10장 모듈

- 파이썬 코드를 작성한 후 파일로 저장하면
- 다른 코드에서도 이 파일의 변수, 함수, 클래스를 불러와 이용할 수가 있음
- 파이썬에서는 코드가 저장된 파일을 모듈(Module)
- 코드를 작성할 때 이미 만들어진 모듈을 활용하면 코드를 효과적으로 작성할 수 있음

10.1 모듈을 사용하는 이유

- 파이썬에서 모듈은 상수, 변수, 함수, 클래스를 포함하는 코드가 저장된 파일
- 모듈은 다른 모듈 에서도 불러서 실행할 수도 있고
- 파이썬(혹은 IPython) 콘솔이나 주피터 노트북에서 불러서 실행할 수 있음

- 모듈로 나누면 코드 작성과 관리가 쉬워짐
- 이미 작성된 코드를 재사용할 수 있음
- 공동작업이 편리해 짐

모듈 생성 및 호출

모듈 만들기

- 모듈을 만드는 방법
- 코드를 '모들이름.py'로 저장

- 내장 마술 명령어(magic command)인 '%%writefile'을 이용해 코드를 파일로 저장

```
%%writefile my_first_module.py
# File name: my_first_module.py

def my_function():
    print("This is my first module.")
```

- 파일이 잘 생성됐는지 확인

```
```

```
!cat my_first_module.py
```

모듈 불러오기

- 생성된 모듈을 사용하는 방법
- 모듈은 import를 이용
- import 모듈명

- 모듈을 임포트한 후에는 '모듈명.변수', '모듈명.함수()', '모듈명.클래스()'와 같은 형식으로 모듈에서 정의한 내용을 사용

```
!ls
```

- 만든 모듈(my_first_module)을 불러서 모듈의 함수를 실행

```
import my_first_module

my_first_module.my_function()
```

- 만든 my_first_module 모듈을 'import 모듈명'으로 불러왔고
- 모듈의 함수는 '모듈명.함수()'로실행

```
%%writefile my_area.py
# File name: my_area.py

PI = 3.14
def square_area(a): # 정사각형의 넓이 반환
    return a ** 2

def circle_area(r): # 원의 넓이 반환
    return PI * r ** 2
```

- 생성한 my_area 모듈을 import로 불러온 후 my_area 모듈에 있는 변수와 함수

```
import my_area    # 모듈 불러오기

print('pi =', my_area.PI) # 모듈의 변수 이용
print('square area =', my_area.square_area(5)) # 모듈의 함수 이용
print('circle area =', my_area.circle_area(2))
```

- 모듈을 불러온 이후에 '모듈명.변수','모둘명.함수()'의 형식으로 모듈 내의 변수와 함수를 호출

- 불러온 모듈에서 사용할 수 있는 변수, 함수, 클래스를 알고 싶다면 'dir(모듈명)'을 이용

```
dir(my_area)
```

- my_area 모듈에서 정의한 변수 PI와 함수 circle_area()와 square_area()가 있음

모듈을 불러오는 다른 형식

모듈의 내용 바로 선언

- 모듈 내의 변수와 함수를 호출하기 위해
- 'import 모듈명'으로 모듈을 불러와 '모듈명.변수', '모듈명.함수()'와 같은 형식을 이용하였음

- 이 경우 모듈 내에서 정의한 내용을 호출하기 위해 계속해서 모듈명을 써야 하므로 코드를 작성할 때 불편

- from 모듈명 import 변수명
- from 모듈명 import 함수명
- from 모듈명 import 클래스명

- 모듈 내에 있는 변수와 힘수, 그리고 클래스를 '모듈명.' 없이 '
- '변수','함수()','클래스()'처럼 직접 호출해서 이용 할 수 있음

- from 모듈명 import 변수명'형식으로 모듈에서 변수를 바로 불러와서 사용

```
from my_area import PI # 모듈의 변수 바로 불러오기

print('pi =', PI) # 모듈의 변수 이용
```

