
Chapter 6

모듈

목차

- 모듈 : 여러 코드를 묶어 다른 곳에서 재사용할 수 있는 코드 모음
- 모듈 사용하기
- 모듈 만들기
- 모듈의 경로
- 모듈 임포트
- 모듈 임포트 파헤치기
- 유용한 팁
- 패키지

모듈 사용하기

- 현재 파이썬 3버전에서는 대략 200개가 넘는 모듈을 지원
 - 문자열(string), 날짜(date), 시간(time), 십진법(decimal), 랜덤(random), 파일(file), os, sqlite3, sys, xml, email, http 등등
 - <http://docs.python.org/3.9/library/index.html>
 - C:\Python39\Lib\ 에 기본으로 제공되는 라이브러리들이 있음
-> 대부분 소스 제공하고 있으니 C:\Python39\Lib\ http\client.py 를 열어 확인해 보자.
-> Python 소스와 모듈의 제작 방식을 배울 수 있다.
- 간단하게 모듈을 사용할 수 있음
- 모듈을 사용하는 이유
 - 코드의 재 사용성
 - 코드를 이름공간으로 구분하고 관리 할 수 있음.
 - 복잡하고 어려운 기능을 포함하는 프로그램을 간단하게 만들 수 있음.
(논리 구성의 단순화)

- **import** : 모듈을 사용할 수 있도록 현재 이름공간으로 가져 올
- (예) math 모듈 : 삼각함수, 제곱근, 로그함수 등 수학과 관련된 기능이 들어 있는 내장 모듈.

```
import math
```

```
print(math.pow(100, 10))
```

```
print(math.log(100))
```

```
print(math.pow(100, 10))
```

```
print(math.pi)
```

```
print(dir(math))
```

#math 모듈을 현재 이름 공간으로 가져 올

import에서 가져온 이름으로써 사용

#math 모듈에 정의된 함수와 데이터 확인

[실행결과]

1024.0

4.605170185988092

1e+20

3.141592653589793

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh',  
'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp',  
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd',  
'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10',  
'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin',  
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

(interpreter mode 말고) VS Code에서 작성하면 함수이름 위에 괄호 열 때 도움말을 볼 수 있다.

- FTP로 서버에 접근해 파일 리스트를 가져오는 프로그램

```
>>> from ftplib import FTP #ftplib 모듈에서 FTP 클래스 가져 옴
```

```
>>> ftp = FTP("ftp1.at.proftpd.org") #ftp 서버 접속
```

```
>>> ftp.login() #login 함수에 아무것도 적지 않으면 anonymous 접속  
'230 Anonymous access granted, restrictions apply'
```

```
>>> ftp.retrlines('LIST') #LIST를 가져 옴  
-rw-rw-r-- 1 ftp ftp 451 Jul 1 2005 README.MIRRORS  
drwxrwxr-x 3 ftp ftp 4096 Jul 1 2005 devel  
drwxrwxr-x 3 ftp ftp 4096 Dec 2 2010 distrib  
drwxrwxr-x 4 ftp ftp 4096 Jul 1 2005 historic  
'226 Transfer complete'
```

```
>>> ftp.quit()  
'221 Goodbye.'
```

모듈 만들기

- 사용자가 직접 모듈을 만들 수 있음.
- 큰 프로젝트의 경우 모듈 단위로 일을 진행하게 됨.
- 모듈은 일반적으로 <모듈이름>.py 라는 파일로 저장됨.
- **simpleset 모듈 만들기**
 - 텍스트 에디터를 이용해 교집합, 차집합, 합집합 함수를 만듭시다.
→ simpleset.py 이름으로 저장
 - → simpleset.py를 파이썬 라이브러리 디렉터리 (c:\python310\Lib)로 옮기기
 - → import 명령을 이용해 simpleset 모듈을 가져와 사용하기
 - >>> **import simpleset**

