
Table of Contents

Introduction	1.1
1-1. 개발환경세팅	1.2
1-2. IPython Notebook 살펴보기	1.3
1-3. 파이썬 코딩스타일	1.4
1-4. 파일과 디렉토리 경로 다루기	1.5
1-5. pip 패키지 관리자	1.6
1-6. 파이썬 소개	1.7
1-7. 파이썬 둘러보기	1.8
2-1. 기본 문법1	1.9
2-2. 기본 문법2	1.10
2-3. 메일보내기	1.11
3-1. 엑셀 다루기	1.12
4-1. 크롤링	1.13
4-2. 웹 스크래핑	1.14
4-3. Selenium 사용하기	1.15
5-1. 파이썬 데이터분석	1.16
5-2. numpy 사용하기	1.17
5-3. numpy	1.18
5-4. pandas	1.19
5-5. matplotlib	1.20
참고. 난수	1.21
참고. 정규표현식	1.22
참고. 직렬화와 역직렬화	1.23
참고. 디버깅	1.24

실용파이썬 참고자료

처음 프로그래밍을 접하는 분들도 쉽게 따라할 수 있는

파이썬을 활용한 업무 효율 향상 교육의 참고용 자료입니다.

1회 - 파이썬 기초 문법

2회 - 파이썬 프로그래밍 연습

3회 - 파이썬으로 스프레드시트 다루기

4회 - 웹 크롤링 및 업무 자동화

5회 - 파이썬 데이터분석 맛보기

자료 출처 (보다 자세한 자료는 아래 각 링크에서 확인할 수 있습니다)

1. 점프투파이썬 : <https://wikidocs.net/book/1>
2. 예제로 배우는 Python 프로그래밍 : <http://pythonstudy.xyz/Python/Applications>
3. 플러닝 - 파이썬 빅데이터 분석 : <https://www.flearning.net/courses/6>

Python 데이터 분석을 위한 환경 구성하기

[Windows 경우] miniconda 설치

Anaconda는 데이터 분석을 위한 오픈 소스 Python 플랫폼입니다. 단 한 번의 설치로 모든 환경 설정을 한 방에 끝내주기 때문에, Windows 사용자들이 사용하기에 아주 좋습니다.

한편 miniconda는 Anaconda의 핵심 부분만을 추출해서 재구성한 플랫폼으로, 고유한 패키지 관리 시스템인 conda와 기본적인 Python만을 포함하고 있기 때문에 설치 소요 시간이 더 짧습니다.

Windows 사용자분들의 경우, 본 동영상에 따라 miniconda를 설치해 주시길 바랍니다.

동영상 중간에 사용하는 링크 및 명령어를 아래와 같이 명시하였습니다.

- miniconda 설치 파일 다운로드 링크:

<http://conda.pydata.org/miniconda.html>

- conda 버전 업데이트:

```
conda update conda
```

- Python 버전 확인:

```
python --version
```

- pip 패키지 매니저를 사용하여 IPython 설치:

```
pip install ipython
```

[Mac OS 일 경우] miniconda 설치

Anaconda는 데이터 분석을 위한 오픈 소스 Python 플랫폼입니다. 단 한 번의 설치로 모든 환경 설정을 한 방에 끝내주기 때문에, Mac OS 사용자들이 사용하기에 아주 좋습니다.

한편 miniconda는 Anaconda의 핵심 부분만을 추출해서 재구성한 플랫폼으로, 고유한 패키지 관리 시스템인 conda와 기본적인 Python만을 포함하고 있기 때문에 설치 소요 시간이 더 짧습니다.

Mac OS 사용자분들의 경우, 본 동영상에 따라 miniconda를 설치해 주시길 바랍니다.

동영상 중간에 사용하는 링크 및 명령어를 아래와 같이 명시하였습니다.

- miniconda 설치 파일 다운로드 링크:

<http://conda.pydata.org/miniconda.html>

- miniconda 설치 파일을 다운로드 디렉터리로 옮긴 뒤, 실행 가능하도록 권한 설정:

```
chmod u+x Miniconda3-latest-MacOSX-x86_64.sh
```

- 권한 설정 후 miniconda 설치 파일 실행:

```
./Miniconda3-latest-MacOSX-x86_64.sh
```

- pip 패키지 매니저를 사용하여 IPython 설치:

```
pip install ipython
```

[공통] Python 데이터 분석 라이브러리 설치하기

Windows 사용자분들의 경우 cmd 프로그램을, Mac OS 사용자분들의 경우 터미널 프로그램을 실행하시고 동영상의 안내를 따라 주시길 바랍니다.

동영상 중간에 사용하는 명령어를 아래와 같이 명시하였습니다.

- numpy 설치:

```
pip install numpy
```

- pandas 설치:

```
pip install pandas
```

- matplotlib 설치:

```
pip install matplotlib
```

- jupyter 설치:

```
pip install jupyter
```

(중요) Mac OS 사용자분들의 경우, matplotlib까지 설치가 끝난 후, 터미널 상에서 다음의 코드를 한 번 실행해 주시길 바랍니다. 이는 추후에 matplotlib가 무리 없이 실행되도록 하기 위해 선행되어야 하는 작업입니다.

```
echo "backend: TkAgg" > ~/.matplotlib/matplotlibrc
```

IPython 살펴보기

IPython의 기본 특징

변수 정의 및 변수에 저장된 값 확인

IPython 콘솔의 좌측을 보시면 'In' 혹은 'Out' 표시와 숫자가 함께 표시되는 것을 확인할 수 있습니다. In이 활성화되어 있는 경우 내가 코드를 입력할 수 있는 상태이며, 실제로 코드를 입력하고 엔터를 누르게 되면 In 옆의 숫자가 1 증가하면서 다음 코드를 입력할 수 있는 상태로 변화하게 됩니다.

IPython에서 방금 전에 정의한 변수에 어떠한 값이 저장되어 있는지 확인하고자 한다면, 해당 변수 이름만 입력하고 엔터를 누르면 Out 표시와 함께 현재 해당 변수에 저장되어 있는 값이 표시됩니다.

변수 이름의 자동 완성

IPython에서는 tab을 통해서 변수의 이름을 자동 완성해주는 기능을 지원합니다. 긴 변수 이름을 일일이 다시 타이핑할 필요없이, 변수 이름의 앞부분 일부만 입력한 상태에서 tab 키를 누르면 원래 변수의 이름이 자동으로 완성됩니다.

복수 개 행의 자동 들여쓰기

복수 개의 행을 필요로 하는 반복문, 조건문, 함수 등의 경우, 매 행을 넘길 때마다 IPython에서 자동으로 들여쓰기를 수행해 줍니다.

이 때, 조건문의 `if` 와 `else` 등과 같이 동일한 위상에 해당하는 코드는 반드시 동일한 수준의 들여쓰기를 적용하여 작성할 수 있도록 신경써야 합니다.

변수 및 함수의 정보 확인

변수나 함수 등의 뒤에 `?` 를 붙이면 이에 대한 일반적인 정보를 제시하는 기능을 제공합니다. 변수 이름 바로 뒤에 `?` 를 붙여주게 되면, 해당 변수에 대한 자료형과 이에 대한 간단한 설명을 확인할 수 있습니다.

함수 이름 바로 뒤에 `?` 를 붙여주기 되면 해당 함수에 대한 간단한 설명을 확인할 수 있습니다.

매직 명령어

맨 앞이 `%` 로 시작하는 IPython 고유 명령어를 매직 명령어(magic command)라고 합니다. 매직 명령어는 IPython에서만 사용되는 특별한 명령어로, 여러 가지 간편한 기능들을 제공합니다.

`%who` 명령어는, 내가 이미 정의하여 현재 메모리에 올라와 있는 변수들의 이름을 모두 나열합니다. `del` 키워드를 사용하여 몇몇 변수를 제거한 뒤 `%who` 명령어를 다시 사용하면, 해당 변수들이 제거된 것을 확인할 수 있습니다.

한편, 현재 메모리에 올라와 있는 모든 변수를 제거하고자 할 경우, `%reset`, `%who` 명령어를 순서대로 실행합니다.

그리고, `%time` 명령어를 사용하면 바로 뒤에 이어지는 코드가 실행되는 데 소요되는 시간을 측정할 수 있습니다.

매직 명령어는 이외에도 여러 가지가 있으며, 구글에서 검색해 보시면 여러 가지 매직 명령어를 확인해보실 수 있을 것입니다.

IPython Notebook 살펴보기

IPython Notebook 실행 방법

IPython Notebook을 실행하기 위해, cmd 혹은 터미널 프로그램을 실행한 뒤 다음과 같이 입력합니다.

```
jupyter notebook
```

그러면 IPython Notebook이 실행되면서 `http://localhost:8888` 에서 실행되고 있다는 메시지를 확인할 수 있습니다. 웹 브라우저를 실행하여 해당 URL로 접속하면, IPython Notebook의 GUI 화면을 확인할 수 있습니다.

IPython Notebook에서의 노트북은 데이터 분석을 한 번 하는데 사용할 일지 하나라고 생각하시면 됩니다. 상단의 `New` 버튼을 클릭한 뒤 맨 하단의 `Python 3` 메뉴를 클릭하면, 새로운 노트북을 생성하고 편집할 수 있습니다.

IPython Notebook 기본 사용법

셀 생성, 실행, 삭제, 이동하기

화면 중앙에 위치한 직사각형의 입력란 하나하나를 셀(cell)이라고 부릅니다. 셀을 클릭하여 코드를 입력할 수 있는데, IPython처럼 한 줄만 입력할 수 있는 것이 아니라 여러 줄 입력할 수도 있습니다. 셀에 표시된 파란색 직사각형은 여러분이 현재 해당 셀을 선택하였거나, 혹은 편집하고 있다는

것을 나타내는 커서입니다.

현재 커서가 위치한 셀을 실행하려면, 화면 상단의 ▶ 버튼을 클릭합니다.

현재 셀의 바로 다음 위치에 새로운 셀을 만드려면, 상단의 + 버튼을 클릭합니다.

만약 기존에 작성한 셀을 삭제하고 싶다면, 삭제하고자 하는 셀을 선택한 후 상단의 가위 버튼을 클릭하면 됩니다. 셀을 단순히 위아래로 이동시키고 싶다면 ↑ 및 ↓ 버튼을 클릭합니다.

셀 실행 초기화, 전체 셀 한번에 실행하기

만약 현재까지 실행된 결과들을 메모리 상에서 모두 초기화하고 싶다면 상단의 원형 화살표 버튼을 클릭합니다. 만약 화면에 출력된 결과물까지 모두 삭제해 버리고 싶다면, 상단의 Kernel 메뉴에서 Restart & Clear Output 을 클릭합니다.

한편 모든 작업을 초기화한 직후에, 이들을 지정된 순서에 따라 한 번에 일괄적으로 실행하고 싶다면, Cell 메뉴에서 Run All 을 클릭합니다.

노트북 export하기

여러분이 작성한 노트북을 다양한 포맷의 파일로 export할 수 있습니다. File 메뉴에서 Download as 메뉴를 확인하면, python 스크립트 파일을 비롯하여 html, pdf 등 다양한 포맷의 파일로 export가 가능합니다.

또, 키보드 버튼을 클릭하시면 각각의 기능에 대한 단축키를 확인할 수 있습니다. 나중에 IPython Notebook 사용이 숙달되었을 때, 단축키를 사용하시면 보다 빠른 속도로 데이터 분석을 수행하실 수 있으니 주목해 보셔도 좋겠습니다.

노트북 저장 및 종료하기

지금까지 작성한 노트북의 상태를 저장하고자 한다면, 디스켓 버튼을 클릭합니다.

현재 작업하고 있는 노트북을 종료하고자 한다면, File 메뉴의 Close and Halt 를 클릭하시면 탭이 닫히면서 처음의 화면으로 이동하게 됩니다. .ipynb 라는 확장자는, IPython Notebook에서 다루는 파일에 부여되는 고유한 확장자입니다.

만약 IPython Notebook 자체를 완전히 종료하고자 하신다면, 웹 브라우저에서 해당 탭을 닫으신 뒤 터미널로 돌아가셔서 **Ctrl+C** 키를 누르면 됩니다.

PEP 8 파이썬 코딩 스타일

Python Enhancement Proposal 8 (PEP 8)은 파이썬 코딩 스타일에 대한 가이드를 제시하고 있다. PEP 8은 2001년 귀도 반 로섬에 의해 처음 제안되었으며, python.org 의 [PEP 링크](#)에 자세히 소개되어 있다. 파이썬 프로그래머들은 일반적으로 이러한 PEP 8 코딩 스타일에 따라 프로그래밍을 하고 있는데, 이러한 일관된 코딩 스타일을 적용하는 것은 자신의 코드를 명료하게 할 뿐만 아니라 특히 다른 개발자 혹은 커뮤니티간 코딩을 공유할 때 매우 효율적이다.

아래는 PEP 8 의 중요한 부분에 대한 요약이다.

카 테 고 리	코딩 스타일	예제
코 드 레 이 아웃	들여쓰기를 할 때 Tab 대신 공백(Space)을 사용한다. 특히 Python 3는 Tab과 공백을 혼용해서 사용하는 것을 허용하지 않는다.	
	문법적으로 들여쓰기를 할 때는 4개의 공백을 사용한다	
	각 라인은 79자 이하로 한다. 라인이 길어서 다음 라인으로 넘어갈 때는 원래 들여쓰기 자리에서 4개 공백을 더 들여쓴다	
	함수나 클래스는 2개의 공백 라인을 추가하여 구분한다. 메서드는 한 개의 공백 라인으로 구분한다	
	import는 (여러 모듈을 콤마로 연결하지 말고) 한 라인에 하나의 모듈을 import한다	No: import os, sys Yes: import os import sys
	컬렉션 인덱스나 함수 호출, 함수 파라미터 등에서 불필요한 공백을 넣지 않는다	No: spam(ham[1], { eggs: 2 }) bar = (0,) spam (1) Yes: spam(ham[1], {eggs: 2}) bar = (0,) spam(1)
	변수 할당시 할당자 앞뒤로 하나의 공백만 넣는다	No: i=i+1 Yes: i = i + 1
명 명 규 칙	함수, 변수, Attribute는 소문자로 단어 간은 밑줄(_)을 사용하여 연결한다	예: total_numbers
	클래스는 단어 첫 문자마다 대문자를 써서 연결하는 CapWords 포맷으로 명명한다	예: CoreClass
	모듈명은 짧게 소문자로 사용하며 밑줄을 쓸 수 있다. 패키지명 역시 짧게 소문자를 사용하지만 밑줄은 사용하지 않는다.	예: serial_reader

	모듈 상수는 모두 대문자를 사용하고 단어마다 밑줄로 연결하는 ALL_CAPS 포맷으로 명명한다	예: MAX_COUNT = 100
	클래스의 public attribute는 밑줄로 시작하지 말아야 한다	예: name
	클래스의 protected instance attribute는 하나의 밑줄로 시작한다	예: _initialized
	클래스의 private instance attribute는 2개의 밑줄로 시작한다	예: __private_var
	인스턴스 메서드는 (객체 자신을 가리키기 위해) self 를 사용한다	예: def copy(self, other):
	클래스 메서드는 (클래스 자신을 가리키기 위해) cls 를 사용한다	예: def clone(cls, other):
문 장 과 표 현 식	if, for, while 블록 문장을 한 라인으로 작성하지 말 것. 여러 라인에 걸쳐 사용하는 것이 더 명료함	No: if a < 0: a = 0 Yes: if a < 0: a = 0
	a는 b가 아니다를 표현할 때 a is not b 를 사용한다. not a is b 를 사용하지 말 것	No: if not a is b Yes: if a is not b
	값이 비어있는지 아닌지를 검사하기 위해 길이를 체크하는 방식을 사용하지 말 것. 대신 if mylist 와 같이 표현함	No: if len(mylist) == 0 Yes: if not mylist No: if len(mylist) > 0 Yes: if mylist
	import 문은 항상 파일의 상단에 위치하며, 표준 라이브러리 모듈, 3rd Party 모듈, 그리고 자신의 모듈 순으로 import 한다	예: import os import numpy import mypkg
	모듈 import시 절대 경로를 사용할 것을 권장한다. 예를 들어, sibling 모듈이 현재 모듈과 같은 폴더에 있더라도 패키지명부터 절대 경로를 사용함. 단, 복잡한 패키지 경로를 갖는 경우 상대경로(.)를 사용할 수 있다.	No: import sibling Yes: import mypkg.sibling from mypkg import sibling from . import sibling # 상대경로 from .sibling import example

파이썬에서 파일과 디렉토리 경로 다루기

파이썬에서 디렉토리와 파일경로를 다루는 주요 함수들에 대해 알아보자. 디렉토리 및 파일 경로에 대한 함수들은 `os` 모듈에 있으므로, `os` 모듈을 `import` 한 것을 전제로 한다. 특히, `os.path` 모듈은 파일명과 파일경로에 대한 유용한 함수들을 많이 제공하고 있다.

용도	함수 예제
현재 작업 폴더 얻기	<code>os.getcwd()</code> # "C:\Temp"
디렉토리 변경	<code>os.chdir("C:\Tmp")</code>
특정 경로에 대해 절대 경로 얻기	<code>os.path.abspath(".\Scripts")</code> # "C:\Python35\Scripts"
경로 중 디렉토리명만 얻기	<code>os.path.dirname("C:/Python35/Scripts/pip.exe")</code> # "C:/Python35/Scripts"
경로 중 파일명만 얻기	<code>os.path.basename("C:/Python35/Scripts/pip.exe")</code> # "pip.exe"
경로 중 디렉토리명과 파일명을 나누어 얻기	<code>dir, file =</code> <code>os.path.split("C:/Python35/Scripts/pip.exe")</code>
파일 각 경로를 나눠 리스트로 리턴하기 <code>os.path.sep</code> 은 OS별 경로 분리자	<code>"C:\Python35\Scripts\pip.exe".split(os.path.sep)</code> # ["C:", "Python35", "Scripts", "pip.exe"]
경로를 병합하여 새 경로 생성	<code>os.path.join('C:\Tmp', 'a', 'b')</code> # "C:\Tmp\a\b"
디렉토리 안의 파일/서브디렉토리 리스트	<code>os.listdir("C:\Python35")</code>
파일 혹은 디렉토리 경로가 존재하는지 체크하기	<code>os.path.exists("C:\Python35")</code>
디렉토리 경로가 존재하는지 체크하기	<code>os.path.isdir("C:\Python35")</code>
파일 경로가 존재하는지 체크하기	<code>os.path.isfile("C:\Python35\python.exe")</code>
파일의 크기	<code>os.path.getsize("C:\Python35\python.exe")</code>

1. pip 패키지 관리자

pip은 파이썬 패키지를 설치하고 관리하는 패키지 관리자(Package Manager)이다. pip은 "Pip Installs Packages"의 약자로서 재귀적인 약어이다.

pip은 Python 2.7.9+와 Python 3.4+에서 디폴트로 설치되어 있다. 만약 pip이 시스템에 설치되지 않은 경우는 다음과 같이 설치할 수 있다.

```
curl https://bootstrap.pypa.io/get-pip.py | python
```

2. pip 패키지 설치와 제거

pip을 이용하면 패키지를 쉽게 설치할 수 있다. pip으로 설치할 수 패키지들의 목록은 [Python Package Index \(PyPI\)](#)에서 찾아 볼 수 있다. 다음은 패키지를 설치하는 pip 명령이다.

```
pip install 패키지명

ex)
$ pip install django
$ pip install numpy
$ pip install matplotlib
```

설치된 패키지를 Uninstall하기 위해서는 아래와 같은 pip uninstall 명령을 사용한다.

```
pip uninstall 패키지명

ex)
$ pip uninstall django
```

3. pip: requirements 파일 생성과 사용

pip을 이용하면 일련의 패키지 그룹을 묶어서 설치할 수도 있다. 즉, requirements.txt 파일에 설치할 패키지 리스트를 정의한 후, 아래와 같은 명령으로 전체를 한꺼번에 설치할 수 있다. 실제 "pip install 패키지명"을 사용하여 패키지를 하나 하나 설치하는 것보다, requirements.txt를 만들어 패키지 리스트를 이 파일에서 관리하고 이 파일을 통해 pip 설치 수행하는 것이 더 좋은 방법이다.

```
$ pip install -r requirements.txt
```

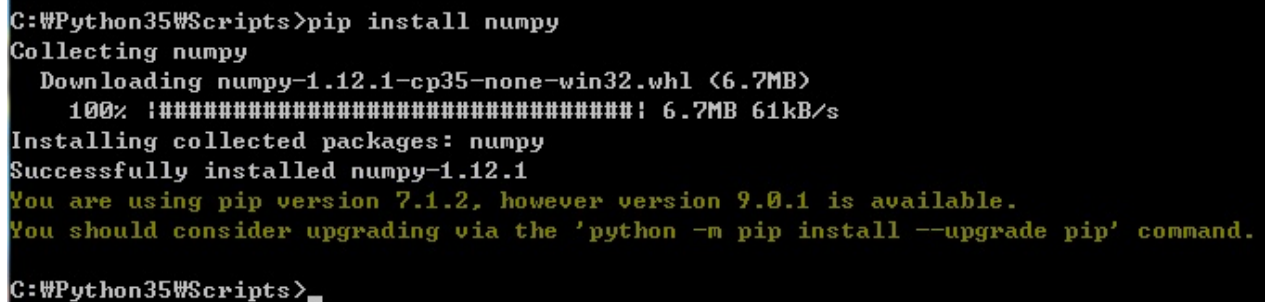
하지만 경우에 따라 requirements.txt를 미리 만들지 않고, 개발서버의 가상환경에 여러 패키지를 이미 설치했을 수도 있다. 이런 경우는 아래와 같은 pip freeze 명령을 사용하여 requirements.txt 파일을 만들 수 있다. 이러한 기능은 개발서버의 가상환경을 Production 가상환경으로

Deployment 하는데 유용하게 쓰일 수 있다.

```
$ pip freeze > requirements.txt
```

4. 윈도우즈에서의 pip 사용

윈도우즈에서 pip 을 사용하기 위해서는 python 설치 폴더 밑의 Scripts 서브디렉토리에서 pip을 실행한다. 예를 들어, 아래 그림은 C:\Python3.5 폴더에 Python 3.5 가 설치되어 있을때, Scripts/pip 명령을 사용하여 numpy 패키지를 설치하는 예이다.



```
C:\Python35\Scripts>pip install numpy
Collecting numpy
  Downloading numpy-1.12.1-cp35-none-win32.whl <6.7MB>
    100% |#####| 6.7MB 61kB/s
Installing collected packages: numpy
Successfully installed numpy-1.12.1
You are using pip version 7.1.2, however version 9.0.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
C:\Python35\Scripts>_
```

파이썬 소개

1. Python 언어

파이썬 (Python)은 범용 프로그래밍 언어로서 코드 가독성(readability)와 간결한 코딩을 강조한 언어이다.

파이썬은 인터프리터(interpreter) 언어로서, 리눅스, Mac OS X, 윈도우즈 등 다양한 시스템에 널리 사용된다.

Python은 원래 그리스 신화에서 그리스 중부 델파이를 지배하였던 큰 뱀인데, 제우스의 아들 아폴로에 의해 화살을 맞고 죽게된다.



2. Python의 간단한 역사

Python은 1989년 12월 네델란드 개발자 Guido van Rossum 에 의해 개발되기 시작하여, 약 1년간 개발하여 1991년 처음 Python 0.9 버전을 세상에 내놓았다. 그후 정식 Python 1.0 버전은 1994년에 출시되었으며, Python 2.0은 2000년에, Python 3.0은 2008년에 각각 출시되었다.



Guido van Rossum

1956년 생의 네덜란드 개발자로서 Python 창시자. 네덜란드 CWI, 미국 NIST 등의 여러 연구소에서 근무하였으며, 구글에서 약 7년간 근무, 현재는 Dropbox에서 일하고 있다.

Python이 구글에서 상당히 많은 프로젝트에 쓰여지고 있다는 점과 Dropbox의 많은 코드가 Python으로 작성되었다는 점은 아마 우연이 아닐 것이다.

Guido는 파이썬의 개발 동기에 대해 이렇게 말하고 있다.

"약 6년 전인 1989년 12월, 크리스마스를 전후하여 취미로 만들어 볼 프로그래밍 프로젝트를 찾고 있었죠. 그 때 사무실은 잠겨있었지만, 집에 컴퓨터가 있었고, 뭐 특별히 할 일도 없었죠. 그래서 그 때 당시 한동안 생각하고 있었던 새 스크립트 언어에 대한 인터프리터를 만들어 보기로 했죠. 유닉스/C 해커들에게 어필할 수 있는, ABC 언어로부터 파생된 언어말이죠. 나는 그 프로젝트명으로 Python이라는 이름을 선택했는데, 그 당시 약간은 불손한 기분이 들어서이기도 했고, 또한 당시 Monty Python's Flying Circus(BBC 코메디)에 열성팬이기도 하여..."

- 1996, Guido

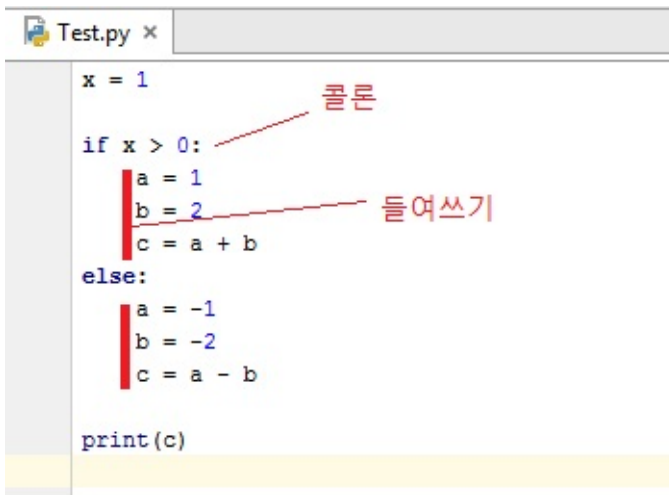
파이썬 코딩의 기초

1. 코딩블럭 들여쓰기 (Indentation)

파이썬은 코딩블럭을 표시하기 위해 들여쓰기(Indentation)를 사용한다. 이는 보통 Curly Bracket ({...})을 사용하는 C, C#, Java 등의 다른 언어들과는 매우 다른 독특한 스타일이다. 코딩블럭을 시작하는 문장들 예를 들어 if, for, def 문들의 끝에는 아래 예제에서 보듯이, 콜론(:)을 사용하고 내부의 코딩블럭은 동일한 들여쓰기를 사용한다.

일반적으로 들여쓰기에는 4개의 공백을 사용할 것을 권장하는데, 동일한 블럭의 들여쓰기는 모두 동일한 수의 공백을 사용해야 한다. 즉, 모두 4개의 공백을 사용하다가 하나만 5개의 공백을 사용하면, IndentationError: unexpected indent라는 에러가 발생한다. 이는 파이썬 컴파일러가 올바른

들여쓰기를 강제하는 것으로, 코드를 일관되고 명료하게 하기 위함이다. 또한 추가로 한가지 주의할 점은 공백과 탭을 혼용해서 사용하지 말아야 한다는 것이다.



2. 파이썬 표준 라이브러리

파이썬은 상당히 많은 표준 라이브러리들을 제공하고 있는데, 이 표준 라이브러리를 불러다 쓰기 위해서는 import문을 사용한다. 예를 들어, 표준 라이브러리 중의 하나인 math에 있는 sqrt()라는 함수를 불러다 쓰기 위해서는, 아래 예제와 같이 "import math" 를 실행하고, math.sqrt() 함수를 호출하면 된다.

```

import math

n = math.sqrt(9.0)

print(n)    # 3.0 출력

```

3. 코멘트

파이썬에서 코멘트를 표시하기 위하여 파운드(#) 사인을 사용한다. 코멘트는 라인의 처음에 올 수도 있고, 라인의 문장이 끝난 부분에 올 수도 있다. 표준 코딩 스타일에서는 # 사인 뒤에 하나의 공백을 두는 것을 권장한다.

```

# 코멘트1
run(1)

run(2) # 코멘트2

```

4. PEP

PEP이란 **Python Enhancement Proposals**의 약자로서 파이썬을 개선하기 위한 제안서를 의미한다. 이러한 PEP은 다음과 같이 크게 3종류로 구분할 수 있으며, Python Software Foundation의 공식 웹사이트인 python.org에서 관리한다.

1. 파이썬에 새로운 기능(Feature)을 추가하거나 구현 방식을 제안하는 **Standard Track PEP**
2. 파이썬 디자인 이슈를 설명하거나 일반적인 가이드라인 혹은 정보를 커뮤니티에 제공하는 **Informational PEP**
3. 파이썬을 둘러싼 프로세스를 설명하거나 프로세스 개선을 제안하는 **Process PEP**. 예를 들어, 프로세스 절차, 가이드라인, 의사결정 방식의 개선, 파이썬 개발 도구 및 환경의 변경 등등.

PEP은 파이썬 언어 자체 뿐만 아니라 코딩 표준, 커뮤니티 이슈 등을 담고 있는 유용한 자료이므로 파이썬을 배우면서 자주 참고하면 좋다. PEP은 각 문서마다 번호가 지정되어 있는데, PEP 번호별 내용은 <https://www.python.org/dev/peps/>에서 찾아 볼 수 있다.

특히, 파인썬 코딩과 관련한 코딩 표준(Coding Convention)에 관한 문서는 PEP 8 (<https://www.python.org/dev/peps/pep-0008>)에 정의되어 있는데, 파이썬의 기초를 익힌 후 참조하면 파이썬 코딩 스타일을 익히는데 도움이 된다.

파이썬 둘러보기

도대체 파이썬이라는 언어는 어떻게 생겼는지, 간단한 코드를 작성하며 알아보자. 파이썬을 자세히 탐구하기 전에 전체적인 모습을 죽 훑어보는 것은 매우 유익한 일이 될 것이다.

"백문이 불여일견, 백견이 불여일타"라고 했다. 직접 따라 해보자.

파이썬 기초 실습 준비하기

파이썬 프로그래밍 실습을 시작하기 전에 기초적인 것을 준비해 보자.

[시작] 메뉴에서 [프로그램 → Python 3.X → Python 3.X(XX-bit)]을 선택하면 다음과 같은 화면이 나타난다.

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:54:25) [MSC v.1900 64 bit (AM...
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

위와 같은 것을 대화형 인터프리터라고 하는데, 앞으로 이 책에서는 이 인터프리터로 파이썬 프로그래밍의 기초적인 사항들에 대해 설명할 것이다.

※ 대화형 인터프리터는 파이썬 셸(*Python shell*)이라고도 한다. 3개의 꺾은 괄호(>>>)는 프롬프트(*prompt*)라고 한다.

대화형 인터프리터를 종료할 때는 `ctrl+z` 를 누른다 (유닉스 계열에서는 `ctrl+d`). 또는 다음의 예와 같이 `sys` 모듈을 사용하여 종료할 수도 있다.

```
>>>import sys
>>>sys.exit()
```

파이썬 기초 문법 따라 해보기

여기서 소개하는 내용들은 나중에 다시 자세하게 다룰 것이니 이해가 되지 않는다고 절망하거나 너무 고심하지 말도록 하자.

파이썬 인터프리터를 실행하여 다음을 직접 입력해 보자.

사칙연산

1 더하기(+) 2는 3이라는 값을 출력해 보자. 보통 계산기 사용하듯 더하기 기호만 넣어 주면 된다.

```
>>>1 + 2
3
```

나눗셈(/)과 곱셈(*) 역시 예상한 대로 결과값을 보여준다.

```
>>>3 / 2.4
1.25
>>>3 * 9
27
```

우리가 일반적으로 알고 있는 ÷ 기호나 × 기호가 아닌 것에 주의하자.

변수에 숫자 대입하고 계산하기

```
>>>a = 1
>>>b = 2
>>>a + b
3
```

a에 1을, b에 2를 대입한 다음 a와 b를 더하면 3이라는 결과값을 보여 준다.

변수에 문자 대입하고 출력하기

```
>>>a = "Python"
>>>print(a)
Python
```

a라는 변수에 Python이라는 값을 대입한 다음 print(a) 라고 작성하면 a의 값을 출력한다.

※ 파이썬은 대소문자를 구분한다. *print*를 *PRINT*로 쓰면 정의되지 않았다는 에러 메시지가 나온다.

[변수에 복소수도 넣을 수 있을까?]

파이썬은 복소수도 지원한다

```
>>>a = 2 + 3j

>>>b = 3
>>>a * b
(6+9j)
```

변수 `a`에 `2+3j`라는 값을 대입하고, 변수 `b`에 `3`을 대입하였다. 여기서 `2+3j`란 복소수를 의미한다. 보통 우리는 고등학교 때 복소수를 표시할 때 알파벳 `i`를 이용해서 `2 + 3i`처럼 사용했지만 파이썬에서는 `j`를 사용한다. 위의 예는 `2 + 3j` 와 `3`을 곱하는 방법이다. 당연히 결과값으로 `6+9j`를 출력한다.

조건문 if

다음은 간단한 조건문 `if`를 이용한 예제이다.

```
>>>a = 3
>>>if a >1:
...     print("a is greater than 1")
...
a is greater than 1
```

※ `print`문 앞의 `'...'`은 아직 문장이 끝나지 않았음을 의미한다.

위 예제는 `a`가 1보다 크면 `"a is greater than 1"`이라는 문장을 출력(`print`)하라는 뜻이다. 위 예제에서 `a`는 3이므로 1보다 크다. 따라서 두 번째 `"..."` 이후에 `Enter` 키를 입력하면 `if`문이 종료되고 `"a is greater than 1"`이라는 문장이 출력된다.

`if a > 1:` 다음 문장은 `Tab` 키 또는 `Spacebar` 키 4개를 이용해 반드시 들여쓰기 한 후에 `print("a is greater than 1")` 이라고 작성해야 한다. 들여쓰기 규칙에 대해서는 05장 제어문에서 자세하게 알아볼 것이다. 바로 뒤에 이어지는 반복문 `for`, `while` 예제도 마찬가지로 들여쓰기가 필요하다.

반복문 for

다음은 `for`를 이용해서 `[1, 2, 3]`안의 값들을 하나씩 출력해 주는 것을 보여주는 예이다.

```
>>>for a in [1, 2, 3]:
...     print(a)
...
1
2
3
```

`for`문을 이용하면 실행해야 할 문장을 여러 번 반복해서 실행시킬 수 있다. 위의 예는 대괄호(`[]`) 사이에 있는 값들을 하나씩 출력한다. 위 코드의 의미는 `"[1, 2, 3]이라는 리스트의 앞에서부터 하나씩 꺼내어 a라는 변수에 대입한 후 print(a) 를 수행하라"`이다. 당연히 `a`에 차례로 1, 2, 3이라는 값이 대입되며 `print(a)`에 의해서 그 값이 차례대로 출력된다.

반복문 while

다음은 **while**을 이용하는 예이다.

```
>>> i = 0
>>> while i < 3:
...     i=i+1
...     print(i)
...
1
2
3
```

while이라는 영어 단어는 "~인 동안"이란 뜻이다. **for**문과 마찬가지로 반복해서 문장을 수행할 수 있도록 해준다. 위의 예제는 *i* 값이 3보다 작은 동안 `i=i+1` 과 `print(i)` 를 수행하라는 말이다. `i=i+1` 이라는 문장은 *i*의 값을 1씩 더하게 한다. *i* 값이 3보다 커지게 되면 **while**문을 빠져나가게 된다.

함수

파이썬의 **함수**는 다음과 같은 형태이다.

```
>>> def sum(a, b):
...     return a+b
...
>>> print(sum(3,4))
7
```

파이썬에서 **def**는 함수를 만들 때 사용하는 예약어이다. 위의 예제는 **sum**이라는 함수를 만들고 그 함수를 어떻게 사용하는지를 보여준다. `sum(a, b)`에서 *a*, *b*는 입력값이고, *a+b*는 결과값이다. 즉 3, 4가 입력으로 들어오면 3+4를 수행하고 그 결과값인 7을 돌려 준다.

이렇게 해서 기초적인 파이썬 문법에 대해서 간략하게 알아보았다.

파이썬 프로그램을 작성할 수 있는 여러 가지 에디터

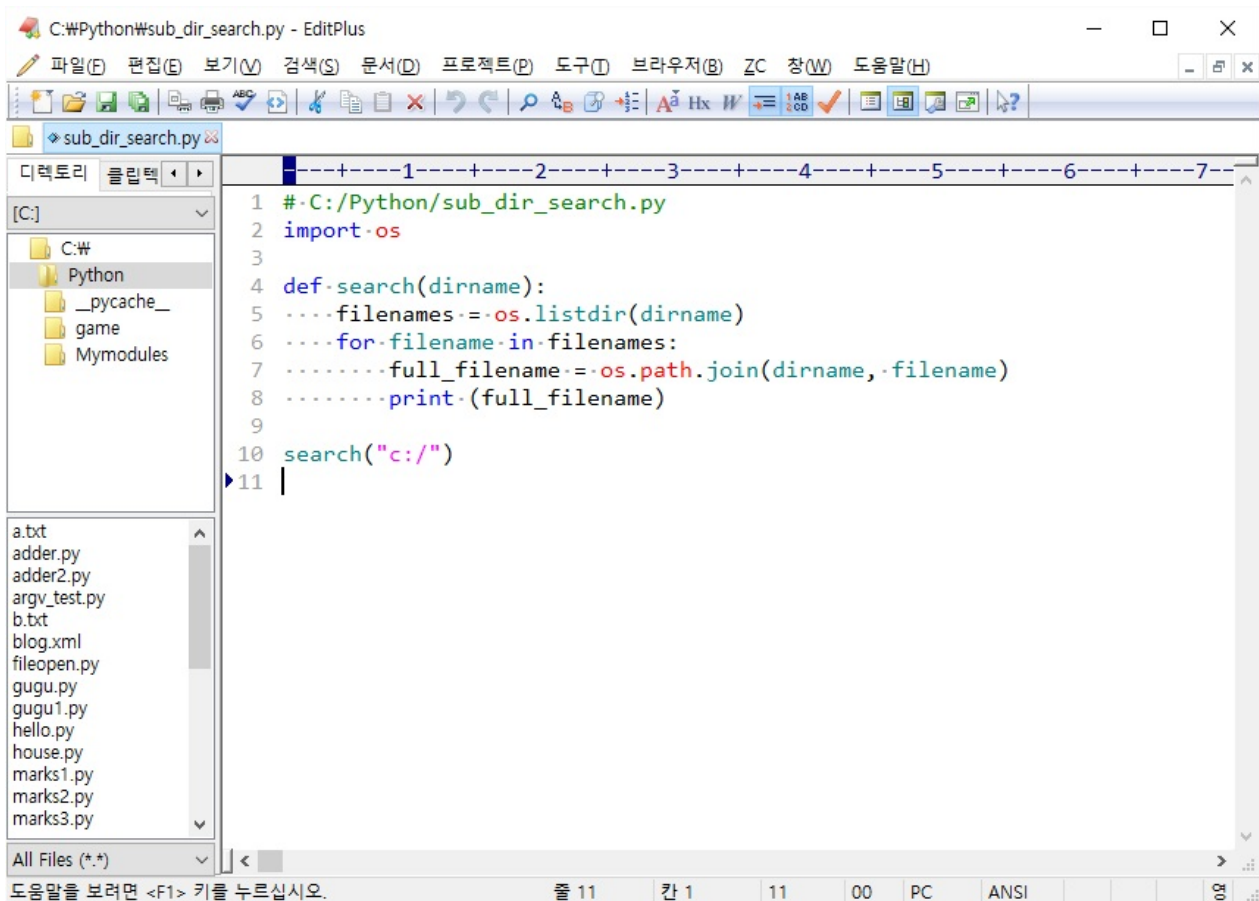
이미 눈치챘을지 모르지만 파이썬 프로그램을 편하게 작성하기 위해서는 인터프리터보다는 에디터를 이용해서 작성하는 것이 좋다. 에디터란 문서를 편집할 수 있는 프로그래밍 툴을 말한다.

에디터는 자신에게 익숙한 것을 사용하면 되는데, 아직 즐겨 사용하는 에디터가 마땅히 없는 독자에게 몇 가지 추천하고 싶은 에디터가 있다. 윈도우 사용자라면 에디트 플러스나 파이참(PyCharm), 노트패드++ 또는 서브라임 텍스트 3, 리눅스 사용자라면 당연히 vi 에디터를 추천한다. 물론 리눅스에는 이맥스라는 좋은 에디터가 있지만 초보자가 다루기는 쉽지 않다.

다음에 여러 가지 에디터를 소개해 두었으니 읽어 보고 자신에게 맞는 에디터를 선택하자. 에디터를 선택하기 어렵다면 파이썬 프로그래밍을 처음 시작하기에 좋은 에디트 플러스를 추천한다.