- 모듈 함수를 'from 모듈명 import 함수명' 형식으로 바로 불러서 사용

```
from my_area import square_area
from my_area import circle_area
```

```
print('square area =', square_area(5)) # 모듈의 함수 이용
print('circle area =', circle_area(2))
```

- 모듈의 변수와 함수를 이용하기 위해 'from 모듈명 import 변수명'과 'from 모듈명 import 함수명'으로
- 필요한 변수와 함수 개수만큼 각각 선언해서 이용

- 'from 모들명 import 변수명/함수명/클래스명'은
- 하나만 선언할 수 있는 것이 아니라 콤마(,)로 구분해서 여러 개를 선언할 수 있음

- 불러오려는 변수와 함수를 콤마를 이용해 여러 개 선언해서 이용

```
from my_area import PI, square_area, circle_area

print('pi =', PI) # 모듈의 변수 이용
print('square area =', square_area(5)) # 모듈의 함수 이용
print('circle area =', circle_area(2))
```

- 모듈의 모든 변수 , 함수, 클래스를 바로 모듈명 없이 바로 이용

- from 모듈명 import *

- 'from 모듈명 import *' 형식으로 선언해서 모듈 내의 모든 변수, 함수, 클래스를 불러옴

```
from my_area import *

print('pi =', PI) # 모듈의 변수 이용
print('square area =', square_area(5)) # 모듈의 함수 이용
print('circle area =', circle_area(2))
```

- 'from 모듈명 import *' 형식으로 선언하는 방법은 일일이 , '모듈명.'을 쓰지 않고
- 모듈 내의 변수, 함수, 클래스를 사용할 수 있어서 편리하기는 하지만
- 모듈을 두 개 이상 이용할 때는 주의가 필요

- my_module1과 my_module2 모듈을 작성

```
%%writefile my_module1.py
# File name: my_module1.py
```

```
def func1():
    print("func1 in  my_module1 ")

def func2():
    print("func2 in  my_module1 ")
```

```
%%writefile my_module2.py
# File name: my_module2.py

def func2():
    print("func2 in  my_module2 ")

def func3():
    print("func3 in  my_module2 ")
```

- 모듈 my_module1 에는 func1 함수와 func2 함수가 있고
- 모듈 my_module2에는 func2 함수와 func3 함수
- 함수 func2는 my_module1 에도 있고 my_module2에도 있음

- my_module1과 my_module2를 'from 모듈명 import *' 방법을 이용해 각 모듈의 함수를 불러온다면
- 모듈 my_module1과 모듈 my_module2에 공통으로 있는 func2는 어떤 것이 실행될까?

```
from my_module1 import *
from my_module2 import *

func1()
func2()
func3()
```

- 코드는 오류 없이 잘 수행
- 모듈 my_module1과 모듈 my_module2에만 각각 존재하는 func1과 func3은 불러오는데 문제가 없음

- 모듈 my_module1 과 모듈 my_module2에 모두 있는 func2 함수를 호출하면
- 나중에 선언해서 불러온 모듈의 함수가 호출됨

- 모듈 선언 순서와는 반대로 'from my_module2 import *'를 'from my_module1 import *'보다

```
from my_module2 import *
from my_module1 import *

func1()
func2()
func3()
```

- func2()를 호출했을 때 이번에는 my_module1 모듈에서 함수를 불러온 것

- 여러 모듈을 동시에 'from 모듈명 import *' 형식으로 선언해서 모듈 내의 변수, 함수, 클래스 이용할 때는 주의

▾ 모듈명을 별명으로 선언

- 'import 모듈명' 형식으로 모듈을 선언해서 이용할 경우
- '모듈명.변수', '모듈명.함수()' , '모듈명.클래스()'와 같은 형식으로 모듈에서 정의한 내용을 불러오는데
- 코드에서 매번 모듈명을 입력하기가 번거로움

