파이썬 기초: 모듈과 파일처리

#모듈/패키지/라이브러리 #파일 I/O #예외처리

목차

모듈(Module)이란?

모듈(Module)의 필요성?

모듈(Module)?

여러 함수, 변수, 클래스 등을 하나의 파일로 묶어 놓은 것

```
1       def func1():
            ...
2
3       def func2():
4           ...
5
6       ...

...     def func100():
4000        ...
```

프로젝트의 규모가 커짐에 따라 코드 양 증가.

이것을 전부 하나의 파일에서 관리?..

수정할 기능 찾기도 어렵고, 유지보수도 어려움
또한, 다른 곳에서 재사용하기도 불편하다

모듈(Module)의 필요성?

모듈(Module)?

여러 함수, 변수, 클래스 등을 하나의 파일로 묶어 놓은 것
(모듈 = 하나의 파일)

```
product.py → 상품 관련 함수/클래스
order.py → 주문 처리 관련 함수/클래스
user.py → 회원 관련 함수/클래스
payment.py → 결제 처리 관련 함수/클래스
```

ex) user.py → 로그인, 회원가입, 회원탈퇴 등등 함수

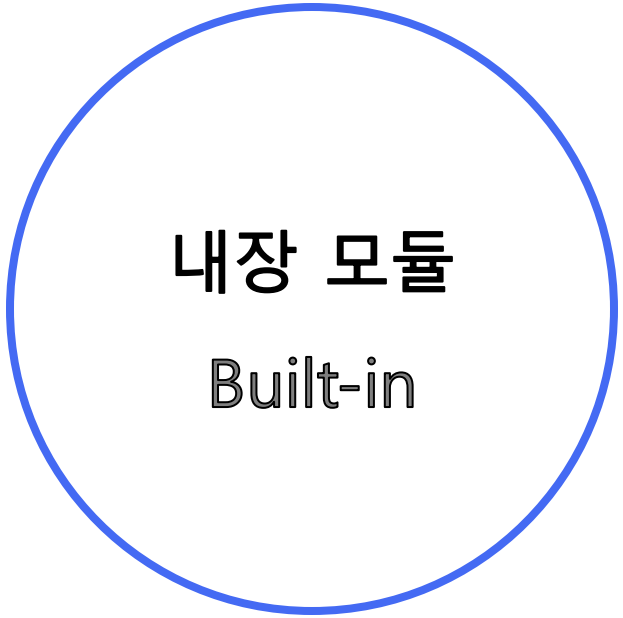
특정 기능(주제)마다 모듈을 분리

유지보수와 재사용에 용이

모듈 사용하기

모듈(Module)

모듈의 종류



파이썬이 기본적으로 제공하는 모듈

사용자가 직접 만든 .py 파일

모듈 불러오기

모듈은 다른 파일이므로, 해당 모듈의 기능을 사용하기 위해 불러와야 한다.

import 키워드를 사용하여 모듈 불러오기

```
import random
```

random 이라는 모듈을 불러오기

**보통 해당 모듈을 사용하기 위한 설정 단계로 코드의 최상단에 위치*

모듈 사용하기

모듈을 불러온 뒤, 모듈에 있는 기능(함수, 변수, 클래스) 사용하기

모듈이름.함수/변수/클래스 형태로 사용

```
import random

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

random 이라는 모듈을 불러오고
해당 모듈에 있는 randrange() 함수 호출

내장 모듈(Built-in)

기능 하나하나 직접 구현하는 것은 어려움.
파이썬이 별도의 설치 없이 사용 가능한 유용한 기능들이 담긴 모듈들

분류	모듈 이름
수학/통계 관련	math, statistics, random
날짜/시간 관련	datetime, time
파일/시스템 관련	os, sys, pathlib
데이터 처리 관련	json, csv, re
네트워크 관련	socket, http, urllib

모듈(Module)

내장 모듈(Built-in): math 모듈

수학 연산과 관련된 기능들 제공

```python
import math

print(math.pi) # 원주율
print(math.e) # 자연상수

print(math.sqrt(16)) # 제곱근
print(math.pow(2, 3)) # 거듭제곱
print(math.ceil(3.1)) # 올림
print(math.floor(3.9)) # 내림
```

상수

함수

```
3.141592653589793
2.718281828459045
4.0
8.0
4
3
```

내장 모듈(Built-in): random 모듈

난수(무작위 수)를 생성해주는 기능을 제공

