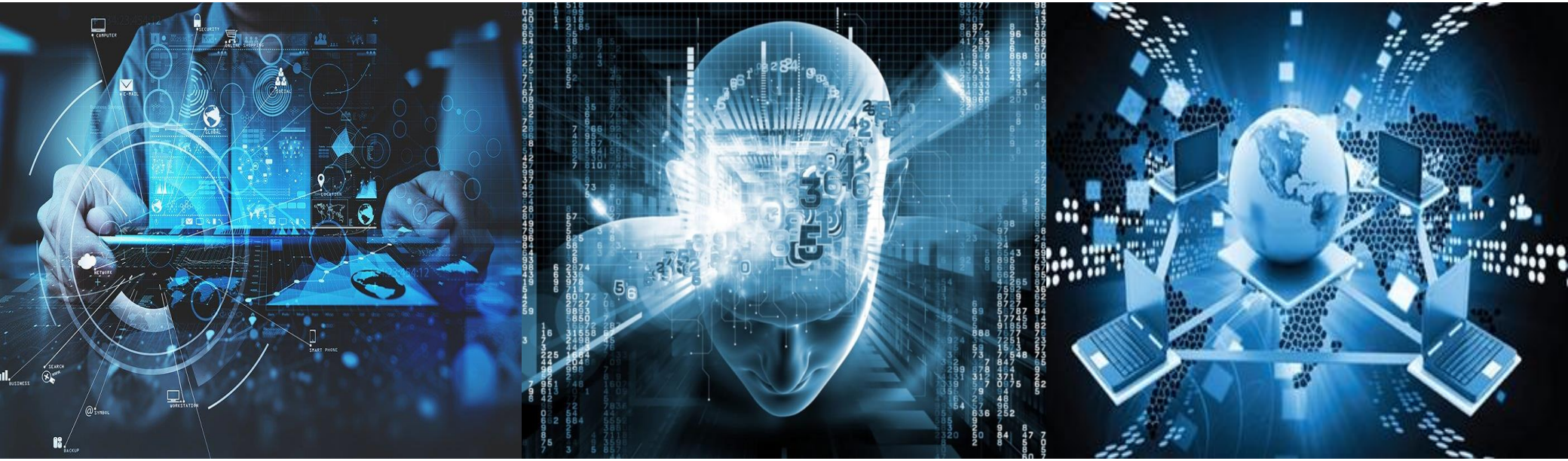
JB Financial Group

비파이 크래프트
Find Information to design valuable society



Data Analytics Based Python

SECT13. 에러와 예외처리

IT Competency Improvement Training
Kim Jin Soo

학습내용

- 구문 에러, Syntax Errors
- 예외, Exceptions
- try-except 구문으로 예외상황 제어, Handling Exceptions
- else와 finally 활용하기
- 사용자 정의 예외, User-defined Exceptions

구문 에러, Syntax Errors

❖ 프로그램에서 발생하는 에러는 완전히 제거할 수 없는 존재

- 어느 정도 성숙한 개발자들은 소스 코드에 에러가 발생하는 것을 두려워하거나 치부라고 생각하지 않는다.
- 오히려, 더 큰 재앙을 막아주는 고마운 존재

❖ 초보 개발자가 가장 많이 발생하는 에러가 무엇일까?

- 바로, 문법상 오류~^^
- SyntaxError는 구문 에러를 뜻한다.
- 즉, 문법 오류로 인해 소스 코드를 실행 하기 전에 발생하는 에러
- 실행이 되기 전에 소스 코드의 구문을 분석 할 때 나는 에러라고 해서 파싱 에러, Parsing Error 라고도 한다.

예외, Exceptions

❖ 예외, Exceptions 이란?

- 문법 상 오류가 없는 소스 코드라면 구문 분석(파싱) 이후에 실행됨
- 이때, **소스 코드 실행 중에 에러가 발생하는 경우** 를 통칭
- 이러한 예외는 예상치 못한 상황이긴 하지만 항상 치명적이지는 않다.

