모듈과 패키지

모듈(Module)

- 함수, 변수 그리고 클래스의 집합
- 다른 파이썬 프로그램에서 가져와 사용할 수 있는 파이썬 파일
- 파이썬에는 다른 사람들이 만들어 놓은 모듈이 굉장히 많음
- 사용자가 모듈은 직접 만들어서 사용할 수도 있음

모듈의 장점

- 단순성(Simplicity)
  - 전체 문제에 초점을 맞추기보다는 문제의 상대적으로 작은 부분에만 초점을 맞춤
  - 단일 모듈로 작업할 수 있는 작은 도메인
  - 개발이 쉬우며 오류 발생이 적음
- 유지보수성(Maintainability)
  - 일반적으로 모듈은 서로 다른 문제 영역간에 논리적 경계를 설정하도록 설계
  - 상호 의존성을 최소화하는 방식으로 모듈을 작성하여 단일 모듈을 수정하면 프로그램의 다른 부분에 영향을 미칠 가능성이 줄어듬
  - 모듈 외부의 응용 프로그램에 대해 전혀 알지 못해도 모듈을 변경할 수 있음
  - 개발팀이 대규모 응용 프로그램에서 공동으로 작업 할 수 있음
- 재사용성(Reusability)
  - 단일 모듈에서 정의된 기능은 응용 프로그램의 다른 부분에서 (적절히 정의된 인터페이스를 통해) 쉽게 재사용 가능
  - 중복 코드를 만들 필요가 없음
- 범위 지정(Scoping)
  - 일반적으로 모듈은 프로그램의 여러 영역에서 식별자 간의 충돌을 피하는 데 도움이 되는 별도의 네임 스페이스를 정의

모듈의 종류

- 사용자 정의 모듈: 사용자가 직접 정의해서 사용하는 모듈
- 표준 모듈: 파이썬에서 기본 제공하는 모듈
- 서드 파티 모듈: 외부에서 제공하는 모듈
  - 파이썬 표준 모듈에 모든 기능이 있지 않음

- 서드 파티 모듈을 이용해 고급 프로그래밍 가능
- 게임 개발을 위한 pygame, 데이터베이스 기능의 SQLAlchemy, 데이터 분석 기능의 NumPy



사용자 정의 모듈

- 사용자가 사용할 모듈을 직접 정의
- 모듈 이름으로 파일명을 사용

- IPython 내장 매직 명령어(magic command) 사용
  - `%%writefile`: 셀의 코드를 .py 파이썬 코드 파일로 저장
  - `%load`: 파이썬 코드 파일 불러오기
  - `%run`: 파이썬 코드 파일 실행

```
%%writefile Module.py
def func1():
  print("Module.py: func1()")

def func2():
  print("Module.py: func2()")

def func3():
  print("Module.py: func3()")
```

```
Overwriting Module.py
```

```
!ls
```

```
Calculator.py  Module.py  package  __pycache__  sample_data
```

```
%load Module.py
```

```
%run Module.py
```

```
import Module
Module.func1()
Module.func2()
Module.func3()
```

```
Module.py: func1()
Module.py: func2()
```

```
Module.py: func3()
```

```
from Module import *
func1()
func2()
func3()
```

```
Module.py: func1()
Module.py: func2()
Module.py: func3()
```

[Lab] 계산기 모듈 만들기

- 사용자 정의 모듈을 이용해서 계산기에 필요한 기능들로 모듈 만들기

```
%%writefile Calculator.py
def add(a, b):
  return a + b

def sub(a, b):
  return a - b

def mul(a, b):
  return a * b

def div(a, b):
  return a / b

def mod(a, b):
  return a % b
```

```
Overwriting Calculator.py
```

```
from Calculator import *

print(add(3, 5))
print(sub(3, 5))
print(mul(3, 5))
print(div(3, 5))
print(mod(3, 5))
```

```
8
-2
15
0.6
3
```

파이썬 표준 모듈

- 파이썬에서 기본으로 내장된 유용한 속성과 함수들이 많음

```
import sys
print(sys.builtin_module_names)
```

```
('_ast', '_bisect', '_blake2', '_codecs', '_collections', '_datetime', '_elementtree', '_fun
```

```
print(dir(__builtins__))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
```

시간 모듈(datetime)

