
Python



1. Python 개요 & 개발환경
2. Python 패키지 설치
3. Python 자료형
4. Python 제어문
5. Python 함수
6. Python 모듈과 패키지
7. Python 클래스
8. Python 예외처리

1. Python 개요 & 개발환경

Python에 대한 개요 & 개발환경에 대한 이해

1. Python 개요
2. Python 언어 특징
3. Python 개발환경
4. Python 실행

1. Python 개요

➤ 파이썬이란?

- 파이썬은 1989년 12월 네델란드 암스테르담에 사는 귀도 반 로섬(Guido van Rossum)이 혼자 집에서 재미삼아 시작한 프로그래밍 프로젝트에서 시작됐다.
- 귀도 반 로섬은 새로운 스크립트 언어용 인터프리터를 사용하기로 하고 프로젝트 이름을 파이썬이라고 지은 것이 파이썬이란 이름이 붙여지게 된 계기이다.

➤ 파이썬은?

- 파이썬은 작업 속도를 높여주고 효과적으로 시스템을 통합해 주는 프로그래밍 언어이다.
- 파이썬을 사용하면 생산성의 향상과 함께 유지 보수 비용이 절감되는 효과를 경험할 수 있다.
- 파이썬은 응용 프로그램의 프로토타입(prototype)을 만들어 내기에 좋은 언어이다.
- 파이썬은 확장성이 뛰어난 언어이다.

2. Python 언어 특징

- 대화식 인터프리터 언어
 - 파이썬은 객체지향을 강력히 지원하는 대화식 인터프리터 언어이다.
- 동적 자료형을 지원
 - 파이썬은 실행 시작에 동적으로 자료형(Data Type)을 결정한다.
 - 자료형에 관계없이 일반화된 코드를 작성할 수 있다.
- 플랫폼에 독립적인 언어
 - 파이썬은 리눅스와 유닉스, 윈도우, Mac OS 등 대부분의 운영 체제에서 실행된다.
 - 플랫폼에 독립적이며 코드 이식이 쉽다.
- 개발 기간 단축에 초점을 맞춘 언어
 - 파이썬은 실행의 효율성보다는 개발 기간 단축에 맞춘 언어이다.

2. Python 언어 특징

➤ 간단하고 쉬운 문법

- 파이썬의 간단한 문법과 깔끔한 구문은 개발자가 아니어도 배우기 쉽고 사용하기 쉽다.
- 객체지향 언어로서 파이썬은 재사용이 가능한 코드를 쉽게 작성할 수 있다.
- 파이썬은 들여쓰기(Indentation)로 블록을 구분해서 코드에 대한 가독성을 높이고 있다.

➤ 고수준의 자료형을 제공

- 파이썬은 리스트(list)와 사전(dictionary), 문자열(string), 튜플(tuple), 집합(set)등 고수준의 자료 구조를 제공한다.

➤ 자동으로 관리되는 메모리

- 파이썬은 쓰레기 수집(Garbage Collection)기능을 사용하여 필요할 때 메모리를 자동으로 할당하고 메모리 사용이 끝나면 자동으로 해제한다.

➤ 팀 단위 작업에 유용한 언어

- 파이썬은 모듈 단위의 코드를 쉽게 작성하고 결합할 수 있다.
- 각 모듈은 메인 프로그램이면서 다른 모듈의 라이브러리이기도 하다.

2. Python 언어 특징

➤ 쉬운 유지 보수

- 파이썬의 깔끔한 코드는 이해하기 쉬워서 코드에 대한 유지 보수가 쉽다.

➤ 수많은 라이브러리를 제공

- 파이썬은 수많은 라이브러리를 제공한다.

➤ 짧아지는 코드

- 파이썬은 일급 함수(First Class Function)을 지원한다. 일급 함수는 함수 객체를 변수에 저장할 수 있고, 함수에서 반환 값으로 사용할 수 있으며, 함수에 인수로 전달할 수 있는 함수를 말한다.
- 파이썬은 다중 상속과 지연 바인딩을 지원하는 객체지향 언어이다.

➤ 높은 확장성

- 파이썬은 일명 접착제 언어(Glue Language)라고도 한다. 다른 언어나 라이브러리에 쉽게 접근해 사용할 수 있기 때문이다.
- C/C++과 잘 결합할 수 있고 소스 없는 라이브러리도 래퍼(Wrapper) 함수만 사용하면 파이썬에서 사용할 수 있다.

2. Python 언어 특징

➤ 확장 및 내장

- C와 C++, Fortran을 이용하여 파이썬 모듈을 작성하는 것이 가능하고, 역으로 C와 C++, Fortran에서 파이썬 함수를 호출하는 것도 가능하다.

➤ 무료

- 파이썬 저작권은 2001년부터 비영리 기구인 파이썬 소프트웨어 재단(Python Software Foundation)에서 관리한다.
- 상용으로 사용할 경우에도 무료이다.
- 파이썬 라이선스는 여러분이 변경한 내용을 공개하지 않고도 배포할 수 있다.
- GPL 호환을 유지하는 이유는 GPL로 배포되는 다른 소프트웨어를 파이썬과 함께 사용하는 것을 가능하게 하기 때문이다.

2. Python 언어 특징

➤ 파이썬 사용 적합 분야

분야	설명
GUI	PyQt와 WxPython이 대표적이 GUI이다. 두 개의 모듈은 플랫폼에 독립적이다.
웹 프로그램	Zope와 Django등 웹 프레임워크가 준비되어 있다.
네트워크 프로그램	소켓 응용 프로그램을 작성하기 쉬우며, SOAP와 같은 RPC 프로토콜, 인터넷 프로토콜등 다양하게 지원한다.
DB 프로그램	SQLite가 내장되어 있으며, 다른 DB에 대한 파이썬 인터페이스가 만들어져 있으며, 응용프로그램의 프로토타입을 짧은 시간에 개발할 수 있다.
텍스트 처리	정규식(Regular Expression)을 강력하게 지원하고 유니코드를 지원하며, XML도 지원한다.
수치 연산	Matlab과 같이 배열 연산을 지원하는 Numpy등 다양한 모듈이 준비되어 있다.
병렬 연산	Ipython을 이용한 병렬 연산을 지원하며, MPI와 PBS 라이브러리의 Python 바인딩이 존재한다.
기타	COM 인터페이스와 AI, 그래픽스, 분산 처리 등 패키지가 다양하게 준비되어 있다.

2. Python 언어 특징

➤ 파이썬 인터프리터 종류

- Cpython : C로 작성된 파이썬 인터프리터, 기본적으로 파이썬이라 하면 Cpython을 의미한다.
- Jython : Java로 구현된 파이썬 인터프리터, JVM에서 작동 가능하다. Java class를 사용이 가능하며, Swing, AWT등도 지원한다.
- IronPython : .Net 플랫폼용으로 개발된 파이썬 인터프리터, C#으로 구현되어 있다.
- PyPy : 파이썬으로 구현된 파이썬 인터프리터, Cpython보다 빠르게 수행되는 것을 목표로 하고 있다.
- 그 외에 여러 분야에서 사용되는 파이썬 인터프리터가 있다.

2. Python 언어 특징

➤ 파이썬 사용 프로젝트

- 파이썬으로 작성된 프로그램 : BitTorrent, MoinMoin, Scons, Trac, Yum등이 있다.
- 웹 프레임워크 : CherryPy, Django등이 있다.
- 파이썬을 임베딩해서 사용한 프로그램 : GIMP, Maya, Paint Shop Pro 등이 있다.
- Youtube.com, Google Groups, Google Maps, Gmail등의 서비스들은 서비스 백엔드(backend)에서 파이썬이 사용되고 있으며, google, NASA, Yahoo, Naver 등의 회사에서도 많이 사용되고 있다.

3. Python 개발 환경

➤ 파이썬 공식 사이트(<https://www.python.org>)



3. Python 개발 환경

- 파이썬 자체만 설치할 경우에는 파이썬 공식 site에서 download하여 설치하는 방법을 사용하는데 이 경우 파이썬 인터프리터를 설치한 후 여러 패키지들을 설치하는 경우 의존성같은 문제 때문에 필요한 패키지들을 일일이 설치하는 것이 불편하다.
- 파이썬 배포판(Distributions) : 기본 파이썬 인터프리터 이외에도 다양한 패키지와 개발 도구를 함께 제공하여 설치할 수 있게 해주는 패키지이다. 배포판은 여러가지가 존재함으로 사용하는 OS, 32/64비트 지원 여부, 무료/상용 여부를 고려하여 선택한다.
 - ❖ 파이썬 인터프리터, 패키지(라이브러리), 패키지 관리 시스템, 콘솔, 강화된 인터프리터등을 포함한다.
 - ❖ 배포판 종류 : Anaconda, ActiveState, pythonxy, winpython, Conceptive, Enthought Canopy, PyIMSL Studio, eGenix PyRun등이 있다.

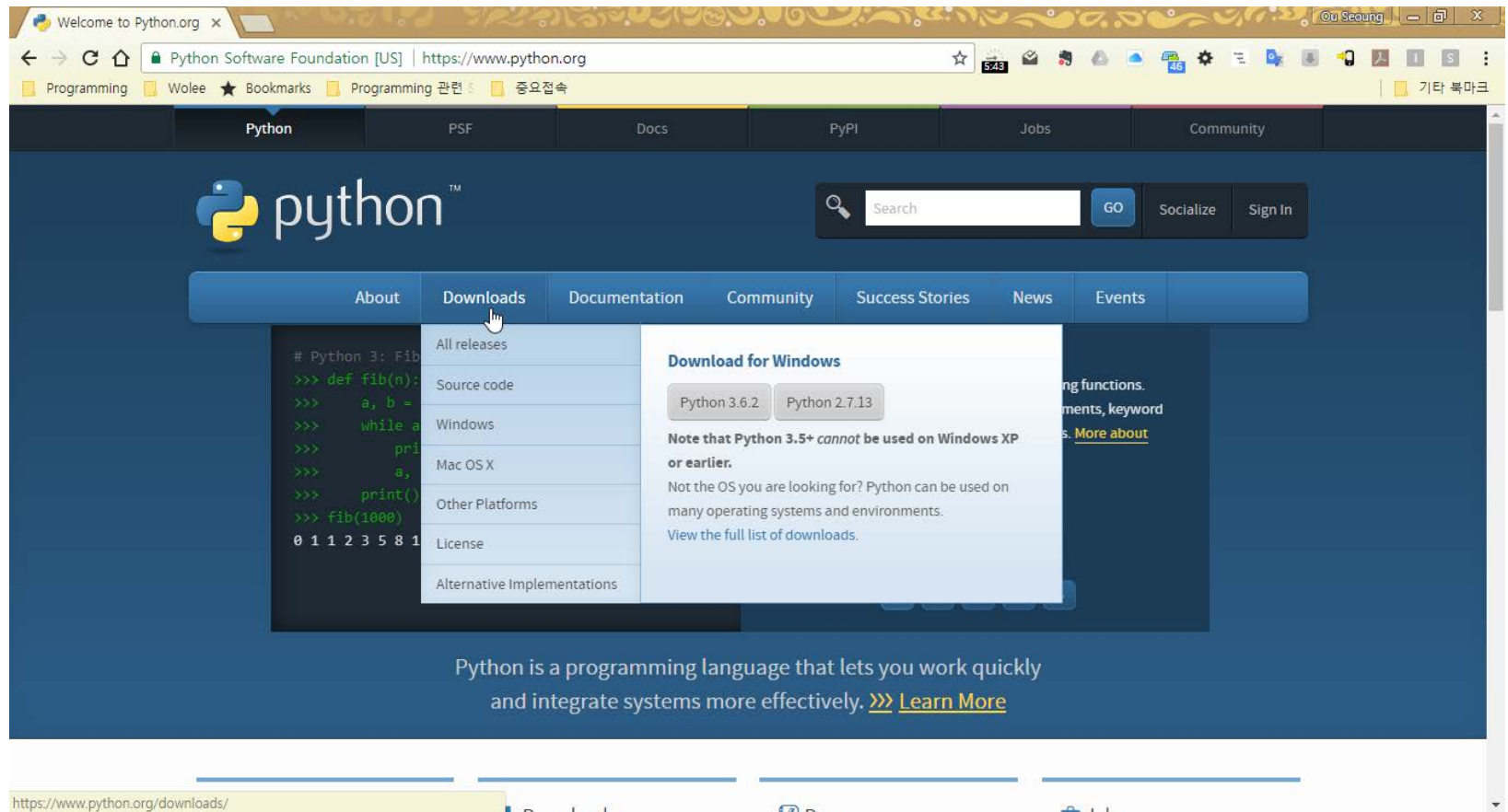
3. Python 개발 환경

- Anaconda(<https://www.anaconda.com/>)
 - 컨티눔(Continuum)사가 제작
 - 가장 늦게 발표되었으나 뛰어난 완성도로 인해 현재 가장 널리 사용되는 사실상의 표준(de facto standard) 파이썬 배포판이다.
 - 모든 패키지가 컴파일이 필요 없는 binary 형태로 제공되기 때문에 설치 속도가 빠르고 패키지 의존성을 관리해주므로 패키지 관리가 편리한다.
 - 모든 플랫폼에 대해 완벽한 패키지 제공

3. Python 개발 환경

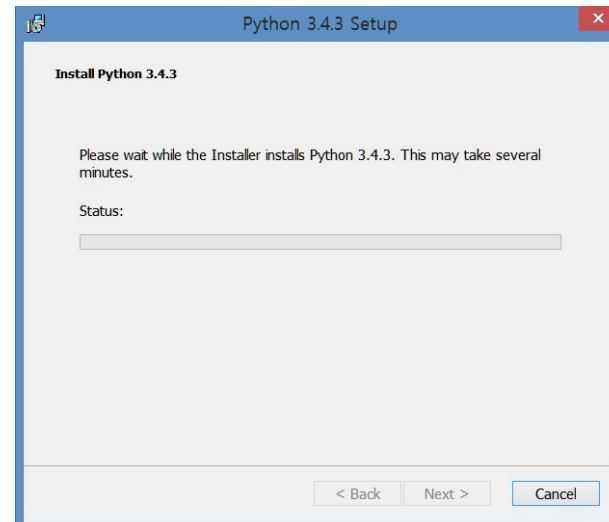
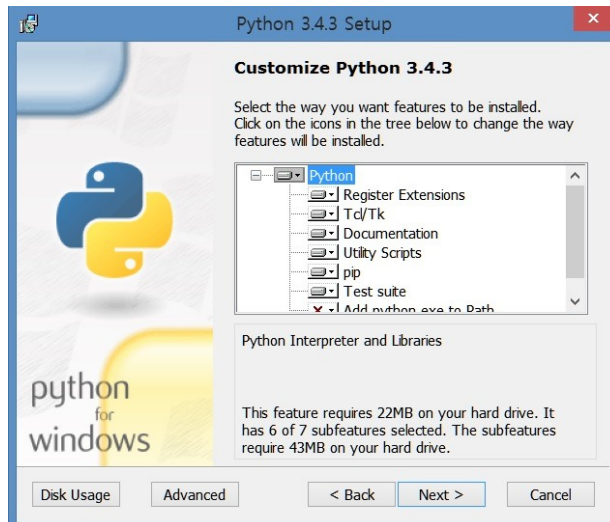
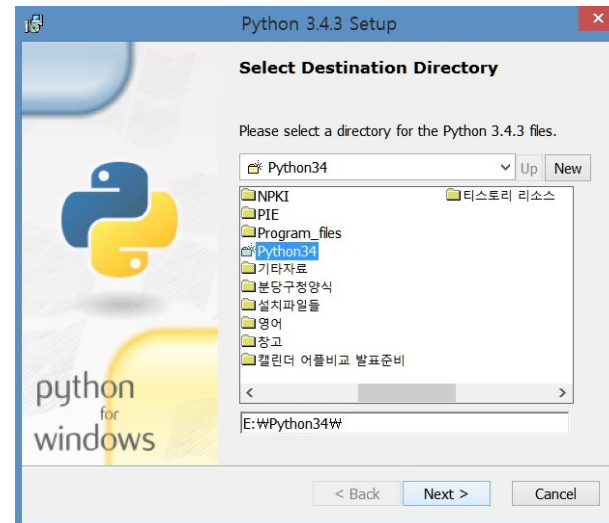
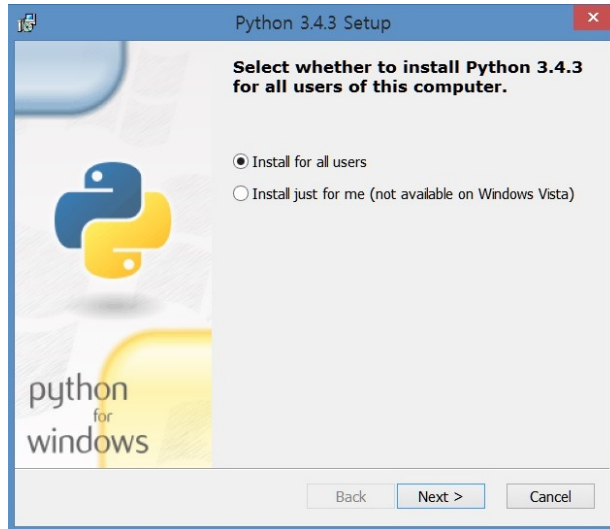
➤ Python 설치

1. Python 설치 파일 Download



3. Python 개발 환경

2. Python 설치



3. Python 개발 환경

3. 환경 변수 설정

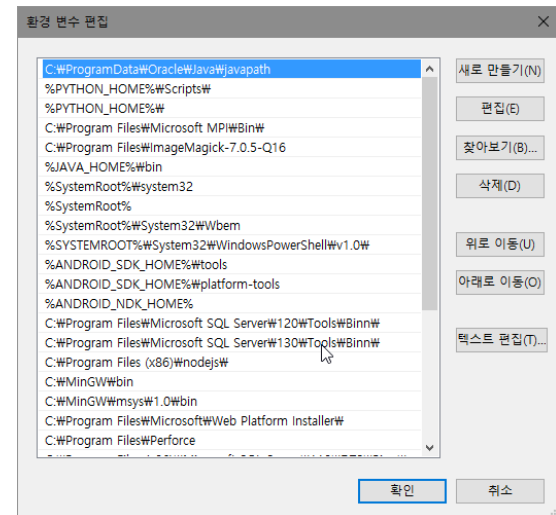
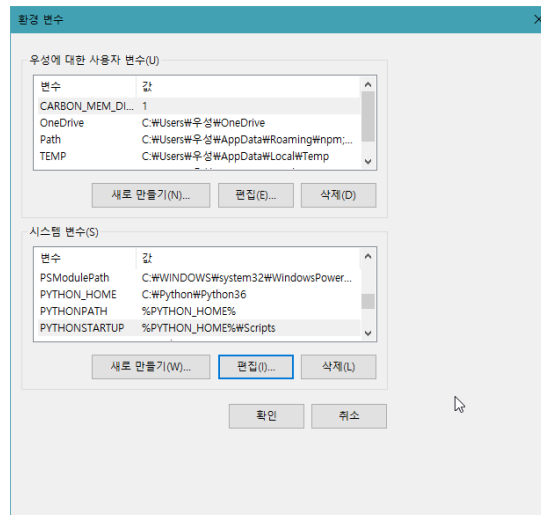
① 설치가 완료된 후 환경 변수를 설정하면 명령 프롬프트 창에서 언제든지 Python을 사용할 수 있다.

② [내컴퓨터]우클릭 - [속성] - [고급 시스템 설정] - [환경변수]

PATH 환경 변수 : python 실행 파일의 경로를 지정한다.

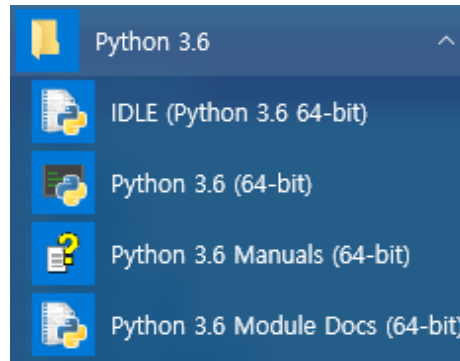
PATHONPATH 환경 변수 : import를 할 때 파이썬 모듈을 찾는 추가 경로

PATHONSTARTUP 환경 변수 : 파이썬 인터프리터를 실행할 때 자동으로 실행되는 파이썬 스크립트 파일을 나타낸다. 대화식 모드에서 매번 사용해야 할 모듈이 있으면 활용한다.



3. Python 개발 환경

4. 파이썬 구성 요소



구성 요소	설명
IDLE(Python GUI)	파이썬의 표준 GUI 인터프리터이다. Tcl/tk로 작성
Python(command Line)	명령 프롬프트 창에서 실행 되는 파이썬 인터프리터이다.
Python Manuals	HTML 형식의 파이썬 온라인 문서이다. 라이브러리 레퍼런스를 주로 참조한다.
Python Module Docs	원하는 모듈을 쉽게 찾을 수 있는 도구이다.

3. Python 개발 환경

➤ Python IDE(통합 개발 환경)

- IDE(통합 개발 환경)은 일반적으로 프로그램과 연관되는 소스 코드 편집, 컴파일 및 빌드, 실행, 디버그 과정을 하나의 도구에 통합시킨 것으로 개발자의 개발 생산성과 효율성에 지대한 영향을 미친다. 따라서 프로젝트를 진행하기 이전에 어떤 통합 개발 환경으로 개발할지를 선택하여야 한다.
- Python의 IDE는 기본 제공 IDE와 3rd-party에서 제공되는 IDE로 나누어 볼 수 있다.
 1. IDLE : Python설치시에 자동으로 설치되는 GUI기반의 인터프리터이면서 IDE의 역할을 수행한다.
 2. Visual Studio를 Python IDE로 사용 : Visual Studio에 PTVS(Python Tools for Visual Studio) extension을 설치하여 사용하는 방법이 있다. PTVS는 .Net Framework 4.5이상의 환경을 요구하고 있다. (<https://pytools.codeplex.com/releases>)
 3. Eclipse를 Python IDE로 사용 : Eclipse에 PyDev plug-in을 추가 하여 사용하는 방법이 있다. (<http://pydev.org/updates>)
 4. Visual Studio Code를 사용 : 크로스 플랫폼 소스 편집기로서 윈도우즈, 리눅스, MacOS 에서 무료로 사용할 수 있다. Visual Studio Code를 설치한 후 여러 Extension들을 설치할 수 있는데, 파이썬 Extension을 설치하면, 파이썬 프로그래밍과 디버깅을 할 수 있다. (<https://code.visualstudio.com>)

4. Python 실행

➤ 파이썬 실행 방법

- 대화식 모드로 실행 하는 방법
 - IDLE나 Python(Command Line)을 실행하여 직접 파이썬 코드를 입력하여 결과를 확인하는 방법
- 스크립트 파일을 일괄적으로 실행하는 방법
 - 에디터 프로그램을 이용하여 파이썬 프로그램을 작성한 후 확장자.py로 저장한 후에 파이썬 인터프리터를 이용하여 일괄적으로 실행하는 방법
 - 파이썬 IDE 환경에서는 소스 코드 편집, 실행, 결과 확인, 디버깅을 일괄적으로 수행할 수 있으므로 편리하다.

2. Python 패키지 설치

Python 패키지 설치 방법에 대한 이해

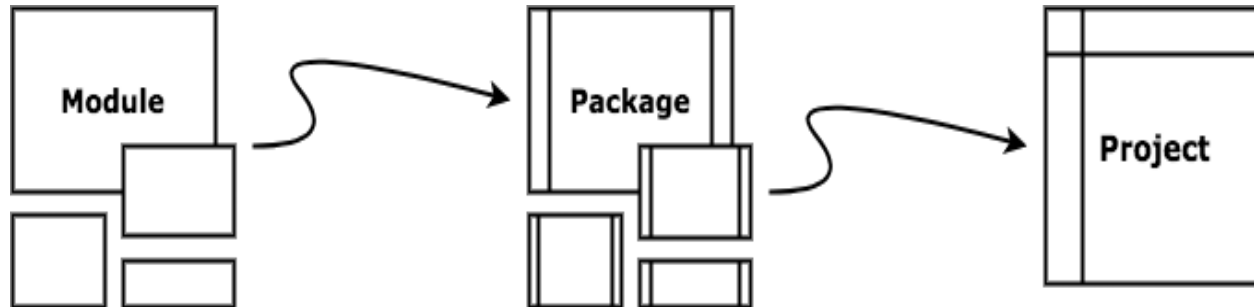
1. Python 패키지
2. 소스로 배포되는 패키지 설치
3. pip를 이용하여 배포되는 패키지 설치
4. pyenv를 이용한 가상 환경 설정

1. Python 패키지

- 파이썬 패키지를 이용하면 대규모 시스템을 분해하고 시스템의 모듈을 사용자와 다른 개발자가 효율적으로 사용하고 재사용할 수 있는 일관된 방식으로 구성할 수 있다.
- 파이썬의 "배터리 포함(Batteries Included)"이라는 모토는 표준 라이브러리에 수많은 유용한 패키지가 미리 로드 되어 있다는 의미이다.
- 모듈은 확장자 .py인 소스 파일로서 프로그램을 구성하는 함수와 클래스가 표현되어 있습니다.
- 패키지는 파이썬의 계층적 네임스페이스 개념을 표현한 것으로서 코드 구성 및 이름 충돌을 방지하는데 이바지합니다.
- 패키지는 하위 패키지 및 모듈의 계층 구조를 나타내지만 하위 패키지가 디렉터리 및 하위 디렉터리인 파일시스템 기반일 필요는 없습니다. 패키지는 그보다 훨씬 더 유연합니다.

1. Python 패키지

➤ 모듈, 패키지, 프로젝트의 관계



```
$PROJECT_ROOT
├── README.rst
├── docs
├── rsc
├── src
│   ├── package
│   │   ├── __init__.py
│   │   └── ...
│   └── main.py ....
├── tests
│   ├── package
│   │   └── ...
│   └── test_main.py
└── setup.py
```

- 파이썬은 디렉토리에 `__init__.py` 파일이 존재하면 이를 패키지라 여긴다.
- 패키지로부터 모듈을 사용하기 위한 파이썬 키워드는 `from / import` 이다.
`from [package name] import [module name]`
- 외부에서 패키지를 참조하는 시점에 해당 패키지의 `__init__.py`이 실행된다는 점을 기억

1. Python 패키지

➤ 설치된 모듈 확인 방법

1. pydoc를 이용하는 방법

[명령 프롬프트] `pydoc modules`

2. help를 이용하는 방법

[명령 프롬프트] `python -c 'help("modules")'`

3. pip를 이용하는 방법

- pip가 설치되어 있어야 한다.
- 기본 내장 모듈은 출력 되지 않는다.

[명령 프롬프트] `pip list`

4. get_installed_distributions를 이용하는 방법

- pip가 설치되어 있어야 한다.
- 기본 내장 모듈은 출력 되지 않는다.
- 설치 위치 확인 가능하다.

[명령 프롬프트] `python -c 'import pip, pprint; pprint.pprint(pip.get_installed_distributions())'`

2. 소스로 배포되는 패키지 설치

➤ 압축 소스 파일로 배포되는 패키지를 내려 받는다면 다음과 같은 방법으로 설치할 수 있다.

1. 압축 소스 파일(*.tar.gz, *.tgz, *.tar.bz2)을 내려 받는다.
2. 내려 받은 파일의 압축을 푼다.
3. 패키지를 설치한다.

▪ 설치 예

```
$ tar zxvf aaa-0.6.27.tar.gz
```

```
$ cd aaa-0.6.27
```

```
$ python setup.py install
```

```
# 파일 압축을 푼다.
```

```
# 소스 디렉터리로 이동한다.
```

```
# 설치한다.
```

3. pip를 이용하여 배포되는 패키지 설치

- 배포되는 파이썬 패키지를 관리하는 방법 중 가장 간단하고 관리하기 쉬운 방법이 pip를 이용하는 것이다.
- pip(Python Package Index, <https://pypi.python.org/pypi>)에 등록된 패키지들을 설치하고 관리해주는 패키지 관리 시스템으로 파이썬 패키지를 설치하는 방법 중 가장 많이 선호된다.
 - 사용 예(django를 설치할 때 pip이용)

\$ pip install django	# 패키지 설치
\$ pip install djnago --upgrade	# 패키지 업그레이드
\$ pip uninstall django	# 패키지 삭제
\$ pip list -outdated	# 최신이 아닌 패키지 목록 표시
\$ pip search django	# 질의어를 포함하는 패키지 검색

4. pyvenv를 이용한 가상 환경 설정

- 파이썬의 "가상 환경(virtual environment)"은 패키지가 시스템에 설치되는 것이 아니라 특정 애플리케이션의 한정된 환경에서 설치할 수 있다.
- 가상 환경은 파이썬 3.4부터 기본으로 추가된 pyvenv 도구로 설정

myenv라는 가상환경을 위한 폴더가 만들어지고 필요한 파일들이 복사

```
python -m venv myenv
```

myenv 가상 환경을 활성화한다.

```
myenv\Scripts\activate
```

여기서 설치되는 패키지는 myenv 폴더 안의 site-packages 폴더에 설치되어 지역적으로만 유효하다.

```
pip install Django
```

가상 환경 사용을 종료하려면 deactivate를 실행한다.

```
myenv\Scripts\deactivate
```

3. Python 자료형

Python 자료형에 대한 이해

1. Python 프로그램 구조
2. Python 표준 입/출력
3. Python 기본 자료형 개요
4. Python bool/bytes/숫자형
5. Python 문자열
6. Python 리스트
7. Python 튜플
8. Python 집합
9. Python 사전
10. Python 객체의 복사와 형변환

1. Python 프로그램 구조

➤ 파이썬 프로그램 구조

```
def hello() :  
    print( 'Hello World' )  
  
hello()
```

- 파이썬 프로그램의 확장자는 .py이다.
- 파이썬 은 별도의 시작함수나 클래스가 존재 하지 않고, 코드 작성 순서대로 실행되는 특징을 가지고 있다.
- 위 코드에서 def는 함수 정의 시, print()는 표준 출력 장치에 출력 시 사용되는 파이썬 함수이며, hello()는 사용자가 정의한 함수이다.

```
def hello() :  
    print( 'Hello World' )  
  
if __name__ == '__main__':  
    hello()
```

- 위 코드에서 "__name__ == '__main__'" 이라는 if 조건절은 인터프리터에 의해서 직접 실행될 경우에 실행하고 싶은 코드 블록이 있는 경우에 사용하는데 다른 언어의 시작 함수와 같은 용도로 사용하는 경우도 있다.