```python
import random

print(random.random()) # 0~1 사이 난수
print(random.randint(1, 10)) # 1~10 사이 정수 난수
print(random.choice(['가위', '바위', '보'])) # 리스트에서 무작위 선택
print(random.sample(range(1, 46), 6)) # 목록에서 k개 샘플 뽑기
```

*실행 결과는 매번 달라진다

```
0.6248983979987514
6
보
[15, 8, 40, 38, 5, 43]
```

내장 모듈(Built-in): datetime 모듈

날짜와 시간을 다룰 때 가장 기본적인 모듈

```python
import datetime

now = datetime.datetime.now() # 현재 날짜와 시간
print("현재 시각:", now)

today = datetime.date.today() # 오늘 날짜
print("오늘 날짜:", today)

future = today + datetime.timedelta(days=7) # 시간 차(간격) 계산
print("일주일 뒤:", future)
```

```
현재 시각: 2025-10-05 01:40:46.172248
오늘 날짜: 2025-10-05
일주일 뒤: 2025-10-12
```

사용자 정의 모듈(User-defined)

사용자가 원하는 내용이 담긴 모듈 별도로 정의할 수 있다 (.py 파일)

- 함수, 변수, 클래스 등이 포함될 수 있음

```python
# main.py
import my_module
```

```python
# my_module.py

var1 = 10
var2 = 20

def my_func():
    ...
```

모듈(Module)

사용자 정의 모듈 만들기 (1)

.py 파일을 생성 후, 함수/변수/클래스를 생성한다.

모듈

```python
# cal.py

def plus(a, b):
    c = a + b
    return c
```

사용자 정의 모듈 만들기 (2)

다른 파일에서 만든 .py을 import를 통해 불러온다.

```python
# cal.py

def plus(a, b):
    c = a + b
    return c
```

```python
# main.py
import cal

print(cal.plus(3, 4))
```

그리고 해당 모듈에 있는 함수/변수/클래스를 불러와 사용

cal.py에 작성된 코드

main.py에 작성된 코드

모듈(Module)

모듈을 불러오는 또 다른 방법

- 일반적인 import 구문

  import 모듈이름

- 모듈에서 특정 함수/변수/클래스만 불러오기

  from 모듈이름 import 함수/변수/클래스

  여러 개를 불러올 때는 콤마(,)로 구분

- 모듈의 모든 함수/변수/클래스 불러오기

  from 모듈이름 import *

- 별칭(as) 붙이기

  import 모듈이름 as 별칭

모듈(Module)

불러오는 방법마다 기능을 사용하는 방법이 다르다

import 방식

모듈이름.함수/변수/클래스

```
import math

print(math.factorial(5))
```

from-import 방식

모듈이름 없이 함수/변수/클래스 이름으로 호출

```
from math import factorial

print(factorial(5))
```

동일한 코드

모듈과 패키지(Package)

패키지(Package)의 필요성

패키지(Package)

여러 개의 모듈을 하나의 디렉토리(폴더)로 묶어 관리하는 단위(= 모듈들의 모음)

프로젝트

