데이터 과학을 위한
파이썬 프로그래밍

2판

Chapter 12
예외 처리와 파일 다루기

목차

학습목표

- 예외의 개념과 사례에 대해 알아본다.
- 예측 가능한 예외와 예측 불가능한 예외에 대해 이해한다.
- 파일의 개념과 종류에 대해 학습한다.
- 파일을 읽고 쓰는 방법을 실습하고, pickle 모듈에 대해 알아본다.

01
예외 처리

1. 예외의 개념과 사례

- **예외(exception):** 프로그램을 개발하면서 예상하지 못한 상황이 발생하는 것

  - 대표적인 사례: ① 사용자의 입력 오류
  - 대표적인 사례: ② MS 오피스에서 지원하는 자동 저장(autosave) 기능

(a) 아이디 생성을 위한 입력 오류 (b) 자동 저장 기능

그림 12-1 예외에 대비한 사례

2. 예측 가능한 예외와 예측 불가능한 예외

2.1 예측 가능한 예외

- 발생 여부를 개발자가 사전에 인지할 수 있는 예외

  - 개발자는 예외를 예측하여 예외가 발생할 때는 대응책을 마련해 놓을 수 있음

  - 대표적으로 사용자 입력란에 값이 잘못 입력되었다면 if문을 사용하여 잘못 입력하였다고 응답하는 방법이 있음. 아주 쉽게 대응이 가능함

2.2 예측 불가능한 예외

- 대표적으로 매우 많은 파일을 처리할 때 문제가 발생할 수 있음
- 예측 불가능한 예외 발생시 인터프리터가 자동으로 사용자에게 알려줌
- 예외 처리는 제품의 완성도를 높이는 차원에서 매우 중요한 개념임

3. 예외 처리 구문

3.1 try-except문

- 파이썬에서 예외 처리의 기본 문법: try-except문

- 작성 방식: try문에 예외 발생이 예상되는 코드를 적고, except문에 예외 발생 시 대응하는 코드를 작성함.

```
try:
    예외 발생 가능 코드
except 예외 타입:
    예외 발생 시 실행되는 코드
```

3. 예외 처리 구문

[코드 12-1] try-except.py

```
1 for i in range(10):
2     try:
3         print(10 / i)
4     except ZeroDivisionError:
5         print("Not divided by 0")
```

[실행결과]

```
Not divided by 0
10.0
5.0
3.3333333333333335
2.5
2.0
1.6666666666666667
1.4285714285714286
1.25
1.1111111111111112
```

3. 예외 처리 구문

여기서 잠깐! 예외의 종류와 예외에 대한 에러 메시지

1. 예외의 종류

파이썬에서 기본적으로 사용할 수 있는 예외의 종류는 다양하다.

먼저, IndexError는 리스트의 인덱스 범위를 넘어갈 때 처리하는 예외이다. 코드를 작성하다 데이터가 150개인 줄 알았는데, 151개가 있어 처리를 요청하는 경우 IndexError가 호출된다.

다음으로 NameError는 코드에서 호출하는 특별한 변수명이 없을 때 호출되는 에러이다. 이 언어가 인터프리터 언어이다 보니 존재하지 않은 변수도 실행 시점에서 잡아낸다. 컴파일러 언어의 경우 실행 전이나 컴파일러 시점에 에러가 발생하는 것과는 대조적이다.

그 외에 파이썬에서 자주 사용하는 내장 예외는 [표 12-1]과 같다.

표 12-1 예외의 종류

예외	내용
IndexError	리스트의 인덱스 범위를 넘어갈 때
NameError	존재하지 않는 변수를 호출할 때
ZeroDivisionError	0으로 숫자를 나눌 때
ValueError	변환할 수 없는 문자나 숫자를 변환할 때
FileNotFoundError	존재하지 않는 파일을 호출할 때

2. 예외에 대한 에러 메시지

내장 예외와 함께 사용하기 좋은 것이 예외에 대한 에러 메시지이다. [코드 12-2]와 같이 except문의 마지막에 'as e' 또는 'as 변수명'을 입력하고, 해당 변수명을 출력하면 된다. 실행 결과 'division by zero'라는 에러 메시지를 확인할 수 있는데, 이 에러 메시지는 파이썬 개발자들이 사전에 정의한 것으로, 특정한 에러를 빠르게 이해할 수 있도록 해준다.