1. Python 프로그램 구조

➤ 변수

- 유니 코드 문자나 밑줄(_)로 시작해야 한다.
- 이름에 공백이 없어야 한다.
- 아스키 코드의 특수 문자는 사용할 수 없다.
- 예약어가 아니어야 한다.
- 파이썬 전체 예약어는 다음과 같이 입력해서 알아 낼 수 있다.

```
>>> import keyword  
>>> keyword.kwlist  
>>> len(keyword.kwlist)
```

- 주석(comment) : 주석은 줄 어디에서나 시작할 수 있으며 "#" 다음에 나오는 텍스트는 주석으로 취급한다.
- 여러 줄을 한 줄로 잇기 : 코드를 한 줄에 적어야 하지만 그렇게 하지 못 할 경우 "\"를 사용하여 다음 줄을 현재 줄과 이어지게 된다.

1. Python 프로그램 구조

➤ 문자열로 표현된 파이썬 코드 실행 하기

- `eval()` : 문자열로 표현된 파이썬 식(expression)을 인수로 받아 파이썬 컴파일 코드로 변환한다. 결국 파이썬 인터프리터가 번역하여 실행할 수 있다.

```
>>> a = 1
```

```
>>> a = eval( 'a + 4' )
```

```
>>> a
```

- `exec()` : 문자열로 표현된 파이썬 문(Statement)을 인수로 받아 파이썬 컴파일 코드로 변환하다. 결국 파이썬 인터프리터가 번역하여 실행할 수 있다.

```
>>> a = 5
```

```
>>> exec( 'a = a + 4' )
```

```
>>> a
```

- `exec(object[, globals[, locals]])`

globals : 전역 영역의 사전

locals : 지역 영역의 사전

1. Python 프로그램 구조

- `eval()/exec()` 함수는 문자열로 표현된 코드를 분석해서 파이썬 컴파일 코드로 변환한다. 하지만 반복적으로 실행하면 변환에 필요한 시간이 늘어난다.
- `compile()` : 문자열을 파이썬 컴파일 코드로 한번 변환하고서, 반복해서 실행 할 때마다 변환된 코드를 실행하는 기능을 제공한다.

```
>>> code = compile( 'a + 1', '<string>', 'eval' )
```

```
>>> a = 1
```

```
>>> a = eval( code )
```

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

- `compile(string, filename, mode)`

string : 코드 문자열

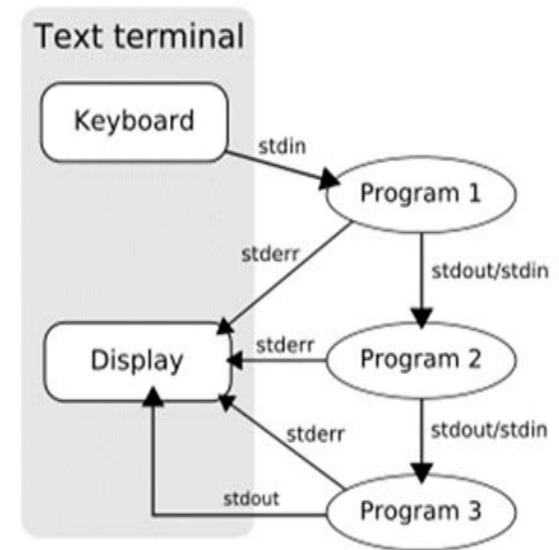
filename : 코드 문자열이 저장된 파일 이름

mode : 어떤 종류의 코드가 컴파일 되어야 하는지 지정

exec이면 여러 개의 문을 컴파일, single이면 하나의 문을 컴파일, eval이면 하나의 식을 컴파일한다.

2. Python 표준 입/출력

- 표준 입력(stdin) : 프로그램으로 들어가는 데이터(보통은 문자열) 스트림이다. 리다이렉션 없이 프로그램을 시작한 경우, 표준 입력 스트림은 키보드에서 받아온다.
- 표준 출력(stdout) : 프로그램이 출력 데이터를 기록하는 스트림이다. 리다이렉션 없이 표준 출력은 프로그램을 시작한 텍스트 터미널이 된다.
- 표준 오류(stderr) : 프로그램이 오류 메시지나 진단을 출력하기 위해 일반적으로 쓰이는 또다른 출력 스트림이다. 표준 출력과는 독립적인 스트림이며 별도로 리다이렉트될 수 있다.



2. Python 표준 입/출력

➤ 표준 입력(stdin) 장치에서 입력 받는 파이썬 함수 : input()

- 변수 = input()
 - Enter를 입력할 때까지 입력 받은 내용을 문자열로 반환한다.
 - 입력 내용을 문자열이 아닌 자료형으로 사용시는 형변환을 해서 사용한다.

```
>>> a = input()
Life is too short, you need python
>>> a
>>> type( a )
```

- 변수 = input("프롬프트 string")
 - 프롬프트 string은 입력 시 사용자에게 출력되는 프롬프트 문자열이다.
 - Enter를 입력할 때까지 입력 받은 내용을 문자열로 반환한다.
 - 입력 내용을 문자열이 아닌 자료형으로 사용시는 형변환을 해서 사용한다.

2. Python 표준 입/출력

```
>>> number = int(input( "input number : " ) )  
input number : 5  
>>> number  
>>> type( number )
```

2. Python 표준 입/출력

➤ 표준 출력(stdout) 장치에 출력하는 파이썬 함수 : print()

- `print(objects, sep='', end='\n', file=sys.stdout, flush=False)`
 - 하나 이상의 문자열 또는 숫자를 쉼표로 구분하여 나열
 - 쉼표로 구분된 데이터 사이에는 출력 시 빈칸이 자동 삽입
 - `objects` : 출력될 값
 - `sep` : default로 연속적인 값을 하나의 공백으로 구분, 다른 키워드 인수를 사용하여 다른 구분 문자열 지정
 - `end` : default 줄 바꿈('\n')을 의미, 다른 출력 문자로 변경 가능
 - `file` : default `sys.stdout`인 출력 스트림 콘솔로 출력
 - `flush` : 출력 버퍼 삭제 기능, 값을 파일에 쓴 후 버퍼 삭제 기능

```
>>> print( 'add : ', 4 + 5, 'sub = ', 4 - 2 )
>>> print( 1, 2 ); print( 3, 4 )
>>> print( 1, 2, end = ' ' )
>>> print( 1, 2, 3 ); print( 1, 2, 3, sep = ', ' )
>>> f = open( 'out.txt', 'w' ); print( 1, 2, 3, file = f )
>>> f.close(); open( 'out.txt' ).read()
```

2. Python 표준 입/출력

- 출력 서식 formatting
- `print("[flag][width][.precision]type" % (value))`
 - flag : %로 지정하여 formatting 시작을 알림(C언어 형식)
 - width : 출력될 value의 전체 자릿수 지정
 - precision : 소수 이하 자릿수 지정
 - type : 출력될 value의 데이터 형을 나타냄
 - value : 출력될 내용

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

2. Python 표준 입/출력

```
>>> print( "%10d" % ( 100 ) )
>>> print( "%010d" % ( 100 ) )
>>> print( "%-10d" % ( 100 ) )
>>> print( "%.2f" % ( 3.141592 ) )
>>> print( "%10.2e" % ( 3.141592 ) )
>>> print( "%d / %d = %.2f" % ( 5, 3, 5 / 3 ) )
```

2. Python 표준 입/출력

- `format(value, format_spec)`
 - `value`를 `format_spec`에 지정된 서식의 문자열로 변환하여 출력한다.

```
>>> print( format( 1.234567, "10.3f" ) )
```

```
>>> print( '{0} {1}'.format( 'apple', 7.77 ) )
```

❖ `{0},{1}`은 `format()`의 첫 번째와 두 번째 인수를 각각 의미한다.

```
>>> print( '{0:<10}{1:5.2f}'.format( 'apple', 7.77 ) )
```

```
>>> print( '{0:>10}{1:5.2f}'.format( 'apple', 7.77 ) )
```

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

❖ `<` : 왼쪽 정렬 `>` : 오른쪽 정렬 `^` : 가운데 정렬

❖ `=<10, =>10, ^=10` : 공백을 '='로 채울 때 사용

```
>>> 'Name : {0}, Phone : {1}'.format( "hong", 5432 )
```

```
>>> '{{ and }}'.format()
```

2. Python 표준 입/출력

➤ 복잡한 데이터를 출력할 때 좀 더 깔끔한 출력 결과를 얻을 때 사용하는

파이썬 함수 : pprint()

- pprint()는 pprint 모듈에 포함된 함수 이므로 'import pprint'를 먼저 실행 한 후 사용해야 한다.

```
>>> import pprint
>>> numbers = [ [1, 2, 3], [4, 5], [6, 7, 8, 9] ]
>>> print( numbers )
>>> print( *numbers )
>>> print( *numbers, sep = '\n' )
>>> pprint.pprint( numbers )
>>> pprint.pprint( numbers, width = 20 )
>>> pprint.pprint( numbers, width = 20, indent = 4 )
```

3. Python 기본 자료형 개요

- 자료형이란 프로그래밍을 할 때 사용되는 숫자, 문자, 문자열 등을 메모리에 저장/이용하는 방법을 의미한다. 즉 변수 생성을 의미한다.
- 파이썬은 내장 자료형으로 다양한 정보를 관리할 수 있도록 제공되고 있으며, 별도의 자료구조 설계 없이도 쉽게 데이터 관리를 할 수 있다.

❖ 파이썬 내장 자료형

자료형	설명	예
bool	True/False를 나타내는 자료형	True, False
int, float, complex	정수/실수/복소수 등 숫자를 표현하는 자료형	123, 1.45, 5+4j
str	유니코드 문자집합을 표현하는 자료형	'apple', "banana", ' 'egg' ', ""hot dog""
bytes	0 ~ 255사이의 코드 모임을 표현하는 자료형	b'Python'
list	순서가 있는 파이썬 객체 집합을 표현하는 자료형	['apple', 'banana']
dict	순서가 없는 파이썬 객체 집합을 표현하는 자료형	{'ham':4, 'spam':5}
tuple	순서가 있는 파이썬 객체 집합을 표현하는 자료형	('ham', 'spam')
set	집합을 표현하는 자료형	{1, 2, 3}

3. Python 기본 자료형 개요

➤ 파이썬은 자료형을 다음과 같이 몇 가지 형태로 분류하고 있으며, 내장 자료형 뿐만 아니라 사용자가 클래스를 이용해서 새로 정의하는 자료형에도 적용된다.

❖ 데이터 저장 방법에 따른 자료형 분류

자료형	설명	예
Direct 형	직접 데이터를 표현하는 자료형이다. 숫자형이 여기에 속한다.	int, float, complex
Sequence 형	다른 데이터를 포함하는 자료형이다. 순서가 있는 집합 자료형이다.	list, str, tuple, bytes, bytearray, range
Mapping 형	다른 데이터를 포함하는 자료형이다.	dict
Set 형	순서가 없고, 중복된 항목도 없다.	set, frozenset

❖ 변경 가능성에 따른 자료형 분류

자료형	설명	예
변경 가능형(Mutable)	데이터의 값을 변경할 수 있다.	list, dict, set, bytearray
변경 불가능형(Immutable)	데이터의 값을 변경할 수 없다.	int, float, complex, str, tuple, frozenset

3. Python 기본 자료형 개요

❖ 데이터 저장 개수에 따른 자료형 분류

자료형	설명	예
리터럴 형(Literal/Scalar)	한 가지 객체만 저장한다.	str, int, float, complex
저장형(Container)	여러 가지 객체를 저장한다.	bytes, bytearray, list, tuple, dict, set, frozenset

❖ 자료형 분류 요약

자료형	저장 모델	변경 가능성	접근 방법
int, float, complex	리터럴	불가능	Direct
str	리터럴	불가능	Sequence
list	저장	가능	Sequence
tuple	저장	불가능	Sequence
dict	저장	가능	Mapping
bytes	저장	불가능	Sequence
bytearray	저장	가능	Sequence
set	저장	가능	반복자
frozenset	저장	불가능	반복자

3. Python 기본 자료형 개요

- 파이썬에서 변수를 정의할 때 별도 자료형을 지정하지 않더라도 변수에 값을 저장할 때 자료형을 판단하는 동적 자료형을 지원하는 언어이다.

변수명 = 변수에 저장할 값

❖ 변수의 자료형은 변수에 저장할 값에 따라 실행 시간에 결정된다.

- 자료형을 분류하는 중요한 기준은 데이터를 변경할 수 있느냐이다.
- 데이터를 변경할 수 있는 자료형을 Mutable, 변경할 수 없는 자료형을 Immutable로 구분한다.
- ❖ Mutable 자료형에는 list, dict, set, bytearray
- ❖ Immutable 자료형에는 숫자형, str, tuple, bytes, frozenset

3. Python 기본 자료형 개요

```
>>> s = [ 1, 2, 3 ]      # list
>>> s
>>> s[1] = 10            # s[1] 내용 변경
>>> s                    # 전체 내용 출력

>>> s = ( 1, 2, 3 )      # dict
>>> s
>>> s[1] = 10            # s[1] 내용 변경
>>> s                    # error 발생
```


3. Python 기본 자료형 개요

- 다음과 같은 경우 값이 변경된 것으로 이해할 수 있지만 실제로는 값이 변경된 것이 아니고 참조만 변경 된 것이다.

```
>>> a = 1
```

```
>>> a = 2
```

- 파이썬에서 객체와 이름은 별도로 관리된다. 이름에 직접 값이 저장되는 것이 아니고, 이름은 언제나 객체를 참조하게 되어 있다. 결국 숫자 객체 1은 값이 변경되지 않았다. 이름 a가 다른 객체를 참조한다. 이때 숫자 객체 1은 참조되지 않으므로 메모리에서 제거된다.

3. Python 기본 자료형 개요

- 다음과 같은 예에서도 객체의 내용이 변경되는 것이 아니고, 이름에 의해 참조만 바뀌는 것이 된다.

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

- 그러나 객체 내부의 값을 변경하려고 할 때 list는 값을 변경할 수 있다.

```
>>> l = [ 4, 5, 6 ]  
>>> l[0] = 100  
>>> l
```

- 어떠한 이름으로 객체 전체를 변경하는 것은 새 객체를 할당하는 것이다. 객체를 변경한다는 의미는 객체 내부의 값을 변경할 수 있는가로 판단한다.

3. Python 기본 자료형 개요

- 파이썬에서 특별히 사용자가 관리해야 할 메모리는 없다. 모든 메모리는 자동으로 할당되고 자동으로 해제된다. 추가로 필요한 메모리가 있으면 시스템에서 알아서 확장해 준다.
- 더 이상 사용하지 않는 객체는 자동으로 제거된다. 이러한 작업을 Garbage Collection이라 한다.
- 파이썬은 모든 것을 객체로 관리하며, 모든 객체는 Reference Count라는 참조 횟수를 값으로 가지고 있다. 이 값은 얼마나 많은 대상이 객체를 참조하고 있는지를 나타내는 정수이다.

```
>>> x = y = z = 500          # 객체 500의 Reference Count는 3
>>> del x                    # 객체 500의 Reference Count는 2로 변경
>>> y = 1
>>> z = 1                    # 객체 500의 Reference Count는 0으로 변경
                              따라서 객체 500은 메모리로 부터 제거된다.
```

3. Python 기본 자료형 개요

- 객체 참조 횟수를 확인하는 방법은 다음과 같다.

```
>>> import sys
>>> a = 10234
>>> sys.getrefcount( a )           # 실제 참조 횟수는 getrefcount()가 넘겨주는 값 보다 1만큼 작은 값이다.
```

- id() 내장 함수는 객체의 고유한 값(id)인 주소를 식별한 수 있다. 두 객체의 id가 동일하면 같은 객체를 참조하는 것이다.

```
>>> a = 500; id( a )
>>> b = a; id( b )
>>> a is b                       # 같은 객체를 참조하고 있는지 비교
>>> a == b                       # 두 객체가 같은 값을 가지고 있는지 비교
>>> l1 = [ 1, 2, 3 ]; l2 = [ 1, 2, 3 ]
>>> l1 is l2    or l1 == l2
```

3. Python 기본 자료형 개요

➤ 메모리에 생성된 변수 삭제하기

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

```
>>> b = 3
```

```
>>> del( a )
```

```
>>> del( b )
```

- ❖ a와 b가 3이라는 객체를 가리켰다가 del()에 의해 삭제 되면 3의 reference count는 0가 되어 메모리에서 사라지게 된다.
- ❖ 하지만 사용한 변수를 del()를 이용하여 일일이 삭제할 필요는 없다. 파이썬이 자동으로 reference count가 0인 객체를 삭제해 준다.

4. Python bool/bytes/숫자형

- bool은 참이나 거짓을 나타내는 True와 False 두 상수를 갖는다.
- bool 값은 정수로 간주되어 True는 1, False는 0으로 간주한다.
- 식에서 bool 값을 알고 싶으면 bool()를 사용하며, bool 식은 주로 if문이나 while문에서 사용한다.

```
>>> a = 1
```

```
>>> a < 0
```

```
>>> a > 0
```

```
>>> True + True
```

```
>>> True + False
```

```
>>> bool( 3 )
```

```
>>> bool( [] )
```

4. Python bool/bytes/숫자형

- bytes는 0 ~ 255 사이 코드의 열이다. 문자열은 유니코드 문자의 열이다.

```
>>> a = b'Python rules'           # bytes 상수 선언은 b로 시작한다.
>>> a
>>> type( a )
>>> a > 0
```

- 문자열과 바이트 간의 차이점은 다음과 같다.

```
>>> s = '홍길동'                  # 문자열은 유니코드를 기본으로 한다.
>>> b = b'홍길동'                 # bytes는 1byte로 표현되는 문자만 가능
>>> b = b'hong'
>>> b'Python' + 'rules'           # bytes와 문자열의 연산은 지원 안됨
```

4. Python bool/bytes/숫자형

- bytes는 문자열에서 사용하는 연산을 거의 제공한다.

```
>>> b = b'Python rules'
>>> b[ 1:5 ]           # 인덱싱과 슬라이싱 모두 지원
>>> b'th' in b         # 멤버 검사 지원
>>> b.upper()          # 내장 함수 지원
>>> b.split()           # bytes를 공백을 기준으로 분리
>>> b.startswith( b'Py' ) # 시작하는 bytes 검사
>>> b.endswith( b'ld' )  # 끝나는 bytes 검사
```

- bytes를 문자열로 변환하려면 decode(), 문자열을 bytes로 변경하려면 encode() 사용

```
>>> b = b'Python rules'; b.decode()      # 기본값으로 UTF-8을 사용
>>> b.decode( 'cp949' )                   # 직접 인코딩을 지정할 수 있다.
>>> s = '홍길동'; s.encode()
```


4. Python bool/bytes/숫자형

- 변경이 가능한 bytes를 원하면 bytearray 자료형을 사용한다.

```
>>> b = b'Python rules'
>>> ba = bytearray( b )           # bytearray로 형변환
>>> ba[7] = ord( 'R' )           # ba[7] 내용 변경
>>> ba
>>> bytes( ba )                   # bytes로 형변환
>>> ba.decode()                   # 문자열로 변환
```

4. Python bool/bytes/숫자형

- 숫자 자료형에는 정수형과 실수형(부동 소수점형), 복소수형이 있다.
- 정수형 상수는 10진수와 2진수, 8진수, 16진수 상수가 있다.
- 정수형 상수의 표현 범위에는 제한이 없다.

```
>>> a = 23                                # 10진 상수
>>> type( a )                             # 자료형 확인
>>> isinstance( a, int )                  # a가 정수형인지 확인
>>> b = 0o23                              # 8진 상수
>>> c = 0x23                             # 16진 상수
>>> d = 0b1101                           # 2진 상수
>>> print( a, b, c, d )                  # 10진수로 출력
>>> 2 ** 1024                             # 정수 표현 범위에 제한은 없다.
>>> n = 2 ** 1024
>>> n.bit_length()                       # 필요한 비트 수를 얻을 수 있다.
```

4. Python bool/bytes/숫자형

- 다른 진수, 실수형, 문자열 값을 10진 정수로 변환 하려면 int() 사용

```
>>> int( '123', 5 )           # 5진수 123을 10진수로 변환
>>> int( 2.9 )                 # 소수점 이하는 버려진다.
>>> int( '123' )               # 문자열을 정수로 변환
>>> int( '123.45' )            # error
>>> int( float( '123.45' ) )   # 문자열을 실수형으로 다시 정수형으로 변환
>>> int( 2 + 3j )              # error
```

- 10진 정수를 2/8/16진수로 변환

```
>>> bin( 23 )                  # 10진수를 2진수로 변환
>>> oct( 23 )                  # 10진수를 8진수로 변환
>>> hex( 23 )                  # 10진수를 16진수로 변환
```

4. Python bool/bytes/숫자형

➤ 정수형 데이터를 다른 수치 자료형으로 변환

```
>>> a = 123
>>> float ( a )           # 실수형으로 변환
>>> str( a )              # 문자열로 변환
>>> complex( a )          # 복소수형으로 변환
```

➤ 정수형 데이터를 직접 바이트 열로 변환

```
>>> (1024).to_bytes( 2, byteorder = 'big' )           # 2byte little endian 변환
>>> (1024).to_bytes( 2, byteorder = 'little' )        # 2byte big endian 변환
>>> (-1024).to_bytes( 4, byteorder = 'big', signed = True )
>>> int.from_bytes( b'\x04\x00', byteorder = 'big' )   # byte열에서 정수형 변환
>>> int.from_bytes( b'\x00\x04', byteorder = 'little' )
>>> int.from_bytes( b'\xff\xff\xfc\x00', byteorder = 'big' )
>>> int.from_bytes( b'\xff\xff\xfc\x00', byteorder = 'big', signed = True )
>>> int.from_bytes( [ 4, 0 ], byteorder = 'big' )
```

4. Python bool/bytes/숫자형

- 실수형 상수는 소수점을 포함하거나 e나 E로 지수를 포함한다. 컴퓨터에서는 실수를 부동 소수점 방식으로 표현하기 때문에 부동 소수점 형이라고 부른다.

```
>>> a = 1.2                                # 소수점을 포함하면 실수이다.
>>> type( a )
>>> isinstance( a, float )
>>> b = 3e3                                # 지수(e)를 포함해도 실수이다.
>>> c = -0.2e-4                             # 지수부는 정수 일수 있으나 실수일수는
>>> print( a, b, c )                       # 없다.
```

- 실수형 상수는 C나 Java 언어에서의 double형과 동일하며 8byte로 표현한다. 수치 표현 범위는 유효 자리 15자리이다.

```
>>> import sys; sys.float_info; sys.float_info.max; sys.float_info.min
```

4. Python bool/bytes/숫자형

- 무한대의 수는 다음과 같이 표현한다.

```
>>> float( 'inf' )  
>>> float( '-inf' )  
>>> infinity = float( 'inf' )  
>>> infinity / 1000
```

- 실수형인 경우는 정수로 오차 없이 표현할 수 있는 값인지를 `is_integer()`를 사용하여 확인할 수 있다.

```
>>> a = 1.2  
>>> a.is_integer()  
>>> a = 2.0  
>>> a.is_integer()
```

4. Python bool/bytes/숫자형

- 실수를 정수로 변환하는 또 다른 방법은 반올림(Round), 올림(Ceil), 내림(Floor)를 사용하는 것이다.

```
>>> round( 1.2 )           # 반올림
>>> round( 1.0 )           # 반올림
>>> import math
>>> math.ceil( 1.2 )        # 1.2보다 같거나 큰 정수
>>> math.floor( 1.9 )       # 1.9보다 같거나 작은 정수
```

- 실수 연산에서 발생하는 반올림 에러는 10진수에서 2진수로 변환 시 진법 변환 에러가 발생하여 나타난다. 컴퓨터에서 실수 연산은 어느 정도의 오차가 발생한다는 것을 유념해 두자!

4. Python bool/bytes/숫자형

- 복소수형 상수는 실수부와 허수부로 표현한다. 허수부에는 j나 J를 숫자 뒤에 붙여야 한다. 실수 부와 허수 부 각각은 실수형 상수로 표현한다.

```
>>> c = 4 + 5j
>>> d = 7 - 2j
>>> print( c * d )
>>> c.real           # 복소수의 실수부만 취함
>>> c.imag           # 복소수의 허수부만 취함
>>> a = 3; b = 4; complex( a, b )   # 일반 실수나 정수로 복소수 생성
>>> c.conjugate()      # 켤레 복소수
```

- cmath 모듈을 사용하면 다양한 함수에서 복소수 연산을 할 수 있다.

```
>>> import cmath
>>> cmath.sin( 0.1 + 0.2j )
>>> cmath.sqrt( -2 ); cmath.log( 2j)
```


4. Python bool/bytes/숫자형

◆ 숫자형에 사용하는 연산자

- 파이썬 기본 산술 연산자: +(덧셈), -(뺄셈), *(곱셈), /(나눗셈), //(몫), **(지수), %(나머지)

```
>>> 5 / 2
```

```
>>> 5 // 2
```

```
>>> 5 % 2
```

```
>>> 5 % -3
```

나머지는 제수의 부호와 같다

```
>>> -5 % 3
```

```
>>> divmod( 5, 2 )
```

몫과 나머지를 한 번에 계산

```
>>> 2 ** 3
```

```
>>> 2 ** 3 ** 2
```

2 ** (3 ** 2)와 같다

```
>>> 5 ** -2.0
```

음수의 지수형 연산

4. Python bool/bytes/숫자형

➤ 산술 연산자의 우선순위

산술 연산자	설명	결합 순서
+ -	단항 연산자, 부호	오른쪽에서 왼쪽으로(<-)
**	지수	오른쪽에서 왼쪽으로(<-)
* / % //	곱하기, 나누기, 나머지, 몫	왼쪽에서 오른쪽으로(->)
+ -	더하기, 빼기	왼쪽에서 오른쪽으로(->)

```
>>> 2 + 3 * 4
```

```
>>> ( 2 + 3 ) * 4
```

```
>>> ++3 # +(3)
```

```
>>> -+3 # -(3)
```

```
>>> 2 ** 3 ** 4
```

```
>>> ( 2 ** 3 ) ** 4
```

4. Python bool/bytes/숫자형

➤ 관계 연산자

관계 연산자	설명
>	큰지 비교
<	작은지 비교
>=	크거나 같은지 비교
<=	작거나 같은지 비교
==	같은지 비교
!=	같지 않은지 비교

```
>>> 6 == 9; 6 != 9; 1 > 3; 4 <= 5
```

```
>>> a = 5; b = 10; a < b; 0 < a < b; 0 < a and a < b
```

```
>>> 'abcd' > 'abd' # 문자열 비교는 사전 순
```

```
>>> [ 1, 2, 4 ] < [ 2, 1, 0 ] # list/tuple은 앞에서부터 하나씩 비교
```

```
>>> 123 < 'abd' # 다른 자료형과의 비교는 불가
```

```
>>> x = [ 1, 2, 3 ]; y = [ 1, 2, 3 ]; z = y
```

```
>>> x == y; x == z; x is y; y is z # 값 비교 ==, 참조 비교 is 사용
```

4. Python bool/bytes/숫자형

➤ 논리 연산자

논리 연산자 (우선순위순)	설명
not x	x가 거짓이면 True, 아니면 False
x and y	x가 거짓이면 x이고, 아니면 y, 왼쪽부터 식을 계산하다 결과가 알려지면 계산 중단하고 해당 시점의 객체를 반환. 반환 값은 참이나 거짓이 아니다. 최종적으로 기여한 객체를 결과로 반환
x or y	x가 참이면 x이고, 아니면 y, 왼쪽부터 식을 계산하다 결과가 알려지면 계산 중단하고 해당 시점의 객체를 반환. 반환 값은 참이나 거짓이 아니다. 최종적으로 기여한 객체를 결과로 반환

➤ 파이썬은 참이면 True, 거짓이면 False이며, 내부적으로 True는 1, False는 0의 값을 가진다.

```
>>> a = 10; b = 30; a > 10 and b < 50
```

```
>>> True + 1; False + 1
```

➤ 정수의 0, 실수의 0.0, 다른 자료형은 빈 객체((), {}, [])가 거짓이며 나머지는 참으로 간주, "None"은 특별한 객체로 '아무것도 없다' 또는 '아무것도 아니다'를 표현하는 파이썬 내장 객체이며 언제나 거짓이다.