- 운영체제가 제공하는 시간 기능을 파이썬에서 사용할 수 있도록 만들어진 모듈
- 시간 모듈을 사용하기 위해서는 `import time` 필요

```
import time
print(dir(time))
```

```
['CLOCK_MONOTONIC', 'CLOCK_MONOTONIC_RAW', 'CLOCK_PROCESS_CPUTIME_ID', 'CLOCK_REALTIME', 'CL
```

- 시간 모듈 `time` 예제

```
import time
print(time)
print(time.time())
print(time.time())
print(time.time())

now = time.gmtime(time.time())
print(now)

year = str(now.tm_year)
month = str(now.tm_mon)
day = str(now.tm_mday)
print(year + "년", month + "월", day + "일")

hour = str(now.tm_hour)
minute = str(now.tm_min)
sec = str(now.tm_sec)
print(hour + "시", minute + "분", sec + "초")
```

```
<module 'time' (built-in)>
1593203327.0664105
1593203327.0666735
1593203327.066869
time.struct_time(tm_year=2020, tm_mon=6, tm_mday=26, tm_hour=20, tm_min=28, tm_sec=47, tm_wd
2020년 6월 26일
20시 28분 47초
```

- 날짜시간 모듈 `datetime`의 `date` 클래스 예제

```
from datetime import date
print(date)
print(date(2000, 1, 1))
print(date(year = 2010, month = 1, day = 1))
print(date.today())

today = date.today()
year = str(today.year)
month = str(today.month)
day = str(today.day)
weekday = "월화수목금토일"[today.weekday()]
print(year + "년", month + "월", day + "일", weekday + "요일")
```

```
<class 'datetime.date'>
2000-01-01
2010-01-01
2020-06-26
2020년 6월 26일 금요일
```

- 날짜시간 모듈 `datetime`의 `time` 클래스 예제

```
from datetime import time
print(time)
print(time(12, 0))
print(time(14, 30))
print(time(16, 30, 45))
print(time(18, 00, 15, 100000))

now = time(20, 40, 15, 20000)
hour = str(now.hour)
minute = str(now.minute)
sec = str(now.second)
msec = str(now.microsecond)
print(hour + "시", minute + "분", sec + "초", msec + "마이크로초")
```

```
<class 'datetime.time'>
12:00:00
14:30:00
16:30:45
18:00:15.100000
20시 40분 15초 20000마이크로초
```

- 날짜시간 모듈 `datetime`의 `datetime` 클래스 예제
- 날짜시간을 문자열로 표현하기 위한 `strftime()` 메소드 예제

표현	설명
%Y	년(YYYY)
%y	년(yy)

표현	설명
%m	월(mm)
%d	일(dd)
%A	요일
%H	시(24)
%I	시(12)
%p	AM, PM
%M	분(MM)
%S	초(SS)
%f	마이크로초

```
from datetime import datetime
print(datetime)
print(datetime(2020, 1, 1))
print(datetime(2020, 1, 1, 1, 15, 45))
print(datetime.now())
now = datetime.now()
print(now.strftime('%Y년 %m월 %d일 %H시 %M분 %S초'))
print(now.strftime('%y/%m/%d %p %I:%M:%S:%f'))
```

```
<class 'datetime.datetime'>
2020-01-01 00:00:00
2020-01-01 01:15:45
2020-06-26 20:32:46.271818
2020년 06월 26일 20시 32분 46초
20/06/26 PM  8:32:46:273260
```

[Lab] 태어난지 몇 일이 되었는가?

- 태어난지 얼마나 지났는지 계산하기

```
from datetime import date
birthday = date(2000, 1, 1)
today = date.today()
day = today - birthday
print(day.days)
```

```
7482
```

수학 모듈(math)

- 파이썬에서 수학에 필요한 math 모듈 제공

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

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'a
```

- `math` 모듈 대표 상수

상수	설명
`math.pi`	원주율
`math.e`	자연상수
`math.inf`	무한대

- `math` 모듈에서 제공하는 대표 함수

함수	설명
`math.factorial(x)`	x 팩토리얼
`math.gcd(a, b)`	a와 b의 최대공약수
`math.floor(x)`	x의 내림값
`math.ceil(x)`	x의 올림값
`math.pow(x, y)`	x의 y승
`math.sqrt(x)`	x의 제곱근
`math.log(x, base)`	base를 밑으로 하는 x 로그
`math.sin(x)`	x 라디안의 사인
`math.cos(x)`	x 라디안의 코사인
`math.tan(x)`	x 라디안의 탄젠트
`math.degrees(x)`	x 라디안을 도 단위로 변환
`math.radians(x)`	x 도를 라디안 단위로 변환

