip install matplotlib

4. Scikit-learn

→ pip install scikit-learn

파이썬으로 무엇을 할 수 있을까?

- › conda 설치

<https://docs.conda.io/en/latest/miniconda.html#windows-installers>

- › \$ conda create -n NAME python=3.7

- › \$ conda activate NAME

- › \$ conda install jupyter

- › \$ jupyter notebook

파이썬 자료형

숫자, 문자열, 리스트, 튜플, 딕셔너리, 집합

숫자 자료형

› 숫자 형태로 이루어진 자료형

항목	예시
정수	123, -345, 0
실수	1.45, -1.4, 3.4e10
8진수	0o34, 0o25
16진수	0x2A, 0xFF

› 숫자 자료형의 기본적인 사용 방법

```
>>> a = 123  
>>> b = -345  
>>> c = 0
```

정수형(Integer)

› 정수를 뜻하는 자료형

– 양의 정수, 음의 정수, 0 이 정수형에 해당

```
>>> a = 123
```

```
>>> b = -345
```

```
>>> c = 0
```

```
>>> a
```

```
123
```

```
>>> b
```

```
-345
```

```
>>> c
```

```
0
```

```
>>> type(a)
```

```
<class 'int'>
```

```
>>> type(b)
```

```
<class 'int'>
```

```
>>> type(c)
```

```
<class 'int'>
```

```
# 8진수와 16진수
```

```
>>> d = 0o27
```

```
>>> e = 0xFF
```

실수형(Float)

› 소수점이 포함된 숫자 자료형

– 양의 정수, 음의 정수, 0 이 정수형에 해당

```
>>> a = 1.2
```

```
>>> b = -3.4
```

```
>>> c = 0.0
```

```
>>> a
```

```
1.2
```

```
>>> b
```

```
-3.3
```

```
>>> c
```

```
0.0
```

```
>>> type(a)
```

```
<class 'float'>
```

```
>>> type(b)
```

```
<class 'float'>
```

```
>>> type(c)
```

```
<class 'float'>
```

컴퓨터식 지수 표현 방식

```
>>> d = 5.6E10
```

```
>>> e = 7.8E-3
```

숫자 자료형 활용 연산자

› 사칙연산 (+, -, *, /)

```
>>> a = 3
```

```
>>> b = 2
```

```
>>> a + b
```

```
5
```

```
>>> a - b
```

```
1
```

```
>>> a * b
```

```
6
```

```
>>> a / b
```

```
1.5
```

```
# 우선 순위 확인
```

```
>>> a+b*a/b
```

숫자 자료형 활용 연산자

› 제공 연산자 **

– $x^{**}y$ 는 (x^y) 를 뜻함

› 나눗셈 나머지 반환 연산자 %

› 나눗셈 몫 반환 연산자 //

```
>>> a = 3
```

```
>>> b = 2
```

```
>>> a ** b
```

```
9
```

```
>>> a % b
```

```
1
```

```
>>> a // b
```

```
1
```

문자열 자료형(String)

› 문자, 단어 등으로 구성된 문자들의 집합

```
>>> a = "hello world"  
>>> b = '123'  
>>> type(a)  
>>> type(b)
```

› 문자열 종류

- 큰따옴표(“ ”)
- 작은따옴표(‘ ’)
- 큰따옴표 3개 연속 사용(“”” ””””)
- 작은따옴표 3개 연속 사용(“ ” ” ” ”)

문자열 자료형(String)

› 문자열 안에 작은 따옴표 또는 큰 따옴표를 포함시키고 싶을 때

```
>>> a = " I'm favorite food is kimchi. "  
>>> b = ' "Python is very easy." he says. '
```

백슬래시(\)를 사용해서 포함 시킬 수 있음

```
>>> c = ' I\'m favorite food is kimchi. '  
>>> d = "\"Python is very easy.\" he says. "
```

문자열 자료형(String)

- › 여러 줄인 문자열을 변수에 대입하고 싶을 때
 - 이스케이프 코드 \n 사용
 - 연속된 따옴표 사용

```
>>> a = "AI is fun.\nbut is difficult."
```

```
>>> b = '''
```

```
... AI is fun.
```

```
... but is difficult.
```

```
... '''
```

```
>>> c = """
```

```
... AI is fun.
```

```
... but is difficult.
```

```
... """
```

```
>>> print(a)
```


이스케이프 코드

- › 미리 정의해 둔 "문자 조합"
- › 주로 출력물을 보기 좋게 정렬하는 용도로 사용

코드	설명
\n	문자열 안에서 줄을 바꿀 때 사용
\t	문자열 사이에 탭 간격을 줄 때 사용
\\	문자 \를 그대로 표현할 때 사용
\'	작은따옴표를 그대로 표현할 때 사용
\"	큰따옴표를 그대로 표현할 때 사용
\r	캐리지 리턴(줄 바꿈 문자, 커서를 가장 앞으로 이동)
\f	폼 피드(줄 바꿈 문자, 커서를 다음 줄로 이동)
\a	벨 소리(출력할 때 PC 스피커에서 '뽕' 소리가 난다)
\b	백 스페이스
\000	널 문자

문자열 연산

› 문자열 더해서 연결 (Concatenation)

```
>>> a = "Python"  
>>> b = " is very easy."  
>>> a + b
```

› 문자열 곱하기

```
>>> a = "Python"  
>>> a * 2  
# 응용하기  
>>> a = "=" * 50  
>>> b = "My Program"  
>>> print(a + "\n" + b + "\n" + a)
```

문자열 인덱싱

- › 문자열은 문자의 조합
- › 각 문자마다 문자의 순서 번호가 존재

```
>>> a = "Python is very easy."  
>>> a[0]  
>>> a[7]
```

- › 파이썬은 0부터 숫자를 카운트
- › 거꾸로 문자를 인덱싱 가능

```
>>> a = "Python is very easy."  
>>> a[-1]  
>>> a[-10]
```

문자열 슬라이싱

- › 문자열에서 한 문자가 아닌 여러 문자를 뽑기
 - 인덱싱으로 여러 문자 뽑기

```
>>> a = "Python is very easy."  
>>> b = a[10] + a[11] + a[12] + a[14]  
>>> b
```

- 슬라이싱으로 여러 문자 뽑기

```
>>> a = "Python is very easy."  
>>> b = a[10:15]  
>>> b
```

- › 슬라이싱은 끝 번호를 포함 하지 않음

문자열 슬라이싱

› 슬라이싱의 여러 방법

- 번호를 생략하면 0 또는 끝까지로 표현
- 인덱싱과 마찬가지로 마이너스 기호 사용 가능

```
>>> a = "Python is very easy."
```

```
>>> b = a[10:]
```

```
>>> b
```

```
>>> c = a[:10]
```

```
>>> c
```

```
>>> a[:]
```

```
>>> a[7:-6]
```

문자열 수정

- › 문자열에서 특정한 문자 수정 및 추가

```
>>> a = "Python"  
# Error  
>>> a[1] = "y"  
>>> a[:1] + "y" + a[2:]
```

- › 문자열은 요솟값을 바꿀 수 없음
(Immutable한 자료형)
- › string Class에서 replace() 와 같은 함수를 사용하여
수정 가능

문자열 포매팅(Formatting)

› 문자열 안의 특정한 값을 바꿔야 할 때 필요한 기법

```
>>> "I eat %d banana." % 2
>>> "I eat %s banana." % "2"
>>> number = 2
>>> "I eat %d banana." % number
# 2개 이상의 값 넣기
>>> number = 12
>>> day = "two"
>>> "I ate %d bananas. so I was sick for %s days." %(number, day)
```

코드	설명
%s	문자열(String)
%c	문자 1개(Character)
%d	정수(Integer)
%f	부동소수(Floating-point)
%o	8진수
%x	16진수
%%	Literal % (문자 % 자체)

문자열 포매팅(Formatting)

› %s 포맷 코드

– 어떤 형태의 값이든 문자열로 변환 가능

```
>>> "I eat %s banana." % "2"  
>>> "I eat %s banana." % 2  
>>> "Rate is %s" % 3.234
```

› % 사용 방법

Error

```
>>> "Error is %d%." % 100  
>>> "Error is %d%%." % 0
```


문자열 포매팅(Formatting)

› 포맷 코드와 숫자 함께 사용

– 정렬과 공백

```
>>> "%10s" % "apple"
```

```
# 왼쪽 정렬
```

```
>>> "%-10s banana" % "apple"
```

– 소수점 표현

```
>>> "%0.4f" % 3.141592
```

format함수 포매팅

› 숫자 바로 대입

```
>>> "I eat {0} banana.".format(2)
```

```
>>> "I eat {} banana.".format(2)
```

› 문자열 바로 대입

```
>>> "I eat {0} banana.".format("two")
```

```
>>> number = "two"
```

```
>>> "I eat {} banana.".format(number)
```

› 2개 이상의 값 넣기

```
>>> number = 12
```

```
>>> day = "two"
```

```
>>> "I ate {0} bananas. so I was sick for {1} days.".format(number, day)
```

format함수 포매팅

› 이름으로 넣기

```
>>> "I ate {number} bananas. so I was sick for {day} days.".format(number=12, day="two")
```

› 인덱스 번호와 이름을 혼용해서 넣기

```
>>> "I ate {0} bananas. so I was sick for {day} days.".format(12, day="two")
```

› 인덱스 번호만 사용 할 경우 보통은 생략

```
>>> number = 12
```

```
>>> day = "two"
```

```
>>> "I ate {} bananas. so I was sick for {} days.".format(number, day)
```

format함수 포매팅

정렬

– 왼쪽 정렬(<)

```
>>> "{0:<10}".format("hi")
```

– 오른쪽 정렬(>)

```
>>> "{0:>10}".format("hi")
```

– 가운데 정렬(^)

```
>>> "{0:^10}".format("hi")
```

– 공백 채우기

```
>>> "{0:=^10}".format("hi")
```

```
>>> "{0:!  
<10}".format("hi")
```

f 문자열 포매팅

› 파이썬 3.6 버전 이상부터 사용 가능

```
>>> number = 12  
>>> day = "two"  
>>> f"I ate {number} bananas. so I was sick for {day} days."
```

› f 문자열 포매팅은 표현식을 지원

```
>>> number = 3  
>>> f"I ate {number+5} bananas."
```

› 정렬 방식은 format 함수와 동일

```
>>> f"{'hi':<10}"  
>>> f"{'hi':^10}"
```

문자열 함수

› 문자 개수 세기 (count)

```
>>> a = "apple"  
>>> a.count('p')
```

› 문자 위치 찾기

– find (문자가 존재하지 않는다면 -1을 반환)

```
>>> a = "hope"  
>>> a.find("p")  
>>> a.find("k")
```