4. Python bool/bytes/숫자형

```
>>> bool( 0 ); bool( 1 ); bool( 100 )
>>> bool( 0.0 ); bool( 0.1 )
>>> bool( 'abc' ); bool( '' )
>>> bool( [] ); bool( [ 1, 2, 3 ] )
>>> bool( () ); bool( ( 1, 2, 3 ) )
>>> bool( {} ); bool( { 1 : 2 } )
>>> bool( None )
>>> 1 and 1; 1 and 0; 0 or 0; 1 or 0
>>> [] or 1; [] or (); [] and 1
>>> [[]] or 1                                     # [[]]은 참으로 간주
>>> [{}] or 1
>>> bool_list = [ True, True, False ]
>>> all( bool_list )                               # 모두가 참인 경우 True
>>> any( bool_list )                               # 하나라도 참인 경우 True
>>> L = [ 1, 3, 2, 5, 6, 7 ]
>>> all( e < 10 for e in L )                       # 모든 요소가 10 미만인가?
>>> any( e < 5 for e in L )                       # 한 요소라도 5 미만인가?
```

4. Python bool/bytes/숫자형

➤ 비트 연산자

논리 연산자 (우선순위순)	설명
~	비트를 반전(1의 보수) 시킨다.
<< >>	비트를 왼쪽으로 이동시키거나, 오른쪽으로 이동시킨다.
&	비트 단위 AND 연산을 수행
^	비트 단위 XOR 연산을 수행
	비트 단위 OR 연산을 수행

```
>>> ~5; ~-1
```

```
>>> a = 3; a << 2; 1 << 128
```

```
>>> a = 4; a >> 1
```

왼쪽이 0으로 채워진다.

```
>>> a = -4; a >> 1
```

왼쪽이 1로 채워진다.

```
>>> a & 2; a | 8; 0x0f ^ 0x06
```

```
>>> a = 2; a & 0x0f
```

a의 마지막 4비트만 그대로, Masking

```
>>> a | 0x0f
```

a의 마지막 4비트를 모두 1로

```
>>> a ^ -1
```

a의 모든 비트 반전

4. Python bool/bytes/숫자형

➤ 연산자 우선순위

연산자	설명
{ key:value... }	사전
[expressions...]	리스트
(expressions...)	튜플
f(arguments...)	함수 호출
x[index:index]	슬라이싱
x[index]	인덱싱
x.attribute	속성 참조
**	지수
~	비트 단위 NOT(1의 보수)
+ -	양수, 음수(단항 연산자)
* / // %	곱하기, 나누기, 몫, 나머지
+ -	더하기, 빼기(이항 연산자)
<< >>	시프트 연산자

연산자	설명
&	비트 단위 AND
^	비트 단위 XOR
	비트 단위 OR
< <= > >= != ==	크기 비교
is, not is in, not in	Identity 확인 멤버 검사
not	논리 연산 not
and	논리 연산 and
or	논리 연산 or
lambda	람다 표현식

4. Python bool/bytes/숫자형

➤ 내장 수치 연산 함수

함수	설명
<code>abs(x)</code>	x의 절대값
<code>int(x)</code>	x를 정수형으로 변환
<code>float(x)</code>	x를 실수형으로 변환
<code>complex(x)</code>	실수부 re와 허수부 im을 가지는 복소수를 구함
<code>c.conjugate()</code>	복소수 c의 켤레 복소수를 구함
<code>divmod(x, y)</code>	(<code>x // y</code> , <code>x % y</code>) 쌍을 구함
<code>pow(x, y)</code>	x의 y승을 구함
<code>max(iterable), min(iterable)</code>	최대값과 최소값을 구함
<code>sum(iterable)</code>	합을 계산

- 내장 수치 연산 함수 이외에 표준 모듈로 만든 `math`(실수 연산), `cmath`(복소수연산)을 사용할 수 있으며, 해당 모듈 사용방법은 다음과 같다.

```
>>> import math                # 모듈을 사용하기 위해 추가
>>> print( math.pi )          # 모듈이름.함수 or 모듈이름.변수
```


5. Python 문자열

◆ Sequence 자료형

- 여러 객체를 저장하는 자료형이며, 각 객체의 순서를 가진다.
- 각 요소는 index를 사용하여 참조할 수 있다.
- Sequence 자료형에 속하는 객체는 문자열, 리스트, 튜플이 있다.
- Sequence 자료형의 특성은 내장 자료형 뿐만 아니라 Sequence 클래스 객체들이 가져야 할 특성이다.

```
>>> s = 'Pythen'
```

```
>>> L = [ 100, 200, 300 ]
```

```
>>> t = ( '튜플', '객체', 1, 2 )
```

```
# 문자열, ' 나 " 로 묶인 문자들 모임
```

```
# 리스트, []안에 둘러싸인 객체들 모임
```

```
# 튜플, ()안에 둘러싸인 객체들의 모임
```

5. Python 문자열

◆ Sequence 자료형의 공통적인 연산

구분	연산	설명
인덱싱	[k] 형식	k번 위치의 값 하나를 취한다.
슬라이싱	[s : t : p] 형식	s부터 t사이 구간의 값을 p 간격으로 취한다.
연결하기	+ 연산자	두 시퀀스형 데이터를 붙여서 새로운 데이터를 만든다.
반복하기	• 연산자	시퀀스형 데이터를 여러 번 반복해서 새로운 데이터를 만든다.
멤버 검사	in 연산자	어떤 값이 시퀀스 자료형에 속하는지를 검사한다.
길이 정보	len() 내장함수	시퀀스형 데이터의 크기를 나타낸다.

- 인덱싱(indexing)은 순서가 있는 데이터에서 index로 하나의 객체를 참조하는 것이며, index는 정수이며 0부터 시작한다.

```
>>> s = 'abcdef'; l = [ 100, 200, 300 ]
>>> s[0]; s[1]
>>> s[-1] # 가장 오른쪽 값
>>> l[1]
>>> l[1] = 900
```

5. Python 문자열

- 슬라이싱(Slicing)은 시퀀스 자료형의 일정 영역에서 새로운 객체를 반환하며, 결과의 자료형은 원래의 자료형과 같다.

[시작오프셋:끝오프셋]

- 오프셋은 생략할 수 있으며, 시작 오프셋을 생략하면 0, 끝 오프셋을 생략하면 마지막 값으로 처리한다.

```
>>> s = 'abcdef'; l = [ 100, 200, 300 ]
>>> s[ 1:3 ]                # 1번위치와 3번위치 사이, 3번위치는 제외
>>> s[ 1: ]
>>> s[ : ]                  # 처음(0)부터 끝까지
>>> s[ -100:100 ]           # 범위를 넘어서면 범위 내 값 자동으로 처리
>>> l[ :-1 ]                # 맨 오른쪽 값을 제외하고 모두
>>> s[ ::2 ]                 # 2칸 단위로 데이터를 취한다. 세번째 step은
                             # 데이터 간격을 뜻하며, 확장 슬라이싱이라 한다.
>>> s[ ::-1 ]               # 반대로 데이터를 취한다.
```

5. Python 문자열

- 연결하기(Concatenation)는 + 연산자로 같은 시퀀스 자료형인 두 객체를 연결하는 것이며, 두 객체의 자료형이 동일해야 하며, 새로 만들어지는 객체도 같은 자료형이다.

```
>>> s = 'abc' + 'def'; s
>>> l = [ 1, 2, 3 ] + [ 4, 5, 6 ]; l
```

- 반복하기(Repetition)는 * 연산자를 사용하여 시퀀스 자료형인 객체를 여러 번 반복해서 연결하는 것이다. 새로운 객체를 반환한다.

```
>>> s = 'Abc'
>>> s * 4
>>> l = [ 1, 2, 3 ]
>>> l * 2
```

5. Python 문자열

◆ 문자열 개요

- 문자열은 시퀀스 자료형이며, 시퀀스 자료형의 특성을 모두 가진다.
- 파이썬 3의 문자열은 유니코드를 사용한다.

```
>>> s = ''                                # 빈 문자열
>>> str1 = 'Python is great!'
>>> str2 = "Yes, it is."
>>> str3 = "It's not like any other languages"
>>> type( s )
>>> isinstance( s, str )
>>> str4 = 'Don\'t walk. "Run"'
>>> print( str4 )
>>> long_str = "This is a rather long string \
containing back slash and new line.\nGood!"
>>> print( long_str )
```

5. Python 문자열

◆ 여러 줄 문자열

- 파이썬에서 ' 나 " 를 세 번 연속해서 사용하여 여러 줄 문자열을 손쉽게 표현한다. 표현식내의 모든 텍스트를 적힌 그대로 표현할 수 있다.

```
>>> multiline = """Python is great!\n Yes, it is."""  
>>> print( multiline )
```

➤ 특수 문자

문자	설명	문자	설명
\<Enter>	다음 줄과 연속임을 표현	\n 또는 \012	줄 바꾸기
\\	\ 문자 자체	\t	<Tab> 키
\'	'문자	\0xx	8 진수 코드 xx
\"	"문자	\xXX	16 진수 코드 XX
\b	백스페이스	\e	<Esc> 키

5. Python 문자열

◆ 문자열 연산

➤ 문자열은 그 자체 값을 변경할 수 없는 변경 불가능 자료형이다.

```
>>> s1 = 'first'
>>> s2 = 'second'
>>> s3 = s1 + ' ' + s2; s3
>>> s1 * 3
>>> s1[2]
>>> s1[ 1:-1 ]
>>> len( s1 )
>>> 'fi' in s1
>>> s1[0] = 'f'                                     # error
>>> s = 'spam and egg'; s = s[ :5 ] + 'cheese ' + s[ 5: ]; s
    -> 슬라이싱을 이용한 새로운 문자열 정의
```

5. Python 문자열

◆ 문자열 서식 지정

- print()를 사용하면 출력 문자열의 서식을 자유롭게 지정할 수 있다. 서식(format)이란 문서의 틀이나 양식을 말한다.
- 문서는 양식과 내용으로 구성된다. 양식은 고정된 틀을 의미하고 빈칸을 갖는다. 문서 양식을 만들어 놓고 빈칸에 필요한 내용을 채워서 문서를 자유롭게 만들어 내는데 이때 서식 지정(Formatting)을 사용한다.

```
>>> int_val = 23; float_val = 2.34567
>>> print( "%3d %s %0.2f" % ( int_val, 'any value', float_val ) ) # 비권장
```

● format()를 사용한 서식 지정

```
>>> print( format( int_val, '3d' ), 'any value', format( float_val, '.2f' ) )
# 권장
```

```
>>> print( format( 123456789, ',d' ) )
>>> print( format( 1234567.89, ',.2f' ) )
```


5. Python 문자열

- format() 메서드를 사용한 서식 지정

여러 값을 출력할 때 가장 많이 사용하는 방식이다. 문자열 내 {}은 데이터로 채워질 자리를 의미한다. 공급 되는 순서대로 데이터를 출력한다.

{index}는 format() 메서드의 해당 index 인수로 치환되며 참조 순서는 바뀔 수 있고 여러 번 나타날 수도 있다.

```
>>> '{} {}'.format( 23, 2.12345 )          # {} 데이터가 채워질 위치
>>> l = [ 1, 5, 3, 7, 4, 5 ]
>>> '최대값:{0}, 최소값:{1}'.format( max( l ), min( l ) ) # 0:첫 번째 1:두 번째
>>> 'sqrt( {:3} ) = {:.0.5}'.format( 2, 2 ** 0.5 )    # 자릿수만 지정
>>> 'sqrt( {0:3} ) = {1:0.5}'.format( 2, 2 ** 0.5 )
>>> 'sqrt( {0:3d} ) = {1:0.5f}'.format( 2, 2 ** 0.5 ) # d:decimal f:float
>>> l = [ 0, 1, 1, 2, 3, 5, 8, 13, 21 ]
>>> 'next value of {0[4]} is {0[5]}'.format( l ) # 리스트 인덱싱
>>> '나이:{age}, 키:{height}'.format( age = 20, height = 173 ) # 키워드 인수
>>> info = { 'size':32, 'height':173, 'age':49 } # 사전이 전달시 format_map()
>>> '나이:{age}, 키:{height}'.format_map( info ) # 메서드를 사용 키로 참조
```

5. Python 문자열

◆ 문자열 메서드

```
>>> s = 'i like programming. i like swimming'
>>> s.upper()                # 대문자로 변환
>>> s.lower()                # 소문자로 변환
>>> 'I like programming.'.swapcase()
>>> s.capitalize()          # 첫 문자를 대문자로 변환
>>> s.title()
>>> s.count( 'like' )        # 부분 문자 발생 횟수
>>> s.find( 'like' )         # 검색
>>> s.find( 'like', 3 )
>>> s.find( 'my' )
>>> s.rfind( 'like' )        # 역순 검색
>>> s.index( 'like' )
>>> s.index( 'my' )          # find()와 동일, 없을 경우 예외 발생
>>> s.rindex( 'like' )
>>> s.startswith( 'i like' ) # 'i like'로 시작하는 문자열인가?
```

5. Python 문자열

>>> s.endswith('swimming')	# swimming으로 끝나는 문자열인가?
>>> u = ' spam and ham '	
>>> u.strip()	# 좌우 공백 제거
>>> u.rstrip()	# 오른쪽 공백 제거
>>> u.lstrip()	# 왼쪽 공백 제거
>>> u.split()	# 공백으로 분리
>>> u.split('and')	# and로 분리
>>> t = u.split()	
>>> ':'.join(t);	# t는 리스트 :는 연결할 문자
>>> print('\n'.join(t))	# t는 리스트 줄바꾸기로 결합
>>> u.center(60)	# 전체 60문자의 가운데에 맞춤
>>> u.ljust(60)	
>>> u.rjust(60)	
>>> u.center(60, '-')	# 공백대신 - 문자로 채운다.
>>> '1\tand\t2'.expandtabs(4)	# \t을 4자 공백으로 변경
>>> ord('A')	# 문자의 코드 값
>>> chr(0x0041)	# 코드 값을 문자로 변환

5. Python 문자열

```
>>> '1234'.isdigit()           # 숫자인가?
>>> '123\u2155\u2156'.isnumeric() # 유니코드 수치 또는 수치 문자인가?
>>> '123\u0661'.isdecimal()      # 유니코드 수치 또는 수치 문자인가?
>>> 'abc한글'.isalpha()          # 영문자인가?
>>> '1abc234'.isalnum()          # 숫자나 영문자인가?
>>> 'abc'.islower()              # 소문자인가?
>>> 'ABC'.isupper()              # 대문자인가?
>>> ' \t\r\n'.isspace()          # 공백 문자인가?
>>> 'This Is A Title'.istitle()  # 제목 문자열인가?
>>> 'a-b'.isidentifier()          # 문자열이 유효한 식별자인가?
>>> ' \n\t'.isprinttable()        # 인쇄 가능한 문자들의 모임인가?
>>> s.zfill( 40 )                 # 40칸 중 앞 빈 자리는 0으로 채워진다.

>>> instr = 'abcdef'; outstr = '123456'
>>> trantab = ''.maketrans( instr, outstr )    # 문자를 mapping하여 변환한 결과
>>> 'as soon as possible'.translate( trantab ) # 를 얻을수 있다
```

6. Python 리스트

◆ 리스트 개요

- 리스트는 순서를 가지는 객체들의 집합으로, 파이썬 자료형들 중에서 가장 유용하게 활용된다.
- 리스트는 시퀀스 자료형이며 변경 가능 자료형이다.
- 시퀀스 자료형의 특성을 모두 가지며, 데이터의 크기를 동적으로 그리고 임의로 조절하거나, 내용을 치환하여 변경할 수 있다.
- 리스트는 []로 표현한다.

```
>>> a = []                                # 빈 리스트
>>> a = [ 1, 2, "Great" ]
>>> print( a[ 0 ], a[ -1 ] )              # 인덱싱
>>> print( a[ 1:3 ], a[:] )               # 슬라이싱
>>> l = list( range( 10 ) )
>>> l[ ::2 ]                             # 확장 슬라이싱
>>> a * 2                                 # 반복하기
>>> a + [ 3, 4, 5 ]                       # 연결하기
>>> Len( a ); 4 in l                      # 리스트 길이 정보, 멤버 검사
```

6. Python 리스트

- 리스트의 일부 값 변경

```
>>> a = [ 'spam', 'eggs', 100, 1234 ]  
>>> a[ 2 ] = a [ 2 ] + 23; a
```

- 리스트의 일부 값 치환

```
>>> a[ 0:2 ] = [ 1, 12 ]; a          # 항목 두개 교체  
>>> a[ 0:2 ] = [ 1 ]; a             # 크기가 달라도 된다
```

- 리스트의 일부 값 삭제

```
>>> a = [ 1, 12, 123, 1234 ]  
>>> a[ 0:2 ] = []; a                # 항목 두 개 삭제
```

- 리스트에서 일부 값 추가

```
>>> a = [ 123, 1234 ]  
>>> a[ 1:1 ] = [ 'spam', 'ham' ]; a  # 항목 추가
```

6. Python 리스트

- 확장 슬라이싱이 오른쪽에 오는 경우 오른쪽과 왼쪽 요소의 개수가 같아야 한다.

```
>>> a = list( range( 4 ) ); a
>>> a[ ::2 ]
>>> a[ ::2 ] = list( range( -10, -12, -1 ) ); a
>>> a[ ::2 ] = range( 3 )           # error
```

- del 문을 이용해서 값 삭제

```
>>> a = [ 1, 2, 3, 4 ]
>>> del a[0]           # 값 삭제
>>> del a[ 1: ]         # 값 일부 삭제
```

- 확장 슬라이싱을 이용하여 값을 삭제

```
>>> a = list( range( 4 ) )
>>> a[ ::2 ]
>>> del a[ ::2 ]
```

6. Python 리스트

◆ 중첩 리스트

- 리스트 안에 또 다른 리스트가 포함되어 있는 경우 중첩 리스트(Nested Lists) 라고 한다.
- 리스트는 다른 객체를 직접 저장하지 않고, 객체의 참조만을 저장한다. 참조란 객체의 주소를 의미한다.

```
>>> s = [ 1, 2, 3 ]
>>> t = [ 'begin', s, 'end' ]; t           # 중첩 리스트
>>> t[1][1]
>>> s[1] = 100; t
>>> l = [ 1, [ 'a', [ 'x', 'y' ], 'b' ], 3 ]   # 중첩 리스트
>>> l[0]                                     # 첫 번째 항목
>>> l[1]                                     # 두 번째 항목
>>> l[1][1]                                  # 두 번째의 두 번째 항목
>>> l[1][1][1]                               # 두 번째의 두 번째의 두 번째 항목
```


6. Python 리스트

◆ 리스트 메서드

메서드	설명
append	데이터를 리스트 끝에 추가(스택의 push) 한다.
insert	데이터를 지정한 위치에 삽입한다.
index	요소를 검색한다.
count	요소의 개수를 알아낸다.
sort	리스트를 정렬한다.
reverse	리스트의 순서를 바꾼다.
remove	리스트의 지정한 값 하나를 삭제한다.
pop	리스트의 지정한 값 하나를 읽어 내고 삭제(스택의 Pop) 한다.
extend	리스트를 추가한다.

6. Python 리스트

```
>>> s = [ 1, 2, 3 ]
>>> s.append( 5 ); s          # 리스트 마지막에 추가
>>> s.insert( 3, 4 ); s      # 3위치에 4를 삽입
>>> s.index( 3 )             # 3의 위치는?
>>> s.count( 2 )             # 2의 개수는?
>>> s.reverse(); s          # 리스트 순서를 반대로
>>> s.sort(); s              # 리스트 내용 정렬
>>> s.sort( reverse = True ); s
>>> l = 'Python is a Programming Language'.split(); l
>>> l.sort(); l; l.sort( key = str.lower ); l      # 대소문자 무시하고 정렬
>>> l = [ 1, 6, 3, 8, 6, 2, 9 ]
>>> newList = l.sort(); print( l ); print( newList )
>>> l = [ 1, 6, 3, 8, 6, 2, 9 ]
>>> newList = sorted( l ); print( l ); print( newList )
>>> s = [ 10, 20, 30, 40, 50 ]
>>> s.remove( 10 ); s        # 10을 삭제
>>> s.extend( [ 60, 70 ] )   # 리스트를 추가
```

6. Python 리스트

◆ 리스트 내장(List Comprehension)

- 시퀀스 자료형으로부터 리스트를 쉽게 만드는 데 유용하다.
- 순차적인 정수 리스트를 만들 때 range() 함수 사용

```
>>> range( 10 ); type( range( 10 ) )
```

```
>>> l = [ k * k for k in range( 10 ) ]
```

```
>>> print( l )
```

❖ k * k : 출력 서식

❖ range(10) : 입력 시퀀스

```
>>> l = [ k * k for k in range( 10 ) if k % 2 == 1 ]
```

```
>>> print( l )
```

❖ if k % 2 == 1 : 조건부 서식

```
>>> seq1 = 'abc'
```

```
>>> seq2 = ( 1, 2, 3 )
```

```
>>> [ ( x, y ) for x in seq1 for y in seq2 ]
```

6. Python 리스트

◆ 리스트 내장 형식

```
[ <식> for <타겟1> in <객체1>
```

```
...
```

```
for <타겟n> in <객체n>
```

```
( if <조건식> ) ]
```

- ❖ for~in 절은 <객체>의 항목을 반복한다.
- ❖ <객체>는 시퀀스형 데이터이어야 한다.
- ❖ for문으로 취해지는 각각의 값은 <식>에서 사용한다.
- ❖ 마지막 if 절은 선택 요소이다. 만일 if 절이 있으면, <식>은 <조건식>이 참일 때만 값을 계산하고 결과에 추가한다.

7. Python 튜플

◆ 튜플 개요

- 튜플(tuple)은 임의의 객체들이 순서를 가지는 모음으로 리스트와 유사한 면이 많다. 차이점은 변경 불가능한 자료형이라는 것이다.
- 튜플은 리스트가 가지고 있는 것만큼 다양한 메서드가 없다.
- 튜플은 시퀀스 자료형이므로 시퀀스 자료형의 특성을 모두 가진다.
- 튜플은 ()로 표현한다.

```
>>> t = ()                                # 빈 튜플
>>> t = ( 1, 2, 3 ); t
>>> t = 1, 2, 3; t                        # ()없이 ,로 데이터를 구분하면 튜플
>>> t = ( 1, )                            # 데이터가 한 개일 때도 ,는 반드시 존재
>>> t = 1,                                # 데이터가 한 개일 때도 ,는 반드시 존재
```

7. Python 튜플

```
>>> t = ( 1, 2, 3 )
>>> t * 2                # 반복하기
>>> t + ( 'apple', 'banana' ) # 연결하기
>>> print( t[ 0 ], t[ 1:3 ] ) # 인덱싱과 슬라이싱
>>> len( t )             # 길이 정보
>>> 1 in t               # 멤버 검사
>>> t[0] = 100           # error
```

- 튜플은 검색에 관련된 메서드 두 개를 제공

```
>>> t.count( 2 )          # 2가 몇 개 있는가?
>>> t.index( 2 )          # 첫 번째 2의 위치는?
>>> t.index( 2, 1 )       # 1 위치부터 검색
```

- 튜플 중첩

```
>>> t = ( 12345, 54321, 'hello' )
>>> u = t, ( 1, 2, 3, 4, 5 ); u
```

7. Python 튜플

- 튜플을 이용하여 좌우 변에 복수 개의 데이터 치환 가능

```
>>> x, y, z = 1, 2, 3
```

```
>>> print( x, y, z )
```

```
>>> ( x1, y1 ), ( x2, y2 ) = ( 1, 2 ), ( 3, 4 )
```

```
>>> print( x1, y1, x2, y2 )
```

- 튜플을 이용하여 두 개의 값도 쉽게 바꿀 수 있다.

```
>>> x, y = 1, 2
```

```
>>> x, y = y, x
```

```
>>> x, y
```

7. Python 튜플

◆ 패킹과 언패킹

- 한 데이터에 여러 개의 데이터를 넣는 것을 패킹(Packing)이라 한다.
- 패킹과 반대로 한 데이터에서 데이터를 각각 꺼내 오는 것을 언패킹(Unpacking)이라 한다.

```
>>> t = 1, 2, 'hello'           # 패킹
>>> x, y, z = t                 # 언패킹
>>> l = [ 'foo', 'bar', 4, 5 ]  # 패킹(리스트)
>>> x, y, z, w = l              # 언패킹(리스트)
```

● 확장된 언패킹

```
>>> t = ( 1, 2, 3, 4, 5 )
>>> a, *b = t; print( a, b )
>>> *a, b = t; print( a, b )
>>> a, b, *c = t; print( a, b, c )
```

❖ *a와 같은 식의 표현은 나머지 전부를 의미한다.

7. Python 튜플

◆ 리스트와의 차이점

- 리스트와의 공통점은 임의의 객체를 저장할 수 있다는 것과 시퀀스 자료형이라는 것이다.
- 리스트와의 차이점은 변경 불가능한 시퀀스 자료형이고, 함수의 가변 인수를 지원한다.
- 변경해야 할 데이터들은 리스트에, 변경하지 말아야 할 데이터는 튜플에 저장한다.
- 리스트와 튜플은 `list()`와 `tuple()` 내장 함수를 사용하여 상호 변환할 수 있다.

```
>>> t = ( 1, 2, 3, 4, 5 )
>>> l = list( t )
>>> l[0] = 100
>>> l
>>> t = tuple( l )
>>> t
```

7. Python 튜플

◆ 튜플의 특별한 활용

1. 함수에 있어서 하나 이상의 값을 반환할 때 활용

```
>>> def calc( a, b ):
    return a + b, a * b           # return앞은 반드시 tab으로 띄기
>>> x, y = calc( 5, 4 ); print( x, y )
```

2. 튜플에 있는 값들을 함수의 인수로 사용할 때 활용

```
>>> args = ( 4, 5 )
>>> calc( *args )
```

3. 파이썬 2 형식의 서식 문자열에 데이터를 공급할 때 활용

```
>>> "%d %f %s" % ( 12, 3.456, 'hello' )
```

7. Python 튜플

◆ 이름 있는 튜플

- 이름 있는 튜플은 튜플에 인덱스는 물론 이름으로도 접근할 수 있도록 관련 기능을 추가한 것이다.
- 모듈 Collections의 namedtuple() 함수로 객체를 생성한다.

```
namedtuple( typename, field_names, verbose = false, rename = false )
```

```
>>> from collections import namedtuple
>>> Point = namedtuple( 'Point', 'x, y' )
>>> Point; Point.__name__
>>> pt1 = Point( 1.0, 5.0 ); pt2 = Point( 2.5, 1.5 ); pt1
>>> from math import sqrt
>>> length = sqrt( ( pt1.x - pt2.x ) ** 2 + ( pt1.y - pt2.y ) ** 2 )
>>> length
>>> length = sqrt( ( pt1[0] - pt2[0] ) ** 2 + ( pt1[1] - pt2[1] ) ** 2 )
>>> length
```

8. Python 집합

◆ 집합 개요

- 집합(Set)은 여러 값을 순서 없이 그리고 중복 없이 모아 놓은 자료형이다.
- 파이썬에서는 set과 frozenset 두 가지 집합 자료형을 제공한다.
- set은 변경 가능한 집합이고 frozenset은 변경 불가능한 집합이다.

● 집합 생성

```
>>> a = set()                # 빈 set 생성
>>> b = { 1, 2, 3 }          # {}를 이용하여 set 객체 생성
>>> a; b
>>> type( a ); type( b )
>>> type( a ) == type( b )
>>> b = a.copy()
>>> a; b
>>> b = { 1, 2, 3 }
>>> a = b.copy()
>>> a; b
```

8. Python 집합

- 반복 가능한(iterable) 객체로부터 집합을 생성할 수 있다.

```
>>> set( ( 1, 2, 3 ) )           # 튜플로부터 집합 생성
>>> set( 'abcd' )                 # 문자열로부터 집합 생성
>>> set( [ 1, 2, 3 ] )           # 리스트로부터 집합 생성
>>> set( ( 1, 2, 3, 1, 2, 3, 1, 2, 3 ) ) # 중복된 원소는 한 번만 표현
>>> set( { 'one':1, 'two':2 } )   # 사전의 반복자는 키 값을 반환
```

- 모든 데이터가 집합의 원소로 사용할 수 있는 것은 아니고 해시 기능(Hashable)이면서 변경 불가능한 자료형만이 집합의 원소로 사용할 수 있다.

```
>>> a = [ 1, 2, 3 ]
>>> b = [ 3, 4, 5 ]
>>> { a, b }                      # error, 리스트는 집합의 원소로 불가
```

8. Python 집합

◆ set 객체의 연산

- 원소 추가 : set 객체에 원소를 추가하는 메서드는 add()와 update()가 있다.
- add() 메서드는 한 원소를 추가하고, update() 메서드는 주어진 객체에 대한 합집합 연산을 한다.
- copy() 메서드를 사용하면 set 객체를 통째로 복사할 수 있다.

```
>>> a = { 1, 2, 3 }
>>> len( a )
>>> a.add( 4 ); a
>>> a.update( [ 4, 5, 6 ] ); a
>>> b = { 6, 7, 8 }
>>> a.update( b ); a
>>> a = { 1, 2, 3 }
>>> a.update( { 4, 5, 6 }, { 7, 8, 9 } ); a
>>> a.copy()
```

8. Python 집합

- 원소 제거 : set 객체에서 원소를 제거하는 메서드는 `clear()`와 `discard()`, `pop()`, `remove()`등이 있다.

```
>>> a = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
>>> a.clear(); a                                # 전체 원소 제거
>>> a = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
>>> a.discard( 3 ); a                            # 원소 한 개 제거
>>> a.discard( 3 ); a                            # 해당 원소가 없으면 그냥 통과
>>> a.remove( 4 ); a                             # 원소 한 개 제거
>>> a.remove( 4 )                                # 해당 원소가 없으면 예외 발생
>>> a.pop()                                       # 원소 한 개를 집합에서 제거하면서
>>> a.pop()                                       # 반환
>>> a
```

8. Python 집합

- 집합 연산 : union(합집합), intersection(교집합), difference(차집합), symmetric_difference(대칭 차집합) 연산이 있다.

```
>>> a = { 1, 2, 3, 4, 5, 6 }
>>> b = { 4, 5, 6, 7, 8, 9 }
>>> c = { 4, 10 }
>>> a.union( b )                # 합집합
>>> a.intersection( b )        # 교집합
>>> a.difference( b )          # 차집합
>>> a.symmetric_difference( b ) # 대칭 차집합
>>> a; a.update( b ); a        # 합집합 결과 a에 저장
>>> a = { 1, 2, 3, 4, 5, 6 }
>>> a.intersection_update( b ); a # 교집합 결과 a에 저장
>>> a = { 1, 2, 3, 4, 5, 6 }
>>> a.difference_update( b ); a # 차집합 결과 a에 저장
>>> a = { 1, 2, 3, 4, 5, 6 }
>>> a.symmetric_difference_update( b ); a # 대칭 차집합 결과 a에 저장
```


8. Python 집합

```
>>> a = { 1, 2, 3, 4, 5, 6 }
>>> 2 in a                      # 멤버 검사
>>> 2 not in a                  # 멤버 검사
>>> a = { 1, 2, 3, 4, 5 }
>>> b = { 1, 2, 3 }
>>> a.issuperset( b )           # a  $\supset$  b
>>> b.issubset( a )             # b  $\subset$  a
>>> a.isdisjoint( b )           # 교집합이 공집합인가?
```

8. Python 집합

- 집합 자료형은 순서가 없는 자료형이므로 인덱싱, 슬라이싱, 정렬을 지원하지 않는다. 하지만, 다른 시퀀스 자료형으로 형변환을 하면 부분 원소를 참조할 수 있다.

```
>>> a = { 1, 2, 3, 4, 5 }
>>> a[ 0 ]                                # error
>>> b = list( a )                         # 리스트로 변환
>>> b[ 0 ]
>>> c = tuple( a )
>>> c[ 0 ]
```

8. Python 집합

- frozenset 객체는 변경 불가능한 집합 자료형이다. 값을 변경하지 않는 범위에서 set 객체와 동일하게 동작한다.
- frozenset 객체의 생성은 집합을 포함한 반복 가능한 자료형으로부터 가능하다.

```
>>> frozenset( [ 1, 2, 3, 4, 5 ] )      # 반복이 가능한 객체로부터 생성한다.  
>>> a = { 1, 2, 3, 4, 5 }  
>>> frozenset( a )                      # set 객체로부터 생성한다.
```

- frozenset 객체는 값을 변경하지 않는 연산만 허용하며 set 객체와 동일하게 동작

```
>>> a = frozenset( ( 1, 2, 3, 4, 5, 6 ) )  
>>> b = frozenset( ( 4, 5, 6, 7, 8, 9 ) )  
>>> a.union( b )                        # 합집합  
>>> a.intersection( b )                # 교집합  
>>> a.difference( b )                   # 차집합  
>>> a.symmetric_difference( b )        # 대칭 차집합
```

8. Python 집합

```
>>> a.copy()
>>> a = frozenset( { 1, 2, 3, 4, 5 } )
>>> b = frozenset( { 1, 2, 3 } )
>>> a.issuperset( b )           # a  $\supset$  b
>>> b.issubset( a )            # b  $\subset$  a
>>> a.isdisjoint( b )          # 교집합이 공집합인가?
```

- 집합 내장 : {}를 이용하면 리스트 내장과 같이 for 문을 통해서 집합을 직접 만들 수 있다.

```
>>> { v * v for v in [ 1, 2, 3, 4 ] }    # 연산 결과가 set 객체로 모인다.
>>> { v for v in 'python' if v not in 'aeiou' }
```

9. Python 사전

◆ 사전 개요

- 사전(Dictionary)은 특정 키를 주면 이와 관련된 값을 돌려주는 내용 기반으로 검색하는 자료형이다.
- 사전은 임의 객체의 집합적 자료형인데, 데이터의 저장 순서가 없다. 집합적이라는 의미에서 리스트와 튜플과 동일하다. 하지만, 데이터의 순서를 정할 수 없는 매핑형이다.
- 시퀀스 자료형은 데이터의 순서를 정할 수 있어서 정수 오프셋에 의한 인덱싱이 가능하지만, 매핑형에서는 키(Key)를 이용한 값(Value)에 접근한다.

```
>>> member = { 'basketball':5, 'soccer':11, 'baseball':9 }  
>>> member[ 'baseball' ]           # 검색
```

- member라는 사전은 세 개의 입력 값을 가지고 있다. 각각의 입력 값은 키와 값으로 구분된다. 사전에서 값을 꺼내려면 키를 사용한다.