- functools 모듈의 **reduce 함수** `reduce(function, sequence[, initial]) -> value`
 - function 을 sequence 아이템들에 차례로 적용해서 single value 로 만들
 - 예) `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) --> (((1+2)+3)+4)+5).`

```
# functools 모듈에서 *가져옴
# (reduce 함수 사용)
from functools import *
```

```
def intersect(*ar):    #교집합
    return reduce(__intersectSC, ar)
```

```
def __intersectSC(listX, listY):
    setList = []
    for x in listX:
        if x in listY:
            setList.append(x)
    return setList
```

```
def difference(*ar):    #차집합
    setList = []
    intersectSet = intersect(*ar) #교집합
    unionSet = union(*ar) #합집합
    # 합집합에는 속하고
    # 교집합에는 속하지 않는 원소들 모으기.
    for x in unionSet:
        if not x in intersectSet:
            setList.append(x)
    return setList
```

```
def union(*ar):         #합집합
    setList = []
    for item in ar:
        for x in item:
            if not x in setList:
                setList.append(x)
    return setList
```

__main__을 사용한 유용한 팁

- 모듈이 (1)직접 실행 혹은 (2)임포트되어 실행 되었는지 구분해 줄 수 있는 `__name__` 어트리뷰트
 - 모듈이 임포트 되었을 때 `__name__`은 **모듈 이름**
 - 모듈이 직접 실행 되었을 때 `__name__`은 “`__main__`”
- `simpleset.py` 맨 하단에 아래를 추가

```
if __name__ == '__main__':  
    print("모듈을 직접 실행하셨습니다")  
else:  
    print("임포트 하셨습니다")
```

- `simpleset.py` 를 실행 → **"모듈을 직접 실행하셨습니다"**
 - 다른 모듈에 import하여 실행:
`test.py` 만들어서 `import simpleset` 만 넣고 수행. → **"임포트 하셨습니다"**
- 모듈 개발할 때
직접 실행 시에는
테스트 코드가 수행되게 하기


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

```
    setA = [1, 3, 7, 10]
```

```
    setB = [2, 3, 4, 9]
```

```
    if union(setA, setB) == [1, 3, 7, 10, 2, 4, 9]:
```

```
        print("union 정상")
```

```
    else:
```

```
        print("union 오류")
```

```
    if intersect(setA, setB, [1,2,3]) == [3]:
```

```
        print("intersect 정상")
```

```
    else:
```

```
        print("intersect 오류")
```

```
    if difference(setA, setB) == [1, 7, 10, 2, 4, 9]:
```

```
        print("difference 정상")
```

```
    else:
```

```
        print("difference 오류")
```


[실행결과]

union 정상

intersect 정상

difference 정상

모듈 만들기 - 어디서나 사용할 수 있게하기

- **simpleset 모듈 → 라이브러리 디렉터리에 복사.**
 - 를 파이썬 라이브러리 디렉터리 (c:\python39\Lib)에 저장.
- 인터프리터에서 simpleset.py 모듈 사용하기.

```
>>> import simpleset
>>> dir(simpleset)
['WRAPPER_ASSIGNMENTS', 'WRAPPER_UPDATES', '__builtins__',
 '__cached__', '__doc__', '__file__', '__intersectSC__', '__loader__', '__name__',
 '__package__', '__spec__', 'cmp_to_key', 'difference', 'intersect', 'lru_cache', 'partial',
 'partialmethod', 'reduce', 'singledispatch', 'total_ordering', 'union', 'update_wrapper',
 'wraps']

>>> setA = [1, 3, 7, 10]
>>> setB = [2, 3, 4, 9]
>>> simpleset.union(setA, setB)
[1, 3, 7, 10, 2, 4, 9]

>>> simpleset.intersect(setA, setB, [1,2,3])
[3]
```

모듈 탐색 경로

- 모듈을 탐색하는 경로 밖의 모듈은 임포트 할 수 없음
- 모듈 경로 탐색 순서.
 - (1) 프로그램이 실행된 디렉터리
 - (2) PYTHONPATH 환경 변수에 등록된 위치
 - 표준 라이브러리 디렉터리 (c:\python39\Lib)
- 모듈을 임포트 했을 때 모듈의 위치를 탐색하는 경로
 - `sys.path` 에서 리스트 형식으로 관리
 - 프로그램 동작 중에도 `sys.path`에 경로를 추가/삭제할 수 있음.

```
>>> import sys
>>> sys.path
['', 'C:\\Python310\\Lib\\idlelib', 'C:\\Python310\\python310.zip',
'C:\\Python310\\DLLs', 'C:\\Python310\\lib', 'C:\\Python310',
'C:\\Python310\\lib\\site-packages']
>>> sys.path.append('c:\\mymodules') #mymodules 경로 추가
>>> sys.path
['', 'C:\\Python310\\Lib\\idlelib', 'C:\\Python310\\python310.zip',
'C:\\Python310\\DLLs', 'C:\\Python310\\lib', 'C:\\Python310',
'C:\\Python310\\lib\\site-packages', 'c:\\mymodules']
>>> sys.path.remove('c:\\mymodules') #mymodules 경로 삭제
```