```
import math
print(math.factorial(3))
print(math.gcd(12, 24))
print(math.floor(math.pi))
print(math.ceil(math.pi))
print(math.pow(2, 10))
print(math.sqrt(10))
print(math.log(10, 2))
print(math.degrees(math.pi))
print(math.radians(180))
print(math.sin(math.radians(90)))
print(math.cos(math.radians(180)))
```

```
6
12
3
4
1024.0
3.1622776601683795
3.3219280948873626
180.0
3.141592653589793
1.0
-1.0
```

순열과 조합 모듈(itertools)

- `itertools` 모듈에서 곱집합, 순열, 조합 등을 구하는 함수 제공

함수	설명
`itertools.product(seq1, ...)`	시퀀스의 곱집합
`itertools.permutations(p, r)`	p 시퀀스의 요소 r개를 나열하는 순열
`itertools.combinations(p, r)`	p 시퀀스의 요소 r개를 선택하는 조합
`itertools.combinations_with_replacement(p, r)`	p 시퀀스의 요소 r개를 중복 허용해 선택하는 조합

```
import itertools
list_1 = ['a', 'b', 'c']
print(list_1)
list_2 = [1, 2]
print(list_2)
list_cp = list(itertools.product(list_1, list_2))
print(list_cp)
list_p = list(itertools.permutations(list_1, 2))
print(list_p)
list_c = list(itertools.combinations(list_1, 2))
print(list_c)
list_cr = list(itertools.combinations_with_replacement(list_1, 2))
print(list_cr)
```

```
['a', 'b', 'c']
[1, 2]
[('a', 1), ('a', 2), ('b', 1), ('b', 2), ('c', 1), ('c', 2)]
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'), ('c', 'b')]
[('a', 'b'), ('a', 'c'), ('b', 'c')]
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'), ('c', 'c')]
```

통계 모듈(statistics)

- `statistics` 모듈에서는 산술평균, 표준편차 등 통계에 필요한 계산 관련 함수들을 제공

함수	설명
`statistics.median(seq)`	시퀀스의 중앙값
`statistics.mean(seq)`	시퀀스의 산술 평균
`statistics.harmonic_mean(seq)`	시퀀스의 조화 평균
`statistics.stdev(seq)`	시퀀스의 표본 표준편차
`statistics.variance(seq)`	시퀀스의 표본 분산

```
import statistics
values = [56, 44, 67, 47, 82, 67, 92, 89, 81, 82]
print(statistics.median(values))
print(statistics.mean(values))
print(statistics.harmonic_mean(values))
print(statistics.stdev(values))
print(statistics.variance(values))
```

```
74.0
70.7
66.42170307761845
17.217884758458442
296.4555555555556
```

랜덤 모듈(random)

- 랜덤 모듈을 사용하기 위해서는 `import random` 필요
  - `random.random()`: 0.0~1.0 미만의 실수값 반환
  - `random.randint(1, 10)`: 1~10 사이의 정수 반환
  - `random.randrange(0, 10, 2)`: 0~10 미만의 2의 배수만 반환
  - `random.choice()`: 자료형 변수에서 임의의 값 반환
  - `random.sample()`: 자료형 변수에서 필요한 개수만큼 반환
  - `random.shuffle()`: 자료형 변수 내용을 랜덤으로 셔플

```
import random
print(random.random())
print(random.randint(1, 10))
print(random.randrange(0, 10, 2))
```

```
0.03718025439520911
10
4
```

```
li = [10, 20, 30, 40, 50]
print(li)
print(random.choice(li))
print(random.sample(li, 2))
random.shuffle(li)
print(li)
```

```
[10, 20, 30, 40, 50]
30
[50, 30]
[20, 50, 30, 40, 10]
```

네임스페이스(Namespace)

- 모듈 호출의 범위 지정
- 모듈 이름에 alias를 생성하여 모듈의 이름을 바꿔 사용

