



모름

모듈?

- 다른 파이썬 프로그램이 사용할 수 있도록 함수, 변수 또는 클래스들을 모아서 만든 파이썬 파일
- 일반적으로 비슷한 기능을 하는 함수 또는 큰 기능을 수행하는 데 필요한 일련의 함수와 데이터 포함
- 장점
 - » 코드의 재사용
 - » 코드를 이름 공간으로 구분하고 관리

모듈 정의와 사용

■ 모듈 정의

- » mod1.py 파일을 만들고 아래의 코드를 작성 후 저장

```
# mod1.py  
  
def sum(a, b):  
    return a + b
```

■ 모듈 사용

- » mod1.py 파일이 저장된 장소에서 파이썬 대화형 프로그램 실행
- » import 구문으로 모듈 로딩
- » import 구문의 위치는 자유롭게 선택 (파일 상단, 메서드 내부 등)

```
import 모듈이름
```

```
>>> import mod1  
>>> print(mod1.sum(3,4))  
7
```

모듈 정의와 사용

- import 실행 시 모듈 검색 위치
 - » 현재 작업 디렉터리
 - » PYTHONPATH 환경 변수에 등록된 위치
 - » 표준 라이브러리 디렉터리

```
import sys
```

```
sys.path
```

```
['d:\\workspace\\python',  
'c:\\Users\\          \\vscode\\extensions\\ms-toolsai.jupyter-2020.11.392013122\\pythonFiles',  
'c:\\Users\\          \\vscode\\extensions\\ms-toolsai.jupyter-2020.11.392013122\\pythonFiles\\lib\\python',  
'C:\\ProgramData\\Miniconda3\\envs\\instructor\\python37.zip',  
'C:\\ProgramData\\Miniconda3\\envs\\instructor\\DLLs',  
'C:\\ProgramData\\Miniconda3\\envs\\instructor\\lib',  
'C:\\ProgramData\\Miniconda3\\envs\\instructor',  
,  
'C:\\Users\\          \\AppData\\Roaming\\Python\\Python37\\site-packages',  
'C:\\ProgramData\\Miniconda3\\envs\\instructor\\lib\\site-packages',  
'C:\\ProgramData\\Miniconda3\\envs\\instructor\\lib\\site-packages\\win32',  
'C:\\ProgramData\\Miniconda3\\envs\\instructor\\lib\\site-packages\\win32\\lib',  
'C:\\ProgramData\\Miniconda3\\envs\\instructor\\lib\\site-packages\\Pythonwin',  
'C:\\ProgramData\\Miniconda3\\envs\\instructor\\lib\\site-packages\\IPython\\extensions',  
'C:\\Users\\          \\ipynb']
```

모듈 정의와 사용

- 모듈 사용 (계속)

- » 모듈 이름 없이 함수 사용 가능

```
from 모듈이름 import 함수
```

```
>>> from mod1 import sum
>>> sum(3, 4)
7
```

- » 여러 개의 함수를 동시에 import

```
from 모듈이름 import 함수1, 함수2, ...
```

```
from mod1 import sum, safe_sum
```

또는

```
from 모듈이름 import *
```

```
from mod1 import *
```

선택적 import

- 모듈을 import 할 때 선택적 실행 관리 가능
 - » `__name__`은 파이썬 파일을 직접 실행했을 때 사용되는 변수
 - › `python abc.py`로 실행한 경우 `__name__`에는 `"__main__"`이 저장됨
- 아래 구문은 파일을 직접 실행한 경우에만 실행되도록 조건 처리
 - » 임포트 한 경우에는 실행되지 않음

```
if __name__ == "__main__":  
    print(safe_sum('a', 1))  
    print(safe_sum(1, 4))  
    print(sum(10, 10.4))
```

```
>>> import mod1  
>>>
```

클래스나 변수를 포함한 모듈

- 클래스 또는 변수를 포함한 모듈 생성

```
PI = 3.141592

class Math:
    def solv(self, r):
        return PI * (r ** 2)

def sum(a, b):
    return a+b

if __name__ == "__main__":
    print(PI)
    a = Math()
    print(a.solv(2))
    print(sum(PI , 4.4))
```

```
C:\Python>python mod2.py
3.141592
12.566368
7.541592
```

직접 실행한 경우 실행 결과 출력

```
C:\Python>python
>>> import mod2
>>>
```

import 한 경우 실행되지 않음

클래스나 변수를 포함한 모듈

- 모듈에 포함된 변수, 클래스, 함수 사용하기

```
>>> print(mod2.PI)  
3.141592
```

```
>>> a = mod2.Math()  
>>> print(a.solv(2))  
12.566368
```

```
>>> print(mod2.sum(mod2.PI, 4.4))  
7.541592
```




패키지

패키지

- 패키지는 파이썬 모듈을 계층적으로 관리하는 도구
 - » 디렉터리와 파이썬 모듈로 구성됨
 - » 각 디렉터리와 모듈은 .(dot)를 사용해서 연결
- 패키지 예제

```
game/  
  __init__.py  
  sound/  
    __init__.py  
    echo.py  
    wav.py  
  graphic/  
    __init__.py  
    screen.py  
    render.py  
  play/  
    __init__.py  
    run.py  
    test.py
```

패키지 사용

- 디렉터리와 모듈을 결합한 형태로 임포트해서 사용

```
>>> import game.sound.echo
>>> game.sound.echo.echo_test()
echo
```

- 디렉터리와 모듈을 분리해서 임포트 구문 작성

```
>>> from game.sound import echo
>>> echo.echo_test()
echo
```