import문의 위치

- 모듈 안의 어트리뷰트 (함수, 데이터)들을 사용하려면 import 해야 함.
import 구문은 어디에서나(함수, 제어문 내부에서도) 사용 가능
- import <모듈이름> ← 기본적인 import 방법
 - 모듈.이름 형식으로 모듈 안의 데이터나 함수를 사용 할 수 있음

```
>>> math.pow # math 모듈을 못찾음
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'math' is not defined
```

```
>>> def fn():
```

```
...     import math # 함수 안에서 import하여 사용  
...     print(math.pow(10,2))  
...
```

```
>>> fn() # 함수 안에서 import하여 사용  
100.0
```

```
>>> math.pow # (함수 namespace가 독립적이므로 )math 모듈을 못찾음
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'math' is not defined
```

```
>>> if True:
```

```
...     import math # 블록 안에서 import  
...
```

```
>>> math.pow # (블록은 별도의 namespace가 아니므로 )math 모듈을 찾음  
<built-in function pow>
```

다양한 import 구문

- **from <모듈> import <어트리뷰트>**

- 앞에 모듈 이름 없이 어트리뷰트 이름만으로 사용 가능.

```
>>> from simpleset import union  
>>> union([1, 2, 3], [3], [3, 4])  
[1, 2, 3, 4]
```

- **from <모듈> import ***

- `밀줄()`로 시작하는 어트리뷰트(함수, 데이터) 제외하고 모두 import.
[비교] `import <모듈>`로 가져오면 `밀줄()`로 시작하는 어트리뷰트도 보임.

- **import <모듈> as <별칭>**

- `<모듈>` 이름을 `<별칭>`으로 변경하여 import.

다양한 import 구문

```
>>> from simpleset import *
```

```
>>> __intersectSC("12", "14")
```

_로 시작하는 함수는 안보임

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name '__intersectSC' is not defined

```
>>> union("12", "14")
```

_로 시작하는 함수는 모듈이름없이 사용가능.

```
['1', '2', '4']
```

```
>>>
```

```
>>> import simpleset
```

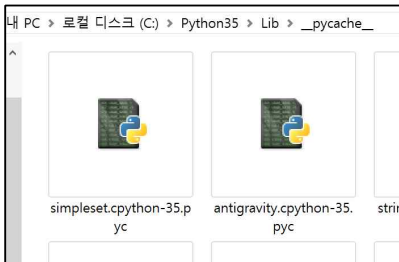
```
>>> simpleset.__intersectSC("12", "14")
```

_로 시작하는 함수도 보임

```
['1']
```

바이트 코드

- 바이트 코드(*.pyc)
 - 컴파일된 파일 (import 하면 생성됨. __pycache__ 서브 디렉터리에 위치.)
 - 모듈의 **임포트를 빠르게** 해 줌
 - 바이트 코드가 이미 있으면 : 바로 바이트 코드 로딩
 - 바이트 코드가 없으면 : 모듈을 컴파일 해서 바이트 코드를 생성하여 로딩
- 바이트 코드가 생성된 모습



Import → reload

- 모듈을 임포트 하면 해당 모듈의 코드가 실행됨
- 모듈이 임포트되어 메모리에 모듈 코드가 로딩되면 프로그램이나 파이썬 인터프리터가 끝나기 전까지 변경되지 않음.
 - 이미 import 된 모듈의 내용을 변경 후 → 다시 import 해도
→ 파이썬 인터프리터 재시작하기 전까지는 새로운 내용이 반영되지 않음.
 - → importlib 모듈을 이용하여 모듈을 reload.