9. Python 사전

- 사전은 변경 가능한 자료형으로 값을 저장할 때도 키를 사용한다. 키가 사전에 등록되어 있지 않으면 새로운 항목으로 만들어지며, 키가 이미 사전에 등록되어 있으면 이 키에 해당하는 값이 변경된다.

```
>>> member[ 'volleyball' ] = 7          # 새 값을 설정
>>> member[ 'volleyball' ] = 6          # 값을 변경
>>> member
>>> len( member )                        # 길이 정보
>>> del member[ 'basketball' ]           # 항목 삭제
>>> 'soccer' in member                   # 멤버 검사
```

- 사전을 출력하면 어떤 순서에 의해서 입력 값들이 표현된다. 그러나 이 순서는 고정된 것이 아니다. 키에 의한 검색 속도를 빠르게 하기 위해 사전은 내부적으로 해시 기법을 이용하여 데이터를 저장한다. 이 기법은 데이터의 크기가 증가해도 빠른 속도로 데이터를 찾을 수 있게 해준다.

9. Python 사전

- 값은 임의의 객체가 될 수 있지만, 키는 해시 기능이라 변경 불가능한 자료형이어야 한다.

```
>>> d = {}                                # 사전 생성
    ❖ dict() 함수를 사용할 수 있다.
>>> d[ 'str' ] = 'abc'; d                 # 사전에 값 추가
>>> d[1] = 4; d                           # 사전에 값 추가
>>> d[ ( 1, 2, 3 ) ] = 'tuple'            # 사전에 tuple로 값 추가
>>> d[ [ 1, 2, 3 ] ] = 'list'             # error, 키가 변경 가능
>>> d[ { 1 : 2 } ] = 3                    # error, 키가 변경 가능
```

- 함수를 키나 값으로 활용할 수도 있다.

```
>>> def add( a, b ):
    return a + b
>>> def sub( a, b ):
    return a - b
>>> action = { 0:add, 1:sub }; action[0]( 4, 5 )
```

9. Python 사전

- 사전의 모든 값을 순차적으로 참조하는 데 사용하는 일반적인 방법은 사전의 반복자를 이용하는 것이다.

```
>>> d = { 'a':1, 'b':2, 'c':3 }
>>> for key in d:
    print( key, d[ key ] )
```

- items() 메서드를 사용하여 key와 value 항목을 함께 꺼내 올 수도 있다. items() 메서드는 (키, 값)의 사전 뷰를 반환한다.

```
>>> d.items()
>>> for key in d:
    print( key, d[ key ] )
```


9. Python 사전

- 뷰(View)란 사전 항목들을 동적으로 볼 수 있는 객체이다. 동적이란 의미는 사전의 내용이 바뀌어도 뷰를 통해서 내용의 변화를 읽어 낼 수 있다는 의미다.
- 사전의 뷰를 반환하는 메서드는 다음과 같다.

- `keys()` 메서드 키에 대한 사전 뷰를 반환
- `values()` 메서드 값에 대한 사전 뷰를 반환
- `items()` 메서드 항목(key, value)들의 사전 뷰를 반환

```
>>> d = { 'one':1, 'two':2, 'three':3 }  
>>> keys = d.keys(); keys                      # 키에 대한 사전 뷰를 얻는다.  
>>> d[ 'four' ] = 4; keys                      # 뷰도 동적으로 변경
```

9. Python 사전

- 뷰 객체는 인덱싱이 가능하지 않지만, 리스트로 변환하거나 멤버 검사가 가능하고 반복자를 제공한다.

```
>>> list( keys )           # 리스트로 변환
>>> len( keys )           # 항목 수
>>> 'one' in keys         # 멤버 검사
>>> iter( keys )          # 반복자 객체를 사용할 수 있다.
>>> for k in keys:
    print( k )
```

- values()와 items() 메서드도 keys() 메서드와 같은 연산이 가능하다.

```
>>> values = d.values(); values
>>> len( values ); 4 in values
>>> list( values )
>>> items = d.items(); items
>>> len( items )
```

9. Python 사전

- 키와 항목에 대한 사전 뷰들은 항목들이 유일하고 중복되지 않는다는 점에서 집합과 유사한 특성이 있다. 사전의 뷰 객체들은 집합 연산을 허용한다.

```
>>> keys
>>> keys & { 'one', 'two', 'five' }      # 교집합
>>> keys - { 'one', 'two', 'five' }      # 차집합
```

9. Python 사전

◆ 사건의 뷰 객체를 얻어내는 `keys()`와 `values()`, `items()` 이외의 메서드

메서드(d는 사전 객체)	설명
<code>d.clear()</code>	사전 d의 모든 항목을 삭제
<code>d.copy()</code>	사전을 복사한다. 얇은 복사에 해당한다.
<code>d.get(key [, x])</code>	값이 존재하면 <code>d[key]</code> 를 반환하고, 아니면 <code>x</code> 를 반환한다.
<code>d.setdefault(key [, x])</code>	<code>get()</code> 와 같으나 값이 존재하지 않을 때 값을 설정한다. <code>d[key] = x</code>
<code>d.update(b)</code>	사전 b의 모든 항목을 d에 갱신한다. <code>for k in b.keys(): d[k] b[k]</code>
<code>d.popitem()</code>	(키, 값) 튜플을 반환하고 사전에서 항목을 삭제한다.
<code>d.pop(key)</code>	key 항목의 값을 반환하고 사전에서 제거한다.
<code>d.fromkeys(seq[, value])</code>	<code>fromkeys()</code> 는 클래스 메서드다. <code>seq</code> 와 <code>value</code> 를 이용하여 만든 새로운 사전을 반환한다.

9. Python 사전

```
>>> d = { 'one':1, 'two':2, 'three':3 }
>>> d2 = d.copy()                    # 사전 복사
>>> d[ 'four' ] = 4; d; d2           # 항목 추가, d2에는 변화가 없다
>>> d3 = { 'nine':9, 'ten':10 }
>>> d.update( d3 ); d               # d3의 사전 내용을 d로 갱신
>>> d.popitem(); d                  # 항목 한 개를 꺼낸다.
>>> d.pop( 'four' ); d              # 'four' 항목을 꺼낸다.
>>> d.get( 'one' )                  # d[ 'one' ]과 동일
>>> d.get( 'ten' )                  # 'ten'이 없으면 None을 default로 반환
>>> d.get( 'ten', 10 )              # 'ten'이 없으면 10을 default로 반환
>>> d.setdefault( 'ten', 10 ); d    # 'ten'이 없을 경우 새로운 항목 추가
>>> dict.fromkeys                  # 클래스 메서드이므로 직접 호출 가능
>>> {}.fromkeys                    # 임의의 사전 객체를 이용해서 호출
>>> {}.fromkeys( 'abcde', 1 )       # 각 key에 대해 동일한 객체 1이 할당
>>> d = {}.fromkeys( 'abcde', [] ); d # 값으로 사용된 []는 한 개의 객체이다
>>> d[ 'a' ].append( 1 ); d         # 한 객체의 수정은 전체에 반영
>>> d2 = dict( ( c, [] ) for c in 'abcde' ); d2; d2[ 'a' ].append( 1 ); d
```

9. Python 사전

◆ 사전 내장(Dictionary Comprehension)

- 리스트 내장과 유사한 방식으로 동작하지만 사전을 만들어 낸다.
- 사전 내용은 {}를 사용하며 키:값 형식으로 항목을 표현한다.

```
>>> { w:1 for w in 'abc' }           # 키:값 = w:1
>>> a1 = 'abcd'
>>> a2 = ( 1, 2, 3, 4 )
>>> { x:y for x, y in zip( a1, a2 ) }
>>> { w:k for k, w in [ ( 1, 'one' ), ( 2, 'two' ), ( 3, 'three' ) ] }
>>> { w:k + 1 for k, w in enumerate( [ 'one', 'two', 'three' ] ) }
```

❖ enumerate() 함수는 시퀀스형 데이터를 (위치, 요소값) 튜플로 넘겨주는 반복자를 반환한다. 이 함수는 주로 for 문과 연계해서 사용한다.

```
>>> d = { 'one':1, 'two':2, 'three':3 }
>>> { v:k for k, v in d.items() }
>>> { ( k, v ):k + v for k in range( 3 ) for v in range( 3 ) }
```

9. Python 사전

◆ 심볼 테이블

- 심볼 테이블(Symbol Table)이란 변수들이 저장되는 공간이다. 파이썬에서는 심볼 테이블을 저장하는 데 사전을 사용한다.
- 심볼 이름은 사전의 키로, 심볼 값은 사전의 값으로 등록된다.

- 전역/지역 심볼 테이블
- `globals()` 함수를 사용하면 전역 영역(모듈 영역)의 심볼 테이블을 얻는다. 반환된 사전은 사용한 변수 이외에 `'__doc__'` 나 `'__name__'`과 같은 내장 이름이 들어 있다.

```
>>> globals()
```

- `locals()` 함수를 사용하면 지역 영역의 심볼 테이블을 얻는다.

```
>>> def f( a, b ):
    c = 10
    print( locals() )
>>> f( 2, 3 )
```

9. Python 사전

◆ 객체의 심볼 테이블

- 이름 공간(심볼이 저장되는 공간)을 가지는 모든 객체는 심볼 테이블을 갖는다. 모듈과 함수, 클래스, 클래스 인스턴스, 함수 모두 이름 공간(Namespace)을 갖는다.
- 어떤 객체의 심볼 테이블은 '__dict__' 속성을 확인해 보면 된다.

```
>>> import re                                # 모듈 추가
>>> re.__dict__                               # 모듈의 심볼 테이블
>>> class C:
        x = 10
        y = 20
>>> C.__dict__
>>> c = C(); c.a = 100; c.b = 200; c.__dict__
>>> def f():
        pass
>>> f.a = 1; f.b = 2; f.__dict__
```


9. Python 사전

- 이름 공간에서 어떤 이름의 값을 얻어내는 방법

```
>>> import math
>>> math.sin
>>> math.__dict__[ 'sin' ]
>>> getattr( math, 'sin' )           # getattr( 이름공간, 이름 )
```

- 이름 공간에 값을 설정하는 방법

```
>>> math.mypi = 3.14; math.mypi
>>> math.__dict__[ 'mypi' ] = 3.14 ; math.mypi
>>> setattr( math, 'mypi', 3.14 ) ; math.mypi
```

- 모듈 내에서 자신의 모듈을 참조하는 방법

```
>>> import sys
>>> current_module = sys.modules[ __name__ ]
>>> a = 10
>>> getattr( current_module, 'a' )
>>> current_module.__dict__[ 'a' ]
```

10. Python 객체의 복사와 형변환

◆ 객체의 복사

- 파이썬에서 복사는 두 가지가 있다. 하나는 참조 복사이고 다른 하나는 객체 복사이다.
- 참조 복사란 객체는 그대로 두고 객체의 참조만 복사하는 것이다. 치환문(=)을 이용하여 참조 복사를 수행한다. 치환문은 오른쪽 객체의 참조를 왼쪽의 변수에 저장하라는 의미이다.

```
>>> a = 1                # 참조 복사
>>> a = 2                # 참조 복사
>>> b = a                # 참조 복사, 객체 참조 횟수 증가
>>> l1 = [ 1, 2, 3 ]
>>> l2 = [ 4, 5, l1 ]
>>> x = [ l1, l2, 100 ]
>>> y = x                # 참조 복사
>>> x.append( 200 )      # x의 변경은 y의 변경과 같다.
>>> x; y
```

10. Python 객체의 복사와 형변환

➤ 객체 자체의 복사 기능을 지원하지 않을 경우 사용하는 copy 모듈은 얇은 복사(Shallow Copy)와 깊은 복사(Deep Copy)를 지원한다.

- 얇은 복사 : 1단계 복합 객체를 생성하고 원래 객체로부터 내용을 복사한다.
- 깊은 복사 : 복합 객체를 재귀적으로 생성하고 내용을 복사한다.

```
>>> import copy
>>> a = [ 1, 2, 3 ]; b = [ 4, 5, a ]; x = [ a, b, 100 ]
>>> y = copy.copy( x )                # 얇은 복사
>>> x[0][1] = 100; print( x ); print( y )
```

- y는 리스트를 만들고 x로부터 내용을 채운다. y는 x와는 다른 객체지만 값들은 x의 내용과 동일하다.

```
>>> z = copy.deepcopy( x )            # 깊은 복사
>>> x[1][0] = 400; print( x ); print( z )
```

- z는 x로부터 재귀적으로 복합 객체를 생성하고 그 내용을 복사한다. 즉, 100과 같은 단순한 객체는 복사되지 않고 참조만 한다.

10. Python 객체의 복사와 형변환

- 깊은 복사가 복합 객체만을 새로 생성하기 때문에 복합 객체가 한 개만 있을 경우는 얕은 복사와 깊은 복사의 차이가 없다.

```
>>> a = [ 'hello', 'world' ]
>>> b = copy.copy( a )           # 얕은 복사
>>> a is b
>>> a[ 0 ] is b[ 0 ]

>>> c = copy.deepcopy( a )       # 깊은 복사
>>> a is c
>>> a[ 0 ] is c[ 0 ]
```

10. Python 객체의 복사와 형변환

- 깊은 복사를 할 때는 객체들이 재귀적인 구조로 되어 있지 않은지 주의해야 한다. 또한, 깊은 복사는 모든 복합 객체를 복사해 별도의 객체로 생성하기 때문에 공유 문제에 신경 써야 한다.
- 내장 자료형은 일부 복사 기능을 지원한다. 리스트의 슬라이싱과 사전의 `dict.copy()`는 얇은 복사이다.
- 슬라이싱은 리스트를 새로 만들고 참조를 복사한다.

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

```
>>> m = l[ 1:4 ]
```

```
>>> l[1] = 200
```

```
>>> l
```

```
>>> m
```

10. Python 객체의 복사와 형변환

◆ 수치형 변환

➤ 정수형으로의 변환 : 다른 자료형에서 정수형으로 형을 변환하려면 기본적으로 `int()` 내장 함수를 사용한다. 변환할 수 없으면 `ValueError` 에러가 발생한다.

- `int()` 내장 함수 : 소수 부분을 없애고 정수 부분만 취한다.

```
>>> int( 1.1 )
```

```
>>> int( -1.9 )
```

- `round()` 내장 함수 : 반올림해서 정수형의 실수를 취한다.

```
>>> round( 1.1 )
```

```
>>> round( 1.9 )
```

```
>>> round( 1.23456, 3 )
```

소수점 세 자리에서 반올림

10. Python 객체의 복사와 형변환

- math 모듈의 floor() 함수 : 주어진 인수보다 작거나 같은 수 중에서 가장 큰 정수형의 실수를 취한다.

```
>>> import math
>>> math.floor( 1.1 )
>>> math.floor( 1.9 )
>>> math.floor( -1.9 )
```

- math 모듈의 ceil() 함수 : 주어진 실수보다 크거나 같은 수 중에서 가장 작은 정수형의 실수를 취한다.

```
>>> math.ceil( 1. 0 )
>>> math.ceil( 1.1 )
>>> math.ceil( 1.9 )
>>> math.ceil( -1.9 )
```

10. Python 객체의 복사와 형변환

- 실수형으로의 변환 : 실수형으로 형을 변환할 때는 `float()` 함수를 사용한다.

```
>>> float( '1234' )  
>>> float( '12.34' )  
>>> float( 1234 )
```

- 복소수로의 변환 : 실수(혹은 정수)로 부터 복소수를 만들려면 `complex()` 함수를 사용한다.

```
>>> complex( 1 )  
>>> complex( 1, 3 )  
>>> complex( 0, 3 )
```


10. Python 객체의 복사와 형변환

➤ 진수 변환

>>> int('64', 16)	# 16진수 64를 10진수로 변환
>>> int('144', 8)	# 8진수 144를 10진수로 변환
>>> int('101111', 2)	# 2진수 101111을 10진수로 변환
>>> hex(100)	# 10진수 100을 16진수 문자열로 변환
>>> oct(100)	# 10진수 100을 8진수 문자열로 변환
>>> bin(100)	# 10진수 100을 2진수 문자열로 변환
>>> "{0:x}".format(100)	
>>> "{0:o}".format(100)	
>>> "{0:b}".format(100)	

10. Python 객체의 복사와 형변환

➤ 시퀀스형으로의 변환

- list() 함수 리스트로 변환
- tuple() 함수 튜플로 변환

```
>>> t = ( 1, 2, 3, 4 )
>>> l = [ 5, 6, 7, 8 ]
>>> s = 'abcd'
>>> list( t )
>>> list( s )
>>> tuple( l )
>>> tuple( s )
>>> d = { 1:'one', 2:'two', 3:'three' }
>>> list( d.keys() ); list( d.values() ); list( d.items() )
>>> keys = [ 'a', 'b', 'c', 'd' ]
>>> values = [ 1, 2, 3, 4 ]
>>> dict( zip( keys, values ) )
```

10. Python 객체의 복사와 형변환

➤ 문자열로의 변환

- `str()` 함수 비형식적인 문자열로 변환
- `repr()` 함수 형식적인 문자열로 변환

```
>>> print( str( [ 1, 2, 3 ] ), str( ( 4, 5, 6 ) ), str( 'abc' ) )
>>> print( repr( [ 1, 2, 3 ] ), repr( ( 4, 5, 6 ) ), str( 'abc' ) )
>>> eval( '[ 5, 6, 7, 8 ]' )                      # 리스트 생성
```

```
>>> a = { 1:'one', 2:'two' }
>>> b = repr( a ); b
>>> c = eval( b )
>>> print( c )
```

```
>>> char( 97 )                                      # 유니코드 -> 문자
>>> ord( 'a' )                                    # 문자 -> 유니코드
```

10. Python 객체의 복사와 형변환

```
>>> b = b'bytes'
>>> type( b )
>>> b.decode( 'utf-8' )           # 바이트에서 문자열로 변환
>>> s = 'string'
>>> s.encode( 'utf-8' )           # 문자열에서 바이트로 변환

>>> hex( ord( 'a' ) )             # 코드 값 확인
>>> import binascii
>>> binascii.hexlify( b'abc' )     # 바이트 열
>>> buf = bytearray( b'abcde' )
>>> binascii.hexlify( buf )        # 바이트 배열
>>> binascii.unhexlify( b'6162636465' ) # 16진 바이트 열 -> 2진 바이트 열

>>> format( 123456789, ',' )
>>> "{:},{:}".format( 10030405, 12345 )
>>> "{0:},{1:}".format( 10030405, 12345 )
```

4. Python 제어문

Python 제어문에 대한 이해

1. if문
2. for문
3. while문

1. if문

◆ if문

- 조건문이라 하며, 특정 조건에 맞으면 문들을 실행하고 그렇지 않으면 건너뛰다.

```
if <조건식1>:
```

```
    <문1>
```

```
elif <조건식2>:
```

```
    <문2>
```

```
else:
```

```
    <문3>
```

- <조건식1>이 참이면 <문1>이 실행되고, 그렇지 않으면 <조건식2>를 검사해서 참이면 <문2>를 실행하고, 그렇지 않으면 <문3>를 실행한다.
- 조건식이나 else 다음에 : 을 잊지 말고 입력해야 한다. : 은 다음 문 들이 현재의 문 내부 블록에 속한다는 것을 알려준다.
- if/for/while과 같은 제어문, def/class와 같이 내부 블록을 가지는 문에서만 : 을 사용한다.
- if ~ elif ~ else과 내부 블록의 문들은 줄(열)이 맞아야 에러가 안 난다. 파이썬은 들여쓰기에 민감하며 블록의 구분은 들여쓰기만으로 구분한다.

1. if문

```
>>> a = 10
>>> if a > 5:
    print( 'Big ' )
else:
    print( 'Small' )
>>> if a > 5: print( 'Big' )
else: print( 'Small' )

>>> n = -2
>>> if n > 0:
    print( 'Positive' )
elif n < 0:
    print( 'Negative' )
else:
    print( 'Zero' )
```


1. if문

```
>>> order = 'spagetti'
>>> if order == 'spam':
    price = 500
elif order == 'ham':
    price = 700
elif order == 'egg':
    price = 300
elif order == 'spagetti':
    price = 900
else:
    price = 0
```

- 선택문에 있어서 때로는 사전을 이용하는 것이 더 편할 때도 있다.

```
>>> order = 'spagetti'
>>> menu = { 'spam':500, 'ham':700, 'egg':300, 'spagetti':900 }
>>> prce = menu.get( order, 0 )          # 0은 키가 없을 때의 기본값
```

1. if문

```
>>> x = a * 2 if a > 5 else a / 2          # 삼항 연산자 이용
```

```
>>> if a > 5:
```

```
    a * 2
```

```
else:
```

```
    a / 2
```

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

```
>>> ( a / 2, a * 2 )[ a > 5 ]
```

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

```
    return a + b
```

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

```
    return a - b
```

```
>>> select = 0
```

```
>>> ( add, sub )[ select ]( 2, 3 )
```

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

```
>>> { True:add, False:sub }[ a > 5 ]( 3, 4 )          # 사전을 이용한 선택문
```

2. for문

◆ for문

➤ 반복문 이다.

```
for <타깃> in <객체>:
```

```
    <문1>
```

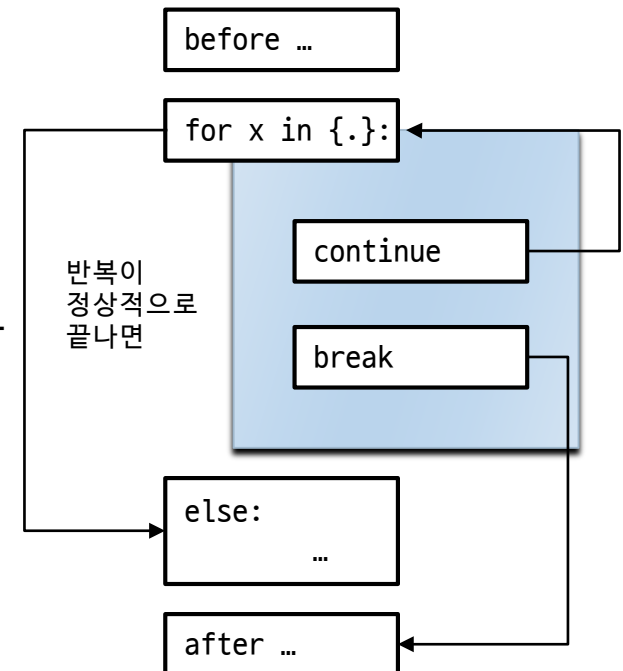
```
else:
```

```
    <문2>
```

➤ <객체>는 시퀀스형 데이터여야 한다.

➤ <객체>의 각 항목은 <타깃>에 치환되어 <문1>이나 <문2>를 실행한다. 반복 횟수는 <객체>의 크기가 된다.

➤ for문의 내부 블록에서 continue를 만나면 시작 부분으로 이동하고 break를 만나면 for문의 내부 블록을 완전히 빠져나온다.



2. for문

```
>>> a = [ 'cat', 'cow', 'tiger' ]
```

```
>>> for x in a:
```

```
    print( len( x ), x )
```

```
>>> for x in [ 1, 2, 3 ]:
```

```
    print( x, end = ' ' )
```

```
>>> for c in 'abcdef':
```

```
    print( c, ord( c ) )
```

- 순차적으로 숫자를 반복하는 경우에는 range() 함수를 사용한다.

```
>>> range( 10 )
```

```
>>> list( range( 10 ) )
```

```
>>> for x in range( 10 ):
```

```
    print( x, end = ' ' )
```

2. for문

```
>>> sum = 0
>>> for i in range( 1, 11 ):
    sum += I
>>> print( sum )
```

- for 문을 이용할 때 요소의 값 뿐 아니라 인덱스의 값도 함께 필요하다면 `enumerate()` 내장 함수를 사용한다. 이 함수는 (인덱스의 요소의 값) 튜플 데이터를 반복해서 넘겨주는 반복자(iterator) 반환한다.

```
>>> l = [ 'cat', 'dog', 'bird', 'pig', 'spam' ]
>>> for k, animal in enumerate( l ):
    print( k, animal )
```

2. for문

- for 문 내부에 break를 사용하면, 반복을 종료하고 내부 블록 밖으로 나간다.

```
# for-break.py
for x in range( 10 ):
    if x > 3: break
    print( x )
print( 'done' )
>>> python for-break.py
```

- for 문 내부에 continue를 사용하면, 내부 블록의 나머지 부분을 실행하지 않고 for 문의 시작 부분으로 이동한다.

```
# for-continue.py
for x in range( 10 ):
    if x < 8: continue
    print( x )
print( 'done' )
>>> python for-continue.py
```

2. for문

- for 문에서 else 블록은 for 문의 break 문으로 중단되지 않고 종료되었을 때 실행된다. break 문으로 중단되면 for 문의 내부 블록 밖으로 제어가 이동한다.

```
# for-else.py
```

```
for x in range( 10 ):
```

```
    print( x, end = ' ' )
```

```
else:
```

```
    print( 'else block' )
```

```
print( 'done' )
```

```
>>> python for-else.py
```

```
# for-else-break.py
```

```
for x in range( 10 ):
```

```
    break
```

```
    print( x, end = ' ' )
```

```
else:
```

```
    print( 'else block' )
```

```
print( 'done' )
```

```
>>> python for-else-break.py
```

2. for문

- for 문 내부에 또 다른 for 문이 있는 것을 중첩되었다고 말한다.

```
# for-nested.py
```

```
for x in range( 2, 4 ):
```

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

```
        print( x, '*', y, '=', x * y )
```

```
    print( '\n' )
```

```
>>> python for-nested.py
```


3. while문

◆ while문

➤ 반복문 이다.

```
while <조건식>:
```

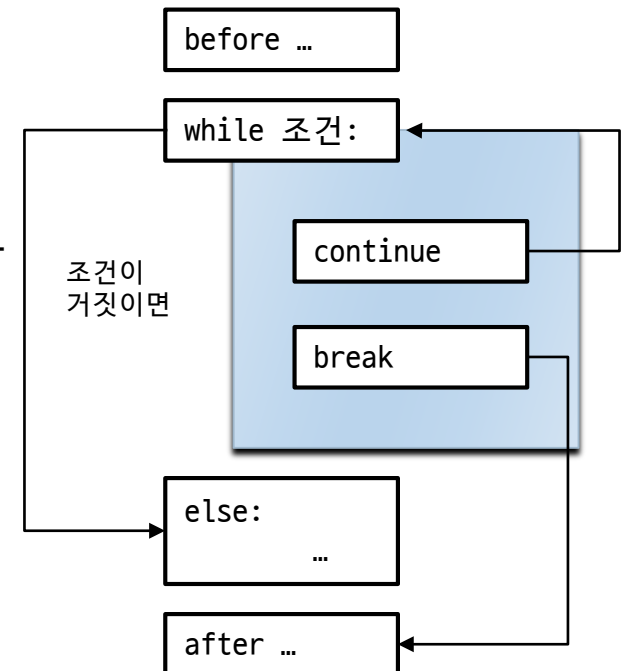
```
    <문1>
```

```
else:
```

```
    <문2>
```

➤ <조건식>을 검사해서 결과가 참인 동안 내부 블록이 반복적으로 실행된다.

➤ else 블록은 <조건식>이 거짓이 되어 while 문을 빠져나올 때 실행된다. break로 빠져나올 때는 else 블록을 실행하지 않는다.



3. *while*문

```
>>> count = 1
>>> while count < 11:
    print( count, end = ' ' )
    count = count + 1
```

```
#while.py
sum = 0
count = 0
while count < 10:
    count = count + 1
    sum = sum + count
print( sum )
```

```
>>> python while.py
```

- while 문도 for 문 에서와 같이 break, continue, else를 사용할 수 있으며 동작 은 for 문과 동일하다.

5. Python 함수

Python 함수에 대한 이해

1. 함수 정의
2. 함수 호출
3. 유효 범위
4. 함수의 인수
5. 함수 안의 함수
6. 함수 객체의 속성
7. 재귀적 프로그래밍
8. 람다 함수
9. 함수적 프로그래밍

1. 함수 정의

- 함수는 문들을 하나로 묶는 코드 블록이다.
- 반복적으로 실행이 가능하며, 기능 중심으로 프로그램을 구성하는데 사용한다.
- 함수는 프로그램을 논리적으로 이해하는 데 도움을 준다. 함수는 단지 반복 실행의 가능성 때문에 정의하는 것은 아니고, 코드의 일정 부분이 별도의 논리적 개념으로 분리 된다는 것이 가능할 때, 함수로 분리한다.
- 복잡한 내용을 단순한 하나의 개념으로 만든 추상화(abstraction)에 해당한다.
- 함수는 코드를 재사용하게 해주고 프로그램을 논리적으로 구성하게 해준다.

1. 함수 정의

➤ 함수 정의 형식

```
def 함수이름( 인수 목록 ):  
    <문> ...  
    return <값>
```

- 함수를 정의하는 키워드는 def이며, 이어서 함수 이름과 괄호 안에 인수 목록을 나열한다.
- 함수 선언부(헤더)의 끝은 항상 : 으로 끝나야 한다.
- 함수의 몸체는 함수 선언부 다음 줄에서 들여쓰기를 해서 시작한다.
- 파이썬은 어떤 형식의 데이터도 전달할 수 있으므로 인수의 자료형은 명시하지 않는다.
- return 키워드는 함수를 호출한 곳으로 되돌려 주는 값을 명시한다.

