プログラミング初級（Python）

例外処理と応用的な関数

早稲田大学グローバルエデュケーションセンター

堅牢で有用なプログラム

もし自分のコードが完璧だと思ったら、それはまだ
十分にテストされていないか、不完全なものです。

— スティーブ・マコネル

Pythonにおけるエラーの種類

- これまでは、エラーメッセージについては簡単に触れるだけだった。
- エラーには、大きく分けて２つの種類がある。
  ：**構文エラー**(syntax error)と**例外**(exception)
- 構文エラーはプログラムを実行する前の**構文解析の段階で検出されるエラー**。構文解析エラー（parsing error）とも呼ばれる。**命令を書き間違えていると起こる**。
- 構文エラーの例：文字列リテラルの終わりの「'」がない例。

```
>>> print('Hello, error!)
  File "<stdin>", line 1
    print('Hello, error!)
          ^
SyntaxError: unterminated string literal (detected at line 1)
```

- 例外は、プログラムの**実行段階で検出されるエラー**。命令の書き方は間違えていなくても、**Pythonで処理できない想定外の命令が実行されたときに起こる**。

例外の例

- 例外はこれまでに経験的にはいくつか見てきた。
- 具体例：IndexError, TypeError, ValueError, IndentationError, etc.

```
1 >>> letters =
  'abcdefghijklmnopqrstuvwxyz'
2 >>> letters[100]
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  IndexError: string index out of range
3 >>> name = 'Cython'; name[0] = 'P'
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: 'str' object does not support item assignment
4 >>> int('31 bottles of beer')
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  ValueError: invalid literal for int() with base 10: '31 bottles of beer'
5 >>> a = 1
6 >>>   print(a)
    File "<stdin>", line 1
      print(a)
  IndentationError: unexpected indent
```

エラーメッセージの最終行で何が起こったかが示される。

エラーメッセージの最終行で何が起こったかが示される。

エラーメッセージの最終行で何が起こったかが示される。

エラーメッセージの最終行で何が起こったかが示される。

例外処理とは？

- 例外はPythonにとって想定外なものであるため異常と判断され、何も対処しなければそのままプログラムは強制終了してしまう。
- 強制終了は、異常終了またはクラッシュ(crash)ともいう。
- プログラムが突発的に使用不能になることはよくない習慣とされる。
- しかし、プログラムの構造上、例外がそもそも発生しえないプログラムを作るということは困難。
- 一方、例外が発生しても想定内のものとしてプログラマに有用なメッセージを出力したり、プログラムを通常どおりに終了することは可能。
- このように、**例外を適切に処理すること**を**例外処理**という。
- 基本の構文は**try-except文**。
- **as節**は省略可能。
- オプションで、**else節**と**finally節**があり。

構文

```
try:
    例外が発生しえるコード
except 例外の種類 as 例外変数:
    例外発生時の処理
```

例外処理：基本の構文とas節の活用

1
```
>>> num = int(input('Please enter a
number:'))
Please enter a number:3.14
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with
base 10: '3.14'
```

2
```
>>> num = int(input('Please enter a
number:'))
Please enter a number:a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with
base 10: 'a'
```

3
```
>>> try:
...     num = int(input('Please enter a
number:'))
... except ValueError:
...     print("Invalid input: The entered
value is not a valid number.")
...
Please enter a number:3.14
Invalid input: The entered value is not a
valid number.
```

これより基本の構文。

エラーメッセージというよりもprint関数でメッセージを通常の扱いで出している。

4
```
>>> try:
...     num = int(input('Please enter a
number:'))
... except ValueError as ex:
...     print("Invalid input: The entered
value is not a valid number.", ex)
...
Please enter a number:a
Invalid input: The entered value is not a
valid number. invalid literal for int()
with base 10: 'a'
```

as節を使うとPython側の固有のメッセージを活用できる。

例外処理：while文の併用、except節の併記、else節の活用

5
```
>>> while True:
...     try:
...         num = int(input("Please enter a
number: "))
...         print(f'You entered {num}')
...         break
...     except ValueError:
...         print("Oops! That was no valid
number. Try again...")
...
Please enter a number: a
Oops! That was no valid number. Try again...
Please enter a number: 3.14
Oops! That was no valid number. Try again...
Please enter a number: 3
You entered 3
```