- 'from 모듈명 import *' 방법으로 선언해서 '모듈명.'을 생략할 수도 있지만 여러 모듈을 임포트할 경우에는 주의가 필요

- import 모듈명 as 별명

- 코드를 작성할 때 '모듈명.변수'、'모듈명.함수()', '모듈명.클래스()' 대신
- '별명.변수','별명.함수()', '별명.클래스()'처럼 지정 할 수 있음

- from 모듈명 import 변수명 as 별명
- from 모듈명 import 함수명 as 별명
- from 모듈명 import 클래스명 as 별명

- 변수명, 함수명, 클래스명 대신 별명으로 이용

- 모듈을 불러올 때 모듈명 대신 별명을 사용

```
import my_area as area # 모듈명(my_area)에 별명(area)을 붙임
```

```
print('pi =', area.PI) # 모듈명 대신 별명 이용
print('square area =', area.square_area(5))
print('circle area =', area.circle_area(2))
```

- 모듈의 변수나 함수를 부를 때 모듈명 my_area 대신 별명인 area를 이용

- 'from 모듈명 import 변수명/함수명/클래스명 as 별명'

```
from my_area import PI as pi
from my_area import square_area as square
from my_area import circle_area as circle

print('pi =', pi) # 모듈 변수의 별명 이용
print('square area =', square(5)) # 모듈 함수의 별명 이용
print('circle area =', circle(2))
```

- 모듈 my_area의 내의 변수 PI의 이름을 pi로, 함수 이름 square_area와 circle_area를
- 각각 square와 circle로 별명을 붙이고 코드에서 별명을 이용

10.3 모듈을 직접 실행하는 경우와 임포트한 후 실행하는 경우 구분하기

- 'import 모듈명 '로 불러온 후에 모듈과 관련된 코드를 실행해야 결과를 확인

- 모듈을 만들 때는 함수나 클래스가 잘 작성됐는지 확인하기 위해 모듈 내에서 함수나 클래스를 직접 호출

- my_module_test1 모듈에는 입력된 숫자를 출력하는 함수가 있음
- 함수가 잘 작성됐는지 확인하기 위해 모듈 안에서 함수를 호출

```
%%writefile my_module_test1.py
# File name: my_module_test1.py

def func(a):
    print("입력 숫자:", a)

func(3)
```

- 저장된 모듈을 실행

```
%run my_module_test1.py
```

- 모듈 내에서 호출한 함수(fund3))가 실행된 것

```
import my_module_test1
```

- 'import 모듈명'을 통해 모듈을 불러오면 모듈 내의 코드를 실행
- 'import my_ module_test1'를 수행하면 my_module_test1 모듈의 코드가 수행

- my_module_test1 모듈 내 코드에서 func(3)는 작성한 함수가 잘 실행되는지를 확인하기 위해 작성해 놓은 것이지
- 모듈을 임포트할 때 실행하기 위해 작성한 코드가 아님

- 모듈을 마지막으로 저장 할 때는 테스트를 위해 작성한 코드는 삭제
- 테스트를 위해 작성한 코드를 일일이 삭제하는 것은 번거로움
- 모듈을 테스트하기 위해 직접 수행하는 경우와 임포트해서 사용하는 경우를 구분

- 모듈을 직접 수행하는 경우와 임포트해서 이용하는 경우를 구분해 사용

- if __name__ == "__main__":
- 〈직접 수행할 때만 실행되는 코드〉

- 코드를 모듈 파일에서 직접 수행하느냐 아니면 임포트해서 사용하느냐에 따라 코드를 다르게 수행
- 같은 모듈에서 코드를 직접 수행할 때만 'if __name__ == "__main__" : ' 안의 코드가 실행되고
- 임포트해서 사용하면 수행되지 않음

- 모듈의 테스트 코드를 수정한 후에 'my_module_test2.py'로 저장

```
%%writefile my_module_test2.py
# File name: my_module_test2.py

def func(a):
    print("입력 숫자:",a)

if __name__ == "__main__":
```

```
    print("모듈을 직접 실행")
    func(3)
    func(4)
```