Import → reload

```
>>> import simpleset      # import하여 사용 [C:\Python39\Lib\simpleset.py 수정]
>>> simpleset.union("12", "14")
['1', '2', '4']
>>>
>>> import simpleset      # 다시 import
>>> simpleset.union("12", "14")
['1', '2', '4']          # 반영 안됨
>>>
>>> import importlib      # importlib 모듈 이용하여
>>> importlib.reload(simpleset)    # reload
<module 'simpleset' from
'C:\Python39\lib\simpleset.py'>
>>> simpleset.union("12", "14")
[]                        # 반영 됨!
```

모듈 수정 →

```
def union(*ar):          #합집합
    setList = []
    """for item in ar:
        for x in item:
            if not x in setList:
                setList.append(x)"""
    return setList
```

패키지

■ 모듈의 모음

(모듈의 한 종류)

(`__path__` 속성을 가진 모듈)

- 파이썬의 모듈 이름공간을 구조화 하는 한 방법
- 파이썬 내장 라이브러리 중 `xml` 패키지의 디렉터리 구조
- 패키지를 이용한 이름 공간의 구조화. 패키지 내의 하위 모듈을 .으로 구분하여 표현

(예: `xml.dom`)

패키지 디렉터리에는 `__init__.py` 파일에 패키지를 초기화하는 코드를 둬. (C:\Python39\Lib\xml\에서 확인하자)

`__all__`에 `import *` 했을 때 직접 접근 가능한 이름들을 리스트로 정의함.

```
|--- __init__.py
    +--- dom
        +--- __init__.py
        +--- domreg.py
        +--- expatbuilder.py
        ...
    +--- etree
        +--- __init__.py
        +--- cElementTree.py
        ...
    +--- parsers
        +--- __init__.py
        +--- expat.py
```

패키지 - xml 패키지를 임포트하는 방법

- [주의!] import는 “모듈” 단위까지 해 줬을 때 하위 attribute들을 사용 가능. “패키지” 명으로 import했을 때 하위의 패키지나 모듈명을 사용할 수는 없음.

```
>>> import xml
```

```
>>> xml.__all__
```

```
['dom', 'parsers', 'sax', 'etree']
```

```
>>> xml.dom # 하위 패키지이므로 별도로 import 필요
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
AttributeError: module 'xml' has no attribute 'dom'
```

```
>>> import xml.dom # 하위 패키지이므로 별도로 import
```

```
>>> xml.dom
```

```
<module 'xml.dom' from 'C:\\Python35\\lib\\xml\\dom\\__init__.py'>
```

```
>>> xml.dom.minidom # 하위 모듈이므로 별도로 import 필요
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
AttributeError: module 'xml.dom' has no attribute 'minidom'
```

```
>>> import xml.dom.minidom as minidom # 별칭 지정하여 import
```

```
>>> minidom.parseString("<foo><bar/></foo>")
```

```
<xml.dom.minidom.Document object at 0x000001EF7876B228>
```

패키지에서의 import 사용 (절대 경로, 상대경로)

- import 문에서의 절대경로와 상대경로
 - . 이나 .. 으로 시작하는 경우 상대경로.
그 외는 모두 절대경로.
 - 절대경로일 때의 최상위 위치는 패키지 루트.
- 확인하기
 - VS Code에서 “폴더 열기”로 “C:\Python39\Lib\xml” 폴더를 열기
 - Control+Shift+F → import 를 “대소문자구분”, “단어 단위로” 찾기
 - dot(.) 로 시작 : 상대경로
 - xml 로 시작 : 절대 경로. (외부에서 패키지 import할 때 “xml”부터 찾아 들어감)

상대 경로 import 사용 시 주의사항

- **메인 모듈에서는 상대 경로 사용 불가.**
 - 즉, 상대경로 import를 포함한 python 모듈은 스크립트로 단독 실행할 수 없음. (antipattern)
 - 상대경로는 `__name__`에 의해 판단하는데 메인 모듈에서는 이 값이 `'__main__'`이므로 상대 위치를 찾을 수 없음. => 자신이 최상위 모듈임.
- 대신 **메인 모듈에서는 implicit relative import 가능.**
 - `import util` 했을 때 `dir(.)` 으로 시작하지 않았지만 `util.py`를 '현재 디렉터리'에서도 찾아줌.
- **import 되는 모듈에서는 implicit relative import 불가! => 반드시 explicit relative import(. 으로 시작) 시켜야 함.**
- 명시적으로 부모 디렉터리 표현하기
 - 부모 .. 2대부모 ... 3대부모

패키지 사용 예

[개발 중에 단독 실행시켜보고 싶을 때]
-m 옵션 : run library module as a script
> cd {calculator의 상위 디렉터리}
> python -m calculator.add_and_multiply

calculator 디렉터리

add_and_multiply.py

패키지 안의 모듈들끼리 import할 때는
명시적인 상대경로 또는 절대경로 사용.

```
from .multiplication import multiply  
# from calculator.multiplication import multiply
```

```
def add_and_multiply(a,b):  
    return multiply(a,b) + (a+b)
```

multiplication.py

```
def multiply(a,b):  
    return(a*b)
```

main.py

```
from calculator.add_and_multiply import add_and_multiply
```

```
if __name__ == '__main__':  
    print(add_and_multiply(1,2))
```

단독 실행할 모듈에서는 암시적인
상대경로 사용.