While文を併用して例外がなくなるまで実行し続けるというプログラムは定番。

6
```
>>> while True:
...     try:
...         a = float(input('Please enter a
real number a:'))
...         b = float(input('Please enter a
real number b:'))
...         result = a / b
...     except ValueError as ex:
...         print('An error occurred:', ex)
...     except ZeroDivisionError as ex:
...         print('Zero division occurred:',
ex)
...     else:
...         print(f'Result: {result}')
...         break
...
Please enter a real number a:hoge
An error occurred: could not convert string
to float: 'hoge'
Please enter a real number a:3.14
Please enter a real number b:0
Zero division occurred: float division by
zero
Please enter a real number a:3.14
Please enter a real number b:3.14
Result: 1.0
```

except節の併記。複数、書いてよい。

else節の活用。例外が起こらなかったときにだけ実行されるブロック。

例外処理：全ての例外を指すExceptionとfinally節

```
>>> try:
...     num = int(input('Please enter an
integer:'))
...     result = 1 / num
... except ZeroDivisionError as ex:
...     print('Error: Division by zero is
not allowed.', ex)
... except Exception as ex:
...     print('An unexpected error
occurred:', ex)
...
Please enter an integer:a
An unexpected error occurred: invalid
literal for int() with base 10: 'a'
>>>
```

Exceptionは全ての例外を指すので特定の例外を指定できないときに便利。

しかし、バグ取りの際は、原因が一意に定まらないので不便になることも。

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")

divide(2, 1)
result is 2.0
executing finally clause
divide(2, 0)
division by zero!
executing finally clause
divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for
/: 'str' and 'str'
```

finally節の活用。例外の有無にかかわらず、実行されるブロック。

話は関数に戻りまして

- 関数の応用的な使い方を学ぶ。
  - 引数の応用的な扱い方
    位置引数、キーワード引数、デフォルト引数、可変長引数
  - **オブジェクトとしての関数**
  - 関数内関数
  - クロージャ
  - **無名関数（ラムダ関数）**
  - ジェネレータ関数
  - デコレータ
  - **再帰関数**

- 上記の規則と例外処理を組み合わせていくことで、これまでよりもさらに実用的なプログラムを書けるようになる。

位置引数とキーワード引数

```
1 >>> def menu(wine, entree, dessert):
...     return {'wine': wine, 'entree':
entree, 'dessert': dessert}
...
2 >>> menu('chardonnay', 'chicken', 'cake')
{'wine': 'chardonnay', 'entree': 'chicken',
'dessert': 'cake'}
3 >>> menu('beef', 'bagel', 'bordeaux')
{'wine': 'beef', 'entree': 'bagel',
'dessert': 'bordeaux'}
```

位置引数。これまで行ってきた方法。使うには一番簡単であるので、よく使われるが、引数の位置が意味することを覚えておかなければならない。

キーワード引数。仮引数名を覚えておけば位置を覚えておかなくても大丈夫になる。

```
4 >>> menu(entree='beef', dessert='bagel',
wine='bordeaux')
{'wine': 'bordeaux', 'entree': 'beef',
'dessert': 'bagel'}
5 >>> menu('frontenac', dessert='flan',
entree='fish')
{'wine': 'frontenac', 'entree': 'fish',
'dessert': 'flan'}
6 >>> menu('frontenac', dessert='flan',
'fish')
  File "<stdin>", line 1
    menu('frontenac', dessert='flan',
'fish')

^
SyntaxError: positional argument follows
keyword argument
7 >>> menu('frontenac', dessert='flan',
entre='fish')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: menu() got an unexpected keyword
argument 'entre'
```

位置引数とキーワード引数の併用。

しかし、キーワード引数を指定した後に位置引数は指定できない。

キーワード引数の名前を間違えた例。

デフォルト引数

```
8  >>> def menu(wine, entree,
   dessert='pudding'):
   ...     return {'wine': wine, 'entree':
   entree, 'dessert': dessert}
   ...
9  >>> menu('chardonnay', 'chiken')
   {'wine': 'chardonnay', 'entree': 'chiken',
   'dessert': 'pudding'}
10 >>> menu('dunkelfelder', 'duck',
   'doughnut')
   {'wine': 'dunkelfelder', 'entree': 'duck',
   'dessert': 'doughnut'}
11 >>> menu('dunkelfelder', 'duck',
   dessert='doughnut')
   {'wine': 'dunkelfelder', 'entree': 'duck',
   'dessert': 'doughnut'}
```

デフォルト引数

通常、実引数と仮引数の数が合わないとエラーになるが、デフォルト引数があると…

デフォルト引数を位置引数で上書きする例。

デフォルト引数をキーワード引数で上書きする例。