3. 예외 처리 구문

[코드 12-2] error_message.py

```
1 for i in range(10):
2     try:
3         print(10 / i)
4     except ZeroDivisionError as e:
5         print(e)
6         print("Not divided by 0")
```

[실행결과]

```
division by zero
Not divided by 0
10.0
5.0
3.3333333333333335
2.5
2.0
1.6666666666666667
1.4285714285714286
1.25
1.1111111111111112
```

3.2 try-except-else문

- if-else 문과 비슷한데, 해당 예외가 발생하지 않는 경우 수행할 코드를 else 문에 작성하면 됨.

- try-except-else문의 기본 형태

```
try:
    예외 발생 가능 코드
except 예외 타입:
    예외 발생 시 실행되는 코드
else:
    예외가 발생하지 않을 때 실행되는 코드
```

3. 예외 처리 구문

[코드 12-3] try-except-else.py

```
1 for i in range(10):
2     try:
3          result =10 / i
4     except ZeroDivisionError:
5          print("Not divided by 0")
6     else:
7          print(10 / i)
```

[실행결과]

```
Not divided by 0
10.0
5.0
3.3333333333333335
2.5
2.0
1.6666666666666667
1.4285714285714286
1.25
1.1111111111111112
```

3.3 try-except-finally문

- try-except-finally문에서 finally문은 try-except문 안에 있는 수행 코드가 아무런 문제 없이 종료되었을 경우 최종으로 호출하는 코드임
- for문에서 사용하는 finally문과 용도가 비슷함

- try-except-finally문의 기본 형태

```
try:
    예외 발생 가능 코드
except 예외 타입:
    예외 발생 시 실행되는 코드
finally:
    예외 발생 여부와 상관없이 실행되는 코드
```

3. 예외 처리 구문

[코드 12-4] try-except-finally.py

```
1 try:
2     for i in range(1, 10):
3         result =10 // i
4         print(result)
5 except ZeroDivisionError:
6     print("Not divided by 0")
7 finally:
8     print("종료되었다.")
```

[실행결과]

```
10
5
3
2
2
1
1
1
1
종료되었다.
```

3.4 raise문

- try-except문과 달리 필요할 때 예외를 발생시키는 코드
- raise문의 기본 형태

```
raise 예외 타입(예외 정보)
```

3. 예외 처리 구문

[코드 12-5] raise.py

```
1 while True:
2     value = input("변환할 정수값을 입력해 주세요: ")
3     for digit in value:
4         if digit not in "0123456789":
5         raise ValueError("숫자값을 입력하지 않았습니다.")
6     print("정수값으로 변환된 숫자 -", int(value))
```

[실행결과]

```
변환할 정수값을 입력해 주세요: 10
정수값으로 변환된 숫자 - 10
변환할 정수값을 입력해 주세요: ab
Traceback (most recent call last):
    File "raise.py", line 5, in <module>
        raise ValueError("숫자값을 입력하지 않았습니다.")
ValueError: 숫자값을 입력하지 않았습니다.
```

3. 예외 처리 구문

3.5 assert문

- 미리 알아야 할 예외 정보가 조건을 만족하지 않을 경우 예외를 발생시킴.
- if문과 함께 raise문을 사용하여 강제로 예외를 발생시켰는데, assert문은 True 또는 False의 반환이 가능한 함수를 사용하여 간단하게 예외를 발생시킬 수 있음. 만약 False가 반환되면 예외가 발생함.

- assert 문의 기본 형태

```
assert 예외 조건
```

3. 예외 처리 구문

[코드 12-6] assert.py

```
1 def get_binary_number(decimal_number):
2     assert isinstance(decimal_number, int)
3     return bin(decimal_number)
4 print(get_binary_number(10))
5 print(get_binary_number("10"))
```

[실행결과]

```
0b1010                                          ← 4행 실행 결과
Traceback (most recent call last):              ← 5행 실행 결과
    File "<C:/.../assert.py>", line 5, in <module>
        print(get_binary_number("10"))
    File "C:/.../assert.py", line 2, in get_binary_number
        assert isinstance(decimal_number, int)
AssertionError
```

02
파일 다루기

1. 파일의 개념

- **파일(file):** 컴퓨터를 실행할 때 가장 기본이 되는 단위
- 일반적으로 윈도의 GUI 환경에서는 아이콘을 더블클릭하여 실행함.