– index(문자가 존재하지 않는다면 에러 발생)

```
>>> a = "hope"  
>>> a.index("p")  
>>> a.index("k")
```

문자열 함수

› 문자열 삽입 (join)

```
>>> ",".join("abcd")
```

› 소문자를 대문자로 변환 (upper)

```
>>> a = "hi"  
>>> a.upper()
```

› 대문자를 소문자로 변환 (lower)

```
>>> a = "HI"  
>>> a.lower()
```

문자열 함수

› 왼쪽 공백 지우기 (lstrip)

```
>>> a = " hi "  
>>> a.lstrip()
```

› 오른쪽 공백 지우기 (rstrip)

```
>>> a = " hi "  
>>> a.rstrip()
```

› 양쪽 공백 지우기 (strip)

```
>>> a = " hi "  
>>> a.strip()
```


문자열 함수

› 문자열 바꾸기 (replace)

– 단 같은 문자가 있을 경우, 모든 문자가 변경

```
>>> a = "Python"
>>> a.replace("i", "y")
>>> b = "apple"
>>> b.replace("p", "z")
```

› 오른쪽 공백 지우기 (split)

```
>>> a = "Life is too short"
>>> a.split()
>>> b = "a:b:c:d"
>>> b.split(":")
```

리스트 자료형 (List)

- › 여러 데이터(숫자, 문자열 등)의 모음을 쉽게 컨트롤 가능

```
>>> data = [1, 2, 3, 4, 5]
```

- › 리스트는 대괄호([])와 쉼표(,)로 구분
- › 또한, 리스트 안에 리스트 사용 가능

```
>>> a = []  
>>> b = [1, 2, 3]  
>>> c = ['Life', 'is', 'too', 'short']  
>>> d = [1, 2, 'Life', 'is']  
>>> e = [1, 2, ['Life', 'is']]
```

리스트 인덱싱

› 리스트 역시 문자열처럼 인덱싱 적용 가능

```
>>> data = [1,2,3,4,5]  
>>> data[0]
```

› 리스트 요소를 뽑아 사칙 연산 가능

```
>>> data[0] + data[4]  
>>> data[3] / data[1]
```

› 리스트 또한 거꾸로 요소를 인덱싱 가능

```
>>> data[-1]  
>>> data[-2] / data[1]
```

리스트 인덱싱

› 이중 리스트 인덱싱

```
>>> data = [1, 2, 3, ['a', 'b', 'c']]
>>> data[0]
>>> data[-1]
>>> data[-1][0]
>>> data[-1][-1]
```

› 삼중 리스트 인덱싱

```
>>> data = [1, 2, ['a', 'b', ['Life', 'is']]]
>>> data[2]
>>> data[2][2]
>>> data[-1][-1]
>>> data[-1][-1][0]
```

리스트 슬라이싱

› 슬라이싱 사용법은 문자열과 동일

```
>>> data = [1, 2, 3, 4, 5]
```

```
>>> data[0:2]
```

```
>>> a = "12345"
```

```
>>> a[0:2]
```

```
>>> b = data[:2]
```

```
>>> c = data[2:]
```

```
>>> a = [1, 2, 3, ['a', 'b', 'c'], 4, 5]
```

```
>>> a[2:5]
```

```
>>> a[3][:2]
```

리스트 연산

› 리스트 더하기

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
```

› 리스트 반복

```
>>> a = [1, 2, 3]
>>> a * 3
```

› 리스트 길이 구하기

```
>>> a = [1, 2, 3]
>>> len(a)
```

리스트 수정과 삭제

› 리스트 요소 수정

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

› 리스트 요소 삭제

```
>>> a = [1, 2, 3]
>>> del a[1]
>>> b = [1, 2, 3, 4, 5]
>>> del b[2:]
>>> b
```

리스트 관련 함수

› 리스트 요소 추가 (append)

```
>>> a = [1, 2, 3]
>>> a.append(4)
>>> a
>>> a.append([5,6])
>>> a
```

› 리스트 요소 정렬 (sort)

```
>>> a = [1, 4, 3, 2]
>>> a.sort()
>>> a
>>> b = ['a', 'c', 'b']
>>> b.sort()
>>> b
```


리스트 관련 함수

› 리스트 뒤집기 (reverse)

```
>>> a = [1, 2, 3]
>>> a.reverse()
>>> a
```

› 리스트 위치 반환 (index)

```
>>> a = ['a', 'e', 'b', 'd', 'c']
>>> a.index('b')
>>> a.index('a')
# Error
>>> a.index('f')
```

리스트 관련 함수

› 리스트 요소 삽입 (insert)

```
>>> a = [1, 2, 3]
>>> a.insert(3, 4)
>>> a
```

› 리스트 요소 제거 (remove)

– 앞 요소부터 순서대로 제거

```
>>> a = [1, 2, 3, 1, 2, 3]
>>> a.remove(3)
>>> a
>>> a.remove(3)
>>> a
```

리스트 관련 함수

› 리스트 요소 끄집어내기 (pop)

– 인덱스 번호를 지정하지 않을 때는 맨 마지막 요소를 pop

```
>>> a = [1, 2, 3]
>>> b = a.pop()
>>> a
>>> b
```

– 지정된 인덱스 번호 요소를 pop

```
>>> a = [1, 2, 3]
>>> a.pop(1)
>>> a
```

리스트 관련 함수

- › 리스트에 지정한 요소 x의 개수 세기 (count)

```
>>> a = [1, 2, 3, 1, 1]
>>> a.count(1)
```

- › 리스트 확장 (extend)

– 리스트 연산에서 '+' 연산과 같음

```
>>> a = [1, 2, 3]
>>> a.extend([4.5])
>>> a
>>> b = [6, 7]
>>> a.extend(b)
>>> a
```

튜플 자료형 (Tuple)

- › 튜플은 () 로 사용
- › 튜플은 요소를 생성, 삭제, 수정이 불가능

```
>>> t1 = (1,)
>>> t2 = (1, 2, 3)
>>> t3 = (1, 2, 3, ('a', 'b'))
```

- › 단 1개의 요소만을 가질려면 반드시 콤마(,)를 붙여야 됨
- › 튜플 안에 리스트를 넣으면 그 리스트는 수정 가능
- › 튜플은 값이 변하지 않기를 바랄 때 사용

튜플 자료형 (Tuple)

› 튜플 요솟값 삭제시, 에러 발생

```
>>> t1 = (1, 2, 3, ('a', 'b'))  
# Error  
>>> del t1[0]
```

› 튜플 요솟값 변경시, 에러 발생

```
>>> t1 = (1, 2, 3, ('a', 'b'))  
# Error  
>>> t1[0] = 'c'
```

› 이외의 튜플 인덱싱, 슬라이싱, 연산은 리스트와 동일

딕셔너리 자료형 (Dictionary)

- › Key, Value 쌍으로 구성
- › {} 기호를 사용
- › 사전처럼 Key 값을 이용하여 원하는 Value를 찾는 형태
- › 리스트와 튜플처럼 순차적으로 요솟값을 저장 하지 않음

```
# {Key1:Value1, Key2:Value2, Key3:Value3, ...}
```

```
>>> dic = {'name':'pey', 'phone':'0119993323', 'birth': '1118'}
```

Key	Value
name	pey
phone	0119993323
birth	1118

딕셔너리 자료형 (Dictionary)

› 딕셔너리 요소 추가

```
>>> a = {1: 'a'}  
>>> a[2] = 'b'  
>>> a  
>>> a['name'] = 'pey'  
>>> a  
>>> a[3] = [1, 2, 3]  
>>> a
```

› 딕셔너리 요소 삭제

```
>>> del a[1]  
>>> a  
>>> del a['name']  
>>> a
```


딕셔너리 자료형 (Dictionary)

› 딕셔너리 Key를 사용하여 Value 얻기

```
>>> a = {1: 'a', 2: 'b', 'name': 3, 'list':[1,2,3,4]}  
>>> a[1]  
>>> a['name']  
>>> b = a['list']  
>>> b  
>>> b.append(5)  
>>> a
```

› 딕셔너리는 같은 Key 삽입 금지

› 리스트로 딕셔너리의 Key 사용 불가

› 튜플은 딕셔너리의 Key로 사용 가능

딕셔너리 관련 함수

› Key 리스트 만들기 (keys)

```
>>> a = {1: 'a', 2: 'b', 'name': 3, 'list': [1,2,3,4]}
```

```
>>> a.keys()
```

› Value 리스트 만들기 (values)

```
>>> a.values()
```

› Key, Value 쌍 얻기 (items)

```
>>> a.items()
```

› Key, Value 쌍 모두 지우기 (clear)

```
>>> a.clear()
```

딕셔너리 관련 함수

› Key로 Value 얻기 (get)

– 없는 key 값을 요청 할 경우 None 값 반환

```
>>> a = {1: 'a', 2: 'b', 'name': 3, 'list':[1,2,3,4]}  
>>> a.get(1)  
>>> a.get('name')  
>>> a.get(10)
```

– 만약 key 값이 없을 경우 반환 될 디폴트 값 지정

```
>>> a.get(3, 'NO')
```

딕셔너리 관련 함수

› 해당 Key가 딕셔너리 안에 존재하는지 조사 (in)

```
>>> a = {1: 'a', 2: 'b', 'name': 3, 'list':[1,2,3,4]}  
>>> 'name' in a  
>>> '3' in a
```

› Key가 존재 한다면 True, 아니라면 False 반환

› in 은 문자열, 리스트, 튜플과 같은 자료형에서 사용 가능

```
>>> b = [1, 2, 3, 4, 5]  
>>> 3 in b  
>>> 'name' in b
```

집합 자료형 (set)

- › 집합에 관련된 것을 쉽게 처리하기 위해 만든 자료형
- › `set()`을 통해 만들 수 있으며, 리스트나 문자열을 입력하여 만들 수 있음
- › 중복을 허용하지 않음
- › 딕셔너리처럼 순서가 존재 하지 않음

```
>>> a = set("Hello")  
>>> a  
>>> b = set([1, 2, 3, 1])  
>>> b
```

집합 자료형 (set)

- › 인덱싱 접근이 불가하므로 리스트나 튜플로 변환 후 사용
- › 중복을 허용하지 않는 특성을 이용하여 중복 제거 필터 역할로도 사용

```
>>> a = set([1, 2, 3, 1])
```

```
>>> a
```

```
>>> b = list(a)
```

```
>>> b[0]
```

```
>>> c = tuple(a)
```

```
>>> c[2]
```

집합 자료형 (set)

› 교집합

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

› 합집합

```
>>> a | b
```

› 차집합

```
>>> a - b  
>>> b - a
```

집합 관련 함수

› 값 1개 추가 (add)

```
>>> a = set([1, 2, 3])  
>>> a.add(4)  
>>> a
```

› 값 여러 개 추가 (update)