```
├── user.py
├── user_auth.py
├── user_profile.py
├── order.py
├── order_db.py
├── order_payment.py
├── utils.py
└── ...
```

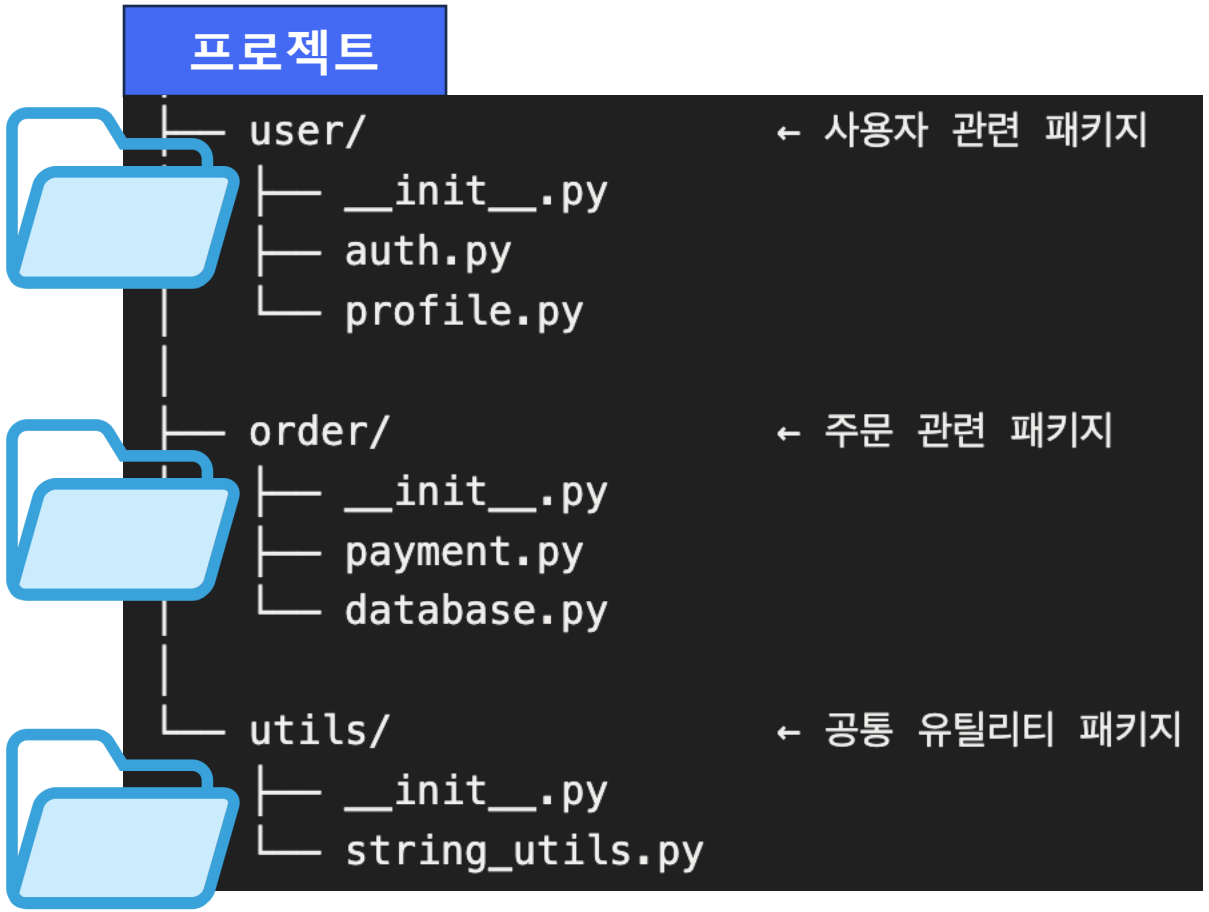
프로젝트의 규모가 커짐에 따라 모듈의 개수가 많아져서 관리가 어려움

모듈이 수십, 수백개라면?..

패키지(Package)의 필요성

패키지(Package)

관련된 모듈들을 폴더 단위로 그룹화한 것!



기능별 구조화를 통해
유지보수와 재사용성 증가

패키지(Package)의 필요성

패키지(Package) 속 모듈 사용하기

다음과 같은 구조가 있을 때 사용 방법은?

user 폴더(패키지)　　cal.py 파일(모듈)　　함수 plus()

패키지(Package)의 필요성

패키지(Package) 속 모듈 사용하기

폴더이름.모듈이름 형태로 import수행

import 방식