에디트 플러스

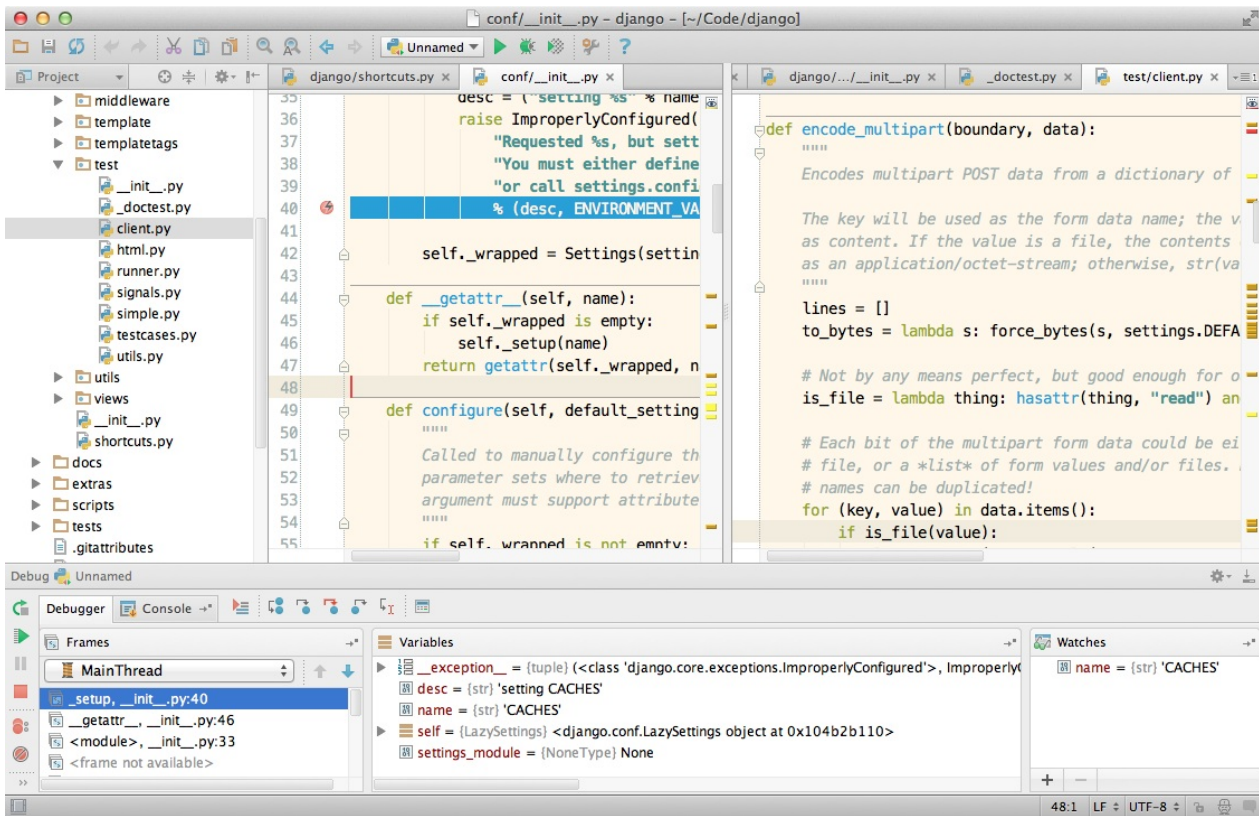
이 프로그램은 무료 소프트웨어가 아니기 때문에 평가판을 이용해야 한다. 다운로드 한 후부터 한 달간 사용할 수 있다. 에디트 플러스 공식 사이트 (<http://www.editplus.com/kr>) 에서 파일을 다운로드해 설치하자.



파이참

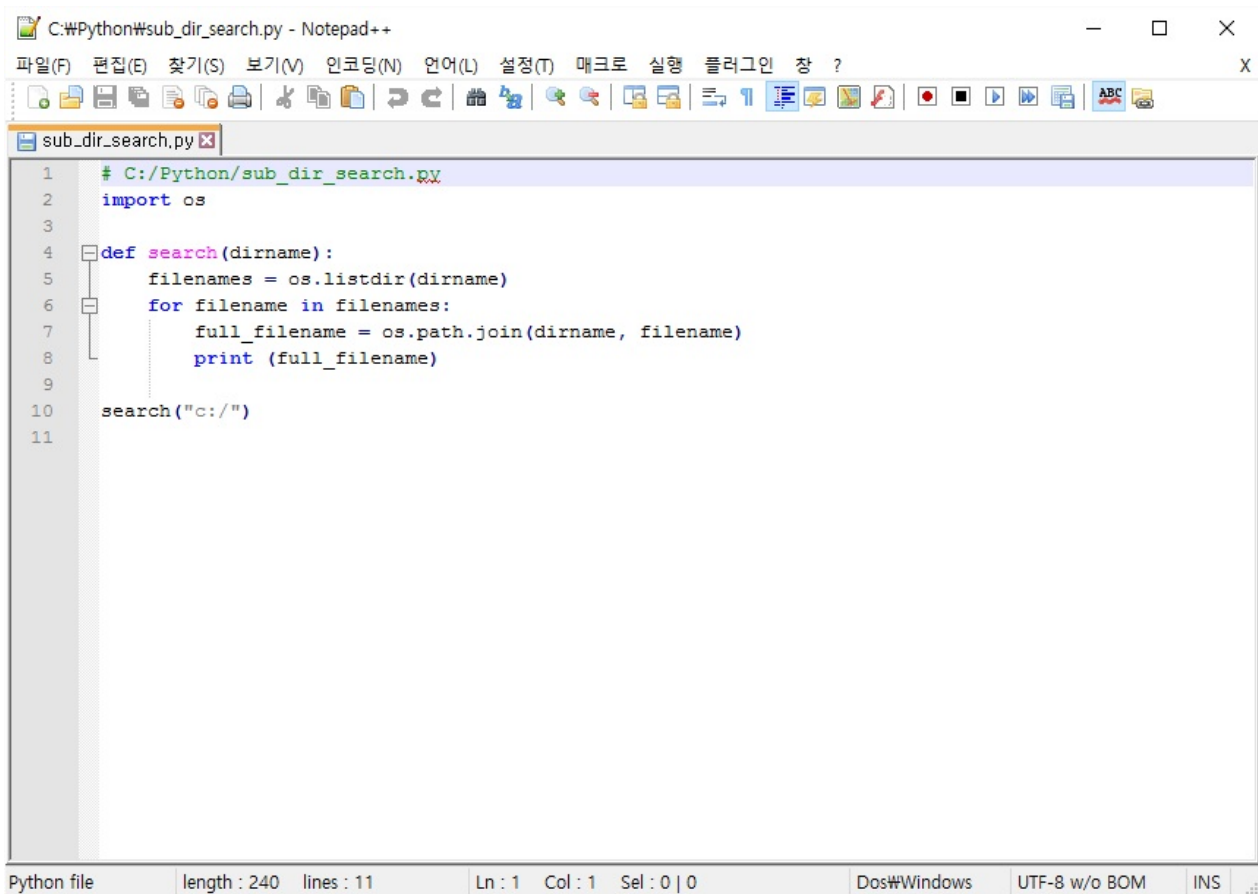
파이썬에 어느 정도 익숙해졌다면 파이참(PyCharm)을 사용해 보기를 적극 추천한다. 파이참은 가장 유명한 파이썬 에디터 중 하나로 코드 작성 시 자동 완성, 문법 체크 등 편리한 기능들을 많이 제공한다.

이 에디터는 파이참 공식 다운로드 사이트 (<http://www.jetbrains.com/pycharm/download>) 에서 다운로드할 수 있다.



노트패드++

노트패드++도 많은 사람들이 추천하는 윈도우용 파이썬 에디터 중의 하나이다. 이 에디터는 노트패드++ 공식 다운로드 사이트 (<https://notepad-plus-plus.org>) 에서 다운로드할 수 있다.



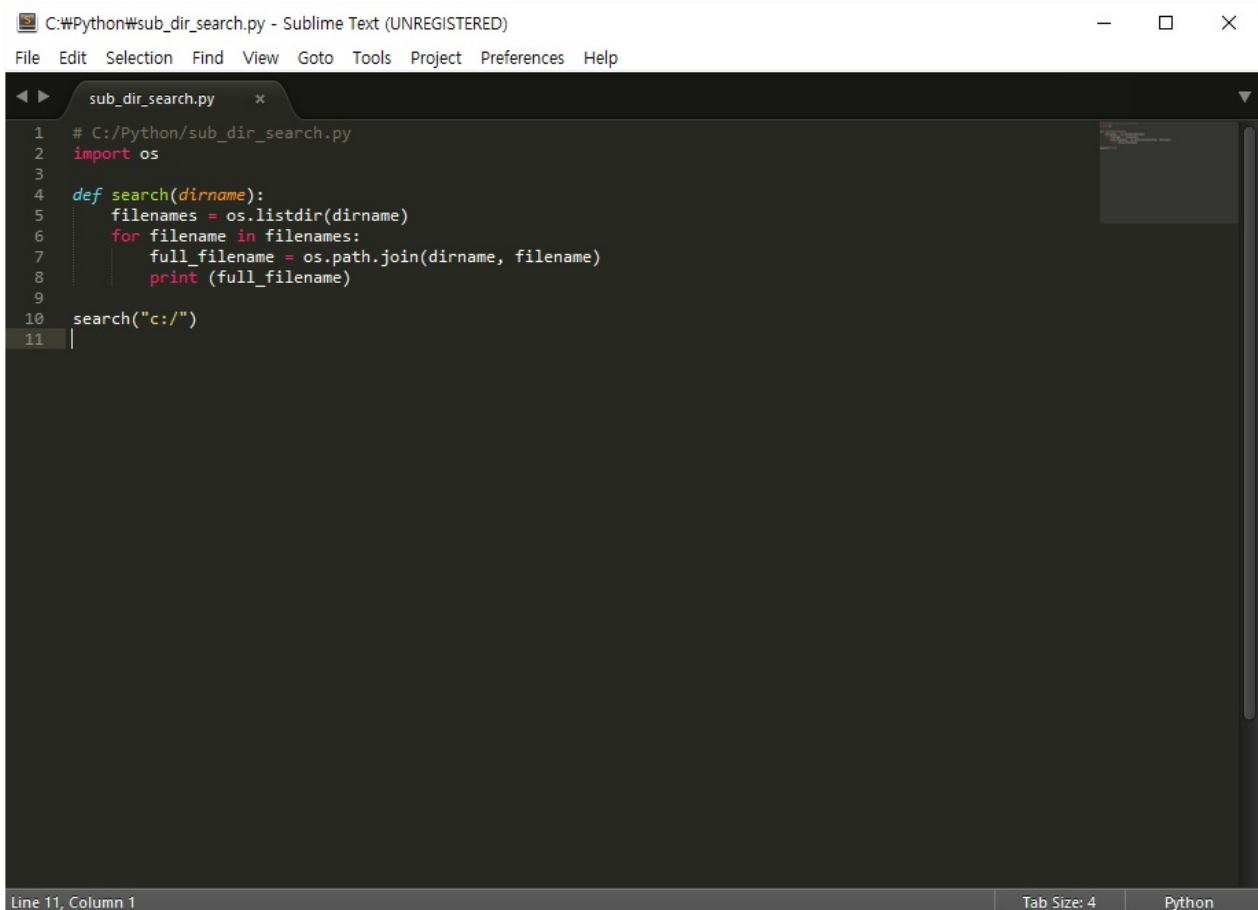
The screenshot shows a Notepad++ window titled "C:\Python\sub_dir_search.py - Notepad++". The menu bar includes File (F), Edit (E), Search (S), View (V), Encoding (N), Language (L), Settings (T), Macro, Run, Plugin, and Help (?). The toolbar contains various icons for file operations, editing, and development. The editor displays a Python script named "sub_dir_search.py" with the following code:

```
1 # C:/Python/sub_dir_search.py
2 import os
3
4 def search(dirname):
5     filenames = os.listdir(dirname)
6     for filename in filenames:
7         full_filename = os.path.join(dirname, filename)
8         print (full_filename)
9
10 search("c:/")
11
```

The status bar at the bottom indicates "Python file", "length : 240", "lines : 11", "Ln : 1", "Col : 1", "Sel : 0 | 0", "Dos#Windows", "UTF-8 w/o BOM", "INS", and a syntax icon.

서브라임 텍스트 3

서브라임 텍스트 3 역시 파이썬 사용자에게 사랑받는 에디터 중의 하나로 심플하면서 세련된 사용자 인터페이스를 자랑한다. 이 에디터는 서브라임 텍스트 3 공식 다운로드 사이트 (<http://www.sublimetext.com/3>) 에서 다운로드할 수 있다.



에디터로 파이썬 프로그램 작성하기

다음과 같은 프로그램을 에디터로 직접 작성해 보자.

```

# hello.py

print(
    "Hello world"
)

```

위의 파일에서 `# hello.py` 라는 문장은 주석이다. `#`으로 시작하는 문장은 `#`부터 시작해서 그 줄 끝까지 프로그램 수행에 전혀 영향을 주지 않는다. 주석은 프로그래머를 위한 것으로, 프로그램 소스에 설명문을 달 때 사용한다.

[여러줄짜리 주석문]

주석문이 여러 줄인 경우 다음의 방법을 사용하면 편리하다.

```
"""
Author: EungYong Park
Date : 2016-01-01
이 프로그램은 Hello World를 출력하는 프로그램이다.
"""
```

여러 줄로 이루어진 주석을 작성하려면 큰따옴표 세 개를 연속으로 사용한 `"""` 기호 사이에 주석문을 작성하면 된다. 큰따옴표 대신 작은따옴표 세 개를(`'''`)를 사용해도 된다.

앞에서와 같이 작성한 파일을 `hello.py`라는 이름으로 `C:\Python` 디렉터리에 저장하자. 에디터로 파이썬 프로그램을 작성한 후 저장할 때는 파일 이름의 확장자명을 항상 `py`로 해야한다. `py`는 파이썬 파일임을 알려주는 관례적인 확장자명이다.

이제 이 `hello.py`라는 프로그램을 실행시키기 위해 [윈도우+R -> cmd 입력 -> Enter]를 눌러 도스창을 연다.

`hello.py`라는 파일이 저장된 곳으로 이동한 후 다음과 같이 입력한다. 이 책에서는 `hello.py` 파일을 `C:\Python` 디렉터리에 저장했다.

```
C:\Users\home>cd C:\Python
C:\Python>python hello.py
Hello World
```

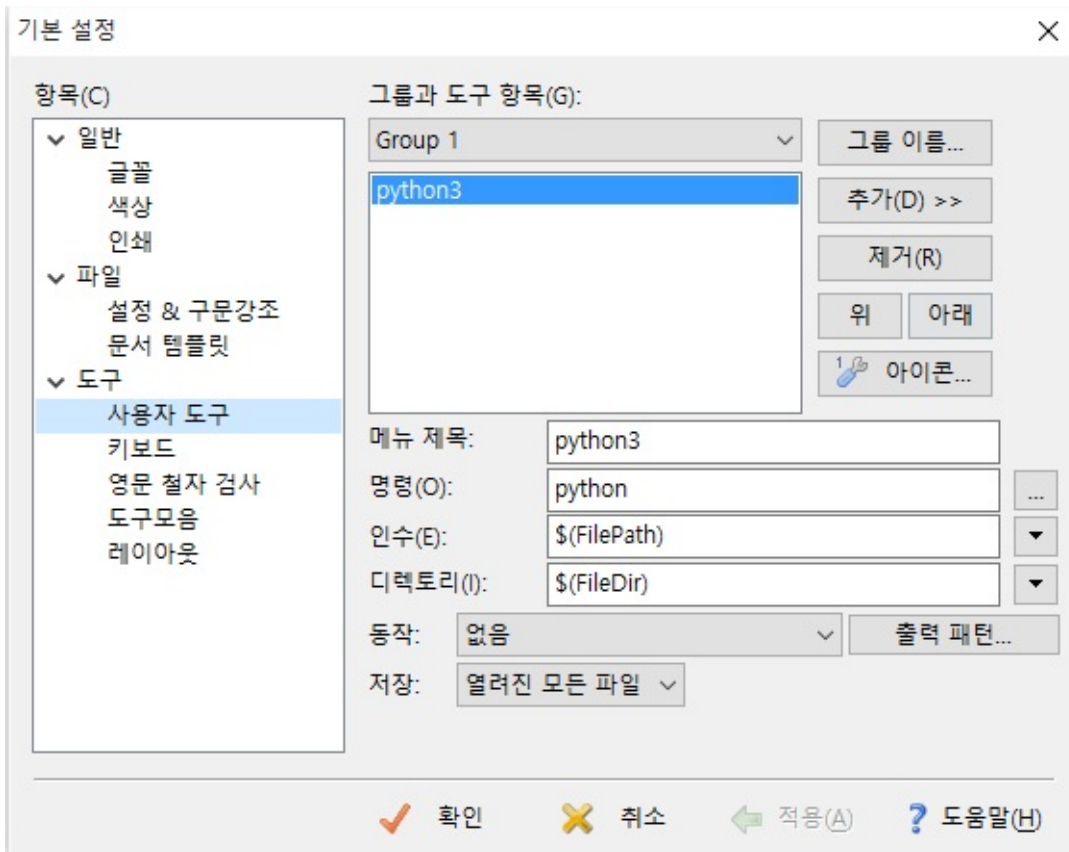
위와 같은 결과값을 볼 수 있을 것이다. 결과값이 위와 같지 않다면 `hello.py` 파일이 `C:\Python` 디렉터리에 존재하는지 다시 한 번 살펴보도록 하자.

이번 예제에서는 Hello world라는 문장을 출력하는 단순한 프로그램을 에디터로 작성했지만 보통 에디터로 작성하는 프로그램은 꽤 여러 줄로 이루어진다. 여기서 중요한 사실은 에디터로 만든 프로그램은 파일로 존재한다는 점이다. 대화형 인터프리터에서 만든 프로그램은 인터프리터를 종료함과 동시에 사라지지만 에디터로 만든 프로그램은 파일로 존재하기 때문에 언제든지 다시 사용할 수 있다.

왜 대부분이 에디터를 이용해서 파이썬 프로그램을 작성하는지 이제 이해가 될 것이다.

[에디트 플러스로 파이썬 프로그램 쉽게 실행하기]

에디트 플러스를 실행한 후 메뉴바에서 [도구 → 기본 설정]을 선택한 다음 [도구 → 사용자 도구]를 선택하면 다음과 같은 화면이 나타난다.



위 화면처럼 내용이 채워지도록 다음과 같이 수정하자.

1. [추가 → 프로그램]을 선택한다.
2. 메뉴 제목 : `python3` 라고 입력한다.
3. 명령 : `python` 이라고 입력한다.
4. 인수 : 오른쪽 버튼을 누르고 첫 번째 항목인 "파일 경로"를 선택한다. 입력창에 `$(FilePath)` 가 자동으로 입력된다.
5. 디렉토리 : 오른쪽 버튼을 누르고 첫 번째 항목인 "파일 디렉토리"를 선택한다. 입력창에 `$(FileDir)` 이 자동으로 입력된다.
6. 하단의 [적용] 버튼을 클릭하여 설정을 저장한다.

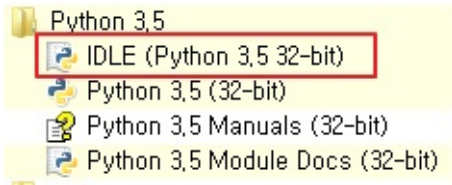
이제 에디트 플러스로 `hello.py`와 같은 프로그램을 작성하고 저장한 후 `ctrl+1` 을 누르면 `hello.py` 프로그램이 자동으로 실행되는 것을 확인할 수 있다.

파이썬 IDLE

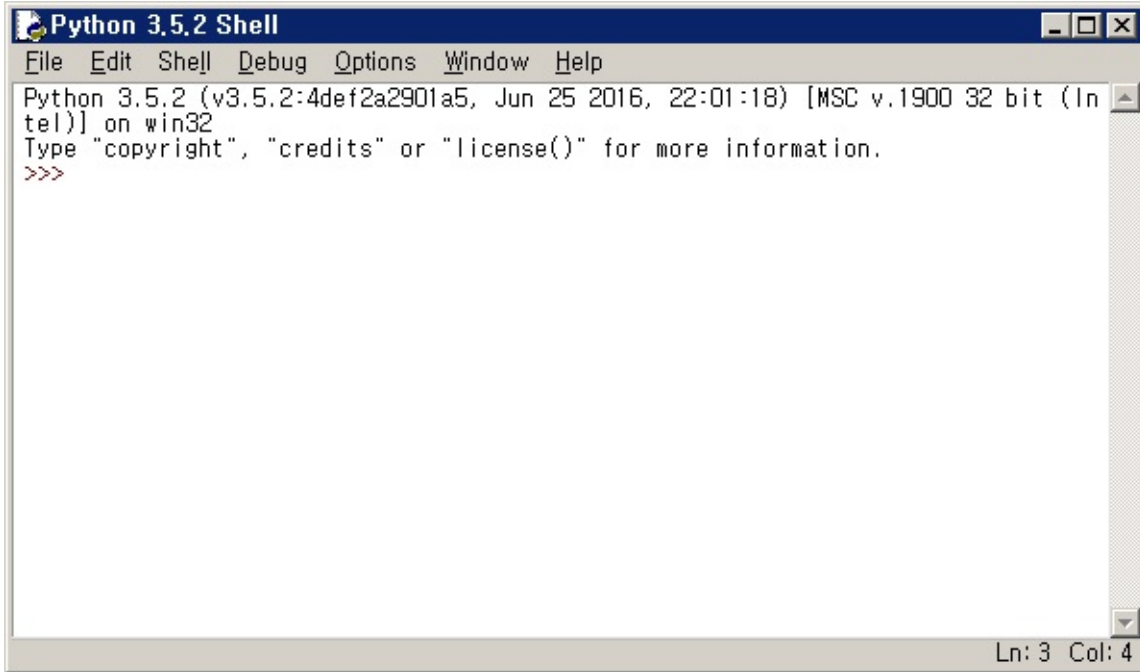
파이썬 IDLE(Integrated Development and Learning Environment)는 파이썬 프로그램 작성을 도와주는 통합 개발환경으로 파이썬 설치시 기본으로 설치되는 프로그램이다. IDLE를 가지고 전문적인 파이썬 프로그램을 만들기에는 좀 부족하지만 파이썬 공부에 목적이거나 더할 나위없이 좋은 도구가 될 것이다.

파이썬 IDLE를 실행 해 보자.

[시작 -> 모든 프로그램 -> Python 3.5 -> IDLE 선택]



그러면 다음과 같은 IDLE 셸(Shell) 창이 나타난다.



IDLE는 크게 두가지 창으로 구성된다.

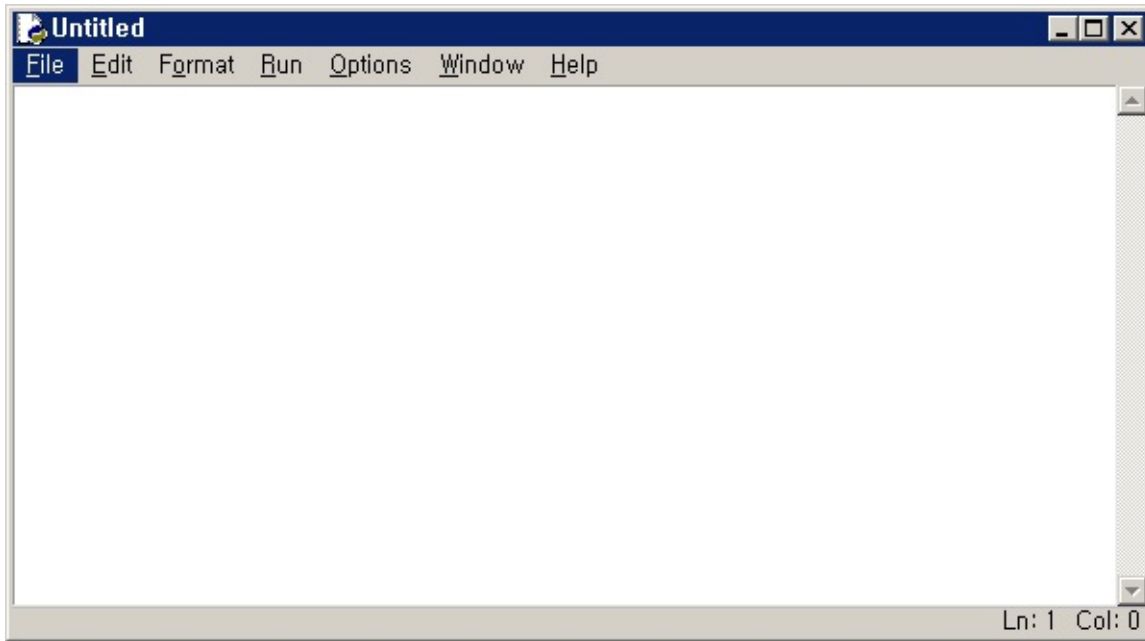
- 셸 창(Shell Window) - 파이썬 셸(Python Shell)이 실행되는 창
- 에디터 창(Editor Window) - 파이썬 에디터(Editor)가 실행되는 창

IDLE 실행 시 가장 먼저 나타나는 창은 셸 창이다. 이 곳에서 파이썬 명령들을 수행하고 테스트 해 볼 수 있다.

이번에는 IDLE 에디터(Editor)를 실행 해 보자.

셸 창 메뉴에서 [File -> New File]을 선택하자.

다음 그림과 같은 IDLE 에디터가 나타날 것이다.



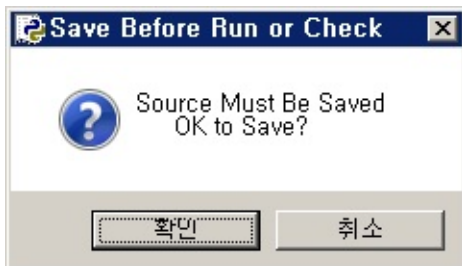
이제 다음과 같은 프로그램을 IDLE 에디터에서 직접 작성해 보자.

```
# hello_idle.py
print(
    "Hello IDLE"
)
```

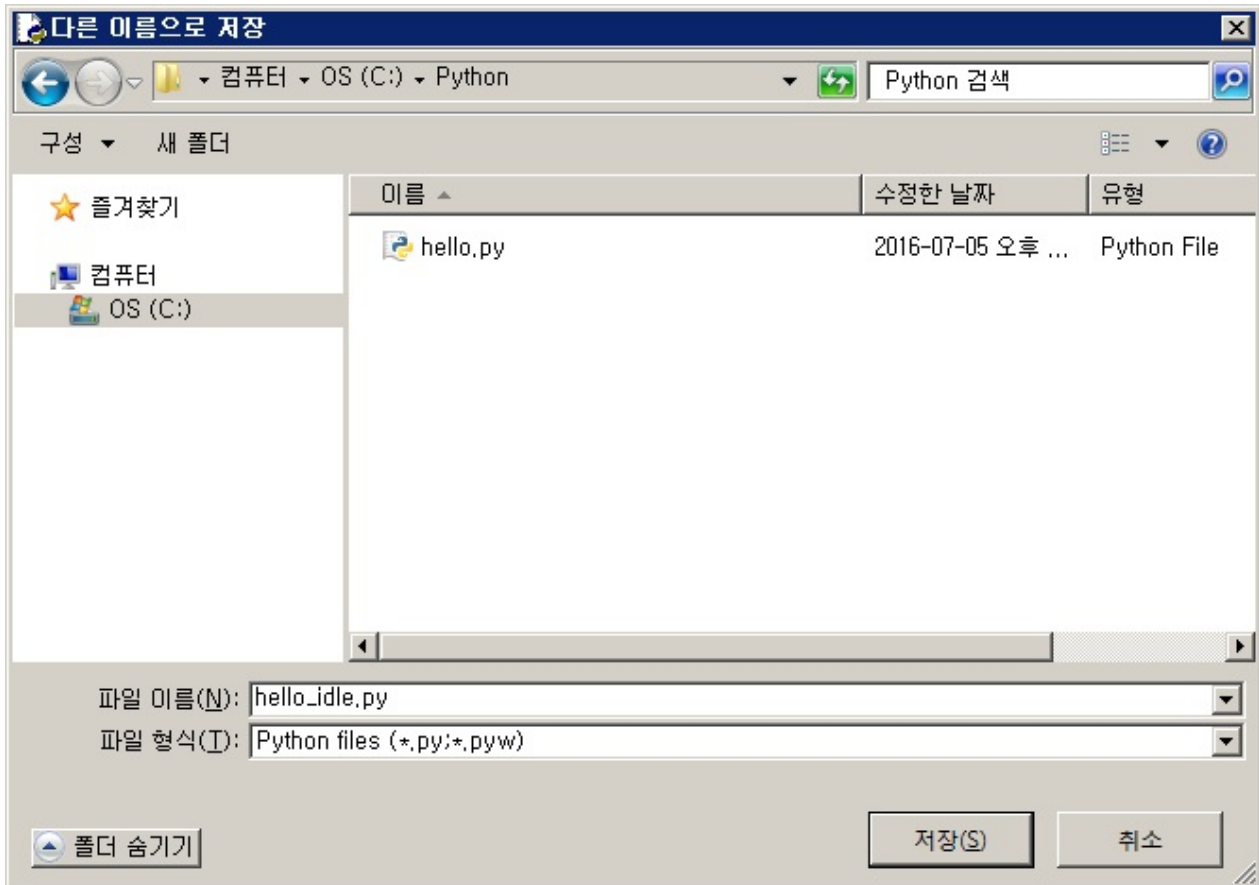
이제 작성한 프로그램을 실행 해 보자.

메뉴에서 [Run -> Run Module]을 선택하자. (단축키: F5)

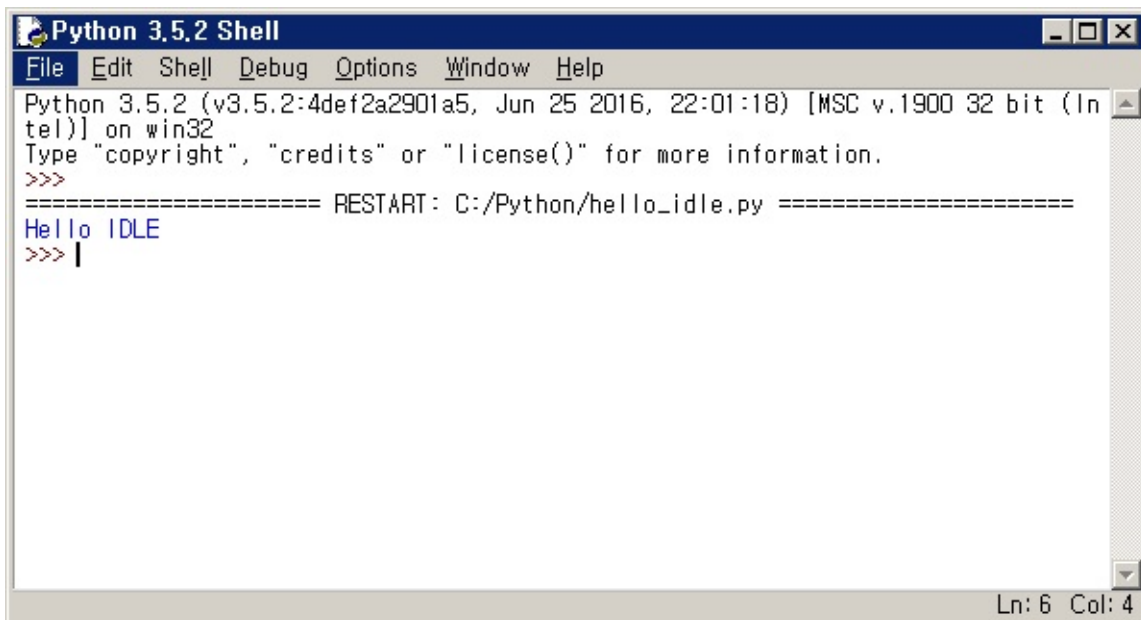
실행 해 보면 파일을 먼저 저장하라는 다음과 같은 다이얼로그가 나올것이다.



"확인"을 선택하고 `c:\Python` 이라는 디렉터리에 `hello_idle.py`라는 이름으로 저장을 하도록 하자.



파일을 저장하면 자동으로 파이썬 프로그램이 실행 된다. 실행 결과는 다음과 같이 IDLE 셸 창에 표시된다.



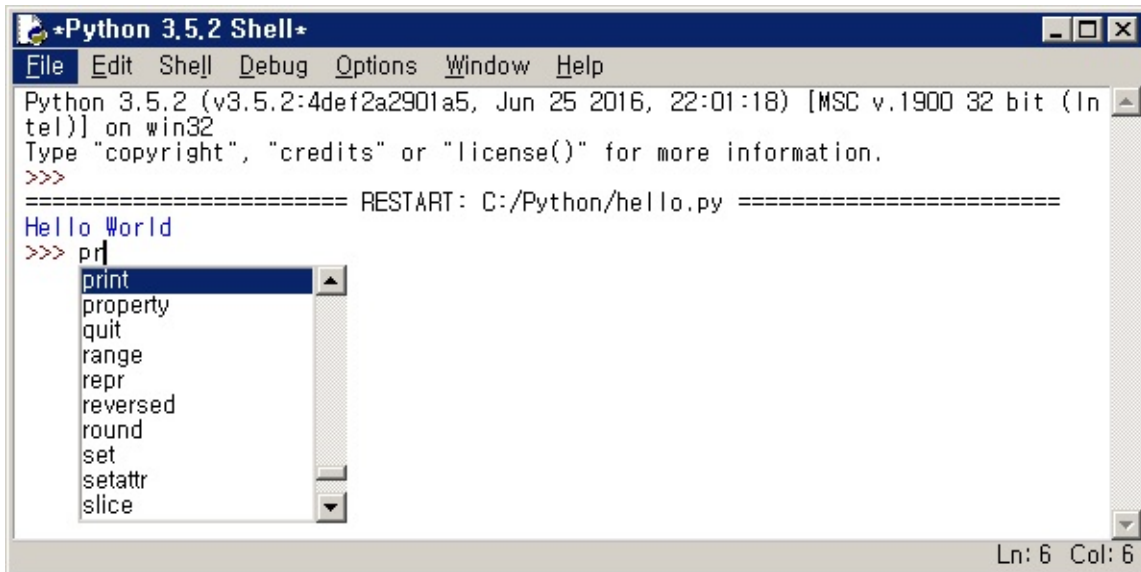
프로그램을 재 실행하려면 에디터 창에서 F5키로 다시한번 실행하면 된다.

IDLE를 사용하면 자동완성(auto completion) 기능을 사용할 수 있다.

IDLE 셸 창이나 에디터 창에서 다음과 같이 입력 해 보자.

```
pr + [Tab]
```

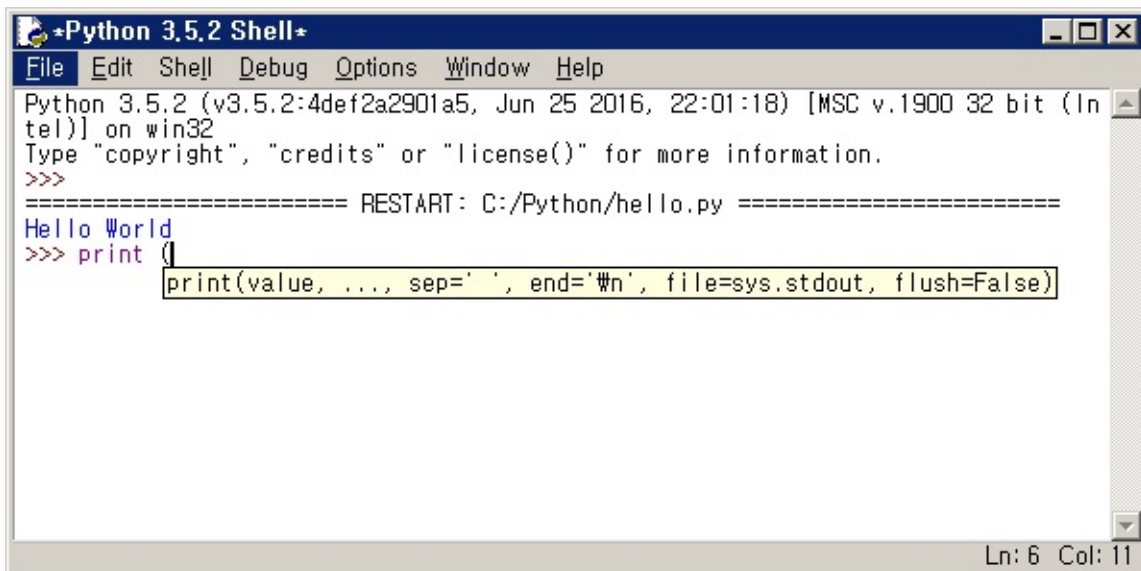
pr로 시작하는 파이썬 키워드가 콤보박스 가장 상단에 표시될 것이다.



print를 선택하고 (선택할 때는 스페이스바로 선택한다.) 다음과 같이 왼쪽 괄호만 입력 해 보자.

```
print(
```

그러면 다음과 같이 print라는 함수의 도움말이 표시(call tips)되는 것을 볼 수 있다.



파이썬 IDLE는 이 외에도 자동 들여쓰기, 코드를 컬러로 표시하기 등의 여러 유용한 기능들을 제공한다. 나머지 IDLE의 메뉴나 기타 기능들에 대해서는 IDLE를 사용하면서 차츰 익혀나가도록 하자.

기본 데이터 타입

1. Python 기본 데이터 타입

파이썬에 사용되는 기본 데이터 타입(Scalar Data Type)에는 아래와 같은 타입들이 있다.

타입	설명	표현 예
int	정수형 데이터	100, 0xFF (16진수), 0o56 (8진수)
float	소수점을 포함한 실수	a = 10.25
bool	참, 거짓을 표현하는 부울린	a = True
None	Null과 같은 표현	a = None

정수형은 소수점을 갖지 않는 정수를 갖는 데이터 타입이며, float는 소수점을 갖는 데이터 타입이다. bool 타입은 True 혹은 False 만을 갖는 타입이고, None은 아무 데이터를 갖지 않는다는 것을 표현하는 것으로 타 언어의 Null과 같은 개념이다. 정수형에 리터럴 데이터를 넣을 때, 10진수 이외에 16진수 혹은 8진수를 위의 예와 같이 사용할 수 있다.

리터럴 데이터를 특정 타입으로 변경하기 위하여 int(), float(), bool() 등과 같은 타입 생성자를 사용할 수 있다. 예를 들어, int(3.5)는 float 데이터를 정수형으로 변경하여 정수값 3 을 리턴한다. float("1.6")은 float형 1.6을 리턴한다.

```
int(3.5)      # 3
2e3          # 2000.0
float("1.6") # 1.6
float("inf")  # 무한대
float("-inf") # -무한대
bool(0)       # False. 숫자에서 0만 False임,
bool(-1)      # True
bool("False") # True
a = None      # a는 None
a is None     # a가 None 이므로 True
```

위의 예제에서 리터럴 2e3 (혹은 2E3 도 같은 표현)은 $2 * (10 ** 3)$ 과 같은 표현이다. 또한, bool 타입은 숫자의 경우 0 만이 거짓이 되고, 0이 아니면 참이 된다. bool() 안에 문자형이나 컬렉션 타입들이 있을 경우 비어있으면 거짓이 되고 값이 있으면 참이된다. 즉, 위의 마지막 예인 bool("False")는 문자열이 비어있지 않으므로 참이 된다.

2. 복소수

파이썬은 복소수 타입을 지원하는데, 복소수는 $a+bj$ 와 같이 표현된다 (수학에서 복소수를 표현할 때 i 를 사용하지만 파이썬에서는 j 를 사용한다). 실수부의 값을 얻기 위해서는 복소수 변수명.real 을, 허수부의 값을 얻기 위해선 변수명.imag 를 사용한다.

```
v = 2 + 3j
v.real # 2
v.imag # 3
```

연산자

파이썬은 산술연산자, 비교연산자, 할당연산자, 논리연산자, Bitwise 연산자, 멤버십연산자, Identity연산자를 지원한다.

