# python 例外処理

## 例外（Exception）

- エラー発生→「例外オブジェクト」が作られる
- その例外をキャッチし、エラーの種類や内容から後続処理を決める仕組みのこと
- エラーは大きく２種類
  - プログラムの「実行前」にわかるエラー（例：SyntaxError）
  - プログラムの「実行後」にわかるエラー（例：ExceptionError）
- 例外オブジェクト
  - エラーの種類や内容が記載されている
  - Exceptionクラスを親として、種類ごとに複数分類されている

## try~except

- 例外を捕まえる
- 例外が発生すると処理が中断されるが、例外が発生しても処理の実行を続けたい、または終了処理を行いたい場合もある
- try ~ except で例外を捕まえ、処理を分ける

```
import sys
for fn in sys.argv[1:]:
      try:                          ←例外が発生したらexcept文へジャンプ
            f = open(fn)
      except FileNotFoundError:  ←クラスを指定して特定の例外だけを受け取る※
            print("{}というファイルは存在しません".format(fn))
      else:                         ←例外が発生しなかった場合の処理
            try:                      elseはexceptより後に記載
                  print(fn, len(f.read()))
      finally:                      ←例外の発生有無に関わらず、実行する処理
            f.close()
```

※：複数列記も可能（例外クラスを丸括弧で囲み、カンマで区切る）

```
while True:
  try:
    x = int(input(‘数値を入れてください：’))
    break
  except ValueError: ・・・※
    print(‘有効な数値ではありません。もう一度どうぞ...’)
```

※箇所において、例外の型が「ValueError」でなかった場合、送出された例外はさらに外側にあるtry文に渡される（try文が入れ子になっている場合）。ハンドラが見つからなければ、これは未処理例外（unhandled exception）となる。

## With文

- 例外とよく似た機能をもつ
- ファイルが存在しなかったらWithブロックに入る前に例外が発生するため、With内の処理は実行されない
- 例外に比べ簡潔で済む

```
with open(fn) as f:
      for line in f:
            print(line)
```

## 例外とトレースバック

- 例外を使うことの最大の利点
    - 「エラーの発生位置とエラー処理を分離することできる」
- raiseやトレースバックはよくわからないので飛ばす

## 実行前のエラー（SyntaxError：構文エラー）

- SyntaxError
    - 文法、構文エラー
    - クオーテーションや括弧がない、インデント処理、など

## 実行中のエラー（Exception：例外エラー）

- NameError
    - 未定義の変数や関数を利用（代入を含む）した時
    - インポートしていないモジュールを利用した時、など
    - ほとんどがタイプミス

- AttributeError
  - オブジェクトやメソッドに未定義の属性を参照しようとした時
  - 関数に引数を渡す場合等に期待している型と異なる型のOBJを渡した時、等
  - ほとんどがタイプミス
- TypeError
  - 処理の過程でふさわしくない型のOBJが使われた時
  - 文字列と数値を足す、など
  - リストの要素をインデックスで参照する際、整数以外のOBJをインデックスとして使った場合、など
  - 関数の引数として、想定されていない型のオブジェクトが渡された時
  - 例
    - unsupported operand type(s) for +: ‘int’ and ‘str’
    - cannot concantenate ‘str’ and ‘int’ objecgts
    - TypeError: list indices must be integers
      - リストのインデックスは整数でなければならない
    - Iteration over non-sequence
      - シーケンス型でないOBJを使ってイテレーションした
- IndexError
  - リストのようなシーケンスをインデックスで参照しようとした場合
  - シーケンスの要素数を超える数を指定した場合
- KeyError
  - 辞書型の要素をキーで参照する際、存在しないキーを指定した場合
- ImportError
  - インポートするモジュール、関数が見つからない等
- UnicodeDecodeError, UnicodeEncodeError
  - 文字列のデコード、エンコードのエラー
- ZeroDivisonError
  - 数値を０で割ろうとした場合

```
def spam(divide_by):
    try:
        return 42 / divide_by
    except ZeroDivisionError:
        print('不正な引数です')

print(spam(2))
print(spam(12))
print(spam(0)) ←これだけが終了する
print(spam(1))

21.0
3.5
不正な引数です
None
42.0
```