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

```
>>> add                                # 함수 객체 확인
```

```
>>> add( 1, 2 )                        # 함수 호출
```

1. 함수 정의

- 함수는 계층 구조로 설계하는데, 입출력이나 기초적인 기능을 하는 함수들 위에 좀 더 고급 기능을 하는 함수를 작성한다. 그 위에 또 다른 고급 함수를 정의한다.
- 파이썬에서 모든 객체는 동적으로 자료형이 결정되므로, 어떤 연산을 수행할 때 해당 객체에 맞는 연산을 자동으로 호출한다.
- 예를 들어, + 연산은 객체가 숫자인 경우 수치 덧셈, 문자열인 경우 문자열 연결하기, 리스트인 경우 리스트 연결하기등 객체에 맞는 연산을 적용한다.

```
>>> def add( a, b ):  
    return a + b  
  
>>> c = add( 1, 3.4 )  
  
>>> d = add( 'dynamic', 'typing' )  
  
>>> e = add( [ 'list' ], [ 'add', 'list' ] )  
  
>>> print( c, d, e )
```

1. 함수 정의

➤ return문

- 인수 없이 return문 만을 사용하면 함수를 호출한 측에 아무 값도 전달하지 않는다. 인수 없이 반환을 하지만, 실제로는 None 객체를 전달한다. None 객체란 파이썬 내장 객체로서 아무 값도 없음을 표현하는 객체이다.

```
>>> def nothing():
```

```
    return
```

```
>>> nothing(); print( nothing() )
```

- return문 없이 종료하는 경우도 None 객체를 전달하기는 마찬가지이다.

```
>>> def simple():
```

```
    pass
```

```
>>> simple(); print( simple() )
```

- return 문에 여러 개의 값을 사용할 경우, 이들은 튜플로 구성하여 전달한다.

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

```
    return b, a
```

```
>>> a = 10; b = 20
```

```
>>> swap( a, b )
```


2. 함수 호출

- 함수에 인수를 전달하는 방법으로는 값에 의한 호출(Call by Value)과 참조에 의한 호출(Call by Reference)이 일반적이다.
- 값에 의한 호출(Call by Value)은 계산한 결과값이 함수의 인수로 전달되는 것이다. 함수 내에서 인수 값이 변경되어도 호출하는 측에 아무런 영향을 미치지 못한다.
- 참조에 의한 호출(Call by Reference)은 호출하는 측의 변수 참조가 호출을 받은 함수의 인수로 전달된다. 따라서 함수 내에서 해당 인수 값이 변경되면 호출하는 측에도 영향을 미친다.
- 파이썬에선 위 두 방법이 아닌 독특한 방법으로 인수를 전달한다. 이를 객체 참조에 의한 호출(Call by Object Reference) 또는 공유에 의한 호출(Call by Sharing)이라 부른다.
- 파이썬은 함수를 호출할 때 객체의 참조를 넘겨준다.

2. 함수 호출

```
>>> def f( t ):
    t = 10                                # t에 새로운 객체 10의 참조가 할당
```

- 함수 f에 변경 불가능한 데이터 a를 전달하면

```
>>> a = 20; f( a )                        # 참조에 의한 호출
>>> print( a )
```

- 변경 불가능한 튜플을 인수로 전달

```
>>> def h( t ):
    t = ( 1, 2, 3 )                      # t에 새로운 튜플의 참조가 할당
>>> a = ( 5, 6, 7 ); h( a ); a           # 참조에 의한 호출
```

- 리스트와 같은 변경 가능한 객체를 전달

```
>>> def g( t ):
    t[1] = 10                            # 인수로 전달된 참조로 리스트 내용 변경
>>> a = [ 1, 2, 3 ]; g( a )              # a의 참조가 g()의 t에 전달
>>> print( a )
```

2. 함수 호출

- 함수 내에서 인수로 전달받은 객체의 참조를 참조하지 않고 다른 객체 값을 치환하는 경우는 변경된 내용이 함수를 호출한 측에 반영되지 않는다.

```
>>> def gg( t ):
```

```
    t = [ 1, 2, 3 ]          # 새로운 리스트 생성
```

```
>>> a = [ 5, 6, 7 ]; gg( a )
```

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

- ❖ 모든 인수는 인수 자체가(함수 f, h, gg와 같이) 다른 객체로 치환될 때, 함수를 호출한 측에 아무런 영향을 미치지 못한다. 변경 가능한 인수는 참조를 이용하여 내부 객체를 변경할 때 변경이 호출한 측에 반영된다.
- ❖ 이러한 파이썬의 독특한 호출 방식을 객체 참조에 의한 호출(Call by Object Reference)이나 공유에 의한 호출(Call by Sharing)이라고 한다.

3. 유효 범위

- 유효 범위 규칙(Scope Rule)이란 변수가 유효하게 사용되는 문맥(Context) 범위를 정하는 규칙이다.
- 변수가 특정 범위에서 유효한지를 결정한다. 변수는 다양한 이름 공간에 저장되어 있다.
- 파이썬에서 이름 공간을 찾는 규칙을 LEGB규칙이라고 한다.
 - L Local, 함수 내에 정의된 지역변수이다.
 - E Enclosing Function Local, 함수를 내포하는 또 다른 함수 영역이다.
 - G Global, 함수 영역에 포함되지 않는 모듈 영역이다.
 - B Built-in, 내장영역이다.
- 변수가 저장되는 이름 공간은 변수가 어디에서 정의(혹은 치환)되었는지에 따라서 결정된다.
- 변수가 함수 내에서 정의되면, 함수의 지역(Local) 변수가 된다. 변수가 함수 외부에서 정의되면 해당 모듈의 전역(Global) 변수가 된다.

3. 유효 범위

```
>>> x = 10                # G에 해당
>>> y = 11
>>> def foo():
    x = 20                # foo함수의 L에 해당, bar함수의 E에 해당
    def bar():
        a = 30            # L에 해당
        print( a, x, y )  # 각 변수는 L, E, G에 해당
    bar()
    x = 40
    bar()
>>> foo()
```

- 파이썬에서 함수 안에 또 다른 함수를 정의하는 것이 가능하다. 변수의 이름은 항상 안쪽에서부터 바깥쪽으로 찾아 나간다.
- 동일한 이름이 있으면 안쪽에 있는 이름 공간의 이름이 먼저 사용되는 것이 원칙이다.

3. 유효 범위

```
>>> abs                                # 내장함수 abs()
>>> abs = 10                           # 전역 변수 abs
>>> abs( -5 )                          # 전역 변수 abs에 의해서 내장함수 abs()가 가려짐
>>> del abs                            # 전역 변수 삭제
>>> abs( -5 )
>>> getattr( __builtins__, 'abs' )    # 내장 함수 직접 참조 방법
>>> dir( __builtins__ )                # 내장 공간 __builtins__ 의 내용 출력
```

- 함수 내부에서 값을 치환해서 사용하는 변수를 전역 변수로 사용하려면 `global` 선언자를 사용하여 변수가 전역 변수임을 선언해야 한다.

```
>>> def f( a ):                        # a는 지역 변수
    global h                          # h는 전역 변수임을 알림
    h = a + 10
>>> a = 10
>>> f( a )
>>> h
```

3. 유효 범위

- 전역 변수의 이름을 정하면서 흔히 하는 실수

```
>>> g = 10
```

```
>>> def f():
```

```
    a = g
```

```
    # g는 지역 변수, g의 변수는 아직 정해지지 않음
```

```
    g = 20
```

```
    # g는 지역 변수
```

```
    return a
```

```
>>> f()
```

3. 유효 범위

- 전역 영역이 아닌 중첩된 함수의 변수를 사용하려고 할 경우에 변수를 `nonlocal`문으로 선언할 수 있다. `global`선언자는 전역 영역의 변수에 접근하는 반면, `nonlocal` 선언자는 가장 가까운 이름공간에서부터 변수를 찾는다.

```
>>> def outer():
    x = 1
    def inner():
        nonlocal x           # 함수 outer의 x
        x = 2                # 함수 inner의 지역 변수가 아님
        print( 'inner : ', x )
    inner()
    print( 'outer : ', x )
>>> outer()
```


4. 함수의 인수

- 인수에서 기본값이란 함수를 호출할 때 인수를 넘겨주지 않아도 인수가 자신의 기본값을 취하도록 하는 기능이다.
- 기본값을 지정하면 꼭 필요한 인수만 넘겨주면 되므로 함수 호출이 편리하다.

```
>>> def inc( a, step = 1 ):
    return a + step
```

```
>>> b = 1
```

```
>>> b = inc( b )
```

```
>>> b
```

```
>>> b = inc( b, 10 )
```

```
>>> b
```

❖ 주의할 점은 기본값이 정의된 인수 다음에 기본값이 없는 인수가 올 수 없다.

```
>>> def dec( step = 1, b ):    # 잘못된 인수 형식, error
    pass
```

4. 함수의 인수

- 함수의 호출에서 키워드 인수란 인수 이름으로 값을 전달하는 방식이다.

```
>>> def area( height, width ):
```

```
    return height * width
```

```
>>> area( width = 20, height = 10 )      # 순서가 아닌 이름으로 값을 전달
```

- ❖ 함수를 호출할 때 키워드 인수의 위치는 보통의 인수 이후이다.

```
>>> area( 20, width = 5 )
```

```
>>> area( width = 5, 20 )                # error
```

- 고정되지 않은 수의 인수를 함수에 전달하는 방법이 있다. 함수를 정의할 때 인수 목록에 반드시 넘겨야 하는 고정 인수를 우선 나열하고, 나머지를 튜플 형식으로 한꺼번에 받는다.

```
>>> def varg( a, *arg ):
```

```
    print( a, arg )
```

```
>>> varg( 1 )
```

```
>>> varg( 2, 3 )
```

```
>>> varg( 2, 3, 4, 5, 6 )
```

4. 함수의 인수

- 가변 인수는 *var 형식으로 인수 목록 마지막에 하나만 나타낼 수 있다. 가변 인수를 이용하면 파이썬으로 C 언어의 printf()를 흉내낼 수 있다.

```
# printf.py
def printf( format, *args ):
    print( format % args )
printf( "I've spent %d days and %d night to do this", 6, 5 )
```

- 키워드 인수를 이용해서 함수를 호출할 때, 만일 미리 정의되어 있지 않은 키워드 인수를 받으려면, 함수를 정의할 때 마지막에 **kw 형식으로 기술한다. 전달받는 형식은 사전이다. 즉, 키는 키워드(변수명)가 되고, 값은 키워드 인수로 전달된 값이 된다.

```
>>> def f( width, height, **kw ):
    print( width, height )
    print( kw )

>>> f( width = 10, height = 5, depth = 10, dimension = 3 )
```

- ❖ 사전 키워드 인수는 함수의 가인수 목록의 제일 마지막에 나와야 한다.

4. 함수의 인수

```
>>> def g( a, b, *args, **kw ):
    print( a, b )
    print( args )
    print( kw )
>>> g( 1, 2, 3, 4, c = 5, d = 6 )
```

- 만일 함수 호출에 사용하는 인수들이 튜플에 있으면 * 를 이용하여 함수를 호출할 수 있다.

```
>>> def h( a, b, c ):
    print( a, b, c )
```

```
>>> args = ( 1, 2, 3 )
```

```
>>> h( *args )
```

❖ 함수 호출에 사용하는 인수들이 사전에 있다면 ** 를 이용하여 함수를 호출한다.

```
>>> dargs = { 'a':1, 'b':2, 'c':3 }
```

```
>>> h( **dargs )
```

```
>>> args = ( 1, 2 ); dargs = { 'c':3 }; h( *args, **dargs )
```

5. 함수 안의 함수

➤ 파이썬의 함수는 모두 일급 함수이다.

- 일급 함수(First Class Function)

- ① 함수를 다른 함수에 인수로 전달할 수 있다.
- ② 함수의 반환 값으로 전달할 수 있다.
- ③ 변수나 자료 구조에 저장할 수 있다.

```
>>> def add( a, b ):
    return a + b
>>> addition = add          # ③
>>> addition( 3, 4 )
>>> def f( g, a, b ):       # ①
    return g( a, b )
>>> f( add, 2, 3 )
```

5. 함수 안의 함수

```
>>> def decorate( type = 'italic' ):
    def italic( s ):
        return '<i>' + s + '</i>'
    def bold( s ):
        return '<b>' + s + '</b>'
    if type == 'italic':
        return italic                # ②
    else:
        return bold                  # ②

>>> dec = decorate()
>>> dec( 'hello' )
```

5. 함수 안의 함수

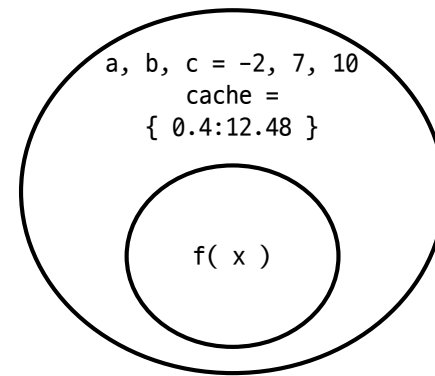
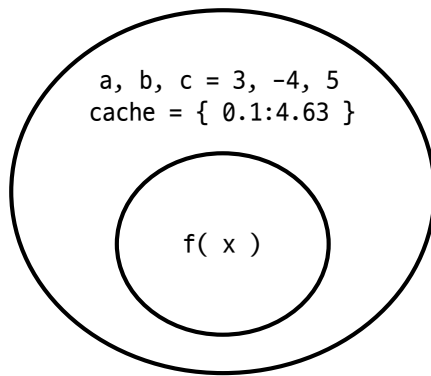
- 함수 클로저(Function Closure)란 함수가 참조할 수 있는 비지역 변수(Non-local Variable)나 자유 변수(Free Variable)를 저장한 심볼 테이블 혹은 참조 환경(Reference Environment)이 함수와 함께 제공되는 것을 의미한다.

```
>>> def quadratic( a, b, c ):
    cache = {}
    def f( x ):
        if x in cache:
            return cache[x]
        y = a * x * x + b * x + c
        cache[ x ] = y
        return y
    return f

>>> f1 = quadratic( 3, -4, 5 ); f1( 0.1 )
>>> f2 = quadratic( -2, 7, 10 ); f2( 0.4 )
```

5. 함수 안의 함수

- 함수 quadratic의 호출에 의해 반환되는 함수 객체 f는 내부에 지역 변수 x, y를 포함하며 정적인(Static) 비지역 변수(Non-local Variable) cache, a, b, c를 갖는다. f1과 f2는 각각 독립적인 비지역 변수를 실행 문맥(Execution Context)으로 갖는다. 함수 f를 둘러싸고 있는 각각의 자유 변수 영역은 객체 f1이나 f2가 삭제될 때까지 유지된다.



5. 함수 안의 함수

- 함수 클로저는 함수마다 독립적인 이름 공간을 제공하는 장점이 있다. 이를 이용하면 마치 인스턴스 객체처럼 별도로 동작하는 함수를 정의할 수 있다.

```
>>> def makeCounter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter
```

```
>>> c1 = makeCounter(); c2 = makeCounter()
```

```
>>> c1(); c1(); c2()
```

- 함수 클로저는 함수 객체의 `__closure__` 속성으로 확인할 수 있다.

```
>>> c1.__closure__
```

```
>>> c1.__closure__[0]
```

```
>>> c1.__closure__[0].cell_contents
```

5. 함수 안의 함수

- `functools` 모듈에서 제공하는 `partial()` 함수는 함수 클로저를 반환하는 함수이다. 이 함수는 기존 함수를 이용하여 일부 인수가 미리 정해진 새로운 함수를 반환한다.

```
>>> from functools import partial
>>> bin2int = partial( int, base = 2 )
>>> bin2int( '10010' )
>>> def quadratic( x, a, b, c ):
    return a * x * x + b * x + c
>>> f1 = partial( quadratic, a = 3, b = -4, c = 5 )
>>> f1( 0.1 )
```

6. 함수 객체의 속성

➤ 함수 객체는 여러 가지 속성을 갖는다.

- `__doc__` : 문서 문자열
- `__name__` : 함수의 이름
- `__defaults__` : 기본 인수 값
- `__code__` : 코드 객체
- `__globals__` : 함수 전역 영역을 나타내는 사전

```
>>> def f( a, b, c = 1 ):
    'func attribute testing'
    localx = 1
    localy = 2
    return 1

>>> f.__doc__
>>> f.__name__
>>> f.__defaults__
>>> f.__code__
>>> f.__globals__
```

6. 함수 객체의 속성

- `__code__` 속성은 함수의 코드 객체이다. 코드 객체는 의사 컴파일(Psuedo-compiled)된 실행 가능한 파이썬 코드이다. 코드 객체는 `compile()` 내장 함수에 의해 반환되고, 함수 객체의 `__code__` 속성으로 참조 된다.
- 코드 객체는 코드에 관한 정보만을 가지고 있는 반면에 함수 객체는 함수를 수행하기 위한 여러 정보를 함께 가지고 있다. 예를 들어, 함수 객체는 기본적인 전역 영역을 가지고 있고, 함수를 호출할 때 전달되지 않으며 자동으로 설정되는 인수의 기본값들을 가지고 있는 반면에 코드 객체는 그렇지 못하다. 또한 코드 객체는 변경 불가능한 자료형의 일종이다.
- `__code__` 속성을 이용하여 함수를 호출할 일은 거의 없지만, 유용한 정보를 추출해 내는 것은 가능하다. 함수 이름, 인수 개수, 지역 변수의 수, 지역 변수 이름 등을 추출해낼 수 있다.

6. 함수 객체의 속성

```
>>> def f( a, b, c, *args, **kw ):
```

```
    calx = 1
```

```
    caly = 2
```

```
    return 1
```

```
>>> code = __code__
```

```
>>> code.co_name
```

```
>>> code.co_argcount
```

```
>>> code.co_nlocals
```

```
>>> code.co_varnames
```

```
>>> code.co_code
```

```
>>> code.co_names
```

```
>>> code.co_filename
```

```
>>> code.co_flags & 0x04
```

```
>>> code.co_flags & 0x08
```

```
>>> code.co_flags & 0x20
```

```
# 코드 객체 참조
```

```
# 함수 이름
```

```
# 필수적인 인수의 개수
```

```
# 전체 지역 변수 수
```

```
# 지역 변수 이름
```

```
# 코드 객체의 바이트 코드 명령어
```

```
# 바이트 코드가 사용하는 이름들
```

```
# 코드 객체를 포함하는 파일 이름
```

```
# 가변 인수를 사용하는가?
```

```
# 키워드 인수를 사용하는가?
```

```
# 발생자인가?
```

7. 재귀적 프로그래밍

- 함수가 자기 자신을 호출하면 재귀(Recursive)라 한다. 재귀적 프로그래밍은 프로그램 언어에서 폭넓게 사용된다. 특히, 자연 언어 처리나 트리 탐색 같은 분야에서는 자주 사용된다.
- ◆ 1부터 N까지의 합을 계산하는 프로그램을 재귀적 프로그래밍으로 작성하면, 우선 합을 계산하는 재귀적 식은 다음과 같이 주어진다.

$\text{sum}(N) = N + \text{sum}(N - 1), N > 1$

$\text{sum}(1) = 1$

```
>>> def sum( N ):
    if N == 1: return 1
    return N + sum( N - 1 )
```

```
>>> sum( 10 )
```

```
>>> sum( 100 )
```

```
>>> sum( 1000 )
```

재귀적 프로그래밍시 발생하는 error

8. 람다 함수

- 람다(Lambda) 함수는 이름이 없는 한 줄짜리 함수이다.
- 람다 함수 정의 형식

lambda <인수들> : <반환할 식>

```
>>> f = lambda:1      # 인수가 없고, 항상 1을 return하는 람다 함수의 참조를 저장
>>> f()               # 람다 함수 호출
```

- 람다 함수는 값을 반환하기 위하여 return 문을 사용하지 않는다. 람다 함수의 몸체(body)는 문이 아닌 하나의 식이다. 람다 함수는 함수 참조를 반환한다.

```
>>> g = lambda x, y: x + y    # 함수 인수 x, y, 반환할 값 x + y
```

```
>>> g( 1, 2 )
```

```
>>> increase = lambda x, inc = 1: x + inc           # 기본값 인수
```

```
>>> increase( 10 ); increase( 10, 5 )
```

```
>>> vargs = lambda x, *args: args                  # 가변 인수
```

```
>>> vargs( 1, 2, 3, 4, 5 )
```

```
>>> kwords = lambda x, *args, **kw: kw             # 키워드 인수
```

```
>>> kwords( 1, 2, 3, a = 4, b = 6 )
```

8. 람다 함수

➤ 일반 함수와 람다(Lambda) 함수의 차이

구분	def로 정의하는 함수	람다 함수
문/식	문(Statement)	식(Expression)
함수 이름	def 다음에 지정한 이름으로 만든 함수 객체를 치환한다.	함수 객체만을 생성한다.
몸체(body)	한 개 이상의 문을 포함한다.	하나의 식만 온다.
반환(return)	return 문에 의해 명시적으로 반환 값을 지정한다.	식의 결과값을 반환한다.
내부 변수 선언	지역 영역에 변수를 만들고 사용하는 것이 가능하다.	지역 영역에 변수를 만드는 것이 가능하지 않다.

8. 람다 함수

➤ 람다(Lambda) 함수 사용 예

```
>>> def f1( x ):
    return x * x + 3 * x - 10

>>> def f2( x ):
    return x * x * x

>>> def g( func ):
    return [ func( x ) for x in range( -10, 10 ) ]

>>> g( f1 ); g( f2 )                # 일반 함수 사용 g() 호출
>>> g( lambda x: x * x + 3 * x - 10 )  # 람다 함수 사용 g() 호출
>>> g( lambda x: x * x * x )
```

- ❖ 일반 함수로 구현한다면 함수를 정의하고 나서 함수를 인수로 전달해야 하지만, 람다 함수는 정의와 동시에 함수 객체로 사용할 수 있다. 이와 같은 차이는 문(Statement)과 식(Expression)에서 온다. 문은 반환 값이 없으며, 식의 일부분으로 사용할 수 없다. def 키워드는 문으로 함수를 정의하고, 람다 함수는 식으로 정의한다. 식은 결과값이 존재하며 다른 식의 일부로 사용할 수 있다. 따라서 정의와 함께 함수 인수로 전달하는 것이 가능하다.

9. 함수적 프로그래밍

- 파이썬은 멀티 패러다임을 추구하는 언어이다. 객체지향 언어이긴 하지만 절차적 언어로 사용하는 것도 가능할 뿐 아니라 함수적 프로그래밍도 가능하도록 설계되었다.
- 함수적 프로그래밍(Functional Programming)이란 함수를 사용하여 문제를 해결하는 프로그래밍 방식을 의미한다. 기본적으로 함수는 입력을 받고 출력을 만들어 내는 단위이며, 동일한 입력에 대해서 다른 출력 결과를 만들어 내는 내부 상태를 가지고 있지 않다.
- 함수적 프로그래밍에서 입력은 일련의 함수들을 계속해서 통과한다. 각 함수는 입력을 받고 출력을 만들어 낸다. 함수적 프로그래밍에서는 내부 상태를 갖는 함수를 지양한다. 같은 입력에 대해 다른 출력을 낼 수 있기 때문이다. 또한 값의 반환 이외에 다른 외부 변수를 변경하는 것도 금한다. 이런 부작용 없는 함수를 순수 함수(Pure Function)라 한다.
- 순수 함수는 출력 값을 입력에 의존해서만 결정한다.

9. 함수적 프로그래밍

- 객체지향 프로그래밍에서 객체들은 내부 상태를 가지고 있고 메서드의 호출은 객체의 상태를 반영한 출력을 만들어 낸다.
- 함수적 프로그래밍은 상태 변화를 가능한 한 최소화하고 함수들을 통과해서 만들어지는 데이터로만 작업을 한다.
- 파이썬에서 객체들을 입력과 출력으로 하는 순수 함수를 정의함으로써 두 가지 접근 방법을 결합할 수 있다.
- 반복자(iterator)는 함수적 프로그래밍에서 중요한 역할을 수행한다. 반복자는 `next()` 함수를 호출 할 때 데이터를 순차적으로 한 번에 하나씩 넘겨주는 자료형이다. 파이썬의 함수적 프로그래밍에서 사용한 순수 함수는 출력으로 반복자를 반환한다. 이 반복자는 출력 값을 필요로 하는 시점에서 값을 계산하는 게으른 계산(Lazy Evaluation)을 한다. 따라서 필요한 만큼의 연산을 수행한 것이 가능하다.

9. 함수적 프로그래밍

➤ map() 내장함수

- ◆ 입력 집합(X)과 사상 함수(f)가 주어져 있을 때, $Y = f(X)$ 를 구한다. 이 함수는 두 개 이상의 인수를 받는다. 첫 인수는 함수(f)이며 두 번째부터는 입력 집합(X)인 시퀀스 자료형(문자열, 리스트, 튜플등)이어야 한다. 첫 번째 인수는 입력 집합 수만큼의 인수를 받는다. map() 함수의 결과인 map객체는 반복자 이다.

```
>>> def f( x ):
    print( 'Calculating..%d' % x )
    return x * x

>>> x = [ 1, 2, 3, 4, 5 ]
>>> map( f, x )
>>> m = map( f, x )          # map객체 반환
>>> next( m )                # 반복자는 next()를 통하여 값을 읽어 낸다.
>>> next( m )                # next()를 읽어 내는 시점에 게으른 계산을 한다.
>>> list( map( f, x ) )
>>> y = map( lambda a: a * a, x )    # 람다 함수 사용
>>> y; list( y )
```

9. 함수적 프로그래밍

➤ map() 내장함수

- ◆ 주어진 시퀀스형 데이터 중에서 필터링하여 참인 요소만 모아 출력한다. 두 개의 인수를 가지면 첫 인수는 함수이고, 두 번째 인수는 시퀀스 자료형이다.

```
>>> f = filter( lambda x : x > 2, [ 1, 2, 3, 34 ] )
```

```
>>> f
```

```
>>> list( f )
```

```
>>> list( filter( lambda x : x % 2 == 1, [ 1, 2, 3, 4, 5, 6 ] ) )
```

```
>>> ''.join( filter( lambda x : x < 'a', 'abcABCdefDEF' ) )
```

- ❖ 조건식이 복잡하면 별도의 함수를 만들어야 할 것이다. 하지만, 이러한 전통적인 방식의 코딩보다는 filter() 함수를 사용함으로써 얻게 되는 간결함과 높은 이해도는 코딩하는 데 많은 이익을 준다.

```
>>> l = [ 'high', 'level', '', 'built-in', '', 'function' ]
```

```
>>> list( filter( None, l ) )
```

- ❖ filter() 함수의 첫 번째 인수로 None 객체를 사용하면 입력 값의 진리값을 판별하는 데 그대로 사용할 수 있다.

6. Python 모듈과 패키지

Python 모듈과 패키지에 대한 이해

1. 모듈
2. 패키지
3. 프로그램 배포

1. 모듈

- 파이썬 모듈은 파이썬 프로그램 파일이나 C, Fortran 확장 파일로 프로그램과 데이터를 정의하고 있으며 client(어떤 모듈을 호출하는 측)가 모듈에 정의된 함수나 변수의 이름을 사용하도록 허용하는 것이다. 다시 말해 파이썬 프로그램으로 작성된 파일(*.py, *.pyc, *.pyo)이나 C, Fortran 등으로 만든 파이썬 확장 파일(*.pyd)일 수 있다.
- 모듈 파일은 어떠한 코드로도 작성할 수 있다. 함수를 정의할 수 있고, 클래스를 정의할 수 있고 변수도 정의할 수 있다. 이렇게 정의된 내용은 다른 모듈에 의해서 호출되고 사용된다.
- 모듈은 코드들을 한 단위로 묶어 사용할 수 있게 하는 하나의 단위이다.
- 모듈은 서로 연관된 작업을 하는 코드들의 모임으로 구성된다.
- 모듈은 관리 가능하고 개념적으로 독립된 형태의 작은 단위로 작성 함으로써 코드 독립성을 유지하고 재사용성을 높일 수 있다.

1. 모듈

➤ 모듈의 종류

- 표준 모듈 : 파이썬 패키지 안에 포함된 생성 모듈
- 사용자 생성 모듈 : 사용자가 생성한 모듈
- 서드 파티(Third Party) 모듈 : 서드 파티 업체나 개인이 생성해서 제공하는 모듈

➤ 모듈을 만드는 방법은 필요한 변수, 함수, 클래스를 파이썬 파일로 작성하면 된다. (*.py)

```
# mymath.py
mypi = 3.14
def add( a, b ):
    return a + b
def area( r ):
    return mypi * r * r
```

```
>>> import mymath                # mymath 모듈을 현재의 모듈로 가져온다.
>>> dir( mymath )                # mymath에 정의된 이름 확인
>>> mymath.mypi; mymath.area( 5 )
```

1. 모듈

- 이름 공간(A)안에 있는 속성들(x, y)을 다른 공간(B)의 속성들(x, y)과 구분하기 위해 다음과 같은 형식을 사용한다.

공간.속성

- ❖ 예를 들어, X.Y.Z는 이름 공간 X에서 Y를 찾고, 이름 공간 Y에서 Z를 찾는다.

```
>>> import os
```

```
>>> os.path.join( 'a', 'b' )           # path, join은 자격이 있는 이름이다.
```

- ❖ '.' 으로 연결되지 않은, 이름 공간이 주어지지 않은 이름은 LEGB 규칙에 따라서 먼저 찾아지는 이름을 취한다. 즉, 지역 영역과 내포된 함수 영역, 전역 영역, 내장 영역 순으로 W 이름을 찾는다.

```
>>> def f():
```

```
    a = 1
```

```
    def g():
```

```
        print( b )           # b는 자격이 없는 이름이다.
```

- ❖ X, Y와 같이 이름 공간이 명백히 주어지 변수를 자격 변수(Qualified Variable)라고 하며, W와 같이 이름 공간이 명확하지 않은 변수를 무자격 변수(Unqualified Variable)라고 한다.

1. 모듈

- 파이썬은 import 한 모듈을 특별히 지정한 폴더에서 찾아 나간다. 이 폴더는 sys.path 변수에서 확인할 수 있다.

```
>>> import sys
```

```
>>> sys.path
```

- ❖ 이 폴더에서 모듈을 찾을 수 없으면 ImportError 에러가 발생한다. sys.path 변수에 직접 경로를 추가해도 검색 경로에 포함된다.

```
sys.path.append( 'c:\\myfolder' )
```

- ❖ 만일 검색 경로에 사용자가 지정한 폴더를 포함시키고 싶으면 환경 변수 PYTHONPATH에 폴더를 추가하면 된다.