그림 12-2 윈도 GUI 환경의 아이콘

- 사실 이것도 아이콘을 더블클릭하여 프로그램을 실행하는 것처럼 보이나, 실제로는 아이콘과 연결된 파일이 실행되는 구조임.

1. 파일의 개념

- 아이콘에서 마우스 오른쪽 버튼을 클릭하고 [속성]을 선택하면 [그림 12-3]과 같은 화면을 볼 수 있음.

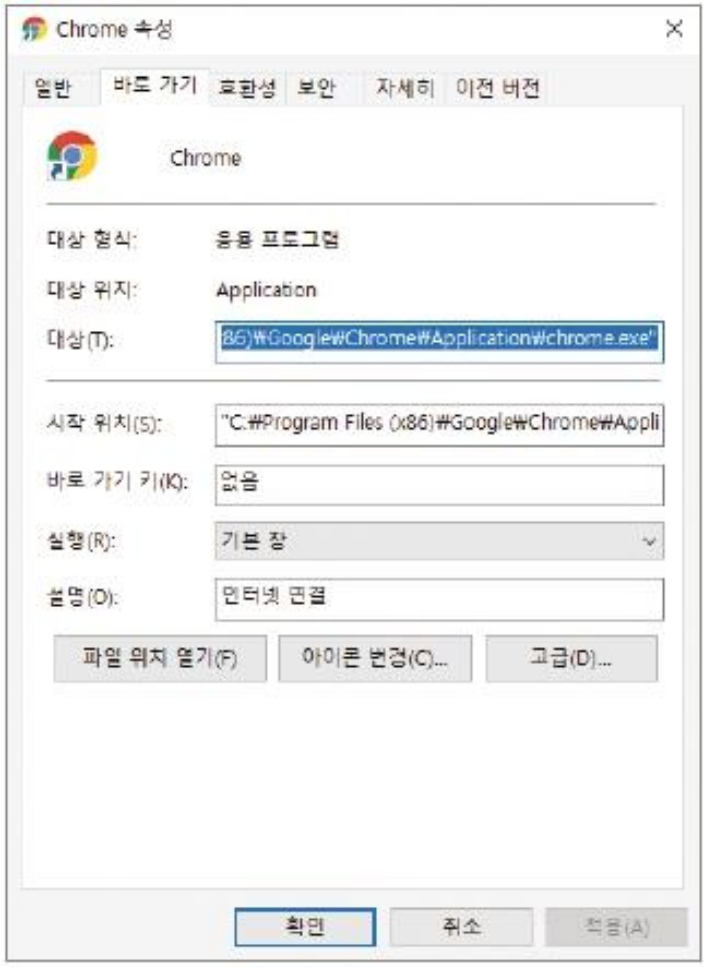

그림 12-3 아이콘의 속성

1. 파일의 개념

여기서 잠깐! 파일과 디렉터리

파일을 이해하기 위해 먼저 파일과 디렉터리에 대해 알아보자. 윈도에서 사용하는 탐색기는 윈도와 E 키를 함께 누르면 나타난다. 이것이 기본 파일 시스템이다. 기본적으로 파일 시스템은 파일과 디렉터리로 구분하는데, 윈도에서는 디렉터리라는 용어 대신 폴더라는 용어를 사용한다.

디렉터리는 파일을 담는 또 하나의 파일로, 여러 파일을 포함할 수 있는 그릇이다. 파일과 다른 디렉터리를 포함할 수 있으므로 직접 프로그램을 실행하지는 않지만 다른 파일들을 구분하고 논리적인 단위로 파일을 묶을 수 있다.

파일은 컴퓨터에서 논리적으로 정보를 저장하는 가장 작은 단위이다. 파일은 일반적으로 파일명과 확장자로 구별한다. 예를 들어, 파이썬 파일로 저장 관리하는 파일들은 py라는 확장자를 가지고 있다. 확장자는 그 파일의 쓰임을 구분하는 글자로, hwp, ppt, doc 같은 것들이 있다. 파일은 다른 정보를 저장하거나 프로그램을 실행하고, 다른 프로그램이 실행될 때 필요한 정보를 제공하는 등 여러 가지 역할을 한다.