1. 산술연산자

산술연산자에는 (1) 사칙연산자 $+$, $-$, $*$, $/$ 와 (2) 제곱을 나타내는 $**$, (3) 나머지를 산출하는 $%$ (Modulus), 그리고 (4) 나누기에 소숫점 이하를 버리는 $//$ 연산자(Floor Division) 등이 있다.

```
5 % 2 # 1
5 // 2 # 2
```

2. 비교연산자

비교연산자는 관계연산자로도 불리우는데, 여기에는 등호($==$), 같지 않음($!=$), 부등호($<$, $>$, $<=$, $>=$) 등이 있다.

```
if a != 1:
    print("1이 아님")
```

3. 할당연산자

할당연산자는 변수에 값을 할당하기 위하여 사용되는데, 기본적으로 $=$ (Equal Sign)을 사용한다. 산술연산자와 함께 사용되어 할당을 보다 간결히 하기 위해 사용되는 $+=$, $-=$, $*=$, $/=$, $%=$, $//=$ 등과 같은 연산자도 할당연산자에 해당된다.

```
a = a * 10
a *= 10 # 위와 동일한 표현
```

4. 논리연산자

논리연산자에는 and, or, not 이 있는데, and 는 양쪽의 값이 모두 참인 경우만 참이 되고, or 는 어느 한쪽만 참이면 참이된다. not 은 참이면 거짓으로 거짓이면 참이 된다. 아래 예제는 No가 출력된다.

```
x = True
y = False

if x and y:
    print("Yes")
else:
    print("No")
```

5. Bitwise 연산자

Bitwise연산자에는 & (AND), | (OR), ^ (XOR), ~ (Complement), <<, >> (Shift)가 있는데, 이 연산자는 비트단위의 연산을 하는데 사용된다.

```
a = 8      # 0000 1000
b = 11     # 0000 1011
c = a
&
b # 0000 1000 (8)
d = a ^ b # 0000 0011 (3)

print(c)
print(d)
```

6. 멤버십 연산자

멤버십연산자에는 in, not in 이 있는데, 이는 좌측 Operand가 우측 컬렉션에 속해 있는지 아닌지를 체크한다.

```
a = [1,2,3,4]
b = 3 in a    # True
print(b)
```

7. Identity 연산자

Identity연산자에는 is, is not 이 있는데, 이는 양쪽 Operand가 동일한 Object를 가리키는지 아닌지를 체크한다.

```
a = "ABC"
b = a
print(a is b) # True
```

문자열과 바이트

1. 문자열

파이썬에서 문자열은 단일인용부호(') 혹은 이중인용부호(") 를 사용하여 표현한다.

예를 들어, 아래 표현은 s 라는 변수에 가나다 라는 문자열을 할당하는 것으로 동일한 표현이다.

```
s = '가나다'
s = "가나다"
```

만약 여러 라인에 걸쳐 있는 문자열을 표현하고 싶다면, ''' 또는 """ 처럼 3개의 인용부호를 사용한다.

```
s = '''아리랑
아리랑
아라리요
'''
print(s)
```

복수 라인 문자열을 한 라인으로 표현하고 싶다면, Escape Sequence (\n)를 사용하면 된다. 즉, 다음 표현은 위와 동일한 표현이다.

실제 리눅스나 Mac OS에서는 Newline이 \n으로 표현되지만, 윈도우즈에서 \r\n을 사용한다. 하지만, 파이썬에서는 Universal Newline이 지원되어 모든 OS에서 공히 \n을 사용한다.

```
s = '아리랑\n아리랑\n아라리요'
print(s)
```

물론 문자열에서 사용되는 Escape Sequence에는 타 언어와 비슷하게 여러 가지를 사용할 수 있다. 예를 들어 탭은 \t, 이중따옴표는 \", 백슬래시는 등과 같이 표현한다.

문자열 포매팅

일정한 포맷에 맞춰 문자열을 조합하는 것을 문자열 포매팅이라하는데, 문자열 포맷 템플릿 안에 대입값이 들어갈 자리를 지정해 두고 나중에 그 값을 채워 넣는 방식이다. 예를 들어, "답: %s" % "A" 와 같은 표현에서 % 앞 부분은 포맷 템플릿이고, % 뒤는 실제 대입할 값이다. 이때 % 를 포맷

팅 연산자 (Formatting Operator)라 부른다. % 앞뒤로 각각 하나의 값만을 받아들이므로 만약 % 뒤의 값이 복수 개이면 튜플로 묶어주어야 한다.

```
p = "이름: %s 나이: %d" % ("김유신", 65)
print(p)
# 출력: 이름: 김유신 나이: 65

p = "X = %0.3f, Y = %10.2f" % (3.141592, 3.141592)
print(p)
# 출력: X = 3.142, Y =          3.14
```

% (Formatting Operator) 앞의 포맷 템플릿에는 %s, %d 등과 같이 대입값 형식을 지정해 주는데 이를 변환 지시어(Conversion Specifier)라 부른다. 아래 표는 Conversion Specifier 들의 의미를 설명한 것이다.

Conversion Specifier	의미
%s	문자열 (파이썬 객체를 str()을 사용하여 변환)
%r	문자열 (파이썬 객체를 repr()을 사용하여 변환)
%c	문자(char)
%d 또는 %i	정수 (int)
%f 또는 %F	부동소수 (float) (%f 소문자 / %F 대문자)
%e 또는 %E	지수형 부동소수 (소문자 / 대문자)
%g 또는 %G	일반형: 값에 따라 %e 혹은 %f 사용 (소문자 / 대문자)
%o 또는 %O	8진수 (소문자 / 대문자)
%x 또는 %X	16진수 (소문자 / 대문자)
%%	% 퍼센트 리터럴

Conversion Specifier는 % 와 Conversion 문자(예: s, d, f) 사이에 전체 자릿수와 소숫점 뒤자리 수를 지정할 수 있다. 예를 들어 %10.2f 는 전체 10자리이고 값이 적으면 앞에 빈칸을 채우게 되고, .2 는 소숫점 2째 자리까지만 표시한다는 것을 의미한다. 만약 %-10.2f 처럼 마이너스로 표현하면 전체 10자리인데 왼쪽으로 정렬한다는 의미이다.

2. str (문자열 클래스)

문자열은 내부적으로 **str** 이라는 클래스 타입인데, 파이썬의 문자열은 기본적으로 **유니코드**이고, 한번 설정되면 다시 변경시킬 수 없는 **Immutable** 타입이다.

문자열은 인덱스를 사용하여 문자열 중 특정위치의 문자를 표현할 수 있다. 인덱스는 0로부터 시작하는데, 문자열 s 에 대하여 첫번째 문자는 s[0], 두번째 문자는 s[1] 과 같이 표현된다.

```
s = "ABC"
type(s)      # class 'str'
v = s[1]     # B
type(s[1])   # class 'str'
```

파이썬에는 C, C# 등에서 존재하는 문자(char) 타입이 존재하지 않는다. 따라서, 위의 예에서 s[1]의 타입이 char가 아닌 문자열 str 타입이 된다. 참고로 type(변수명)은 해당 변수의 타입을 리턴한다.

문자열을 표현할 때, **r'문자열'** 과 같이 사용하면, 이는 Escape Sequence를 표현하지 않고 Raw String을 직접 사용한다는 것을 의미한다. 예를 들어, 윈도우즈에서 파일경로를 간략히 표현하기 위해 아래와 같이 Raw String 표현을 사용할 수 있다.

```
path = r'C:\Temp\test.csv'
print(path)
```

3. 자주 사용되는 str 메서드

문자열 str 클래스에는 여러 유용한 메서드들이 제공되고 있는데, 이 중 흔히 사용되는 몇가지만 소개한다.

str.join()

우선 여러 개의 문자열을 하나로 결합하는 join() 메서드가 있는데, join() 메서드는 문자열을 결합하는데 사용되는 Separator를 join 메서드 앞에 사용한다. 아래 예제에 보듯이, 콤마를 사용하여 문자열 리스트 요소들을 결합할 수도 있으며, 또한 빈 문자열을 사용하여 문자열들을 결합하는 방법도 자주 사용된다.

```
s = ','.join(['가나', '다라', '마바'])
print(s)
# 출력: 가나,다라,마바

s = ''.join(['가나', '다라', '마바'])
print(s)
# 출력 : 가나다라마바
```

str.split()

split() 메서드는 join() 메서드의 반대로서 특정 separator를 기준으로 문자열을 분리하여 리스트를 리턴한다. 아래 예제에서 split() 메서드는 하나의 문자열을 콤마로 분리해서 3개의 요소를 갖는 리스트를 리턴한다.

```
items = '가나,다라,마바'.split(',')
print(items)
# 출력 : ['가나', '다라', '마바']
```

str.partition()

partition() 메서드는 문자열을 partition() 메서드의 첫번째 파라미터로 분리하여 그 앞부분(prefix), partition 분리자(separator), 뒷부분(suffix) 등 3개의 값을 Tuple로 리턴한다. 아래 예제는 Dash (-) 로 문자열을 분리하여 3개의 값을 리턴하는 코드이다. 일반적으로 separator는 사용하지 않아서 _ 를 사용하였다.

```
departure, _, arrival = "Seattle-Seoul".partition('-')
print(departure)
# 출력 : Seattle
```

str.format()

마지막으로 str 클래스에서 가장 많이 사용되는 메서드 중의 하나로 format() 메소드를 들 수 있다. format() 메서드는 다양한 방식의 문자열 포매팅을 지원하는데, 아래는 흔히 사용되는 3가지 방식을 예시하고 있다. 먼저 위치를 기준으로 한 포매팅은 {0},{1},... 등의 필드들을 format() 파라미터들의 순서대로 치환하게 된다. 두번째 필드명을 기준으로 한 포매팅은 {name}, {age}와 같이 임의의 필드명을 지정하고 format() 파라미터에 이들 필드명을 사용하여 값을 지정하는 것이다. 그리고 세번째 인덱스 및 키 사용 방식은 Python 오브젝트가 format()의 파라미터로 지정되고, 포맷에서 이 오브젝트의 인덱스(컬렉션의 경우) 혹은 속성, 키 등을 이용하는 것이다.

```
# 위치를 기준으로 한 포매팅
s = "Name: {0}, Age: {1}".format("강정수", 30)
print(s) #출력: Name: 강정수, Age: 30

# 필드명을 기준으로 한 포매팅
s = "Name: {name}, Age: {age}".format(name="강정수", age=30)
print(s) #출력: Name: 강정수, Age: 30

# object의 인덱스 혹은 키를 사용하여 포매팅
area = (10, 20)
s = "width: {x[0]}, height: {x[1]}".format(x = area)
print(s) #출력: width: 10, height: 20
```

4. bytes (바이트 클래스)

bytes 클래스는 일련의 바이트들을 표현하는 클래스로서 bytes는 한번 설정되면 다시 변경할 수 없는 Immutable 타입이다. 문자열을 바이트로 표현하기 위해 b'문자열' 와 같이 접두어 b를 앞에 붙인다.

```
[>>> text = b"Hello"
>>> for c in text:
[...     print(c)
[...
72
101
108
108
111
>>> █
```

문자열을 바이트들로 변경하는 인코딩을 위해 `encode()`를 사용한다. 반대로 바이트들을 문자열로 변경하는 디코딩을 위해 `decode()`를 사용한다. 문자열 안에서 유니코드값을 사용하려면, `\u`에 이어 유니코드값을 적으면 된다. 아래 예제는 문자열 `s1`을 UTF-8 인코딩을 사용하여 바이트들로 변경하고, 이를 다시 문자열로 디코딩하는 예제이다.

```
>>>
>>> s1 = "A\u00e5"
>>> s1
'Aå'
>>> b = s1.encode('utf-8')
>>> b
b'A\xc3\xa5'
>>> s2 = b.decode('utf-8')
>>> s2
'Aå'
>>> █
```

조건문 : if

파이썬에서 조건문을 사용하기 위하여 `if` 문을 사용한다. `if` 키워드 다음에 조건식을 적게 되고, 조건식 다음에 콜론(:)을 써서 `if` 조건식 끝을 표현한다 (`if`문은 한 라인에 모두 쓸 수 있으므로 문법상 조건식 뒤에 콜론 사용).

```
if x
<
10:
    print(x)
    print("한자리수")

# 한 라인에서 표현된 if 문
if x
<
100:    print(x)
```

`if` 문 조건식이 참이 아닐 때, 다음 `if` 문을 체크하기 위해서 `elif` 문을 사용할 수 있고, 모든 `if` 문이 거짓일 때 `else` 문 불력을 실행할 수 있다. 아래 예제는 `if...elif...else`를 모두 사용한 예이다. 파이썬에는 다른 언어에 있는 `switch` 문이 존재하지 않으므로, `switch` 문 기능은 `if...elif...elif...` 문으로 수행한다.

```
x = 10
if x < 10:
    print("한자리수")
elif x < 100:
    print("두자리수")
else:
    print("세자리 이상")
```

if 조건문 안에서 특정 블록/문장을 수행하지 않고 그냥 Skip하기 위하여 **pass** 라는 키워드를 사용할 수 있다. 아래 예제는 n 이 10보다 작은 경우는 아무 문장도 실행하지 않고 지나가고, 10보다 크거나 같을 때는 n 값을 출력한다.

```
if n < 10:
    pass
else:
    print(n)
```

반복문

1. 반복문 : while

파이썬에서 반복되는 루프를 만들기 위해 while 문이나 for 문을 사용할 수 있다. 먼저 while문은 while 키워드 다음의 조건식이 참일 경우 계속 while 안의 블록을 실행한다. 예를 들어, 아래의 예제를 보면 while의 조건식은 i가 10보다 작거나 같은 경우인데, i 값이 이 조건하에 있으면 계속 루프를 돌게 된다. 따라서, 아래 예제는 1부터 10까지 값을 출력하게 된다.

```
i=1
while i <= 10:
    print(i)
    i++
```

2. 반복문 : for

반복문 for는 C#, Java 에서의 foreach 와 비슷한 것으로, 컬렉션으로부터 하나씩 요소(element)를 가져와, 루프 내의 문장들을 실행하는 것이다. 리스트, Tuple, 문자열 등의 컬렉션은 "for 요소변수 in 컬렉션" 형식에서 in 뒤에 놓게 된다.

아래 예제는 0부터 10까지를 더하는 코드이다. 파이썬 내장함수인 range(n) 함수는 0 부터 n-1 까지의 숫자를 갖는 리스트를 리턴한다. for 루프는 이 리스트 컬렉션으로부터 요소를 하나씩 가져와서 for 블록의 문장을 실행하게 된다.


```
sum = 0
for i in range(11):
    sum += i
print(sum)
```

아래 예제는 문자열 요소를 갖는 리스트로부터 각 문자열들을 순차적으로 출력하는 예이다.

```
list = ["This", "is", "a", "book"]
for s in list:
    print(s)
```

for 루프는 이 밖에도 리스트 안에 내포되어 사용될 수도 있는데, 이는 [리스트 컬렉션](#) 편에서 자세히 설명한다.

3. break / continue

반복문 안에서 루프를 빠져나오기 위해 **break** 문을 사용할 수 있다. 또한, **continue**문을 사용하면 루프 블록의 나머지 문장들을 실행하지 않고 다음 루프로 직접 돌아가게 할 수 있다. 아래 예제는 **break**와 **continue**문을 사용 예시를 위한 것으로, *i*가 5인 경우는 **continue**가 실행되어 직접 다시 **while**문으로 이동하여 밑의 합계에 포함되지 않는다. 또한, *i*가 10보다 큰 경우 **while** 루프를 빠져나오게 된다. 따라서, 이 예제는 1부터 10까지 합을 구하는데, 5인 경우만 제외한 값 즉 50을 출력한다.

```
i = 0
sum = 0
while True:
    i += 1
    if i == 5:
        continue
    if i > 10:
        break
    sum += i

print(sum)
```

4. range

반복문과 직접적인 연관은 없지만, 흔히 반복문과 연동되어 많이 사용되는 **range**에 대해 간략히 소개한다. **range()** 함수는 보통 아래와 같이 1~3개의 파라미터를 갖는데, 파라미터는 파라미터 갯수에 따라 아래와 같이 다른 의미를 갖는다.

예제	파라미터 의미	리턴값
range(3)	Stop	0, 1, 2
range(3,6)	Start, Stop	3, 4, 5
range(2,11,2)	Start, Stop, Step	2, 4, 6, 8, 10

```
numbers = range(2, 11, 2)

for x in numbers:
    print(x)

# 출력: 각 라인에 2 4 6 8 10 출력
```

특히, for 반복문에서 몇 번 루프를 도는가를 표시하기 위해 range() 함수를 종종 함께 사용한다. 예를 들어, 아래는 Hello 문자열을 10번 (0부터 9까지) 출력하는 예제이다.

```
for i in range(10):
    print("Hello")
```

리스트

1. 리스트 (List)

리스트는 여러 요소들을 갖는 집합(컬렉션)으로 새로운 요소를 추가하거나 갱신, 삭제하는 일이 가능하다. 파이썬의 리스트는 **동적배열(Dynamic Array)**로서 자유롭게 확장할 수 있는 구조를 갖는다. 리스트는 그 안의 요소(element)들은 그 값을 자유롭게 변경할 수 있는 **Mutable** 데이터 타입이다.

리스트의 요소들은 Square bracket([])으로 둘러싸여 컬렉션을 표현하는데, 각 요소들은 서로 다른 타입이 될 수 있으며, 쉼표(,)로 구분한다. 요소가 없는 빈 리스트는 "[]"와 같이 표현한다.

```
a = []      # 빈 리스트
a = ["AB", 10, False]
```

2. 리스트 인덱싱(Indexing)

리스트의 특정 한 요소만을 선택하기 위하여 인덱싱(Indexing)을 사용하는데, 첫번째요소는 "리스트[0]", 두번째 요소는 "리스트[1]" 처럼 표현한다. 즉, 아래 예제에서 처럼 리스트 a 가 있을 때, a[1] 는 두번째 요소 10을 가리킨다. 파이썬 인덱싱에서 한가지 특별한 표현은 인덱스에 -1, -2 같은 음수를 사용할 수 있다는 점이다. 이 때, -1은 현재 리스트의 마지막 요소를, -2는 뒤에서 두번째 요소를 가리킨다.

```
a = ["AB", 10, False]
x = a[1]           # a의 두번째 요소 읽기
a[1] = "Test"     # a의 두번째 요소 변경
y = a[-1]         # False
```

3. 리스트 슬라이싱(Slicing)

리스트에서 일부 부분 요소들을 선택하기 위하여 **슬라이스(Slice)**를 사용한다. 슬라이스는 "리스트[처음인덱스:마지막인덱스]"와 같이 인덱스 표현에서 부분집합의 범위를 지정하는 것이다. 인덱스는 0 부터 시작하며, 마지막 인덱스를 원하는 "마지막 요소의 인덱스 + 1"을 의미한다. 만약 처음 인덱스가 생략되면, 0 부터 시작되며, 마지막 인덱스가 생략되면, 리스트의 끝까지 포함됨을 의미한다.

```
a = [1, 3, 5, 7, 10]
x = a[1:3]      # [3, 5]
x = a[:2]       # [1, 3]
x = a[3:]       # [7, 10]
```

4. 리스트 요소 추가,수정,삭제

리스트에 새로운 요소를 추가하기 위해서는 "리스트.append()"를 사용한다. 리스트 요소를 갱신하기 위해서는 리스트 인덱싱을 사용하여 특정요소에 새 값을 넣는다. 리스트 요소를 삭제하기 위해서는 "del 요소"와 같이 특정 요소를 지운다.

```
a = ["AB", 10, False]
a.append(21.5) # 추가
a[1] = 11     # 변경
del a[2]      # 삭제
print(a)      # ['AB', 11, 21.5]
```

5. 리스트 병합과 반복

두 개의 리스트를 병합하기 위해서는 플러스(+)를 사용한다. 이 때 두 리스트는 순서대로 병합된 새로운 하나의 리스트가 된다. 하나의 리스트를 N 번 반복하기 위해서는 "리스트 * N"와 같이 표현할 수 있다. 이는 동일한 리스트를 계속 반복한 새 리스트를 만들게 된다.

```
# 병합
a = [1, 2]
b = [3, 4, 5]
c = a + b
print(c) # [1, 2, 3, 4, 5]

# 반복
d = a * 3
print(d) # [1, 2, 1, 2, 1, 2]
```

6. 리스트 검색

리스트 안에 특정 요소를 검색하기 위해서 index() 메서드를 사용한다. 또한 특정 요소가 몇 개 있는지 체크하기 위해서 count() 메서드를 사용할 수 있다.

```
mylist = "This is a book That is a pencil".split()
i = mylist.index('book') # i = 3
n = mylist.count('is')   # n = 2
print(i, n)
```

7. List Comprehension

리스트의 [...] 괄호 안에 for 루프를 사용하여 반복적으로 표현식(expression)을 실행해서 리스트 요소들을 정의하는 특별한 용법이 있는데, 이를 List Comprehension 이라 부른다. 이는 아래와 같은 문법으로 컬렉션으로부터 요소를 하나씩 가져와 표현식을 실행하여 그 결과를 리스트에 담는 방식이다. 여기서 if 조건식은 옵션으로 추가될 수 있는데 for 루프에서 이 조건식에 맞는 요소만 표현식을 실행하게 된다.

```
[표현식 for 요소 in 컬렉션 [if 조건식]]
```

아래 예제는 0부터 9까지 숫자들중 3으로 나눈 나머지가 0인 숫자에 대해 그 제곱에 대한 리스트를 구한 예이다.

```
list = [n ** 2 for n in range(10) if n % 3 == 0]

print(list)
# 출력: [0, 9, 36, 81]
```

튜플

1. 튜플 (Tuple)

Tuple은 리스트와 비슷하게 여러 요소들을 갖는 컬렉션이다. 리스트와 다른 점은 Tuple은 새로운 요소를 추가하거나 갱신, 삭제하는 일을 할 수 없다. 즉, Tuple은 한번 결정된 요소를 변경할 수 없는 Immutable 데이터 타입이다. 따라서, Tuple은 컬렉션이 항상 고정된 요소값을 갖기를 원하거나 변경되지 말아야 하는 경우에 사용하게 된다.

튜플의 요소들은 둥근 괄호(...) 를 사용하여 컬렉션을 표현하는데, 각 요소들은 서로 다른 타입이 될 수 있으며, 콤마(,)로 구분한다. 요소가 없는 빈 튜플은 "()"와 같이 표현한다.

```
t = ("AB", 10, False)
print(t)
```

특히 요소가 하나일 경우에는 요소 뒤에 콤마를 붙여 명시적으로 Tuple임을 표시해야 한다. 아래 예제를 보면 첫번째 (123) 의 경우, 이는 산술식의 괄호로 인식하여 t1의 타입이 정수가 된다. 이러한 혼동을 방지하기 위해 t2 에서 처럼 (123,) 콤마를 붙여 명시적으로 Tuple임을 표시한다.

```
t1 = (123)
print(t1) # int 타입

t2 = (123,)
print(t2) # tuple 타입
```

2. Tuple 인덱싱과 슬라이싱

Tuple은 리스트와 마찬가지로 한 요소를 리턴하는 인덱싱과 특정 부분집합을 리턴하는 슬라이싱을 지원한다. 단, 요소값을 변경하거나 추가 혹은 삭제하는 일은 할 수 없다.

```
t = (1, 5, 10)

# 인덱스
second = t[1]      # 5
last = t[-1]       # 10

# 슬라이스
s = t[1:2]          # (5)
s = t[1:]           # (5, 10)
```

3. Tuple 병합과 반복

Tuple은 리스트와 마찬가지로 두 개의 튜플을 병합하기 위해 플러스(+)를 사용하고, 하나의 튜플을 N 번 반복하기 위해서는 "튜플 * N"와 같이 표현한다.

```
# 병합
a = (1, 2)
b = (3, 4, 5)
c = a + b
print(c) # (1, 2, 3, 4, 5)

# 반복
d = a * 3 # 혹은 "d = 3 * a" 도 동일
print(d) # (1, 2, 1, 2, 1, 2)
```

4. Tuple 변수 할당

Tuple 데이터를 변수에 할당할 때, 각 요소를 각각 다른 변수에 할당할 수도 있다. 예를 들어, 아래 예제에서 첫번째 예의 `name` 변수는 튜플 전체를 할당받는 변수이지만, 두번째의 `firstname`, `lastname` 변수는 튜플에 있는 각 요소를 하나씩 할당받는 변수들이다.

```
name = ("John", "Kim")
print(name)
# 출력: ('John', 'Kim')

firstname, lastname = ("John", "Kim")
print(lastname, ",", firstname)
# 출력: Kim, John
```

딕셔너리

1. Dictionary (dict)

Dictionary는 "키(Key) - 값(Value)" 쌍을 요소로 갖는 컬렉션이다. Dictionary는 흔히 Map 이라고도 불리우는데, 키(Key)로 신속하게 값(Value)을 찾아내는 해시테이블(Hash Table) 구조를 갖는다.

파이썬에서 Dictionary는 "dict" 클래스로 구현되어 있다. Dictionary의 키(key)는 그 값을 변경할 수 없는 Immutable 타입이어야 하며, Dictionary 값(value)은 Immutable과 Mutable 모두 가능하다. 예를 들어, Dictionary의 키(key)로 문자열이나 Tuple은 사용될 수 있는 반면, 리스트는 키로 사용될 수 없다.

Dictionary의 요소들은 Curly Brace "{...}" 를 사용하여 컬렉션을 표현하는데, 각 요소들은 "Key:Value" 쌍으로 되어 있으며, 요소간은 콤마로 구분한다. 요소가 없는 빈 Dictionary는 "{}"와 같이 표현한다. 특정 요소를 찾아 읽고 쓰기 위해서는 "Dictionary변수[키]"와 같이 키를 인덱스처럼 사용한다.

```
scores = {"철수": 90, "민수": 85, "영희": 80}
v = scores["민수"] # 특정 요소 읽기
scores["민수"] = 88 # 쓰기
print(t)
```

파이썬의 Dictionary는 생성하기 위해 위의 예제와 같이 {...} 리터럴(Literal)을 사용할 수도 있지만, 또한 dict 클래스의 dict() 생성자를 사용할 수도 있다. dict() 생성자는 (아래 첫번째 예처럼) Key-Value 쌍을 갖는 Tuple 리스트를 받아들이거나 (두번째 예처럼) dict(key=value, key=value, ...) 식의 키-값을 직접 파라미터로 지정하는 방식을 사용할 수 있다.

```
# 1. Tuple List로부터 dict 생성
persons = [('김기수', 30), ('홍대길', 35), ('강찬수', 25)]
mydict = dict(persons)

age = mydict["홍대길"]
print(age)    # 35

# 2. Key=Value 파라미터로부터 dict 생성
scores = dict(a=80, b=90, c=85)
print(scores['b'])    #90
```

2. 추가,수정,삭제,읽기

Dictionary 요소를 수정하기 위해서는 "Dictionary[키]=새값"와 같이 해당 키 인덱스를 사용하여 새 값을 할당하면 된다. Dictionary에 새로운 요소를 추가하기 위해서는 수정 때와 마찬가지로 ("맵[새키]=새값") 새 키에 새 값을 할당한다. Dictionary 요소를 삭제하기 위해서는 "del 요소"와 같이 하여 특정 요소를 지운다.

```
scores = {"철수": 90, "민수": 85, "영희": 80}
scores["민수"] = 88    # 수정
scores["길동"] = 95    # 추가
del scores["영희"]
print(scores)
# 출력 {'철수': 90, '길동': 95, '민수': 88}
```

Dictionary에 있는 값들을 모두 출력하기 위해서는 다음과 같이 루프를 사용할 수 있다. 아래 예제에서 for 루프는 scores 맵으로부터 키를 하나씩 리턴하게 된다. 이때 키는 랜덤하게 리턴되는데, 이는 해시테이블의 속성이다. 각 키에 따른 값을 구하기 위해서는 scores[key]와 같이 사용한다.

```
scores = {"철수": 90, "민수": 85, "영희": 80}

for key in scores:
    val = scores[key]
    print("%s : %d" % (key, val))
```

3. 유용한 dict 메서드

Dictionary와 관련하여 dict 클래스에는 여러 유용한 메서드들이 있다. dict 클래스의 keys()는 Dictionary의 키값들로 된 dict_keys 객체를 리턴하고, values()는 Dictionary의 값들로 된 dict_values 객체를 리턴한다.


```
scores = {"철수": 90, "민수": 85, "영희": 80}

# keys
keys = scores.keys()
for k in keys:
    print(k)

# values
values = scores.values()
for v in values:
    print(v)
```

dict의 items()는 Dictionary의 키-값 쌍 Tuple 들로 구성된 dict_items 객체를 리턴한다. 참고로 dict_items 객체를 리스트로 변환하기 위해서는 list()를 사용할 수 있다. 이는 dict_keys, dict_values 객체에도 공히 적용된다.

```
scores = {"철수": 90, "민수": 85, "영희": 80}

items = scores.items()
print(items)
# 출력: dict_items([('민수', 85), ('영희', 80), ('철수', 90)])

# dict_items를 리스트로 변환할 때
itemsList = list(items)
```

dict.get() 메서드는 특정 키에 대한 값을 리턴하는데, 이는 Dictionary[키]를 사용하는 것과 비슷하다. 단, Dictionary[키]를 사용하면 키가 없을 때 에러(KeyError)를 리턴하는 반면, get()은 키가 Dictionary에 없을 경우 None을 리턴하므로 더 유용할 수 있다. 물론 get()을 사용하는 대신 해당 키가 Dictionary에 존재하는지 체크하고 Dictionary[키]를 사용하는 방법도 있다. 키가 Dictionary에 존재하는지를 체크하지 위해서는 [멤버십연산자 in](#) 을 사용하면 된다.

```
scores = {"철수": 90, "민수": 85, "영희": 80}
v = scores.get("민수") # 85
v = scores.get("길동") # None
v = scores["길동"]     # 에러 발생

# 멤버십연산자 in 사용
if "길동" in scores:
    print(scores["길동"])

scores.clear() # 모두 삭제
print(scores)
```

dict.update() 메서드는 Dictionary 안의 여러 데이터를 한꺼번에 갱신하는데 유용한 메서드이다. 아래 예제에서 처럼, update() 안에 Dictionary 형태로 여러 데이터의 값을 변경하면, 해당 데이터들이 update() 메서드에 의해 한꺼번에 수정된다.

```
persons = [('김기수', 30), ('홍대길', 35), ('강찬수', 25)]
mydict = dict(persons)

mydict.update({'홍대길':33, '강찬수':26})
```

집합

1. Set

Set은 중복이 없는 요소들 (unique elements)로만 구성된 집합 컬렉션이다. Set은 Curly Brace { } 를 사용하여 컬렉션을 표현하는데, 내부적으로 요소들을 순서대로 저장하기 않기 때문에, 순서에 의존하는 기능들을 사용할 수 없다. 만약 set을 정의할 때, 중복된 값을 입력하는 경우, set은 중복된 값을 한번만 가지고 있게 된다. 리스트나 튜플 등을 set으로 변경하기 위해서는 set() 생성자를 사용한다. 이는 리스트에 중복된 값들이 있을 때, 중복 없이 Unique한 값만을 얻고자 할 때 유용하다.

```
# set 정의
myset = { 1, 1, 3, 5, 5 }
print(myset)      # 출력: {1, 3, 5}

# 리스트를 set으로 변환
mylist = ["A", "A", "B", "B", "B"]
s = set(mylist)
print(s)          # 출력: {'A', 'B'}
```

2. Set에서의 추가 및 삭제

Set에 하나의 새로운 요소를 추가하기 위해서는 set 클래스의 add() 메서드를 사용하고, 여러 개의 요소들을 한꺼번에 추가하기 위해서는 update() 메서드를 사용한다. 또한 Set에서 하나의 요소를 삭제하기 위해서는 remove() 혹은 discard() 메서드를 사용하고, 전체를 모두 지우기 위해서는 clear() 메서드를 사용한다.

```

myset = {1, 3, 5}

# 하나만 추가
myset.add(7)
print(myset)

# 여러 개 추가
myset.update({4, 2, 10})
print(myset)

# 하나만 삭제
myset.remove(1)
print(myset)

# 모두 삭제
myset.clear()
print(myset)

```

3. 집합 연산

수학에서 두개의 집합 간의 연산으로 교집합, 합집합, 차집합이 있는데, `set` 클래스는 이러한 집합 연산 기능을 제공한다. 즉, `a`와 `b`가 `set` 일 때, 교집합은 `a & b` (혹은 `a.intersection(b)`), 합집합은 `a | b` (혹은 `a.union(b)`), 차집합은 `a - b` (혹은 `a.difference(b)`) 와 같이 구할 수 있다.

```

a = {1, 3, 5}
b = {1, 2, 5}

# 교집합
i = a & b
# i = a.intersection(b)
print(i)

# 합집합
u = a | b
# u = a.union(b)
print(u)

# 차집합
d = a - b
# d = a.difference(b)
print(d)

```

함수

1. 함수

함수(function)은 일정한 작업을 수행하는 코드블럭으로 보통 반복적으로 계속 사용되는 코드들을 함수로 정의하여 사용하게 된다. 파이썬에서 함수는 `def` 키워드를 사용하여 정의되며, 다음과 같은 문법을 갖는다. 여기서 입력파라미터나 리턴값은 구현하는 내용에 따라 있을 수도 있고, 없을 수도 있다.

```
def 함수명(입력파라미터):
    문장1
    문장2
    [return 리턴값]
```

아래 예제는 두개의 입력파라미터를 받아들여 이를 더한 값을 리턴하는 함수를 정의하고 이를 사용하는 예이다.

```
def sum(a, b):
    s = a + b
    return s

total = sum(4, 7)
print(total)
```

2. 파라미터 전달방식

파이썬 함수에서 입력 파라미터는 **Pass by Assignment**에 의해 전달된다. 즉, 호출자(Caller)는 입력 파라미터 객체에 대해 레퍼런스를 생성하여 레퍼런스 값을 복사하여 전달한다. 또한 전달되는 입력파라미터는 **Mutable**일 수도 있고, **Immutable**일 수도 있으므로 각 경우에 따라 다른 결과가 일어난다.

만약 입력파라미터가 **Mutable** 객체이고, 함수가 그 함수 내에서 해당 객체의 내용을 변경하면, 이러한 변경 사항은 호출자(caller)에게 반영된다. 하지만, 함수 내에서 새로운 객체의 레퍼런스를 입력파라미터에 할당한다면, 레퍼런스 자체는 복사하여 전달되었으므로, 호출자에서는 새로운 레퍼런스에 대해 알지 못하게 되고, 호출자 객체는 아무런 변화가 없게 된다.

만약 입력파라미터가 **Immutable** 객체이면, 입력파라미터의 값이 함수 내에서 변경될 수 없으며, 함수 내에서 새로운 객체의 레퍼런스를 입력파라미터에 할당되어도 함수 외부(Caller)의 값은 변하지 않는다.

```
# 함수내에서 i, mylist 값 변경
def f(i, mylist):
    i = i + 1
    mylist.append(0)

k = 10      # k는 int (immutable)
m = [1,2,3] # m은 리스트 (mutable)

f(k, m)     # 함수 호출
print(k, m) # 호출자 값 체크
# 출력: 10 [1, 2, 3, 0]
```

위의 예제에서 함수 f()는 하나의 정수(i)와 하나의 리스트(mylist)를 입력받아, 함수 내에서 그 값들을 변경한다. 정수는 **Immutable** 타입이므로 함수 내에서 변경된 것이 호출자에 반영되지 않으며, 리스트는 **Mutable** 타입이므로 추가된 요소가 호출자에서도 반영된다.

3. Default Parameter

함수에 전달되는 입력파라미터 중 호출자가 전달하지 않으면 디폴트로 지정된 값을 사용하게 할 수 있는데, 이를 디폴트 파라미터 혹은 **Optional** 파라미터라 부른다. 아래 예제에서 **factor**는 디폴트 파라미터로서 별도로 전달되지 않으면, 그 값이 1로 설정된다.

```
def calc(i, j, factor = 1):
    return i * j * factor

result = calc(10, 20)
print(result)
```

4. Named Parameter

함수를 호출할 때 보통 함수에 정의된 대로 입력파라미터를 순서대로 전달한다. 이러한 순서의 의한 전달 방식 외에 또 다른 호출 방식으로 "파라미터명=파라미터값" 과 같은 형식으로 파라미터를 전달할 수도 있는데, 이를 **Named Parameter**라 부른다. **Named Parameter**를 사용하면 어떤 값이 어떤 파라미터로 전달되는지 쉽게 파악할 수 있는 장점이 있다.

```
def report(name, age, score):
    print(name, score)

report(age=10, name="Kim", score=80)
```

5. 가변길이 파라미터

함수의 입력파라미터의 갯수를 미리 알 수 없거나, 0부터 N개의 파라미터를 받아들이도록 하고 싶다면, 가변길이 파라미터를 사용할 수 있다. 가변길이 파라미터는 파라미터명 앞에 *를 붙여 가변 길이임을 표시한다. 아래 예제에서 *numbers는 가변길이 파라미터이므로, total()을 호출할 때, 임의의 숫자의 파라미터들을 지정할 수 있다.

```
def total(*numbers):
    tot = 0
    for n in numbers:
        tot += n
    return tot

t = total(1,2)
print(t)
t = total(1,5,2,6)
print(t)
```

6. 리턴값

함수로부터 호출자로 리턴하기 위해서는 return 문을 사용한다. return문은 단독으로 쓰이면 아무 값을 호출자에게 전달하지 않으며, "return 리턴값" 처럼 쓰이면, 값을 호출자에게 전달한다. 함수에서 리턴되는 값은 하나 이상일 수 있는데, 필요한 수만큼 return 키워드 다음에 콤마로 구분하여 적는다. 예를 들어, return a,b,c 는 3개의 값을 리턴한다. 하지만, 기술적으로 좀 더 깊이 설명하면, 이는 (a,b,c) 세개의 값을 포함하는 Tuple 하나를 리턴하는 것으로 함수는 항상 하나의 리턴값을 전달한다고 볼 수 있다.

```
def calc(*numbers):
    count = 0
    tot = 0
    for n in numbers:
        count += 1
        tot += n
    return count, tot

count, sum = calc(1,5,2,6) # (count, tot) 튜플을 리턴
print(count, sum)
```

모듈

1. 모듈

모듈(Module)은 파이썬 코드를 논리적으로 묶어서 관리하고 사용할 수 있도록 하는 것으로, 보통 하나의 파이썬 .py 파일이 하나의 모듈이 된다. 모듈 안에는 함수, 클래스, 혹은 변수들이 정의될 수 있으며, 실행 코드를 포함할 수도 있다.

파이썬은 기본적으로 상당히 많은 표준 라이브러리 모듈들을 제공하고 있으며, 3rd Party에서도 많은 파이썬 모듈들을 제공하고 있다. 이러한 모듈들을 사용하기 위해서는 모듈을 import하여 사용하면 되는데, import 문은 다음과 같이 하나 혹은 복수의 모듈을 불러들일 수 있다.

```
import 모듈1[, 모듈2[, ... 모듈N]
```

예를 들어, 아래 예제는 표준 라이브러리 중 수학과 관련된 함수들을 모아 놓은 "math" 모듈을 import 하여 그 모듈 안에 있는 factorial() 함수를 사용하는 예이다.

```
import math
n = math.factorial(5)
```

하나의 모듈 안에는 여러 함수들이 존재할 수 있는데, 이 중 하나의 함수만을 불러 사용하기 위해서는 아래와 같이 "from 모듈명 import 함수명"이라는 표현을 사용할 수 있다. 이렇게 from...import... 방식으로 import 된 함수는 호출시 "모듈명.함수명"이 아니라 직접 "함수명"만을 사용한다.

```
# factorial 함수만 import
from math import factorial

n = factorial(5) / factorial(3)
```

하나의 모듈 안에는 있는 여러 함수를 사용하기 위해 from... import (함수1, 함수2) 와 같이 import 뒤에 사용할 함수를 나열할 수도 있다. 또한, 모든 함수를 불러 사용하기 위해서는 "from 모듈명 import *" 와 같이 asterisk(*)를 사용할 수 있다. 이렇게 from...import... 방식으로 import 된 함수는 호출시 모듈명 없이 직접 함수명을 사용한다.

```
# 여러 함수를 import
from math import (factorial, acos)
n = factorial(3) + acos(1)

# 모든 함수를 import
from math import *
n = sqrt(5) + fabs(-12.5)
```

함수의 이름이 길거나 어떤 필요에 의해 함수의 이름에 Alias를 주고 싶은 경우가 있는데, 이 때는 아래와 같이 "함수명 as Alias" 와 같은 표현을 사용할 수 있다.

```
# factorial() 함수를 f()로 사용 가능
from math import factorial as f
n = f(5) / f(3)
```

2. 모듈의 위치

파이썬에서 모듈을 import 하면 그 모듈을 찾기 위해 다음과 같은 경로를 순서대로 검색한다.

1. 현재 디렉토리
2. 환경변수 PYTHONPATH에 지정된 경로
3. Python이 설치된 경로 및 그 밑의 라이브러리 경로

이러한 경로들은 모두 취합되어 시스템 경로인 `sys.path`에 리스트 형태로 저장된다. 따라서, 모듈이 검색되는 검색 경로는 `sys.path`를 체크하면 쉽게 알 수 있다. 모듈을 import 하면 `sys.path`에 있는 경로 순서대로 모듈을 찾아 import하다가 만약 끝까지 찾지 못하면 에러가 발생된다.