- 'my_module_test2.py' 파일을 실행

```
%run my_module_test2.py
```

- 'if __name__ = "__main__" : ' 내의 코드가 실행

- my_module_test2 모듈을 임포트

```
import my_module_test2
```

- 실행 결과를 보면 아무것도 출력되지 않음
- 'if __name__ == "__main__" : ' 내에 작성한 코드는 임포트한 경우에는 실행되지 않기 때문

- 모듈에서 코드를 직접 수행하는 경우와 임포트해서 사용하는 경우를 구분해서 코드를 실행하기 위한 구조

- if __name__ == "__main__" :
- < 직점 수행할 때만 실행되는 코드>
- else:
- <임포트됐을 때만 실행되는 코드>

- my_moulde_test3.py로 저장

```
%%writefile my_module_test3.py
# File name: my_module_test3.py

def func(a):
    print("입력 숫자:",a)

if __name__ == "__main__":
    print("모듈을 직접 실행")
    func(3)
    func(4)
else:
    print("모듈을 임포트해서 실행")
```

- 저장된 모듈을 직접 실행

```
%run my_module_test3.py
```

- 직접 수행했을 때는 'if __name__ == "__main__" : ' 내의 코드가 수행

- my_module_test3 모듈을 임포트

```
import my_module_test3
```

- 모듈을 임포트했을 때 출력된 결과를 보면 해당 모듈을 임포트한 경우에만 실행하게 만든 코드만 실행
- if __name__ == "__main__" : ~ else' 구조를 이용하면
- 모듈을 직접 실행하는 경우와 임포트해서 실행하는 경우를 구분해서 코드를 실행

10.4 내장 모듈

- 임의로 숫자(난수)를 발생시키는 random 모듈,
- 날짜 및 시간 관련 처리할 수 있는 datetime 모듈 ,
- 연도/월/주 등 달력과 관련된 처리를 할 수 있는 calendar 모듈을 이용하는 방법

- 파이썬 표준 라이브러리에 관해 설명 한 사이트
- https://docs.python.org/3.9/library/

난수 발생 모듈

- 코드에서는 정해진 숫자가 아닌 실행할 때마다 임의의 숫자를 사용해야 할 때
- 임의의 숫자를 난수(random number)
- 파이썬에서는 random 모듈을 이용해 난수를 만들 수 있음

- random 모듈을 이용하려면 'import random'으로 random 모듈을 먼저 불러온 후에
- random 모듈의 함수를 이용

- import random
- random.random모듈함수()

- random 모듈에서 0~1 사이의 임의의 실수를 발생시키는 random() 함수

```
import random

random.random()
```

- 'import random'으로 random 모듈을 불러왔고
- 'random.random()으로 0 ~ 1 사이의 실수중에서 임의의 숫자를 생성

- random은 모듈명이고 random()은 random 모듈의 함수
- random.random()를 실행할 때마다 다른 값이 출력
- random.random() 코드를 실행하면 결과가 다를 것(난수를 발생시키는 함수의 특징)

- 특정 범위의 정수 안에서 임의의 정수를 발생시키려면 randint() 함수를 이용

```
import random

dice1 = random.randint(1,6) # 임의의 정수가 생성됨
dice2 = random.randint(1,6) # 임의의 정수가 생성됨
print('주사위 두 개의 숫자: {0}, {1}'.format(dice1, dice2))
```

- randrange() 함수
- 정해진 범위 안에서 특정 수만큼 차이가 나는 정수를 발생시킬 때 이용

```
import random

random.randrange(0,11,2)
```

- 0 ~ 10(11-1) 범위의 임의의 짝수가 발생한 것
- 홀수를 발생시킬 때나 10단위로 숫자를 발생시킬 때도 이용

```
import random

num1 = random.randrange(1, 10, 2) # 1 ~ 9(10-1) 중 임의의 홀수 선택
num2 = random.randrange(0,100,10) # 0 ~ 99(100-1) 중 임의의10의 단위
print('num1: {0}, num2: {1}'.format(num1,num2))
```