Chapter 8

입출력

- 표준 입출력
- 파일 입출력
- pickle

표준 입출력 - 출력

- `print()` 함수 : 파이썬 버전 2.x에서는 함수가 아니었지만, 3.0에서는 함수로 변경됨
- 괄호 안에 출력할 인자 표시

```
>>> print(1)
1
>>> print('hi, guyz')
hi, guyz
```

- 인자 개수, 변수 타입과 상관 없음
 - `repr()` 실제 값을 있는 그대로 표현하는 문자열을 반환.
 - `eval(repr(obj))` 하면 `obj`가 나옴.

```
>>> x = 0.2
>>> print(x)
0.2
>>> str(x)
'0.2'
>>> print(str(x))
0.2
>>> a = repr('hello\n')
>>> print(a)
'hello\n'
```


표준 입출력 - 출력

- 인자가 여러 개(commma(,)로 구분)면 공백을 출력하여 구분
- 공백을 없애려면 문자열의 '+'연산자 사용하여 하나의 문자열 만들어 출력.

```
>>> age = 22
>>> print(age,'세 입니다')
22 세 입니다
>>> print(str(age)+'세 입니다')
22세 입니다
```

- 구분자(sep): 기본값은 공백(' ')
- 끝라인(end): 기본값은 줄바꿈('\n')
- 출력(file): 기본값은 표준출력(sys.stdout)

```
>>> f = open('test.txt','w')
>>> print("welcom to", "python", sep="~", end="!", file=f)
>>> f.close()
```

- test.txt 파일 열어서 내용 확인해 보기

출력정렬 - 문자열 정렬함수

- string 객체 `rjust()` 오른쪽 정렬, `ljust()` 왼쪽 정렬, `center()` 가운데 정렬

```
>>>
```

```
1 * 1 = 1  
2 * 2 = 4  
3 * 3 = 9  
4 * 4 = 16  
5 * 5 = 25
```

- string 객체 `zfill()` : 빈칸 대신 0으로 채움

```
>>>
```

```
1 * 1 = 001  
2 * 2 = 004  
3 * 3 = 009  
4 * 4 = 016  
5 * 5 = 025
```

출력정렬 - 문자열의 format함수

- 문자열 내에서 값이 들어가길 원하는 곳은 **{}**로 표시하고, {}안은 **숫자**로 표현 가능. (숫자 생략하면 순서대로 매치됨)

```
>>> print("{0} is {1}".format("apple", "red"))  
apple is red
```

{ } 안의 값을 지정할 때 format인자로써 (**키워드 인자** 사용 가능)

```
>>> print("{item} is {color}".format(item="apple",color="red"))  
apple is red
```

- 사전을 키워드 인자로 주기

```
>>> dic = {"item":"apple", "color":"red"}  
>>> print("{0[item]} is {0[color]}".format(dic))  
apple is red
```

첫번째 인자의 "item"값
첫번째 인자의 "color"값

- **locals(), globals()** (사전형식의 지역변수, 전역변수) 를 키워드 인자로 사용 가능
- ****** 을 사용하면 사전의 내용을 키워드 인자로 넘기는 것이므로 “키값”을 그대로 사용하면 됨.

```
def fn():  
    item = 'apple'  
    color = 'red'  
    print("{0[item]} is {0[color]}".format(locals()))  
    print("{item} is {color}".format(**locals()))  
    print('locals=', locals())  
    print('globals=', globals())
```

fn()

[실행결과]

apple is red

apple is red

locals= {'item': 'apple', 'color': 'red'}

globals= {'__name__': '__main__', '__doc__': None,}

- 변수명[인덱스], 인스턴스의 멤버 변수 사용 가능

```
def fn():  
    종락  
fn()  
  
class foo:  
    var = 0.14  
  
numbers = [5,4,3,2,1]  
print("{numbers}".format(**locals()))  
print("{numbers[0]}".format(**locals()))  
f = foo()  
print("{f.var}".format(**locals()))
```

[실행결과]

.....