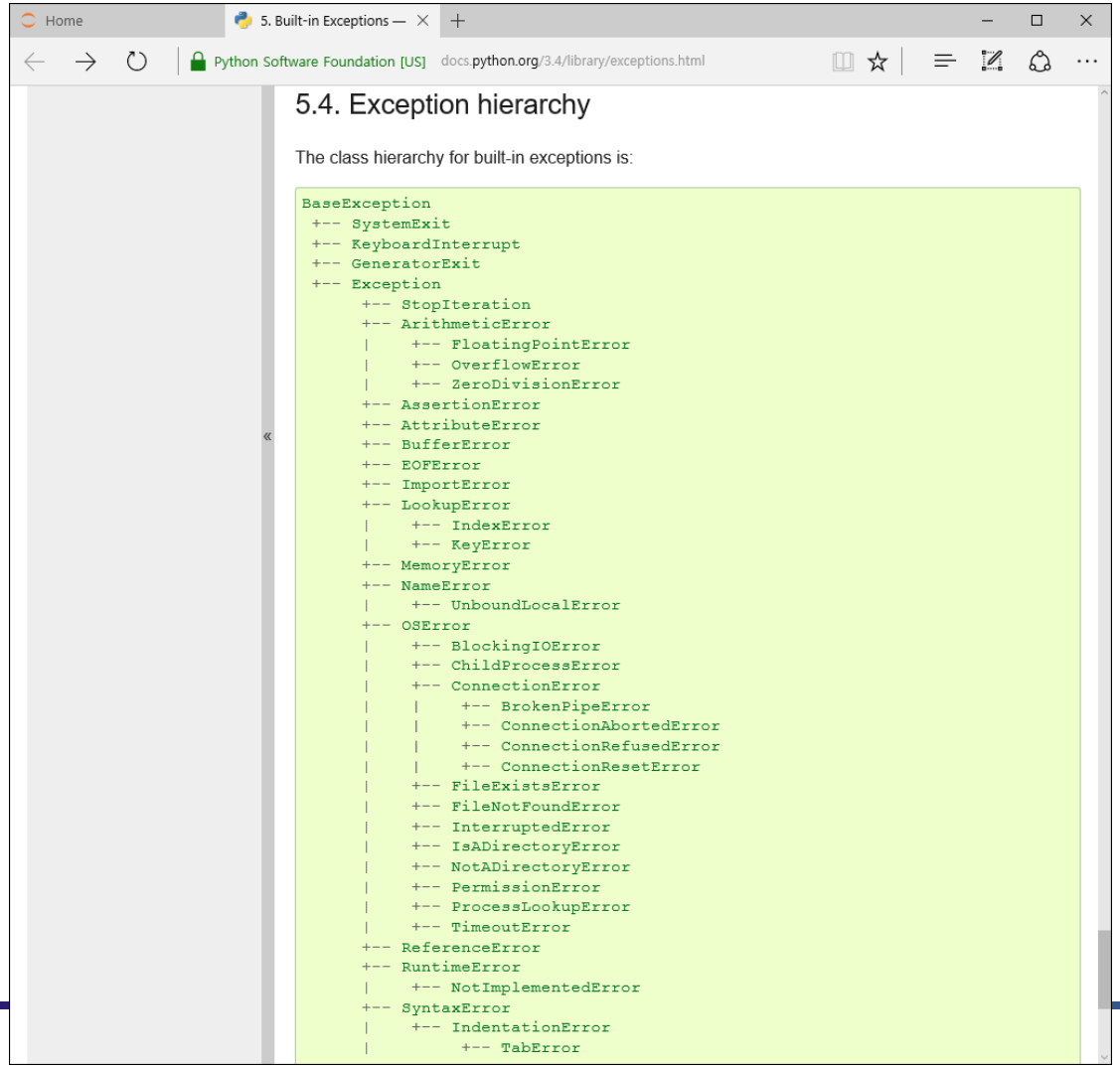
❖ 파이썬에서 정의한 에러 형

- 파이썬은 다양한 유형의 에러를 미리 정해놓고, 데이터 형과 같이 에러 형을 정의하였다.
- 대부분의 에러명은 그것 만으로도 어떤 에러가 발생하였는지 알 수 있으며,
- 상세한 에러 메시지를 출력함으로써 개발자에게 에러가 발생한 위치와 상세 내용을 전달하여 문제 해결에 실마리를 제공한다.
- 미리 정해 놓은 예외들을 **'Built-in Exceptions'** 라고 부른다.