- 시퀀스(리스트, 튜플) 데이터에서 임의의 항목을 하나 선택할 때 이용할 수 있는 choice() 함수

- 선택할 수 있는 메뉴 전체를 리스트 변수로 지정한 후 choice() 함수를 실행해 나온 임의 결과를 선택

```
import random

menu = ['비빔밥', '된장찌개', '볶음밥', '불고기', '스파게티', '피자',
random.choice(menu)
```

- 여러 항목을 임의로 선택하려면 어떻게 할까요?
- 이때 사용하는 것이 sample()함수
- 시퀀스(리스트, 튜플)로 이뤄진 모집단 데이터에서 정해진 숫자만큼 임의의 인자를 중복 없이 선택하고 싶을 때 사용

- sample() 함수를 이용해 모집단(리스트 데이터)에서 지정한 개수만큼 중복 없이 임의의 인자를 반환

```
import random

random.sample([1, 2, 3, 4, 5], 2) # 모집단에서 두 개의 인자 선택
```

날짜 및 시간 관련 처리 모듈

- 날짜와 시간을 다룰 수 있는 파이썬 내장 모듈인 datetime 모듈
- datetime 모듈에는 날짜를 표현하는 date 클래스, 시간을 표현하는 time 클래스 , 날짜와 시간을 표현하는 datetime 클래스

- datetime 모듈을 이용하려면 먼저 'import datetime'으로 datetime 모듈을 불러와야 함
- datetime 모듈을 불러온 후에는 클래스에서 객체를 생성해 이용하는 방법이 있고
- 각 클래스의 클래스 메서드를 이용하는 방법

- datetime 모듈의 각 클래스에서 객체를 생성해 이용하는 방법

- import datetime
- date_obj = datetime.date(year, month, day)
- time_obj = datetime.time(hour, minute, second)
- datetime_obj = datetime.datetime(year, month, day, hour, minute, second)

- 생성한 객체를 이용해 각 클래스의 속성을 이용

- date 클래스에는 year, month, day의 속성이 있으며 time 클래스에는 hour, minute, second의 속성이 있음
- datetime 클래스는 date 클래스라 time 클래스의 모든 속성이 있음

- 내장 모듈 datetime을 이용하는 다른 방법은 객체를 생성하지 않고 각 클래스의 클래스 메서드를 이용 하는 것

- import datetime
- date_var = datetime.date.date_classmethod()
- time_var = datetime.time.time.classmethod()
- datetime_var = datetime .datetime .datetime_classmethod()

- 클래스 메서드를 이용하는 경우에도 각 클래스의 속성은 그대로 이용
- date 클래스, time 클래스, datetime 클래스를 이용하는 방법

- 날짜를 표현하는 date 클래스

```
import datetime

set_day = datetime.date(2021, 8, 2)
print(set_day)
```

- date 객체를 생성할 때 인자로 연도, 월, 일을 입력할 수 있음
- 생성된 date 객체는 print()로 입력한 날짜를 출력

- 연도, 월, 일을 각각 구하려면 date 클래스의 속성(year, month, day)을 이용

```
print('{0}/{1}/{2}'.format(set_day.year ,set_day.month,set_day.day ))
```

- datetime 모듈의 date 객체는 타입이 date로 그 객체끼리 빼기 연산을 할 수 있음
- 빼기 연산에서 앞의 객체의 날짜가 뒤의 객체의 날짜보다 더 나중이면 결과 날짜가 양수로 나오고
- 더 먼저이면 결과 날짜가 음수로 나옴
- 빼기 연산을 수행한 후에 결과의 데이터 타입은 timedelta로 바뀜

```
import datetime
```

```
day1 = datetime.date(2021, 4, 1)
day2 = datetime.date(2021, 7, 10)
diff_day = day2 - day1
print(diff_day)
```