`sys.path`를 사용하기 위해서는 `sys`라는 시스템 모듈을 import 해야 하며, `sys.path`는 임의로 수정할 수도 있다. 예를 들어, 기존 `sys.path`에 새 경로를 추가(`append`)하면, 추가된 경로도 이후 모듈 검색 경로에 포함된다

아래는 `sys.path`를 출력해 본 예인데, `sys.path[0]`의 값은 빈 문자열(empty string)로 이는 현재 디렉토리를 가리킨다. 즉, 먼저 현재 디렉토리부터 찾는다는 뜻이다. 마지막 라인은 `sys.path.append()`를 사용하여 새 경로를 추가하는 예를 든 것이다.

```
>>> import sys
>>> sys.path
['', 'C:\\Python35\\Lib\\idlelib', 'C:\\Python35\\python35.zip', 'C:\\Python35\\DLLs', 'C:\\Python35\\lib', 'C:\\Python35', 'C:\\Python35\\lib\\site-packages']
>>> sys.path[0]
''
>>> sys.path.append('C:\\PySrc')
>>>
```

현재 검색경로를 표시함
첫번째는 빈 문자열로 현재 디렉토리를 가리킴
새 폴더를 검색경로에 추가함

3. 모듈의 작성

프로그램을 모듈로 나누어 코딩하고 관리하는 것은 종종 많은 잇점이 있다. 사용자 함수 혹은 클래스를 묶어 모듈화하고, 이를 불러 사용하는 방법을 간략히 살펴보자. 우선 아래 두 개의 함수(`add`와 `subtract`)를 `mylib.py` 라는 모듈에 저장한다.

```
# mylib.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

모듈 `mylib.py`가 있는 디렉토리에서 그 모듈을 import 한 후, `mylib`의 함수들을 사용한다.

```
# exec.py
from mylib import *

i = add(10,20)
i = subtract(20,5)
```


파이썬 모듈 .py 파일은 import 하여 사용할 수 있을 뿐만 아니라, 해당 모듈 파일 안에 있는 스크립트 전체를 바로 실행할 수도 있다. 파이썬에서 하나의 모듈을 import 하여 사용할 때와 스크립트 전체를 실행할 때를 동시에 지원하기 위하여 흔히 관행적으로 모듈 안에서 `__name__` 을 체크하곤 한다. 파이썬에서 모듈을 import해서 사용할 경우 그 모듈 안의 `__name__` 은 해당 모듈의 이름이 되며, 모듈을 스크립트로 실행할 경우 그 모듈 안의 `__name__` 은 `"__main__"` 이 된다. 예를 들어, `run.py`이라는 모듈을 import 하여 사용할 경우 `__name__` 은 `run.py`가 되며, `"python3.5 run.py"`와 같이 인터프리터로 스크립트를 바로 실행할 때 `__name__` 은 `__main__` 이 된다.

```
# run.py
import sys
def openurl(url):
    #..본문생략..
    print(url)

if __name__ == '__main__':
    openurl(sys.argv[1])
```

위와 같은 `run.py` 모듈을 아래와 같이 스크립트로 실행할 때 `"if __name__ ..."` 조건문은 참이 되어 `openurl(sys.argv[1])` 가 실행된다. 여기서 참고로 `sys.argv`는 Command Line 아규먼트들을 갖는 리스트로서 아래 예제에서 `argv[0]`은 `run.py`, `argv[1]`은 `google.com`이 된다.

```
$ python3.5 run.py google.com
google.com
```

하지만 아래와 같이 모듈을 import하여 사용할 때는 `"if __name__ ..."` 문이 거짓이 되어 `openurl()` 함수가 바로 실행되지 않고, 그 함수 정의만 import 된다. 따라서 이 경우 사용자는 명시적으로 `openurl()` 함수를 호출하여 사용해야 한다.

```
$ python3.5

>>> from run import *

>>> openurl('google.com')
```

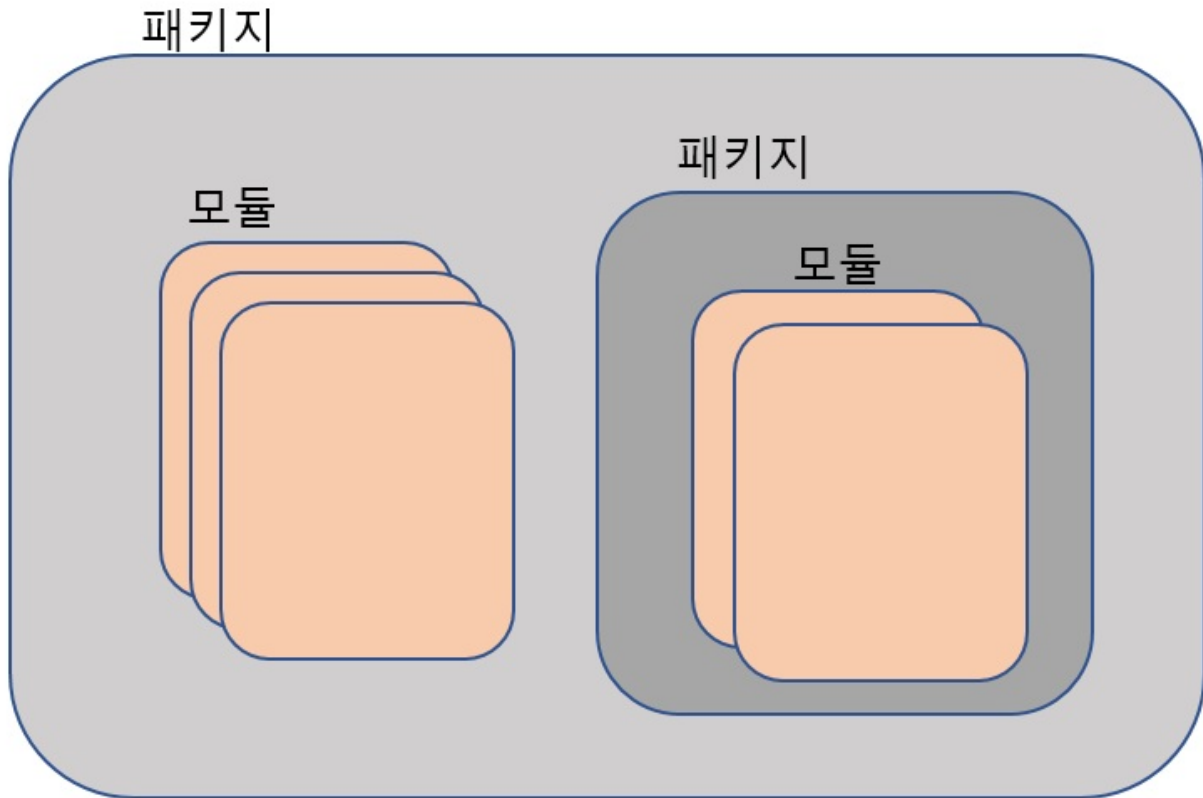
패키지

1. 패키지

파이썬에서 모듈은 하나의 .py 파일을 가리키며, 패키지는 이러한 모듈들을 모은 컬렉션을 가리킨다. 파이썬의 패키지는 하나의 디렉토리에 놓여진 모듈들의 집합을 가리키는데, 그 디렉토리에는 일반적으로 `__init__.py` 라는 패키지 초기화 파일이 존재한다 (주: Python 3.3 이후부터는 `init` 파일

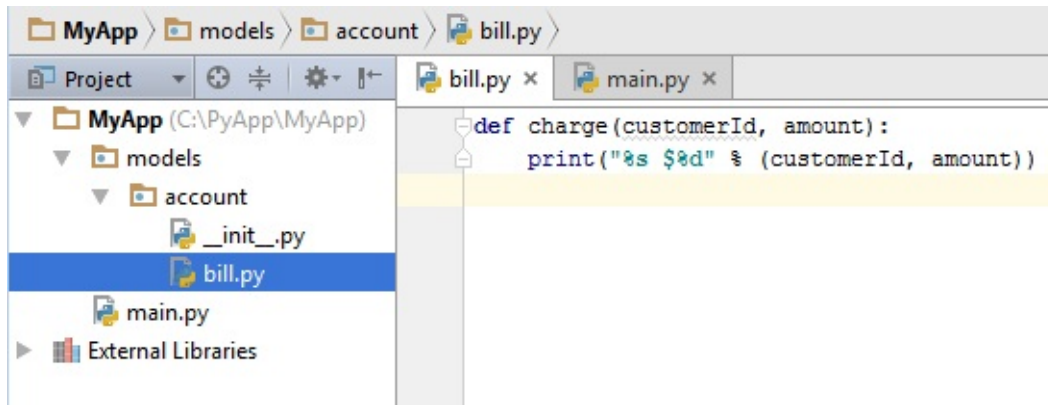
이 없어도 패키지로 인식이 가능하다).

패키지는 모듈들의 컨테이너로서 패키지 안에는 또다른 서브 패키지를 포함할 수도 있다. 파일시스템으로 비유하면 패키지는 일반적으로 디렉토리에 해당하고, 모듈은 디렉토리 안의 파일에 해당한다.



파이썬으로 큰 프로젝트를 수행하게 될 때, 모든 모듈을 한 디렉토리에 모아 두기 보다는 계층적인 카테고리로 묶어서 패키지별로 관리하는 것이 편리하고 효율적이다. 파이썬 프로젝트의 루트로부터 각 영역별로 디렉토리/서브디렉토리를 만들고 그 안에 논리적으로 동일한 기능을 하는 모듈들을 같이 두어 패키지를 만들 수 있다. 이때 패키지는 "디렉토리.서브디렉토리"와 같이 액세스하고 패키지내 모듈은 "디렉토리.서브디렉토리.모듈" 과 같이 액세스할 수 있다. 즉, 각 디렉토리 및 모듈 사이에 점(.)을 사용한다.

간단한 예로 다음 그림과 같이 `models/account` 폴더를 만들고, 그 안에 `bill.py` 라는 모듈이 있다고 가정하자. `models/account` 폴더에는 그 폴더가 일반 폴더가 아닌 패키지임을 표시하기 위해 빈 `__init__.py` 파일을 만들었다 (버전 3.3+ 에선 Optional).



패키지 안에 있는 모듈을 import 하여 사용하기 위해서는 일반 모듈처럼 import문 혹은 from...import... 문을 사용한다. 먼저 import문을 보면 import문은 모듈을 import 하는 것이므로, 패키지내 모듈을 import하기 위해서는 "import 패키지명.모듈명"과 같이 패키지명을 앞에 붙여 사용한다. 아래 예제에서 보면, bill.py 모듈을 import 하기 위해 "import models.account.bill" 와 같이 전체 패키지명을 함께 표시하였음을 볼 수 있다. 또한 모듈내 함수를 사용하기 위하여 models.account.bill.charge()와 같이 패키지명과 모듈명도 함께 써 주어야 한다.

```
# 모듈 import
# import 패키지.모듈
import models.account.bill
models.account.bill.charge(1, 50)
```

다음으로 from ... import ... 문을 살펴보자. 먼저 패키지 모듈을 import 하기 위해 "from 패키지명 import 모듈명" 문을 사용할 수 있다. 아래 예제를 보면, from 뒤에 패키지명 models.account를 사용하였고, import 다음 모듈명 bill을 사용하였다. 이 방식은 해당 모듈 내의 모든 함수를 사용할 수 있는데, bill.charge()와 같이 모듈명.함수명()으로 함수를 호출한다.

만약 패키지 모듈 내의 특정 함수만 import하여 사용하고 싶다면, "from 패키지명.모듈명 import 함수명" 과 같이 from 에 "패키지명.모듈명"을 적고 import 뒤에 함수명을 적는다.

```
# 모듈안의 모든 함수 import
# from 패키지명 import 모듈명
from models.account import bill
bill.charge(1, 50)

# 특정 함수만 import
# from 패키지명.모듈명 import 함수명
from models.account.bill import charge
charge(1, 50)
```

2. __init__.py

패키지에는 `__init__.py` 라는 특별한 파일이 있는데, 이 파일은 기본적으로 그 폴더가 일반 폴더가 아닌 패키지임을 표시하기 위해 사용될 뿐만 아니라, 패키지를 초기화하는 파이썬 코드를 넣을 수 있다. 버전 3.3 이상에서는 이 파일이 없어도 패키지로 사용할 수 있지만, 호환성을 위해 두는 것이 좋다. `__init__.py` 파일에서 중요한 변수로 `__all__` 이라는 리스트 변수가 있는데, 이 변수는 "from 패키지명 import *" 문을 사용할 때, 그 패키지 내에서 import 가능한 모듈들의 리스트를 담고 있다. 즉, `__all__` 에 없는 모듈은 import 되지 않고 에러가 발생한다.

아래 예제는 `__init__.py` 파일 안에 `bill` 모듈을 허락한 후, `from ... import *` 를 사용하여 해당 패키지로부터 허락된 모든 모듈을 import 한 후 `bill.charge()` 함수를 사용하는 예이다.

```
# __init__.py 파일의 내용
```

```
__all__ = ['bill']
```

```
# 패키지내 모든 모듈 import
# from 패키지명 import *
from models.account import *
bill.charge(1, 50)
```

클래스

1. 클래스

파이썬은 객체지향 프로그래밍(OOP, Object Oriented Programming)을 기본적으로 지원하고 있다. 파이썬에서 객체지향 프로그래밍의 기본 단위인 클래스를 만들기 위해서는 아래와 같이 "class 클래스명" 을 사용하여 정의한다. 클래스명은 PEP 8 Coding Convention에 가이드된 대로 각 단어의 첫 문자를 대문자로 하는 CapWords 방식으로 명명한다. 아래 예제는 MyClass라는 클래스를 정의한 것으로 별도의 클래스 멤버를 정의하지 않은 가장 간단한 빈 클래스이다. 클래스 정의 내의 `pass`문은 빈 동작 혹은 아직 구현되지 않았음을 의미하는 것으로 여기서는 빈 클래스를 의미한다.

```
class MyClass:
    pass
```

2. 클래스 멤버

클래스는 메서드(method), 속성(property), 클래스 변수(class variable), 인스턴스 변수(instance variable), 초기자(initializer), 소멸자(destructor) 등의 여러 멤버들을 가질 수 있는데, 크게 나누면 데이터를 표현하는 필드와 행위를 표현하는 메서드로 구분할 수 있다. 파이썬에서 이러한 필드와

메서드는 모두 그 객체의 **attribute** 라고 한다. 다른 OOP 언어와 달리 파이썬은 **Dynamic Language**로서 새로운 **attribute**를 동적으로 추가할 수 있고, 메서드도 일종의 메서드 객체로 취급하여 **attribute**에 포함한다.

메서드

메서드는 클래스의 행위를 표현하는 것으로 클래스 내의 함수로 볼 수 있다. 파이썬에서 메서드는 크게 인스턴스 메서드(**instance method**), 정적 메서드(**static method**), 클래스 메서드(**class method**)가 있다. 가장 흔히 쓰이는 인스턴스 메서드는 인스턴스 변수에 액세스할 수 있도록 메서드의 첫번째 파라미터에 항상 객체 자신을 의미하는 **"self"**라는 파라미터를 갖는다. 아래 예제에서 **calcArea()**가 인스턴스 메서드에 해당된다. 인스턴스 메서드는 여러 파라미터를 가질 수 있지만, 첫번째 파라미터는 항상 **self**를 갖는다.

```
class Rectangle:
    count = 0 # 클래스 변수

    # 초기자(initializer)
    def __init__(self, width, height):
        # self.* : 인스턴스변수
        self.width = width
        self.height = height
        Rectangle.count += 1

    # 메서드
    def calcArea(self):
        area = self.width * self.height
        return area
```

클래스 변수

클래스 정의에서 메서드 밖에 존재하는 변수를 클래스 변수(**class variable**)라 하는데, 이는 해당 클래스를 사용하는 모두에게 공용으로 사용되는 변수이다. 클래스 변수는 클래스 내외부에서 "클래스명.변수명"으로 액세스할 수 있다. 위의 예제에서 **count**는 클래스변수로서 **"Rectangle.count"**와 같이 액세스할 수 있다.

인스턴스 변수

하나의 클래스로부터 여러 객체 인스턴스를 생성해서 사용할 수 있다. 클래스 변수가 하나의 클래스에 하나만 존재하는 반면, 인스턴스 변수는 각 객체 인스턴스마다 별도로 존재한다. 클래스 정의에서 메서드 안에서 사용되면서 **"self.변수명"**처럼 사용되는 변수를 인스턴스 변수(**instance variable**)라 하는데, 이는 각 객체별로 서로 다른 값을 갖는 변수이다. 인스턴스 변수는 클래스 내부에서는 **self.width**과 같이 **"self."**을 사용하여 액세스하고, 클래스 밖에서는 **"객체변수.인스턴스변수"**와 같이 액세스한다.

Python은 다른 언어에서 흔히 사용하는 public, protected, private 등의 접근 제한자 (Access Modifier)를 갖지 않는다. **Python 클래스는 기본적으로 모든 멤버가 public이라고 할 수 있다.** Python 코딩 관례(Convention)상 내부적으로만 사용하는 변수 혹은 메서드는 그 이름 앞에 하나의 밑줄(_)을 붙인다. 하지만 이는 코딩 관례에 따른 것일 뿐 실제 밑줄 하나를 사용한 멤버도 public 이므로 필요하면 외부에서 액세스할 수 있다.

만약 특정 변수명이나 메서드를 **private**으로 만들어야 한다면 두개의 밑줄(__)을 이름 앞에 붙이면 된다.

```
def __init__(self, width, height):
    self.width = width
    self.height = height

    # private 변수 __area
    self.__area = width * height

# private 메서드
def __internalRun(self):
    pass
```

Initializer (초기자)

클래스로부터 새 객체를 생성할 때마다 실행되는 특별한 메서드로 `__init__()` 이라는 메서드가 있는데, 이를 흔히 클래스 **Initializer** 라 부른다 (주: 파이썬에서 두개의 밑줄 (__) 시작하고 두개의 밑줄로 끝나는 레이블은 보통 특별한 의미를 갖는다). **Initializer**는 클래스로부터 객체를 만들 때, 인스턴스 변수를 초기화하거나 객체의 초기상태를 만들기 위한 문장들을 실행하는 곳이다. 위의 `__init__()` 예제를 보면, `width`와 `height`라는 입력 파라미터들을 각각 `self.width`와 `self.height`라는 인스턴스변수에 할당하여 객체 내에서 계속 사용할 수 있도록 준비하고 있다.

(주: Python의 Initializer는 C#/C++/Java 등에서 일컫는 생성자(Constructor)와 약간 다르다.

Python에서 클래스 생성자(Constructor)는 실제 런타임 엔진 내부에서 실행되는데, 이 생성자(Constructor) 실행 도중 클래스 안에 Initializer가 있는지 체크하여 만약 있으면 Initializer를 호출하여 객체의 변수 등을 초기화한다.).

정적 메서드와 클래스 메서드

인스턴스 메서드가 객체의 인스턴스 필드를 `self`를 통해 액세스할 수 있는 반면, **정적 메서드**는 이러한 `self` 파라미터를 갖지 않고 인스턴스 변수에 액세스할 수 없다. 따라서, 정적 메서드는 보통 객체 필드와 독립적이지만 로직상 클래스내에 포함되는 메서드에 사용된다. 정적 메서드는 메서드 앞에 `@staticmethod` 라는 Decorator를 표시하여 해당 메서드가 정적 메서드임을 표시한다.

클래스 메서드는 메서드 앞에 `@classmethod` 라는 Decorator를 표시하여 해당 메서드가 클래스 메서드임을 표시한다. 클래스 메서드는 정적 메서드와 비슷한데, 객체 인스턴스를 의미하는 `self` 대신 `cls` 라는 클래스를 의미하는 파라미터를 전달받는다. 정적 메서드는 이러한 `cls` 파라미터를 전달받지 않는다. 클래스 메서드는 이렇게 전달받은 `cls` 파라미터를 통해 클래스 변수 등을 액세스할 수 있다.

일반적으로 인스턴스 데이터를 액세스 할 필요가 없는 경우 클래스 메서드나 정적 메서드를 사용하는데, 이때 보통 클래스 변수를 액세스할 필요가 있을 때는 클래스 메서드를, 이를 액세스할 필요가 없을 때는 정적 메서드를 사용한다.

아래 예제에서 `isSquare()` 메서드는 정적 메서드로서 `cls` 파라미터를 전달받지 않고 메서드 내에서 클래스 변수를 사용하지 않고 있다. 반면, `printCount()` 메서드는 클래스 메서드로서 `cls` 파라미터를 전달받고 메서드 내에서 클래스 변수 `count` 를 사용하고 있다.

```
class Rectangle:
    count = 0 # 클래스 변수

    def __init__(self, width, height):
        self.width = width
        self.height = height
        Rectangle.count += 1

    # 인스턴스 메서드
    def calcArea(self):
        area = self.width * self.height
        return area

    # 정적 메서드
    @staticmethod
    def isSquare(rectWidth, rectHeight):
        return rectWidth == rectHeight

    # 클래스 메서드
    @classmethod
    def printCount(cls):
        print(cls.count)

# 테스트
square = Rectangle.isSquare(5, 5)
print(square) # True

rect1 = Rectangle(5, 5)
rect2 = Rectangle(2, 5)
rect1.printCount() # 2
```

그 밖의 특별한 메서드들

파이썬에는 Initializer 이외에도 객체가 소멸될 때 (Garbage Collection 될 때) 실행되는 소멸자 (`__del__`) 메서드, 두 개의 객체를 (+ 기호로) 더하는 `__add__` 메서드, 두 개의 객체를 (- 기호로) 빼는 `__sub__` 메서드, 두 개의 객체를 비교하는 `__cmp__` 메서드, 문자열로 객체를 표현할 때 사용하는 `__str__` 메서드 등 많은 특별한 용도의 메서드들이 있다. 아래 예제는 이 중 `__add__()` 메서드에 대한 예이다.

```
def __add__(self, other):
    obj = Rectangle(self.width + other.width, self.height + other.height)
    return obj

# 사용 예
r1 = Rectangle(10, 5)
r2 = Rectangle(20, 15)
r3 = r1 + r2 # __add__()가 호출됨
```

3. 객체의 생성과 사용

클래스를 사용하기 위해서는 먼저 클래스로부터 객체(Object)를 생성해야 한다. 파이썬에서 객체를 생성하기 위해서는 "객체변수명 = 클래스명()"과 같이 클래스명을 함수 호출하는 것처럼 사용하면 된다. 만약 `__init__()` 함수가 있고, 그곳에 입력 파라미터들이 지정되어 있다면, "클래스명(입력 파라미터들)"과 같이 파라미터를 괄호 안에 전달한다. 이렇게 전달된 파라미터들은 `Initializer` 에서 사용된다.

아래 예제를 보면, `Rectangle` 클래스로부터 `r` 이라는 객체를 생성 하고 있는데, `Rectangle(2, 3)`와 같이 2개의 파라미터를 전달하고 있다. 이는 `Rectangle` 초기자에서 각각 `width`와 `height` 인스턴스 변수를 초기화하는데 사용된다.

```
# 객체 생성
r = Rectangle(2, 3)

# 메서드 호출
area = r.calcArea()
print("area = ", area)

# 인스턴스 변수 액세스
r.width = 10
print("width = ", r.width)

# 클래스 변수 액세스
print(Rectangle.count)
print(r.count)
```

클래스로부터 생성된 객체(Object)로부터 클래스 멤버들을 호출하거나 액세스할 수 있다. 인스턴스 메서드는 "객체변수.메서드명()"과 같이 호출할 수 있는데, 위의 예제에선 `r.calcArea()` 이 메서드 호출에 해당된다. 인스턴스 변수는 "객체변수.인스턴스변수" 으로 표현되며, 값을 읽거나 변경하는 일이 가능하다. 위의 예제 `r.width = 10` 은 인스턴스변수 `width` 에 새 값을 할당하는 예이다.

파이썬에서 특히 클래스 변수를 액세스할 때, "클래스명.클래스변수명" 혹은 "객체명.클래스변수명"을 둘 다 허용하기 때문에 약간의 혼란을 초래할 수 있다. 예를 들어, 위의 예제에서 `Rectangle.count` 혹은 `r.count`은 모두 클래스 변수 `count`를 액세스하는 경우로서 이 케이스에는 동일한 값을 출력한다.

하지만, 아래 예제와 같이 `Rectangle` 클래스의 클래스 변수 `count`를 `Rectangle.count`로 할당하지 않고 객체 `r`로부터 할당하면 혼돈스러운 결과를 초래하게 된다.

파이썬에서 한 객체의 `attribute`에 값이 할당되면 (예를 들어, `r.count = 10`), 먼저 해당 객체에 이미 동일한 `attribute`가 있는지 체크해서 있으면 새 값으로 치환하고, 만약 그 `attribute`가 없으면 객체에 새로운 `attribute`를 생성하고 값을 할당한다. 즉, `r.count = 10`의 경우 클래스 변수인 `count`를 사용하는 것이 아니라 새로 그 객체에 추가된 인스턴스 변수를 사용하게 되므로 클래스 변수값은 변경되지 않는다.

파이썬에서 한 객체의 `attribute`를 읽을 경우에는 먼저 그 객체에서 `attribute`를 찾아보고, 없으면 그 객체의 소속 클래스에서 찾고, 다시 없으면 상위 `Base` 클래스에서 찾고, 그래도 없으면 에러를 발생시킨다. 따라서, 위 예제에서 클래스 변수값이 출력된 이유는 값을 할당하지 않고 읽기만 했기 때문에, `r` 객체에 새 인스턴스 변수를 생성하지 않게 되었고, 따라서 객체의 `attribute`가 없어서 클래스의 `attribute`를 찾았기 때문이다.

이러한 혼돈을 피하기 위해 클래스 변수를 액세스할 때는 클래스명을 사용하는 것이 좋다.

```
r = Rectangle(2, 3)

Rectangle.count = 50
r.count = 10    # count 인스턴스 변수가 새로 생성됨

print(r.count, Rectangle.count) # 10 50 출력
```

4. 클래스 상속과 다형성

파이썬은 객체지향 프로그래밍의 상속(Inheritance)을 지원하고 있다. 클래스를 상속 받기 위해서는 파생클래스(자식클래스)에서 클래스명 뒤에 베이스클래스(부모클래스) 이름을 괄호와 함께 넣어 주면 된다. 즉, 아래 예제에서 `Dog` 클래스는 `Animal` 클래스로부터 파생된 파생클래스이며, `Duck` 클래스도 역시 `Animal` 베이스클래스로부터 파생되고 있다. (주: 파이썬은 복수의 부모클래스로부터 상속 받을 수 있는 Multiple Inheritance를 지원하고 있다.)

```
class Animal:
    def __init__(self, name):
        self.name = name
    def move(self):
        print("move")
    def speak(self):
        pass

class Dog (Animal):
    def speak(self):
        print("bark")

class Duck (Animal):
    def speak(self):
        print("quack")
```

파생클래스는 베이스클래스의 멤버들을 호출하거나 사용할 수 있으며, 물론 파생클래스 자신의 멤버들을 사용할 수 있다.

```
dog = Dog("doggy") # 부모클래스의 생성자
n = dog.name # 부모클래스의 인스턴스변수
dog.move() # 부모클래스의 메서드
dog.speak() # 파생클래스의 멤버
```

파이썬은 객체지향 프로그래밍의 다형성(Polymorphism)을 또한 지원하고 있다. 아래 예제는 `animals` 라는 리스트에 `Dog` 객체와 `Duck` 객체를 넣고 이들의 `speak()` 메서드를 호출한 예이다. 코드 실행 결과를 보면 객체의 타입에 따라 서로 다른 `speak()` 메서드가 호출됨을 알 수 있다.

```
animals = [Dog('doggy'), Duck('duck')]

for a in animals:
    a.speak()
```

예외처리

1. 예외처리

프로그램에서 에러가 발생했을 때, 에러를 핸들링하는 기능으로 `try...except` 문을 사용할 수 있다. 즉, `try` 블록 내의 어느 문장에서 에러가 발생하면, `except` 문으로 이동하고 예외 처리를 할 수 있다. `try` 문은 또한 `finally` 문을 가질 수도 있는데, `finally` 블록은 `try` 블록이 정상적으로 실행되든, 에러가 발생하여 `except` 블록이 실행되든 상관없이 항상 마지막에 실행된다.

```
try:
    문장1
    문장2
except:
    예외처리
finally:
    마지막에 항상 수행
```

위의 `except` 문은 `except` 뒤에 아무것도 쓰지 않았는데, 이는 어떤 에러이든 발생하면 해당 `except` 블록을 수행하라는 의미이다. `except` 뒤에 "에러타입"을 적거나 "에러타입 as 에러변수"를 적을 수가 있는데, 이는 특정한 타입의 에러가 발생했을 때만 해당 `except` 블록을 실행하라는 뜻이다. 에러 변수까지 지정했으면, 해당 에러변수를 `except` 블록 안에서 사용할 수 있다. 아래 예제를 보면, `except`가 2개 있는데, 첫번째는 `IndexError`가 발생했을 때만 그 블록을 실행하며, 두번째는 일반적

인 모든 `Exception` 이 발생했을 때 해당 블록을 실행하라는 의미이다. 즉, 먼저 `IndexError` 인지 검사하고, 아니면 다음 `except`를 계속 순차적으로 체크하게 된다. `except`가 여러 개인 경우는 범위가 좁은 특별한 에러타입을 앞에 쓰고 보다 일반적인 에러타입을 뒤에 쓰게 된다.

```
def calc(values):
    sum = None
    # try...except...else
    try:
        sum = values[0] + values[1] + values[2]
    except IndexError as err:
        print('인덱스에러')
    except Exception as err:
        print(str(err))
    else:
        print('에러없음')
    finally:
        print(sum)

# 테스트
calc([1, 2, 3, 6]) # 출력: 에러없음 6
calc([1, 2])      # 출력: 인덱스에러 None
```

또한, 위의 예제에서 `else`문 있는데, 이는 에러가 발생하지 않을 때 실행하게 되는 블록이다. 그리고 `finally` 블록은 항상 마지막에 실행되는 코드 블록이다.

만약 복수 `Exception`들이 동일한 `except` 블록을 갖는다면, 아래와 같이 이들 `Exception`들을 하나의 `except` 문에 묶어서 쓸 수도 있다.

```
def calc(values):
    sum = None
    try:
        sum = values[0] + values[1] + values[2]
    except (IndexError, ValueError):
        print('오류발생')

    print(sum)
```

2. 에러무시와 에러생성

발생된 `Exception`을 그냥 무시하기 위해서는 보통 `pass` 문을 사용하며, 또한 개발자가 에러를 던지기 위해서는 `raise`문을 사용한다.

`raise` 뒤에 아무것도 없는 경우는 현재 `Exception`을 그대로 던지게 된다. 또한 `raise` 뒤에 특정한 에러타입과 에러메시지 (Optional)를 넣어 개발자가 정의한 에러를 발생시킬 수 있다. 예를 들어 아래 예제는 `raise` 뒤에 `Exception` 에러타입과 에러메시지를 넣어 특별한 에러메시지를 전달하고 있다.

```
# pass 를 사용한 예
try:
    check()
except FileExistsError:
    pass

# raise 를 사용한 예
if total
<
0:
    raise Exception('Total Error')
```

3. 파일 에러 처리 예제

아래 예제는 전형적인 파일 에러 처리를 보여주는 코드이다. 파일을 오픈할 때 에러가 발생하면, `except IOError` 블록을 실행한다. 파일오픈을 성공하면, `try` 블록을 실행하고, `finally` 블록에서 파일을 닫게 된다.

```
try:
    fp = open("test.txt", "r")
    try:
        lines = fp.readlines()
        print(lines)
    finally:
        fp.close()
except IOError:
    print('파일에러')
```

참고로 다음은 `with` 문을 써서 해당 블록이 끝나면 자동으로 파일을 닫는 코드의 예이다. Python의 `with` 문은 C#의 `using` 문과 비슷한 것으로 `with` 블록이 끝날 때 자동으로 리소스를 해제하는 역할을 하는데, 특히 주목할 점은 `with` 블록 내에서 어떤 `Exception`이 발생하더라도 반드시 리소스를 해제한다는 점이다.

```
with open('test.txt', 'r') as fp:
    lines = fp.readlines()
    print(lines)
```

Comprehension

1. Python Comprehension

Python의 Comprehension은 한 Sequence가 다른 Sequence (Iterable Object)로부터 (변형되어) 구축될 수 있게한 기능이다. Python 2에서는 List Comprehension (리스트 내포)만을 지원하며, Python 3에서는 Dictionary Comprehension과 Set Comprehension을 추가로 지원하고 있다. 또한, 종종 Generator Comprehension이라고 일컬어 지는 Generator Expression이 있는데, 이는 다음 아티클에서 Generator와 함께 설명한다.

2. List Comprehension

List Comprehension (리스트 내포)는 입력 Sequence로부터 지정된 표현식에 따라 새로운 리스트 컬렉션을 빌드하는 것으로, 아래와 같은 문법을 갖는다.

```
[출력표현식 for 요소 in 입력Sequence [if 조건식]]
```

여기서 입력 Sequence는 입력으로 사용되는 Iteration이 가능한 데이터 Sequence 혹은 컬렉션이다. 입력 Sequence는 for 루프를 돌며 각각의 요소를 하나씩 가져오게 되고, if 조건식이 있으면 해당 요소가 조건에 맞는지 체크하게 된다. 만약 조건에 맞으면 출력 표현식(Output Expression)에 각 요소를 대입하여 출력 결과를 얻게 된다. 이러한 과정을 모든 요소에 대해 실행하여 결과를 리스트로 리턴하게 된다. 쉽게 설명하면 for 루프를 돌면 특정 조건에 있는 입력데이터를 변형하여 리스트로 출력하는 코드를 간단한 문법으로 표현한 것이다.

아래 예제에서 입력 Sequence (oldlist)는 숫자, 문자 그리고 Boolean 요소를 모두 갖는 리스트이다. List Comprehension 문장을 보면 이 입력 Sequence "oldlist"로부터 요소 i 를 하나씩 가져와 i의 타입이 정수형인지 체크하고, 만약 그렇다면 표현식 "i * i" 를 실행하여 i의 제곱을 계산한다. 이렇게 모든 요소에 대해 계산하면 [1, 4, 9] 라는 결과 리스트(newlist)를 얻게 된다.

```
oldlist = [1, 2, 'A', False, 3]

newlist = [i*i for i in oldlist if type(i)==int]

print(newlist)
# 출력: [1, 4, 9]
```

3. Set Comprehension

Set Comprehension은 입력 Sequence로부터 지정된 표현식에 따라 새로운 Set 컬렉션을 빌드하는 것으로, 아래와 같은 문법을 갖는다. List Comprehension과 거의 비슷하지만, 결과가 Set {...}으로 리턴된다는 점이 다르다.

```
{출력표현식 for 요소 in 입력Sequence [if 조건식]}
```

아래 예제에서 입력 Sequence (oldlist)는 중복된 숫자를 갖는 리스트이다. 결과 Set은 중복을 허용하지 않으므로 중복된 데이터는 자연스럽게 제거된다. 또한 Set은 요소의 순서를 보장하지 않으므로, 아래 결과에서 보듯이 순서가 랜덤하게 바뀐 결과를 출력하게 된다.

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

newlist = {i*i for i in oldlist}

print(newlist)
# 출력 : {16, 1, 9, 4}
```

4. Dictionary Comprehension

Dictionary Comprehension은 입력 Sequence로부터 지정된 표현식에 따라 새로운 Dictionary 컬렉션을 빌드하는 것으로, 아래와 같은 문법을 갖는다. Set Comprehension과 거의 비슷하지만, 출력표현식이 Key:Value Pair로 표현된다는 점이 다르며, 결과로 dict 가 리턴된다.

```
{Key:Value for 요소 in 입력Sequence [if 조건식]}
```

아래 예제는 Id로 이름을 찾는 Dictionary (id_name) 를 반대로 Lookup 하기 위해 Key와 Value 서로 바꾼 새로운 Dictionary (name_id) 를 만든 예이다. 새 Dictionary "name_id"는 이름으로 Id를 찾는 Dictionary이다.

```
id_name = {1: '박진수', 2: '강만진', 3: '홍수정'}

name_id = {val:key for key,val in id_name.items()}

print(name_id)

# 출력 : {'박진수': 1, '강만진': 2, '홍수정': 3}
```

또 다른 예제로 아래는 if 조건식 안에 필터링 함수를 사용한 경우이다. 복잡한 조건식일 경우에는 이처럼 필터링 함수를 사용하면 편리하다. 아래 예제는 1부터 100까지 홀수를 Dictionary Key로 하고, 그 홀수의 제곱을 Value로 하는 dict 객체를 생성한다.

```
def isodd(val):
    return val % 2 == 1

mydict = {x:x*x for x in range(101) if isodd(x)}
print(mydict)
```

Iterator와 Generator

1. Iterator

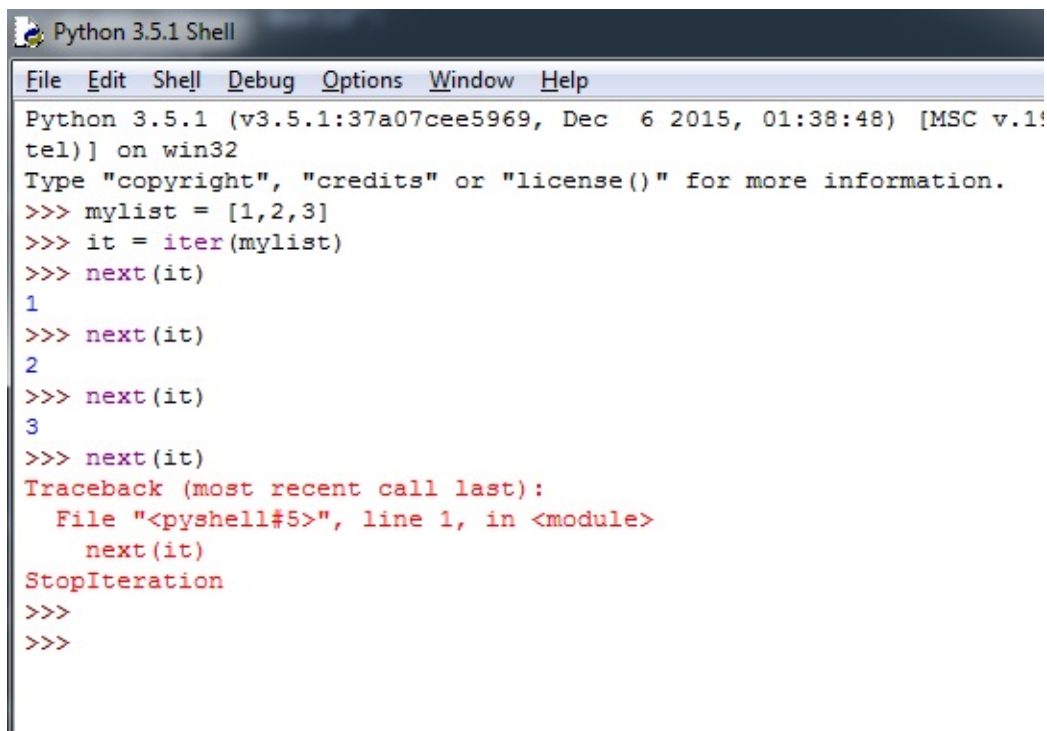
리스트, Set, Dictionary와 같은 컬렉션이나 문자열과 같은 문자 Sequence 등은 for 문을 써서 하나씩 데이터를 처리할 수 있는데, 이렇게 하나 하나 처리할 수 있는 컬렉션이나 Sequence 들을 Iterable 객체(Iterable Object)라 부른다.

```
# 리스트 Iterable
for n in [1,2,3,4,5]:
    print(n)

# 문자열 Iterable
for c in "Hello World":
    print(c)
```

내장 함수 iter()는 "iter(Iterable객체)" 와 같이 사용하여 그 Iterable 객체의 iterator를 리턴한다. Iterable 객체에서 실제 Iteration을 실행하는 것은 iterator로서, iterator는 next 메서드를 사용하여 다음 요소(element)를 가져온다. 만약 더이상 next 요소가 없으면 StopIteration Exception을 발생시킨다.

Iterator의 next 메서드로서 Python 2에서는 "iterator객체.next()" 를 사용하고, Python 3에서는 "iterator객체.__next__()" 메서드를 사용한다. 또한, 버전에 관계없이 사용할 수 있는 방식으로 내장 함수 "next(iterator객체)" 를 사용할 수 있다. 아래는 한 리스트에 대해 list iterator를 구한 후, next() 함수를 계속 호출해 본 예이다.



```
Python 3.5.1 Shell
File Edit Shell Debug Options Window Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1400] on win32
Type "copyright", "credits" or "license()" for more information.
>>> mylist = [1,2,3]
>>> it = iter(mylist)
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    next(it)
StopIteration
>>>
>>>
```

어떤 클래스를 Iterable 하게 하려면, 그 클래스의 iterator를 리턴하는 __iter__() 메서드를 작성해야 한다. 이 __iter__() 메서드가 리턴하는 iterator는 동일한 클래스 객체가 될 수도 있고, 별도로 작성된 iterator 클래스의 객체가 될 수도 있다. 어떠한 경우든 Iterator가 되는 클래스는

`__next()` 메서드 (Python 2 인 경우 `next()` 메서드) 를 구현해야 한다. 실제 for 루프에 `Iterable Object`를 사용하면, 해당 `Iterable`의 `__iter__()` 메서드를 호출하여 `iterator`를 가져온 후 그 `iterator`의 `next()` 메서드를 호출하여 루프를 돌게 된다.