[5, 4, 3, 2, 1]

5

0.14

출력정렬 - 문자열의 format함수

- { 순서번호 : 채우기문자 정렬 부호표시 자릿수 콤마표시 }
- 채우기문자 : 어떤 문자든 가능
- 정렬 : '>'오른쪽정렬, '<'왼쪽정렬, '^'가운데정렬, '='공백 문자 앞에 부호표시
- 부호표시 : '+'양수도 부호표시, '-'음수만 부호표시, ''양수는 공백으로 표시
- 자릿수 : (전체자릿수).(소수점 아래 자릿수)
콤마표시 : ,

```
print("{0:10}".format(500)) # 첫인자 | 10칸
print("{0:#>10}".format(500)) # 첫인자 | 채우기문자'#' | 오른쪽정렬 | 10칸
print("{0: >+10}".format(500)) # 첫인자 | 채우기문자' ' | 오른쪽정렬 | 양수도 부호표시 | 10칸
print("{0:_<10}".format(500)) # 채우기문자'_' | 왼쪽정렬
print("{0:13,}".format(100000000)) # 3자리수 마다 콤마
print("{0:+13,}".format(100000000)) # 양수도 부호표시 | 3자리수 마다 콤마
print("{0:+13,}".format(-100000000)) # 양수도 부호표시 | 3자리수 마다 콤마
print("{0:.3f}".format(5/3)) # 소수점 아래 3째 자리까지만 표시
```

[실행결과]

```
500
#####500
      +500
500_____
   100,000,000
  +100,000,000
 -100,000,000
1.667
```

```
print("{0:$>+5}".format(10)) # 음수,양수 모두 부호표시
print("{0:$>+5}".format(-10))
print("{0:$>-5}".format(10)) # 음수만 부호표시
print("{0:$>-5}".format(-10))
print("{0:$> 5}".format(10)) # 음수는 부호표시, 양수는 공백
print("{0:$> 5}".format(-10))
```

[실행결과]

```
$$+10
$$-10
$$$10
$$-10
$$ 10
$$-10
```


- 진법: 'b' 이진수, 'd' 십진수, 'x' 16진수, 'o' 8진수, 'c' 코드를 문자
- 진법표시: #x 16진수(0x), #o 8진수(0o), #b 2진수(0b)

```
print("{0:x}".format(10))  
print("{0:b}".format(10))  
print("{0:o}".format(10))  
print("{0:c}".format(0x41))
```

```
print("{0:#x}".format(10))  
print("{0:#b}".format(10))  
print("{0:#o}".format(10))  
print("{0:c}".format(0x41))
```

```
print("{0:#10x}".format(10))  
print("{0:#10b}".format(10))  
print("{0:#10o}".format(10))  
print("{0:10c}".format(0x41))
```

[실행결과]

```
a  
1010  
12  
A  
0xa  
0b1010  
0o12  
A  
  
0xa  
0b1010  
0o12  
A
```

출력정렬 - 문자열의 format함수

- 실수 변환: 'e' 지수표현, 'f' 고정소수점 표현, '%' 퍼센트 표현
- '.' 소수점 몇 번째 자리 표현 지정

```
print("{0:e}".format(4 / 3))  
print("{0:f}".format(4 / 3))  
print("{0:%}".format(4 / 3))  
print("{0:.3f}".format(4 / 3))
```

[실행결과]

```
1.333333e+00  
1.333333  
133.333333%  
1.333
```

표준 입출력 - 입력

- input() 함수
 - 인자로써 화면에 출력할 프롬프트 지정
 - 결과값은 문자열 반환

```
>>> a = input("insert any keys :")
insert any keys :test
>>> print(a)
test
```

파일 입출력

- ~~open~~ 함수의 file 인자로 write mode open된 파일 전달 가능.
 - 반드시 open() 함수를 통해 파일을 연 후 작업 가능.