```
>>> a.update([4, 5, 6, 7])  
>>> a
```

› 특정 값 제거 (remove)

```
>>> a.remove(4)  
>>> a
```


불 자료형 (bool)

- › 참(True)와 거짓(False)을 나타내는 자료형
- › 오직 2가지 값만을 가질 수 있으며,
첫 문자는 반드시 대문자를 사용

```
>>> a = True
>>> b = False
>>> type(a)
>>> a
>>> b
```

- › 불 자료형은 조건문의 반환 값으로 사용

```
>>> 1 == 1
>>> 2 > 1
>>> 2 < 1
```

자료형의 참과 거짓

- 각 자료형의 참과 거짓은 매우 중요한 특징이며, 자주 사용 됨

```
>>> a = [1, 2, ,3]
>>> while a:
...     print(a.pop())
>>> if 0:
...     print("False")
... else:
...     print("True")
```

- bool 내장 함수를 이용하여 참과 거짓 식별 가능

```
>>> bool("123")
>>> bool([])
```

값	참 or 거짓
"123"	True
""	False
[1, 2, 3]	True
[]	False
()	False
{}	False
1	True
0	False
None	False

파이썬 변수

› 파이썬에서 변수는 =(assignment) 기호를 사용 하여 선언

- 변수 이름 = 변수에 저장할 객체

› 파이썬에서 사용하는 변수란 객체를 뜻함

› 리스트, 튜플, 딕셔너리 등 자료형 또한 하나의 객체

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

```
>>> id(a)
```

› 선언된 변수에는 각 객체의 주소 값을 저장

› 변수에 변수 값을 대입하면 주소 값이 복사

```
>>> b = a
```

```
>>> id(a)
```

```
>>> id(b)
```

파이썬 변수

- 주소가 복사된 변수는 같은 객체를 가리키므로 데이터 수정 시 주의를 요함

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

```
>>> b = a
```

"is" 는 참/거짓을 파악 할 수 있는 파이썬 명령어

```
>>> a is b
```

```
>>> b.append(5)
```

```
>>> b
```

```
>>> a
```

파이썬 변수

- 주소 복사가 아닌 값의 복사를 원한다면
`copy` 모듈 또는 `[:]` 사용

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

```
>>> b = a[:]
```

```
>>> a is b
```

```
>>> b.append(5)
```

```
>>> b
```

```
>>> a
```

```
>>> c = a.copy()
```

```
>>> a is c
```

```
>>> from copy import copy
```

```
>>> d = copy(a)
```

```
>>> a is d
```

파이썬 변수

› 파이썬 변수를 만드는 여러 가지 방법

```
>>> a, b = [1, 2]
```

```
>>> a
```

```
>>> b
```

```
>>> c = (a, b) = 3, 4
```

```
>>> c
```

```
>>> a
```

```
>>> b
```

```
>>> c = [a, b] = [[5, 6], (7, 8)]
```

```
# 매우 간단한 스왑
```

```
>>> a, b = b, a
```

```
>>> a
```

```
>>> b
```

실습 – 연습 문제

1. 주민등록번호 000101-3682746 에서 연월일(YYYYMMDD)로 출력
2. 위의 번호에서 성별을 나타내는 숫자 출력
3. `a = "a:b:c:d"` 를 `"a_b_c_d"`로 변경
4. `[1, 3, 5, 4, 2]`를 `[5, 4, 3, 2, 1]` 내림차순 정렬
5. `[1, 2, 2, 1, 3, 4, 4, 4, 5]` 중복 숫자 제거
6. 표에 제시되는 과목을 Key, 점수를 Value로 하는 딕셔너리를 작성하고 평균 점수를 구하기

subject	score
English	75
math	95
science	85

파이썬 제어문

IF 문

- › 조건에 맞는 상황을 수행 할 때 사용
- › "돈이 있으면 택시를 타고, 돈이 없으면 걸어 간다"

if 조건문:

수행할 문장

...

else:

수행할 문장

...

```
>>> money = True
```

```
>>> if money:
```

```
...     print("택시를 타고 가라")
```

```
... else:
```

```
...     print("걸어 가라")
```

IF 문

- › 파이썬은 들여쓰기(indentation)을 매우 조심해야 됨
- › 들여쓰기는 tab과 space와 다르게 적용

```
>>> money = True
>>> if money:
...     print("택시를 타고 가라")
... else:
...     print("걸어 가라")
# 반드시 예러
>>> if money:
...     print("택시를")
...     print("타고")
...     print("가라")
```

비교연산자

- › 조건문에는 참 / 거짓으로 판단
- › 비교연산자를 사용해서 조건문 사용 가능

```
>>> x = 3
>>> y = 2
>>> x > y
>>> y < x
>>> x == y
>>> x is not y
>>> money = 2000
>>> if money >= 3000:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
```

비교연산자	설명
$x < y$	x가 y보다 작다
$x > y$	x가 y보다 크다
$x == y$	x와 y가 같다
$x != y$	x와 y가 같지 않다
$x >= y$	x가 y보다 크거나 같다
$x <= y$	x가 y보다 작거나 같다
$x \text{ is } y$	x와 y가 같다
$x \text{ is not } y$	x와 y가 같지 않다

논리연산자

› 비교연산자 말고도 논리연산자 존재

논리연산자	설명
x or y	x와 y 둘중에 하나만 참이어도 참
x and y	x와 y 모두 참이어야 참
not x	x가 참이면 거짓 x가 거짓이면 참

```
>>> money = 2000
>>> card = True
>>> if money >= 3000 or card:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
```

또 다른 조건 연산자 in

- 리스트, 튜플, 문자열, 딕셔너리의 비교할 값이 존재 하는지 확인 하는 연산자

in	설명
x in 리스트	x가 리스트 안에 존재 하면 참
x in 튜플	x가 튜플 안에 존재 하면 참
x in 문자열	x가 문자열 안에 존재 하면 참
x in 딕셔너리	x가 딕셔너리의 keys에 존재 하면 참

- not in을 사용하면 x 값이 존재 하지 않을 때 참을 반환

```
>>> 1 in [1, 2, 3]
>>> 1 not in [1, 2, 3]
>>> a = {1: [1, 2, 3], 2: [4, 5, 6]}
>>> 1 in a
```

IF 문

- › 조건문에서 아무 일도 하지 않게 설정

```
>>> pocket = ["paper", "money", "cellphone"]
>>> if "money" in pocket:
...     pass
... else:
...     print("카드를 꺼내라")
```

- › `pass` 는 주로 조건문 안의 내용 구현을 미룰 때 자주 사용
함수에서도 내용 구현을 미룰 때 `pass` 사용

IF 문

› 다양한 조건을 판단하는 elif

```
>>> pocket = ["paper", "cellphone"]
>>> card = True
>>> if "money" in pocket:
...     print("택시를 타고가라")
... elif card:
...     print("카드를 꺼내라")
... else:
...     print("걸어가라")
```

IF 문

› if와 else가 수행할 문장이 단 한줄이면 if문 간략화 가능

```
>>> pocket = ["paper", "cellphone"]  
>>> if "money" in pocket:  
...     pass  
... else:  
...     print("걸어가라")
```



```
>>> pocket = ["paper", "cellphone"]  
>>> if "money" in pocket: pass  
... else: print("걸어가라")
```


IF 문 조건부 표현식

- › 조건문이 참인 경우 if 조건문 else 조건문이 거짓인 경우
- › 가독성에 유리하며 한 줄로 작성 할 수 있어 활용성이 좋음

```
>>> if score >= 60:  
...     message = "success"  
... else:  
...     message = "failure"
```



```
>>> message = "success" if score >= 60 else "failure"
```

while 문

- › while 문은 반복해서 문장을 수행해야 할 경우 사용
- › while 문은 조건문이 참인 동안에 반복해서 수행
- › 보통 항상 반복되어야 하는 작업 (예: 쓰레드)에서 사용

```
>>> treeHit = 0
>>> while treeHit < 10:
...     treeHit = treeHit + 1
...     print(f"나무를 {treeHit}번 찍었습니다.")
...     if treeHit == 10:
...         print("나무가 넘어갑니다.")
```

while 문

› 여러 가지 선택지 중 하나를 선택해서 입력 받는 예제

```
>>> while True:
...     number = int(input())
...     if number == 4:
...         break
...     else:
...         print("4번으로 while문 종료 가능.")
```

› break를 이용하면 반복문을 강제로 빠져 나옴

› while에서 True를 쓰면 무한 루프가 발생하여 작성된 문장들이 무한하게 수행

for 문

- › 반복문 중에 하나인 for 문은 직관적인 특징을 가지고 있음

```
>>> for 변수 in 리스트(또는 튜플 등):  
...     수행할 문장
```

- › 리스트, 튜플, 문자열 등의 자료형에서 첫 번째 요소부터 마지막 요소까지 차례로 변수대 대입되어 수행

```
>>> test = [1, 2, ,3]  
>>> for i in test:  
...     print(i)
```

다양한 for 문의 사용

- › 반복해서 읽을 자료형의 요소가 여러 개일 때, 변수를 여러 개로 받을 수 있음

```
>>> a = [(1, 2), (3, 4), (5, 6)]  
>>> for first, last in a:  
...     print(first, last, first + last)
```

- › 요소값의 인덱스를 알기 위해 enumerate() 내장 함수 사용

```
>>> for i, (first, last) in enumerate(a):  
...     print(i, first + last)
```

for 문의 응용

- › "총 5명의 학생이 시험을 보았는데 점수가 60점 이상 합격 그렇지 않다면 불합격이다."