```
import user.cal

print(cal.plus(3, 4)) # 7
```

from-import 방식

```
from user.cal import plus

print(plus(3, 4)) # 7
```

user 폴더 안에 있는 cal 모듈을 호출

라이브러리(Library)

라이브러리(Library)란?

여러 패키지의 모음으로, 특정 분야 전체를 지원하는 도구 세트!

프로젝트가 커짐에 따라 여러 분야를 다루게 되면
패키지 하나로는 부족하다

그렇다고 직접 다 구현?...

다른 개발자들이 만들어 놓은 도구로써,
특정 작업을 수행할 때 사용하는, 재사용 가능한
코드의 집합

패키지보다 상위 개념 또는 유사한 개념

라이브러리(Library)와 패키지(Package)의 혼동?

pandas 라이브러리?... pandas 패키지?...

- **패키지**는 파이썬의 구조 관점(형태)에서 기술적 용어
  - 여러 모듈이 담겨 있는 폴더 구조

- **라이브러리**는 기능과 목적 관점에서 개념적 용어
  - 관련 기능을 묶어 제공하는 도구 세트

Pandas

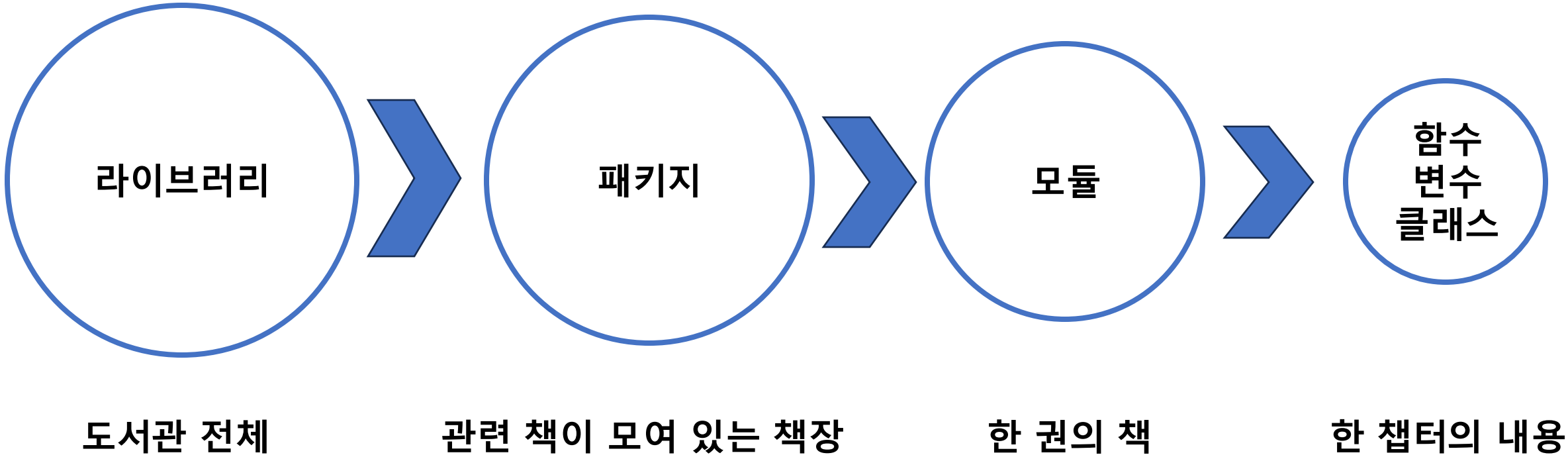
구조로는 여러 하위 모듈을 가진 "패키지"

기능적으로는 "데이터 분석용 라이브러리"

관련 기능을 제공받아서 가져다 쓸 수 있는 도구 모음의 의미로 관례적으로 라이브러리 용어를 주로 사용

총 정리

라이브러리, 패키지, 모듈, 함수?



파이썬의 대표적인 라이브러리

대표적인 파이썬의 외부 라이브러리

외부 라이브러리 설치하기 (PyPI, pip)

외부 라이브러리는 별도로 설치를 해야함

"pip"

Python 패키지 관리자

외부 라이브러리 설치

pip 명령어를 통해 외부 라이브러리 관리

터미널(명령 프롬프트)에서 명령어 입력

명령어	설명
pip install <라이브러리>	라이브러리 최신 버전 설치
pip install <라이브러리>==<버전>	라이브러리 특정 버전으로 설치
pip install --upgrade <라이브러리>	라이브러리 최신 버전으로 업데이트
pip list	설치된 라이브러리 목록 확인
pip show <라이브러리>	설치한 라이브러리의 정보 확인
pip uninstall <라이브러리>	설치된 라이브러리 삭제

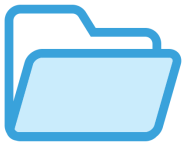
파일 처리(File I/O)

파일(File)이란?

파일(File)?

바이트의 묶음 저장된 의미가 있는 데이터 한 덩어리(의미가 있는 정보를 구분하는 단위)

파일 폴더가 아닌 파일, 실행파일/문서/압축 파일이 속한다



폴더 폴더와 파일을 계층형 구조로 보관할 수 있도록 만든다

파일(File)이란?

파일의 종류

종류	설명	파일 예시	사람이 읽을 수 있나?
실행 가능한 파일	기계를 제어하는 내용이 담긴 파일	.exe .bat	No
이진 파일	0과 1로 인코딩된 파일 형태 (비문자 형태)	.jpg .png .mp4 .avi	No
단순 텍스트	특정 형식 없이 작성된 파일	.txt	Yes
형식화된 텍스트	특정 형식에 맞게 작성된 파일 (사람과 컴퓨터 모두 읽고 처리 가능)	.html .json .csv	Yes

파일(File)이란?

파이썬으로 파일 다루기?

파일 관련 라이브러리를 통해 파일 읽고/쓰기/이동/삭제 등등 조작에 유용함

1. 여러 파일의 이름을 한번에 변경
2. 파일을 자동으로 분류해서 폴더에 정리
3. 파일에서 필요한 부분만 추출
4. 파일 이동이나 정리, 오래되거나 큰 파일 찾기
5. 폴더 통째로 압축파일로 백업하기
6. 중복 파일 찾기
7. 파일 병합

os, sys, pathlib, shutil

...

파일 입출력(open, close)

파일 열기(open)

파일을 코드 상으로 열어서(불러와서) 파일 관련 처리를 수행