# 例外を起こす

- 例外を起こすとは「コード実行を停止して、プログラム実行をexcept文に移せ」という意味

- try～except、raiseを利用した例

```
def box_print(symbol, width, height):
    if len(symbol) != 1:
        raise Exception('symbolは１文字でなければならない')
    if width <= 2:
        raise Exception('widthは２より大きくなければならない')
    if height <= 2:
        raise Exception('heightは２より大きくなければならない')
    print(symbol * width)
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)
for sym, w, h, in (('*', 4, 4), ('0', 20, 5), ('x', 4, 3), ('/', 3, 3)):
    try:
        box_print(sym, w, h)
    except Exception as err:
        print('例外が起こりました： ' + str(err))

****
*  *
*  *
****
00000000000000000000
0                  0
0                  0
0                  0
00000000000000000000
例外が起こりました： widthは２より大きくなければならない
例外が起こりました： symbolは１文字でなければならない
```

# トレースバック

- エラー情報、のこと
- 記載されている内容
    - エラーMSG
    - エラーの起こった行番号
- エラーに至る関数呼び出しの並び（コールスタック）
- どの呼び出しがエラーを招いたかわかる

```
1 def spam():
2     bacon()
3
4 def bacon():
5     raise Exception('これはエラーMSGです')
6
7 spam()
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-45-378937f16f03> in <module>
      5     raise Exception('これはエラーMSGです')   ←③spam()は7行目から呼び出されている
      6
----> 7 spam()

<ipython-input-45-378937f16f03> in spam()   ←②:①のbacon()は2行目でspam()関数で呼び出されている
      1 def spam():
----> 2     bacon()
      3
      4 def bacon():
      5     raise Exception('これはエラーMSGです')

<ipython-input-45-378937f16f03> in bacon()
      3
      4 def bacon():
----> 5     raise Exception('これはエラーMSGです')   ←①:5行目でエラー発生:in bacon()
      6
      7 spam()

Exception: これはエラーMSGです
```

- 例外がきちんと処理されないと常にトレースバックが表示される
- traceback.format_exc()を呼び出すと文字列として取得可能
- 使い方としては、例外が起こったらプログラムを異常停止させず、トレースバック情報をログに書き出し、プログラムの実行を継続可能
- 後でログファイルを確認し、でバックすることができる

```
import pathlib
f_p = pathlib.Path(r'/Users/mbp441/Desktop/errorInfo.txt')

import traceback
try:
    raise Exception('これはエラーMSGです')
except:
    error_file = open(f_p, 'w')
    error_file.write(traceback.format_exc())
    error_file.close()
    print('トレースバック情報をerrorInfo.txtに書き込みました')
```

```
# トレースバック情報をerrorInfo.txtに書き込みました

# Traceback (most recent call last):
#   File "<ipython-input-48-760621251615>", line 6, in <module>
#     raise Exception('これはエラーMSGです')
# Exception: これはエラーMSGです
```

# アサート

- プログラムが正常に動けば通る処理だが、念のためチェックしておきたい時に利用
- 本番では使えない
- 使用する目的は、バグの根本原因と思われる箇所を素早く見つけ出せるようにするため。
- 構文
    - assert 条件式, 条件式がFalseの場合に出力するメッセージ
    - 条件式が True の場合は何も起こりません。
    - 条件式が False の場合、AssertionError の例外が発生し、必要に応じEMGが生成される

```
>>> kitai = 100
>>> input = 1
>>> assert kitai == input, '期待する値[{0}], 入力値[{1}]'.format(kitai, input)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: 期待する値[100], 入力値[1]
```

ちゃんと例外を処理するならこんな感じ。

```
>>> try:
...     kitai = 100
...     input = 1
...     assert kitai == input, '期待する値[{0}], 入力値[{1}]'.format(kitai, input)
... except AssertionError as err:
...     print('AssertionError :', err)
...

AssertionError : 期待する値[100], 入力値[1]
```

```
def f(a, b):
    assert type(a) == int, 'invalid a'
    assert type(b) == str, 'invalid b'
    return a + int(b)

# 本番モードでは以下になってしまうため
def f(a, b):
    return a + int(b)


# 本来であればこう書くべき（本番モード）
def f(a, b):
    if not isinstance(a, int):
        raise TypeError('invalid a')
    if not isinstance(b, str):
        raise TypeError('invalid b')
    return a + int(b)
```