- 메서드를 임포트해서 사용

```
>>> from game.sound.echo import echo_test
>>> echo_test()
echo
```


__init__.py 파일

- 특정 디렉터리가 패키지의 일부임을 표시하는 파일
 - » 패키지가 import 될 때 자동으로 실행
- 모듈을 와일드카드(*) 형태로 임포트 할 때 __init__.py 파일에 __all__이라는 변수를 선언하고 임포트 할 수 있는 모듈을 정의해야 함

```
# C:/Python/game/sound/__init__.py
__all__ = ['echo']
```

- » 단, 위 규칙은 모듈을 와일드카드로 임포트 할 경우에만 해당되며 메서드에 대한 와일드카드 사용은 항상 가능

예외 처리

The background features a large, flowing, abstract shape in shades of green and white. The shape starts as a thin, light green line on the left, curves upwards and to the right, then downwards and to the right, and finally curves back towards the bottom right corner. The shape has a soft, ethereal quality with varying opacities and gradients. A solid dark green horizontal bar is positioned at the very bottom of the image.

예외

- 프로그램 실행 중 다양한 형태의 오류 발생

```
>>> f = open("나없는파일", 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '나없는파일'
```

```
>>> 4 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> a = [1,2,3]
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- 파이썬은 exceptions 모듈에 포함된 다양한 예외 클래스를 사용해서 관리.
 - » 최상위 예외 클래스 → BaseException
 - » ArithmeticError, ZeroDivisionError, IOError 등 다양한 종류의 예외 클래스 지원

예외 처리

- 예외 처리는 프로그램 실행 중 발생하는 오류를 적절하게 관리하는 기법
- 파이썬은 try, except, else, finally 를 사용하는 예외 처리 구문 지원

```
try:  
    예외 발생 가능성이 있는 문장  
except 예외 종류:  
    예외 처리 문장  
except (예외 종류1, 예외 종류2):  
    예외 처리 문장  
except 예외 종류 AS 변수:  
    예외 처리 문장  
else:  
    예외가 발생하지 않은 경우 실행되는 문장  
finally:  
    예외 발생 여부와 관계 없이 실행되는 문장
```

- 여러 개의 예외를 처리하는 경우 예외 클래스 상속 관계의 하위 클래스로부터 상위 클래스의 순서로 작성 → 좁은 범위의 예외로부터 넓은 범위의 예외로 확장 하면서 작성

예외 처리

■ 오류 처리 구문 형식

» try, except 만 사용

```
try:  
    ...  
except:  
    ...
```

» 발생 오류만 포함한 except

```
try:  
    ...  
except 발생 오류:  
    ...
```

» 발생 오류와 오류 메시지 변수까지 포함한 except

```
try:  
    ...  
except 발생 오류 as 오류 메시지 변수:  
    ...
```


예외 처리

■ 오류 처리 구문 형식 (계속)

» try ... else (예외가 발생하지 않을 경우 실행할 구문)

```
try:
    f = open('foo.txt', 'r')
except FileNotFoundError as e:
    print(str(e))
else:
    data = f.read()
    f.close()
```

» try ... finally (예외 발생 여부와 상관 없이 실행되는 코드 구성)

```
f = open('foo.txt', 'w')
try:
    # 무언가를 수행한다.
finally:
    f.close()
```

예외 처리

- 오류 처리 구문 형식

- » 다중 예외 처리 구문

```
try:
    a = [1,2]
    print(a[3])
    4/0
except ZeroDivisionError as e:
    print(e)
except IndexError as e:
    print(e)
```

- » 오류 회피 → 발생한 오류에 대해 특별한 처리 없이 정상 흐름으로 복귀

```
try:
    f = open("i_am_not_exist.txt", 'r')
except FileNotFoundError:
    pass
```

강제 예외 발생

- 파이썬은 `raise` 명령을 사용해서 예외를 상위 예외처리 영역으로 전달

```
def raise_error_func():  
    # raise NotImplementedError  
    raise NotImplementedError('예외 관련 전달인자')
```

```
try:  
    raise_error_func()  
except:  
    print('오류가 발생했으며 정상적으로 처리했습니다.')
```

오류가 발생했으며 정상적으로 처리했습니다.

사용자 정의 예외

- 내장 예외로 표현할 수 없는 예외 상황을 처리하는 경우 사용자 정의 예외 클래스를 만들어서 사용
 - » Exception 또는 Exception 하위 클래스로부터 상속된 클래스를 통해 정의

```
class NegativeDivisionError(Exception): # Exception 상속 사용자 정의 예외 클래스
    def __init__(self, value):
        self.value = value

    def PositiveDivide(a, b):
        if(b < 0):
            raise NegativeDivisionError(b)
        return a / b

try:
    ret = PositiveDivide(10, -3)
    print('10 / 3 = {0}'.format(ret))
except NegativeDivisionError as e:
    print('Error - Second argument of PositiveDivide is ', e.value)
except ZeroDivisionError as e:
    print('Error - ', e.args[0])
except :
    print("Unexpected exception!")
```

assert

- 주로 개발 과정에서 사용하는 구문

```
assert 조건식, 관련 데이터
```

```
if __debug__:
    if not 조건식:
        raise AssertionError(관련 데이터)
```

- » 조건식이 참일 경우 관련 데이터를 전달인자로 사용하는 AssertionError 발생
- » 실행 시 -O (최적화 옵션)을 설정하면 __debug__를 False로 만들어서 실행되지 않음