```
>>> marks = [90, 25, 67, 45, 80]
>>> for i, mark in enumerate(marks):
...     if mark >= 60:
...         print(f"{i+1}번 학생은 합격입니다.")
...     else:
...         print(f"{i+1}번 학생은 불합격입니다.")
```

for 문의 응용

- › `continue` 문을 사용하면 특정 상황에 더 이상 문장이 수행하지 않고 처음으로 돌아감

```
>>> marks = [90, 25, 67, 45, 80]
>>> for i, mark in enumerate(marks):
...     if mark < 60:
...         continue
...     print(f"{i+1}번 학생은 합격입니다.")
```

- › 원하는 만큼 `for`문을 진행하고 싶을 때 `range` 함수를 사용

```
>>> for i in range(10):
...     print(f"{i+1}번 진행")
```

for 문의 응용

- › range 함수는 공통점이 있는 여러 리스트들을 동시에 참조하여 사용할 때도 좋음

```
>>> math_list = [90, 25, 67, 45, 80]
>>> english_list = [80, 85, 61, 99, 25]
>>> for i in range(len(math_list)):
...     average = (math_list[i] + english_list[i]) // 2
...     if average < 60:
...         continue
...     print(f'{i+1}번 학생은 합격입니다.')
```


리스트 내포(List comprehension)

- › 리스트 내포는 리스트 안에 for문을 포함하여 좀 더 편리하고 직관적인 문장 작성

```
>>> a = [1, 2, 3, 4]
>>> result = []
>>> for num in a:
...     result.append(num*3)
```



```
>>> a = [1, 2, 3, 4]
>>> result = [num * 3 for num in a]
>>> result
```

리스트 내포(List comprehension)

› for문 뿐만 아니라 if 조건도 사용 가능

```
>>> a = [1, 2, 3, 4]
>>> result = [num * 3 for num in a if num % 2 == 0]
>>> result
```

› 다중 for 문 또한 가능

```
>>> result = [x * y for x in range(2, 10)
...           for y in range(1, 10) ]
>>> result
```

실습 – 연습 문제

1. while 문을 사용하여 1 ~ 1000 까지의 자연수 중 3의 배수의 합을 구하기
2. 아래와 같이 print 되도록 반복문 사용

```
*  
***  
*****  
*****
```

3. A 학급의 평균 점수 구하기

```
marks = [70, 60, 55, 75, 95, 90, 80, 80, 85, 100]
```

실습 – 연습 문제

4. 아래의 코드를 리스트 내포(List comprehension)을 사용하여 표현

```
>>> numbers = [1, 2, 3, 4, 5]
>>> result = []
>>> for n in numbers:
...     if n % 2 == 1:
...         result.append(n*2)
```

파이썬 클래스와 함수

함수 란?

- › 입력값을 가지고 어떤 일을 수행한 다음 그 결과물을 내어 놓는 것을 함수라고 함
- › 프로그래밍에서 함수는 반복적으로 사용되는 가치 있는 부분을 한 덩치로 묶어 작성

```
def 함수명(매개변수):
```

```
    <수행할 문장1>
```

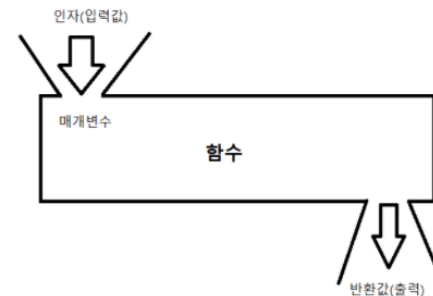
```
    <수행할 문장2>
```

```
    ...
```



```
>>> def add(a, b):
```

```
...     return a+b
```



매개변수와 인수

- › 매개변수(parameter)는 함수에 입력으로 전달된 값을 받는 변수를 의미
- › 인수(arguments)는 함수를 호출 할 때 전달하는 입력값을 의미

```
# a, b는 매개변수
```

```
>>> def add(a, b):
```

```
...     return a+b
```

```
# 3, 4는 인수
```

```
>>> temp = add(3, 4)
```

```
>>> print(temp)
```

함수의 형태

› 다양한 함수의 형태를 가짐

일반적 함수

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

입력값이 없는 함수

```
>>> def say():  
...     return "Hi"
```

결괏값이 없는 함수

```
>>> def add(a, b):  
...     print("%d, %d의 합은 %d입니다." % (a, b, a+b))
```

둘 다 없는 함수

```
>>> def say():  
...     print("Hi")
```


함수의 형태

› 입력값이 유동적으로 바뀔 때 사용 되는 방법

```
>>> def add_many(*args):  
...     result = 0  
...     for i in args:  
...         result = result + i  
...     return result
```

› *매개변수는 여러 개의 입력값을 전부 모아 튜플로 만듦

```
>>> r = add_many(1, 2, 3)  
>>> print(r)  
>>> r2 = add_many(1, 2, 3, 4, 5, 6, 7)  
>>> print(r2)
```

함수의 형태

- › 키워드 파라미터 kwargs 를 통해서도 다양한 입력값 사용 가능

```
>>> def print_kwargs(**kwargs):  
...     print(kwargs)
```

- › **매개변수는 여러 개의 입력값을 딕셔너리 형태로 저장
- › 반드시 입력값에 key, value 값으로 줘야 함

```
>>> print_kwargs(a=1)  
>>> print_kwargs(name='god', age=10)
```

함수의 형태

› 파이썬의 함수는 여러 개의 값을 반환 가능

```
>>> def add_and_mul(a, b):  
...     return a+b, a*b  
>>> result = add_and_mul(3, 4)  
>>> result  
>>> r1, r2 = add_and_mul(3, 4)  
>>> r1  
>>> r2
```

› 여러 개의 값이 반환 될 때에는 튜플 형태로 묶여서 나옴

함수의 형태

› return은 함수를 빠져나가고 싶을 때도 사용 가능

```
>>> def say_nick(nick):  
...     if nick == "바보":  
...         return  
...     print("나의 별명은 %s 입니다." % nick)  
...
```



```
>>> say_nick("야호")  
>>> say_nick("바보")
```

함수의 형태

- › 매개변수에 초깃값 미리 설정
- › 초깃값이 있는 매개 변수는 반드시 일반 매개 변수보다 뒤에 존재

```
>>> def say_myself(name, old, man=True):  
...     print("나의 이름은 %s 입니다." % name)  
...     print("나이는 %d살입니다." % old)  
...     if man:  
...         print("남자입니다.")  
...     else:  
...         print("여자입니다.")
```

```
>>> say_myself("박응용", 27)  
>>> say_myself("박응용", 27, True)
```

변수의 범위

- › 함수 안에서 선언된 변수는 지역 변수이므로 밖에서 사용 불가능
- › 또한, 함수 밖에 같은 이름의 변수가 있더라도 함수 안에 있는 변수가 사용

```
>>> a = 1
>>> def vartest(a):
...     a = a + 1
>>> vartest(a)
>>> print(a)
```

변수의 범위

- › **global** 명령어를 사용하여 변수를 공유해서 사용 가능

```
>>> a = 1
>>> def vartest():
...     global a
...     a = a + 1
>>> vartest()
>>> print(a)
```

- › 리스트, 튜플, 딕셔너리를 복사하지 않고 매개변수로 넣으면 공유해서 사용

```
>>> a = [1, 2, 3]
>>> def add_value(tmp_list):
...     tmp_list.append(4)
>>> add_value(a)
>>> print(a)
```

lambda 함수

- › 함수를 생성할 때 사용하는 예약어로 def와 동일한 역할
- › 보통 함수를 한줄로 간결하게 만들 때 사용
- › 쓰고 버리는 일시적인 함수이며 이름이 없는 함수, 익명함수라고도 불림

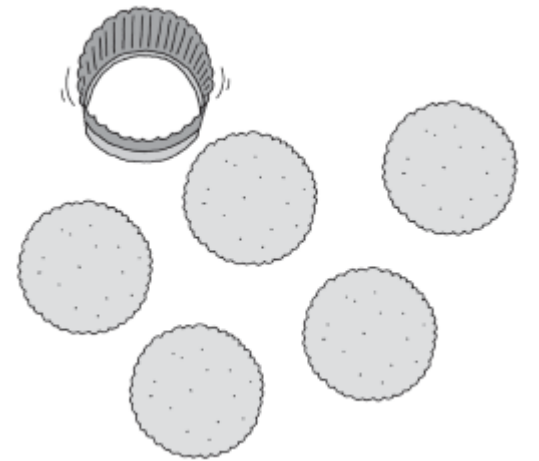
lambda 매개변수1, 매개변수2, ... : 매개변수를 이용한 표현식

```
>>> add = lambda a, b: a+b  
>>> result = add(3, 4)  
>>> print(result)
```


클래스와 객체

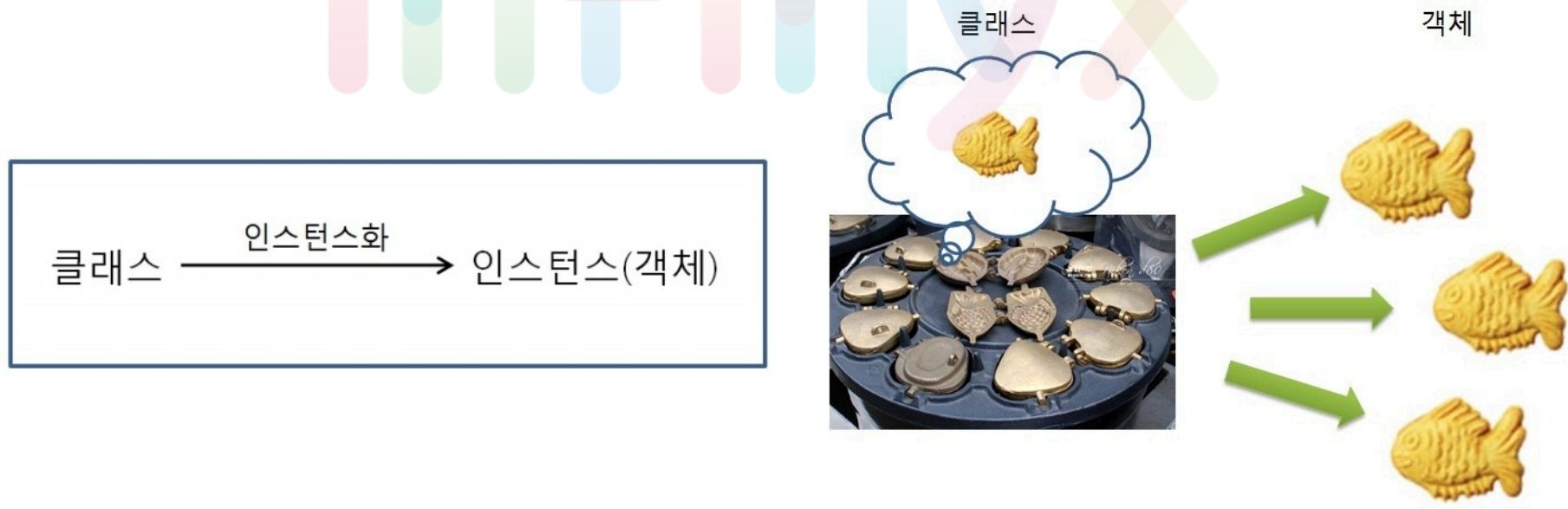
- › 클래스(class)란 똑같은 무엇 인가를 계속해서 만들어 낼 수 있는 설계 도면(과자 틀)
- › 객체(object)란 클래스로 만든 피조물을 뜻 함 (과자 틀을 사용해 만든 과자)
- › 클래스로 만든 객체는 객체마다 고유한 성격을 가짐

```
>>> class Cookie:  
...     pass  
  
>>> a = Cookie()  
>>> b = Cookie()  
>>> a is b
```



클래스와 객체

- › 클래스로부터 객체를 만드는 과정을 ‘인스턴스화’라 함
- › 어떤 클래스로 부터 만들어진 객체를 ‘그 클래스의 인스턴스’라 함
- › 인스턴스 = 객체



사칙연산 클래스 만들기

- › 데이터를 set하고 add, sub, mul, div 함수가 있는 사칙연산 클래스 생성