```
f = open("filename.txt")

# 파일 관련 처리 수행

f.close()
```

f = open(파일 경로 문자열)

- 파일 객체를 반환(읽기 또는 쓰기 관련 작업이 가능한 객체)
- 파일을 열 때, 여러가지 모드로 열 수 있다. (Default는 읽기 모드)
  - 읽기 전용인데 파일이 존재하지 않으면 FileNotFoundError 발생
  - 읽기 모드에서는 오직 파일 읽기 가능
  - 쓰기 모드에서는 오직 파일 쓰기 가능
- 파일 관련 작업이 모두 끝난 후, 객체의 close() 메서드를 통해 파일을 닫아줘야 한다.

파일 열기(open) 옵션

open 함수의 두번째 매개변수(또는 mode 매개변수)에 옵션 문자열 추가

f = open(파일 경로 문자열, 열기 옵션)

사용 옵션		처리 옵션		추가 옵션	
r	읽기 전용(디폴트)	t	텍스트 모드(디폴트)	+	읽기 및 쓰기
w	쓰기 전용 (파일 없으면 생성, 있으면 기존 내용에 덮어씀)	b	바이너리 모드		
x	쓰기 전용 (파일이 존재하면 오류)				
a	쓰기 전용 (파일 없으면 생성, 파일 존재하면 기존 내용 뒤에 이어 씀)				

위 3가지 옵션을 조합하여 사용

파일 열기(open) 옵션 예시

```
f = open("file.txt")

f = open("file.txt", "w")

f = open("file.txt", "a")

f = open("file.bin", "wb+")
```

옵션X → 디폴트로 읽기 모드 + 텍스트 모드 (rt와 동일)

쓰기 모드(write)로 열기 → 파일의 모든 내용 삭제

쓰기 모드(append)로 열기 → 기존 내용은 유지되고 이어서 쓰기

쓰기 모드 + 바이너리 모드 + 읽기/쓰기 모드

파일 닫기(close)

파일을 열었으면 닫아야 한다!

```
f = open("filename.txt")

# 파일 관련 처리 수행

f.close()
```

파일 객체의 close() 메서드로 호출

사용이 끝난 파일은 안전하게 닫아야 함

파일을 열어둔 상태에서는 파일이 닫히기 전까지 잠김
→ 파일 삭제/이동 불가능

닫지 않으면, 시스템 자원 고갈과 변경 사항이 제대로 반영되지 않는 문제 발생할 수 있다.

파일 관련 작업이 모두 끝나고 마지막에 호출

파일 읽고 쓰기(read, write)

파일 읽기(read): read() 메서드

파일에 쓰여진 내용 가져오기; 전체 또는 일부 가져오기

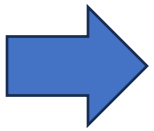
```
Monty
Python's
Flying
Circus
```

file.txt

현재 위치에서 마지막까지의 내용을 읽는 메서드
파일의 전체 내용 또는 일부를 읽을 때 유용

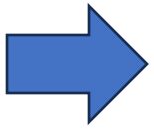
인자로 숫자를 지정하면, 해당 수 만큼 문자를 읽음
- 한글의 경우 UTF-8 기준 3 bytes
  - 텍스트 모드에서는 1만 지정해도 한글이라면 3 bytes를 읽음

```
f = open("file.txt")

print(f.read())
```



```
Monty
Python's
Flying
Circus
```

```
f = open("file.txt")

print(f.read(3))
```



```
Mon
```

파일 읽기(read): readline() 메서드

파일에 쓰여진 내용 가져오기; 한 줄 가져오기

```
Monty
Python's
Flying
Circus
```

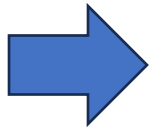
file.txt

현재 위치에서 하나의 줄을 읽는 메서드

파일 앞 부분의 몇 줄만 필요한 경우 유용

줄을 읽을 때, 마지막 줄 바꿈 문자(₩n)가 같이 붙게 된다.