- 빼기 연산을 수행한 후 데이터 타입이 어떻게 변경되는지 보기 위해 type() 함수를 이용
- 앞에서 생성한 객체와 빼기 연산을 한 결과의 데이터 타입을 확인

```
type(day1)
```

```
type(diff_day)
```

- date 데이터 타입이 빼기 연산을 수행한 후에는 timedelta 데이터 타입으로 변경

- 앞의 두 날짜차이 계산에서 날짜만출력하려면 timedelta 클래스 속성인 days를 이용

```
print("** 지정된 두 날짜의 차이는 {}일입니다. **".format(diff_day.days
```

- datetime 모듈의 date 클래스에는 오늘 날짜를 반환하는 클래스 메서드인 today()를 제공

- 오늘 날짜를 확인
- 클래스 메서드 today()는 인자 없이 호출

```
import datetime

print(datetime.date.today())
```

- 오늘과 특정 날짜의 차이를 알려면 빼기 연산을 수행

```
import datetime

today = datetime.date.today()
special_day = datetime.date(2020, 12, 31)
print(special_day - today)
```

- 시간(시각)과 관련된 처리할 수 있는 time 클래스

- tine 클래스에서 객체를 생성할 때 시, 분, 초를 인자로 입력

```
import datetime

set_time = datetime.time(15, 30, 45)
print(set_time)
```

- time 클래스의 속성(hour, minute, second)을 이용해 시, 분, 초를 각각 출력

```
print('{0}:{1}:{2}'.format(set_time.hour,set_time.minute,set_time.seco
```

- 날짜와 시간을 모두 다룰 수 있는 datetime 클래스

```
import datetime

set_dt = datetime.datetime(2021, 8, 2, 10, 20, 0)
print(set_dt)
```

- 'import datetime'의 datetime은 모듈 이름이고
- datetime()은 datetime 모듈 안에 있는 클래스 이름
- 모듈 이름과 클래스 이름이 같아서 혼동될 수 있는데 다른 것이니 사용할 때 주의

- datetime 클래스의 경우에도 속성을 이용해 연, 월 , 일 , 시, 분, 초를 각각 수행

```
print('날짜 {0}/{1}/{2}'.format(set_dt.year, set_dt.month, set_dt.day
print('시각 {0}:{1}:{2}'.format(set_dt.hour, set_dt.minute, set_dt.sec
```

- 현재 시각을 구하려면 datetime 클래스의 클래스 메서드인 now()를 이용

```
import datetime

now = datetime.datetime.now()
print(now)
```

- now()로 얻은 결과는 오늘 날짜(연, 월, 일)와 현재 시각(시, 분, 초)
- 초는 소수점 이하의 초까지 반환

- 각 클래스의 속성을 이용해 날짜와 시간을 출력했지만 날짜 및 시간 출력 양식을 지정해 출력

```
print("Date & Time: {:%Y-%m-%d, %H:%M:%S}".format(now))
```

- %Y, %m, %d는 각각 연도, 월 , 일을 나타내고 %H, %M, %S는 각각 시, 분, 초를 나타냄
- 이 값들은 '{:}' 안에 있어야 하며 일부만 사용할 수도 있음

- 날짜와 시각을 각각 출력

```
print("Date: {:%Y, %m, %d}".format(now))
print("Time: {:%H/%M/%S}".format(now))
```

- 'date' 클래스의 객체와 마찬가지로 datetime 클래스의 객체도 빼기 연산을 할 수 있음

- 현재 날짜 및 시각과 특정일의 날짜 및 시각의 차이를 구하기

```
now = datetime.datetime.now()
set_dt = datetime.datetime(2020, 12, 1, 12, 30, 45)

print("현재 날짜 및 시각:", now)
print("차이:", set_dt - now)
```

- 결과에서 차이가 음수로 나온 것은 set_dt가 now보다 과거이기 때문

- datetime 모듈의 객체를 'import 모듈명'을 수행한 후에 사용했지만
- 'from 모듈명 import 클래스명 ' 방법을 이용하면 모듈명 없이 바로 클래스 이름이나 클래스 메서드 이름으로 이용할 수 있음.

```
from datetime import date, time, datetime

print(date(2021, 7, 1))
```

```
print(date.today())
```

```
print(time(15, 30, 45))
```

```
print(datetime(2021, 2, 14, 18, 10, 50))
```

```
print(datetime.now())
```