```
>>> a = FourCal()
>>> a.setdata(4,2)
>>> a.add()
>>> a.sub()
>>> a.mul()
>>> a.div()
```

사칙연산 클래스 만들기

- › 클래스 안에 구현된 함수는 다른 말로 메서드(Method)
- › 메서드는 첫 번째 매개변수 이름을 관례적으로 `self`를 사용
- › `self`는 말 그대로 자기 자신인 클래스 객체를 뜻함[®]

```
>>> class FourCal:  
...     def setdata(self, first, second):  
...         self.first = first  
...         self.second = second  
  
>>> a = FourCal()  
>>> a.setdata(4, 2)
```

```
>>> a = FourCal()  
>>> FourCal.setdata(a, 4, 2)
```

사칙연산 클래스 만들기

- › self.변수명에 값을 넣게 되면 선언된 객체에서 호출하여 사용 가능

```
>>> a = FourCal()
>>> a.setdata(4, 2)
>>> a.first
>>> a.second
```

- › 클래스 생성 단계에서 선언 및 초기화를 하지 않았거나 메서드를 통해 해당 변수를 선언하지 않았다면 호출 불가능

```
>>> b = FourCal()
>>> b.first
```

사칙연산 클래스 만들기

- › 클래스에서 선언된 변수는 `self` 를 통해 호출하여 사칙연산 가능

```
>>> class FourCal:
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...     def add(self):
...         result = self.first + self.second
...         return result
>>> a = FourCal()
>>> a.setdata(4, 2)
>>> a.add()
```

클래스 생성자 (Constructor)

- › 생성자란 객체가 생성될 때 자동으로 호출되는 메서드를 의미
- › 메서드 이름으로 `__init__`를 만들고 사용

```
>>> class FourCal:
...     def __init__(self, first, second):
...         self.first = first
...         self.second = second
# 생성자에서 전달 받아야 할 값이 있다면 반드시 생성 할때 값을 전달
>>> a = FourCal()

>>> a = FourCal(4, 2)
```

클래스 상속 (Inheritance)

› 상속을 통해 a의 b제곱(a^b) 계산 메서드를 추가

```
>>> class MoreFourCal(FourCal):  
...     def pow(self):  
...         result = self.first ** self.second  
...         return result  
>>> a = MoreFourCal(4, 2)  
>>> a.pow()  
  
>>> b = FourCal(4, 2)  
# FourCal는 pow 메서드가 없기 때문에 사용 불가능  
>>> b.pow()
```

부모

자식1

자식2

메서드 오버라이딩(Overriding)

- 자식 클래스에서 부모 클래스에 존재하는 동일한 이름의 메서드를 다시 재정의 하는 것을 오버라이딩이라 함

```
>>> a = FourCal(4, 0)
# 0으로 나누려고 하기 때문에 에러 발생
# 오버라이딩으로 에러 예외처리 필요
>>> a.div()
```

```
>>> class SafeFourCal(FourCal):
...     def div(self):
...         if self.second == 0:
...             return 0
...         else:
...             return self.first / self.second
```

클래스 변수

- 객체 변수는 다른 객체들에 영향 받지 않고 독립적으로 값을 유지하지만 클래스 변수는 공유 되어 영향을 받음

```
>>> class Family:  
...     lastname = "김"  
>>> Family.lastname
```

```
>>> a = Family()  
>>> b = Family()  
>>> a.lastname  
>>> b.lastname  
>>> Family.lastname = "박"  
>>> a.lastname  
>>> b.lastname
```

클래스 필드 (Field)

- › 클래스에 소속된 변수들을 멤버 변수 또는 필드라고 부름
- › 필드는 해당 클래스 또는 객체 내부에서 의미가 있음(in namespace)
- › 필드에는 객체 변수와 클래스 변수가 존재한다.
- › 필드를 소유하고 있는 대상이 클래스냐 객체냐에 따라 객체 변수인지 클래스 변수인지 구분됨

```
>>> class FourCal:
    name = "계산기" # 클래스변수
...     def __init__(self, first, second):
...         self.first = first # 객체변수
...         self.second = second
```

모듈

- › 함수나 변수 또는 클래스를 모아 놓은 파일
- › 모듈은 다른 파이썬 프로그램에서 불러와 사용 할 수 있게끔 만든 파이썬 파일

```
# mod1.py  
def add(a, b):  
    return a + b  
  
def sub(a, b):  
    return a - b
```

```
>>> import mod1  
>>> mod1.add(3, 4)  
>>> mod1.sub(4, 2)
```

모듈

- › 모듈은 기본적으로 "import 모듈이름"으로 사용되나, 모듈 이름 없이 함수 이름만 쓰고 싶은 경우 "from 모듈이름 import 모듈함수" 로 사용 가능

```
>>> from mod1 import add  
>>> add(3, 4)
```

- › 또한 사용하고 싶은 함수가 더 있다면 import에 모듈 함수를 콤마로 구분하여 사용

```
>>> from mod1 import add, sub  
>>> add(3, 4)  
>>> sub(4, 2)
```

모듈

- › 모듈에 속해 있는 모든 함수를 불러오고 싶을 때는 "*"를 사용
- › 단, "*"은 모든 함수를 불러오기 때문에 다른 모듈에서 같은 함수가 존재 한다면 뒤에 선언 된 모듈의 함수가 사용

```
>>> from mod1 import *  
>>> add(3, 4)  
>>> sub(4, 2)
```

모듈

› 클래스와 변수가 포함 된 모듈 작성

```
# mod2.py
```

```
PI = 3.141592
```

```
class Math:
```

```
    def solv(self, r):
```

```
        return PI * (r ** 2)
```

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

```
    return a+b
```

```
>>> import mod2
```

```
>>> mod2.PI
```

모듈

› 보통 모듈로 가지고 오는 변수들은 대부분 상수처럼 사용

```
>>> import mod2  
>>> a = mod2.Math()  
>>> a.solve(2)  
  
>>> mod2.add(mod2.PI, 4.4)
```

› from import를 이용하여 원하는 클래스 및 변수 사용 가능

```
>>> from mod2 import PI, add  
>>> add(PI, 4.4)
```


모듈

- › 다른 위치에 존재하는 모듈을 사용하기 위해서는
sys 모듈 사용

```
>>> import sys
>>> sys.path
>>> sys.path.append("추가 할 모듈 경로")
>>> sys.path
```

- › **PYTHONPATH** 환경 변수를 사용하여 모듈을 불러오는 방법도 존재

```
terminal> set PYTHONPATH="경로"
terminal> python
>>> import 모듈이름
```

패키지 (Packages)

› 패키지는 도트(.)를 사용하여 파이썬 모듈을 계층적으로 관리

```
mypackages/  
  __init__.py  
  calc/  
    __init__.py  
    add.py  
    div.py  
    mul.py  
    sub.py  
  math/  
    __init__.py  
    factorial.py
```

패키지 (Packages)

- › 패키지는 `__init__.py` 에서 정의한 것만 참조하여 사용
- › `__init__.py` 파일은 해당 디렉터리가 패키지의 일부임을 알려주는 역할
- › 만약 `__init__.py` 파일이 없다면 패키지로 인식되지 않음

```
# mypackages/calc/__init__.py
# sub 모듈을 추가 안하면 calc 위치에서 사용 불가
__all__ = ['div', 'mul', 'add']
```

- › `__init__.py` 에서 직접 같은 폴더 아래의 파이썬 모듈을 `import` 해도 되지만 `__all__` 를 이용해서도 추가 가능

패키지 (Packages)

```
from mypackages.calc.div import div
from mypackages.calc.mul import my_mul
from .add import *
# 주식 처리 함으로서 sub쪽은 참조 불가능
# from .sub import my_sub
```

- › sub를 추가하지 않았기 때문에 sub 사용 불가능
- › `__init__.py` 에서 모듈안에 함수를 import 하여 바로 접근 가능

```
>>> from mypackages import calc
>>> calc.add.my_add(5, 4)
>>> calc.my_add(3, 8)
# __init__.py 에 sub에 대한 선언이 되어 있지 않아 에러 발생
>>> calc.sub.my_sub(5, 4)
```

패키지 (Packages)

› 패키지 안에서는 모듈을 relative하게 import 가능

```
# mypackages/calc/__init__.py  
from .add import *
```

```
# mypackages/math/factorial.py  
from ..calc.mul import my_mul
```



```
>>> from mypackages import calc, math  
>>> calc.add.my_add(5, 4)  
>>> math.my_factorial(5)
```

실습 – 연습 문제

1. 주어진 자연수가 홀수인지 짝수인지 판별해 주는 함수(is_odd)를 작성
2. 입력으로 들어오는 모든 수의 평균 값을 계산해 주는 함수(is_average) 작성
(단, 입력으로 들어오는 수의 개수는 정해져 있지 않음)
3. Calculator 클래스를 상속 받아 UpgradeCalculator를 만들고 나머지 사칙 연산 메서드 추가

```
class Calculator:  
    def __init__(self):  
        self.value = 0  
    def add(self, val):  
        self.value += val
```

실습 – 연습 문제

4. Calculator 클래스를 상속 받아 MaxLimitCalculator를 만들고 value가 100 이상의 값은 가질 수 없도록 제한하는 add 메서드로 변경

```
>>> cal = MaxLimitCalculator()
>>> cal.add(70) # 70 더하기
>>> cal.add(60) # 60 더하기
>>> print(cal.value) # 130이 아닌 100 출력
```

파이썬 반복 가능 객체 & 예외처리

예외 처리

- › 각 종 오류들을 방지 하기 위한 예외 처리 방법으로 `try except` 구문 사용

```
>>> 4 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```



```
>>> try:
...     a = 4 / 0
...     print(a)
... except ZeroDivisionError:
...     print("Zero Division Error!!!!")
```

```
try:
    실행 코드
except:
    예외가 발생했을 때 수행할 코드
else:
    예외가 발생하지 않았을 때 수행할 코드
finally:
    예외 발생 여부와 상관없이 항상 수행할 코드
```

예외 처리

- › 순수하게 "try, except"를 사용하면 모든 오류에 대해서 예외 처리 수행
- › "try, except 발생 오류"를 사용하면 지정 한 오류에 대해서만 예외 처리 수행
- › "try, except 발생 오류 as 오류 메시지 변수"를 사용하면 2번째와 똑같은 작업을 수행하지만 오류 내용을 확인 가능

```
>>> try:
...     a = 4 / 0
... except ZeroDivisionError as e:
...     print(f"오류발생 >>> {e}")
```