아래 예제는 간단한 `Iterator`를 예시한 것으로 `__iter__()` 메서드에서 `self` 를 리턴함으로써 `Iterable`과 동일한 클래스에 `Iterator`를 구현했음을 표시하였고, 그 클래스 안에 `Iterator`로서 필요한 `__next__()` 메서드 (Python 3)를 구현하였다.

```
class MyCollection:
    def __init__(self):
        self.size = 10
        self.data = list(range(self.size))

    def __iter__(self):
        self.index = 0
        return self

    def __next__(self):
        if self.index >= self.size:
            raise StopIteration

        n = self.data[self.index]
        self.index += 1
        return n

coll = MyCollection()
for x in coll:
    print(x)
```

2. Generator

Generator는 `Iterator`의 특수한 한 형태이다.

Generator 함수(Generator function)는 함수 안에 `yield` 를 사용하여 데이터를 하나씩 리턴하는 함수이다. Generator 함수가 처음 호출되면, 그 함수 실행 중 처음으로 만나는 `yield` 에서 값을 리턴한다. Generator 함수가 다시 호출되면, 직전에 실행되었던 `yield` 문 다음부터 다음 `yield` 문을 만날 때까지 문장들을 실행하게 된다. 이러한 Generator 함수를 변수에 할당하면 그 변수는 `generator` 클래스 객체가 된다.

아래 예제는 간단한 Generator 함수와 그 호출 사례를 보인 것이다. 여기서 `gen()` 함수는 Generator 함수로서 3개의 `yield` 문을 가지고 있다. 따라서 한번 호출시마다 각 `yield` 문에서 실행을 중지하고 값을 리턴하게 된다.


```
# Generator 함수
def gen():
    yield 1
    yield 2
    yield 3

# Generator 객체
g = gen()
print(type(g)) #
<class 'generator'>

# next() 함수 사용
n = next(g); print(n) # 1
n = next(g); print(n) # 2
n = next(g); print(n) # 3

# for 루프 사용 가능
for x in gen():
    print(x)
```

위의 예에서 `g = gen()` 문은 Generator 함수를 변수 `g`에 할당한 것인데, 이때 `g`는 `generator`라는 클래스의 객체로서 `next()` 내장함수를 사용하여 Generator의 다음 값을 계속 가져올 수 있다. Generator는 물론 예제의 마지막 부분과 같이 `for` 루프에서 사용될 수 있다.

리스트나 **Set**과 같은 컬렉션에 대한 **iterator**는 해당 컬렉션이 이미 모든 값을 가지고 있는 경우이나, **Generator**는 모든 데이터를 갖지 않은 상태에서 **yield**에 의해 하나씩만 데이터를 만들어 가져온다는 차이점이 있다. 이러한 Generator는 데이터가 무제한이어서 모든 데이터를 리턴할 수 없는 경우나, 데이터가 대량이어서 일부씩 처리하는 것이 필요한 경우, 혹은 모든 데이터를 미리 계산하면 속도가 느려서 그때 그때 On Demand로 처리하는 것이 좋은 경우 등에 종종 사용된다.

3. Generator Expression

Generator Expression은 Generator Comprehension으로도 불리우는데, List Comprehension과 외관상 유사하다. List Comprehension은 앞뒤를 [...] 처럼 대괄호로 표현한다면, Generator Expression (...) 처럼 둥근 괄호를 사용한다. 하지만 Generator Expression은 List Comprehension과 달리 실제 리스트 컬렉션 데이터 전체를 리턴하지 않고, 그 표현식만을 갖는 Generator 객체만 리턴한다. 즉 실제 실행은 하지 않고, 표현식만 가지며 위의 `yield` 방식으로 Lazy Operation을 수행하는 것이다.

아래 예제는 1부터 1000개까지의 숫자에 대한 제곱값을 Generator Expression으로 표현한 것으로 여기서 Generator Expression을 할당받은 변수 `g`는 Generator 타입 객체이다. 첫번째 `for` 루프를 사용하여 10개의 `next()` 문을 실행하여 처음 10개에 대한 제곱값만을 실행하였다. 두번째 `for` 루프에서는 11번째부터 마지막까지 모두 실행하게 된다. Generator 객체 `g`는 상태를 유지하고 있으므로 두번째 `for` 루프에서 다음 숫자 11부터 계산을 수행한 것이다.

```
# Generator Expression
g = (n*n for n in range(1001))

# g는 generator 객체
print(type(g)) #
<class 'generator'>

# 리스트로 일괄 변환시
# mylist = list(g)

# 10개 출력
for i in range(10):
    value = next(g)
    print(value)

# 나머지 모두 출력
for x in g:
    print(x)
```

메일 보내기 (SMTP Mail)

파이썬에서 이메일을 보내기 위해서는 파이썬에 기본 내장된 `smtplib` 라는 모듈을 사용한다.

SMTP는 Simple Mail Transfer Protocol의 약자로서 메일을 보내는데 사용되는 프로토콜이다. 개인이나 회사가 SMTP 서버를 설치해서 이를 통해 메일을 발송할 수 있지만, 요즘은 구글, 마이크로소프트 등 많은 회사들이 SMTP 서버를 사용할 수 있도록 오픈하고 있기 때문에 이들 SMTP 서버를 사용해서 메일을 발송할 수 있다.

메일 Provider	SMTP 서버명, 포트
Live	smtp.live.com, 587
GMail	smtp.gmail.com, 587

파이썬에서 SMTP 서버에 접속하기 위해서는 `smtplib` 모듈을 `import` 한 후, SMTP 서버와 포트로 SMTP 객체를 생성한다. SMTP 서버의 Encryption 방식에 따라 TTL 혹은 SSL을 사용하는데, TLS를 사용하는 경우 `smtplib.SMTP()`를, SSL을 사용하는 경우 `smtplib.SMTP_SSL()` 을 사용한다. TLS는 보통 포트 587을, SSL은 465를 사용한다.

SMTP 객체를 생성한 후에는 프로토콜 상 가장 먼저 SMTP 서버에 Hello 메시지를 보내는데, `ehlo()` 메서드가 이 기능을 한다. Hello 메시지 이후, TLS 인 경우는 `starttls()` 를 실행하여 TLS Encryption을 시작하는데, SSL인 경우에는 이 메서드를 호출하지 않는다.

```
# -*- coding:utf-8 -*-

import smtplib
from email.mime.text import MIMEText

smtp = smtplib.SMTP('smtp.live.com', 587)
smtp.ehlo()      # say Hello
smtp.starttls()  # TLS 사용시 필요
smtp.login('lee@live.com', 'password')

msg = MIMEText('본문 테스트 메시지')
msg['Subject'] = '테스트'
msg['To'] = 'kim@naver.com'
smtp.sendmail('lee@live.com', 'kim@naver.com', msg.as_string())

smtp.quit()
```

이렇게 기본 연결을 마친 후, `login`(계정, 암호) 메서드를 호출하여 계정과 암호를 넣고 사용자 인증을 받게 된다. 이후, `sendmail`(송신자, 수신자, 메시지) 메서드를 사용하여 메일을 보내게 되는데, 여기서 메시지는 간단한 메시지의 경우는 `email.mime.text.MIMEText` 을, 좀 더 복잡한 메시지의 경우는 `email.mime.multipart.MIMEMultipart` 등을 사용할 수 있다. 마지막으로 SMTP와 연결을 끊고 종료하기 위해서는 `quit()` 메서드를 호출한다.

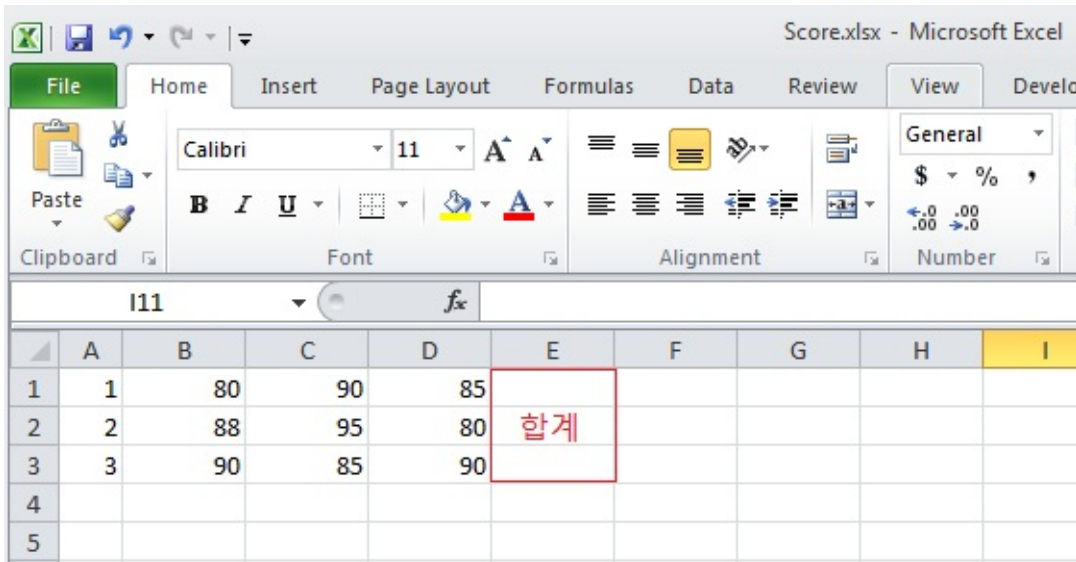
1. 파이썬에서 엑셀 사용하기

파이썬에서 엑셀 데이터를 핸들링하기 위해서는 `openpyxl`, `xlrd`, `xlrw` 등의 외부 패키지를 설치해서 사용한다. 여기서는 `openpyxl` 패키지를 아래와 같이 설치하여 사용한다.

```
pip install openpyxl
```

2. 엑셀 데이터 읽고 쓰기

엑셀을 다루는 가장 기초적인 부분은 데이터를 읽고 쓰는 동작이므로, 먼저 엑셀 파일을 열고 그 안의 데이터를 읽고 몇 개의 데이터를 쓰는 동작을 살펴 보자. 아래 예제는 국영수 점수를 담은 엑셀 파일을 읽어 각 학생별 국영수 합계를 구하여 다시 저장하는 예이다. 엑셀 입력 파일은 아래와 같다고 가정한다. 합계 부분은 추가해야 하는 부분이다.



The screenshot shows the Microsoft Excel interface with the file 'Score.xlsx'. The 'Home' tab is active, showing the ribbon with Font, Alignment, and Number groups. The spreadsheet has columns A through I and rows 1 through 5. The data is as follows:

	A	B	C	D	E	F	G	H	I
1	1	80	90	85					
2	2	88	95	80	합계				
3	3	90	85	90					
4									
5									

The formula bar at the top shows the active cell is E11, containing the text '합계' (Total).

```

import openpyxl

# 엑셀파일 열기
wb = openpyxl.load_workbook('score.xlsx')

# 현재 Active Sheet 얻기
ws = wb.active
# ws = wb.get_sheet_by_name("Sheet1")

# 국영수 점수를 읽기
for r in ws.rows:
    row_index = r[0].row    # 행 인덱스
    kor = r[1].value
    eng = r[2].value
    math = r[3].value
    sum = kor + eng + math

    # 합계 쓰기
    ws.cell(row=row_index, column=5).value = sum

    print(kor, eng, math, sum)

# 엑셀 파일 저장
wb.save("score2.xlsx")
wb.close()

```

먼저 엑셀 파일을 오픈하기 위해 `openpyxl.load_workbook(엑셀파일명)` 함수를 호출하여 `Workbook` 객체를 얻는다. 하나의 `Workbook`에는 여러 개의 `Worksheet` 들이 있는데 통상 엑셀은 기본으로 `Sheet1`, `Sheet2`, `Sheet3` 등 3개의 시트를 생성한다. 엑셀 파일을 열면 보통 첫번째 시트가 `Active Sheet`가 되므로 `Worksheet` 객체의 `active` 를 통해 현재 워크시트를 가져올 수 있지만, 엑셀은 이전 저장시 마지막에 선택된 시트를 `Active Sheet`로 하므로 `wb.get_sheet_by_name(시트명)` 을 사용하는 것이 더 안전하다.

워크시트는 행(`Row`)과 열(`Column`)로 구성되어 있는데, 시트 내에 데이터가 있는 부분의 행들은 시트객체.`rows` 를 통해 액세스할 수 있고, 마찬가지로 시트객체.`columns`는 유효 컬럼들을 액세스 하는데 사용한다. 위의 예제에서 각 행을 하나씩 가져오기 위해 `for` 루프로 `ws.rows`로부터 한 `row` 씩 가져오고 있다. 각 `row`는 그 행 안에 있는 `cell` 들의 집합으로 처음 `cell`은 `r[0]`과 같이 인덱스 0을 사용한다. 첫 `cell` 즉 `r[0]`의 값을 리턴하기 위해 `r[0].value` 을 사용한다.

특정 `cell` 에 값을 지정하기 위해 `cell.value` 에 값을 넣으면 되는데, 시트에서 `cell`을 지정하기 위해 `ws["A1"]`과 같이 엑셀식 `cell` 지정법을 사용할 수 있고, 또한 행열 인덱스를 사용하여 `ws.cell(row=행인덱스, column=열인덱스)` 표현을 사용할 수도 있다.

엑셀의 변경 내용을 저장하기 위해서는 `Workbook` 객체에서 `save()` 메서드를 사용하며, 엑셀 사용이 모두 끝난 경우 `close()` 메서드를 호출한다.

웹 크롤링(스크래핑) 이해하기

웹 크롤링(스크래핑)의 개념

- **웹 스크래핑(web scraping)**: 웹 사이트 상에서 원하는 부분에 위치한 정보를 컴퓨터로 하여금 자동으로 추출하여 수집하도록 하는 기술
- **웹 크롤링(web crawling)**: 자동화 봇(bot)인 웹 크롤러(web crawler)가, 정해진 규칙에 따라 복수 개의 웹 페이지를 브라우징하는 행위 (aka 웹 스파이더링(web spidering))

즉, 일반적으로 '웹 크롤링'이라고 알려진 과정은, 우리가 제작한 웹 크롤러가 웹 크롤링 과정에서 방문한 페이지들의 콘텐츠를 저장한 뒤, 이에 웹 스크래핑 기술을 적용하여 원하는 부분의 정보만을 추출하여 데이터셋의 형태로 저장하는 과정을 모두 포괄합니다.

우리가 취급하는 모든 웹 페이지는 기본적으로 HTML과 CSS 기술에 근거하여 만들어져 있으며, 근래에 접어들면서 JavaScript 기술의 적용 부분도 확장되고 있습니다. 따라서 웹 크롤링 및 스크래핑을 무리 없이 수행하기 위해서는 HTML, CSS, JavaScript 등의 기술에 대한 기초적인 이해가 필요합니다.

만약 HTML과 CSS의 기초를 학습하지 않으신 분의 경우, Flearning의 프론트엔드 강좌를 먼저 수강해 주시길 바랍니다.

<https://www.flearning.net/courses/1>

웹 크롤링 및 스크래핑을 위한 라이브러리

- **scrapy(스크래파이)**: 웹 데이터 분석을 위해, 웹 사이트를 크롤링하고 스크래핑을 통해 정보를 추출하여 데이터셋의 형태로 저장하는 데 특화된 Python 라이브러리

scrapy의 기초적인 기능만을 사용할 경우, 우리가 '웹 브라우저로 보고 있는' 화면을 그대로 스크래핑할 수 없는 상황이 발생합니다. 대표적인 두 가지 유형이 있습니다.

1. 동적으로 콘텐츠를 생성하는 '동적 웹 페이지(dynamic web page)'

동적 웹 페이지의 경우, 일단 HTML/CSS로만 작성된 정적(static) 웹 페이지를 띄운 뒤, 브라우저로 하여금 서버 측으로 추가적인 요청을 보내도록 한 뒤 이에 대한 응답으로 필요한 콘텐츠를 웹 페이지 상에 사후적으로 채워넣는 방식입니다.

- 예시: English Premier League 2015/16 시즌 최종 순위표

<https://www.premierleague.com/tables?co=1&se=42&mw=-1&ha=-1>

2. 웹 페이지에 대한 요청 시 로그인 정보 등을 함께 보내야 하는 웹 페이지

어느 웹 사이트 상에서 회원의 신분으로 로그인하여 회원에게만 제공되는 웹 페이지를 보고 싶을 경우, 웹 페이지를 요청할 때마다 매번 '내가 현재 회원 신분으로 로그인되어 있다'는 것을 입증할 수 있는 증거 자료(쿠키(cookie))를 서버 측으로 함께 제시해야 합니다.

- 예시: Flearning에서 제공하는 강좌 소개 페이지 -> 첫 번째 강의 시청하기

<https://www.flearning.net/classes/15>

위와 같은 유형의 웹 사이트에 대한 스크래핑을 제대로 수행하기 위해서는, HTML, CSS, JavaScript 및 HTTP 통신 등과 관련된 어느 정도 전문적인 이해가 필요합니다. 이런 이해가 갖춰져 있다면, scrapy의 고급 기능을 활용하여 위 유형의 웹 사이트에 대해 원하는 웹 페이지를 크롤링할 수 있습니다.

본 강의에서는 Python 자동화 브라우저 라이브러리인 '**selenium(셀레늄)**'의 webdriver 모듈을 추가로 사용하여, 웹 기술에 대한 이해가 없더라도 위 유형에 해당하는 웹 페이지들을 '웹 브라우저로 보고 있는' 모습 그대로 스크래핑할 수 있도록 하는 가장 간단한 꼼수를 안내할 것입니다.

scrapy 및 selenium 설치하기

scrapy 및 selenium 설치하기 (Windows)

scrapy 설치하기

현재 scrapy는 Windows의 Python 3에 대해서는 지원을 하고 있지 않으므로, conda를 사용하여 Python 2 가상 환경을 따로 구축해야 합니다. 크롤링을 하는 경우에 한해서만 Python 2 가상 환경을 사용하고, 데이터 분석을 하는 경우에는 원래의 Python 3 환경을 사용할 것입니다.

1. Python 2 가상 환경 생성 및 activate/deactivate

- 가상의 Python 2 환경 구성:

```
> conda create -n py27 python=2.7
```

- 가상의 Python 2 환경을 activate:

```
> activate py27
```

- 현재 Python의 버전을 확인하여, 어떤 버전의 Python이 activate 되어 있는지 확인:

```
> python --version
```

- 가상의 Python 2 환경을 deactivate:

```
> deactivate py27
```

이제 Python 2 가상 환경을 activate한 상태에서, 이어지는 과정을 수행합니다.

2. lxml 및 pywin32 설치

- lxml의 Windows용 wheel 파일 다운로드:

<http://www.lfd.uci.edu/~gohlke/pythonlibs/#lxml>

lxml-3.6.1-cp27-cp27m-win32.whl (32비트 운영체제의 경우) *lxml-3.6.1-cp27-cp27m-win_amd64.whl* (64비트 운영체제의 경우)

- 다운로드받은 wheel 파일을 사용하여 lxml 설치(64비트 운영체제 예시):

```
> pip install lxml-3.6.1-cp27-cp27m-win_amd64.whl
```

- pywin32 설치:

```
> pip install pypiwin32
```

3. Microsoft Visual C++ Compiler for Python 2.7 설치

(주의) 사용하시는 Windows 버전 혹은 이전의 업데이트 내역에 따라, **Microsoft Visual C++ Compiler**가 이미 설치되어 있는 경우도 있습니다. 따라서 2번 과정까지 완료하신 뒤 4번 과정 (**scrapy** 설치)을 먼저 시도해 주시고, 그 과정에서 오류가 발생하는 경우에 한해서만 본 3번 과정을 진행해주시길 바랍니다.

- Microsoft Visual C++ Compiler for Python 2.7 다운로드 및 설치:

<http://www.microsoft.com/en-us/download/details.aspx?id=44266>

4. scrapy 설치

```
> pip install scrapy
```

selenium 설치하기

```
> pip install selenium
```

ChromeDriver 다운로드 및 설치하기

ChromeDriver는, selenium과 같은 자동화 브라우저 라이브러리에서 크롬 브라우저를 사용하기 위해 필요로 하는 드라이버입니다.

- chromedriver_win32.zip 파일 다운로드:

<https://sites.google.com/a/chromium.org/chromedriver/downloads>

다운로드받은 파일의 압축을 해제한 후, 이를 홈 디렉터리로 이동하고 그 경로를 기록합니다.

예시: *C:\Users\kilho\chromedriver.exe*

위 경로를, selenium을 사용해서 크롬을 실행할 때마다 다음과 같이 입력할 것입니다:

```
browser = webdriver.Chrome("C:\Users\kilho\chromedriver.exe")
```

scrapy 테스트

cmd에서 다음을 실행합니다:

```
> scrapy shell "https://www.flearning.net/classes/15"
```

IPython이 실행되면, 다음을 실행합니다:

```
response.text
```

HTML 태그가 길게 표시된 경우, 해당 URL의 페이지를 제대로 스크래핑한 것으로 해석할 수 있습니다. 다음을 실행하여 종료합니다:

```
exit()
```

selenium 테스트

IPython을 실행한 뒤, 다음을 순서대로 실행합니다:

```
from selenium import webdriver
browser = webdriver.Chrome("C:\Users\kilho\chromedriver.exe")
browser.get("http://flearning.net")
browser.quit()
exit()
```

scrapy 및 selenium 설치하기

scrapy 및 selenium 설치하기 (Mac OS)

scrapy 설치하기

```
> pip install scrapy
```

selenium 설치하기

```
> pip install selenium
```

ChromeDriver 다운로드 및 설치하기

ChromeDriver는, selenium과 같은 자동화 브라우저 라이브러리에서 크롬 브라우저를 사용하기 위해 필요로 하는 드라이버입니다.

- chromedriver_mac64.zip 파일 다운로드:

<https://sites.google.com/a/chromium.org/chromedriver/downloads>

다운로드받은 파일의 압축을 해제한 후, 이를 홈 디렉터리로 이동하고 그 경로를 기록합니다.

예시: `/Users/kilho/chromedriver`

위 경로를, selenium을 사용해서 크롬을 실행할 때마다 다음과 같이 입력할 것입니다:

```
browser = webdriver.Chrome("/Users/kilho/chromedriver")
```

scrapy 테스트

터미널에서 다음을 실행합니다:

```
> scrapy shell "https://www.flearning.net/classes/15"
```

IPython이 실행되면, 다음을 실행합니다:

```
response.text
```

HTML 태그가 길게 표시된 경우, 해당 URL의 페이지를 제대로 스크래핑한 것으로 해석할 수 있습니다. 다음을 실행하여 종료합니다:

```
exit()
```

selenium 테스트

IPython을 실행한 뒤, 다음을 순서대로 실행합니다:

```
from selenium import webdriver
browser = webdriver.Chrome("C:\Users\kilho\chromedriver.exe")
browser.get("http://flearning.net")
browser.quit()
exit()
```

scrapy의 구조 및 웹 스크래핑 맛보기

scrapy의 기본 구조

Spider

Spider는 어떤 웹 사이트들을 어떠한 규칙에 의거하여 크롤링할 것인지 명시하고, 각각의 웹 페이지의 어떤 부분을 스크래핑할 것인지 등을 일괄적으로 명시하는 클래스입니다.

Selector

Spider 상에 웹 페이지 상의 어느 부분을 스크래핑할 것인지 명시하고자 할 때, 특정 HTML 요소를 간편하게 선택할 수 있도록 하는 메커니즘을 scrapy에서는 **Selector** 클래스로 구현하였습니다.

Selector를 사용하면 CSS 선택자를 직접 사용하여 특정 HTML 요소를 선택할 수도 있으나, HTML 상에서의 특정 요소를 선택하는 데 특화된 언어인 **'XPath'**를 사용하는 것이 더 권장됩니다.

HTML과 CSS의 기초를 잘 알고 있다면, XPath는 몇 개의 예제를 따라서 작성하는 것만으로 금방 학습할 수 있습니다. 만약 XPath에 대해 좀 더 자세하게 배우고 싶다면, 다음의 링크를 참조하시길 바랍니다.

http://www.w3schools.com/xsl/xpath_intro.asp

Item과 Item pipeline

Item은 scrapy에서 기본 제공하는 자료구조 클래스입니다. 새로운 Item 클래스를 정의하고 여기에 우리가 수집하고자 하는 정보들을 명시하면, Spider 상에서 실제 스크래핑을 수행한 결과물을 간편하게 관리할 수 있습니다.

Item pipeline 클래스를 새로 정의하고 여기에 각 Item들을 어떻게 처리할 것인지 명시하면, 해당 규칙에 의거하여 데이터를 가공하거나 혹은 외부 파일로 간편하게 저장할 수 있습니다.

Settings

Spider나 Item pipeline 등이 어떻게 동작하도록 할 지에 대한 세부적인 설정 사항을 **Settings** 상에 명시합니다.

거의 모든 웹 사이트에서는 크롤링을 수행하는 크롤러 봇들의 행동을 제한하고자 robots.txt라는 파일을 게시합니다.

<https://www.flearning.net/robots.txt>

예를 들어, 우리가 제작한 Spider가 이 robots.txt 파일에 명시된 규칙을 따를 것인지, 혹은 무시할 것인지 등을 Settings 상에서 설정할 수도 있습니다.

scrapy shell과 크롬 개발자 도구를 사용하여 웹 스크래핑 체험하기

scrapy shell

scrapy에서는 지정한 웹 페이지를 스크래핑한 결과를 인터랙티브하게 확인할 수 있도록 하는 기능인 **scrapy shell**을 제공합니다. 예를 들어 아래와 같은 URL의 웹 페이지를 스크래핑하고자 한다고 합시다.

<http://www.dmoz.org/Computers/Programming/Languages/Python/Books>

cmd 혹은 터미널 상에서 scrapy shell을 실행할 때, 다음과 같이 URL을 명시해줍니다.

```
> scrapy shell "http://www.dmoz.org/Computers/Programming/Languages/Python/Books"
```

그러면 IPython이 실행되면서, 해당 웹 페이지에 대해 스크래핑을 할 준비가 완료됩니다.

크롬 개발자 도구

크롬 개발자 도구(Developer Tools)는, 웹 페이지 상에서 원하는 요소의 XPath를 한 번에 찾아서 제공하는 기능을 지원합니다. 크롬 개발자 도구의 사용법과 관련해서는, 동영상의 내용을 참조해주시길 바랍니다.

웹 스크래핑 체험하기

동영상 강의의 scrapy shell에서 실행한 코드를 순서대로 아래와 같이 나타내었습니다.

- 'Sites' 리스트 상의 하나의 아이템의 제목 텍스트 스크래핑하기:

```
response.xpath('//*[id="site-list-content"]/div[1]/div[3]/a/div')
response.xpath('//*[id="site-list-content"]/div[1]/div[3]/a/div/text()')
response.xpath('//*[id="site-list-content"]/div[1]/div[3]/a/div/text()').extract()
response.xpath('//*[id="site-list-content"]/div[1]/div[3]/a/div/text()').extract()[0]
```

- 'Sites' 리스트 상의 모든 아이템의 제목 텍스트 스크래핑하기:

```
titles = response.xpath('//*[id="site-list-content"]/div')
title = titles[0]
title.xpath('./div[3]/a//div/text()')
for title in titles:
    print(title.xpath('./div[3]/a//div/text()'))
```

본 강의에서 사용한 XPath 설명

- // : 기준이 되는 요소의 자손 요소들을 모두 살펴보고, 이 중에서 이어지는 조건을 만족하는 요소를 찾음
- / : 기준이 되는 요소의 직속 자식 요소들만을 살펴보고, 이 중에서 다음에 이어지는 조건을 만족하는 요소를 찾음
- * : 모든 종류의 태그에 해당하는 요소를 탐색
- [id="site-list-content"] : 탐색하고자 하는 요소의 id 값에 대한 조건을 명시
- [1] : 명시한 조건을 만족하는 요소가 여러 개인 경우, 이들 중 몇 번째 요소를 가져올 것인지에 대한 인덱스(1부터 시작)
- /text() : 해당 요소의 텍스트 부분만을 추출

정적 웹 페이지에서 데이터 추출하고 수집하기

영화 데이터 수집하기: rottentomatoes.com

- Rotten Tomatoes 2015년 TOP 100 영화 리스트 페이지:

<https://www.rottentomatoes.com/top/bestofrt/?year=2015>

해당 웹 페이지에서, 영화 제목 하나를 클릭하면 그 영화의 상세 정보를 나타내는 웹 페이지가 표시 됩니다. 영화 제목이 여러 개 있으므로 이런 페이지들도 여러 개 얻어질 것인데, 우리는 각 페이지에서 영화 제목, 평점, 장르, 총평 등의 정보를 가져올 것이라고 가정합니다.

새 scrapy 프로젝트 생성

```
> scrapy startproject rt_crawler
```

rt_crawler 프로젝트 폴더 구성

```
rt_crawler/
  scrapy.cfg
  rt_crawler/          # 해당 프로젝트의 Python 모듈
    __init__.py
    items.py           # 프로젝트 Item 파일
    pipelines.py       # 프로젝트 Item pipeline 파일
    settings.py        # 프로젝트 Settings 파일
    spiders/           # Spider 저장 폴더
      __init__.py
      ...
```

Item 클래스 정의

rt_crawler/items.py 파일을 연 뒤, 다음의 내용을 추가합니다:

```
class RTItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    title = scrapy.Field()
    score = scrapy.Field()
    genres = scrapy.Field()
    consensus = scrapy.Field()
```

scrapy.Item 클래스를 상속받는 RTItem 이라는 클래스를 생성하였는데, 이 클래스의 객체에 우리가 수집하고자 하는 정보들을 저장할 것입니다. 우리는 영화 제목, 평점, 장르, 총평 등의 정보를 수집할 것이므로 각각을 RTItem 클래스의 내부 변수로 정의하고, scrapy.Field() 함수를 실행하여 이들을 '필드(field)'의 형태로 생성합니다.

Spider 클래스 정의

`rt_crawler/spiders/rt_spider.py` 파일을 새로 생성하고, 다음의 내용을 추가합니다:

```
import scrapy

from rt_crawler.items import RTItem

class RTSpider(scrapy.Spider):
    name = "RottenTomatoes"
    allowed_domains = ["rottentomatoes.com"]
    start_urls = [
        "https://www.rottentomatoes.com/top/bestofrt/?year=2015"
    ]

    def parse(self, response):
        for tr in response.xpath('//*[@id="top_movies_main"]/div/table/tr'):
            href = tr.xpath('./td[3]/a/@href')
            url = response.urljoin(href[0].extract())
            yield scrapy.Request(url, callback=self.parse_page_contents)

    def parse_page_contents(self, response):
        item = RTItem()
        item["title"] = response.xpath('//*[@id="movie-title"]/text()')[0].extract().strip()
        item["score"] = response.xpath('//*[@id="tomato_meter_link"]/span[2]/span/text()')[0].extract()
        item["genres"] = response.xpath('//*[@id="mainColumn"]/section[3]/div/div/div[2]/div[4]//span/text()').extract() # list of genre
        consensus_list = response.xpath('//*[@id="all-critics-numbers"]/div/div[2]/p//text()').extract()[2:]
        item["consensus"] = ' '.join(consensus_list).strip()
        yield item
```

`scrapy.Spider` 클래스를 상속받는 `RTSpider` 클래스를 정의하고, `name`, `allowed_domains`, `start_urls` 변수를 순서대로 명시하였습니다. 각각의 변수는 다음과 같은 의미를 지닙니다.

- `name` : 해당 Spider의 이름. 웹 크롤링 및 스크래핑을 실행할 때 여기서 지정한 이름을 사용합니다.
- `allowed_domains` : Spider로 하여금 크롤링하도록 허가한 웹 사이트의 도메인 네임.
- `start_urls` : 웹 크롤링의 시작점이 되는 웹 페이지 URL. 해당 웹 페이지에서 출발하여 이어지는 웹 페이지들을 크롤링함.

`start_urls` 변수에 명시된 URL의 웹 페이지를 Spider가 서버에 요청하고, 이에 대한 응답을 `response` 라는 이름의 변수로 받으며, 이를 `parse()` 함수에서 처리됩니다. `parse()` 함수에서 얻어진 시작 웹 페이지의 어떠한 부분을 스크래핑할지 명시해주는 부분입니다.

`parse()` 함수 내에서, 첫 페이지의 리스트 상의 전체 100개 제목의 하이퍼링크 URL을 얻은 뒤, 이들 각각에 대하여 `scrapy.Request()` 함수를 사용하여 웹 페이지 요청을 보내도록 하였고, 이에 대한 응답을 어떻게 처리할지 `parse_page_contents()` 함수에 명시하였습니다. 그리고 `scrapy.Request()` 함수의 `callback` 인자의 값으로 `parse_page_contents` 를 입력하였습니다.

`parse_page_contents()` 함수는 각 영화 페이지 상에서 스크래핑을 어떻게 수행할지 명시합니다. 영화 페이지가 여러 개 존재함에도 불구하고 이 함수를 하나만 정의해도 되는 이유는, 모든 영화 페이지가 거의 동일한 구조를 가지고 있기 때문입니다. 크롬 개발자 도구를 사용하여 영화 제목, 평점, 장르, 총평 각각의 요소의 XPath를 알아냅니다. 그리고 앞서 정의했던 `RTItem` 클래스의 객체를 `item` 이라는 이름으로 생성한 뒤, 앞서 정의한 필드들에 각 요소들을 대입합니다.

`yield` 키워드에 관해 궁금하신 분은, 다음의 링크를 참조하시길 바랍니다:

<https://jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/>

Spider 실행

```
> scrapy crawl RottenTomatoes
```

Spider가 수집한 정보를 CSV 파일로 저장하고 싶은 경우, `-o` 옵션을 추가합니다:

```
> scrapy crawl RottenTomatoes -o rt.csv
```

Item pipeline 정의

웹 스크래핑을 통해 수집된 데이터를 어떻게 가공하고 이를 외부 파일로 어떻게 저장할지 좀 더 구체적으로 명시하기 위해, `Item pipeline`을 정의할 수 있습니다. `rt_crawler/pipelines.py` 파일을 새로 생성하고, 다음의 내용을 추가합니다:

```
import csv

class RTPipeline(object):

    def __init__(self):
        self.csvwriter = csv.writer(open("rt_movies_new.csv", "w"))
        self.csvwriter.writerow(["title", "score", "genres", "consensus"])

    def process_item(self, item, spider):
        row = []
        row.append(item["title"])
        row.append(item["score"])
        row.append(' | '.join(item["genres"]))
        row.append(item["consensus"])
        self.csvwriter.writerow(row)
        return item
```

`RTPipeline` 클래스를 새로 정의한 뒤, 생성자 함수인 `__init__()` 함수에서 Python의 `csv` 모듈을 사용하여, 매 행을 `rt_movies_new.csv` 파일에 쓰도록 선언합니다.

`process_item()` 함수에서는, `Spider`를 통해 수집한 각 `Item`을 어떻게 처리할지 명시합니다. 여기에서 Python 리스트 하나를 생성하여 `Item`의 각 필드의 값을 `append`하는데, "genres" 필드의 경우 원래 Python 리스트 형태였으므로 해당 리스트 내 성분을 '|' 문자로 연결하여 새로운 문자열을 생성한 뒤 이를 `append`하였습니다. 맨 마지막 부분에서 `csvwriter`를 통해 CSV 파일 상에 한 행을 쓰도록 하였습니다.

`Item pipeline`을 정의하였으면, 크롤링 실행 시 `Settings` 상에서 해당 `Item pipeline`을 사용하도록 설정해줘야 합니다. `rt_crawler/settings.py` 파일을 연 뒤, `ITEM_PIPELINES` 관련 부분의 내용을 다음과 같이 수정해줍니다.

```
ITEM_PIPELINES = {
    'rt_crawler.pipelines.RTPipeline': 300,
}
```

'RottenTomatoes' `Spider`를 다시 한 번 실행하면, 수집된 `Item`들이 `Item pipeline`에 의해 `rt_movies_new.csv` 파일에 CSV 포맷으로 쓰여집니다.

```
> scrapy crawl RottenTomatoes
```

`scrapy`의 사용과 관련한 자세한 내용을 알고 싶으신 경우, `scrapy`의 공식 웹 페이지 문서를 참조하시길 바랍니다.

<http://scrapy.readthedocs.io>

동적 웹 페이지에서 데이터 추출하고 수집하기

유럽 축구 데이터 수집하기: premierleague.com

- Premier League의 2015/16 시즌 최종 순위표:

<https://www.premierleague.com/tables?co=1&se=42&mw=-1&ha=-1>

해당 웹 페이지에서, 순위표 상에 나와있는 순위(Position), 팀명(Club), 경기수(Played), 승(Won), 무(Drawn), 패(Lost), 득점(GF), 실점(GA), 득실차(GD), 승점(Points) 등의 정보를 가져올 것이라고 가정합니다.

scrapy 프로젝트 생성, Item 클래스 정의

scrapy 프로젝트를 하나 생성합니다:

```
> scrapy startproject epl_crawler
```

Item 클래스를 하나 정의합니다. `epl_crawler/items.py` 파일을 연 뒤, 다음의 내용을 추가합니다:

```
class EPLItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    club_name = scrapy.Field()
    position = scrapy.Field()
    played = scrapy.Field()
    won = scrapy.Field()
    drawn = scrapy.Field()
    lost = scrapy.Field()
    gf = scrapy.Field()
    ga = scrapy.Field()
    gd = scrapy.Field()
    points = scrapy.Field()
```

selenium webdriver를 사용하여 '눈에 보이는' 웹 페이지 스크래핑하기

Spider 클래스를 새로 정의하고, 그 내용을 명시합니다. 이 때, `response` 변수를 사용하여 스크래핑을 하는 대신, `selenium webdriver` 모듈을 사용하여 목표 웹 페이지 URL에 대한 요청을 다시 보내고, 이에 대한 응답을 `selector` 라는 새로운 변수로 받아 이를 스크래핑할 것입니다.

`epl_crawler/spiders/epl_spider.py` 파일을 새로 생성하고, 다음의 내용을 추가합니다:

```

import scrapy
from selenium import webdriver

from epl_crawler.items import EPLItem

class EPLSpider(scrapy.Spider):
    name = "PremierLeague"
    allowed_domains = ["premierleague.com"]
    start_urls = [
        "https://www.premierleague.com/tables?co=1&se=42&mw=-1&ha=-1"
    ]

    def __init__(self):
        scrapy.Spider.__init__(self)
        self.browser = webdriver.Chrome("/Users/kilho/chromedriver")

    def parse(self, response):
        self.browser.get(response.url)
        time.sleep(5)

        html = self.browser.find_element_by_xpath('//*').get_attribute('outerHTML')
        selector = Selector(text=html)
        rows = selector.xpath('//*[@id="mainContent"]/div/div[1]/div[3]/div/div/table/
tbody/tr[not(@class="expandable")]')
        for row in rows:
            item = EPLItem()
            item["club_name"] = row.xpath('./td[3]/a/span[2]/text()')[0].extract()
            item["position"] = row.xpath('./td[2]/span[1]/text()')[0].extract()
            item["played"] = row.xpath('./td[4]/text()')[0].extract()
            item["won"] = row.xpath('./td[5]/text()')[0].extract()
            item["drawn"] = row.xpath('./td[6]/text()')[0].extract()
            item["lost"] = row.xpath('./td[7]/text()')[0].extract()
            item["gf"] = row.xpath('./td[8]/text()')[0].extract()
            item["ga"] = row.xpath('./td[9]/text()')[0].extract()
            item["gd"] = row.xpath('./td[10]/text()')[0].extract()
            item["points"] = row.xpath('./td[11]/text()')[0].extract()
            yield item

```

`EPLSpider` 클래스의 생성자인 `__init__()` 함수에서 `webdriver.Chrome()` 함수를 호출, `webdriver`로 하여금 크롬 브라우저를 띄우도록 한 뒤, 이를 `self.browser` 변수에 저장하였습니다. `parse()` 함수에서는 `self.browser` 를 사용하여 `.get()` 함수를 호출, 해당 URL의 웹 페이지를 다시 불러오는 과정을 진행하고 있습니다.

`self.browser` 가 불러들인 웹 페이지의 HTML 소스 코드를 가져오기 위해, `self.browser.find_element_by_xpath('//*').get_attribute('outerHTML')` 를 실행하고, 그 결과물을 `html` 변수에 저장하였습니다. 그 다음 `scrapy.Selector()` 함수를 사용하여 `scrapy`의 `Selector` 객체를 새로 생성하는데, 이 때 `text` 인자에 `html` 변수를 입력합니다.

이 때, `self.browser.get()` 함수를 호출하는 부분과

`self.browser.find_element_by_xpath().get_attribute()` 함수 호출 부분 사이에 `time.sleep(5)` 함수를 호출합니다. 이는 크롬 브라우저로 하여금 동적 컨텐츠까지 충분히 로드할 시간을 주기 위해 인위적으로 5초를 대기하도록 한 것입니다.

Spider 실행

```
> scrapy crawl PremierLeague -o pl.csv
```

본 수업에서는 로그인에 필요한 웹 페이지에 대한 크롤링 및 스크래핑 방법은 다루지 않았습니다. 그 방법에 대한 아이디어를 얻고자 하시는 분은, `selenium` 공식 웹 사이트의 문서를 참조하시길 바랍니다.