```
f = open("file.txt")

print(f.readline())

print(f.readline())
```



```
Monty

Python's
```

파일 읽기(read): readlines() 메서드

파일에 쓰여진 내용 가져오기; 모든 줄 가져오기

```
Monty
Python's
Flying
Circus
```

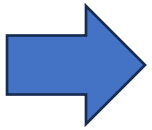
file.txt

현재 위치에서 모든 줄을 읽어 리스트로 반환하는 메서드

파일을 줄 단위로 가공할 때 유용(단, 내용이 많으면 메모리 효율 저하)

각 줄마다, 마지막에 줄 바꿈 문자(₩n)가 같이 붙게 된다.

```
f = open("file.txt")

print(f.readlines())
```



```
['Monty\n', "Python's\n", 'Flying\n', 'Circus']
```

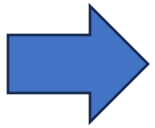
파일 쓰기(write): write() 메서드

파일에 새로운 내용을 쓰는 것

```
f = open("file.txt", "w")
f.write("Monty\n")
f.write("Python")

f.close()
```

현재 위치에서 문자열을 파일에 쓰는 메서드

줄 바꿈이 필요하면 줄 바꿈 문자(₩n)를 사용
(또는 삼중 따옴표(“””) 표기로 한 번에 여러 줄 쓰기 가능)



```
Monty
Python
```

file.txt

파일 쓰기(write): writelines() 메서드

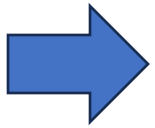
파일에 새로운 내용을 쓰는 것

```
f = open("file.txt", "w")
f.writelines(["Monty\n", "Python"])

f.close()
```

현재 위치에서 문자열 리스트를 파일에 쓰는 메서드

리스트의 각 원소마다 줄 바꿈이 자동으로 되는 것이 아님!

줄 바꿈이 필요한 해당 원소 문자열 마지막에 줄 바꿈 문자(₩n)를 붙여야 한다.



```
Monty
Python
```

file.txt

파일의 위치(tell, seek)

파일의 현재 위치 가져오기: tell() 메서드

```
Monty Python's Flying Circus
```

file.txt

```
f = open("file.txt")

print(f.read(3)) # Mon

print(f.tell()) # 3
```

파일의 현재 위치를 반환하는 메서드

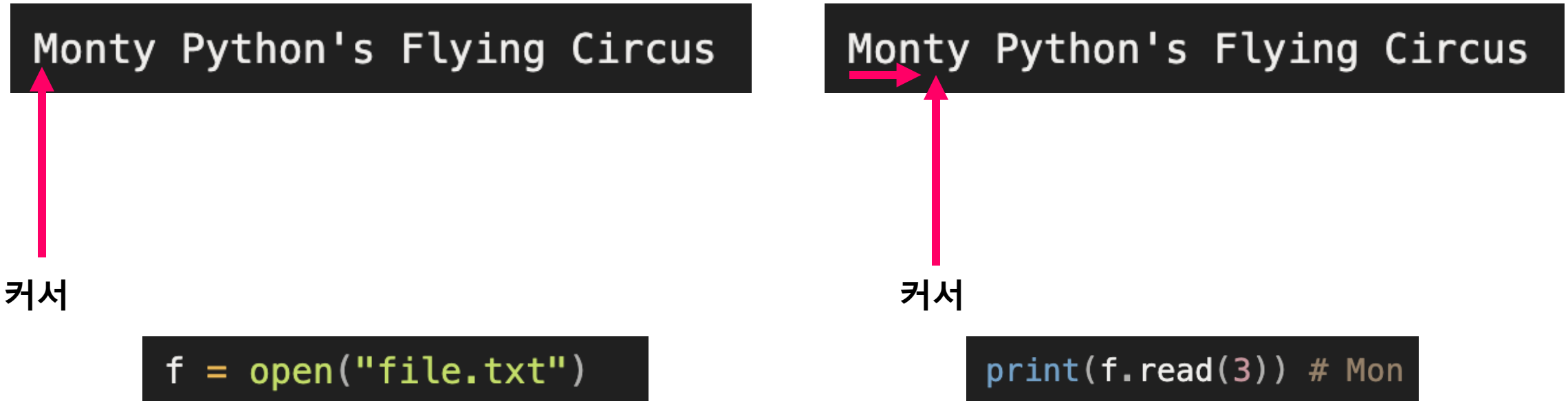
처음 위치(0)에서 Byte 단위로 몇 번째에 있는지 반환
* Byte 단위이기 때문에 한글 한 글자에 대해서 위치가 3 바뀜

파일의 위치는 읽기 작업 또는 쓰기 작업을 수행하면 이동함.

파일을 최초로 연 경우 가장 앞(0)에 위치한다.
(단, a 모드는 항상 파일의 끝에서 시작)

현재 위치?

현재 파일의 내용 중에서 어디를 가리키고 있는지에 대한 위치(= 커서, 포인터)



파일의 현재 위치 변경하기: seek() 메서드