예외 처리

- › "try, finally"는 try문 수행 도중 오류 발생 여부에 상관없이 항상 수행하며, 보통 사용한 리소스를 close 할 때 사용

```
>>> f = open("test.txt", "w")
>>> try:
...     # 무언가의 작업 내용들
... finally:
...     f.close()
```

- › "try, except 발생 오류 , except 발생 오류 ..."는 여러 개의 오류를 상황 별로 예외 처리 하기 위해 사용

예외 처리

› 여러 개의 오류 처리 예시

```
>>> try:
...     a = [1,2]
...     print(a[3])
...     4/0
>>> except ZeroDivisionError:
...     print("0으로 나눌 수 없습니다.")
>>> except IndexError:
...     print("인덱싱 할 수 없습니다.")
```

```
>>> try:
...     a = [1,2]
...     print(a[3])
>>> except (ZeroDivisionError, IndexError) as e:
...     print(e)
```

예외 처리

- › 일부러 오류를 발생시킬 경우 raise 명령어 사용

```
class Bird:  
    def fly(self):  
        raise NotImplementedError
```

- › 부모 클래스에서 구현하지 않고 자식 클래스에게 반드시 해당 함수를 구현하게끔 강제할 때 사용

```
class Eagle(Bird):  
    def fly(self):  
        print("very fast")  
  
eagle = Eagle()  
eagle.fly()
```

예외 처리

- › Exception 클래스를 상속 받아 직접 예외 클래스를 생성 가능

```
class MyError(Exception):  
    def __str__(self):  
        return "허용되지 않는 별명입니다."  
  
    def say_nick(nick):  
        if nick == '바보':  
            raise MyError()  
        print(nick)  
  
>>> try:  
...     say_nick("천사")  
...     say_nick("바보")  
... except MyError as e:  
...     print(e)
```

컨테이너 (Container)

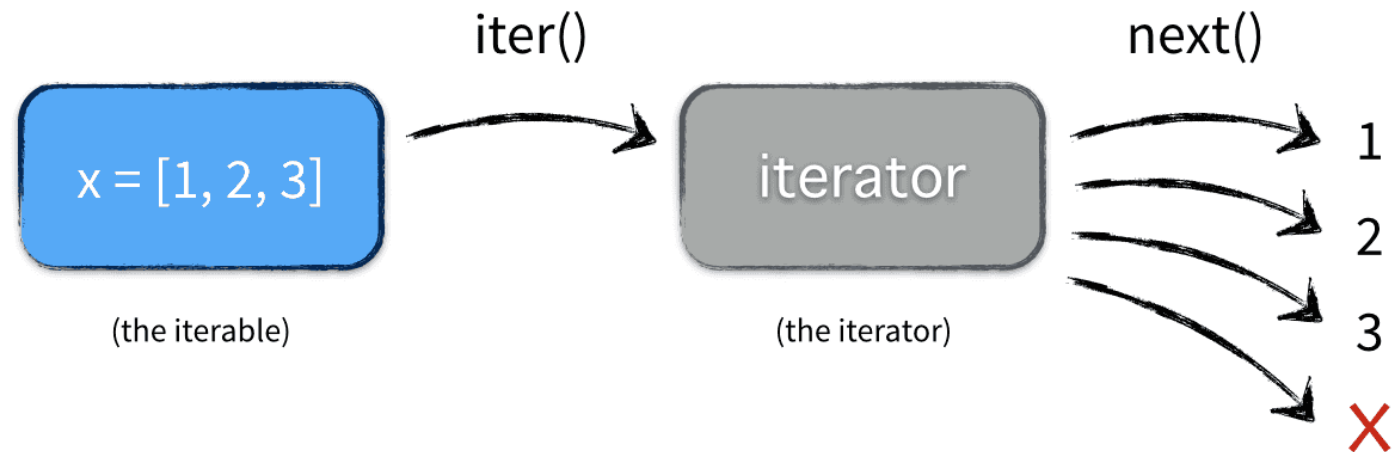
- › 컨테이너는 요소들을 가지고 있는 데이터 구조(하나 이상의 숫자나, 문자열을 가지고 있는 묶음)
- › 메모리에 상주하는 데이터 구조로 보통 모든 요소값들을 메모리에 가지고 있음
- › 파이썬에서 자주 사용되는 컨테이너 목록

- list, deque, ...
- set, frozenset, ...
- dict, defaultdict, OrderedDict, ...
- tuple, namedtuple, ...
- str

- › 어떤 객체가 특정한 요소를 포함하고 있는지 아닌지 판단 가능하다면 컨테이너로 볼 수 있음

이터레이블 (Iterable)

- › 이터레이블은 반복 가능하다는 것을 의미
- › 이터레이블은 반드시 데이터 구조일 필요는 없으며 이터레이터 (iterator)를 반환 할 수 있는 모든 객체
- › 대부분의 컨테이너는 이터레이블이며 파일, 소켓 등 또한 이터레이블 이라 볼 수 있음



이터레이터 (Iterator)

- › 이터레이터는 반복자 즉, 반복 가능한 자료형을 의미
- › 특정 자료형 혹은 함수나 메서드의 결과를 반복 가능하게 만들어줄 수 있는 자료형
- › 이터레이터 자료형을 만들기 위해서는 `__iter__()` 메서드 또는 `iter()` 함수를 이용
- › `next()` 함수를 호출 할 때 다음 값을 생성해내는 모든 객체들을 이터레이터로 볼 수 있음

```
>>> a = [1, 2, 3]
>>> b = iter(a)
>>> next(b)
>>> b.__next__()
```

제너레이터 (Generator)

- › 제너레이터는 생성자를 뜻하며 이터레이터와 같은 루프의 작용을 컨트롤하기 위해 쓰여지는 특별한 함수 또는 루틴
- › 이터레이터가 기존에 있는 요소를 하나씩 꺼내는 식으로 반환을 했다면, 제너레이터는 필요할 때마다 직접 생성하여 요소를 반환
- › `yield` 키워드를 통해 제너레이터 함수를 생성
- › 모든 제너레이터는 이터레이터라 볼 수 있음
- › 또한 CPU/메모리 효율을 높일 수 있음

제너레이터 (Generator)

- › 제너레이터는 이터레이터를 생성해주는 함수
- › 이터레이터는 클래스에 `__iter__`, `__next__`, 또는 `__getitem__` 메서드를 구현해야 하지만, 제너레이터는 함수 안에서 `yield` 키워드만 사용하면 됨
- › 제너레이터는 이터레이터보다 훨씬 간단하게 작성할 수 있음
- › 제너레이터는 발생자라고 부르기도 함

제너레이터 (Generator)

```
def number_generator():
```

```
    yield 0
```

```
    yield 1
```

```
    yield 2
```

```
for i in number_generator():
```