<http://selenium-python.readthedocs.io/>

1. 파이썬 Web Scraping

웹사이트에서 HTML을 읽어와 필요한 데이터를 긁어오는 것을 Web Scraping이라 한다. 이 과정은 크게 웹페이지를 읽어오는 과정과 읽어온 HTML 문서에서 필요한 데이터를 뽑아내는 과정으로 나눌 수 있다.

웹페이지를 읽어오는 일은 여러 모듈을 사용할 수 있는데, 파이썬에서 기본적으로 제공하는 `urllib`, `urllib2` 을 사용하거나 편리한 HTTP 라이브러리로 많이 쓰이고 있는 `requests` 를 설치해 사용할 수 있다. 만약 기존 코드를 유지보수하는 일이 아니라면 `requests` 를 사용할 것을 권장한다.

2. requests - 웹페이지 읽어오기

HTTP 라이브러리인 `requests`를 사용하기 위해서는 먼저 아래와 같이 `pip`을 이용하여 `requests` 패키지를 설치한다.

```
pip install requests
```

기본적인 `requests` 기능을 먼저 살펴보면, `requests`는 HTTP GET, POST, PUT, DELETE 등을 사용할 수 있으며, 편리한 데이터 인코딩 기능을 제공하고 있다. 즉, 데이터를 Dictionary로 만들어 GET, POST 등에서 사용하면 필요한 Request 인코딩을 자동으로 처리해 준다.

```
import requests

# GET
resp = requests.get('http://httpbin.org/get')
print(resp.text)

# POST
dic = {"id": 1, "name": "Kim", "age": 10}
resp = requests.post('http://httpbin.org/post', data=dic)
print(resp.text)

resp = requests.put('http://httpbin.org/put')
resp = requests.delete('http://httpbin.org/delete')
```

`requests.get(url)` 함수를 사용하면 해당 웹페이지 호출 결과를 가진 `Response` 객체를 리턴한다. `Response` 객체는 HTML Response와 관련된 여러 attribute들을 가지고 있는데, 예를 들어, `Response`의 `status_code` 속성을 체크하여 HTTP Status 결과를 체크할 수 있고, `Response`에서 리턴된 데이터를 문자열로 리턴하는 `text` 속성이 있으며, `Response` 데이터를 바이트(bytes)로 리턴하는 `content` 속성 등이 있다. 또한, 만약 `Response`에서 에러가 있을 경우 프로그램을 중단하도록 할 때는 `Response` 객체의 `raise_for_status()` 메서드를 호출할 수 있다.

아래 예제는 다음 홈페이지에 접속해서 HTML 문서를 가져와 화면에 출력하는 예이다.

```
import requests

resp = requests.get( 'http://daum.net' )
# resp.raise_for_status()

if (resp.status_code == requests.codes.ok):
    html = resp.text
    print(html)
```

requests 에서의 한글 깨짐 문제

여기서 간혹 겪게되는 한글 깨짐 문제에 대해 잠깐 짚고 넘어가자. `requests` 에서 웹 호출을 진행한 후 결과는 `Response` 객체에 담기게 되는데, `Response`의 `text` 속성은 `str` 클래스 타입으로서 보통 `requests` 모듈에서 자동으로 데이터를 인코딩해 준다. 즉, `requests`는 HTTP 헤더를 통해 결과 데이터의 인코딩 방식을 추측하여 `Response` 객체의 `encoding` 속성에 그 값을 지정하고, `text` 속성을 액세스할 때 이 `encoding` 속성을 사용한다. 만약 인코딩 방식을 변경해야 한다면, `text` 속성을 읽기 전에 `Response`의 `encoding` 속성을 변경하면 된다.

이제 실제 예를 들어 보면, 네이버 홈페이지는 한글 출력에 문제가 없지만, 네이버 증권사이트 웹페이지는 (영문 OS에서 테스트한 결과) 한글이 깨져 보이게 된다. 원인을 찾아보기 위해 `Response` 객체가 어떤 인코딩인지 체크해 보았다. 네이버 홈페이지는 UTF-8을 사용하고, 네이버 증권사이트는 ISO-8859-1을 사용하고 있다.

```
>
>
>
resp = requests.get('http://naver.com') # 네이버 홈

>
>
>
resp.encoding
'UTF-8'

>
>
>
resp = requests.get('http://finance.naver.com') # 증권

>
>
>
resp.encoding
'ISO-8859-1'
```


인코딩이 유니코드 인코딩(예: UTF-8 등)이거나 한글 인코딩(예: EUC-KR)이면 일반적으로 한글이 깨지지 않지만, ISO-8859-1와 같이 영문 인코딩이면 한글이 깨지게 된다. 이를 해결하는 방법은 미리 Response 객체의 encoding 을 한글인코딩(예: EUC-KR)이나 None (인코딩 추측을 하지 않도록) 으로 지정한 후, text 속성을 읽으면 된다. 예를 들어, 아래 예제는 네이버 증권사이트의 ISO-8859-1 인코딩 문제를 처리한 코드이다.

```
import requests

resp = requests.get('http://finance.naver.com/')
resp.raise_for_status()

resp.encoding=None    # None 으로 설정
#resp.encoding='euc-kr'  # 한글 인코딩

html = resp.text
print(html)
```

3. BeautifulSoup - 웹페이지 파싱

웹페이지 HTML 문서를 파싱(Parsing)하기 위해서는 BeautifulSoup 라는 모듈을 사용할 수 있다. 먼저 BeautifulSoup 를 아래와 같이 설치한다.

```
pip install beautifulsoup4
```

BeautifulSoup를 사용하기 위해서는 먼저 BeautifulSoup 모듈을 import하여야 하는데 모듈명은 bs4 이다. bs4 모듈이 import 된 후, bs4.BeautifulSoup(HTML문서) 생성자를 호출하여 BeautifulSoup 객체를 생성한다.

```
import bs4

html = """
<
html
>
<
body
>
...생략...
<
/body
>
<
/html
>
"""

bs = bs4.BeautifulSoup(html, 'html.parser')
```

BeautifulSoup 객체에서 특정 HTML 태그(들)을 찾기 위해 `select()` 메서드를 사용하는데, 이 메서드의 파라미터로 어떤 태그(들)을 찾을 지를 CSS 스타일의 Selector로 지정하면 된다. 예를 들어, `select('.news li')` 는 news 라는 CSS 클래스 안에 모든 li 태그들을 리턴하게 된다.

리턴된 결과는 태그(`s4.element.Tag`)들의 리스트(list) 인데, 각 태그 요소(`bs4.element.Tag`)로부터 태그내 문자열을 리턴하기 위해서는 `getText()`를, 특정 태그 attribute를 얻기 위해서는 `get('attribute명')` 메서드 등을 사용할 수 있다.

아래 예제는 네이버 증권사이트에서 주요 Top 뉴스 제목을 발췌하는 코드이다.

```
# -*- coding: utf-8 -*-

import requests, bs4

resp = requests.get('http://finance.naver.com/')
resp.raise_for_status()

resp.encoding='euc-kr'
html = resp.text

bs = bs4.BeautifulSoup(html, 'html.parser')
tags = bs.select('div.news_area h2 a') # Top 뉴스
title = tags[0].getText()
print(title)
```

1. Selenium 소개

Selenium은 웹 브라우저를 컨트롤하여 웹 UI 를 Automation 하는 도구 중의 하나이다. Selenium 은 Selenium Server와 Selenium Client가 있는데, 로컬 컴퓨터의 웹 브라우저를 컨트롤하기 위해서는 Selenium Client 를 사용한다 (여기서는 Selenium 3 사용). Selenium Client는 WebDriver 라는 공통 인터페이스(Common interface)와 각 브라우저 타입별(IE, Chrome, FireFox 등)로 하나씩 있는 Browser Driver로 구성되어 있다.

2. Selenium 설치

Selenium을 설치하기 위해서는 먼저 아래와 같이 pip 을 사용하여 Selenium Client 모듈을 설치한다.

```
pip install selenium
```

다음으로 사용할 브라우저별 Selenium 드라이버를 설치한다. 드라이버가 설치된 후, 해당 드라이버의 경로를 실행 PATH에 넣어 준다. 아래는 대표적인 브라우저별 설치 링크이다. 특별한 이유가 없다면 Selenium이 가장 잘 동작하는 Firefox를 사용하는 것이 좋다.

- Firefox : <https://github.com/mozilla/geckodriver/releases>
- Chrome : <https://sites.google.com/a/chromium.org/chromedriver/downloads>
- Edge : <https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/>

3. Selenium 사용법

Selenium을 사용하기 위해서는 먼저 `selenium.webdriver` 모듈을 import 한 후, `webdriver.Firefox()` 를 호출하여 브라우저를 실행시킨다. 만약 크롬을 사용할 경우 `webdriver.Chrome()`을 호출하고, Edge를 사용할 경우 `webdriver.Edge()`을 호출한다.

브라우저를 띄운 상태에서 특정 웹사이트로 이동하기 위해서는 아래와 같이 `browser` 객체의 `get()` 메서드를 사용한다.

```
from selenium import webdriver

browser = webdriver.Firefox()
# browser = webdriver.Chrome()

browser.get("http://python.org")
```

Selenium은 웹페이지 내의 특정 요소(들)을 찾는 많은 메서드들을 제공하고 있는데, 이들은 보통 한 요소를 리턴하는 `find_element_*`() 혹은 복수 요소를 리턴하는 `find_elements_*`() 메서드로 구분된다. 자주 사용되는 몇가지 검색 메서드를 예를 들면, 특정 태그 id 로 검색하는 `find_element_by_id()`, 특정 태그 name 속성으로 검색하는 `find_element_by_name()`, CSS 클래스명으로 검색하는 `find_element_by_class_name()`, CSS selector를 사용해 검색하는 `find_element_by_css_selector()` 등이 있는데, 예상되는 결과가 복수이면 `find_element_*` 대신 `find_elements_*` 를 사용한다.

검색 결과 리턴되는 객체는 `FirefoxWebElement` 와 같이 `*WebElement` 타입의 객체가 되는데, 리턴된 요소는 `WebElement` 타입 타입의 속성이나 메서드를 사용하여 데이터를 얻거나 특정 행위를 할 수 있다. 예를 들어, `WebElement`의 `text`는 요소 내의 문자열을 리턴하고, `tag_name` 은 해당 요소의 태그명 (예: `a`, `span`) 을 리턴하며, `clear()` 메서드를 호출하면 `text` 입력 영역을 초기화하고, `click()` 메서드를 호출하면 해당 요소를 클릭한다.

아래 예제는 `python.org` 웹사이트를 방문해서 상단 메인 메뉴 문자열을 출력하고, PyPI 메뉴를 클릭한 후 5초 후에 브라우저를 종료하는 예이다.

```
from selenium import webdriver
import time

browser = webdriver.Firefox()
browser.get("http://python.org")

menus = browser.find_elements_by_css_selector('#top ul.menu li')

pypi = None
for m in menus:
    if m.text == "PyPI":
        pypi = m
        print(m.text)

pypi.click() # 클릭

time.sleep(5) # 5초 대기
browser.quit() # 브라우저 종료
```

파이썬 데이터 분석

데이터 분석이란?

데이터란, 어떤 사실에 대한 측정이나 관찰을 통해 모아놓은 값들의 모음을 말합니다. 보통 '모음'이라는 사실을 강조하기 위해 '데이터셋(data set)'이라는 표현도 많이 쓰입니다.

분석이란, 어떤 대상을 더 잘 이해하기 위해, 이를 나누고 쪼개는 것을 말합니다.

즉 **데이터 분석**이란, 어떤 대상에 대한 사실을 설명하는 값의 모음을 유효적절한 방법으로 나누고 쪼개서 그 대상을 더 잘 이해하는 것이라고 할 수 있습니다.

빅 데이터에 대한 탐색적 데이터 분석

탐색적 데이터 분석(EDA. exploratory data analysis)은 기계 학습, 자연어 처리, 패턴 인식 등 모든 데이터 과학의 출발점이라고 할 수 있습니다. 이는 각종 통계적 기법을 적용하여 데이터의 주요 특징을 발견하고, 이를 시각화하여 표현하는 일련의 과정을 포괄합니다.

본 수업에서는 이 탐색적 데이터 분석 방법을 배울 것이며, 이를 곧 '데이터 분석'으로 지칭할 것입니다.

데이터 분석의 가치

1. 버락 오바마 미국 대통령의 재선 사례

오바마 선거 캠프에서는 유권자들의 64%가 인터넷을 사용하여 대선 후보자들의 발언의 진위 여부를 조사한다는 연구 결과에 주목하여, 유권자들이 구글에서 어떤 검색어를 사용하였을 때 선거 캠프 웹 페이지로 가장 많이 유입되는지를 데이터 분석을 통해 실시간으로 조사하였습니다.

데이터 분석에 기반한 이러한 전략은, 결과적으로 버락 오바마의 재선을 성공시키는 주된 요인으로 작용하였습니다.

2. 아마존의 추천 시스템 사례

온라인 서점으로 유명한 아마존에서는 고객들이 이전에 구매했던 서적 데이터에 대한 분석을 통해, 해당 고객과 유사한 구매 성향의 고객들이 구매한 서적을 추천해 주는 방식으로 자체적인 추천 시스템을 구축하였습니다.

이 추천 시스템 도입으로 인한 매출은, 오늘날 아마존이 서적 판매를 통해 거둬들이는 전체 매출의 35% 가량을 차지한다고 합니다.

3. 2014 브라질 월드컵에서의 독일 국가대표 축구팀 사례

독일축구협회에서는 SAP AG 측으로 독일 국가대표 축구팀의 경기 영상을 촬영한 비디오 데이터와, 선수 개인별 기록 데이터, 팀 기록 데이터 등에 대한 분석을 의뢰하였습니다. SAP AG 측에서는 해당 데이터에 대한 분석을 통해 선수 개인의 기량을 어떻게 향상시킬 수 있을지, 팀워크를 어떻게 향상시킬 수 있을지에 대한 디테일한 피드백을 제공하였습니다.

그 결과, 독일 대표팀은 준결승전에서 홈 팀인 브라질을 7:1로 대파하고, 결승전에서 아르헨티나마저 꺾어버리는 놀라운 성적을 보였습니다.

'빅 데이터'의 등장

빅 데이터(big data)는, 양이 너무 많고 복잡해서, 통상적으로 사용되는 데이터 분석 방법이 잘 적용되지 않는 데이터를 의미합니다.

통계 이론의 핵심적인 이론인 '**큰 수의 법칙(law of large numbers)**'에 의하면, 표본의 크기가 커질수록 그 데이터의 평균이 실제 모집단 데이터의 평균에 가까워진다는 사실이 알려져 있습니다. 이는 양면으로 구성된 동전을 더 많이 던질수록, 전체 던진 횟수 대비 앞면이 나온 횟수의 비율이 (이론적으로 동전의 앞면이 나올 확률인) 0.5에 가까워진다는 결과를 통해 확인할 수 있습니다.

이 법칙을 데이터의 측면에서 바라보면, 어떤 대상과 관련된 사실의 단면을 보여주는 데이터의 양이 많을수록, 분석을 통해 대상에 대한 이해를 보다 사실에 가깝게 할 수 있을 가능성이 높아진다는 것을 입증하고 있습니다. 다시 말해, 데이터의 양이 많아지면서 빅 데이터에 가까워질수록, 분석 결과를 '**일반화(generalization)**'하기가 더 수월해진다는 것입니다.

이 때문에 데이터 분석을 하고자 하는 사람들의 입장에서는 "어떻게 하면 더 많은 데이터를 수집할 수 있을까?"가 항상 화두였는데, 오늘날 기술의 발전으로 데이터가 축적되는 속도 자체가 매우 높아졌으며, 이로 인해 동일한 시간과 노력을 들이더라도 더 많은 데이터를 수집하는 것이 가능해졌습니다.

오늘날, 분야를 막론하고 빅 데이터에 대한 분석은 가히 일의 성패를 좌우할 수 있는 중요한 과제로 부각되고 있습니다.

왜 Python인가?

여러분이 사용할 '빅 데이터'

오늘날의 빅 데이터는 일반적으로 테라바이트(TB) 혹은 페타바이트(PB) 수준의 용량을 가지는 데이터를 지칭하나, 여러분과 같은 개인의 수준에서 저 정도 용량을 다루는 것은 불가능에 가깝습니다.

그러므로, 본 수업에서 여러분이 다룰 '빅 데이터'는 대략 수 백 메가바이트(MB) 혹은 수 기가바이트(GB) 수준 용량을 가지는 데이터라고 재정의하도록 하겠습니다.

샘플 데이터: US Baby Names

[US Baby Names 데이터셋 다운로드] -

<https://drive.google.com/file/d/0B9fcvsgEhJNsN0FUaGZudFYyTVU/view>

엑셀(Excel)을 사용하여 데이터 분석을 해 보고자, 'US Baby Names'라는 이름의 샘플 데이터셋을 사용하도록 하겠습니다. `NationalNames.csv` 라는 이름의 CSV 파일로 구성되어 있으며, 용량이 약 42.2MB 정도 됩니다. 이 데이터셋은 미국에서 1880년부터 2014년까지 태어난 남자 아이와 여자 아이의 이름(first name)에 따른 출생횟수를 나타내는 데이터셋입니다.

엑셀을 사용한 샘플 데이터 분석

엑셀을 사용하여 `NationalNames.csv` 파일을 연 뒤 분석하려고 하면, 몇 가지 치명적인 문제가 있는 것을 확인할 수 있습니다.

- 데이터셋에서 최대 1,048,576행까지밖에 로드할 수 없습니다. US Baby Names 데이터셋이 총 1,825,434 행으로 구성되어 있으므로, 거의 절반에 육박하는 분량의 데이터를 흘려보낸 것입니다.
- 부분합 등의 단순한 작업도, 실행 속도가 너무 오래 걸립니다.
- 좀 더 복잡한 분석을 하려고 하면, 작성해야 하는 엑셀 함수가 너무 복잡해집니다.

Python을 사용한 샘플 데이터 분석

Python을 사용하여 `NationalNames.csv` 파일을 열어 분석하면, 전체 데이터셋을 온전히 로드할 수 있으며 분석 작업을 빠른 속도로 수행할 수 있습니다. Python 라이브러리의 특성 상 매 분석 결과를 화면에 모두 표시해주지 않으므로, 이로 인해 빠른 분석 속도를 발휘할 수 있습니다.

그럼 왜 수많은 프로그래밍 언어 중 Python인가?

다른 수많은 프로그래밍 언어들 중, 데이터 분석을 위해 Python을 추천하는 이유는 다음과 같습니다.

- 언어 자체가 너무 쉽습니다.
- 컴퓨터와 대화하듯이 프로그래밍이 가능합니다.
- numpy, pandas, matplotlib 등의 강력한 데이터 분석 라이브러리를 제공합니다.
- 모두 오픈 소스(open source)이며, **공짜**입니다.

R 언어의 경우에도 위와 같은 장점을 모두 가지고 있는 스크립트 언어이나, R의 경우 통계 분석 및 리서치 작업 등에 특화된 성격이 강합니다. 반면, Python의 경우 다양한 목적으로 범용적으로 사용할 수 있는 언어입니다. 이는 Python으로 데이터 분석한 결과물을 웹 사이트를 통해 외부에 공개하고자 할 때 매우 유리한 특징이라고 할 수 있습니다.

원활한 Python 데이터 분석을 위한 핵심 도구

Python에서는 Python shell이라고 부르는 기본적인 대화식 프로그래밍 툴을 제공하는데, **IPython**은 이 기본 툴에 몇 가지 강력한 기능을 덧붙인 툴이라고 할 수 있습니다.

IPython Notebook은, IPython의 대화식 프로그래밍 방식을 기본적으로 제공하면서, 여러분이 데이터 분석을 하는 과정을 노트 형식으로 보기 쉽게 기록하고 정리해 놓을 수 있도록 도와주는 강력한 툴입니다.

Python 데이터 분석 라이브러리

numpy

numpy는 주요한 Python 데이터 분석 라이브러리들의 기본 베이스가 되는 라이브러리입니다. numpy는 특히 벡터(vector) 및 행렬(array) 연산에 있어 엄청난 편의성을 제공하는 라이브러리입니다.

여러분이 데이터 분석을 하는 데 직접적으로 많이 사용하지는 않겠지만, 추후 많이 사용하게 될 pandas와 matplotlib의 기반이 되는 라이브러리라고 할 수 있습니다.

pandas

여러분은 **pandas**를 가장 많이 사용하게 될 것입니다. pandas는 고유하게 정의한 Series 및 DataFrame 등의 자료구조를 활용하여, 빅 데이터 분석에 있어 높은 수준의 퍼포먼스를 가능하게 하는 라이브러리입니다.

여러분이 기존에 엑셀로 하던 모든 분석을 더 큰 스케일의 데이터에 적용할 수 있으며, 더 빠른 속도로 수행할 수 있습니다.

matplotlib

matplotlib은 데이터 시각화를 위한 라이브러리입니다. 엑셀에서 차트를 그릴 경우 데이터의 양이 조금만 많아지더라도 엄청나게 버벅이는데, **matplotlib**을 사용하면 데이터 분석 결과에 대한 시각화를 빠르고 깔끔하게 수행해줍니다.

1. numpy 패키지

numpy는 과학 계산을 위한 라이브러리로서 다차원 배열을 처리하는데 필요한 여러 유용한 기능을 제공하고 있다.

numpy는 pip을 사용하여 아래와 같이 간단히 설치할 수 있다.

```
$ pip install numpy
```

2. numpy 배열

numpy에서 배열은 동일한 타입의 값들을 가지며, 배열의 차원을 **rank** 라 하고, 각 차원의 크기를 튜플로 표시하는 것을 **shape** 라 한다. 예를 들어, 행이 2이고 열이 3인 2차원 배열에서 rank는 2 이고, shape는 (2, 3) 이 된다.

numpy 배열을 생성하는 방법은 파이썬 리스트를 사용하는 방법과 numpy에서 제공하는 함수를 사용하는 방법이 있다. 아래 예제에서 list1은 4개의 요소를 갖는 리스트인데, 이를 array() 함수에 넣어 numpy 배열을 생성하는데, 이 배열의 rank는 1이 되고, shape는 (4,) 가 된다. 튜플에 하나의 요소만 있으면 문법상 콤마를 뒤에 붙인다. 두번째 배열 b는 2x3 배열로서 shape는 (2, 3)이 되는데, 한가지 주의할 점은 array() 안에 하나의 리스트만 들어가므로 리스트의 리스트를 넣어야 한다.

```
import numpy as np

list1 = [1, 2, 3, 4]
a = np.array(list1)
print(a.shape) # (4, )

b = np.array([[1,2,3],[4,5,6]])
print(b.shape) # (2, 3)
print(b[0,0]) # 1
```

numpy에서 제공하는 함수를 사용하여 numpy 배열을 만드는 방법을 살펴보자. 이러한 기능을 제공하는 함수로는 zeros(), ones(), full(), eye() 등이 있는데, zeros()는 해당 배열에 모두 0을 집어 넣고, ones()는 모두 1을 집어 넣는다. full()은 배열에 사용자가 지정한 값을 넣는데 사용하고, eye()는 대각선으로는 1이고 나머지는 0인 2차원 배열을 생성한다.

아래 예제는 이들 함수들을 사용하여 numpy 배열을 생성한 예이다. 그리고 마지막 예는 0부터 n-1까지의 숫자를 생성하는 range(n) 함수와 배열을 다차원으로 변형하는 reshape()를 통해 간단하게 샘플 배열을 생성해 본 것이다.

```
import numpy as np

a = np.zeros((2,2))
print(a)
# 출력:
# [[ 0.  0.]
#  [ 0.  0.]]

a = np.ones((2,3))
print(a)
# 출력:
# [[ 1.  1.  1.]
#  [ 1.  1.  1.]]

a = np.full((2,3), 5)
print(a)
# 출력:
# [[5 5 5]
#  [5 5 5]]

a = np.eye(3)
print(a)
# 출력:
# [[ 1.  0.  0.]
#  [ 0.  1.  0.]
#  [ 0.  0.  1.]]

a = np.array(range(20)).reshape((4,5))
print(a)
# 출력:
# [[ 0  1  2  3  4]
#  [ 5  6  7  8  9]
#  [10 11 12 13 14]
#  [15 16 17 18 19]]
```

3. numpy 슬라이싱

numpy 배열은 파이썬 리스트와 마찬가지로 슬라이스(Slice)를 지원한다. numpy 배열을 슬라이싱 하기 위해서는 각 차원별로 슬라이스 범위를 지정한다.

```
import numpy as np

lst = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
arr = np.array(lst)

# 슬라이스
a = arr[0:2, 0:2]
print(a)
# 출력:
# [[1 2]
#  [4 5]]

a = arr[1:, 1:]
print(a)
# 출력:
# [[5 6]
#  [8 9]]
```

4. numpy 정수 인덱싱 (integer indexing)

numpy 슬라이싱이 각 배열 차원별 최소-최대의 범위를 정하여 부분 집합을 구하는 것이라면, 정수 인덱싱은 각 차원별로 선택되어지는 배열요소의 인덱스들을 일렬로 나열하여 부분집합을 구하는 방식이다. 즉, 임의의 numpy 배열 *a* 에 대해 *a*[[row1, row2], [col1, col2]] 와 같이 표현하는 것인데, 이는 *a*[row1, col1] 과 *a*[row2, col2] 라는 두 개의 배열요소의 집합을 의미한다.

예를 들어, 아래 예제에서 *a*[[0, 2], [1, 3]] 은 정수 인덱싱으로서 이는 *a*[0, 1] 과 *a*[2, 3] 등 2개의 배열요소를 가리킨다.

```
import numpy as np

lst = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
a = np.array(lst)

# 정수 인덱싱
s = a[[0, 2], [1, 3]]

print(s)
# 출력
# [2 12]
```

5. numpy 부울린 인덱싱 (boolean indexing)

numpy 부울린 인덱싱은 배열 각 요소의 선택여부를 True, False로 표현하는 방식이다. 만약 배열 **a** 가 2 x 3 의 배열이이라면, 부울린 인덱싱을 정의하는 numpy 배열도 2 x 3 으로 만들고 선택할 배열요소에 True를 넣고 그렇지 않으면 False를 넣으면 된다.

예를 들어, 아래 예제에서 3 x 3 배열 **a** 중 짝수만 뽑아내는 부울린 인덱싱 배열(numpy 배열)을 사용하여 짝수 배열 **n** 을 만드는 예이다.

```
import numpy as np

lst = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
a = np.array(lst)

bool_indexing_array = np.array([
    [False, True, False],
    [True, False, True],
    [False, True, False]
])

n = a[bool_indexing_array];
print(n)
```

부울린 인덱싱 배열에 True/False 값을 일일이 지정하는 방법 이외에 표현식을 사용하여 부울린 인덱싱 배열을 생성하는 방법이 있다. 예를 들어, 배열 **a** 에 대해 짝수인 배열요소만 True로 만들고 싶다면, `bool_indexing = (a % 2 == 0)` 와 같이 표현할 수 있다. 아래 예제는 이러한 표현을 사용하는 것을 예시한 것이고, 특히 마지막에 `a[a % 2 == 0]` 와 같이 부울린 인덱싱 표현식을 배열 인덱스안에 넣어 간단하게 표현할 수도 있다.

```
import numpy as np

lst = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
a = np.array(lst)

# 배열 a 에 대해 짝수면 True, 홀수면 False
bool_indexing = (a % 2 == 0)

print(bool_indexing)
# 출력: 부울린 인덱싱 배열
# [[False  True False]
#  [ True False  True]
#  [False  True False]]

# 부울린 인덱스를 사용하여 True인 요소만 뽑아냄
print(a[bool_indexing])
# 출력:
# [2 4 6 8]

# 더 간단한 표현
n = a[ a % 2 == 0 ]
print(n)
```

6. numpy 연산

numpy를 사용하면 배열간 연산을 쉽게 실행할 수 있다. 연산은 +, -, *, / 등의 연산자를 사용할 수도 있고, add(), subtract(), multiply(), divide() 등의 함수를 사용할 수도 있다. 예를 들어, 아래 예제와 같이 배열 a 와 b 가 있을때, a + b를 하면 각 배열요소의 합을 구하는데 즉 a[0]+b[0], a[1]+b[1], ... 과 같은 방식으로 결과를 리턴하게 된다.

```
import numpy as np

a = np.array([1,2,3])
b = np.array([4,5,6])

# 각 요소 더하기
c = a + b
# c = np.add(a, b)
print(c) # [5 7 9]

# 각 요소 빼기
c = a - b
# c = np.subtract(a, b)
print(c) # [-3 -3 -3]

# 각 요소 곱하기
# c = a * b
c = np.multiply(a, b)
print(c) # [4 10 18]

# 각 요소 나누기
# c = a / b
c = np.divide(a, b)
print(c) # [0.25 0.4 0.5]
```

numpy에서 vector와 matrix의 product를 구하기 위해서 dot() 함수를 사용한다. 아래 예제는 두 matrix의 product를 구한 예이다.

```
import numpy as np

lst1 = [
    [1,2],
    [3,4]
]

lst2 = [
    [5,6],
    [7,8]
]
a = np.array(lst1)
b = np.array(lst2)

c = np.dot(a, b)
print(c)
# 출력:
# [[19 22]
#  [43 50]]
```

numpy은 배열간 연산을 위한 위해 많은 함수들을 제공하는데, 예를 들어, 각 배열 요소들을 더하는 `sum()` 함수, 각 배열 요소들을 곱하는 `prod()` 함수 등을 사용할 수 있다. 이들 함수에 선택옵션으로 `axis` 을 지정할 수 있는데, 예를 들어 `sum()`에서 `axis`가 1 이면 행끼리 더하는 것이고, `axis`가 0 이면 열끼리 더하는 것이다.

```
import numpy as np

a = np.array([[1,2],[3,4]])

s = np.sum(a)
print(s)    # 10

# axis=0 이면, 컬럼끼리 더함
# axis=1 이면, 행끼리 더함
s = np.sum(a, axis=0)
print(s)    # [4 6]

s = np.sum(a, axis=1)
print(s)    # [3 7]

s = np.prod(a)
print(s)    # 24
```


numpy ndarray 이해하기

numpy는 특히 벡터 및 행렬 연산에 있어 편의성을 제공하는 라이브러리로, 여러분들이 추후 많이 사용하게 될 pandas와 matplotlib의 기반이 되는 라이브러리라고 할 수 있습니다.

numpy에서는 기본적으로 **array(어레이)**라는 단위로 데이터를 관리하고, 이에 대한 연산을 수행하게 됩니다. 이 때 array는, 고등 수학에서 말하는 '행렬(matrix)'과 그 성격이 거의 유사하다고 보면 됩니다.

array 정의하기

array는 기본적으로 `np.array()` 함수를 사용하여 정의합니다. 이 때 Python 리스트 혹은 기존에 정의된 다른 array가 함수의 인자로 입력됩니다. array를 생성할 때 인자로 입력되는 Python 리스트는 1차원 혹은 그 이상의 차원을 가지는 것이 될 수 있습니다.

numpy에서 array를 생성하고 사용할 때는, 해당 array의 크기(혹은 모양)에 대한 정보를 항상 트래킹하고 있는 것이 중요합니다. array에 대하여 `.shape` 멤버 변수를 확인하면, 해당 array의 크기를 즉각적으로 확인할 수 있습니다.

한편, 몇몇 특수한 array의 경우 한 번의 함수 호출로 생성할 수 있습니다. 예를 들어 모든 성분이 0으로 되어 있는 array를 정의하고자 할 경우, `np.zeros()` 함수를 사용하면 한 방에 됩니다. 혹은 모든 성분이 1로 되어 있는 array를 정의하고자 할 경우 `np.ones()` 함수를 사용하면 됩니다.

그리고 `np.arange()` 함수는 일정한 차이를 갖는 연속적인 값의 성분들로 구성된 array를 생성할 때 사용합니다. 이는 Python에서의 `range()` 함수와 그 형태가 매우 유사합니다.

`np.zeros()`, `np.ones()`, `np.arange()` 등의 함수를 호출할 시, 생성할 array의 크기가 반드시 인자로 입력되어야 합니다.

array의 데이터형

모든 array는 고유한 데이터형을 가집니다. array를 정의할 때 사용자가 데이터형을 따로 명시하지 않은 경우, numpy에서는 가장 적절한 데이터형을 자동으로 인식하여 해당 array에 부여합니다.

여러분이 array를 정의할 때 데이터형을 직접 지정할 수도 있습니다. array의 데이터형을 지정하고자 할 경우, `np.array()` 함수를 호출할 시 아래의 예시와 같이 `dtype` 인자의 값을 명시하면 됩니다.

```
arr = np.array(data, dtype=np.float64)
```

int나 float 뒤에 붙는 숫자는, 해당 array의 각 성분을 표현할 때 메모리 상에서 몇 비트를 사용할 것인지 나타내는 숫자입니다. 이 숫자가 더 클 수록, 해당 array 상에서 길이가 더 긴 정수 혹은 실수를 사용할 수 있습니다. array의 데이터형을 간단하게 나타내기 위해, numpy에서는 'i', 'u', 'f', 'c' 등과 같이 축약된 코드를 사용하기도 합니다.

여러분들이 기존에 정의한 array의 데이터형을 변환할 수도 있는데, 이 때는 `.astype()` 함수를 사용하면 됩니다. `.astype()` 함수의 인자에, 변환할 데이터형을 명시해 줍니다.

numpy에서 사용 가능한 데이터형(dtype) 정리

데이터형	데이터형 코드	설명
int8, int16, int32, int64	i1, i2, i4, i8	부호가 있는 [8, 16, 32, 64]비트 정수
uint8, uint16, uint32, uint64	u1, u2, u4, u8	부호가 없는 [8, 16, 32, 64]비트 정수
float16, float32, float64, float128	f2, f4, f8, f16	[16, 32, 64, 128]비트 실수
complex64, complex128, complex256	c8, c16, c32	[64, 128, 256]비트 복소수
bool	b	불리언 (True 또는 False)
object	O	Python 오브젝트 형
string_	S	문자열
unicode_	U	유니코드 문자열

array 관련 연산

두 array 간에는 더하기, 빼기, 곱하기, 나누기 등의 연산을 수행할 수 있습니다. '+', '-', '*', '/' 등과 같이 여러분이 일반 숫자에 대해 사용하는 연산자를 사용하면 됩니다.

이 때, 이러한 더하기, 빼기, 곱하기, 나누기 등의 연산은 두 array 상의 동일한 위치의 성분끼리 이루어지게 됩니다. 그러므로, 당연히 두 array의 모양이 같아야 계산이 가능합니다.

하나의 array와 일반 숫자 간에 연산을 하는 것도 가능한데, 이 경우 해당 숫자와 array 내 모든 성분 간에 해당 연산이 이루어지게 됩니다.

array 인덱싱 이해하기

기본 인덱싱

numpy array의 꽃은 **인덱싱(indexing)**입니다. 기본적인 인덱싱은 Python 리스트와 매우 유사합니다. `arr` 이라는 array가 있을 때, `arr[5]` 와 같이 특정한 인덱스를 명시할 수도 있고, `arr[5:8]` 과 같이 범위 형태의 인덱스를 명시할 수도 있습니다. `arr[:]` 의 경우, 해당 array의 전체 성분을 모두 선택한 결과에 해당합니다.

이러한 인덱싱 방식은 2차원 array에 대해서도 아주 유사한 방식으로 적용됩니다. `arr2d` 라는 2차원 array를 정의한 뒤, `arr2d[2, :]` 를 실행하게 되면 `arr2d` 에서 인덱스가 2에 해당하는 행(3행)의 모든 성분이 1차원 array의 형태로 얻어집니다. `arr2d[:, 3]` 을 실행하면, `arr2d` 에서 인덱스가 3에 해당하는 열(4열)의 모든 성분이 1차원 array의 형태로 얻어집니다.

2차원 array에서는 이렇게 두 개의 인덱스를 받을 수 있는데, ','를 기준으로 앞부분에는 '행'에 대한 인덱스가, 뒷부분에는 '열'에 대한 인덱스가 입력됩니다. `arr2d[1:3, :]` 혹은 `arr2d[:, :2]` 와 같이, 행 또는 열에 범위 인덱스를 적용하여 여러 개의 행 혹은 열을 얻을 수도 있습니다.

한편 2차원 array에서 4행 3열에 위치한 하나의 성분을 얻고자 하는 경우, `arr2d[3, 2]` 를 실행하면 됩니다. 인덱싱을 통해 선택한 성분에 새로운 값을 대입하는 경우에도, `arr2d[:2, 1:3] = 0` 과 같이 하면 됩니다.

여러 가지 방법으로 인덱싱을 시도하면서, array에 대한 인덱싱 방식에 익숙해지시길 바랍니다.

불리언 인덱싱

array와 관련하여 중요하게 사용되는 또 다른 인덱싱 방식이 불리언 인덱싱입니다.

```
names = np.array(["Charles", "Kilho", "Hayoung", "Charles", "Hayoung", "Kilho", "Kilho"])
data = np.array([[ 0.57587275, -2.84040808,  0.70568712, -0.1836896 ],
                 [-0.59389702, -1.35370379,  2.28127544,  0.03784684],
                 [-0.28854954,  0.8904534 ,  0.18153112,  0.95281901],
                 [ 0.75912188, -1.88118767, -2.37445741, -0.5908499 ],
                 [ 1.7403012 ,  1.33138843,  1.20897442, -0.58004389],
                 [ 1.11585923,  1.02466538, -0.74409379, -1.55236176],
                 [-0.45921447,  2.53114818,  0.5029578 , -0.24088216]])
```

위와 같은 `names` array와 `data` array가 정의되었다고 가정합니다. `names` array 내 각각의 성분이, `data` array의 각 행에 순서대로 대응된다고 가정합니다. 이러한 상황에서 이름이 "Charles"인 사람의 행 데이터만을 추출하고 싶다고 할 때, `names == "Charles"` 를 실행하면 다음과 같은 결과를 얻을 수 있습니다.

```
names == "Charles"
>>> array([ True, False, False,  True, False, False, False], dtype=bool)
```

얻어진 `array`를 잘 보면, 값이 "Charles"인 성분의 위치에는 `True`가, 그 외의 위치에는 `False`가 들어 있는 것을 확인할 수 있습니다. 이렇게 `names == "Charles"` 와 같은 조건식 형태의 코드를 실행하였을 때 생성되는 불리언 `array`를 다른 말로 '**마스크(mask)**'라고 합니다.

이런 마스크는 다른 `array`를 인덱싱하는 데 사용할 수 있습니다. 예를 들어 `data[names == "Charles", :]` 를 실행하면, 위에서 보인 마스크가 `True`에 해당하는 행만을 `data` `array`로부터 가져오게 되고, 이들만으로 구성된 `array`를 얻을 수 있습니다.

```
data[names == "Charles", :]
>>> array([[ 0.57587275, -2.84040808,  0.70568712, -0.1836896 ],
           [ 0.75912188, -1.88118767, -2.37445741, -0.5908499 ]])
```

이러한 조건을 여러 개 추가할 수도 있습니다. 예를 들어 값이 "Charles" 혹은 "Kilho"인 성분에 대응되는 행을 얻고 싶다면, 다음과 같이 하면 됩니다.

```
data[(names == "Charles") | (names == "Kilho"), :]
>>> array([[ 0.57587275, -2.84040808,  0.70568712, -0.1836896 ],
           [-0.59389702, -1.35370379,  2.28127544,  0.03784684],
           [ 0.75912188, -1.88118767, -2.37445741, -0.5908499 ],
           [ 1.11585923,  1.02466538, -0.74409379, -1.55236176],
           [-0.45921447,  2.53114818,  0.5029578 , -0.24088216]])
```

혹은 기존의 `data` `array`의 각 성분의 값을 기준으로 불리언 인덱싱을 수행할 수도 있습니다. 예를 들어 `data[:, 3]` 을 실행하여 4열만의 값을 본다고 할 때, `data[:, 3] < 0` 을 실행하여 4열 상에서 그 값이 0보다 작은 성분을 조사하면, 다음과 같은 마스크를 얻을 수 있습니다.

```
data[:, 3] < 0
>>> array([ True, False, False,  True,  True,  True,  True], dtype=bool)
```

`data` `array`의 4열의 값이 0보다 작은 행에 대해서는 행 내 모든 성분에 0을 대입하도록 해 봅시다. 다음과 같이 하면 됩니다.

```
data[data[:, 3] < 0, :] = 0
```

불리언 인덱싱은 처음 접할 때 굉장히 헷갈릴 수 있기 때문에, 여러분들이 직접 다양한 `array`에 대하여 반복 연습해 보면서 제대로 익힐 필요가 있습니다.

array 관련 함수 사용하기

각 성분에 적용되는 함수

numpy에서는 array에 적용할 수 있는 다양한 함수를 제공합니다. 이들 중 array의 각 성분에 대하여 특정한 계산을 일괄적으로 수행하기 위한 함수들이 있습니다.