```
Monty Python's Flying Circus
```

file.txt

```
f = open("file.txt")

print(f.read(3)) # Mon

print(f.tell()) # 3
```

파일의 현재 위치를 변경하는 메서드

seek(offset) 형태로 사용

- offset은 위치를 의미하며, 처음에서 offset만큼 이동
- 만약 현재 위치에서 이동하려면 f.seek(f.tell() + offset) 형태로 사용해야 함

*Byte 단위이기에 한글 한 글자에 대해 이동하려면 3씩 이동해야함

with 구문

with 구문

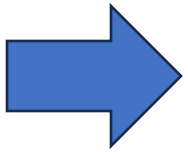
자원 객체(파일 객체, DB 세션 등)에 대상으로 한정된 구역에서 사용하도록 하는 구문

파일을 열고, close()를 통해 닫아줘야 하는데 사용자의 실수로 누락되거나, 중간에 에러로 close() 실행이 되지 않아 리소스 누수나 데이터 손실 문제 발생 가능성 있다

with구문을 사용하면 파일을 열고, 작업, 제때 닫아주는 동작을 자동으로 보장해준다.

```
f = open("file.txt")

content = f.read()
print(content)

f.close()
```

직접 닫기



```
with open("file.txt") as f:
    content = f.read()

print(content)
```

자동 닫기

with 구문

블럭 내에서 해당 객체를 활용한 작업이기에 가독성 및 유지보수에 효과적
특별한 이유가 없다면 with 구문 활용하자

with 자원호출함수 as 변수:
<작업>

```
f1 = open("file1.txt")
f2 = open("file2.txt")

content1 = f1.read()
content2 = f2.read()

f1.close()
f2.close()
print(content1)
print(content2)
```

여러 파일을 열어 작업할 때도 효과적

```
with (
  open("file1.txt", encoding="utf-8") as f1,
  open("file2.txt", encoding="utf-8") as f2
):
    content1 = f1.read()
    content2 = f2.read()
print(content1)
print(content2)
```

예외 처리(Exception Handling)

예외 처리(Exception Handling)

예외(Exception)?

코드를 작성하다 보면 예상치 못한 오류가 발생하는 경우가 비일비재
프로그램 상에서 논리적인 오류가 발생했을 때 예외(Exception)이라고 한다

오류가 발생하면 프로그램은 비정상적으로 즉시 종료됨

예상치 못한 상황을 만나도 프로그램이 비정상적으로 중단되지 않고
대응하여 정상 흐름으로 복귀시키는 등의 작업을 수행하는 것이 **예외 처리**

프로그램은 작성자에 의해 완벽한 통제가 불가능!

- 사용자의 예상을 벗어난 입력
- 찾는 파일이 없음
- 서버와의 통신 시, 응답하지 않는 경우

예외 처리(Exception Handling): try-except 구문

발생할 수 있는 예외에 대해 오류 및 종료하는 것 이외의 동작을 정의하는 것

```
try:
    1 / 0
except: ZeroDivisionError:
    print("0으로 나눌 수 없습니다.")
```

try 블럭
예외가 발생할 가능성이 있는 코드

except 블럭
예외 상황이 발생했을 때 처리할 동작 정의
(+어떤 예외인지 명시)

버그를 숨기는 것이 아니라 현실 세계의 예측 불가능함을 견디게 하는 안전벨트의 역할

예외 처리(Exception Handling)

파이썬 예외처리 철학: EAFP

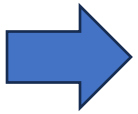
EAFP: Easier to Ask Forgiveness than Permission (허락보다 용서가 더 쉽다)

문제가 생기면 예외를 잡으면 된다!

파이썬은 실행 후, 예외를 처리하는 것을 권장