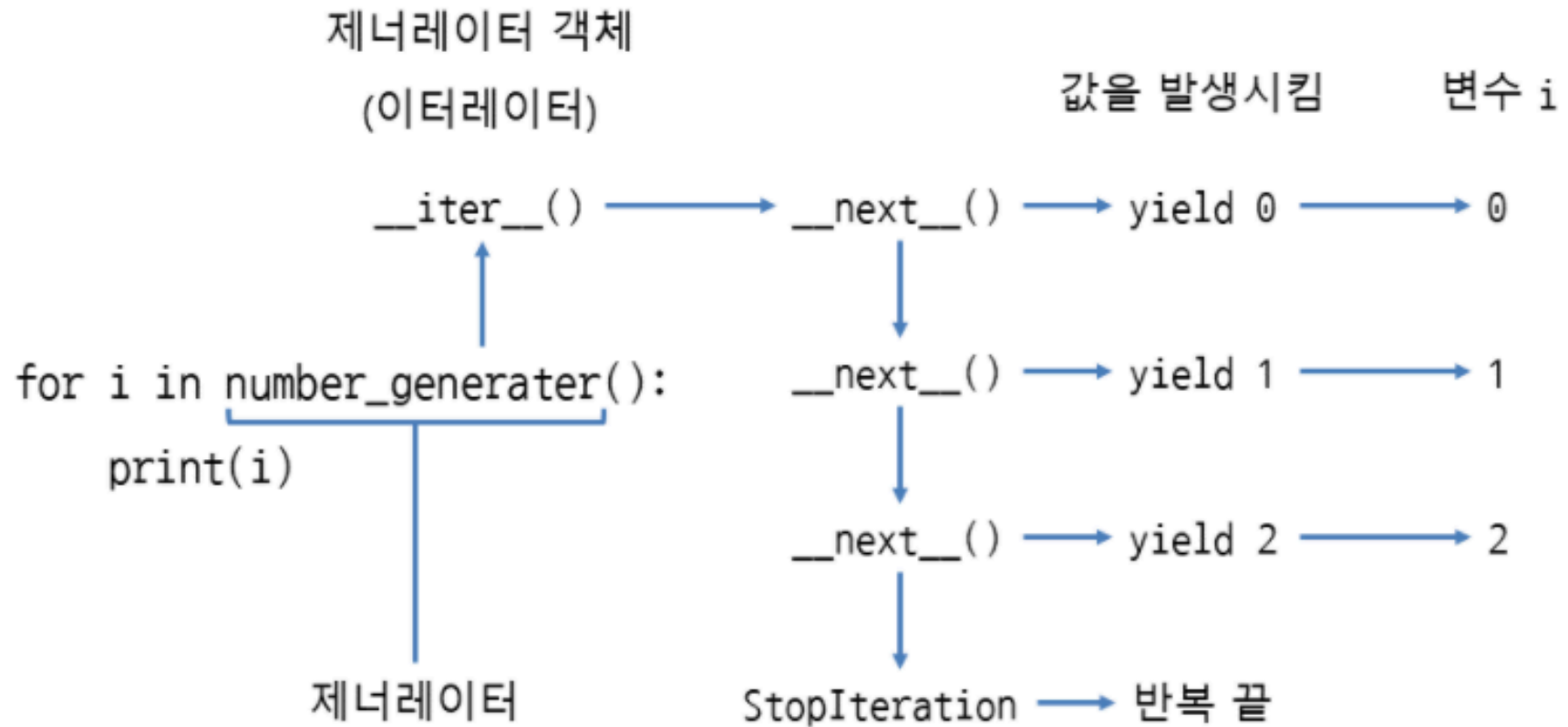
```
    print(i)
```

0

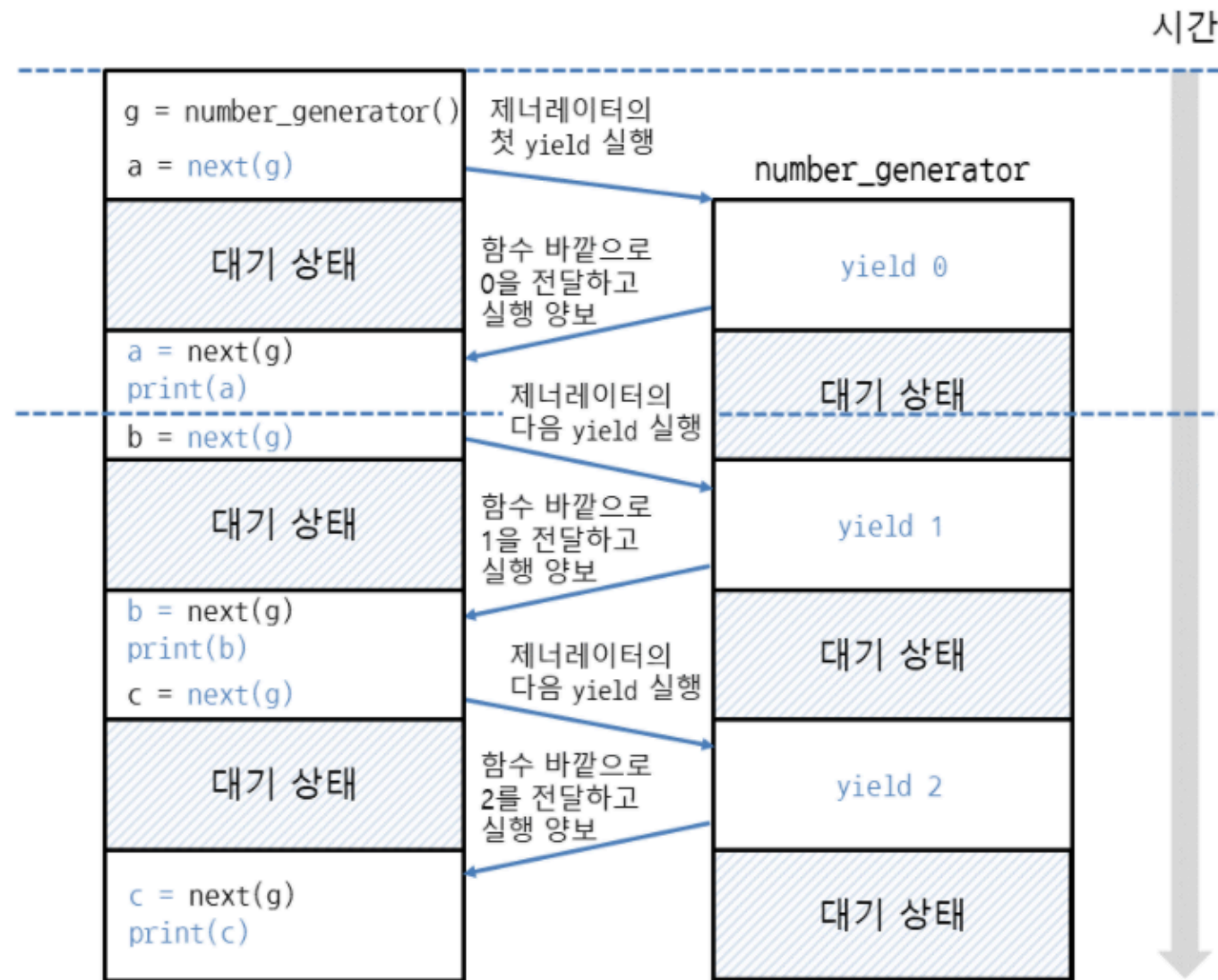
1

2

제너레이터 (Generator)



제너레이터 (Generator)



if `__name__ == "__main__"`: 의 의미

- › 실행시킨 파이썬 파일이 메인 일때 if문을 만족하여 작성된 문장들을 실행
- › 주로 프로그램의 시작이 되는 메인 파일에 사용 됨
- › 이외에도 파이썬 파일에 작성된 특정 함수 및 클래스를 메인 파이썬 파일을 통해서가 아닌 테스트 목적으로 해당 파일을 직접 실행 시켜 확인 할 때 사용

```
# mod1.py
```

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

```
    return a+b
```

```
if __name__ == "__main__":
```

```
    print(add(1, 4))
```

파이썬 내/외장 함수

내장 함수

- › 파이썬의 내장 함수는 import를 별도로 하지 않고 바로 사용 가능한 함수이며, print(), type() 등이 이에 속함
- › abs(x)는 숫자를 입력하면 절댓값을 반환

```
>>> abs(3)
>>> abs(-1.2)
```

- › all(x)는 반복 가능한 자료형(iterable) x를 입력 인수로 받아 x가 모두 참이면 True, 거짓이 하나라도 있으면 False 반환

```
>>> all([1, 2, 3])
>>> all([1, 2, 3, 0])
```

내장 함수

- › `any(x)`는 `x`를 입력 인수로 받아 하나라도 참이면 `True`, 모두 거짓일때 `False` 반환

```
>>> any([1, 2, 0])
```

```
>>> any([0, ""])
```

- › `chr(x)`는 아스키 코드 값을 입력 받아 해당하는 문자 반환
`ord(x)`는 문자를 아스키 코드 값으로 반환

```
>>> chr(97)
```

```
>>> chr(48)
```

- › `dir(x)`은 객체가 가지고 있는 변수나 함수 목록을 반환

```
>>> dir([1, 2])
```

```
>>> dir({'1':"a"})
```

내장 함수

- › `divmod(a, b)`는 2개의 숫자를 입력 받고 `a`를 `b`로 나눈 몫과 나머지를 튜플 형태로 반환

```
>>> divmod(7, 3)
```

- › `enumerate(x)`는 순서가 있는 자료형(리스트, 튜플 등)을 입력으로 받아 인덱스 값을 포함하여 반환

```
>>> for i, name in enumerate(['body', 'foo', 'bar']):  
...     print(i, name)  
...
```

내장 함수

- › `eval(x)`은 실행 가능한 문자열을 입력 받아 실행한 결과값을 돌려주는 함수

```
>>> eval('1+2')  
>>> eval("'hi' + 'a'")  
>>> eval('divmod(4, 3)')
```

- › `filter(f, x)`는 첫 번째 인수로 함수를 두 번째 인수로 함수에 차례로 들어갈 반복 가능한 자료형을 넣어 반환 값이 참인 것만 걸러 내서 반환

```
>>> def positive(x):  
...     return x > 0  
>>> list(filter(positive, [1, -3, 2, 0, -5, 6]))  
>>> list(filter(lambda x: x > 0, [1, -3, 2, 0, -5, 6]))
```

내장 함수

› `id(x)`는 객체를 입력 받아 객체의 고유 주소 값을 반환

```
>>> a = 3  
>>> id(3)  
>>> id(a)  
>>> b = a  
>>> id(b)
```

› `input()`은 사용자 입력을 받는 함수

```
>>> a = input()  
>>> a  
>>> b = input("Enter: ")
```

내장 함수

- › `int(x)`는 문자열 형태의 숫자나 소수점이 있는 숫자 등을 정수 형태로 반환

```
>>> int('3')  
>>> int(3.4)
```

- › `isinstance(x, class)`는 첫 번째 인수로 인스턴스, 두 번째 인수로 클래스를 받으며 인스턴스가 해당 클래스의 인스턴스인지 판단하여 참이면 `True`, 거짓이면 `False` 반환

```
>>> a = 3  
>>> isinstance(a, str)  
>>> isinstance(a, int)
```

내장 함수

- › `len(x)`은 입력값의 길이를 반환

```
>>> len('python')  
>>> len([3,4,5])
```

- › `list(x)`는 반복 가능한 자료형을 입력 받아 리스트로 변환시켜 반환

```
>>> list("python")  
>>> list((1,2,3))  
>>> a = [1, 2, 3]  
# 리스트 복사  
>>> b = list(a)  
>>> b is a
```

내장 함수

- › `max(x)`는 반복 가능한 자료형을 입력 받아 최댓값을
`min(x)`은 최솟값을 반환

```
>>> max([1, 2, 3])  
>>> min([1, 2, 3])  
>>> min("python")
```

- › `map(f, x)`은 첫 번째 인수로 함수를 두 번째 인수로 함수에 차례로
들어갈 반복 가능한 자료형을 넣어 각 요소를 함수가 수행한 결과를
묶어서 반환

```
>>> def two_times(x):  
...     return x * 2  
  
>>> list(map(two_times, [1, -3, 2, 0, -5, 6]))  
  
>>> list(filter(lambda x: x * 2, [1, -3, 2, 0, -5, 6]))
```


내장 함수

- › `open(filename, mode)`은 파일 이름과 읽기 방법을 입력받아 파일 객체를 반환하며, 읽기방법(mode)을 생략하면 기본값인 읽기 전용 모드(r)로 파일 객체를 만들어 반환

```
>>> fread = open("read_mode.txt")
>>> fappend = open(" read_mode.txt", 'a')

# b는 w, r, a와 함께 사용하며, rb는 바이너리 읽기 모드를 뜻함
>>> f = open("binary_file", "rb")

>>> fwrite = open("write_mode.txt", 'w')
```

mode	설명
w	쓰기 모드로 파일 열기
r	읽기 모드로 파일 열기
a	추가 모드로 파일 열기
b	바이너리 모드로 파일 열기

내장 함수

- › `pow(x, y)`는 `x`의 `y` 제곱한 결과값을 반환

```
>>> pow(2, 4)
>>> pow(3, 3)
```

- › `range(start, stop, step)`는 주로 `for`문과 함께 자주 사용되며, 입력 받은 숫자에 해당하는 범위 값을 반복 가능한 객체로 만들어 반환

```
>>> list(range(5))
>>> list(range(1, 6))
>>> list(range(1, 11, 2))
>>> for i in range(10):
...     print(f'{i}번째!!')
```

내장 함수

- › `round(x[, ndigits])`는 숫자를 입력 받아 반올림해서 반환
또한 필요하다면 소수점 n번자리까지만 반올림 가능

```
>>> round(4.6)
>>> round(4.2)
>>> round(3.141592, 2)
```

- › `sorted(x[, key, reverse])`는 입력값을 정렬한 이후 리스트로 반환하며 `key` 매개변수는 정렬의 기준을 지정 가능

```
>>> sorted([3, 1, 2])
>>> sorted(['a', 'c', 'b'])
>>> a = [['a', 3], ['b', 1], ['c', 2]]
>>> sorted(a, key=lambda x: x[1])
>>> sorted(a, key=lambda x: x[1], reverse=True)
```

내장 함수

› `str(x)`은 문자열 형태로 객체를 변환하여 반환

```
>>> str(3)
>>> class str_test:
...     def __str__(self):
...         return "str 테스트!!"
>>> str(str_test())
```

› `sum(x)`은 입력받은 리스트나 튜플의 모든 요소의 합을 반환

```
>>> sum([3, 1, 2])
>>> sum((4, 5, 6))
```