`arr` array가 정의되어 있다고 할 때, `np.sqrt()` 함수를 적용하면 `arr` array의 각 성분에 대해 제곱근을 일괄적으로 계산합니다. 반면 `np.log10()` 함수를 적용하면 각 성분에 대해 밑이 10인 상용로그 값을 계산합니다.

numpy에서 한 개의 array의 각 성분에 적용되는 함수 정리

함수	설명
<code>abs</code>	각 성분의 절댓값 계산
<code>sqrt</code>	각 성분의 제곱근 계산 (<code>array ** 0.5</code> 의 결과와 동일)
<code>square</code>	각 성분의 제곱 계산 (<code>array ** 2</code> 의 결과와 동일)
<code>exp</code>	각 성분을 무리수 e 의 지수로 삼은 값을 계산
<code>log</code> , <code>log10</code> , <code>log2</code>	자연로그(밑이 e), 상용로그(밑이 10), 밑이 2인 로그를 계산
<code>sign</code>	각 성분의 부호 계산 (+인 경우 1, -인 경우 -1, 0인 경우 0)
<code>ceil</code>	각 성분의 소수 첫 번째 자리에서 올림한 값을 계산
<code>floor</code>	각 성분의 소수 첫 번째 자리에서 내림한 값을 계산
<code>isnan</code>	각 성분이 NaN(Not a Number)인 경우 True를, 그렇지 않은 경우 False를 반환
<code>isinf</code>	각 성분이 무한대(infinity)인 경우 True를, 그렇지 않은 경우 False를 반환
<code>cos</code> , <code>cosh</code> , <code>sin</code> , <code>sinh</code> , <code>tan</code> , <code>tanh</code>	각 성분에 대한 삼각함수 값을 계산

만약 크기가 같은 두 개의 array `x`, `y`가 아래와 같이 주어졌을 때, `np.maximum()` 함수를 적용하면 두 array를 동일한 위치의 성분끼리 비교하여 값이 더 큰 성분을 선택, 새로운 array의 형태로 제시합니다.

```
x = np.array([ 0.97121145,  1.74277758,  0.17706708, -1.14078851,  1.02197222,
               -0.75747493,  0.4994057 , -0.03462392])
y = np.array([ 0.91984849,  1.98745872, -0.11232596,  1.47306221,  1.24527437,
               -0.77047603,  0.30708743, -1.76476678])
np.maximum(x, y)
>>> array([ 0.97121145,  1.98745872,  0.17706708,  1.47306221,  1.24527437,
            -0.75747493,  0.4994057 , -0.03462392])
```

numpy에서 두 개의 array의 각 성분에 적용되는 함수 정리

함수	설명
add	두 array에서 동일한 위치의 성분끼리 더한 값을 계산 (arr1 + arr2의 결과와 동일)
subtract	두 array에서 동일한 위치의 성분끼리 뺀 값을 계산 (arr1 - arr2의 결과와 동일)
multiply	두 array에서 동일한 위치의 성분끼리 곱한 값을 계산 (arr1 * arr2의 결과와 동일)
divide	두 array에서 동일한 위치의 성분끼리 나눈 값을 계산 (arr1 / arr2의 결과와 동일)
maximum	두 array에서 동일한 위치의 성분끼리 비교하여 둘 중 최댓값을 반환
minimum	두 array에서 동일한 위치의 성분끼리 비교하여 둘 중 최솟값을 반환

통계 함수

numpy에서는 array 내 전체 성분에 대한 통계량을 계산하는 함수 또한 제공합니다. 예를 들어 arr array가 정의되어 있을 때, `.sum()` 함수를 적용하면 arr array 내 모든 성분의 합을 계산하며, `.mean()` 함수를 적용하면 arr array 내 전체 성분에 대한 평균값을 계산합니다.

array에 대하여 합이나 평균 등을 계산할 때, 전체 성분 대신 '행 방향' 혹은 '열 방향'의 성분들에 대해서만 합 혹은 평균 등을 계산하도록 할 수 있습니다. 2차원 array에 대하여 `.sum(axis=0)` 을 실행하면 '행 방향'으로 성분들의 합을 구하게 되어, 결과적으로 각 열의 합을 계산한 결과를 산출합니다. 반면 `.sum(axis=1)` 을 실행하면 '열 방향'으로 성분들의 합을 구하게 되어, 곧 각 행의 합을 계산한 결과를 산출합니다.

numpy의 array에 적용되는 통계 함수 정리

함수	설명
sum	전체 성분의 합을 계산
mean	전체 성분의 평균을 계산
std, var	전체 성분의 표준편차, 분산을 계산
min, max	전체 성분의 최솟값, 최댓값을 계산
argmin, argmax	전체 성분의 최솟값, 최댓값이 위치한 인덱스를 반환
cumsum	맨 첫번째 성분부터 각 성분까지의 누적합을 계산 (0에서부터 계속 더해짐)
cumprod	맨 첫번째 성분부터 각 성분까지의 누적곱을 계산 (1에서부터 계속 곱해짐)

한편 이런 통계 함수들은, 특정 조건을 만족하는 array 내 성분의 개수가 몇 개인지 셀 때도 유용하게 사용될 수 있습니다. `arr` array에 대하여 `arr > 0` 을 실행하여 마스크를 얻으면, 해당 마스크에 대하여 `(arr > 0).sum()` 을 실행하여 마스크 내 True 값의 개수를 계산함으로써, 결과적으로 `arr > 0` 이라는 조건을 만족하는 성분의 총 개수를 산출할 수 있습니다.

정렬 함수 및 기타 함수

데이터 상에서 TOP 30 혹은 상위 5%에 해당하는 값이 무엇인지 추출해야 하는 경우가 자주 있는데, 정렬 함수는 이와 같은 작업을 위해 필수적인 함수라고 할 수 있습니다.

```
arr = array([-0.21082527, -0.0396508 , -0.75771892, -1.9260892 , -0.18137694,
            -0.44223898,  0.32745569,  0.16834256])
```

위와 같은 `arr` array가 정의되어 있다고 할 때, `np.sort()` 함수를 적용하면 `arr` array의 성분이 오름차순으로(크기가 커지는 순서대로) 정렬됩니다.

```
np.sort(arr)
>>> array([-0.21082527, -0.0396508 , -0.75771892, -1.9260892 , -0.18137694,
            -0.44223898,  0.32745569,  0.16834256])
```

만약 위의 경우에서 내림차순으로 정렬하고 싶다면, `np.sort(arr)[::-1]` 을 실행하면 됩니다. 이는 numpy 인덱싱 시 사용되는 일종의 꼼수이니 잘 기억해 두시면 유용할 것입니다.

정렬 함수는 2차원 array에도 적용할 수 있습니다.

```
arr2d = np.array([[ -1.25627232,  1.65117477, -0.04868035],
                  [  0.7405744 , -0.67893699, -0.28428494],
                  [  0.02640821, -0.29027297,  0.34441534],
                  [  0.68394722,  0.26180229,  0.76742614],
                  [  1.00806827,  0.77977295, -1.36273314]])
```

2차원 `arr2d` array를 하나 정의하였을 때, `np.sort(arr, axis=0)` 을 실행하면 '행 방향'으로 오름차순 정렬이 이루어집니다. `axis` 인자의 값을 1로 바꾸면 '열 방향'으로 오름차순 정렬이 이루어질 것입니다.

```
np.sort(arr, axis=0)
>>> array([[ -1.25627232, -0.67893699, -1.36273314],
           [  0.02640821, -0.29027297, -0.28428494],
           [  0.68394722,  0.26180229, -0.04868035],
           [  0.7405744 ,  0.77977295,  0.34441534],
           [  1.00806827,  1.65117477,  0.76742614]])
```

만약 길이가 아주 긴 `large_arr` array가 정의되어 있을 때, 해당 array 상에서 상위 5%에 위치하는 값은 다음과 같이 찾아낼 수 있습니다.

```
np.sort(large_arr)[: -1][int(0.05 * len(large_arr))]
```

`large_arr` 을 먼저 오름차순으로 정렬한 뒤, `int(0.05 * len(large_arr))` 을 통해 상위 5%에 해당하는 인덱스를 계산한 후, 이를 사용하여 정렬 결과에 대한 인덱싱을 수행하는 과정으로 진행하였습니다.

마지막으로, 중복된 성분을 포함하고 있는 array에 대하여, `np.unique()` 함수를 사용하면 중복된 값을 제외한 유니크한 값만을 추출할 수 있습니다. 예를 들어 해당 array의 이름이 `names` 였다면, `np.unique(names)` 을 실행하면 중복된 성분이 제거된 새로운 array를 얻을 수 있습니다.

numpy를 사용한 데이터 분석 맛보기

파일 읽기

[\[MovieLens 1M 데이터셋 다운로드\]](#)

대부분의 데이터셋은 열과 열을 구분하기 위한 구분자로 특정한 문자를 사용합니다. 예를 들어 MovieLens 1M dataset의 경우, 콜론이 두 개 붙은 문자 '::'가 구분자라고 할 수 있습니다.

구분자는 보통 콤마 ','를 많이 쓰는데, 콤마를 구분자로 사용한 파일을 특별히 '**CSV(comma-separated values) 파일**'이라고 부릅니다.

일정한 구분자를 사용하는 데이터셋 파일은 `np.loadtxt()` 함수를 사용하여 손쉽게 읽어들이 수 있습니다. 해당 함수를 호출할 때, 읽어들이 데이터셋 파일의 경로, 사용할 구분자, array의 데이터 형 등이 인자로 입력됩니다.

```
data = np.loadtxt("data/movielens-1m/ratings.dat", delimiter="::", dtype=np.int64)
```

MovieLens 1M dataset 분석하기

동영상 강의의 내용을 참조해 주시길 바랍니다.

파일 쓰기

여러분이 얻은 array 형태의 데이터 분석 결과물을 외부 파일로 쓰는 작업을 진행해보도록 합시다.

`np.savetxt()` 함수를 사용하면, 현재 **array** 형태로 들고 있는 분석 결과물을 외부 파일로 손쉽게 쓸 수 있습니다. 해당 함수를 호출할 때, 저장할 파일의 경로, 저장할 **array**, 저장 형태(format), 구분자 등을 인자로 같이 입력합니다.

```
np.savetxt("mean_rating_by_user.csv", mean_rating_by_user_array,  
           fmt='%.3f', delimiter=',')
```

pandas의 고유한 자료구조 - Series와 DataFrame 이해하기

pandas에서는 고유하게 정의한 자료 구조인 Series와 DataFrame을 사용하여, 빅 데이터 분석에 있어 높은 수준의 성능을 발휘합니다. Series는 동일한 데이터형의 복수 개의 성분으로 구성된 자료 구조이며, DataFrame은 서로 같거나 다른 데이터형의 여러 개의 열에 대하여 복수 개의 성분으로 구성된 '표와 같은 형태'의 자료 구조입니다.

Series

Series는 `pd.Series()` 함수를 사용하여 정의합니다. 이 때 Python 리스트나, 혹은 numpy array 등이 함수의 인자로 입력됩니다.

Series는 각 성분의 인덱스와, 이에 대응되는 값으로 구성되어 있습니다. Series 생성 시 인덱스를 따로 명시해주지 않으면, 0으로 시작하는 정수 형태의 기본 인덱스가 부여됩니다. 기본 인덱스 대신 Series 생성 시 각 성분에 대한 인덱스를 사용자가 직접 명시할 수도 있습니다.

```
obj = pd.Series([4, 7, -5, 3])
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

어떤 Series `obj`의 인덱스만을 추출하고 싶을 경우 `obj.index`를, 값만을 추출하고 싶을 경우 `obj.values`를 사용하면 됩니다. 그리고 `obj.dtype`를 사용하면, 해당 Series에 부여된 데이터형을 확인할 수 있습니다.

Series 자체의 이름과 Series의 인덱스에 대한 이름을 지정해줄 수 있습니다. 각각 `obj.name`과 `obj.index.name`에 값을 대입해 주면 됩니다.

DataFrame

DataFrame은 `pd.DataFrame()` 함수를 사용하여 정의합니다. DataFrame에 입력할 데이터는 Python 딕셔너리 혹은 numpy의 2차원 array 등의 형태가 될 수 있습니다.

```
data = {"names": ["Kilho", "Kilho", "Kilho", "Charles", "Charles"],
        "year": [2014, 2015, 2016, 2015, 2016],
        "points": [1.5, 1.7, 3.6, 2.4, 2.9]}
df = pd.DataFrame(data)
```

DataFrame에서는 서로 다른 두 종류의 인덱스가 각각 행 방향과 열 방향에 부여되어 있으며, 이들이 교차하는 지점에 실제 값이 위치해 있는 것을 확인할 수 있습니다. 이렇게 DataFrame에는 두 종류의 인덱스가 있으며, 이들을 구분하기 위해 특별히 행 방향의 인덱스를 '인덱스', 열 방향의 인덱스를 '컬럼'이라고 부릅니다. DataFrame `df`에 대하여, `df.index`와 `df.columns`를 사용하여 인덱스와 컬럼을 각각 확인할 수 있습니다. 그리고 `df.values`를 사용하면 DataFrame의 값에 해당하는 부분만 2차원 array의 형태로 얻을 수 있습니다.

DataFrame의 인덱스와 컬럼에도 Series와 유사한 방식으로 이름을 지정해줄 수 있습니다. 각각 `df.index.name`과 `df.columns.name`에 값을 대입해 주면 됩니다.

* DataFrame 관련 함수 혹은 변수 등을 조회하고자 할 경우

어떤 특정한 DataFrame `df`와 관련하여 실행할 수 있는 함수 혹은 변수의 정확한 이름이 제대로 기억나지 않을 경우, IPython Notebook에서 `df.`까지만 입력하고 **TAB** 키를 누르면 사용 가능한 모든 변수 및 함수가 표시됩니다.

* NaN (not a number)

NaN은 numpy나 python에서 'not a number'를 표시하는 약어로, 특정 위치에 값이 존재하지 않을 때 이를 표시하기 위한 기호라고 할 수 있습니다. 여러분이 데이터셋 파일을 DataFrame의 형태로 읽어들이는 과정에서, 만약 특정 부분의 값이 포함되어 있지 않았을 경우 해당 위치에 이렇게 NaN으로 표시됩니다.

NaN으로 표시된 부분에 대해서는 추후에 어떠한 연산도 수행할 수 없기 때문에, 추후에 이 부분을 실제 값으로 메꿀 방법을 생각해 보아야 합니다.

describe 함수

DataFrame `df`에 대하여 `df.describe()` 함수를 실행하게 되면, 계산이 가능한 컬럼에 한해서 각 컬럼의 평균, 분산, 최솟/최댓값 등 기본 통계량을 산출한 결과를 보여줍니다. 이는 데이터셋을 DataFrame 형태로 막 읽어들이는 직후에, 데이터셋을 전체적으로 살펴보고자 할 때 사용하기 유용합니다.

DataFrame 인덱싱 이해하기

기본 인덱싱

`df`라는 이름의 DataFrame을 다음과 같이 정의합니다.

```
data = {"names": ["Kilho", "Kilho", "Kilho", "Charles", "Charles"],
        "year": [2014, 2015, 2016, 2015, 2016],
        "points": [1.5, 1.7, 3.6, 2.4, 2.9]}
df = pd.DataFrame(data, columns=["year", "names", "points", "penalty"],
                  index=["one", "two", "three", "four", "five"])
```

열 선택하고 조작하기

df 에서 'year' 열만을 얻고 싶을 경우, `df["year"]` 를 실행합니다. 그러면 'year' 열의 값들이 Series의 형태로 인덱스와 함께 표시됩니다. 하나의 열을 가져오는 경우 `df.year` 와 같이 해도 동일한 결과를 얻습니다. 두 가지 방법 중에서 여러분이 편한 것을 사용하시면 됩니다.

만약 복수 개의 열을 가져오고자 할 경우에는, `df[["year", "points"]]` 와 같은 형태로 실행하시면 됩니다. 이 때, 중괄호 안에 컬럼 이름으로 구성된 리스트가 들어간다는 점을 주의하시길 바랍니다.

특정 열을 이렇게 선택한 뒤, 값을 일괄적으로 대입해줄 수도 있습니다. `df["penalty"] = 0.5` 또는 `df["penalty"] = [0.1, 0.2, 0.3, 0.4, 0.5]` 를 실행하면, 기존에 NaN으로 표시되었던 'penalty' 열에 지정한 값을 대입하게 됩니다. 새로운 'zeros'라는 이름의 열을 추가하고자 할 경우, `df["zeros"] = np.arange(5)` 와 같이 실행하시면 됩니다.

pandas의 Series를 DataFrame의 새로운 열의 값으로 대입할 수도 있는데, 이 경우 해당 Series에서 명시된 인덱스와 DataFrame 상의 인덱스를 비교하여 대응되는 인덱스의 값을 대입하는 형태로 이루어집니다. 이것이 열의 값을 Python 리스트나 numpy array로 입력하는 경우와, pandas Series로 입력하는 경우의 핵심적인 차이입니다.

```
val = pd.Series([-1.2, -1.5, -1.7], index=["two", "four", "five"])
df["debt"] = val
```

새로운 열의 값을 기존 열의 값을 사용하여 입력할 수도 있습니다. 예를 들어 'net_points'라는 열의 값이 'points'에서 'penalty'를 뺀 것이라고 합시다. 그리고 'high_points' 열은 'net_points'의 값이 2.0보다 클 때 True, 그렇지 않을 때 False를 가지도록 합시다. 다음과 같이 실행하면 됩니다.

```
df["net_points"] = df["points"] - df["penalty"]
df["high_points"] = df["net_points"] > 2.0
```

기존의 열을 DataFrame 상에서 삭제해 버리는 것은 아주 간단합니다. `del` 키워드와 함께 삭제할 열을 명시하면 됩니다.

```
del high_points
del net_points
del zeros
```

행 선택하고 조작하기

pandas DataFrame의 행을 인덱싱할 수 있는 방법은 무수하게 많습니다. 그리고 열을 인덱싱하는 방법에 있어서도 바로 위에서 소개했던 방법은 여러 가지 중 하나에 불과합니다. 더욱 머리를 아프게 하는 것은, pandas의 버전이 올라가면서 이러한 인덱싱 방법도 미묘하게 달라지고 있다는 것입니다.

다른 기능들과의 혼동을 방지하기 위해, DataFrame의 행을 선택하고자 할 때, 저는 `.loc` 이나 `.iloc` 을 사용하시길 권장드리고자 합니다.

df.loc

`.loc` 은, 실제 인덱스를 사용하여 행을 가져올 때 사용합니다. 예를 들어 `df.loc["two"]` 를 실행하면, 인덱스가 'two'인 행의 값을 모두 가져옵니다. `df.loc["two":"four"]` 을 실행하면, 인덱스가 'two'부터 'four'까지의 범위에 있는 행의 모든 값을 가져옵니다.

`.loc` 을 사용하면, 인덱스 뿐만 아니라 컬럼 역시 동시에 명시할 수 있습니다. 예를 들어 'two'부터 'four'까지 인덱스의 값들 중 'points' 열의 값만을 가져오고자 할 경우, `df.loc["two":"four", "points"]` 를 실행하면 됩니다. 이 경우에도 하나의 열을 가져왔으므로 결과물은 Series가 됩니다. `df.loc["three":"five", "year":"penalty"]` 와 같이 실행하면, 인덱스와 컬럼 모두 범위 인덱싱이 가능합니다.

새로운 행을 추가하고자 할 경우, `df.loc["six", :] = [2013, "Hayoung", 4.0, 0.1, 2.1]` 과 같이 `.loc` 을 사용하여 새로운 인덱스의 행을 선택하고, 여기에 대입할 값을 리스트 혹은 array의 형태로 입력해주면 됩니다.

df.iloc

`.iloc` 은, numpy의 array 인덱싱 방식으로 행을 가져올 때 사용합니다. 예를 들어 `df.iloc[3]` 을 실행하면, 인덱스 3에 해당하는 4행을 모두 가져옵니다. `df.iloc[3:5, 0:2]` 와 같이 행과 열에 대한 범위 인덱싱이 가능하며, `df.iloc[[0, 1, 3], [1, 2]]` 와 같이 원하는 인덱스만을 명시하여 가져올 수도 있습니다.

DataFrame의 행을 선택하는 방법은 위에서 소개해 드린 방법 외에도 몇 가지가 더 있습니다. 그럼에도 불구하고, 여러분들께 `.loc` 과 `.iloc` 만을 사용하여 행을 선택하고 조작하시길 권장드립니다. 그래야 나중에 혼란이 적을 것입니다.

불리언 인덱싱

pandas의 DataFrame에서도 열이나 행을 선택할 때, 불리언 인덱싱이 가능합니다.

`df` 에서, 'year' 열의 값이 2014보다 큰 행만을 선택하고 싶다고 가정합니다. `df["year"] > 2014` 를 실행하면, 값이 2014보다 큰 위치에는 True, 그 외에는 False가 들어가 있는 불리언 Series를 얻게 됩니다. 이를 **마스크**라고 부릅니다.

이런 마스크는 DataFrame을 인덱싱하는 데 사용될 수 있습니다. `df.loc[df["year"] > 2014, :]` 를 실행하면, 앞서 확인했던 마스크가 True에 해당하는 행만으로 구성된 DataFrame을 얻을 수 있습니다. `df.loc[df["names"] == "Kilho", ["names", "points"]]` 를 실행하면, 'names' 열의 값이 "Kilho"인 행의 값 중에서 'names'와 'points' 열에 해당하는 값들만을 얻을 수 있습니다.

만약 사용할 조건, 즉 마스크가 여러 개 필요하다면, `df.loc[(df["points"] > 2) & (df["points"] < 3), :]` 와 같이 실행하면 됩니다. 조건과 조건 간에 '&'가 들어가 있는 것을 주목하시길 바랍니다. '&'(and) 연산자를 사용하는 경우 두 개의 마스크가 동시에 True인 경우에만 해당 인덱스를 조회하며, '|' (or) 연산자를 사용하는 경우 두 개의 마스크 중 최소 하나가 True인 경우에 해당 인덱스를 조회합니다.

DataFrame 이리저리 조작하기

결측값 또는 이상치 처리하기

여러분이 pandas를 사용하여 읽어들이는 데이터셋 파일에 NaN의 형태로 빠진 값이 존재하거나, 혹은 정상 범주에서 벗어난 값이 포함되어 있는 경우가 얼마든지 발생할 수 있습니다. 이러한 값들을 각각 **결측값(missing value)** 및 **이상치(outlier)**라고 하는데, 본격적인 데이터 분석에 앞서 이들을 적절히 처리할 필요가 있습니다.

```
df = pd.DataFrame(np.random.randn(6, 4))
df.columns = ["A", "B", "C", "D"]
df.index = pd.date_range("20160701", periods=6)
```

랜덤한 숫자들로 구성된 DataFrame `df` 를 정의하고, 인덱스와 컬럼을 명시하였습니다. 이 때, pandas에서 제공하는 `pd.date_range()` 함수를 사용하였는데, 이는 일자와 시각을 나타내는 데이터형인 **datetime**으로 구성된 인덱스를 생성할 때 사용하는 함수입니다. 여러분들이 많이 접할 데이터셋 중 하나가 바로 시계열(time series) 형태의 데이터셋인데, pandas는 이렇게 datetime과 관련된 간편한 기능을 많이 제공하고 있습니다.

`df` 의 'F' 열을 새로 정의하고, 여기에 다음과 같이 값을 대입합니다.

```
df["F"] = [1.0, np.nan, 3.5, 6.1, np.nan, 7.0]
```

NaN을 인위적으로 사용하고자 할 때, numpy에서 제공하는 `np.nan` 을 사용합니다.

`df.dropna(how="any")` 를 실행하면, 각각의 행을 살펴보면서 행 내 값들 중에 NaN이 하나라도 포함되어 있는 경우 해당 행을 DataFrame 상에서 삭제합니다. 반면 `df.dropna(how="all")` 을 실행할 경우, 행 내 값들 모두가 NaN인 경우 해당 행을 DataFrame 상에서 삭제합니다.

아니면 `df` 의 NaN을 실제 수치값으로 대체하고 싶을 수 있습니다. 이러한 경우

`df.fillna(value=5.0)` 와 같이 실행하면, NaN을 지정한 값으로 대체한 결과물을 얻을 수 있습니다.

한편, `df.isnull()` 함수를 실행하면 DataFrame 상에서 NaN이 포함되어 있는 위치에 대한 불리언 마스크를 얻을 수 있습니다. 이를 응용하여 `df.loc[df.isnull()["F"], :]` 를 실행하면, 'F'열에 NaN을 포함하고 있는 행만을 선택할 수 있습니다.

다음으로 각 행의 값을 관찰하여, 이상치가 포함되어 있는 행을 DataFrame 상에서 직접 삭제하는 방법을 알아보도록 하겠습니다. 예를 들어 2016년 7월 1일 행 데이터를 선택하고자 할 때, `pd.to_datetime("20160701")` 을 실행하면 "20160701"이라는 문자열을 datetime 데이터형으로 변환하며, 이를 인덱스로 사용할 수 있습니다.

`df.drop(pd.to_datetime("20160701"))` 을 실행한 결과를 보면, 2016년 7월 1일 행 데이터가 DataFrame 상에서 삭제된 것을 확인할 수 있습니다. 이렇게 인덱스를 기준으로 특정 행을 삭제하고자 할 경우 `.drop()` 함수를 사용하면 됩니다. 두 개 이상의 인덱스를 기준으로 복수 개의 행을 삭제하고자 할 경우, `df.drop([pd.to_datetime("20160702"), pd.to_datetime("20160704")])` 와 같이 `.drop()` 함수의 매개변수로 datetime 리스트를 입력하면 됩니다.

`.drop()` 함수를 사용하면, 행 뿐만 아니라 열도 삭제할 수 있습니다. `df.drop("F", axis=1)` 와 같이 삭제할 컬럼을 명시한 후 `axis=1` 매개변수를 명시해 주면, 'F' 열이 삭제된 것을 확인할 수 있습니다. 이 경우에도 `df.drop(["B", "F"], axis=1)` 의 형태로 복수 개의 열을 명시할 수도 있습니다.

DataFrame 데이터 분석용 함수 사용하기

통계 함수

pandas의 DataFrame에서는 다양한 통계 함수를 지원합니다. DataFrame을 사용하여 데이터 분석을 할 때 가장 중요한 기능이라고 할 수 있겠습니다.

```
data = [[1.4, np.nan],
        [7.1, -4.5],
        [np.nan, np.nan],
        [0.75, -1.3]]
df = pd.DataFrame(data, columns=["one", "two"], index=["a", "b", "c", "d"])
```

`df` 라는 `DataFrame`을 정의하여, '행 방향' 또는 '열 방향'의 합을 구하도록 할 수 있습니다.

`df.sum(axis=0)` 을 실행하면 '행 방향'으로 성분들의 합을 구하게 되며, `df.sum(axis=1)` 을 실행하면 '열 방향'으로 성분들의 합을 구하게 됩니다. 이렇게 별다른 옵션 지정 없이 `.sum()` 함수를 그냥 호출할 경우, 기본적으로 `NaN`은 배제하고 합을 계산합니다.

특정 행 또는 특정 열에 대해서만 합을 계산할 수도 있습니다. `df["one"].sum()` 을 실행하면 'one' 열의 합만을 얻을 수 있으며, `df.loc["b"].sum()` 을 실행하면 'b' 행의 합만을 얻을 수 있습니다.

평균이나 분산 등의 경우 사용하는 함수만 달라질 뿐, 방식은 완전히 동일합니다. 각각 `.mean()` 과 `.var()` 함수를 사용하면 됩니다.

만약 `NaN`을 모두 감안해서 통계량을 산출하고 싶다면, 통계 함수에 `skipna=False` 라는 인자를 추가로 입력하면 됩니다. 예를 들어 `df.mean(axis=1, skipna=False)` 를 실행하고 그 결과를 관찰해 보면, `NaN`이 포함된 행은 평균값이 제대로 계산되지 않고 `NaN`이라고 표시되어 있는 것을 확인할 수 있습니다.

pandas의 DataFrame에 적용되는 통계 함수 정리

함수	설명
count	전체 성분의 (<code>NaN</code> 이 아닌) 값의 갯수를 계산
min, max	전체 성분의 최솟, 최댓값을 계산
argmin, argmax	전체 성분의 최솟값, 최댓값이 위치한 (정수)인덱스를 반환
idxmin, idxmax	전체 인덱스 중 최솟값, 최댓값을 반환
quantile	전체 성분의 특정 사분위수에 해당하는 값을 반환 (0~1 사이)
sum	전체 성분의 합을 계산
mean	전체 성분의 평균을 계산
median	전체 성분의 중간값을 반환
mad	전체 성분의 평균값으로부터의 절대 편차(absolute deviation)의 평균을 계산
std, var	전체 성분의 표준편차, 분산을 계산
cumsum	맨 첫 번째 성분부터 각 성분까지의 누적합을 계산 (0에서부터 계속 더해짐)
cumprod	맨 첫 번째 성분부터 각 성분까지의 누적곱을 계산 (1에서부터 계속 곱해짐)

이들 통계 함수를 사용하여 `DataFrame`의 결측값을 채울 수도 있습니다. 예를 들어 'one' 열의 `NaN`은 나머지 값들의 평균값으로, 'two' 열의 `NaN`은 해당 열의 최솟값으로 대체하고자 한다고 가정합시다. 다음과 같이 실행하면, 'one' 열과 'two' 열의 결측값들이 각각 의도했던 대로 채워질 것입니다.


```
one_mean = df.mean(axis=0)["one"]
two_min = df.min(axis=0)["two"]
df["one"] = df["one"].fillna(value=one_mean)
df["two"] = df["two"].fillna(value=two_min)
```

DataFrame에서 특히 유용한 통계 함수 중 하나가, 상관 계수(correlation coefficient) 혹은 공분산(covariance)을 계산하는 함수입니다. 새로운 `df2` DataFrame을 정의합니다.

```
df2 = pd.DataFrame(np.random.randn(6, 4),
                    columns=["A", "B", "C", "D"],
                    index=pd.date_range("20160701", periods=6))
```

`df2` 에서 'A' 열의 데이터와 'B' 열의 데이터 간의 상관계수를 산출하고자 할 경우, `df2["A"].corr(df2["B"])` 을 실행하면 됩니다. 같은 방식으로, 'B'열 데이터와 'C'열 데이터 간의 공분산을 산출하고자 할 경우 `df2["B"].cov(df2["C"])` 을 실행하면 됩니다.

혹은 특정한 두 개의 열을 명시할 필요 없이, `df2.corr()` 또는 `df2.cov()` 를 실행하면 `df2` 의 모든 열들 간의 상관계수 및 공분산을 한 번에 계산해 줍니다. 데이터 분석 시 어느 한 변수가 다른 변수에 미치는 영향 등을 알고자 할 때, 이러한 상관 분석이 대단히 유용할 수 있습니다.

정렬 함수 및 기타 함수

정렬 함수

pandas의 DataFrame에서는 인덱스 기준 정렬과 값 기준 정렬을 지원합니다. 기존 `df2` 의 인덱스와 컬럼 순서를 무작위로 섞어보도록 하겠습니다.

```
dates = df2.index
random_dates = np.random.permutation(dates)
df2 = df2.reindex(index=random_dates, columns=["D", "B", "C", "A"])
```

`df2` 를 확인하면, 인덱스와 컬럼의 순서가 불규칙하게 변화한 것을 확인할 수 있습니다. 실제로 시계열 데이터같은 경우에도 이렇게 시간에 따라 제대로 정렬되어 있지 않은 데이터셋을 얻을 가능성이 존재합니다.

이 때 인덱스가 오름차순이 되도록 행들을 정렬하고 싶은 경우, `df2.sort_index(axis=0)` 을 실행하여 '행 방향'으로의 정렬을 수행합니다. 만약 컬럼이 오름차순이 되도록 열들을 정렬하고 싶은 경우, `df2.sort_index(axis=1)` 을 실행하여 '열 방향'으로의 정렬을 수행합니다.

만약 인덱스가 내림차순이 되도록 행들을 정렬하고 싶은 경우, `df2.sort_index(axis=0, ascending=False)` 와 같이 `ascending=False` 인자를 추가로 입력해 주면 됩니다.

다음으로 값 기준 정렬입니다. 예를 들어 'D'열의 값이 오름차순이 되도록 행들을 정렬하고 싶은 경우, `df2.sort_values(by="D")` 를 실행합니다. 만약 'B'열의 값이 내림차순이 되도록 행들을 정렬하고 싶은 경우, `df2.sort_values(by="B", ascending=False)` 를 실행하면 됩니다.

```
df2["E"] = np.random.randint(0, 6, size=6)
df2["F"] = ["alpha", "beta", "gamma", "gamma", "alpha", "gamma"]
```

`df2` 에 'E'열과 'F'열을 추가한 뒤, 'E' 열과 'F' 열의 값을 동시에 기준으로 삼아 각각 오름차순이 되도록 행들을 정렬하고 싶은 경우, `df2.sort_values(by=["E", "F"])` 와 같이 여러 개의 컬럼을 리스트 형태로 명시해 주면 됩니다.

기타 함수

`df.unique()` 함수는 지정한 행 또는 열에서 중복을 제거한 유니크한 값들만을 제시하는 함수입니다.

```
uniques = df2["F"].unique()
```

`df.value_counts()` 함수는 특정한 행 또는 열에서 값에 따른 갯수를 제시하는 함수입니다.

```
df2["F"].value_counts()
```

`df.isin()` 함수를 사용하면, 특정한 행 또는 열의 각 성분이 특정한 값들을 포함되어 있는지 확인할 수 있으며, 불리언 마스크를 반환합니다. 그래서 이렇게 얻은 불리언 마스크를 사용하여, 원하는 행만을 선택할 수도 있습니다.

```
df2["F"].isin(["alpha", "beta"])
df2.loc[df2["F"].isin(["alpha", "beta"]), :]
```

사용자 정의 함수

사용자가 직접 정의한 함수를 `DataFrame`의 행 또는 열에 적용하는 방법에 대해 알아보도록 하겠습니다. 새로운 `df3` `DataFrame`을 먼저 정의합니다.

```
df3 = pd.DataFrame(np.random.randn(4, 3), columns=["b", "d", "e"],
                    index=["Seoul", "Incheon", "Busan", "Daegu"])
```

`lambda` 함수를 정의한 뒤, 이를 `func` 변수에 저장합니다. 이 때 `func` 함수는, 해당 행 또는 열에서의 최댓값 - 최솟값을 계산하는 함수입니다.

```
func = lambda x: x.max() - x.min()
```

`df.apply()` 함수를 사용하여 해당 함수 인자로 `func` 를 명시하면, 해당 `DataFrame`의 각 행 또는 열에 대하여 `func` 함수를 적용하고 그 결과값을 얻을 수 있습니다. 예를 들어

`df3.apply(func, axis=0)` 을 실행하면 이를 '행 방향'으로 진행한 결과를, `df3.apply(func, axis=1)` 을 실행하면 이를 '열 방향'으로 진행한 결과를 얻습니다.

pandas를 사용한 데이터 분석 맛보기

파일 읽기

[[Lending Club Loan 데이터셋 다운로드](#)]

본 강의에서는, Lending Club Loan dataset 2007-2015를 사용합니다.

대부분의 데이터셋은 열과 열을 구분하기 위한 구분자로 특정한 문자를 사용합니다. 구분자는 보통 콤마 ','를 많이 쓰는데, 콤마를 구분자로 사용한 파일을 특별히 '**CSV(comma-separated values) 파일**'이라고 부릅니다.

CSV 파일은 `pd.read_csv()` 함수를 사용하여 손쉽게 읽어들이 수 있습니다. 해당 함수를 호출할 때, 읽어들이 데이터 파일의 경로와 구분자 등을 입력합니다.

```
df = pd.read_csv("data/lending-club-loan-data/loan.csv", sep=",")
```

Lending Club Loan dataset 분석하기

Lending Club Loan dataset의 주요 컬럼 요약

- `loan_amnt`: 대출자의 대출 총액
- `funded_amnt`: 해당 대출을 위해 모금된 총액
- `issue_d`: 대출을 위한 기금이 모금된 월
- `loan_status`: 대출의 현재 상태*
- `title`: 대출자에 의해 제공된 대출 항목
- `purpose`: 대출자에 의해 제공된 대출 목적
- `emp_length`: 대출자의 재직 기간
- `grade`: LC assigned loan grade**
- `int_rate`: 대출 이자율
- `term`: 대출 상품의 기간 (36-month vs. 60-month)

* 불량 상태(bad status): "Charged Off", "Default", "Does not meet the credit policy.
Status:Charged Off", "In Grace Period", "Default Receiver", "Late (16-30 days)", "Late (31-120 days)"

** LC loan grade 참고: <https://www.lendingclub.com/public/rates-and-fees.action>

필요한 열 발췌 및 결측값 제거하기

```
df2 = df[["loan_amnt", "loan_status",
          "grade", "int_rate", "term"]]
df2 = df2.dropna(how="any")
```

'36개월 대출'과 '60개월 대출'의 대출 총액 파악

```
term_to_loan_amnt_dict = {}
uniq_terms = df2["term"].unique()
for term in uniq_terms:
    loan_amnt_sum = df2.loc[df2["term"] == term, "loan_amnt"].sum()
    term_to_loan_amnt_dict[term] = loan_amnt_sum
term_to_loan_amnt = pd.Series(term_to_loan_amnt_dict)
```

각 대출 상태(불량/우량)에 따른 대출 등급 분포 파악

```
total_status_category = df2["loan_status"].unique()
bad_status_category = total_status_category[[1, 3, 4, 5, 6, 8]]
df2["bad_loan_status"] = df2["loan_status"].isin(bad_status_category)
bad_loan_status_to_grades = \
    df2.loc[df2["bad_loan_status"] == True, "grade"].value_counts()
bad_loan_status_to_grades.sort_index()
```

대출 총액과 대출 이자율 간의 상관관계 파악

```
df2["loan_amnt"].corr(df2["int_rate"])
```

파일 쓰기

여러분이 얻을 `bad_loan_status_to_grades` Series를 외부 파일로 쓰는 작업을 진행해보도록 합시다. 이는 DataFrame에 대해서도 동일하게 적용됩니다.

`Series.to_csv()` 혹은 `df.to_csv()` 함수를 사용하면, 현재 `Series` 혹은 `DataFrame` 형태로 들고 있는 분석 결과물을 외부 파일로 손쉽게 쓸 수 있습니다. 해당 함수를 호출할 때, 저장할 파일의 경로와 구분자 등을 인자로 같이 입력합니다.

```
bad_loan_status_to_grades.to_csv("bad_loan_status.csv", sep=",")
```

matplotlib의 플롯팅 함수 사용하기

matplotlib은 numpy나 pandas를 사용하여 데이터를 분석한 결과를 시각화하는 데 사용되는 대표적인 Python 데이터 시각화 라이브러리입니다. matplotlib에서는 DataFrame 혹은 Series 형태의 데이터를 가지고 다양한 형태의 플롯을 만들어 주는 기능을 지원합니다.

IPython Notebook에서 플롯을 그리기에 앞서, `%matplotlib` 라는 매직 명령어를 사용해서 플롯팅 옵션을 먼저 지정해야 합니다. `%matplotlib nbagg` 를 실행하는 경우, 노트북 상에서 생성되는 플롯을 인터랙티브하게 조작할 수 있습니다. 한편 `%matplotlib inline` 을 실행하면, 노트북 상의 특정 셀에서 플롯을 일단 생성하면 이를 조작할 수 없습니다.

본 강의에서는 `%matplotlib nbagg` 을 실행하여 적용합니다.

```
%matplotlib nbagg
```

라인 플롯(line plot)

라인 플롯은 연속적인 직선으로 구성된 플롯입니다. 어떤 특정한 독립변수 X가 변화함에 따라 종속변수 Y가 어떻게 변화하는지를 나타내고자 할 때 라인 플롯을 사용합니다.

랜덤한 값들로 구성된 Series `s` 를 인덱스와 함께 생성한 뒤 `s.plot()` 을 실행하면, `s` 의 인덱스와 값을 사용하여 라인 플롯을 그려줍니다. 만약 여러분이 import한 `matplotlib.pyplot` 모듈 `plt` 를 사용하여 `plt.plot(s)` 를 실행하더라도 동일한 결과를 얻을 수 있습니다.

```
s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))
s.plot()
```

생성된 플롯에 대하여, 화면에 표시된 부분을 이동시키거나 특정 부분을 확대하며, 현재 보여진 플롯을 이미지 파일 형태로 내보내는 등의 인터랙티브한 조작을 가할 수 있습니다. 플롯 우측 상단의 파란색 버튼을 클릭하게 되면, 해당 플롯을 더 이상 수정할 수 없는 상태가 됩니다.

랜덤한 값들로 구성된 DataFrame `df` 을 인덱스, 컬럼과 함께 생성한 뒤 `df.plot()` 을 실행하면, `df` 의 인덱스와 각 컬럼 값을 사용하여 여러 개의 라인 플롯을 그려줍니다. `df` 에 포함되어 있는 열의 갯수만큼 라인 플롯이 화면에 그려진 것을 확인할 수 있습니다. 만약 여러분이 import한 `matplotlib.pyplot` 모듈 `plt` 를 사용하여 `plt.plot(df)` 를 실행하더라도 동일한 결과를 얻을 수 있습니다.