파이썬 표준라이브러리의 Exceptions

❖ Built-in Exceptions 의 목록

< https://docs.python.org/3.4/library/exceptions.html >



5.4. Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
```

try-except 구문으로 예외상황 제어

❖ 예외 상황이 발생하면 어떤 문제가 발생할까?

- 소스코드를 IDLE에서 작성해서 실행시켜 보면, 어떤 에러가 발생하였는지 상세한 설명과 함께 확인이 가능하다.
- 에러 발생시 **추적이 가능한 정보들**을 표기한 에러 메시지를 **트레이스백 Traceback** 메시지라고 부른다.
- 소스 코드 실행 중 에러가 발생하면 프로그램은 중단되게 마련이고, 심각한 장애로 이어질 수 있다.
- 파이썬에서는 이런 예외 상황에 대한 처리를 위하여 **try-except문**을 제공하고 있다.

❖ Exception 구문 사용시 발생한 에러는 삼켜 먹으면 안된다.

- except 구문을 사용하게 되면, 에러 발생시 except 블록문이 수행
- 만약, except 구문이 없었다면 해당 에러가 발생하면 프로그램이 중단
- 하지만, except 구문을 사용하여 에러가 발생하였는데도 불구하고, 프로그램을 단지 멈추지 않게 하는 것은 무척이나 위험한 행동이다.
- 실무현장에서는 보통 에러가 발생하였을 때 어떤 식으로 조치를 해야 하는지 표준이 있기 마련

Exceptions Handling 예시1

❖ 예외상황 테스트를 위한 함수

```
# 예외상황 테스트를 위한 함수
def exception_test():
    print("[1] Can you add 2 and '2' in python? ")
    print("[2] Try it~! ", 2+'2')      # 예외 발생
    print("[3] It's not possible to add integer and string together. ")

exception_test()
```

```
[1] Can you add 2 and '2' in python?

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-c971d69df651> in <module>()
      5     print("[3] It's not possible to add integer and string together. ")
      6
----> 7 exception_test()

<ipython-input-7-c971d69df651> in exception_test()
      2 def exception_test():
      3     print("[1] Can you add 2 and '2' in python? ")
----> 4     print("[2] Try it~! ", 2+'2')      # 예외 발생
      5     print("[3] It's not possible to add integer and string together. ")
      6

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Exceptions Handling 예시2

❖ 예외상황에 대한 처리를 구현한 함수

```
# 예외상황에 대한 처리를 구현한 함수
def exception_test2():
    print("[1] Can you add 2 and '2' in python? ")

    try:
        print("[2] Try it~! ", 2+'2')      # TypeError 발생
    except TypeError:
        print("[2] I got TypeError! ")     # 에러 메시지 출력


    print("[3] It's not possible to add integer and string together. ")

exception_test2()
```

```
[1] Can you add 2 and '2' in python?
[2] I got TypeError!
[3] It's not possible to add integer and string together.
```

Exceptions Handling 예시3

❖ 예외상황에 대한 에러메시지를 상세히 나타낸 함수

```
# 예외상황에 대한 에러메시지를 상세히 나타낸 함수
def exception_test3():
    print("[1] Can you add 2 and '2' in python? ")

    try:
        print("[2] Try it~! ", 2+'2')      # TypeError 발생
    except TypeError as err:
        print("[2] TypeError: {}".format(err))     # 에러 메시지 출력


    print("[3] It's not possible to add integer and string together. ")

exception_test3()
```

```
[1] Can you add 2 and '2' in python?
[2] TypeError: unsupported operand type(s) for +: 'int' and 'str'
[3] It's not possible to add integer and string together.
```