내장 함수

› **tuple(x)**은 반복 가능한 자료형을 입력받아 튜플 형태로 바꾸어 반환

```
>>> tuple("abc")  
>>> tuple([1, 2, 3])
```

› **type(x)**은 입력값의 자료형이 무엇인지 알려 주는 함수

```
>>> type("abc")  
>>> type([])  
>>> type()
```

› **zip(*x)**은 동일한 개수로 이루어진 자료형을 묶어 주는 함수

```
>>> list(zip([1, 2, 3], [4, 5, 6]))  
>>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))  
>>> list(zip("abc", "def"))
```

외장 함수(pickle)

- › pickle은 객체의 형태를 그대로 유지하면서 파일에 저장하고 불러올 수 있게 하는 모듈이며,
딥러닝에서 학습된 파라미터들을 저장할 때 pickle 사용

```
>>> import pickle
>>> f = open("test.txt", 'wb')
>>> data = {1: 'python', 2: 'you need'}
>>> pickle.dump(data, f)
>>> f.close()
```

```
>>> f = open("test.txt", 'rb')
>>> data = pickle.load(f)
>>> print(data)
>>> f.close()
```

외장 함수(os)

› os 모듈은 환경 변수, 디렉터리, 파일 등의 os 자원을 제어 할 수 있게 해주는 모듈

› os.environ은 자신의 시스템 환경 변수 값을 반환

```
>>> import os  
>>> os.environ
```

› os.getcwd()은 현재 자신의 디렉터리 경로를 반환

› os.chdir(path)은 현재 디렉터리의 경로를 변경

```
>>> os.getcwd()  
>>> os.chdir("C:/")  
>>> os.getcwd()
```

외장 함수(os)

- › `os.mkdir(path)`는 입력받은 경로에 디렉터리를 생성
단, 입력받은 경로의 상위(부모)디렉터리가 존재 하지 않는다면 오류 발생
- › `os.makedirs(path)`는 입력받은 경로에 디렉터리를 생성
`mkdir`과는 다른 점은 입력받은 경로에 존재하지 않는 디렉터리들은 전부다 생성 시켜 입력받은 경로에 디렉터리가 반드시 생성 되게 함

C드라이브 아래에 test 폴더가 없다면 오류 발생

```
>>> os.mkdir("C:/test/create")
```

test, create 폴더가 없더라도 전부다 생성

```
>>> os.makedirs("C:/test/create/good")
```


외장 함수(os)

- › `os.rmdir(path)`는 입력받은 경로의 디렉토리를 삭제
단, 디렉터리가 비어있어야 삭제가 가능

```
# 디렉터리가 비어있지 않다면 오류 발생
>>> os.rmdir("C:/test/create")
>>> os.rmdir("C:/test/create/good")
```

- › `os.walk(path)`는 입력 받은 경로로 부터 그 아래의 모든 디렉토리
및 파일을 검색

```
>>> for (path, dir, files) in os.walk("c:/"):
...     for filename in files:
...         print(f"경로: {path} / 이름: {filename}")
```

외장 함수(os)

- › os모듈 아래에 os.path 모듈 또한 자주 사용 되며, 파일 경로를 생성 및 수정하고, 파일 정보를 쉽게 다룰 수 있게 함
- › os.path.abspath(path)는 현재 경로를 prefix로 하여 입력 받은 경로를 절대경로로 바꿔서 반환

```
>>> import os  
>>> os.path.abspath("tmp")
```

- › os.path.basename(path)은 입력받은 경로의 가장 마지막 경로의 파일 및 디렉터리 이름 반환

```
>>> os.path.basename("C:/test")  
>>> os.path.basename("C:/test/python.txt")
```

외장 함수(os)

- › `os.path.dirname(path)`은 입력받은 경로의 부모 디렉터리의 경로를 반환

```
>>> os.path.dirname("C:/test")  
>>> os.path.dirname("C:/test/python.txt")
```

- › `os.path.exists(path)`는 입력받은 경로가 존재하면 `True`, 존재하지 않는 경우 `False`를 반환
단, `os`에서 읽기 권한이 없는 경우에는 `False` 반환

```
>>> os.path.exists("C:/test/python.txt")  
>>> os.path.exists("C:/test")
```

외장 함수(os)

- › `os.path.isdir(path)`는 입력받은 디렉터리 경로가 존재하면 `True`, 없으면 `False` 반환
- › `os.path.isfile(path)`는 입력받은 파일 경로가 존재하면 `True`, 없으면 `False` 반환
- › 단, 입력받은 경로가 실제로 존재하더라도
 `isdir`에서 입력받은 경로가 파일이라면 `False`를 반환하고,
 `isfile`에서 입력받은 경로가 디렉터리라면 `False`를 반환

```
>>> os.path.isdir("C:/test")  
>>> os.path.isdir("C:/test/python.txt")  
>>> os.path.isdir("C:/test")  
>>> os.path.isfile("C:/test/python.txt")
```

외장 함수(os)

- › `os.path.isabs(path)`는 경로가 절대경로일 경우 `True`, 그 외의 경우 `False` 반환
(경로 존재와 상관없이 문자열을 가지고 검사)

```
>>> os.path.isabs("C:/test/python.txt")  
>>> os.path.isabs("test")
```

- › `os.path.join(path1[, path2[, ...]])` 해당 OS 형식에 맞도록 입력 받은 경로를 연결하여 반환
(입력 중간에 절대경로가 나오면 이전 경로는 제거)

```
>>> os.path.join("C:/test", "create")  
>>> os.path.join("C:/test", "create", "python.txt")  
>>> os.path.join("C:/test", "create", "D:/python", "test.txt")
```

외장 함수(os)

- › `os.path.split(path)`는 입력 받은 경로를 마지막 파일 및 디렉터리를 분리하여 튜플로 반환

```
>>> os.path.split("C:/test/python.txt")  
>>> os.path.split("C:/test")
```

- › `os.path.splitdrive(path)`는 드라이브 부분과 나머지 부분으로 분리하여 튜플로 반환

```
>>> os.path.splitdrive("C:/test/python.txt")
```

- › `os.path.splitext(path)`는 확장자 부분과 그 이외의 부분으로 분리하여 튜플로 반환

```
>>> os.path.splitext("C:/test/python.txt")  
>>> os.path.splitext("C:/test")
```

외장 함수(shutil)

- › shutil은 파일 및 디렉터리를 복사, 이동 및 삭제에 유용한 모듈
- › shutil.copy(src, dst)는 입력 받은 경로를 내가 원하는 경로에 파일을 복사

```
>>> shutil.copy("C:/test/python.txt", "C:/test/copy.txt")  
>>> shutil.copy("C:/test/python.txt", "C:/test/create")
```

- › shutil.copytree(src, dst)는 입력 받은 경로 아래의 모든 디렉터리와 파일을 복사

```
>>> shutil.copy("C:/test", "C:/copy_test")
```

외장 함수(shutil)

- › `shutil.move(src, dst)`는 입력 받은 경로를 내가 원하는 경로로 이동

```
>>> shutil.move("C:/test", "C:/move")
>>> shutil.move("C:/move/python.txt", "C:/move/create")
>>> shutil.move("C:/move/create/python.txt", "C:/move/create/rename.txt")
```

- › `shutil.rmtree(path)`는 입력 받은 경로의 디렉토리를 영구적 삭제

- › 단, 파일을 지정하여 삭제는 불가능하며 파일이 포함된 디렉토리를 지정하여 전체 삭제는 가능

```
>>> shutil.rmtree("C:/copy_test/python.txt")
>>> shutil.rmtree("C:/copy_test")
```


외장 함수(time)

- › time 모듈은 시간과 관련된 함수이며 굉장히 많은 수의 함수가 존재
- › time.time()은 UTC를 사용하며 현재 시간을 실수 형태로 반환

```
>>> import time  
>>> time.time()
```

- › time.localtime(x)은 time.time()로 반환 된 실수 값을 입력값으로 넣어 연도, 월 등의 형태로 변경하여 반환

```
>>> info = time.localtime(time.time())  
>>> info  
>>> info[0]  
>>> info.tm_year
```

외장 함수(time)

- time.strftime(format code, time.struct_time)는 시간포맷을 원하는 모양으로 가공하여 문자열로 반환

포맷코드	설명
%a	요일 줄임말
%A	요일
%b	달 줄임말
%B	달
%c	날짜와 시간 출력
%d	날
%H	시간(24시간)
%I	시간(12시간)
%j	1년 누적 날짜
%m	달

포맷코드	설명
%M	분
%p	AM or PM
%S	초
%U	1년 중 누적 주(일 시작)
%w	숫자로된 요일(0-일요일)
%W	1년 중 누적 주(월 시작)
%x	현재 날짜 출력
%X	현재 시간 출력
%Y	년도 출력
%v	세기제외한 년도 출력

외장 함수(time)

› time.strftime() 함수 예시

```
>>> info = time.localtime(time.time())  
>>> time.strftime("%c", info)  
>>> time.strftime("%Y-%m-%d %X", info)
```

› time.sleep(x)는 입력 받은 숫자(초단위)만큼 프로그램 정지

› 보통 for문과 같은 loop 또는 thread에서 자주 사용

```
>>> for i in range(10):  
...     time.sleep(1)  
...     print(f"{i+1}초")
```

외장 함수(random)

- › random은 난수를 발생시키는 모듈
- › random.random()은 0~1.0 사이의 실수 중에서 난수 값을 반환

```
>>> import random  
>>> random.random()  
>>> random.random()
```

- › random.randint(start, end)는 입력한 범위의 정수 중에서 난수 값을 반환

```
>>> random.randint(1, 10)  
>>> random.randint(1, 100)
```

외장 함수(random)

- › `random.choice(x)`는 리스트, 튜플 등과 같은 자료형에서 하나의 요소를 무작위로 선택하여 반환
- › 딕셔너리 같은 경우 `key` 값을 숫자로 넣는다면 무작위로 선택되어 `value` 값을 반환

```
>>> random.choice([1,2,3,4,5])
>>> random.choice([[1,2],[3,4],[5,6]])
>>> random.choice(('a', 'b', 'c', 'd'))
>>> random.choice("test")
>>> random.choice({0:'a', 1:'b', 2:'c'})
```

외장 함수(random)

- › `random.shuffle(x)`은 리스트의 요소를 무작위로 섞음
(반환 값 없이 원본 x 리스트를 무작위로 섞음)

```
>>> a = [1, 2, 3, 4, 5]
>>> random.shuffle(a)
>>> a
```

- › `random.sample(x, num)`은 입력받은 리스트, 튜플 및 문자열을 받아 무작위로 뽑고 싶은 요소 개수를 입력하여
리스트로 반환

```
>>> a = [1, 2, 3, 4, 5]
>>> s = random.sample(a, 2)
>>> s
```