흔히 탐색기 프로그램에서 파일과 디렉터리는 트리 구조로 표현되는데, 그 이유가 바로 디렉터리와 파일이 서로 포함 관계를 가지기 때문이다.

2. 파일의 종류

➢ **바이너리 파일(binary file)**

- 컴퓨터만 이해할 수 있는 이진 정보로 저장된 파일
- 비트(bit) 형태로 저장되어 메모장으로 열면 내용이 보이지 않거나 내용을 확인할 수 없는 파일

➢ **텍스트 파일(text file)**

- 사람이 이해할 수 있는 문자열로 저장된 파일
- 메모장으로 그 내용을 확인할 수 있음

2. 파일의 종류

표 12-2 바이너리 파일과 텍스트 파일

바이너리 파일	텍스트 파일
• 컴퓨터만 이해할 수 있는 형태인 이진(법) 형식으로 저장된 파일 • 일반적으로 메모장으로 열면 내용이 깨져 보임(메모장에서 해석 불가) • 엑셀이나 워드와 같은 프로그램 파일	• 사람도 이해할 수 있는 형태인 문자열 형식으로 저장된 파일 • 메모장으로 열면 내용 확인이 가능 • 메모장에 저장된 파일이나 HTML 파일, 파이썬 코드 파일 등

- 텍스트 파일도 사실 컴퓨터가 처리하기 위해 바이너리 형태로 저장된 파일

  - 사람이 확인할 수 있는 파일이라고 해서 컴퓨터가 그런 형태로 저장된 파일을 확인할 수 있는 것은 아님.

  - 텍스트 파일은 컴퓨터가 이해할 수 있는 형태로 변경하여 저장됨.

  - 이렇게 변경하는 기준을 아스키코드(ASCII)나 유니코드(Unicode)로 하고, 이 표준에 따라 텍스트 파일을 컴퓨터가 이해할 수 있도록 바꿈.

  - 컴퓨터는 오직 이진수만 이해할 수 있으므로 모든 문자열 값도 전부 이진수로 변경하여 저장함.

3. 파일 읽기

- 파이썬에서는 텍스트 파일을 다루기 위해 open( ) 함수 사용
- open( ) 함수와 파일명, 파일 열기 모드를 입력하면 그 옵션에 따라 파일을 다룰 수 있음

```
f = open("파일명", "파일 열기 모드")
f.close()
```

표 12-3 파일 열기 모드

종류	설명
r	읽기 모드: 파일을 읽기만 할 때 사용
w	쓰기 모드: 파일에 내용을 쓸 때 사용
a	추가 모드: 파일의 마지막에 새로운 내용을 추가할 때 사용

3. 파일 읽기

3.1 파일 읽기

[코드 12-7] fileopen1.py

```
1 f = open("dream.txt", "r")
2 contents = f.read()
3 print(contents)
4 f.close()
```

[실행결과]

```
I have a dream a song to sing
to help me cope with anything
if you see the wonder of a fairy tale
you can take the future even
if you fail I believe in angels
something good in everything
```

3. 파일 읽기

3.2 with문과 함께 사용하기

- with문과 함께 open( ) 함수 사용 - with문은 들여쓰기를 하는 동안에는 open( ) 함수가 유지되고, 들여쓰기가 끝나면 open( ) 함수도 종료되는 방식.

[코드 12-8] fileopen2.py

```
1 with open("dream.txt","r") as my_file:
2     contents = my_file.read()
3     print(type(contents), contents)
```

[실행결과]

```
<class 'str'> I have a dream a song to sing
to help me cope with anything
if you see the wonder of a fairy tale
you can take the future even
if you fail I believe in angels
something good in everything
```

3. 파일 읽기

3.3 한 줄씩 읽어 리스트형으로 반환하기

- readlines( ) 함수를 사용하여 한 줄씩 내용을 읽어와 문자열 형태로 저장.
- 한 줄의 기준은 \n으로 나뉘어지고 리스트로 반환될 때 for문 등 활용.