Exceptions Handling 예시4

❖ 처음에 보았던 트레이스백 메시지와 함께 나타낸 함수

```
import traceback

# 처음에 보았던 트레이스백 메시지와 함께 나타낸 함수
def exception_test4():
    print("[1] Can you add 2 and '2' in python? ")

    try:
        print("[2] Try it~! ", 2+'2')      # TypeError 발생
    except TypeError:
        print("[2] I got TypeError! Check below! ")     # 에러 메시지 출력
        traceback.print_exc()                           # 트레이스백 메시지 출력

    print("[3] It's not possible to add integer and string together. ")

exception_test4()
```

```
[1] Can you add 2 and '2' in python?
[2] I got TypeError! Check below!
[3] It's not possible to add integer and string together.
Traceback (most recent call last):
  File "<ipython-input-14-c6979eb4973d>", line 8, in exception_test4
    print("[2] Try it~! ", 2+'2')      # TypeError 발생
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

✓ 트레이스백 메시지를 출력하려면, **traceback 모듈**을 호출해야 한다.

else와 finally 활용하기

❖ 파이썬의 예외 처리 방식

- 파이썬에서는 예외 처리 방식으로 else 라는 옵션 구문을 제공
  - 예외 상황이 발생하면 except 구문을 실행하고, 발생하지 않으면 else 구문을 실행한다.
- 또 다른 예외 처리 방식의 옵션 구문, finally 구문
  - try 블록문 내의 소스 코드에서 예외 상황이 발생하든 발생하지 않든 간에 반드시 실행이 되어야 하는 소스 코드를 finally 블록문에 위치
  - 예외와 상관 없이, finally 블록문에 항상 실행
  - finally 블록문은 실제 프로젝트 수행 시 외부 자원(파일, 네트워크 연결 등)을 열었다가 닫을 때 유용하게 사용이 된다.

Exceptions Handling 예시5

❖ 예외 처리 방식에서의 else 옵션 구문 활용

```
# 예외 처리 방식에서의 else 옵션 구문 활용
def exception_test5(file_path):
    try:
        f = open(file_path, 'r')              # 파일 열기 시도
    except IOError:
        print('cannot open', file_path)       # 에러 메시지 출력
    else:
        print('Fiel has', len(f.readlines()), 'lines')     # 파일 라인 수 출력
        f.close()                             # 파일 닫기

# 정상 상황
exception_test5('C:\Python34\README.txt')
```

```
Fiel has 194 lines
```

```
# 없는 파일을 찾을때
exception_test5('C:\Python34\README_X.txt')
```

```
cannot open C:\Python34\README_X.txt
```

Exceptions Handling 예시6

❖ 예외 처리 방식에서의 finally 옵션 구문 활용

```
# 예외 처리 방식에서의 finally 옵션 구문 활용
def exception_test6(file_path):
    try:
        f = open(file_path, 'r')            # 파일 열기 시도
    except IOError:
        print('cannot open', file_path)     # 에러 메시지 출력
    else:
        print('Fiel has', len(f.readlines()), 'lines')    # 파일 라인 수 출력
        f.close()                           # 파일 닫기
    finally:
        # 예외 발생 상관 없이 무조건 실행
        print('I just tried to read this file.', file_path)

# 정상 상황
exception_test6('C:\Python34\README.txt')
```

```
Fiel has 194 lines
I just tried to read this file. C:\Python34\README.txt
```