1. 모듈

- 파이썬의 import에는 절대 import와 상대 import 두가지가 있다.
- 절대 import는 항상 sys.path 변수에 정해진 순서대로 폴더를 검색해서 모듈을 가져온다.
- 상대 import는 현재 모듈이 속해 있는 패키지를 기준으로 상대적인 위치에서 가져올 모듈을 찾는다.
- '.'으로 시작하지 않는 것은 모두 절대 import 이다.

```
>>> import math                # math.sin( math.pi )와 같은 형식으로 사용
>>> from math import sin, cos, pi  # 모듈에서 특정 이름만을 현재 이름 공간으로 import
>>> from math import *           # 모듈에 정의한 모든 이름을 현재 이름 공간으로 import, 모듈 수준에서만 가능
>>> import numpy as np          # 모듈 이름을 다른 이름으로 사용하고자 할 때 사용
>>> from re import sub as substitute # 모듈에 정의한 이름을 다른 이름으로 사용하고자 할 때 사용
>>> from re import sub as sub1, subn as sub2
>>> from Tkinter import (Tk, Frame, Button, Entry, Canvas, Text,
                           LEFT, DISABLED, NORMAL, RIDGE, END)
# 하나의 모듈에서 여러 개의 이름을 가져올 때 괄호를 사용할 수 있다. 여러줄에 걸쳐서 import 할 수 있다.
```

1. 모듈

- 모듈 이름이 문자열로 표현되어 있을 때, 해당 이름의 모듈을 가져오는 방법은 `__import__()` 함수를 사용하는 것이다.

```
>>> modulename = 're'
>>> re = __import__( modulename )
>>> re
```

- 한번 import된 모듈은 다른 모듈에서 가져오기가 요구되어도 이미 import된 모듈을 공유한다.
- 한번 import된 모듈은 다시 import가 되지 않는다. 만일 소스를 수정하여 다시 강제로 import를 하고 싶으면 `imp` 모듈의 `reload()` 함수를 사용한다.

```
>>> import math
>>> import imp
>>> imp.reload( math )
```

1. 모듈

- 파이썬을 실행하고 나서 한 번이라도 가져온 모든 모듈은 `sys.modules` 변수에 남아 있다.

```
>>> import sys
```

```
>>> type( sys.modules )
```

```
>>> print( '\n'.join( sys.modules.keys() ) )
```

- 새로 가져오기를 하지 않고 `sys.modules` 변수에서 모듈을 사용하는 것도 가능하다.

```
>>> sys.modules[ 'heapq' ]
```

- 파이썬 클래스나 함수는 그들이 속한 모듈 이름을 갖는 `__module__` 속성을 갖는다.

```
>>> from math import sin; sin.__module__
```

```
>>> from cmd import Cmd; Cmd.__module__
```

```
>>> sys.modules[ 'cmd' ]
```

- `__name__` 변수를 이용하면 코드가 실행되는 현재 모듈을 얻어낼 수도 있다.

```
>>> a = 1
```

```
>>> current_module = sys.modules[ __name__ ]
```

```
>>> getattr( current_module, 'a' )
```

1. 모듈

- 모듈은 모듈의 이름을 나타내는 내장 변수 `__name__`을 갖는다. 이 변수는 일반적으로 자신의 모듈 이름을 가진다.

```
>>> import math; math.__name__
```

```
# prname.py
```

```
print( __name__ )
```

```
>>> import prname
```

d:\>python prname.py -> 최상위 모듈이라면 `__name__`변수는 `__main__` 형식의 이름을 가진다.

```
# name_attr_test.py
```

```
def test():
```

```
    print( 'Python is becoming popular.' )
```

```
if __name__ == '__main__':
```

```
    test()
```

가장 먼저 실행되는 최상위 모듈일 때만 실행

```
>>> import name_attr_test
```

```
>>> name_attr_test.test() # 다른 모듈에 의해 가져올 때는 호출시에만 실행
```

- ❖ 파이썬 모듈은 독립적으로 실행 될 수 있으며, 다른 모듈에 의해 라이브러리처럼 실행 될 수도 있다. 이것은 파이썬 모듈을 독립적으로 만드는 좋은 특징이다.

2. 패키지

- 패키지(Package)는 모듈을 모아 놓은 단위이다. 관련된 여러 개의 모듈을 계층적인 몇 개의 디렉터리로 분류해서 저장하고 계층화 한다.
- 패키지 구조

Speech/	-> 최상위 패키지
__init__.py	
SignalProcessing/	-> 신호 처리 하위 패키지
__init__.py	
LPC.py	
FilterBank.py	
Recognition/	-> 음성 인식 하위 패키지
__init__.py	
Adaptation/	
__inin__.py	
ML.py	
HMM.py	
Synthesis/	-> 음성 합성 하위 패키지
__init__.py	
Tagging.py	

2. 패키지

- 각 디렉터리에는 `__init__.py` 파일이 반드시 있어야 한다. 이 파일은 패키지를 가져올 때 자동으로 실행되는 초기화 스크립트이다. 이 파일이 없으면 해당 폴더는 파이썬 패키지로 간주하지 않는다.

- `__init__.py` 파일은 패키지를 초기화하는 어떠한 파이썬 코드도 포함할 수 있다.

```
__all__ = [ 'Recognition', 'SignalProcessing', 'Synthesis' ]
```

```
__version__ = '1.2'
```

```
from . import Recognition          # 상대 import
```

```
from . import SignalProcessing
```

```
from . import Synthesis
```

- `__all__` 변수는 `from Speech import *` 문에 의해서 가져오기를 할 모듈이나 패키지 이름들을 지정한다.

```
>>> from Speech import *
```

```
>>> dir()
```

- `__init__.py` 이름 공간은 패키지 `Speech` 이름 공간이다. 즉, `__init__.py` 이름 공간에 정의하는 이름들은 패키지 `Speech` 이름 공간에 그대로 드러난다.

```
>>> Speech.__version__; Speech.__all__; dir( Speech ); Speech.Recognition
```

2. 패키지

- `from . import Recognition`과 같은 문에 의해서 하위 패키지인 `Recognition`이 `import` 된다. 이 경우 역시 `Recognition/__init__.py` 파일이 실행되어 패키지를 초기화한다.

```
__all__ = [ 'Adaptation', 'HMM' ]  
from . import Adaptation  
from . import HMM
```

- 하위 패키지들이 모두 자동으로 초기화 되었다. 따라서 다음과 같이 접근할 수 있다.

```
>>> import Speech  
>>> Speech.Recognition.HMM.Train()           # 모듈 내 함수 호출
```

2. 패키지

- 상대 import는 현재 모듈이 속해 있는 패키지를 기준으로 상대적인 접근 경로를 기술하는 것이다. 상대 import는 '.'으로 시작한다.
- python의 인수로 *.py 파이썬 프로그램이 아닌 디렉터리를 지정할 수 있다.

python my_program_dir

- 만일 my_program_dir 디렉터리 안에 __main__.py 파일이 있으면 이 파일이 자동으로 실행된다. 디렉터리가 아닌 zip 파일인 경우도 __main__.py 파일을 포함하고 있으면 __main__.py 파일을 실행한다.

3. 프로그램 배포

- distutils(Python Distribution Utility) 모듈은 파이썬 프로그램을 배포하고 설치할 때 사용하는 파이썬 표준 도구이다.
- 소스와 문서, 데이터, 스크립트 등을 한 번에 묶어서 배포할 수 있다. 배포하는 파일은 작성자의 의도대로 설치된다.
- 배포와 설치를 하려면 최상위 디렉터리에 setup.py 파일을 만들어야 한다.

```
# setup.py
from distutils.core import setup

setup( name = "spam", version = "1.0", description = "setup test", author = "hong",
      author_email = "hong@kore.co.kr", url = "http://hong.kr",

      py_modules = [ 'A', 'B', 'mymath02' ],
      packages = [ 'Speech',
                  'Speech/Recognition', 'Speech/SignalProcessing', 'Speech/Synthesis' ],
      # package_dir = { 'Speech': 'Speech2' }
      package_data = { 'Speech': [ 'images/*.jpg' ] },
      data_files = [( 'data', [ 'Speech/images/a1.jpg' ] )] )
```

3. 프로그램 배포

- `setup()` 함수에는 다양한 옵션이 있다.
- 개별적인 모듈을 지정하려면 인수 `py_modules`를 사용한다. `['A', 'B']`와 같이 확장자를 뺀 모듈 이름만 추가하면 된다.
- 파일의 위치가 패키지 안에 있을 경우는 `package.module`과 같이 지정할 수 있다.
- 패키지를 포함하려면 인수 `packages`를 사용한다.
- 하위 패키지는 루트 패키지를 지정하면 자동으로 포함되지 않으므로 각각 지정해야 한다.
- 만일 패키지 이름과 패키지가 저장된 폴더가 다를 경우에는 옵션 `package_dir`을 사용한다.
- 패키지 안에 있는 데이터 폴더 등은 자동으로 포함되지 않는다. 따라서 추가로 패키지 데이터를 지정하려면 인수 `package_data`를 다음과 같은 형식으로 사용한다.

`'패키지이름': ['패키지에_복사되어야_할_파일의_상대경로']`

- 패키지에 포함되지 않은 데이터 파일을 지정하려면 인수 `data_files`를 사용하는데 다음과 같은 형식으로 지정한다.

`'설치경로': [파일목록]`

- 설치 경로가 상대 경로라면 `sys.prefix` 변수의 위치를 기준으로 한다.

```
>>> import sys; sys.prefix
```

3. 프로그램 배포

➤ 배포판 만들기

- `setup.py` 파일을 준비하였으면 다음 명령을 이용하여 소스 배포판을 만들 수 있다.

```
python setup.py sdist
```

- 배포판은 기본적으로 `dist` 디렉터리에 `spam-1.0.zip`이나 `spam-1.0tar.gz`라는 이름으로 만들어 진다.
- 소스 배포판을 얻어서 내 시스템에 설치하려면 압축을 풀고 다음 명령을 입력한다.

```
python setup.py install
```

- 바이너리 배포판이란 소스가 아닌 실제로 설치할 결과물을 만들고 해당 결과물을 패키징하는 것을 의미한다.

```
python setup.py bdist
```

- `build` 디렉터리에 가상으로 설치하고 해당 결과물을 `dist` 디렉터리에 저장한다.
윈도우는 `spam-1.0.win32.zip` 리눅스는 `spam-1.0.linux-i686.tar.gz` 파일이 생성된다. 이 배포판은 설치할 시스템 폴더를 만드므로 올바른 위치에서 압축을 풀어야 한다.

3. 프로그램 배포

- 파이썬 소스 파일로 구성된 경우에는 소스 배포판을 이용해서 설치하기가 오히려 쉽다. 하지만, 확장 모듈이 있으면 컴파일한 결과를 배포판에 넣어 주는 편리함은 있다.

- 윈도우용 바이너리 배포판 생성 명령

```
python setup.py bdist_wininst
```

- dist 디렉터리에 spam-1.0.win32.exe파일이 만들어진다. 이 파일을 실행하면 GUI 설치 파일이 실행된다.

➤ 실행 파일 만들기

- 파이썬이 없어도 실행 가능한 형태로 배포판을 만들고 싶으면 이에 맞는 서드 파티 모듈을 사용한다.
 - py2exe 파이썬 프로그램을 윈도우용 실행 파일로 변환해 준다. 파이썬이 시스템에 설치되어 있지 않아도 실행할 수 있다.(<http://py2exe.org/>)
 - py2app 맥 OS X에 독립 실행형 응용 프로그램을 만들어 준다.
(<https://pypi.python.org/pypi/py2app>)
 - cx_Freeze 독립 실행형 실행 파일을 만들어 주는데 플랫폼에 독립적으로 실행한다.(https://anthony-tuininga.github.io/cx_Freeze/)

7. Python 클래스

Python 클래스에 대한 이해

1. Python 클래스란
2. 클래스 정의와 인스턴스 객체 생성
3. 메서드 정의와 호출
4. 클래스 멤버와 인스턴스 멤버
5. 연산자 중복
6. 장식자
7. 상속

1. Python 클래스란

- 클래스는 새로운 이름 공간을 지원하는 단위이다. 이 이름 공간에는 함수와 변수가 포함될 수 있는데 이 점은 모듈과 유사하다.
- 모듈은 파일 단위로 이름 공간을 구성하고, 클래스는 클래스 이름 공간과 클래스가 생성하는 인스턴스 이름 공간을 각각 갖는다.
- 클래스 이름 공간과 인스턴스 이름 공간은 유기적인 관계로 연결되어 있으며 상속 관계에 있는 클래스 간의 이름 공간도 유기적으로 연결되어 있다.
- 클래스를 정의하는 것은 새로운 자료형을 하나 만드는 것이고, 인스턴스는 이 자료형의 객체를 생성하는 것이다.

1. Python 클래스란

➤ 클래스는 하나의 이름 공간이다.

```
>>> class S1:
    a = 1
>>> S1.a
>>> S1.b = 2          # 클래스 이름 공간에 새로운 이름을 만든다.
>>> S1.b
>>> dir( S1 )         # 속성
>>> del S1.b          # 이름 공간 S1에서 이름 b를 삭제한다.
```

1. Python 클래스란

- 클래스 인스턴스(Class Instance)는 클래스의 실제 객체이다. 인스턴스 객체도 독자적인 이름 공간을 갖는다. 클래스는 하나 이상의 인스턴스 객체를 생성하는 자료형과 같다.

```
>>> x = S1()           # S1 클래스의 인스턴스 객체 x를 생성
>>> x.a = 10           # 클래스 인스턴스 x의 이름 공간에 이름 a 생성
>>> x.a
>>> S1.a               # 클래스 이름 공간과 인스턴스 이름 공간은 다르다.
>>> y = S1()           # 또 다른 클래스 인스턴스 생성
>>> y.a = 300          # 인스턴스 객체 y의 이름 공간에 이름 a 생성
>>> y.a
>>> x.a               # x 이름 공간의 이름 a를 확인한다.
>>> S1.a
```

1. Python 클래스란

- 클래스는 상속(Inheritance)이 가능하다. 상속받은 클래스(Subclass, 하위클래스)는 상속해 준 클래스(Superclass, 상위클래스)의 모든 속성을 자동으로 물려받는다.
- 따라서 상속받은 클래스는 물려받은 속성 이외에 추가로 필요한 개별적인 속성만을 정의하면 된다.

```
>>> class A:
    def f( self ):
        print( 'base' )

>>> class B( A ):
    pass
# 클래스 A의 속성을 모두 상속받는다.

>>> b = B()

>>> b.f()
# 클래스 A의 메서드 f가 호출된다.
```

1. Python 클래스란

- 클래스는 연산자 중복을 지원한다. 파이썬이 사용하는 모든 연산자(산술/논리 연산자, 슬라이싱, 인덱싱등)의 동작을 직접 재정의할 수 있다.
- 연산자를 중복하면 내장 자료형과 비슷한 방식으로 동작하는 클래스를 설계할 수 있다.

```
>>> class MyClass:
    def __add__( self, x ):      # __add__는 + 연산자를 중복한다.
        print( 'add {} called'.format( x ) )
        return x

>>> a = MyClass()
>>> a + 3
```

1. Python 클래스란

➤ 파이썬에서 클래스 관련 용어

- 클래스(Class) : class문으로 정의하며, 멤버와 메서드를 가지는 객체이다.
- 클래스 객체(Class Object) : 클래스와 같은 의미로 사용한다. 클래스는 특정 대상을 가리키지 않고 일반적으로 언급하기 위해서 사용하는 반면에, 클래스 객체는 어떤 클래스를 구체적으로 지정하기 위해 사용하기도 한다.
- 클래스 인스턴스(Class Instance) : 클래스를 호출하여 생성된 객체이다.
- 클래스 인스턴스 객체(Class Instance Object) : 클래스 인스턴스와 같은 의미이다. 인스턴스 객체라고 부르기도 한다.
- 멤버(Member) : 클래스 혹은 클래스 인스턴스 공간에 정의된 변수이다.
- 메서드(Method) : 클래스 공간에 정의된 함수이다.
- 속성(Attribute) : 멤버와 메서드 전체를 가리킨다. 즉, 이름 공간의 이름 전체를 의미한다.
- 상속(Inheritance) : 상위 클래스의 속성과 행동을 그대로 받아들이고 추가로 필요한 기능을 클래스에 덧붙이는 것이다. 소프트웨어의 재사용 관점에서 상속은 대단히 중요한 역할을 하며, 프로그램의 개발 시간을 단축해 준다. 다른 프로그래밍 기법과 객체지향 프로그래밍을 구분하는 중요한 특징이다. A 클래스를 상위 클래스로 하는 클래스 B를 만들면 B 'is-a' A 관계가 성립한다.

1. Python 클래스란

- 상위 클래스(Superclass) : 기반 클래스(Base class)라고 하기도 한다. 어떤 클래스의 상위에 있으며 각종 속성을 하위 클래스로 상속해 준다.
- 하위 클래스(Subclass) : 파생 클래스(Derived class)라고 하기도 한다. 상위 클래스로부터 상속받는 하위의 클래스를 말한다. 상위 클래스로부터 각종 속성을 상속받으므로 코드와 변수를 공유한다.
- 다형성(Polymorphism) : 상속 관계 내의 다른 클래스의 인스턴스들이 같은 메서드 호출에 대해 각각 다르게 반응하도록 하는 기능이다.
- 정보 은닉(Encapsulation) : 메서드와 멤버를 클래스 내에 포함하고 외부에서 접근할 수 있도록 인터페이스만을 공개한다. 그리고 다른 속성은 내부에 숨기는 것이다.
- 다중 상속(Multiple Inheritance) : 두 개 이상의 상위 클래스로부터 상속받는 것을 말한다.

2. 클래스 정의와 인스턴스 객체 생성

➤ 클래스 구조

```
class 클래스 이름( 부모 클래스명 ):
```

```
    <클래스 변수들>
```

```
    def __init__( self, [, 인수1, 인수2, ... ] ):           # 생성자
```

```
        <수행할 문장들>
```

```
    def __del__( self ):                                     # 소멸자
```

```
        <수행할 문장들>
```

```
    def 클래스 함수1( self, [, 인수1, 인수2, ... ] ):      # 클래스 메서드
```

```
        <수행할 문장들>
```

- ❖ 클래스 내부에서 선언할 수 있는 클래스 변수와 클래스 함수의 개수는 제한되어 있지 않다.

2. 클래스 정의와 인스턴스 객체 생성

➤ 클래스 정의

```
>>> class Simple:           # Header, 실행문이다. 따라서 아무 곳에서나 기술할 수 있다.
    pass                     # Body, 들여쓰기가 된 상태로 기술. pass 키워드는 아무 일도
                             # 하지 않는, 자리를 채우는 문이다.
```

```
>>> Simple
```

❖ 파이썬 클래스는 object 클래스를 기반(Base) 클래스로 한다.

```
>>> Simple.__bases__
>>> object.__dict__
```

```
>>> s1 = Simple()           # 인스턴스 객체 생성
>>> s2 = Simple()
>>> s1
```

- ❖ 인스턴스 객체를 생성하는 방법은 클래스를 호출하면 된다. 클래스를 호출하면 인스턴스 객체를 생성한 후 해당하는 참조를 반환한다.
- ❖ 각각의 인스턴스 객체도 독립적인 이름 공간을 갖는다.

2. 클래스 정의와 인스턴스 객체 생성

```
>>> s1.stack = []                # 인스턴스 s1에 stack 생성
>>> s1.stack.append( 1 )         # 값 추가
>>> s1.stack.append( 2 )
>>> s1.stack.append( 3 )
>>> s1.stack                     # 인스턴스 s1.stack 값 출력
>>> s1.stack.pop()              # 값을 읽어내고 삭제
>>> s1.stack.pop()
>>> s1.stack
>>> s2.stack                     # Error, s2에는 stack을 정의한 적이 없다.
>>> del s1.stack
```

- ❖ 동적으로 외부에서 멤버를 생성할 수 있다. 클래스는 하나의 이름 공간에 불과하다. 클래스 인스턴스 객체 s1은 클래스 Simple안에 내포된 독립적인 이름 공간을 가지며, 이 이름 공간에서는 동적으로 이름을 설정하는 것이 가능하다.

3. 메서드 정의와 호출

➤ 일반 메서드 정의와 호출

- 클래스 내부에서 메서드를 정의하는 방법은 일반 함수를 정의하는 방법과 동일하다. 다른 점은 메서드의 첫 번째 인수는 반드시 해당 클래스의 인스턴스 객체이어야 한다. 관례로 `self`란 이름으로 첫 번째 인수를 선언한다.

```
>>> class MyClass:
    def set( self, v ):          # 메서드의 첫 인수 self는 반드시 인스턴스 객체이어야 한다.
        self.value = v
    def get( self ):
        return self.value
```

- ❖ 메서드 호출하는 방법 첫 번째는 클래스를 이용하여 호출하는 것으로 언바운드 메서드 호출(Unbound Method Call)이라 부른다.

```
>>> c = MyClass()              # 인스턴스 객체 생성
>>> MyClass.set                # 메서드 확인
>>> MyClass.set( c, 'egg' )    # 언바운드 메서드 호출
>>> MyClass.get( c )
```

3. 메서드 정의와 호출

- ❖ 메서드 호출 방법 두 번째는 인스턴스 객체를 이용하여 호출하는 것으로 바운드 메서드 호출(Bound Method Call)이라 부른다. 두 번째 방법으로 메서드를 호출하면 첫 인수(self)로 인스턴스 객체가 자동으로 전달된다.

```
>>> c = MyClass()
```

```
>>> c
```

```
>>> c.set                                     # set() 메서드는 c에 바운드되어 있다.
```

- ❖ MyClass.set 메서드는 이미 인스턴스 객체 c와 연결되어 있다는 것을 알 수 있다. 즉, c.set의 set() 메서드는 이미 인스턴스 객체로 c와 연결되어 있다는 의미이다.

```
>>> c.set( 'spam' )                          # 바운드 메서드 호출
```

- ❖ 이미 인스턴스 객체로 c와 연결되어 있으므로 self를 명시적으로 전달할 필요가 없다. c.set()의 호출에서 self는 인스턴스 객체 c가 암시적으로 전달한다.

```
>>> c.get()
```

3. 메서드 정의와 호출

➤ 클래스 내부에서 메서드 호출

- 클래스 내부에서 인스턴스 멤버나 클래스 메서드를 호출할 때는 인수 self를 이용해야 한다.

```
>>> class MyClass2:
    def set( self, v ):
        self.value = v
    def get( self ):
        return self.value
    def increase( self ):
        self.value += 1          # 인스턴스 멤버 참조
        return self.get()       # 메서드 호출

>>> c2 = MyClass2()
>>> c2.set( 1 )
>>> c2.get()
>>> c2.increase()
```

- ❖ 클래스의 멤버나 메서드를 참조하려면 언제나 self를 이용하는 것을 잊지 말자!!!

3. 메서드 정의와 호출

➤ 생성자/소멸자

- 파이썬은 이름 공간을 관리하는 측면에서 동적인 특성이 강하다. 따라서 클래스의 인스턴스 객체를 생성할 때 인스턴스 멤버가 자동으로 생성되지는 않는다.

```
>>> c2 = MyClass2()          # 빈 이름 공간이 만들어진다.
>>> c2.set( 1 )              # 이름 공간에 value가 만들어 진다.
>>> c2.get()
```

- ❖ 클래스에서 사용될 멤버들은 인스턴스 객체를 생성하면서 먼저 정의하고 초기화되는 것이 일반적이다.

- 클래스는 생성자(Constructor)와 소멸자(Destructor)라 불리는 메서드를 정의할 수 있다. 생성자는 인스턴스 객체가 생성될 때 초기화를 위해 자동으로 호출되는 초기화 메서드이고, 소멸자는 인스턴스 객체를 사용하고서 메모리에서 제거할 때 자동으로 호출되는 메서드이다.

- 파이썬 클래스에서 생성자/소멸자를 위해 특별한 이름을 정해놓았다.

- ❖ 생성자 함수 이름 __init__
- ❖ 소멸자 함수 이름 __del__

3. 메서드 정의와 호출

```
>>> class MyClass3:
    def __init__( self ):          # 인스턴스 객체가 생성될 때 자동 호출
        self.value = 0
    def set( self, v ):
        self.value = v
    def get( self ):
        return self.value

>>> c3 = MyClass3()
>>> c3.get()                      # 멤버 value가 0으로 초기화되어 있다.

>>> from time import time, ctime, sleep
>>> class Life:
    def __init__( self ):          # 생성자
        self.birth = ctime(); print( 'Birthday', self.birth )
    def __del__( self ):           # 소멸자
        print( 'Deathday', ctime() )

>>> life = Life()                 # 인스턴스 객체가 생성되면 자동으로 생성자 호출
>>> del life                      # 참조 횟수가 0이 되면 소멸자가 호출되고 인스턴스 객체가
                                   # 제거된다.
```


3. 메서드 정의와 호출

- ❖ 소멸자(__del__)는 자주 정의되지는 않는다. 대부분의 메모리나 자원 관리가 자동으로 이루어지기 때문에 특별한 조치를 취하지 않아도 인스턴스 객체가 제거되면서 자원이 원상 복귀되기 때문이다.

```
>>> class Member:
    def __init__( self, name, nick, birthday ):
        self.name = name
        self.nick = nick
        self.birthday = birthday
    def getName( self ):
        return self.name
    def getNick( self ):
        return self.nick
    def getBirthday( self ):
        return self.birthday
```

```
>>> m1 = Member( 'hong', 'h', '2018-01-01' )
```

```
>>> m1.getName()
```

```
>>> m2 = Member( name = 'kim', nick = 'k', birthday = '2017-01-01' )
```

- ❖ 인스턴스 객체를 생성할 때 열거된 인수들은 생성자 인수에 전달된다. self는 결과적으로 m1과 같은 참조가 되며, 나머지는 순서대로 전달된다. 함수에서와 같은 사용 방법이 적용된다.

4. 클래스 멤버와 인스턴스 멤버

- 멤버에는 클래스 멤버(Class Member)와 인스턴스 멤버(Instance Member) 두 가지가 있다
 - 클래스 멤버는 클래스의 이름 공간에 생성된다.
 - 인스턴스 멤버는 인스턴스 객체의 이름 공간에 생성된다.
 - 클래스 멤버는 모든 인스턴스 객체에 의해서 공유된다.
 - 인스턴스 멤버는 각각의 인스턴스 객체 내에서만 참조된다.

```
>>> class Var:
    c_mem = 100                # 클래스 멤버
    def f( self ):
        self.i_mem = 200      # 인스턴스 멤버
    def g( self ):
        return self.i_mem, self.c_mem
```

- 클래스 멤버는 메서드 바깥에 정의한다. 인스턴스 멤버는 메서드 내부에서 `self`를 이용하여 정의한다.
- 클래스 내부에서 멤버를 참조할 때는 '`self.멤버명`'과 같은 형식을 사용한다.

4. 클래스 멤버와 인스턴스 멤버

- 클래스 외부에서 참조할 때는 다음과 같은 형식으로 호출할 수 있다.
 - 클래스 멤버 클래스.멤버(혹은 인스턴스 멤버)
 - 인스턴스 멤버 인스턴스.멤버
 - ❖ 인스턴스.멤버 형식을 사용할 때 인스턴스 객체의 이름 공간에 멤버가 없으면 클래스의 멤버로 인식한다.

```
>>> Var.c_mem                    # 클래스 객체를 통하여
>>> v1 = Var()
>>> v1.c_mem                    # 인스턴스 객체를 통하여
>>> v1.f()                      # 인스턴스 멤버 i_mem을 생성
>>> v2 = Var()
```

- ❖ self.c_mem 형식이나 인스턴스.멤버 형식으로 멤버를 참조할 때, 검색 순서
 - ① 먼저 인스턴스 객체의 이름 공간에서 멤버를 참조한다.
 - ② 만일 인스턴스 객체의 이름 공간에 멤버가 없으면 클래스의 멤버를 참조한다.
- ❖ 클래스 멤버는 클래스의 모든 인스턴스 객체가 공유하는 멤버이고, 인스턴스 멤버는 각각의 인스턴스 객체가 별도로 가지고 있는 멤버이다. 즉, 각 인스턴스 객체의 특성을 나타낸다고 할 수 있다.

4. 클래스 멤버와 인스턴스 멤버

```
>>> v1.c_mem          # 클래스 멤버 참조
>>> v2.c_mem          # 클래스 멤버 참조
>>> v1.c_mem = 50      # 인스턴스 객체의 이름 공간에 c_mem을 생성
>>> v1.c_mem          # 인스턴스 멤버 참조
>>> v2.c_mem          # 인스턴스 멤버가 없으므로 클래스 멤버 참조
>>> Var.c_mem         # 클래스 멤버 참조
```

- 클래스에 정의한 이름 이외의 속성을 만들 수 없게 하는 것이 필요하면 `__slots__` 속성을 이용한다. 클래스 멤버 `__slots__`는 리스트로 설정 가능한 속성의 이름을 갖는다. `__slots__` 속성에 등록되지 않은 이름은 사용할 수 없게 된다. 이 속성이 사용되는 경우에는 이름 공간을 표현하는 `__dict__`는 사용되지 않는다.

```
>>> class Person:
    __slots__ = [ 'name', 'tel' ]
>>> m1 = Person()
>>> m1.name = 'hong'          # __slots__ 속성에 등록된 속성
>>> m1.tel = '1234'          # __slots__ 속성에 등록된 속성
>>> m1.address = 'seoul'     # Error, __slots__ 속성에 등록되지 않은 속성
```

5. 연산자 중복

- 연산자 중복(Operator Overloading)은 프로그램 언어에서 지원하는 연산자에 대해 클래스가 새로운 동작을 정의하는 것이다. 파이썬에서는 내장 자료형에 사용하는 모든 연산자를 클래스 내에서 새롭게 정의할 수 있다.
- 수치 연산자 중복

1. 이항 연산자

```
>>> class MyStr:
    def __init__( self, s ):
        self.s = s
    def __truediv__( self, b ):
        return self.s.split( b )
    def __add__( self, b ):
        return self.s + b

>>> s1 = MyStr( 'a:b:c:' )
>>> s1 / ':'
>>> s1 + ':d'
```

나누기(/) 연산자 중복

더하기(+) 연산자 중복

나누기(/) 연산이 가능해졌다.

더하기(+) 연산이 가능해졌다.

- ❖ 파이썬은 모든 연산자에 대응하는 적절한 이름의 메서드가 정해져 있어서 연산자가 사용 될 때 해당 메서드로 확장된다.