[코드 12-9] fileopen3.py

```
1 with open("dream.txt","r") as my_file:
2     content_list = my_file.readlines()  # 파일 전체를 리스트로 반환
3     print(type(content_list))           # 자료형 확인
4     print(content_list)                 # 리스트값 출력
```

[실행결과]

```
<class 'list'>
['I have a dream a song to sing \n', 'to help me cope with
anything \n', 'if you see the wonder of a fairy tale \n', 'you can
take the future even \n', 'if you fail I believe in angels \n',
'something good in everything \n']
```

3.4 실행할 때마다 한 줄씩 읽어 오기

- **read( ) 함수:** 파일을 한 번 열 때 파일의 처음부터 끝까지 모든 파일의 내용을 읽어오는 함수
- readline( ) 함수는 호출될 때마다 한 줄씩 읽어오는 함수

[코드 12-10] fileopen4.py

```
1 with open("dream.txt", "r") as my_file:
2     i = 0
3     while 1:
4         line = my_file.readline()
5         if not line:
6             break
7         print(str(i)+" === "+ line.replace("\n","")) # 한 줄씩 값 출력
8         i = i + 1
```

3. 파일 읽기

[실행결과]

```
0 === I have a dream a song to sing
1 === to help me cope with anything
2 === if you see the wonder of a fairy tale
3 === you can take the future even
4 === if you fail I believe in angels
5 === something good in everything
```

3.5 파일에 저장된 글자의 통계 정보 출력하기

[코드 12-11] fileopen5.py

```
1 with open("dream.txt", "r") as my_file:
2 contents = my_file.read()
3 word_list = contents.split(" ")    # 빈칸 기준으로 단어를 분리하여 리스트
4 line_list = contents.split("\n")   # 한 줄씩 분리하여 리스트
5
6 print("총 글자의 수:", len(contents))
7 print("총 단어의 수:", len(word_list))
8 print("총 줄의 수:", len(line_list)
```

[실행결과]

```
총 글자의 수: 188          ← 6행 실행 결과
총 단어의 수: 35           ← 7행 실행 결과
총 줄의 수: 7              ← 8행 실행 결과
```

4. 파일 쓰기

- **인코딩:** 텍스트 파일을 저장하기 위해서 저장할 때 사용하는 표준을 지정하는 것

[코드 12-12] filewrite1.py

```
1 f = open("count_log.txt", 'w', encoding = "utf8")
2 for i in range(1,11):
3     data = "%d번째 줄이다.\n"% i
4     f.write(data)
5 f.close()
```

4. 파일 쓰기

4.1 파일 열기 모드 a로 새로운 글 추가하기

- 쓰기 모드인 w는 늘 새로운 파일을 생성함

[코드 12-13] filewrite2.py

```
1 with open("count_log.txt", 'a', encoding = "utf8") as f:
2     for i in range(1, 11):
3         data = "%d번째 줄이다.\n"% i
4         f.write(data)
```