```
# 없는 파일을 찾을때
exception_test6('C:\Python34\README_X.txt')
```

```
cannot open C:\Python34\README_X.txt
I just tried to read this file. C:\Python34\README_X.txt
```

사용자 정의 예외

- **파이썬에서는 다양한 Built-in Exceptions을 정의 해놓았지만, 이러한 에러들은 대부분 일반적인 용도로 사용이 된다.**
- **사용자 정의 예외**
  - 의도적으로 본인이 만든 예외를 만들어야 할 경우, 예외 클래스를 새로 만들면 된다.
  - 예외 클래스의 실행은 일반 클래스와는 조금 다르다.
  - 매번 예외 클래스에 대한 객체를 개발자가 생성할 수는 없는 노릇
  - 예외 클래스의 실행을 위해 파이썬에서는 raise 라는 구문을 제공
  - 사용자 정의 예외는 개발자가 작성하는 소스 코드의 예외 상황에 대한 시나리오에 따라서 다양하게 정의하여 활용할 수 있다.
    - 예를 들어, 은행에서 계좌 관련 된 프로그램 작성시
    - 계좌에서 돈을 인출하는 로직을 만들 경우, 반드시 계좌의 잔고를 확인하여 인출하려고 하는 돈의 인출 가능성 유무를 확인해야 한다.
    - 만약, 잔고보다 더 많은 돈을 인출하려고 한다면 예외 상황이 발생하므로 사용자가 정의가 에러를 발생(raise) 시킬 수 있을 것이다.

User Defined Exceptions 예시1

❖ 사용자 정의 예외 클래스, 'TooBigNumError'

```
# 예외 클래스
class TooBigNumError(Exception):
    def __init__(self, val):
        self.val = val
    def __str__(self):
        return 'too big number {}. Use 1~10! '.format(self.val)
```

```
raise TooBigNumError(15)
```

```
---------------------------------------------------------------------------
TooBigNumError                                Traceback (most recent call last)
<ipython-input-28-6e086af09243> in <module>()
----> 1 raise TooBigNumError(15)

TooBigNumError: too big number 15. Use 1~10!
```

✓ 매번 예외 클래스에 대한 객체를 개발자가 생성할 수는 없다.

✓ 이 **예외 클래스의 실행**을 위해 파이썬에서는 **raise** 라는 구분을 제공

User Defined Exceptions 예시2

❖ 사용자 정의 예외를 위한 테스트 함수

```
# 사용자 정의 예외를 위한 테스트 함수
def user_defined_exception_test():
    num = int(input('1부터 10 사이의 점수를 입력하세요! : '))    # 숫자 입력
    if num > 10:
        raise TooBigNumError(num)                               # 에러 발생
    print('숫자 {} 를 입력하셨군요! '.format(num))                 # 정상인 경우 출력문
```

```
# 정상 Case 입력
user_defined_exception_test()
```

```
1부터 10 사이의 점수를 입력하세요! : 5
숫자 5 를 입력하셨군요!
```

```
# 에러 Case 입력
user_defined_exception_test()
```

```
1부터 10 사이의 점수를 입력하세요! : 15

---------------------------------------------------------------------------
TooBigNumError                             Traceback (most recent call last)
<ipython-input-31-0749cb398f53> in <module>()
      1 # 에러 Case 입력
----> 2 user_defined_exception_test()

<ipython-input-29-d20a75db2bf3> in user_defined_exception_test()
      3     num = int(input('1부터 10 사이의 점수를 입력하세요! : '))    # 숫자 입력
      4     if num > 10:
----> 5         raise TooBigNumError(num)                              # 에러 발생
      6     print('숫자 {} 를 입력하셨군요! '.format(num))                # 정상인 경우 출력문

TooBigNumError: too big number 15. Use 1~10!
```

Wrap-up

- 파이썬에서 발생하는 에러와 예외 상황에 대해 살펴보았다.
- 소스 코드가 실행 되기 전에 문법상 오류로 인해 발생 하는 구문 에러(파싱 에러)를 살펴보았다.
- 소스 코드 실행 중 예상치 못한 상황에 의해 발생하는 여러 예외 상황들을 살펴보았다.
- 이런 예외 상황이 발생 하였을 때 프로그램이 비정상적으로 종료되는 것을 피하기 위해, try-except 블록을 통해 제어
- 옵션 블록인 else와 finally 블록도 살펴보았다.
- 사용자 정의 예외 클래스를 작성하는 방법을 살펴보았다.

예외 처리는 초보 개발자에게 다소 어렵거나 간과하고 지나칠 수 있다.

하지만, 예외 처리 없는 소스 코드는 마치 시한폭탄과도 같다.

모든 소스 코드에는 예외 처리를 해야 한다는 각오로 코딩을 하기 바람 !!