```
df = pd.DataFrame(np.random.randn(10, 4).cumsum(axis=0),
                  columns=["A", "B", "C", "D"],
                  index=np.arange(0, 100, 10))
df.plot()
```

만약 특정한 하나의 열에 대해서만 라인 플롯을 그리고 싶다면, 다음과 같이 해당 열을 Series 형태로 추출한 뒤 라인 플롯을 그리면 됩니다.

```
df["B"].plot()
```

바 플롯(bar plot)

바 플롯은 막대 형태의 플롯입니다. 독립변수 X가 변화하면서 종속변수 Y가 변화하는 양상을 나타낼 때, X가 연속적인 숫자에 해당하는 경우 라인 플롯을 그렸다면, X가 유한 개의 값만을 가질 경우 바 플롯을 사용하면 유용합니다.

랜덤한 값들로 구성된 Series s2 를 인덱스와 함께 생성한 뒤 s2.plot(kind="bar") 를 실행하면, s2 의 인덱스와 값을 사용하여 수직 방향의 바 플롯을 그려줍니다.

```
s2 = pd.Series(np.random.rand(16), index=list("abcdefghijklmnop"))
s2.plot(kind="bar")
```

만약 바 플롯을 수평 방향으로 그리고자 할 경우, s2.plot(kind="barh") 를 실행하면 됩니다.

```
s2.plot(kind="barh")
```

랜덤한 값들로 구성된 DataFrame df2 를 인덱스, 컬럼과 함께 생성한 뒤

df2.plot(kind="bar") 를 실행하면, df2 의 인덱스와 각 컬럼 값을 사용하여 여러 개의 바 플롯을 그려줍니다. 이 때, 하나의 인덱스에 대하여 이에 대응되는 복수 개의 열 값이 여러 개의 바 플롯으로 나타난 것을 확인할 수 있습니다.

```
df2 = pd.DataFrame(np.random.rand(6, 4),
                  index=["one", "two", "three", "four", "five", "six"],
                  columns=pd.Index(["A", "B", "C", "D"], name="Genus"))
df2.plot(kind="bar")
```

바 플롯을 그릴 때 stacked=True 인자를 넣어주면, 하나의 인덱스에 대한 각 열의 값을 한 줄로 쌓아 표시해줍니다. 이는 하나의 인덱스에 대응되는 각 열 값의 상대적 비율을 확인할 때 유용합니다.

```
df2.plot(kind="barh", stacked=True)
```

히스토그램(histogram)

히스토그램의 경우 어느 하나의 변수 X가 가질 수 있는 값의 구간을 여러 개 설정한 뒤, 각각의 구간에 속하는 갯수를 막대 형태로 나타낸 플롯입니다. Series로부터 히스토그램을 그릴 때는 인덱스를 따로 명시할 필요가 없으며, 그저 값들만 가지고 있으면 됩니다.

랜덤한 값들로 구성된 Series `s3` 를 생성한 뒤 `s3.hist()` 를 실행하면, `s3` 의 값을 사용하여 히스토그램을 그려줍니다.

```
s3 = pd.Series(np.random.normal(0, 1, size=200))
s3.hist()
```

각 구간에 속하는 값의 갯수를 카운팅할 때, 구간의 개수는 자동으로 10개로 설정되어 있습니다. 이 구간을 'bin(빈)'이라고 부릅니다. 여러분이 히스토그램을 그릴 때, 다음과 같이 bin의 갯수를 직접 설정할 수도 있습니다.

```
s3.hist(bins=50)
```

만약 `normed=True` 인자를 넣어주면, 각 bin에 속하는 갯수를 전체 갯수로 나눈 비율, 즉 정규화한(normalized) 값을 사용하여 히스토그램을 그립니다.

```
s3.hist(bins=100, normed=True)
```

산점도(scatter plot)

라인 플롯이나 바 플롯의 경우 어떤 독립변수 X가 변화함에 따라 종속변수 Y가 어떻게 변화하는지 나타내는 것이 목적이었다면, 산점도의 경우 이 보다는 서로 다른 두 개의 독립변수 X1, X2 간에 어떠한 관계가 있는지 알아보고자 할 때 일반적으로 많이 사용합니다. 즉 산점도는, 두 독립변수 X1과 X2의 값을 각각의 축으로 하여 2차원 평면 상에 점으로 나타낸 플롯입니다.

랜덤한 값들로 구성된 두 개의 array를 생성한 뒤, `np.concatenate()` 함수를 사용하여 이들을 열 방향으로 연결합니다.

```
x1 = np.random.normal(1, 1, size=(100, 1))
x2 = np.random.normal(-2, 4, size=(100, 1))
x = np.concatenate((x1, x2), axis=1)
```

이렇게 생성된 `x` array를 사용하여 새로운 DataFrame `df3` 를 생성하면, `df3` 에는 'x1'과 'x2'의 두 개의 열이 포함되어 있습니다. `plt.scatter(df3["x1"], df3["x2"])` 를 실행하면, 두 열 간의 값을 기준으로 산점도를 그립니다.


```
df3 = pd.DataFrame(X, columns=["x1", "x2"])
plt.scatter(df3["x1"], df3["x2"])
```

얻어진 산점도의 수평축에는 'x1'의 값, 수직축에는 'x2'의 값을 사용하여 해당하는 위치에 점을 찍어서 데이터를 표현합니다.

플롯 모양 변형하기

Figure, subplots 및 axes

matplotlib에서는 '**figure(피겨)**'라는 그림 단위를 사용하여, 이 안에 한 개 혹은 복수 개의 플롯을 그리고 관리할 수 있도록 하는 기능을 지원합니다. 이 때, figure 안에 들어가는 플롯 공간 하나를 '**subplot(서브플롯)**'이라고 부릅니다.

새로운 figure를 직접 생성하고자 할 경우, `plt.figure()` 함수를 사용합니다. `fig` 라는 이름의 figure에 subplot을 하나 추가하고 싶으면, `fig.add_subplot()` 함수를 실행하여 그 반환값을 새로운 변수로 받습니다.

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
```

`fig.add_subplot()` 함수에는 총 3개의 인자가 들어갑니다. 앞의 두 개는 해당 figure 안에서 subplot들을 몇 개의 행, 몇 개의 열로 배치할 것인지를 나타냅니다. 맨 마지막 인자는, 이렇게 지정한 subplot들의 배치 구조 상에서, 해당 subplot을 실제로 어느 위치에 배치할지를 나타내는 번호입니다.

`fig.add_subplot()` 함수의 반환값을 `ax1` 이라는 변수에서 받는데, 이는 해당 subplot에 그려진 빈 좌표 평면을 나타내는 변수입니다. matplotlib에서는 이 빈 좌표평면을 '**axes(엑시스)**'라고 부릅니다. figure 안의 subplot에 axes를 생성한 순간부터, 비로소 여기에 플롯을 그릴 수 있는 상태가 됩니다.

같은 방법으로 `fig.add_subplot()` 함수를 여러 번 실행하여, 각 subplot 위치별로 새로운 axes를 생성함으로써 플롯을 그릴 준비를 갖추 수 있습니다.

```
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
```

이렇게 생성된 각각의 subplot 내 axes들에 실제 플롯을 그려보도록 합시다. 만약 `plt.plot()` 함수를 실행하여 플롯을 그리는 경우, 현재 활성화되어 있는 figure의 맨 마지막 위치에 해당하는 subplot의 axes부터 차례대로 플롯이 그려지게 됩니다.

```
plt.plot(np.random.randn(50).cumsum())
```

반면 `ax1.hist()` 와 같이 특정 **axes**를 나타내는 변수 `ax1` 을 직접 지정하여 플롯을 그리는 경우, 해당 **axes**에 플롯을 그립니다.

```
ax1.hist(np.random.randn(100), bins=20)
ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))
```

figure와 **subplot**을 그릴 때, `plt.subplots()` 함수를 사용하면 좀 더 직관적으로 할 수 있습니다.

```
fig, axes = plt.subplots(2, 3)
```

예를 들어 위와 같이 실행하게 되면, `fig` **figure** 안에 총 6개의 **subplot**들을 2x3으로 배치하며, 각각의 내부에 **axes**를 생성합니다. 이 때 반환받은 `axes` 함수에는, **subplot**의 구조와 동일한 구조를 가지는 2x3 크기의 **array**가 들어가며, 각각의 성분이 곧 대응되는 위치의 **axes**가 되므로 이를 사용하여 원하는 위치에 플롯을 그릴 수 있습니다.

색상, 마킹 및 라인 스타일

라인 플롯의 경우, 라인 색상과 마킹 기호 및 라인 스타일 등을 지정할 수 있습니다.

`plt.plot()` 함수를 사용해서 라인 플롯을 그릴 때, `color`, `marker`, `linestyle` 인자의 값을 함께 입력하면 각각 라인 색상, 점을 마킹하는 기호, 라인 스타일을 지정할 수 있습니다.

```
plt.plot(np.random.randn(30), color="g", marker='o', linestyle="--")
```

이들 각각에 입력할 수 있는 값은 **matplotlib**에서 따로 정의되어 있습니다. 예를 들어

`color="g"` 면 녹색, `marker='o'` 면 O 모양의 마킹 기호, `linestyle="--"` 이면 점선 스타일을 적용하게 됩니다.

matplotlib에서 사용 가능한 주요 `color`, `marker`, `linestyle` 값을 본 강의노트 맨 하단에 정리해 놓았으니 참고하시길 바랍니다.

만약 여러분들이 코드를 길게 작성하기 귀찮은 경우, 라인 플롯의 색상, 마킹 및 라인 스타일을 나타내는 값들을 하나의 문자열로 붙여서 입력할 수도 있습니다.

```
plt.plot(np.random.randn(30), "k.-")
```

한편 바 플롯이나 히스토그램, 산점도 등에는 색상과 알파값 등을 지정할 수 있습니다. 다음 코드를 실행하여 결과를 관찰해 보시다.

```
fig, axes = plt.subplots(2, 1)
data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))
data.plot(kind="bar", ax=axes[0], color='k', alpha=0.7)
data.plot(kind="barh", ax=axes[1], color='g', alpha=0.3)
```

눈금, 레이블 및 범례 등

여러분이 그린 플롯의 눈금, 레이블, 범례 등을 수정할 수 있습니다. 우선 **figure**를 하나 만든 뒤, **subplot axes**를 하나 추가하고 여기에 랜덤한 값들의 누적합을 나타내는 라인 플롯을 하나 그립니다.

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(np.random.randn(1000).cumsum())
```

플롯의 수평축 혹은 수직축에 나타난 눈금을 matplotlib에서는 '**틱(tick)**'이라고 부릅니다. 특별히 수평축의 눈금은 '**xtick**', 수직축의 눈금은 '**ytick**'이라고 부릅니다. 수평축의 눈금을 다른 것으로 변경하고자 할 경우, `ax.set_xticks()` 함수를 사용합니다.

```
ticks = ax.set_xticks([0, 250, 500, 750, 1000])
```

`ax.set_xticklabels()` 함수를 사용하여, 수평축의 눈금을 숫자가 아닌 문자열 레이블로 대체할 수도 있습니다.

```
labels = ax.set_xticklabels(["one", "two", "three", "four", "five"],
                             rotation=30, fontsize="small")
```

만약 수직축의 눈금을 변경하고자 한다면, 수평축의 경우와 완전히 동일한 방식으로 `ax.set_yticks()` 등의 함수를 사용하면 됩니다.

axes의 제목을 입력하고자 할 경우, `ax.set_title()` 함수를 사용하면 됩니다. 만약 수평축과 수직축에 이름을 붙이고 싶다면, 각각 `ax.set_xlabel()`, `ax.set_ylabel()` 함수를 사용하면 됩니다.

```
ax.set_title("Random walk plot")
ax.set_xlabel("Stages")
ax.set_ylabel("Values")
```

만약 하나의 **axes**에 표시한 플롯의 개수가 많다면, 범례(legend)를 표시해야 할 필요가 있습니다. 먼저 새로운 **figure** 안에 **subplot axes**를 하나 생성합니다.

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
```

다음으로, 랜덤 워크 플롯을 `ax` axes에 3개 추가합니다. 이 때, `ax.plot()` 함수를 사용할 시 `label` 인자의 값을 함께 입력해 줍니다. 입력한 `label` 인자의 값이, 나중에 axes에 범례를 표시 할 때 각각의 이름으로 제시됩니다.

```
ax.plot(np.random.randn(1000).cumsum(), 'k', label="one")
ax.plot(np.random.randn(1000).cumsum(), "b--", label="two")
ax.plot(np.random.randn(1000).cumsum(), "r.", label="three")
```

`ax.legend()` 함수를 실행하면, axes 상에 범례가 표시됩니다. 이 때 `loc="best"` 인자를 넣어주 면, 현재 제시된 axes 상에서 최적의 위치에 범례를 자동으로 배치합니다.

```
ax.legend(loc="best")
```

현재 axes에 표시된 수평축 값의 범위와 수직축 값의 범위를 변경하고자 한다면, `ax.set_xlim()` 함수와 `ax.set_ylim()` 함수를 사용하면 됩니다.

```
ax.set_xlim([100, 900])
ax.set_ylim([-100, 100])
```

부록: matplotlib에서 사용 가능한 주요 color, marker, linestyle 값

matplotlib에서 사용 가능한 주요 color 값

값	색상
"b"	blue
"g"	green
"r"	red
"c"	cyan
"m"	magenta
"y"	yellow
"k"	black
"w"	white

matplotlib에서 사용 가능한 주요 marker 값

값	마킹
"."	point
","	pixel
"o"	circle
"v"	triangle_down
"^"	triangle_up
"<"	triangle_left
">"	triangle_right
"8"	octagon
"s"	square
"p"	pentagon
"*"	star
"h"	hexagon
"+"	plus
"x"	x
"D"	diamond

matplotlib에서 사용 가능한 주요 linestyle 값

값	라인 스타일
"_"	solid line
"--"	dashed line
"-."	dash-dotted line
".."	dotted line
"None"	draw nothing

matplotlib를 사용한 데이터 시각화 맛보기

Game of Thrones 데이터셋 분석하기

[\[Game of Thrones 데이터셋 다운로드\]](#)

Game of Thrones 데이터셋의 주요 컬럼 요약

battles.csv

- name: String variable. The name of the battle.
- year: Numeric variable. The year of the battle.
- battle_number: Numeric variable. A unique ID number for the battle.
- attacker_king: Categorical. The attacker's king. A slash indicators that the king charges over the course of the war. For example, "Joffrey/Tommen Baratheon" is coded as such because one king follows the other in the Iron Throne.
- defender_king: Categorical variable. The defender's king.
- attacker_1: String variable. Major house attacking.
- attacker_2: String variable. Major house attacking.
- attacker_3: String variable. Major house attacking.
- attacker_4: String variable. Major house attacking.
- defender_1: String variable. Major house defending.
- defender_2: String variable. Major house defending.
- defender_3: String variable. Major house defending.
- defender_4: String variable. Major house defending.
- attacker_outcome: Categorical variable. The outcome from the perspective of the attacker. Categories: win, loss, draw.
- battle_type: Categorical variable. A classification of the battle's primary type. Categories: *pitched_battle*: Armies meet in a location and fight. This is also the baseline category. *ambush*: A battle where stealth or subterfuge was the primary means of attack. *siege*: A prolonged of a fortified position. *razing*: An attack against an undefended position
- major_death: Binary variable. If there was a death of a major figure during the battle.
- major_capture: Binary variable. If there was the capture of the major figure during the battle.
- attacker_size: Numeric variable. The size of the attacker's force. No distinction is made between the types of soldiers such as cavalry and footmen.
- defender_size: Numeric variable. The size of the defenders's force. No distinction is made between the types of soldiers such as cavalry and footmen.
- attacker_commander: String variable. Major commanders of the attackers. Commander's names are included without honoric titles and commandders are seperated by commas.
- defender_commander: String variable. Major commanders of the defener. Commander's names are included without honoric titles and commandders are seperated by commas.
- summer: Binary variable. Was it summer?
- location: String variable. The location of the battle.
- region: Categorical variable. The region where the battle takes place. Categories: Beyond the Wall, The North, The Iron Islands, The Riverlands, The Vale of Arryn, The

Westerlands, The Crownlands, The Reach, The Stormlands, Dorne

- note: String variable. Coding notes regarding individual observations.

character-deaths.csv

- Name: character name
- Allegiances: character house
- Death Year: year character died
- Book of Death: book character died in
- Death Chapter: chapter character died in
- Book Intro Chapter: chapter character was introduced in
- Gender: 1 is male, 0 is female
- Nobility: 1 is nobel, 0 is a commoner
- GoT: Appeared in first book
- CoK: Appeared in second book
- SoS: Appeared in third book
- FfC: Appeared in fourth book
- DwD: Appeared in fifth book

* 참고: <https://www.kaggle.com/mylesoneill/game-of-thrones>

작품 번호에 따른 인물들의 죽음 횟수 시각화하기 - 라인 플롯

```
book_nums_to_death_count = deaths["Book of Death"].value_counts().sort_index()
ax1 = book_nums_to_death_count.plot(color="k", marker="o", linestyle="--")
ax1.set_xticks(np.arange(1, 6))
ax1.set_xlim([0, 6])
ax1.set_ylim([0, 120])
```

대규모 전투 상에서 공격군과 수비군 간의 병력 차이 시각화하기 - 박스 플롯

```
battles = battles.set_index(["name"])
large_battles_mask = battles["attacker_size"] + battles["defender_size"] > 10000
large_battles = battles.loc[large_battles_mask, ["attacker_size", "defender_size"]]
ax2 = large_battles.plot(kind="barh", stacked=True, fontsize=8)
```

```

large_battles["attacker_pcts"] = \
    large_battles["attacker_size"] / (large_battles["attacker_size"] + large_battles["
defender_size"])
large_battles["defender_pcts"] = \
    large_battles["defender_size"] / (large_battles["attacker_size"] + large_battles["
defender_size"])
ax3 = large_battles[["attacker_pcts", "defender_pcts"]].plot(kind="barh", stacked=
True, fontsize=8)

```

전체 전투 중 각 가문의 개입 빈도 시각화하기 - 히스토그램

```

col_names = battles.columns[4:12]
house_names = battles[col_names].fillna("None").values
house_names = np.unique(house_names)
house_names = house_names[house_names != "None"]
houses_to_battle_counts = pd.Series(0, index=house_names)

```

```

for col in col_names:
    houses_to_battle_counts = \
        houses_to_battle_counts.add(battles[col].value_counts(), fill_value=0)
ax4 = houses_to_battle_counts.hist(bins=10)

```


난수 (random)

파이썬에서 난수(random number)를 사용하기 위해서는 기본적으로 제공되는 random 모듈을 사용한다. random 모듈에서 자주 사용되는 함수는 다음과 같다.

- randint(최소, 최대) : 입력 파라미터인 최소부터 최대까지 중 임의의 정수를 리턴한다
- random() : 0 부터 1 사이의 부동소수점(float) 숫자를 리턴한다
- uniform(최소, 최대) : 입력 파라미터인 최소부터 최대까지 중 임의의 부동소수점(float) 숫자를 리턴한다
- randrange(시작,끝[,간격]) : 입력 파라미터인 시작부터 끝값까지 (지정된 간격으로 나열된) 숫자 중 임의의 정수를 리턴한다

```
from random import *

i = randint(1, 100) # 1부터 100 사이의 임의의 정수
print(i)

f = random()      # 0부터 1 사이의 임의의 float
print(f)

f = uniform(1.0, 36.5) # 1부터 36.5 사이의 임의의 float
print(f)

i = randrange(1, 101, 2) # 1부터 100 사이의 임의의 짝수
print(i)

i = randrange(10) # 0부터 9 사이의 임의의 정수
print(i)
```

아래는 난수를 사용한 간단한 예제로서 난수를 생성한 후 어떤 숫자인지 맞추는 프로그램이다. 즉, 사용자가 입력한 숫자가 난수보다 큰지 작은지를 알려주고 계속 추측해서 난수값을 맞추게하는 예제이다.

```

from random import randint

n = randint(1, 100)

while True:
    ans = input("Guess my number (Q to exit): ")
    if ans.upper() == "Q":
        break
    ians = int(ans)
    if (n == ians):
        print("Correct!")
        break
    elif (n
>
    ians):
        print("Choose higher number")
    else:
        print("Choose lower number")

```

샘플링 (sample)

random 모듈에서 또 한가지 유용한 기능은 리스트, set, 튜플 등과 같은 컬렉션으로부터 일부를 샘플링해서 뽑아내는 기능이다. random 모듈에서 sample(컬렉션, 샘플수) 함수는 지정된 컬렉션으로부터 샘플수만큼 랜덤 추출을 하는 함수이고, 이보다 좀 더 복잡한 choices() 함수는 샘플링에 가중치를 주어 추출하는 기능을 가지고 있다. 아래 예제는 (1) 숫자 리스트로부터 3개를 랜덤하게 샘플링하고 (2) 튜플에서 2개의 문자열을 샘플링하는 예이다.

```

import random

# (1) 숫자리스트 샘플링
numlist = [1,2,3,4,5,6,7,8,9]
s = random.sample(numlist, 3)
print(s) # [1, 2, 8]

# (2) 튜플 샘플링
frutes = ('사과', '귤', '포도', '배')
s = random.sample(frutes, 2)
print(s) # ['배', '사과']

```

1. 정규 표현식 (Regular Expression)

정규 표현식은 특정한 규칙을 가진 문자열의 패턴을 표현하는 데 사용하는 표현식(Expression)으로 텍스트에서 특정 문자열을 검색하거나 치환할 때 흔히 사용된다. 예를 들어, 웹페이지에서 전화번호나 이메일 주소를 발췌한다거나 로그파일에서 특정 에러메시지가 들어간 라인들을 찾을 때 정규 표현식을 사용하면 쉽게 구현할 수 있다. 정규 표현식은 간단히 정규식, **Regex** 등으로 불리우곤 한다.

2. 정규 표현식 사용

정규식에서 가장 단순한 것은 특정 문자열을 직접 리터럴로 사용하여 해당 문자열을 검색하는 것이다. 예를 들어, 로그 파일에 "에러 1033" 이라는 문자열을 검색하여 이 문자열이 있으면 이를 출력하고 없으면 **None**을 리턴하는 경우이다. 이러한 간단한 검색을 파이썬에서 실행하는 방법은 아래와 같다.

먼저 파이썬에서 정규표현식을 사용하기 위해서는 **Regex**를 위한 모듈인 **re** 모듈을 사용한다. **re** 모듈의 **compile** 함수는 정규식 패턴을 입력으로 받아들여 정규식 객체를 리턴하는데, 즉 **re.compile(검색할문자열)** 와 같이 함수를 호출하면 정규식 객체 (**re.RegexObject** 클래스 객체)를 리턴하게 된다.

re.RegexObject 클래스는 여러 메서드들을 가지고 있는데, 이 중 여기서는 특정 문자열을 검색하여 처음 맞는 문자열을 리턴하는 **search()** 메서드를 사용해 본다. 이 **search()** 는 처음 매칭되는 문자열만 리턴하는데, 매칭되는 모든 경우를 리턴하려면 **findall()** 을 사용한다. **search()**는 검색 대상이 있으면 결과를 갖는 **MatchObject** 객체를 리턴하고, 맞는 문자열이 없으면 **None** 을 리턴한다. **MatchObject** 객체로부터 실제 결과 문자열을 얻기 위해서는 **group()** 메서드를 사용한다.

```
import re
text = "에러 1122 : 레퍼런스 오류\n 에러 1033: 아규먼트 오류"
regex = re.compile("에러 1033")
mo = regex.search(text)
if mo != None:
    print(mo.group())
```

3. 전화번호 발췌하기

정규 표현식은 단순한 리터럴 문자열을 검색하는 것보다 훨씬 많은 기능들을 제공하는데, 즉 특정 패턴의 문자열을 검색하는데 매우 유용하다. 그 한가지 예로 웹페이지나 텍스트에서 특정 패턴의 전화번호를 발췌하는 기능에 대해 알아보자. 전화번호의 패턴은 032-232-3245 와 같이 3자리-3자리-4자리로 구성되어 있다고 가정하자. 정규식에서 숫자를 의미하는 기호로 **\d** 를 사용한다. 여기서 **d**는 **digit** 을 의미하고 0 ~ 9 까지의 숫자 중 아무 숫자나 될 수 있다. 따라서, 위 전화번호 패턴을 정규식으로 표현하면 **\d\d\d-\d\d\d-\d\d\d\d** 와 같이 될 수 있다. 아래는 이러한 패턴을 사용하여 전화번호를 발췌하는 예이다.

```
import re

text = "문의사항이 있으면 032-232-3245 으로 연락주시기 바랍니다."

regex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
matchobj = regex.search(text)
phonenumber = matchobj.group()
print(phonenumber)
```

위 예제에서 `re.compile(전화번호패턴)` 함수는 전화번호 패턴에 갖는 정규식 객체를 리턴하게 되고, `search()`를 사용하여 첫번째 전화번호 패턴에 매칭되는 번호를 리턴한다. 그리고 이로부터 실제 전화번호를 얻기 위해서는 `group()` 메서드를 사용하였다.

4. 다양한 정규식 패턴 표현

위의 전화번호 예제에서는 숫자를 표현하는 `\d` 만을 살펴보았는데, 정규 표현식에는 매우 다양한 문법과 기능들이 제공되고 있다. 아래는 이러한 다양한 정규식 표현 중 자주 사용되는 패턴들을 정리한 것이다.

패턴	설명	예제		
<code>^</code>	이 패턴으로 시작해야 함	<code>^abc</code> : abc로 시작해야 함 (abcd, abc12 등)		
<code>\$</code>	이 패턴으로 종료되어야 함	<code>xyz\$</code> : xyz로 종료되어야 함 (123xyz, strxyz 등)		
<code>[문자들]</code>	문자들 중에 하나이어야 함. 가능한 문자들의 집합을 정의함.	<code>[Pp]ython</code> : "Python" 혹은 "python"		
<code>[^문자들]</code>	[문자들]의 반대로 피해야할 문자들의 집합을 정의함.	<code>[^aeiou]</code> : 소문자 모음이 아닌 문자들		
<code>\</code>		두 패턴 중 하나이어야 함 (OR 기능)	<code>a \</code>	<code>b : a</code> 또는 <code>b</code> 이어야 함
<code>?</code>	앞 패턴이 없거나 하나이어야 함 (Optional 패턴을 정의할 때 사용)	<code>\d?</code> : 숫자가 하나 있거나 없어야 함		
<code>+</code>	앞 패턴이 하나 이상이어야 함	<code>\d+</code> : 숫자가 하나 이상이어야 함		
<code>*</code>	앞 패턴이 0개 이상이어야 함	<code>\d*</code> : 숫자가 없거나 하나 이상이어야 함		
패턴 <code>{n}</code>	앞 패턴이 n번 반복해서 나타나는 경우	<code>\d{3}</code> : 숫자가 3개 있어야 함		
패턴 <code>{n,m}</code>	앞 패턴이 최소 n번, 최대 m 번 반복해서 나타나는 경우 (n 또는 m 은 생략 가능)	<code>\d{3,5}</code> : 숫자가 3개, 4개 혹은 5개 있어야 함		
<code>\d</code>	숫자 0 ~ 9	<code>\d\d\d</code> : 0 ~ 9 범위의 숫자가 3개를 의미 (123, 000 등)		
<code>\w</code>	문자를 의미	<code>\w\w\w</code> : 문자가 3개를 의미 (xyz, ABC 등)		
<code>\s</code>	화이트 스페이스를 의미하는데, <code>[\t\n\r\f]</code> 와 동일	<code>\s\s</code> : 화이트 스페이스 문자 2개 의미 (<code>\r\n</code> , <code>\t\t</code> 등)		
<code>.</code>	뉴라인(<code>\n</code>) 을 제외한 모든 문자를 의미	<code>.{3}</code> : 문자 3개 (F15, 0x0 등)		

정규식 패턴의 한 예로 "에러 {에러번호}"와 같은 형식을 띄는 부분을 발췌해 내는 예제를 살펴보자. 여기서 에러 패턴은 "에러" 라는 리터럴 문자열과 공백 하나, 그 뒤에 1개 이상의 숫자이다. 이를 표현하면 아래와 같다.

```
import re
text = "에러 1122 : 레퍼런스 오류\n 에러 1033: 아규먼트 오류"
regex = re.compile("에러\s\d+")
mc = regex.findall(text)
print(mc)
# 출력: ['에러 1122', '에러 1033']
```

위 예제는 첫번째 패턴 매칭값을 리턴하는 `search()` 메서드 대신 패턴에 매칭되는 모든 결과를 리턴하는 `findall()`을 사용하였다. `findall()`는 결과 문자열들의 리스트(list)를 리턴한다.

5. 정규식 그룹(Group)

정규 표현식에서 () 괄호는 그룹을 의미한다. 예를 들어, 전화번호의 패턴을 `\d{3}-\d{3}-\d{4}` 와 같이 표현하였을 때, 지역번호 3자를 그룹1으로 하고 나머지 7자리를 그룹2로 분리하고 싶을 때, `(\d{3})-(\d{3}-\d{4})` 와 같이 둥근 괄호로 묶어 두 그룹으로 분리할 수 있다.

이렇게 분리된 그룹들은 `MatchObject`의 `group()` 메서드에서 그룹 번호를 파라미터로 넣어 값을 가져올 수 있는데, 첫번째 그룹 지역번호는 `group(1)` 으로, 두번째 그룹은 `group(2)` 와 같이 사용한다. 그리고 전체 전화번호를 가져올 때는 `group()` 혹은 `group(0)` 을 사용한다.

```
import re

text = "문의사항이 있으면 032-232-3245 으로 연락주시기 바랍니다."

regex = re.compile(r'(\d{3})-(\d{3}-\d{4})')
matchobj = regex.search(text)
areaCode = matchobj.group(1)
num = matchobj.group(2)
fullNum = matchobj.group()
print(areaCode, num) # 032 232-3245
```

그룹을 위와 같이 숫자로 인덱싱하는 대신 그룹이름을 지정할 수도 있는데 이를 정규식에서 **Named Capturing Group** 이라 한다. 파이썬에서 **Named Capturing Group**을 사용하는 방법은 (?P<그룹명>정규식) 와 같이 정규식 표현 앞에 ?P<그룹명>을 쓰면 된다. 그리고 이후 `MatchObject`에서 `group('그룹명')` 을 호출하면 캡처된 그룹 값을 얻을 수 있다.

```
import re

text = "문의사항이 있으면 032-232-3245 으로 연락주시기 바랍니다."

regex = re.compile(r'(?P\d{3})-(?P\d{3}-\d{4})')
matchobj = regex.search(text)
areaCode = matchobj.group("area")
num = matchobj.group("num")
print(areaCode, num) # 032 232-3245
```


직렬화와 역직렬화

파이썬의 객체를 일련의 바이트들로 변환한 후 나중에 다시 파이썬 객체로 복원하게 할 수 있는데, 이렇게 파이썬 객체를 일련의 바이트들로 변환하는 것을 직렬화(Serialization)라 하고, 다시 바이트들을 파이썬 객체로 메모리 상에 복원하는 것을 역직렬화(Deserialization)이라 한다. 직렬화된 바이트들은 외부 장치에 저장하거나 다른 시스템으로 전송할 수 있으며, 파이썬을 사용하는 다른 시스템은 (OS와 같은 플랫폼 환경에 상관없이) 다시 이를 파이썬 객체로 역직렬화할 수 있다.

파이썬에서 직렬화(Serialization)와 역직렬화(Deserialization)를 사용하기 위해서는 pickle 모듈 혹은 cPickle 모듈을 사용한다.

(참고: pickle 모듈은 파이썬으로 작성되었으며 cPickle 모듈은 C로 작성되어 있는데, cPickle 모듈이 더 빠르기 때문에 성능이 중요한 경우 cPickle 을 사용하는 것이 좋다. 하지만, cPickle은 서브클래스를 만들 수 없기 때문에, 서브클래스를 작성할 필요가 있으면 pickle 을 사용해야 한다. cPickle은 Python 2 에선 "import cPickle"을 사용하여 import 하고, Python 3에서는 "import _pickle"을 사용한다.)

아래 예제는 사각형 클래스의 객체를 이진 파일에 직렬화한 후 이를 다시 읽어들여 사각형 객체로 복원하는 역직렬화 과정을 예시한 것이다. 직렬화하면 바이너리 형태이므로 이진 파일 모드로 파일을 열어야 한다. 아래 예제에서 복원된 객체 r 은 width, height 등의 attribute를 이전에 저장한 값 그대로 가지고 있다.

```
import pickle

class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height

rect = Rectangle(10, 20)

# 사각형 rect 객체를 직렬화 (Serialization)
with open('rect.data', 'wb') as f:
    pickle.dump(rect, f)

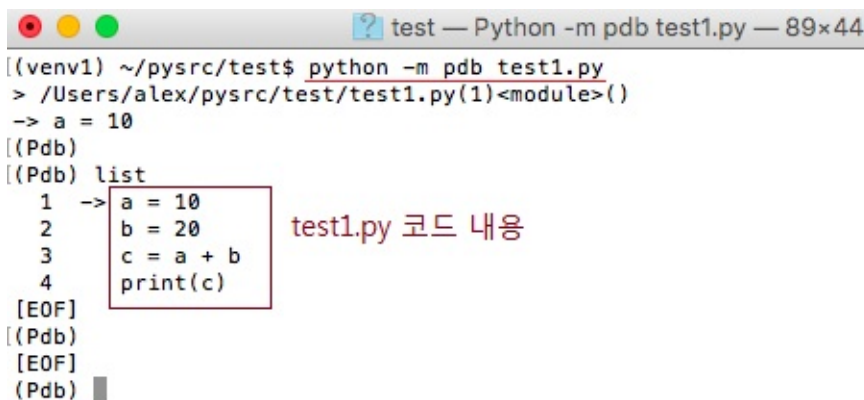
# 역직렬화 (Deserialization)
with open('rect.data', 'rb') as f:
    r = pickle.load(f)

print("%d x %d" % (r.width, r.height))
```


1. Python 디버깅

Python은 디버깅을 위해 pdb 라는 Python Debugger 모듈을 제공하고 있다. 이 디버거는 Step over/Step into, 중단점(breakpoint) 설정, 콜스택 검사, 소스 리스팅, 변수 치환 등 다양한 기능을 가지고 있다.

Python 디버깅에서 사용되는 방법 중 하나로 아래와 같이 "python -m pdb 파이선파일.py" 를 사용하는 방법이 있다. "-m pdb" 를 사용하게 되면, 디버거 하에서 파이선 파일을 실행하게 된다. 예를 들어, 아래 예를 보면 test1.py라는 파이선 모듈의 첫번째 라인에서 프로그램을 중단하고 디버거 프롬프트인 (Pdb) 를 표시함을 볼 수 있다. (밑에서 설명하지만 아래는 Pdb 프롬프트에서 list를 사용하여 test1.py의 소스코드를 출력하였다)



```

test — Python -m pdb test1.py — 89x44
[(venv1) ~/pysrc/test$ python -m pdb test1.py
> /Users/alex/pysrc/test/test1.py(1)<module>()
-> a = 10
[(Pdb)
[(Pdb) list
1  -> a = 10
2      b = 20
3      c = a + b
4      print(c)
[EOF]
[(Pdb)
[EOF]
(Pdb) █

```

test1.py 코드 내용

Python 디버깅에서 흔히 사용되는 두번째 디버깅 방법은 pdb 모듈을 import 한 후, pdb.set_trace()를 중단하고 싶은 곳에 넣는 방식이다. 이렇게 하면 파이선 프로그램 실행시 pdb.set_trace() 문장이 있는 곳에서 실행을 중지하고 디버거 세션을 시작하게 한다. 아래 그림을 보면 pdb.set_trace() 다음 문장에서 (Pdb) 프롬프트가 표시되어 있음을 볼 수 있다.

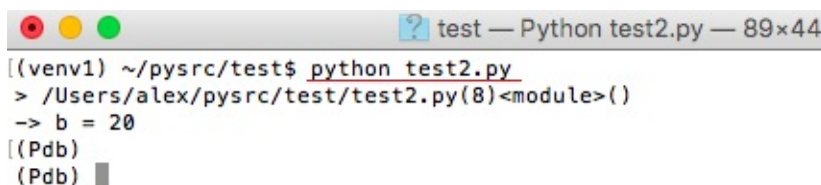
```

# test2.py 소스
import pdb

def sum(x, y):
    z = x + y
    return z

a = 10
pdb.set_trace() # 이곳에서 프로그램 중단
b = 20
c = sum(a, b)
print(c)

```



```

test — Python test2.py — 89x44
[(venv1) ~/pysrc/test$ python test2.py
> /Users/alex/pysrc/test/test2.py(8)<module>()
-> b = 20
[(Pdb)
(Pdb) █

```

test2.py 코드 내용

2. PDB 사용법

PDB를 사용하여 디버깅 모드로 진입하게 되면, (Pdb) 프롬프트가 나오게 되는데 여기서 여러 PDB 명령을 사용할 수 있다. 즉, 다음 문장을 실행하거나(next), 변수 값을 프린트하거나(print), 소스코드를 리스팅하거나(list), 함수 안으로 들어가거나(step into) 하는 일들을 PDB 명령을 사용하여 실행할 수 있다.

다음은 자주 사용되는 PDB 명령들을 요약한 것이다. 명령어를 단어 전체를 사용해도 되지만, 보통 약어로 앞의 한 글자만 사용할 수 있다. 즉, next 대신 n 을 사용할 수 있다.

PDB 명령어	실행내용
help	도움말
next	다음 문장으로 이동
print	변수값 화면에 표시
list	소스코드 리스트 출력. 현재 위치 화살표로 표시됨
where	콜스택 출력
continue	계속 실행. 다음 중단점에 멈추거나 중단점 없으면 끝까지 실행
step	Step Into 하여 함수 내부로 들어감
return	현재 함수의 리턴 직전까지 실행
!변수명 = 값	변수에 값 재설정

아래 그림은 여러 PDB 명령들을 사용하여 (위에서 예시한) test2.py 코드를 디버깅한 예이다.

```

test — Python test2.py — 89x44
(venv1) ~/pysrc/test$ python test2.py
> /Users/alex/pysrc/test/test2.py(8)<module>()
-> b = 20
(Pdb) list
   3     def sum(x, y):
   4         return x + y
   5
   6     a = 10
   7     pdb.set_trace()
   8 -> b = 20
   9     c = sum(a, b)
  10     print(c)
[EOF]
(Pdb) next
> /Users/alex/pysrc/test/test2.py(9)<module>()
-> c = sum(a, b)
(Pdb) step Step Into
--Call--
> /Users/alex/pysrc/test/test2.py(3)sum()
-> def sum(x, y):
(Pdb) n
> /Users/alex/pysrc/test/test2.py(4)sum()
-> return x + y
(Pdb) where
/Users/alex/pysrc/test/test2.py(9)<module>()
-> c = sum(a, b)
> /Users/alex/pysrc/test/test2.py(4)sum()
-> return x + y
(Pdb) return
--Return--
> /Users/alex/pysrc/test/test2.py(4)sum()->30
-> return x + y
(Pdb) n
> /Users/alex/pysrc/test/test2.py(10)<module>()
-> print(c)
(Pdb) p a,b,c
(10, 20, 30)
(Pdb) !c = 100 변수값 변경
(Pdb) n
100
--Return--
> /Users/alex/pysrc/test/test2.py(10)<module>()->None
-> print(c)
(Pdb) quit

```

3. IDE 디버깅

PyCharm, PTVS, Spyder와 같은 IDE는 UI에서 쉽게 디버깅을 할 수 있는 기능을 제공하고 있다. 일반적으로 코드 상에 중단점(Breakpoint)를 걸어 디버깅 실행 버튼을 눌러 디버깅을 시작하고, Step Over, Step Into 등의 버튼을 사용하여 한 라인씩 디버깅해 나갈 수 있다. 아래는 Spyder에서 디버깅하는 예이다.

The screenshot shows the Spyder Python IDE interface. The main editor displays a Python script named `temp.py` with the following content:

```
1 # -*- coding: utf-8 -*-
2 """
3 Spyder Editor
4
5 This is a temporary script file.
6 """
7
8 def sum(x, y):
9     return x + y
10
11 a = 10
12 b = 20
13 c = sum(a, b)
14 print(c)
```

The **Variable explorer** panel on the right shows the current state of variables:

Name	Type	Size	Value
a	int	1	10
b	int	1	20
c	int	1	30
x	int	1	10
y	int	1	20

The **IPython console** at the bottom shows the execution of the `sum()` function:

```
ipdb>
> /Users/alex/.spyder2-py3/temp.py(9)sum()
      8 def sum(x, y):
1----> 9     return x + y
      10
```

The status bar at the bottom indicates: Permissions: RW, End-of-lines: LF, Encoding: UTF-8, Line: 9, Colu: 1, Memory: 57 %.