5. 연산자 중복

➤ 수치 연산자 메서드

메서드	연산자
<code>__add__(self, other)</code>	<code>+</code>
<code>__sub__(self, other)</code>	<code>-</code>
<code>__mul__(self, other)</code>	<code>*</code>
<code>__truediv__(self, other)</code>	<code>/</code>
<code>__floordiv__(self, other)</code>	<code>//</code>
<code>__mod__(self, other)</code>	<code>%</code>
<code>__divmod__(self, other)</code>	<code>divmod()</code>
<code>__pow__(self, other[, modulo])</code>	<code>pow()</code> <code>**</code>
<code>__lshift__(self, other)</code>	<code><<</code>
<code>__rshift__(self, other)</code>	<code>>></code>
<code>__and__(self, other)</code>	<code>&</code>
<code>__xor__(self, other)</code>	<code>^</code>
<code>__or__(self, other)</code>	<code> </code>

5. 연산자 중복

2. 역 이항 연산자 : 피연산자의 순서가 바뀐 경우의 연산자 중복

```
>>> class MyStr:
    def __init__( self, s ):
        self.s = s
    def __truediv__( self, b ):
        return self.s.split( b )
    def __add__( self, b ):
        return self.s + b
    def __radd__( self, b ):
        return b + self.s
```

```
>>> s1 = MyStr( 'a:b:c:' )
```

```
>>> 'z:' + s1
```

❖ a + b 연산에서 a.__add__(b)를 우선 시도하고, 이것이 구현되어 있지 않으면 b.__radd__(a)를 시도한다.

5. 연산자 중복

➤ 피연산자가 바뀐 경우의 수치 연산자 메서드

메서드	연산자
<code>__radd__(self, other)</code>	<code>+</code>
<code>__rsub__(self, other)</code>	<code>-</code>
<code>__rmul__(self, other)</code>	<code>*</code>
<code>__rtruediv__(self, other)</code>	<code>/</code>
<code>__rfloordiv__(self, other)</code>	<code>//</code>
<code>__rmod__(self, other)</code>	<code>%</code>
<code>__rdivmod__(self, other)</code>	<code>divmod()</code>
<code>__rpow__(self, other[, modulo])</code>	<code>pow()</code> <code>**</code>
<code>__rlshift__(self, other)</code>	<code><<</code>
<code>__rrshift__(self, other)</code>	<code>>></code>
<code>__rand__(self, other)</code>	<code>&</code>
<code>__rxor__(self, other)</code>	<code>^</code>
<code>__ror__(self, other)</code>	<code> </code>

5. 연산자 중복

3. 확장 산술 연산자 메서드 : `s += b` -> `a.__iadd__(b)`

메서드	연산자
<code>__iadd__(self, other)</code>	<code>+=</code>
<code>__isub__(self, other)</code>	<code>-=</code>
<code>__imul__(self, other)</code>	<code>*=</code>
<code>__itruediv__(self, other)</code>	<code>/=</code>
<code>__ifloordiv__(self, other)</code>	<code>//=</code>
<code>__imod__(self, other)</code>	<code>%=</code>
<code>__ipow__(self, other[, modulo])</code>	<code>**=</code>
<code>__ilshift__(self, other)</code>	<code><<=</code>
<code>__irshift__(self, other)</code>	<code>>>=</code>
<code>__iand__(self, other)</code>	<code>&=</code>
<code>__ixor__(self, other)</code>	<code>^=</code>
<code>__ior__(self, other)</code>	<code> =</code>

5. 연산자 중복

4. 단항 연산자와 형변환 연산자 메서드

`-s` \rightarrow `s.__neg__()`

`+s` \rightarrow `s.__pos__()`

`abs(s)` \rightarrow `s.__abs__()`

메서드	연산자
<code>__neg__(self)</code>	-
<code>__pos__(self)</code>	+
<code>__abs__(self)</code>	<code>abs()</code>
<code>__invert__(self)</code>	~(비트 반전)

5. 연산자 중복

5. 기타 형변환 메서드

메서드	연산자
<code>__complex__(self)</code>	<code>complex()</code>
<code>__int__(self)</code>	<code>int()</code>
<code>__float__(self)</code>	<code>float()</code>
<code>__round__(self)</code>	<code>round()</code>
<code>__index__(self)</code>	<code>operator.index()</code>

❖ 객체를 인수로 하여 함수 형태로 호출되며, 이에 따라 적절한 값을 반환해 주어야 한다.

`complex(s)` \rightarrow `s.__complex__()`

`int(s)` \rightarrow `s.__int__()`

`float(s)` \rightarrow `s.__float__()`

`round(s)` \rightarrow `s.__round__()`

`operator.index(s)` \rightarrow `s.__index__()`

5. 연산자 중복

- ❖ `a.__index()` 메서드는 `operator.index(a)`에 의해 호출되며, `a`를 정수로 변환할 수 있는지 확인해서 정수 값을 반환한다. 만일 `a`가 정수가 아니면 `TypeError`가 발행한다. 인덱스로 사용할 자료형은 정수로만 표현해야 하는데 이를 검사하기 위해서 `__index__()` 메서드가 존재한다.

```
>>> operator.index( 5 )           # 정수는 인덱스로 사용할 수 있다.
>>> operator.index( 5.5 )        # float는 인덱스로 사용할 수 없다.
```

- ❖ `bin()`, `hex()`, `oct()`와 같은 함수들은 정수 인수가 필요하며 내부적으로 `__index__()` 메서드를 사용한다.

```
operator.index( s )               -> s.__index__()
bin( s )                         -> bin( s.__index__() )
hex( s )                         -> hex( s.__index__() )
oct( s )                         -> oct( s.__index__() )
```

5. 연산자 중복

```
>>> class Index:
    def __index__( self ):
        print( '__index__ called' )
        return 3

>>> l = [ 1, 2, 3, 4, 5 ]
>>> i = Index()
>>> l[ i: ]
>>> bin( i )
>>> oct( i )
>>> hex( i )
```

5. 연산자 중복

- 컨테이너 자료형(시퀀스 자료형 + 매핑 자료형)의 연산자 중복
 - 기본적으로 `__len__()`, `__contains__()`, `__getitem__()`, `__setitem__()`, `__delitem__()` 메서드를 구현해야 한다.
 - 시퀀스형이면서 변경 가능한 자료형을 만들겠다면 `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()`, `sort()` 메서드 등을 구현하면 된다.
 - 사전과 같은 매핑 자료형은 `keys()`, `values()`, `items()`, `get()`, `clear()`, `copy()`, `setdefault()`, `pop()`, `popitem()`, `update()` 같은 메서드를 구현하면 된다.
 - 산술 연산으로는 `__add__()`, `__radd__()`, `__iadd__()`, `__mul__()`, `__rmul__()`, `__imul__()` 메서드 등을 구현하면 된다.
 - 반복자를 지원하려면 `__iter__()` 메서드를 구현한다.

5. 연산자 중복

➤ 시퀀스 자료형 메서드

메서드	연산자
<code>object.__len__(self)</code>	<code>len(object)</code>
<code>object.__contains__(self, item)</code>	<code>item in object</code>
<code>object.__getitem__(self, key)</code>	<code>object[key]</code>
<code>object.__setitem__(self, key, value)</code>	<code>object[key] = value</code>
<code>object.__delitem__(self, key)</code>	<code>del object[key]</code>

5. 연산자 중복

1. 인덱싱 : 인덱싱은 시퀀스 자료형에서 순서에 의해서 데이터에 접근하기 위한 방법을 제공한다. 기본적으로는 `__getitem__()` 메서드를 정의해야 한다.

```
>>> class Square:
    def __init__( self, end ):
        self.end = end
    def __len__( self ):
        return self.end
    def __getitem__( self, k ):
        if type( k ) != int:
            raise TypeError( '...' )
        if k < 0 or self.end <= k:
            raise IndexError( 'index {} out of range'.format( k ) )
        return k * k

>>> s1 = Square( 10 )
>>> len( s1 )           # s1.__len__()
>>> s1[ 4 ]             # s1.__getitem__( 4 )
>>> s1.__getitem__( 4 ) # s1[ 4 ]
>>> s1[ 20 ]            # Error
>>> s1[ 'a' ]           # Error
```


5. 연산자 중복

```
>>> for x in s1:
```

```
    print( x, end = ' ' )
```

- ❖ for 문은 인스턴스 객체 s1의 `__getitem__()` 메서드를 0부터 호출하기 시작한다. 인스턴스 객체 s1은 제한된 범위 내에서 시퀀스형 객체로서의 역할을 충실히 수행한다.

```
>>> list( s1 )
```

```
>>> tuple( s1 )
```

- ❖ `__getitem__()` 메서드만 정의되어 있으면, 다른 시퀀스 자료형으로 변환하는 것이 가능하다.
- ❖ 이와 같은 방식으로 연산을 지원하면 내장 자료형과 사용법이 같은 일관된 연산을 적용할 수 있고, 이것은 통일성과 편리함을 가져다준다.
- ❖ 사용자가 정의한 클래스가 기존 자료형의 코딩 스타일을 그대로 따르므로 일관된 코딩 스타일을 유지할 수가 있다.
- ❖ `__getitem__()` 메서드가 치환 연산자 오른쪽에서 인덱싱에 의해서 호출되는 메서드라면 `__setitem__()` 메서드는 치환 연산자 왼쪽에서 호출되는 메서드이다. `__delitem__()` 메서드는 `del` 을 사용시 호출된다.

5. 연산자 중복

2. 슬라이싱 : 슬라이싱은 인덱싱에서와 같이 `__getitem__()`, `__setitem__()`, `__delitem__()` 메서드를 사용하지만 인수가 아닌 slice 객체를 전달한다.

- slice 객체는 start와 step, stop 세 개의 멤버를 가지는 단순한 객체로 이해하면 된다.

```
slice( [start, ] stop [, step] )
```

```
>>> s = slice( 1, 10, 2 )
```

```
>>> s
```

```
>>> type( s )
```

```
>>> s.start, s.stop, s.step
```

```
>>> slice( 10 )
```

인수가 생략되면 None 객체를 기본값으로 가진다.

```
>>> slice( 1, 10 )
```

```
>>> slice( 1, 10, 3 )
```

- ❖ 슬라이싱 `m[1:5]`는 `m.__getitem__(slice(1, 5))`를 호출한다. 즉, 인덱싱의 정수 인덱스 대신에 slice 객체가 범위를 나타내는데 사용된다. 확장 슬라이싱 `m[1:10:2]`는 `m.__getitem__(slice(1, 10, 2))`를 호출한다.

5. 연산자 중복

```
>>> class Square:
    def __init__( self, end ):
        self.end = end
    def __len__( self ):
        return self.end
    def __getitem__( self, k ):
        if type( k ) == slice:           # slice 자료형인가?
            start = k.start or 0
            stop = k.stop or self.end
            step = k.step or 1
            return map( self.__getitem__, range( start, stop, step ) )
        elif type( k ) == int:           # 인덱싱
            if k < 0 or self.end <= k:
                raise IndexError( 'index {} out of range'.format( k ) )
            return k * k
        else:
            raise TypeError( '...' )
```

5. 연산자 중복

```
>>> s = Square( 10 )
>>> s[ 4 ]                # 인덱싱
>>> list( s[ 1:5 ] )      # 슬라이싱
>>> list( s[ 1:10:2 ] )   # 간격은 2로
>>> list( s[ : ] )        # 전체 범위
```

- ❖ `__getitem__()` 메서드 정의에서 `k`의 자료형이 `slice` 형인지 검사해서 참이면 슬라이싱을, 아니면 인덱싱을 적용한다. 슬라이싱 부분에서 `start`, `stop`, `step`을 별도의 지역 변수에 치환한 이유는 `range()` 함수가 정수 인수만을 요구하기 때문이다. 최종적으로 `map()` 함수에 의해서 인덱스 값의 제공에 대한 리스트를 반환한다.

5. 연산자 중복

3. 매핑 자료형 : 매핑 자료형에서 `object.__getitem__(self, key)` 등의 메서드의 `key`는 사전의 키로 사용할 수 있는 임의의 객체가 될 수 있다. 만일 `key`에 대응하는 값을 찾을 수 없으면 `KeyError` 를 발생시킨다.

```
>>> class MyDict:
    def __init__( self ):
        self.d = {}
    def __getitem__( self, k ):
        return self.d[ k ]
    def __setitem__( self, k, v ):
        self.d[ k ] = v
    def __len__( self ):
        return len( self.d )

>>> m = MyDict()                                # __init__()
>>> m[ 'day' ] = 'light'                         # m.__setitem__( 'day', 'light' )
>>> m.__setitem__( 'night', 'darkness' )         # m[ 'night' ] = 'darkness'
>>> m[ 'day' ]; m[ 'night' ]                     # m.__getitem__( 'night' )
>>> len( m )                                     # __len__()
```

5. 연산자 중복

➤ 문자열 변환 연산

- 인스턴스 객체를 `print()` 함수로 출력할 때 내가 원하는 형식으로 출력하거나, 인스턴스 객체를 사람이 읽기 좋은 형태로 변환하려면 문자열로 변환하는 기능이 필요하다.

1. 문자열로의 변환 : `__str__()`과 `__repr__()` 메서드

인스턴스 객체를 문자열로 변환하는 메서드는 `__str__()`과 `__repr__()` 두 가지이다. 이 두 메서드는 호출되는 시점이 다르다.

```
>>> class StringRepr:
    def __repr__( self ):
        return 'repr called'
    def __str__( self ):
        return 'str called'

>>> s = StringRepr()
>>> print( s )
>>> str( s )
>>> repr( s )
```

5. 연산자 중복

- ❖ `print()` 함수와 `str()` 함수에 의해서 `__str__()` 메서드가 호출되며, `repr()` 함수에 의해서 `__repr__()` 메서드가 호출된다. `__repr__()` 메서드의 목적은 객체를 대표해서 유일하게 표현할 수 있는 문자열을 만들어 내는 것이다. 즉, 다른 객체의 출력과 혼동되지 않는 모양으로 표현해야 한다는 의미이다.

```
>>> repr( 2 )
```

```
>>> repr( '2' )
```

```
>>> repr( 'abc' )           # 문자열 'abc'에 대한 repr 문자열
```

```
>>> repr( [ 1, 2, 3 ] )     # 리스트 [ 1, 2, 3 ]에 대한 repr 문자열
```

- ❖ `__str__()` 메서드의 목적은 사용자가 읽기 편한 형태의 표현으로 출력한다.

```
>>> str( 2 )
```

```
>>> str( '2' )
```

- ❖ 컨테이너 자료형(리스트와 사전 등)의 `__str__()` 메서드는 내부 객체의 `__repr__()` 메서드를 사용한다.

```
>>> l = [ 2, '2' ]
```

```
>>> str( l )
```

```
>>> repr( l )
```

```
>>> str( l ) == repr( l )
```

5. 연산자 중복

- ❖ 만일 `__str__()` 메서드를 호출할 상황에서 `__str__()` 메서드가 정의되어 있지 않으면 `__repr__()` 메서드가 대신 호출된다.

```
>>> class StringRepr:
    def __repr__( self ):
        return 'repr called'

>>> s = StringRepr()
>>> str( s )
>>> repr( s )
```

- ❖ 그러나 `__repr__()` 메서드가 정의되어 있지 않은 경우에 `__str__()` 메서드가 `__repr__()` 메서드를 대신하지 않는다.

```
>>> class StringRepr:
    def __str__( self ):
        return 'str called'

>>> s = StringRepr()
>>> str( s )
>>> repr( s )
```


5. 연산자 중복

2. 바이트로의 변환 : `__bytes__()` 메서드

문자열이 아닌 바이트 자료형으로 변환하려면 `__bytes__()` 메서드를 사용한다. `b.__bytes__()` 메서드는 `bytes(b)` 함수에 의해 호출된다.

```
>>> class BytesRepr:
    def __bytes__( self ):
        return 'bytes called'.encode( 'utf-8' )

>>> b = BytesRepr()
>>> bytes( b )
```

3. 서식 기호 새로 지정하기 : `__format__()` 메서드

`__format__()` 메서드는 `format()` 함수나 문자열의 `format()` 메서드에 의해서 호출된다.

```
>>> x = 10
>>> format( x, "0" )           # x.__format__( "0" )
>>> "x:{:0}".format( x )      # x.__format__( "0" )
```

5. 연산자 중복

- ❖ `__format__()` 메서드는 `format()` 함수나 문자열의 `format()` 메서드에 의해서 호출된다. 변환 기호가 요구될 때는 `__format__()` 메서드가 호출된다.

```
>>> class MyStr:
```

```
    def __init__( self, s ):
```

```
        self.s = s
```

```
    def __format__( self, fmt ):
```

```
        print( fmt )
```

서식 문자열을 확인한다.

```
        if fmt[ 0 ] == 'u':
```

u이면 대문자로 변환한다.

```
            s = self.s.upper()
```

```
            fmt = fmt[ 1: ]
```

```
        elif fmt[ 0 ] == 'l':
```

l이면 소문자로 변환한다.

```
            s = self.s.lower()
```

```
            fmt = fmt[ 1: ]
```

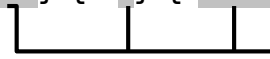
```
        else:
```

```
            s = str( self.s )
```

```
        return s.__format__( fmt )
```

```
>>> s = MyStr( 'Hello' )
```

```
>>> print( '{0:u^20} {0:l} {0:*^20}'.format( s ) )
```

 `__format__` 메서드의 인수로 입력되며, 반환 값으로 치환된다.

5. 연산자 중복

➤ 진리값과 비교 연산

1. `__bool__()` 메서드

클래스 인스턴스의 진리값은 `__bool__()` 메서드의 반환 값으로 결정된다. 만일 이 메서드가 정의되어 있지 않으면 `__len__()` 메서드를 호출한 결과가 0 이면 False로 간주하고 아니면 True로 간주한다. 만일, `__len__()` 과 `__bool__()` 메서드 모두가 정의되어 있지 않으면 모든 인스턴스는 True가 된다.

```
>>> class Truth:
    def __init__( self, num ):
        self.num = num
    def __bool__( self ):
        return self.num != 0

>>> bool( Truth( 0 ) )
>>> bool( Truth( 3 ) )
```

5. 연산자 중복

1. 비교 연산 : 모든 비교 연산은 중복이 가능하도록 메서드 이름이 준비됨

연산자	메서드
<	object.__lt__(self, other)
<=	object.__le__(self, other)
>	object.__gt__(self, other)
>=	object.__ge__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)

- ❖ $x < y$ 는 $x.__lt__(y)$ 메서드로 확장되며, $x \leq y$ 는 $x.__le__(y)$ 메서드로 확장된다. 다른 연산자도 같은 방식으로 적용된다. `__eq__()`와 `__ne__()` 메서드는 각자의 논리로 적용될 수 있다. 즉, $o == other$ 이 참이라고 해서 $o != other$ 가 거짓이 아닐 수도 있다는 것이다. 하지만, `__ne__()` 메서드가 정의되어 있지 않고, `__eq__()` 메서드만 정의되어 있을 경우 $o != other$ 는 `not(o == other)`의 논리가 적용된다.

5. 연산자 중복

```
>>> class Compare:
    def __init__( self, n ):
        self.n = n
    def __eq__( self, o ):
        print( '__eq__ called' )
        return self.n == 0
    def __lt__( self, o ):
        print( '__lt__ called' )
        return self.n < 0
    def __le__( self, o ):
        print( '__le__ called' )
        return self.n <= 0

>>> c = Compare( 10 )
>>> c < 10                # __lt__() 메서드
>>> c <= 10               # __le__() 메서드
>>> c > 10                # Error, __gt__가 정의되어 있지 않다.
>>> c == 10              # __eq__() 메서드
>>> c != 10              # __ne__() 메서드나 not __eq__() 메서드 결과
```

5. 연산자 중복

➤ 해시 값에 접근하기 : `__hash__()` 메서드

- 해시 값을 돌려주는 내장 함수 `hash(m)`가 호출될 때 `m.__hash__()` 메서드가 호출된다. `hash()` 함수를 사용한 예로, 사전은 (키, 값)쌍을 저장할 때 키에 대한 `hash()` 함수의 호출 결과를 값을 저장하기 위한 해시 키로 사용한다. `__hash__()` 메서드는 정수를 반환해야 한다. 이 메서드를 정의한 클래스는 `__eq__()` 메서드로 함께 정의해야 해시 가능한 객체로 취급한다.

```
>>> class Obj:
    def __init__( self, a, b ):
        self.a = a
        self.b = b
    def _key( self ):
        return ( self.a, self.b )
    def __eq__( self, o ):
        return self._key() == o._key()
    def __hash__( self ):
        return hash( self._key() )
```

5. 연산자 중복

```
>>> o1 = Obj( 1, 2 )
>>> o2 = Obj( 3, 4 )
>>> hash( o1 )
>>> hash( o2 )
>>> d = { o1:1 o2:2 }
```

- ❖ 해시 키는 변경이 가능해서는 안 된다. 만일 변경 가능한 자료형으로 클래스를 정의하면 hash() 함수를 호출할 때 TypeError 예러를 반환해서 해시 키로 사용할 수 없도록 해야 한다.

```
>>> class Obj2:
    def __init__( self, a, b ):
        self.a = a
        self.b = b
    def __hash__( self ):
        raise TypeError( 'not proper type' )

>>> o1 = Obj( 1, 2 )
>>> d = { o1:1 }                                # 키로 사용할 수 없다.
```

5. 연산자 중복

➤ 속성 값에 접근하기

인스턴스 객체의 속성을 다루는 메서드

메서드	설명
<code>__getattr__(self, name)</code>	정의되어 있지 않은 속성을 참조할 때, 이 메서드가 호출된다. 속성 이름 name은 문자열이다.
<code>__getattribute__(self, name)</code>	<code>__getattr__()</code> 메서드와 같으나 속성이 정의되어 있어도 호출된다.
<code>__setattr__(self, name, value)</code>	<code>self.name = value</code> 와 같이 속성에 치환(대입)이 일어날 때 호출된다.
<code>__delattr__(self, name)</code>	<code>del self.name</code> 에 의해서 호출된다.

1. `__getattr__()`와 `__getattribute__()` 메서드

인스턴스 객체에 대한 일반적인 접근 방법인 `obj.attr()` 메서드는 `getattr(obj, 'attr')`로 수행한다.

- `__getattr__()` 메서드는 이름 공간에 정의되지 않는 이름에 접근할 때 호출되며 이에 대해서 처리를 할 수 있다.

5. 연산자 중복

```
>>> class GetAttr1( object ):
    def __getattr__( self, x ):
        print( '__getattr__', x )
        if x == 'test':
            return 10
        raise AttributeError

>>> g1 = GetAttr1()
>>> g1.c = 10
>>> g1.c          # 정의된 이름을 호출
>>> g1.a          # 정의되지 않은 이름을 호출
>>> g1.test       # 정의되지 않았지만 준비된 이름
```

- `__getattr__()` 메서드는 이름 정의 여부에 관계없이 모든 속성에 접근하면 호출된다. 따라서 `__getattr__()` 메서드는 호출되는 이름 전체에 대한 제어권을 얻어낸다.

5. 연산자 중복

```
>>> class GetAttr2( object ):
    def __getattr__( self, x ):
        print( '__getattr__ called..', x )
        return object.__getattr__( self, x )

>>> g2 = GetAttr2()
>>> g2.c = 10
>>> g2.c          # 정의된 이름 호출
>>> g2.a          # 정의되지 않은 이름 호출
```

5. 연산자 중복

```
>>> class GetAttr3( object ):
    def __getattr__( self, x ):
        print( '__getattr__', x )
        raise AttributeError
    def __getattribute__( self, x ):
        print( '__getattribute__ called..', x )
        return object.__getattribute__( self, x )

>>> g3 = GetAttr3()
>>> g3.c = 10
>>> g3.c                # 정의된 속성에 접근
>>> g3.a                # 정의되지 않은 속성에 접근
```

- ❖ 주의할 점은 `self.__getattribute__(x)`와 같이 호출해서는 안 된다. 재귀적으로 자기 자신을 무한히 호출하게 되므로 상위 클래스를 통해서 `object.__getattribute__(self, x)`와 같은 식으로 접근해야 한다.

5. 연산자 중복

2. `__setattr__()`와 `__delattr__()` 메서드

인스턴스 객체에서 속성을 설정할 때는 `__setattr__()` 메서드를, 속성을 삭제할 때는 `__delattr__()` 메서드를 사용한다.

`obj.x = o`는 `setattr(obj, 'x', o)`로 수행되며 `obj.__setattr__('x', o)`를 호출하고, `del obj.x`는 `delattr(obj, 'x')`로 수행되며 `obj.__delattr__('x')`를 호출한다.

```
>>> class Attr:
    def __setattr__( self, name, value ):
        print( '__setattr__((s)=%s called' % ( name, value ) )
        object.__setattr__( self, name, value )
    def __delattr__( self, name ):
        print( '__delattr__((s) called' % name )
        object.__delattr__( self, name )

>>> a = Attr()
>>> a.x = 10
>>> del a.x
```

5. 연산자 중복

➤ 인스턴스 객체를 호출하기

1. `__call__()` 메서드

어떤 클래스 인스턴스가 `__call__()` 메서드를 가지고 있으면, 해당 인스턴스 객체는 함수와 같은 모양으로 호출할 수 있다. 인스턴스 객체 `x`에 대해 다음과 같이 확장된다.

`x(a1, a2, a3)` \rightarrow `x.__call__(a1, a2, a3)`

- ❖ 클래스 `Factorial`은 고속 처리를 위하여 기억 기법(Memorization Technique)을 사용한다. 한 번 계산된 팩토리얼 값은 인스턴스 객체의 `cache` 멤버에 저장되어 있다가 필요할 때 다시 사용한다. 팩토리얼 계산은 인스턴스 객체의 `__call__()` 메서드를 호출하여 이루어진다.

5. 연산자 중복

```
>>> class Factorial:
    def __init__( self ):
        self.cache = {}
    def __call__( self, n ):
        if n not in self.cache:
            if n == 0:
                self.cache[ n ] = 1
            else:
                self.cache[ n ] = n * self.__call__( n - 1 )
        return self.cache[ n ]

>>> fact = Factorial()
>>> for i in range( 10 ):
    print( '{} != {}'.format( i, fact( i ) ) )
```

5. 연산자 중복

2. 호출 가능한지 확인하기

어떤 객체가 호출한지 알아보려면 `collections.Callable`의 인스턴스 객체인지 확인한다.

```
>>> def f():  
    pass  
  
>>> isinstance( f, collections.Callable )# 함수 객체를 확인한다.  
>>> fact = Factorial()  
>>> isinstance( fact, collections.Callable )
```

5. 연산자 중복

➤ 인스턴스 객체 생성하기 : `__new__()` 메서드

클래스의 `__init__()` 메서드는 객체가 생성된 이후에 객체를 초기화하기 위해 호출되는 메서드이다. 반면 `__new__()`는 객체의 생성을 담당하는 메서드로 `__new__()` 메서드에 의해서 생성된 객체가 `__init__()` 메서드에 의해서 초기화된다. `__new__()` 메서드는 `object` 클래스의 `__new__()` 메서드를 통해서 인스턴스 객체를 생성해야 한다.

```
>>> class NewTest:
    def __new__( cls, *args, **kw ):
        # cls는 NewTest
        print( "__new__ called", cls )
        instance = object.__new__( cls )
        # 인스턴스 객체를 생성한다.
        return instance
    def __init__( self, *args, **kw ):
        # self는 생성된 인스턴스 객체이다.
        print( "__init__ called", self )
>>> t = NewTest()
```


5. 연산자 중복

- ❖ 만일 `__new__()` 메서드가 인스턴스 객체를 반환하면 `__init__()` 메서드가 호출되지만, 그렇지 않으면 `__init__()` 메서드는 호출되지 않는다.

```
>>> class Super:
    def __new__( cls, *args, **kw ):
        obj = object.__new__( cls )
        obj.data = []
        return obj
>>> class Sub( Super ):
    def __init__( self, name ):
        self.name = name
>>> s = Sub( 'hong' )
>>> s.name
>>> s.data
```

- ❖ 위 예는 `__new__()` 메서드를 사용하여 멤버 값을 초기화하는 예로, 멤버값의 초기화는 일반적으로 `__init__()` 메서드에서 이루어지지만 상위 클래스의 `__init__()` 메서드를 명시적으로 호출하지 않으면 상위 클래스의 `__init__()` 메서드는 실행되지 않는다. `__new__()` 메서드를 사용하여 상위 클래스의 `__init__()` 메서드 호출 여부와 관계없이 멤버 값의 초기화를 수행 예이다.

5. 연산자 중복

```
>>> class Singleton:
    __instance = None          # 유일한 객체를 저장하기 위한 클래스 변수
    def __new__( cls, *args, **kw ):
        if cls.__instance is None:
            cls.__instance = object.__new__( cls )
        return cls.__instance

>>> class Sub( Singleton ):
    pass

>>> s1 = Sub(); s2 = Sub()

>>> s1 is s2
```

- ❖ 싱글톤(Singleton)이란 인스턴스 객체를 오직 하나만 생성해 내는 클래스를 의미한다. 유일하게 하나만 시스템에 존재해야 하는 객체를 정의할 때 유용하다.

6. 장식자

- 장식자(Decorator)는 함수를 인수로 받는 함수 클로저(Function Closure)이다.
- 함수 클로저를 간단히 말하면 함수 코드와 그 함수의 변수 참조 영역을 묶은 객체라고 할 수 있다. 함수를 인스턴스화하는데 유용하다. 따라서 장식자에 유용하게 활용된다.

```
>>> def wrapper( func ):
    def wrapped_func():
        print( 'before..' )
        func()
        print( 'after..' )
    return wrapped_func
# 장식자는 함수 객체를 인수로 받는다.
# 내부에서는 wrapper() 함수를 정의한다.
# 원래 함수 이전에 실행되어야 할 코드이다.
# 원래 함수이다.
# 원래 함수 이후에 실행되어야 할 코드이다.
# 원래 함수를 감싼 함수 클로저를 반환한다.

>>> def myfunc():
    print( 'I am here' )
# 원래 함수를 정의한다.

>>> myfunc()
# 원래 함수 실행

>>> myfunc = wrapper( myfunc )
# 장식자에 함수를 전달한다.

>>> myfunc()
# 장식된 함수를 실행한다.
```

6. 장식자

- ❖ 앞의 예에서 `wrapper()` 함수는 장식자이다. 함수를 인수로 받으며 함수 클로저를 반환한다. `myfunc = wrapper(myfunc)`로 반환된(장식된) 함수는 실행할 때 마다 실제로는 `wrapped_func()` 함수가 실행된다. 이것이 장식자이다.
- ❖ 장식자를 좀 더 간단히 표현하는 방법이 있는데 `@wrapper` 형식의 선언을 함수 앞에 하는 것이다.