달력 생성 및 처리 모듈

- 파이썬 내장 모듈인 calendar 모듈을 이용해 다양한 형태로 달력을 생성해 출력하고
- 날짜와 관련된 정보(연도, 월, 주)를 구하는 방법

- calendar 모듈은 달력과 관련된 클래스와 함수를 제공
- calendar 모듈의 주요 함수의 사용법

- calendar 모듈에서 요일(weekday)을 지정할 때 숫자 0(월요일) ~ 6(일요일)을 이용해도 되지만 정의된 상수를 이용

- calendar 모듈을 이용하려면 먼저 'import calendar'로 모듈을 불러와야 함
- calendar() 함수를 이용해 지정한 연도의 전체 달력을 출력하는 방법

```
import calendar

print(calendar.calendar(2021))
```

- calendar() 함수의 기본적인 달력 출력 양식은 달을 3열로 출력

- 달의 출력 양식을 변경하고 싶다면 calendar()함수에 'm=숫자' 인자를 추가
- 달력 출력 형식을 4열로 지정

```
print(calendar.calendar(2021, m=4))
```

- 지정한 연도의 특정 월만 표시하려면 month() 함수를 이용

```
print(calendar.month(2021,9))
```

- 연도와 월을 지정해 그달 1일이 시작하는 요일과 그달의 날짜 수를 알고 싶다면 monthrange()함수를 이용

```
calendar.monthrange(2021,2)
```

- 결과로 두 개의 숫자가 반환
- 첫 번째 숫자는 해당 월의 1일의 요일에 해당하는 숫자{월요일 ~ 일요일을 의미하는 0〜 6 중 하나가 반환됨)이고

- 두 번째 숫자는 해당 월의 날짜 수

- 출력된 달력을 보면 일주일의 시작 요일이 월요일인 것을 알 수 있음
- 달력에서 일주일의 시작 요일을 구하려면 firstweekday() 함수를 실행

```
calendar.firstweekday()
```

- 결과로 0이 출력돼 달력에서 일주일의 시작 요일이 월요일로 지정된
- 시작 요일을 지정하려면 setfirstweekday(weekday) 함수를 이용
- 시작 요일을 일요일로 지정하려면 weekday 에는 6(혹은 calendar.SUNDAY)을 입력

- setfirstweekday (calendar.SUNDAY) 로 달력에서 일주일의 시작 요일을 일요일로 바꾸고 달력을 출력

```
calendar.setfirstweekday(calendar.SUNDAY)
print(calendar.month(2021,8))
```

- calendar 모듈 함수는 weekday() 함수
- 해당 날짜의 요일을 반환

```
print(calendar.weekday(2021, 8, 1))
```

- isleap(year) 함수를 이용해 어떤 연도가 윤년인지를 확인

```
print(calendar.isleap(2020))
print(calendar.isleap(2021))
```

출력 결과에서 2020년은 윤년인 것을 알 수 있음

10.5 패키지

- 파이썬에서 모듈은 코드가 저장된 파일
- 어떤 기능을 구현할 때 하나의 모듈로 구성하 기보다는 여러 개의 모듈로 구현하는 경우가 많음
- 여러 모듈을 체계적으로 모아서 꾸러미로 관리하면 편리
- 파이썬에서는 이런 꾸러미를 패키지(Package)
- 파이썬 패키지는 여러 모듈을 폴더로 묶어서 계층적으로 관리