```
try:
    print(1 / 0)
except ZeroDivisionError:
    print("0으로 나눌 수 없습니다.")
```

LBYL: Look Before You Leap (뛰기 전에 보라)



Java 계열의 예외처리 철학

실행하기 전에 에러가 날만한 것을 잡아라!

조건문을 통해 에러가 날만한 상황을 검사

```
if dominator != 0:
    print(1 / dominator)
else:
    print("0으로 나눌 수 없습니다.")
```

예외 처리(Exception Handling)

예외의 종류

다양한 상황에 대한 예외 종류

- **ZeroDivisionError**: 0으로 나누었을 때
- **FileNotFoundError**: 존재하지 않는 파일을 열었을 때
- **ValueError**: 잘못된 값 전달
- **TypeError**: 잘못된 자료형 연산
- **IndexError**: 인덱스 범위 초과
- **KeyError**: 딕셔너리에서 존재하지 않는 키 참조
- **AttributeError**: 존재하지 않는 멤버속성 접근
- **ImportError**: 모듈 불러오기 실패
- **MemoryError**: 메모리 부족
- **Exception**: 모든 예외를 포괄적으로 처리(최상위 예외 클래스)

예외의 종류 (1): ZeroDivisionError

어떤 수를 0으로 나누었을 때 발생

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print("0으로 나눌 수 없습니다:", e)
```

"as e" 형태로 예외 객체를 생성하여 발생한 예외의 정보(유형, 메시지)를 확인할 수 있음

예외의 종류 (2): FileNotFoundError

존재하지 않는 파일에 대해 접근했을 때 발생

```
try:
    f = open("없는파일.txt", "r")
except FileNotFoundError as e:
    print("파일을 찾을 수 없습니다:", e)
```

예외의 종류 (3): ValueError

함수의 매개변수 등 잘못된 값이 전달될 때 발생

```
try:
    num = int("abc")
except ValueError as e:
    print("정수로 변환할 수 없습니다:", e)
```

예외의 종류 (4): IndexError

리스트나 투플의 존재하지 않는 인덱스를 접근할 때 발생

```
try:
    data = [1, 2, 3]
    print(data[5])
except IndexError as e:
    print("인덱스가 범위를 벗어났습니다:", e)
```

예외 처리(Exception Handling)

예외처리 구문의 확장: try-except-else-finally

```
try:
    a, b = map(int, input("두 정수를 입력하세요:").split())
except ValueError:
    print('입력이 옳지 않습니다.')
except Exception:
    print('알 수 없는 오류 발생')
else:
    print(a + b)
finally:
    print('프로그램 실행을 마쳤습니다.')
```

try: 예외가 발생할 가능성 있는 코드

except: 예외가 발생 했을 때 동작
* 0개 이상

else: 예외가 발생하지 않았을 때의 동작
* 0 or 1개

finally: 예외와 상관없이 항상 실행되는 동작
* 0 or 1개

예외처리 구문의 확장: try-except-else-finally

다중 except 구문의 의미

```
try:
    a, b = map(int, input("두 정수를 입력하세요:").split())
except ValueError:
    print('입력이 옳지 않습니다.')
except Exception:
    print('알 수 없는 오류 발생')
else:
    print(a + b)
finally:
    print('프로그램 실행을 마쳤습니다.')
```

하나의 try블럭에서
여러 개의 예외 상황이 발생할 것 같을 때
다중 except 구문 설정

각 예외 상황에 대응되는 except 블럭 실행

단, Exception은 모든 예외에 대응되기에
가장 마지막 except 구문에 추가

먼저 구체적인 예외 정의,
그 다음 최후의 보루인 Exception

예외처리 구문의 확장: try-except-else-finally

시나리오 (1): 예외가 발생하지 않았을 때

```
try:
    a, b = map(int, input("두 정수를 입력하세요:").split())
except ValueError:
    print('입력이 옳지 않습니다.')
except Exception:
    print('알 수 없는 오류 발생')
else:
    print(a + b)
finally:
    print('프로그램 실행을 마쳤습니다.')
```

try → else → finally

예외처리 구문의 확장: try-except-else-finally

시나리오 (2): 예외가 발생했을 때 (사용자 입력 오류)

```
try:
    a, b = map(int, input("두 정수를 입력하세요:").split())
except ValueError:
    print('입력이 옳지 않습니다.')
except Exception:
    print('알 수 없는 오류 발생')
else:
    print(a + b)
finally:
    print('프로그램 실행을 마쳤습니다.')
```

try → except → finally

감사합니다!