<code>@wrapper</code>		<code>def f():</code>
<code>def f():</code>	<code>-></code>	<code>~생략</code>
<code>~생략</code>		<code>f = wrapper(f)</code>

```
>>> @wrapper                # 장식자이다.
def myfunc2():              # myfunc2 = wrapper( myfunc2 )
    print( 'I am here 2..' )
>>> myfunc2()               # 장식된 함수를 실행한다.
```

6. 장식자

❖ 장식자를 사용하는 대표적인 예는 정적 메서드와 클래스 메서드이다.

```
>>> class D:
    @staticmethod                      # add를 static method로
    def add( x, y ):
        return x + y

>>> class D:
    def add( x, y ):
        return x + y
    add = staticmethod( add )

>>> D.add( 2, 4 )
```

- ❖ 장식자는 다양한 경우에 유용하게 활용된다. 예를 들어, 외부에서 제공되는 수정할 수 없는 라이브러리의 행동 특성을 변경하기 위해서, 소스를 건드리지 않고 디버깅 정보를 출력하기 위해서, 같은 방식으로 여러 함수를 확장하기 위해서 등등이 될 수 있다.
- ❖ 파이썬에서 표준으로 제공되는 장식자들은 거의 없다. 파이썬에서는 단지 기능만을 제공할 뿐 무한한 사용 가능성에 대해서는 사용자에게 일임하고 있다. 마치 클래스나 함수를 만드는 기능을 제공하는 것과 마찬가지로이다. 직접 만들어 보거나 여러 유용한 장식자가 공개되고 있으므로 필요에 따라 검색해 보면 좋을 것이다.

6. 장식자

- 정적 메서드(Static Method)란 인스턴스 객체를 생성하지 않고도, 혹은 인스턴스 객체를 이용하지 않고도 클래스를 이용하여 직접 호출할 수 있는 메서드 이다.
- 일반 메서드는 첫 번째 인수로 인스턴스 객체를 반드시 전달해야 하지만 정적 메서드는 일반 함수와 동일한 방식으로 호출한다.
- 정적 메서드는 일반 메서드와 달리 첫 인수로 self를 받지 않는다. 필요한 만큼의 인수를 선언하면 된다.
- @staticmethod를 메서드 앞에 장식하면 정적 메서드가 된다.

```
>>> class D:
    @staticmethod                # add를 정적 메서드로
    def add( x, y ):
        return x + y

>>> D.add( 2, 3 )                # 인스턴스 객체 없이 클래스에서 직접 호출한다.
>>> d = D()
>>> d.add( 2, 3 )                # 인스턴스 객체를 통해서도 호출할 수 있다.
```

6. 장식자

- ❖ 정적메서드는 인스턴스 객체와 관계없이 실행되어야 하므로 인스턴스 변수를 참조할 수 없다. 대신 클래스 멤버는 참조할 수 있다.

```
>>> class E:
    acc = 0
    @staticmethod
    def accumulate( v ):
        E.acc += v
        return E.acc                                # 클래스 멤버

>>> E.accumulate( 10 )
>>> E.accumulate( 20 )
>>> E.acc
```

6. 장식자

- 클래스 메서드(Class Method)는 첫 인수로 클래스 객체를 전달받는다. 정적 메서드와 같이 클래스를 통하여 직접 호출하는 것이 일반적이지만 첫 인수로 클래스 객체가 전달되는 것이 다르다.
- 클래스 메서드는 @classmethod 장식자에 의해서 선언된다.

```
>>> class CM:
    acc = 0
    @classmethod
    def accumulate( cls, v ):          # 클래스 메서드
        cls.acc += v
        return cls.acc

>>> CM.accumulate( 10 )
>>> CM.accumulate( 20 )
>>> CM.acc

>>> c = CM()
>>> c.accumulate( 5 )                # 인스턴스 객체를 통한 호출이 가능하다.
>>> CM.acc
>>> c.__class__ is CM                 # c.__class__와 CM이 같은 객체인가?
```


6. 장식자

- Property 속성이란 멤버 변수와 같은 접근 방식을 사용하지만 실제로는 메서드의 호출로 처리되는 속성을 말한다. 즉, 메서드로 정의되어 있지만 호출은 멤버 변수를 사용하는 것처럼 한다.
- property 함수를 통해서 속성을 정의할 수 있는데, 이 함수는 변수에 값을 저장하는 메서드(fset), 읽는 메서드(fget), 삭제하는 메서드(fdel)를 지정하고 관련 연산이 이들 메서드를 통해서 이루어지는 객체를 생성한다.

```
property( fget = None, fset = None, fdel = None, doc = None )
```

- ❖ fget은 값을 읽을 때, fset은 값을 쓸 때, fdel은 값을 삭제할 때 자동으로 호출되는 메서드이다.

6. 장식자

```
>>> class PropertyClass:
    def get_deg( self ):
        return self.__deg
    def set_deg( self, d ):
        return self.__deg = d % 360
    deg = property( get_deg, set_deg )
```

```
>>> p = PropertyClass()
```

```
>>> p.deg = 390; p.deg
```

```
>>> p.deg = -370; p.deg
```

❖ 장식자를 이용한 방법

```
>>> class PropertyClass:
    @property                                # getter 메서드를 등록한다.
    def get_deg( self ):
        return self.__deg
    @property                                # deg의 setter 메서드를 등록한다.
    def set_deg( self, d ):
        return self.__deg = d % 360
```

```
>>> p = PropertyClass()
```

```
>>> p.deg = 390; p.deg
```

```
>>> p.deg = -370; p.deg
```

7. 상속

- 상속(Inheritance)은 클래스가 갖는 중요한 특징이다. 상속이 중요한 이유는 재사용성에 있다. 상속받은 클래스는 상속해 준 클래스의 속성을 사용할 수 있으므로, 추가로 필요한 기능만을 정의 하거나, 기존의 기능을 변경해서 새로운 클래스를 만들면 된다.
- 클래스 A에서 상속된 클래스 B가 있다고 할때, 클래스 A를 기반(Base) 클래스, 부모(Parent) 클래스 또는 상위(Super) 클래스라고 하며, 클래스 B를 파생(Derived) 클래스, 자식(Child) 클래스 또는 하위(Sub) 클래스라 한다.
- 상속 관계는 is-a 관계를 갖는다.
 - ❖ '사람은 포유류이고 포유류는 동물이다.'에서 사람->포유류->동물의 관계가 성립한다. 동물은 포유류의 상위 클래스이고 포유류는 사람의 상위 클래스이다.
 - ❖ is-a 관계는 두 개의 클래스의 계층적인 관계를 따질 때 사용한다.

7. 상속

- ❖ 하위 클래스는 상위 클래스의 모든 속성을 그대로 상속받으므로 하위 클래스에는 상위 클래스에 없는 새로운 기능이나 수정하고 싶은 기능만을 재정의하면 된다.

```
>>> class Person:
    def __init__( self, name, phone = None ):
        self.name = name
        self.phone = phone
    def __repr__( self ):
        return '<Person {} {}>'.format( self.name, self.phone )

>>> Person.__bases__          # base 클래스 확인
```

- ❖ 파이썬의 모든 클래스의 base 클래스는 object 이다.

7. 상속

❖ 상속 표현 I

```
>>> class Employee( Person ):                # base 클래스는 괄호 안에 표현한다.
    def __init__( self, name, phone, position, salary ):
        Person.__init__( self, name, phone ) # Person 클래스 생성자 호출
        self.position = position
        self.salary = salary
```

❖ 상속 표현 II(선호하는 방식)

```
>>> class Employee( Person ):                # base 클래스는 괄호 안에 표현한다.
    def __init__( self, name, phone, position, salary ):
        super().__init__( name, phone )      # 상위 클래스 생성자 호출
        self.position = position
        self.salary = salary
```

7. 상속

```
>>> m1 = employee( 'hong', 5564, '대리', 200 )
>>> m2 = employee( 'kim', 8546, '과장', 300 )
>>> print( m1.name, m1.position )
>>> print( m2.name, m2.postion )
>>> print( m1 )                # Person.__repr__() 메서드 호출
>>> print( m2 )
```

- ❖ 상위 클래스와 하위 클래스는 별도의 이름 공간을 가지며 계층적인 관계를 가진다. 클래스 객체와 인스턴스 객체도 역시 모두 별도의 이름 공간과 계층적인 관계를 가진다.

7. 상속

- 메서드 대치는 상위 클래스의 같은 메서드를 재정의한 경우로, 기능을 대치하는 효과가 발생한다. 하위 클래스와 상위 클래스에 같은 메서드가 있을 때 하위 클래스의 메서드를 먼저 취하기 때문이다. 즉, 메서드의 검색 우선순위는 하위 클래스이다.

```
>>> class Employee( Person ):
    def __init__( self, name, phone, position, salary ):
        super().__init__( self, name, phone )
        self.position = position
        self.salary = salary
    def __repr__( self ):
        return '<Employee {} {} {} {}>'.format( self.name,
                                                self.phone, self.position, self.salary )

>>> m1 = Employee( 'son', 5564, '대리', 200 )
>>> print( m1 )
```

7. 상속

- 메서드 확장은 하위 클래스에서 그 속성을 변화시키지 위해서 상위 클래스의 메서드를 호출하고, 그 결과를 활용하는 것을 말한다.
- 메서드의 확장과 치환은 하위 클래스에서 상위 클래스 메서드를 호출하느냐 하지 않느냐에 따라 구분된다.

```
>>> class Person:
    def __init__( self, name, phone = None ):
        self.name = name
        self.phone = phone
    def __repr__( self ):
        return '<Person {} {}>'.format( self.name, self.phone )
```


7. 상속

```
>>> class Employee( Person ):
    def __init__( self, name, phone, position, salary ):
        super().__init__( self, name, phone )
        self.position = position
        self.salary = salary
    def __repr__( self ):
        s = super().__repr__()
        return s + '<Employee {} {}>'.format( self.position, self.salary )

>>> p1 = Person( 'lee', 5284 )
>>> print( p1 )
>>> m1 = Employee( 'park', 5564, '대리', 200 )
>>> print( m1 )
```

7. 상속

❖ 상속 클래스 예

inheritance_ex.py

import math

class Point:

def __init__(self, x, y):

self.x = x

self.y = y

def area(self): # 점의 면적은 0

return 0;

def move(self, dx, dy):

self.x += dx

self.y += dy

def __repr__(self):

return 'x = {} y = {}'.format(self.x, self.y)

7. 상속

```
class Circle( Point ):
    def __init__( self, x, y, r ):
        super().__init__( x, y )
        self.radius = r
    def area( self ):
        return math.pi * self.radius * self.radius;
    def __repr__( self ):
        return '{} radius = {}'.format( super().__repr__(), self.radius )
```

7. 상속

```
class Cylinder( Circle ):
    def __init__( self, x, y, r, h ):
        super().__init__( x, y, r )
        self.height = h
    def area( self ):    # 원주의 표면적 = 위아래 원의 면적 + 기둥의 표면적
        return 2 * Circle.area( self ) + 2 * math.pi * self.radius * self.height
    def volume( self ): # 체적
        return Circle.area( self ) * self.height
    def __repr__( self ):
        return '{} height = {}'.format( super().__repr__(), self.height )
```

7. 상속

```
if __name__ == '__main__':  
    p1 = Point( 3, 5 )  
    c1 = Circle( 3, 4, 5 )  
    c2 = Cylinder( 3, 4, 5, 6 )  
  
    print( p1 )  
    print( c1 )  
    print( c2 )  
  
    print( c2.area(), c2.volume() )  
    print( c1.area() )  
  
    c1.move( 10, 10 )  
    print( c1 )
```

7. 상속

- 파이썬 클래스의 모든 메서드는 가상 함수이다. 가상 함수란 메서드의 호출이 참조되는 클래스 인스턴스에 따라서 동적으로 결정되는(Dynamic Binding) 함수를 말한다.
- 파이썬 클래스의 모든 메서드는 가상함수이다.

```
>>> class Base:
    def f( self ):
        self.g()                # g()를 호출
    def g( self ):
        print( 'Base' )
>>> class Derived( Base ):
    def g( self ):
        print( 'Derived' )
>>> b = Base()
>>> b.f()
>>> a = Derived()              # 객체에 따라서 해당 객체에 연관된 함수 호출되는 것을
>>> a.f()                      # 가상함수라 한다.
```

7. 상속

- 두 개 이상의 클래스로부터 상속받는 것을 다중 상속(Multiple Inheritance)이라고 한다. 다중 상속 클래스 정의는

```
class Employee( Person, Job ):
```

와 같이 기반 클래스 이름들을 나열하면 된다.

7. 상속

❖ 다중 상속 예

multiple_inheritance_ex.py

class Person:

def __init__(self, name, phone = None):

self.name = name

self.phone = phone

def __repr__(self):

return 'name = {} tel = {}'.format(self.name, self.phone)

class Job:

def __init__(self, position, salary):

self.position = position

self.salary = salary

def __repr__(self):

return 'position = {} salary = {}'.format(self.position, self.salary)

7. 상속

```
class Employee( Person, Job ):  
    def __init__( self, name, phone, position, salary ):  
        Person.__init__( self, name, phone )           # 언바운드 메서드 호출  
        Job.__init__( self, position, salary )          # 언바운드 메서드 호출  
    def raisesalary( self, rate ):  
        self.salary = self.salary * rate  
    def __repr__( self ):  
        # 언바운드 메서드 호출  
        return Person.__repr__( self ) + ' ' + Job.__repr__( self )  
  
if __name__ == '__main__':  
    e = Employee( 'hong', 5244, 'prof', 300 )  
    e.raisesalary( 1.5 )  
    print( e )
```

7. 상속

- ❖ 다중 상속과 단일 상속은 이름 공간이 두 개 이상 연결되어 있다는 점을 제외하고는 다르지 않다. 만일 같은 이름의 두 상위 클래스 모두에 정의되어 있으면 이름을 찾는 순서가 의미가 있게 된다.
- ❖ Employee 클래스는 두 상위 클래스(Person, Job) 중에서 왼쪽에 먼저 기술된 Person 클래스의 이름 공간을 먼저 찾는다.

7. 상속

➤ 메서드 처리 순서

```
>>> class A:
```

```
    def __init__( self ):
```

```
        pass
```

```
>>> class B:
```

```
    def __init__( self ):
```

```
        pass
```

```
>>> class AA( A ): pass
```

```
>>> class BB( B ): pass
```

```
>>> class C( AA, BB ): pass
```

```
>>> c = C()
```

❖ `__init__()` 메서드의 검색 순서는 다음과 같다.

C -> AA -> A -> BB -> B -> Object

```
>>> C.__mro__          # 검색 순서 확인
```

```
>>> C.mro()           # 검색 순서 확인
```

7. 상속

- 상위 클래스를 동적으로 얻어내는 `super()` 함수는 다중 상속으로 가면 클래스 상속 관계에 따라서 다른 결과를 내기도 한다.
- `super()` 함수는 `mro()` 함수가 출력하는 클래스 순서에 따라 `super()` 함수를 호출하는 현재 클래스의 다음 클래스를 결과로 반환한다.
- 예를 들어 `mro()` 함수가 [A, B, C]이고 `super()` 함수를 호출하는 클래스가 B이면 `super()` 함수의 출력은 C가 된다.
- 인스턴스 객체의 클래스를 알아내려면 `__class__` 속성을 이용하고, 인스턴스 객체와 클래스와의 관계를 파악하려면 `isinstance(instance, class)` 메서드를 사용한다.

```
>>> class A: pass
>>> class B: pass
>>> class C( B ): pass
>>> c = C(); c.__class__          # 인스턴스 객체의 클래스
>>> isinstance( c, A )           # c는 A의 인스턴스 객체가 아님
>>> isinstance( c, C )           # c는 C의 인스턴스 객체
>>> isinstance( c, B )           # c는 B의 인스턴스 객체
```

7. 상속

- 두 클래스 간의 상속 관계를 알아내려면 `issubclass()` 함수 사용

```
>>> issubclass( C, B )
```

```
>>> issubclass( C, A )
```

- 어떤 클래스의 상위 클래스를 알아보려면 `__bases__` 멤버를 이용한다.
`__bases__`는 어떠한 클래스로부터 직접 상속받았는지를 튜플로 알려주는 변수이다.

```
>>> class A: pass
```

```
>>> class B( A ): pass
```

```
>>> class C( B ): pass
```

```
>>> C.__bases__                # 바로 위의 상위 클래스 목록
```

- 전체적으로 상위 클래스의 목록을 얻으려면 `inspect` 모듈의 `getmro()` 함수를 사용한다.

```
>>> import inspect
```

```
>>> inspect.getmro( C )
```

7. 상속

- 전체적으로 내포된 클래스 구조를 알고 싶으면 inspect 모듈의 getclasstree() 함수를 사용한다.

```
>>> class A: pass
>>> class AA( A ): pass
>>> class B: pass
>>> class BB( B ): pass
>>> class C( AA, BB ): pass

>>> import pprint
>>> pp = pprint.PrettyPrinter( indent = 4 )
>>> pp.pprint( inspect.getclasstree( [C] ) )
```

7. 상속

- 다형성(Polymorphism)이란 '여러 형태를 가진다.'는 의미의 그리스어에서 유래된 말로, 상속 관계에서 다른 클래스의 인스턴스 객체들이 같은 멤버 함수의 호출에 대해 각각 다르게 반응하도록 하는 기능이다.
- 예를 들어, $a + b$ 라는 연산을 수행할 때 $+$ 연산은 객체 a 와 b 에 따라 동적으로 결정된다. $a.__add__(b)$ 가 호출되는 것인데, 객체 a 와 b 가 정수이면 정수형 객체의 $__add__()$ 메서드를, 문자열이면 문자열 객체의 $__add__()$ 가 호출된다. 이처럼 동일한 이름의 연산자라 해도 객체에 따라 다른 메서드가 호출되는 것이 다형성이다.
- 다형성은 객체의 종류에 관계없이 하나의 이름으로 원하는 유사한 작업을 수행시킬 수 있으므로 프로그램의 작성과 코드의 이해를 쉽게 해준다.

7. 상속

- 캡슐화(Encapsulation)라 필요한 메서드와 멤버를 하나의 단위로 묶어 외부에서 접근 가능하도록 인터페이스를 공개하는 것을 의미한다.
- 파이썬에서 캡슐화는 코드를 묶는 것(패키지화하는 것)의 의미하며, 반드시 정보를 숨기는 것이 아님을 유의해야 한다.
- 캡슐화는 완전히 내부 정보가 숨겨지는 방식(Black Box)으로 구현될 수도 있고, 외부에서 접근 가능하도록 공개된 방식(White Box)으로 구현될 수도 있다. 정보를 숨기는 것을 정보 은닉(Information Hiding)이라는 용어를 사용한다.
- 파이썬은 주로 공개 방식의 캡슐화를 주로 사용한다. 파이썬의 모든 정보는 기본적으로 공개되어 있다. 관례로 내부적으로만 사용하거나 차기 버전에서 변경 가능성 있는 이름은 _(밑줄)로 시작한다. 단지 변수가 내부용이라는 것이지 완전히 숨기는 것은 아니다.

7. 상속

- 위임(Delegation)은 상속 체계 대신에 사용되는 기법으로, 어떤 객체가 자신이 처리할 수 없는 메시지(메서드 호출)를 받으면, 해당 메시지를 처리할 수 있는 다른 객체에 전달하는 것이다.
- 또는 다른 객체의 메서드 호출을 중간에 있는 클래스가 대신 위임받아 처리하는 것이다.
- 위임은 상속 체계보다 융통성이 있고 일반적이다.
- 파이썬에서 위임은 `__getattr__()` 메서드로 구현한다. 이 메서드는 정의되지 않는 속성을 참조하려고 했을 때 호출된다.

```
__getattr__( self, name )
```

- ❖ 참조하는 속성 이름이 `name`을 통해 전달된다. 이 메서드는 구해진 속성 값을 전달하거나, 속성 값이 없다는 것을 나타내기 위해 `AttributeError` 예외를 발생시켜야 한다.

7. 상속

```
# delegation.py
class Delegation:
    def __init__( self, data ):
        self.stack = data

    def __getattr__( self, name ):          # 정의되지 않은 속성을 참조할 때 호출
        print( 'Delegation {} '.format( name ), end = ' ' )
        return getattr( self.stack, name)  # self.stack의 속성을 대신 이용

a = Delegation( [ 1, 2, 3, 1, 5 ] )
print( a.pop() )
print( a.count( 1 ) )
```

- ❖ `__getattr__()` 메서드가 모든 메서드를 대신 호출해 주는 것은 아니다. `__getattr__()` 메서드는 `__getitem__()`과 `__repr__()`, `__len__()` 처럼 `__`로 시작하는 메서드를 필요로 하는 이름들은 잡아내지 못한다.
- ❖ `__getattr__()`를 요구하는 `a[0]`을 수행하면 `__getattr__()` 메서드가 수행되지 않고 `AttributeError` 에러가 발생한다.

8. Python 예외 처리

Python 예외 처리에 대한 이해

1. 예외 처리란
2. try~except~else문
3. try문에서 finally절 사용하기
4. raise 문으로 예외 발생시키기
5. assert 문으로 예외 발생시키기

1. 예외 처리란

- 프로그램을 수행하다 보면 문법은 맞으나 실행 중 더 이상 진행할 수 없는 상황이 발생한다. 이것을 예외(Exception)라고 한다.
- 예외 상황에는 여러 가지 경우가 있을 수 있다. 예를 들면, 0으로 숫자 나누기, 문자열과 숫자 더하기, 참조 범위를 넘어서 인덱스 참조하기 등이다.

```
>>> a, b = 5, 0
>>> c = a / b                # ZeroDivisionError 발생
>>> 4 + spam * 3            # NameError 발생
>>> '2' + 2                 # TypeError 발생
```

- 예외가 발생하면 에러 메시지가 나온다. 스택 추적 형태로 상황을 알려주며, 마지막 부분에 최종적으로 에러가 발행한 정보를 표시해 준다.
- 모든 예외는 클래스로 표현된다. 최상위에 있는 클래스는 BaseException 클래스이다.

1. 예외 처리란

<https://docs.python.org/3/library/exceptions.html?highlight=baseexception#exception-hierarchy>

BaseException

--- SystemExit

--- KeyboardInterrupt

--- GeneratorExit

--- Exception

 --- StopIteration

 --- StopAsyncIteration

 --- ArithmeticError

 | --- FloatingPointError

 | --- OverflowError

 | --- ZeroDivisionError

 --- EOFError

 --- ImportError

 --- NameError

 --- RuntimeError

 --- SyntaxError

 --- TypeError

 --- ValueError

 | --- UnicodeError

 --- Warning

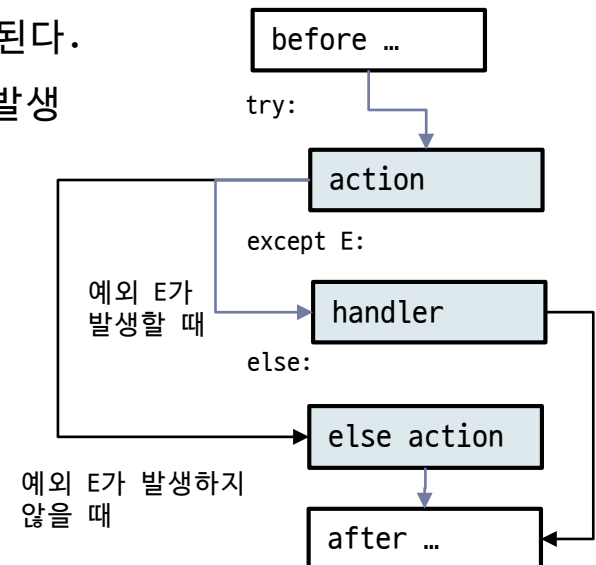
 --- SyntaxWarning

2. try~except~else문

- 예외가 발생하면 프로그램에서 try~except 문을 사용하여 예외를 잡아낼 수 있다. 사용하는 구문의 형식은 다음과 같다.

```
try:
    <문1>
except <예외 종류1>:
    <문2>                # <문1>을 수행하는 중 <예외 종류1>이 발생하면 수행
else:                    # else이하는 생략할 수 있다.
    <문3>                # 예외가 발생하지 않으면 <문3>가 수행된다.
```

- 처리 순서는, 우선 try 절(try와 except 사이)이 실행된다. 예외가 발생하지 않으면 else 절을 수행한다. 예외가 발생하면 except 절을 수행하고 try~except 문을 마친다. else 절은 생략할 수 있다.



2. try~except~else문

```
>>> x = 0
>>> try:
    print( 1.0 / x )
except ZeroDivisionError:                # 0으로 나누기 exception
    print( 'has no inverse' )

>>> name = 'notexistingfile'
>>> try:
    f = open( name, 'r' )
except IOError:                          # IO exception
    print( 'cannot open', name )
else:
    print( name, 'has', len( f.readlines() ), 'lines' )
    f.close()
```


2. try~except~else문

```
>>> try:
    spam()
except NameError as x:      # NameError 객체를 x로 받는다.
    print( x )
```

```
>>> def this_fails():
    x = 1 / 0
```

```
>>> try:
    this_fails()
except ZeroDivisionError as err:
    print( 'Runtime error : ', err )
```

- try:는 try 절에서 발생하는 예외뿐 아니라, 간접적으로 호출한 함수의 내부 예외도 처리한다.

2. try~except~else문

```
>>> try:
    some_job()
except ZeroDivisionError:
    ~ 생략 ~
except NameError:
    ~ 생략 ~
except ( TypeError, IOError ):          # 두 예외는 같은 루틴을 공유한다.
    ~ 생략 ~
else:
    ~ 생략 ~
```

- 여러 개의 예외가 발생할 수 있고, 각 경우마다 별도의 처리를 원한다면 except 절을 여러 번 사용할 수 있다.

```
>>> try:
    some_job()
except:
    ~ 생략 ~
```

- except 절에 아무것도 기술하지 않으면 모든 예외를 처리한다.

2. *try~except~else*문

- 예외 클래스는 같은 종류의 예외 간에 계층적인 관계를 가지고 있으므로 이를 이용하면 여러 가지의 예외를 한꺼번에 검출해 낼 수 있다. except 절에 기술된 예외 클래스는 하위의 파생 클래스의 예외까지 함께 처리할 수 있다.

```
>>> def dosomething():
        a = 1 / 0

>>> try:
        dosomething()
except ArithmeticError:
        print( 'Exception occurred' )
```

2. *try~except~else*문

- 좀 더 구체적으로 어떤 예외가 발생하는지 알고 싶다면 sys 모듈의 exc_info() 함수를 사용할 수 있다. exc_info() 함수는 예외 클래스, 예외 인스턴스 객체, traceback 객체를 반환한다. 예외 정보를 출력하려면 traceback 모듈의 print_exception()나 print_exc() 함수를 사용할 수 있다.

```
>>> import sys
>>> import traceback
>>> x = 0
>>> try:
    1 / x
except:
    etype, evalue, tb = sys.exc_info()
    print( 'Exception class = ', etype )
    print( 'value = ', evalue )
    print( 'traceback object = ', tb )
    traceback.print_exception( etype, evalue, tb )
    traceback.print_exc()
```

3. try 문에서 finally절 사용하기

- try 문에서 finally 절을 사용할 수 있는데, finally 절에 포함된 문들은 예외 발생 여부에 관계없이 모두 수행된다.

```
>>> f = open( filename, 'w' )
>>> try:
    do_something_with( f )
finally:
    f.close()
```

- do_something_with(f) 문에서 예외가 발행하지 않으면 finally 절의 f.close()를 수행하고, 예외가 발생해도 f.close()를 수행한다. 이 상황은 어떤 경우에도 파일을 close 할 경우에 유용하게 사용된다.

```
>>> x = 0
>>> try:
    1 / x
except:
    print( sys.exc_info() )
finally:
    print( 'All the time' )
```

4. raise 문으로 예외 발생시키기

- 이미 시스템에 내장 되어있는 예외를 raise 문을 이용하여 발생시킬 수 있다. raise 문은 예외 클래스의 인스턴스 객체를 매개 변수로 받아들인다.

```
>>> raise IndexError( "range error" )
```

- 만일 raise 문이 인수 없이 사용된다면 가장 최근에 발생했던 예외가 다시 발행한다. 만일 최근에 발생한 예외가 없다면 RuntimeError 가 발생한다.

```
>>> try:
```

```
    raise IndexError( 'a' )
```

```
except:
```

```
    print( 'in except...' )
```

```
    raise
```

- 이름 공간을 이용하면 예외 객체에 정보를 넘기는 것도 가능하다.

```
>>> ie = IndexError( 'a' ); ie.value = 10; ie.count = 3
```

```
>>> try:      raise ie
```

```
except IndexError as x:
```

```
    print( x, x.value, x.count )
```

4. *raise* 문으로 예외 발생시키기

- 사용자 예외를 발생시키는 표준 방법은 `BaseException` 클래스의 하위 클래스를 이용하는 것이다. 상속받을 수 있는 클래스는 내장 예외의 어떤 것일 수도 있다.

```
# user_defined_exception.py
import sys

class Big( Exception ):
    pass

class Small( Big ):
    pass

def dosomething():
    raise Big( 'big exception' )

def dosomething2():
    raise Small( 'small exception' )

for f in (dosomething, dosomething2 ):
    try:  f()
    except Big: print( sys.exc_info() )
```

5. assert 문으로 예외 발생시키기

- 예외를 발생시키는 특수한 경우로 assert 문에 의한 AssertionError 에러가 있다. assert 문은 주로 디버깅할 때 많이 사용한다. 프로그램이 바르게 진행되는지를 시험할 때 사용한다.

assert <시험 코드>, <데이터>

- <시험 코드>가 거짓이면 'raise AssertionError, 데이터 ' 예외를 발생시킨다. <시험 코드>가 참이면 그냥 통과한다. <데이터>는 사용하지 않아도 된다.

```
# asserttest.py
```

```
a = 30
```

```
margin = 2 * 0.2
```

```
assert margin > 10, 'not enough margin'
```

- ❖ 파이썬을 -O 옵션으로 실행하면 assert 문은 컴파일된 바이트 코드로부터 자동으로 삭제된다. 또한, -O 옵션은 __debug__ 플래그가 0이 되어 디버깅 상태가 아님을 알린다.

참고

- 파이썬3 바이블 - 이강성(프리렉)
- 점프 투 파이썬 - 박응용(이지스퍼블리싱)
- Wikipedia.org를 통한 검색
- 기타 Web 검색