- 복잡하고 규모가 큰 프로그램을 작성할 때는 각 모듈을 묶어서 패키지로 만들면 좀 더 효율적으로 코드를 관리

패키지의 구조

- 파이썬 패키지는 폴더 구조로 돼 있으며 각 폴더에는 '__init__.py'라는 특별한 파일
- '__init__.py' 파일은 해당 폴더가 패키지의 일부인 것을 알려주는 역할
- '__init__.py' 파일은 패키지를 초기화하는 코드를 넣을 수도 있고 아무 코드도 없는 빈 파일일 수도 있음
- 패키지를 만들 때 '__init__.py' 파일이 없어도 되지만 하위 호환성을 고려하면 '__init__.py'파일을 포함하는 것이 좋음

패키지 만들기

```
mkdir /content/packages
```

```
pwd
```

```
mkdir /content/packages/image
```

```
mkdir /content/packages/image/io_file
```

- 빈 '__init__.py'파일을 생성

```
%%writefile /content/packages/image/__init__.py
# File name: __init__.py
```

```
%%writefile /content/packages/image/io_file/__init__.py
# File name: __init__.py
```

- imgread 모듈에는 pngread() 함수와 jpgread() 함수를 만듬

```
%%writefile /content/packages/image/io_file/imgread.py
# File name: imgread.py

def pngread():
```

```
    print("pngread in imgread module")

def jpgread():
    print("jpgread in imgread module")
```

패키지 사용하기

- 패키지 모듈을 이용하려면 'import 패키지 내 모듈명'으로 선언
- 패키지명에서 시작해 모듈명까지 구분하기 위해 패키지명, 폴더명, 모듈명 사이에 온점(.)을 입력

- 패키지 폴더 안에 바로 모듈이 있다면 'import 패키지명.모들명'으로 모듈을 호출하고
- 패키지와 모듈 사이에 폴더가 있다면 'import 패키지명.폴더명.모둘명'으로 모듈을 호출

- 패키지에서 모듈 내의 함수를 호출하는 방법

```
import image.io_file.imgread # image 패키지 io_file 폴더의 imgread 모

image.io_file.imgread.pngread() # imgread 모듈 내의 pngread() 함수 호
image.io_file.imgread.jpgread() # imgread 모듈 내의 jpgread() 함수 호
```

- 'from A import B' 형식을 이용하면 패키지 안에 있는 모듈 내 함수를 더 간단하게 호출
- 첫 번째 방법은 'from A import B'에서
- A에는 '패키지명[.폴더명]'을 입력하고
- B에는 사용할 모듈명을 입력

```
from image.io_file import imgread

imgread.pngread()
imgread.jpgread()
```

- 'from A import B'에서 A에는 '패키지명[.폴더명].모듈명'을 입력하고 B는 사용할 항수명을 입력

```
from image.io_file.imgread import pngread

pngread()
```

- imgread 모듈의 모든 함수를 바로 부르려면 'from 패키지명[.폴더명].모듈명 import *'를 이용

```
from image.io_file.imgread import *

pngread()
jpgread()
```

- 'from 패키지명[.폴더명].모듈명 import *'를 이용해
- 전체 함수를 임포트한 후에는 모듈 내의 모든 함수를 바로 불러서 이용할 수 있음

- 전체 함수를 임포트하지 않고 필요한 함수만 선택 적으로 임포트할 수도 있음

```
from image.io_file.imgread import pngread, jpgread

pngread()
jpgread()
```

- 패키지의 모듈 이름에 별명을 붙여서 모듈을 이용할 수 있음

```
from image.io_file import imgread as img

img.pngread()
img.jpgread()
```

- 패키지의 모듈 안에 있는 변수, 함수 , 클래스에도 별명을 붙일 수 있음
- 모듈 내의 함수에 별명을 붙여서 호출

```
from image.io_file.imgread import pngread as pread
from image.io_file.imgread import jpgread as jread

pread()
jread()
```

10.6 정리

- 모듈이 무엇인지 살펴보고 모듈의 생성 방법과 활용법
- 파이썬 내장 모듈인 난수 발생 모듈(random)과 날짜 및 시간 관련 모듈(datetime), 달력 관련 모듈(calendar)의 사용법
- 패키지의 구조와 생성 및 사용 방법