```
import random as rd

print(rd.random())
print(rd.randrange(0, 10, 2))
```

```
0.6960806655231452
2
```

- from 구문을 사용하여 모듈에서 특정 함수 또는 클래스만 호출

```
from random import random, randrange
```

```
from random import random, randrange

print(random())
print(randrange(0, 10, 2))
```

```
0.07750405361556745
4
```

- '*'을 사용하여 모듈 안에 모든 함수, 클래스, 변수를 가져옴

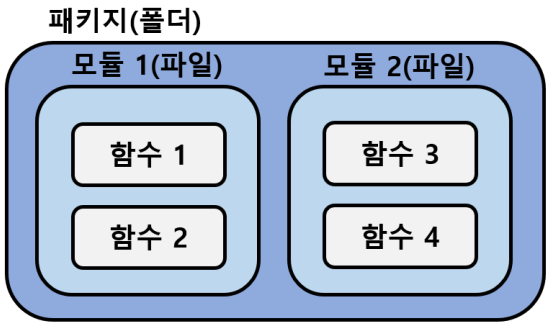
```
from random import *

print(random())
print(randrange(0, 10, 2))
```

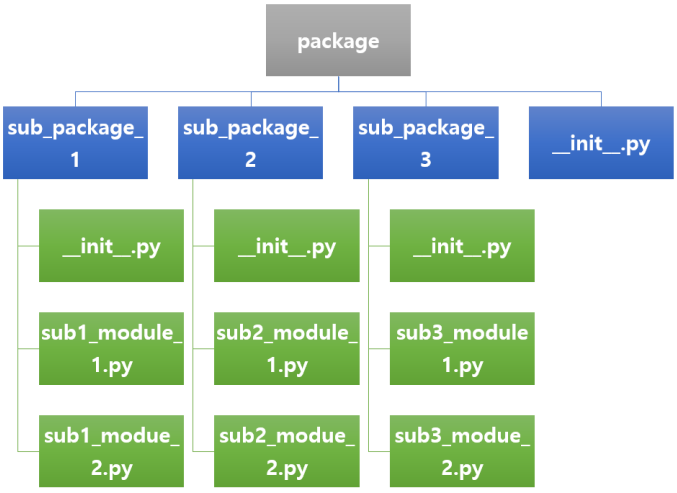
```
0.4802513180943273
8
```

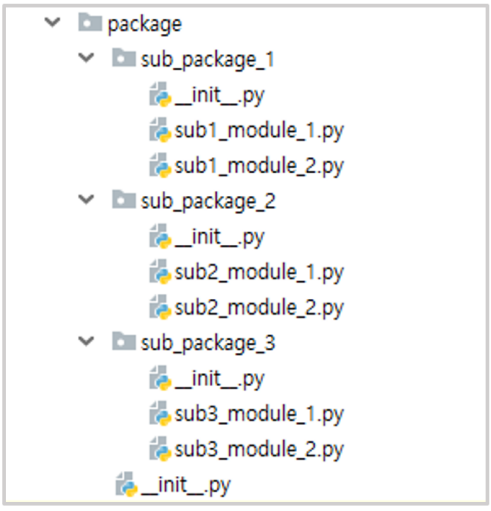
패키지(Packages)

- 패키지는 모듈의 집합
- 패키지 안에 여러 모듈이 존재
- 모듈을 주제별로 분리할 때 사용
- 디렉터리와 같이 계층적인 구조로 관리
- 모듈들이 서로 포함 관계를 가지며 거대한 패키지를 가짐
- 파이썬에서는 패키지가 하나의 라이브러리



패키지 구조 예제






```
!mkdir package
!mkdir package/sub_package_1
!mkdir package/sub_package_2
!mkdir package/sub_package_3
```

```
!ls package
```

```
sub_package_1  sub_package_2  sub_package_3
```

```
%%writefile package/sub_package_1/sub1_module_1.py
def print_module():
  print("sub_package_1/sub1_module_1")
```

```
Writing package/sub_package_1/sub1_module_1.py
```

```
%%writefile package/sub_package_1/sub1_module_2.py
def print_module():
  print("sub_package_1/sub1_module_2")
```

```
Writing package/sub_package_1/sub1_module_2.py
```

```
%%writefile package/sub_package_2/sub2_module_1.py
def print_module():
  print("sub_package_2/sub2_module_1")
```