파일객체 = open(file, mode)

file	파일명
mode	r : 읽기 모드 (디폴트) w : 쓰기 모드 a : 쓰기 + 이어쓰기 모드 + : 읽기 + 쓰기 모드 b : 바이너리 모드 (mp3 등 바이너리 파일 다룰 때 사용) t : 텍스트 모드 (디폴트)

```
>>> f = open('test.txt', 'w') #쓰기 모드 파일 오픈
>>> f.write('plow deep\nwhile sluggards sleep') #write한 바이트 수를 반환
31
>>> f.close() #파일 닫기
>>> f = open('test.txt') #디폴트 : 읽기/텍스트 모드 오픈
>>> f.read() #파일내용 모두 읽기
'plow deep\nwhile sluggards sleep'
>>> f.close() #파일 닫기
```

```
f = open('test.txt', 'w')      #쓰기 모드 파일 오픈
bcount = f.write('plow deep\nwhile sluggards sleep') #write한 바이트 수를 반환
print('write한 byte 수:', bcount)
f.close()                      #파일 닫기

f = open('test.txt')           #디폴트 : 읽기/텍스트 모드 오픈
content = f.read()             #전체 내용을 읽어 하나의 문자열로 반환
print('읽은 내용:\n', content, sep='')
print('파일포인터 위치:', f.tell())

f.close()
```

[실행결과]

```
write한 byte 수: 31
읽은 내용:
plow deep
while sluggards sleep
파일포인터 위치: 32
```

```
f = open('test.txt')           #디폴트 : 읽기/텍스트 모드 오픈
```

```
line = None
while True:
    line = f.readline()        # 한 줄 씩 읽음
    if not line: break
    print(line, end='')
print()
print('파일포인터 위치:', f.tell() )
```

[실행결과]

```
plow deep
while sluggards sleep
파일포인터 위치: 32
plow deep
while sluggards sleep
파일포인터 위치: 32
```

```
f.seek(0)
lines = f.readlines()         # 모두 읽어 줄단위 리스트로 반환
for line in lines:
    print(line, end='')
print()
print('파일포인터 위치:', f.tell() )

f.close()
```

- with 구문을 사용해 파일을 열 경우 with 코드 블록을 벗어나면 자동으로 파일이 닫히므로 실수를 줄여 줌.

with open('test.txt') as f:


#디폴트 : 읽기/텍스트 모드 오픈

```
line = None
while True:
    line = f.readline() # 한 줄 씩 읽음
    if not line: break
    print(line, end='')
print()
print('파일포인터 위치:', f.tell() )
```

[실행결과]

```
plow deep
while sluggards sleep
파일포인터 위치: 32
plow deep
while sluggards sleep
파일포인터 위치: 32
```

```
f.seek(0)
lines = f.readlines() # 모두 읽어 줄단위 리스트로 반환
for line in lines:
    print(line, end='')
print()
print('파일포인터 위치:', f.tell() )
```

- 리스트나 클래스 등 객체를 파일로 저장하고 다시 읽을 때 유용
 - 파일 저장 : `pickle.dump()` 파일에서 읽기 : `pickle.load()`
 -  사용할 때는 반드시 바이너리 모드로 파일에 접근해야 함.

```
import pickle
```

```
colors = ['red', 'green', 'black']  
print('colors=', colors)
```

```
f = open('file_colors', 'wb')  
pickle.dump(colors, f)  
f.close()
```

```
del colors
```

```
f = open('file_colors', 'rb')  
colors = pickle.load(f)  
f.close()  
print('colors=', colors)
```

[실행결과]

```
colors= ['red', 'green', 'black']  
colors= ['red', 'green', 'black']
```

#pickle 사용을 위해 바이너리 쓰기 파일 오픈
#colors 리스트를 file_colors로 dump

#colors 리스트 삭제

#pickle 사용을 위해 바이너리 읽기 파일 오픈
#파일에서 리스트 load

pickle 모듈

- 사용자 정의 클래스의 객체도 사용 가능
- load할 때 주의 : 미리 test 클래스 정의를 해 놓아야 함.

```
import pickle
```

```
class test:  
    var = None
```

```
a = test()  
a.var = 'Test'
```

```
f = open('file_test', 'wb')  
pickle.dump(a, f)  
f.close()
```

```
f = open('file_test', 'rb')  
b = pickle.load(f)  
f.close()
```

```
print('b=', b)  
print('b.var=', b.var)
```

[실행결과]

```
b= <__main__.test object at 0x000001D19AAA8A00>  
b.var= Test
```

인스턴스 객체의 변수 변경

pickle을 위해 바이너리 쓰기 파일오픈
인스턴스 객체 a를 파일로 dump

pickle을 위해 바이너리 읽기 파일오픈
파일에서 인스턴스 객체 load