4.2 디렉터리 만들기

- **log:** 디렉터리를 생성하는 코드

[코드 12-14] mkdir1.py

```
1 import os
2 os.mkdir("log")
```

4. 파일 쓰기

- [코드 12-14]를 실행하면 폴더 아래에 log라는 새로운 폴더가 생성

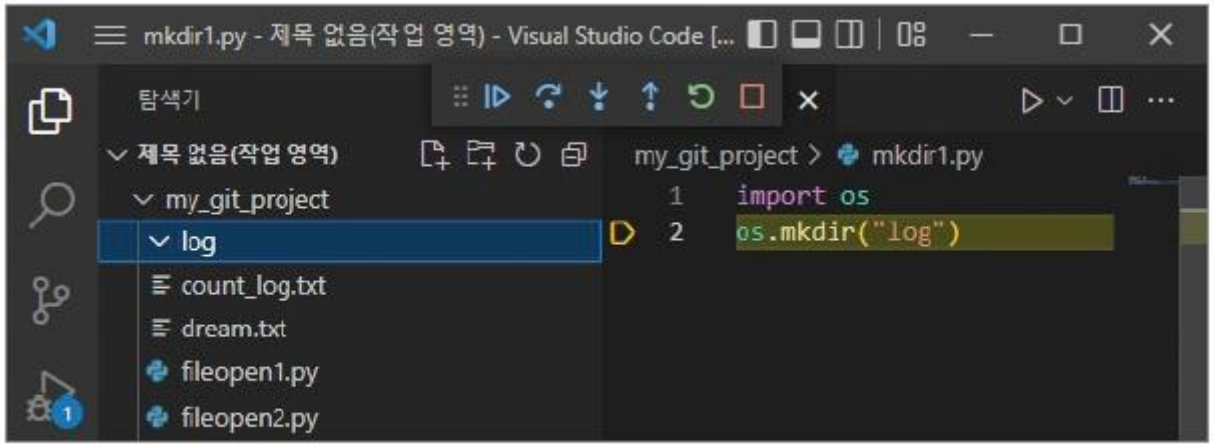

그림 12-4 log 폴더 생성

- 하지만 프로그램 대부분이 새로 실행되므로 기존에 해당 디렉터리가 있는지 확인하는 코드가 필요함

☞ 이 경우 [코드 12-15]와 같이 os.path.isdir 코드를 사용하여 기존 디렉터리의 존재 여부를 확인하면 됨.

4. 파일 쓰기

[코드 12-15] mkdir2.py

```
1 import os
2 os.mkdir("log")
3
4 if not os.path.isdir("log"):
5     os.mkdir("log")
```

[실행결과]

```
Traceback (most recent call last):
    File "mkdir2.py", line 2, in <module>
        os.mkdir("log")
FileExistsError: [WinError 183] 파일이 이미 있으므로 만들 수 없습니다: 'log'
```

4. 파일 쓰기

4.3 로그 파일 만들기

- **로그 파일:** 프로그램이 동작하는 동안 중간 기록을 저장하는 역할의 파일.

[코드 12-16] logfile.py

```
 1 import os
 2
 3 if not os.path.isdir("log"):
 4     os.mkdir("log")
 5
 6 if not os.path.exists("log/count_log.txt"):
 7     f = open("log/count_log.txt", 'w', encoding = "utf8")
 8     f.write("기록이 시작된다.\n")
 9     f.close()
10
11 with open("log/count_log.txt", 'a', encoding = "utf8") as f:
12     import random, datetime
13     for i in range(1, 11):
14         stamp = str(datetime.datetime.now())
15         value = random.random() * 1000000
16         log_line = stamp + "\t" + str(value) + "값이 생성되었다." + "\n"
17 f.write(log_line)
```

5. pickle 모듈

- **영속화(persistence):** 필요한 객체를 파일로 저장시켜 다시 사용할 수 있도록 하는 것

  ☞ 파이썬은 pickle 모듈을 제공해 메모리에 로딩된 객체를 영속화할 수 있도록 지원

- **pickle 모듈 사용하기**
- 파일을 생성할 때는 'w'가 아닌 'wb'로 열어야 함
- dump( ) 함수에서는 저장할 객체, 저장될 파일 객체를 차례대로 인수로 넣으면 해당 객체가 해당 파일에 저장됨.

```
>>> import pickle
>>>
>>> f = open("list.pickle", "wb")
>>> test = [1, 2, 3, 4, 5]
>>> pickle.dump(test, f)
>>> f.close()
```

5. pickle 모듈

- 저장된 pickle 파일을 불러오는 프로세스

```
>>> f = open("list.pickle", "rb")
>>> test_pickle = pickle.load(f)
>>> print(test_pickle)
[1, 2, 3, 4, 5]
>>> f.close()
```

- 다음 코드와 같이 곱셈을 처리하는 클래스를 생성한다고 가정

  - 이 코드의 클래스는 처음 객체를 생성할 때 초기값을 생성하고, multiply( ) 함수를 부를 때마다 '초기값 * number'의 값을 호출하는 클래스임. 일종의 곱셈 클래스.

```
>>> class Mutltiply(object):
... def __init__(self, multiplier):
... self.multiplier = multiplier
... def multiply(self, number):
... return number * self.multiplier
...
>>> muliply = Mutltiply(5)
>>> muliply.multiply(10)
50
```

5. pickle 모듈

- 프로그램을 작성하다 보면 매우 복잡한 연산도 따로 저장하여 사용할 때가 있음
- 저장 모듈을 효율적으로 사용하기 위해 다음 코드처럼 pickle 모듈을 사용할 수 있음

```
>>> import pickle
>>>
>>> f = open("multiply_object.pickle", "wb")
>>> pickle.dump(muliply, f)
>>> f.close()
>>>
>>> f = open("multiply_object.pickle", "rb")
>>> multiply_pickle = pickle.load(f)
>>> multiply_pickle.multiply(5)
25
```

Thank you!

한빛아카데미 Hanbit Academy, Inc.