```
Writing package/sub_package_2/sub2_module_1.py
```

```
%%writefile package/sub_package_2/sub2_module_2.py
def print_module():
  print("sub_package_2/sub2_module_2")
```

```
Writing package/sub_package_2/sub2_module_2.py
```

```
%%writefile package/sub_package_3/sub3_module_1.py
def print_module():
```

```
def print_module():
  print("sub_package_3/sub3_module_1")
```

```
Writing package/sub_package_3/sub3_module_1.py
```

```
%%writefile package/sub_package_3/sub3_module_2.py
def print_module():
  print("sub_package_3/sub3_module_2")
```

```
Writing package/sub_package_3/sub3_module_2.py
```

패키지 실행

- 정의한 패키지의 모듈 실행

```
from package.sub_package_1 import sub1_module_1, sub1_module_2
sub1_module_1.print_module()
sub1_module_2.print_module()
```

```
sub_package_1/sub1_module_1
sub_package_1/sub1_module_2
```

```
from package.sub_package_2 import sub2_module_1, sub2_module_2
sub2_module_1.print_module()
sub2_module_2.print_module()
```

```
sub_package_2/sub2_module_1
sub_package_2/sub2_module_2
```

```
from package.sub_package_3 import sub3_module_1, sub3_module_2
sub3_module_1.print_module()
sub3_module_2.print_module()
```

```
sub_package_3/sub3_module_1
sub_package_3/sub3_module_2
```

```
from package import *
sub1_module_1.print_module()
sub1_module_2.print_module()
sub2_module_1.print_module()
sub2_module_2.print_module()
sub3_module_1.print_module()
sub3_module_2.print_module()
```

```
sub_package_1/sub1_module_1
sub_package_1/sub1_module_2
sub_package_2/sub2_module_1
sub_package_2/sub2_module_2
sub_package_3/sub3_module_1
sub_package_3/sub3_module_2
```

패키지 구성 파일

- __init__.py
  - 파이썬 패키지를 선언하는 초기화 스크립트
  - 패키지에 대한 메타데이터에 해당하는 내용 포함
  - 파이썬의 거의 모든 라이브러리에 포함
  - 파이썬 버전 3.3 부터는 __init__.py 파일이 없어도 패키지로 인식
  - 파이썬 버전 3.3 밑의 하위 버전과 호환을 위해 __init__.py 파일 생성
  - __all__이라는 리스트형의 변수에 하위 패키지의 이름을 작성
  - `__all__=['sub_package_1', 'sub_package_2', 'sub_package_3']`

```
%%writefile package/__init__.py
__all__ = ['sub_package_1', 'sub_package_2', 'sub_package_3']
```

```
Writing package/__init__.py
```

```
%%writefile package/sub_package_1/__init__.py
__all__ = ['sub1_module_1', 'sub1_module_2']
```

```
Writing package/sub_package_1/__init__.py
```

```
%%writefile package/sub_package_2/__init__.py
__all__ = ['sub2_module_1', 'sub2_module_2']
```

```
Writing package/sub_package_2/__init__.py
```

```
%%writefile package/sub_package_3/__init__.py
__all__ = ['sub3_module_1', 'sub3_module_2']
```

```
Writing package/sub_package_3/__init__.py
```

```
!ls package
```

```
__init__.py  sub_package_1  sub_package_2  sub_package_3
```

```
!ls package/sub_package_1
```

```
__init__.py  __pycache__  sub1_module_1.py  sub1_module_2.py
```

- __main__.py
  - 패키지 자체를 실행하기 위한 용도
  - 패키지를 실행시키면 __main__.py 실행

```
%%writefile package/__main__.py
from sub_package_1 import *
from sub_package_2 import *
from sub_package_3 import *
```

```
if __name__ == '__main__':
  sub1_module_1.print_module()
  sub1_module_2.print_module()
  sub2_module_1.print_module()
  sub2_module_2.print_module()
  sub3_module_1.print_module()
  sub3_module_2.print_module()
```

```
Overwriting package/__main__.py
```

```
!python package
```

```
sub_package_1/sub1_module_1
sub_package_1/sub1_module_2
sub_package_2/sub2_module_1
sub_package_2/sub2_module_2
sub_package_3/sub3_module_1
sub_package_3/sub3_module_2
```