```
12 >>> def menu(wine, entree='bagel',
   dessert='pudding'):
   ...     return {'wine': wine, 'entree':
   entree, 'dessert': dessert}
   ...
13 >>> menu('dunkelfelder', 'duck',
   dessert='doughnut')
   {'wine': 'dunkelfelder', 'entree': 'duck',
   'dessert': 'doughnut'}
14 >>> def menu(wine, entree='bagel',
   dessert):
     File "<stdin>", line 1
       def menu(wine, entree='bagel',
   dessert):
                                      ^^^^^^^
   SyntaxError: parameter without a default
   follows parameter with a default
```

複数のデフォルト引数。

デフォルト引数を指定するとそれ以降の引数もデフォルト引数を指定しないとエラーになる。

デフォルト引数は**仮引数**に指定するもの。
キーワード引数は**実引数**に指定するもの。

引数にミュータブルな値を利用する際の注意

```
15 >>> def append(param, result=[]):
   ...     result.append(param)
   ...     print(result)
   ...
16 >>> append('a')
   ['a']
17 >>> append('b')
   ['a', 'b']
18 >>> def append(param):
   ...     result = []
   ...     result.append(param)
   ...     return result
   ...
19 >>> append('a')
   ['a']
20 >>> append('b')
   ['b']
21 >>> def append(param, result=None):
   ...     if result is None:
   ...         result = []
   ...     result.append(param)
   ...     print(result)
   ...
```

デフォルト引数が評価されるのは関数が定義されたとき。

空のリストに毎回appendするという想定だと意図どおりにはならない。

```
22 >>> append('a')
   ['a']
23 >>> append('b')
   ['b']
24 >>> def test(arg):
   ...     arg[1] = 'hoge'
   ...
25 >>> outside = [1, 2, 3]
26 >>> test(outside)
27 >>> outside
   [1, 'hoge', 3]
```

global文同様、このような使い方はあまりしないほうが良い。必要ないときもどこかのタイミングで当該の関数を利用したことにより、グローバル変数の値が、気付かぬままに変わってしまっていることがあり得る。

また、グローバル変数を更新する関数がたくさんあると問題が生じた際にどの関数が原因となっているのか分かりにくくなる。

どうしても必要な際は、docstringで丁寧な説明を付与したり、元の値に影響を与えない新しいオブジェクトを操作対象にしたりなど、対策が望まれる。

可変長引数（タプルと辞書）

```
>>> def print_param(*params):
...     print('Params: ', params)
...
>>> print_param()
Params:  ()
>>> print_param(3, 2, 1, 'Go!')
Params:  (3, 2, 1, 'Go!')
>>> def print_param(param1, param2,
*params):
...     print('Param1: ', param1)
...     print('Param2: ', param2)
...     print('Params: ', params)
...
>>> print_param('Windows', 'Mac', 'Linux',
'ChromeOS', 'FreeBSD')
Param1:  Windows
Param2:  Mac
Params:  ('Linux', 'ChromeOS', 'FreeBSD')
>>> params = ('Windows', 'Mac', 'Linux',
'ChromeOS', 'FreeBSD')
>>> print_param(*params)
Param1:  Windows
Param2:  Mac
Params:  ('Linux', 'ChromeOS', 'FreeBSD')
```

*を1つつけると引数がタプルの可変長になる。

何個引数が必要が未定の場合に便利。

位置引数と可変長引数の併用。

実引数で可変長引数の変数を渡しても良い。

```
>>> *params
  File "<stdin>", line 1
SyntaxError: can't use starred expression here
>>> def print_param(param1, *params,
param2):
...     print('Param1: ', param1)
...     print('Param2: ', param2)
...     print('Params: ', params)
...
>>> print_param('Windows', 'Mac', 'Linux',
'ChromeOS', 'FreeBSD')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: print_param() missing 1 required
keyword-only argument: 'param2'
>>> def print_kwarams(**kwarams):
...     print('Keyword params: ', kwarams)
...
>>> print_kwarams()
Keyword params:  {}
>>> print_kwarams(os='Linux', cpu='Intel',
pc='Apple')
Keyword params:  {'os': 'Linux', 'cpu':
'Intel', 'pc': 'Apple'}
```

可変長引数の表記は仮引数か実引数でしか使えない。

可変長引数の後に通常の引数を書こうとするとエラーになる。

**とすると引数が辞書の可変長になる。

辞書として返される。

キーワード専用引数

```
41 >>> def print_data(data, *, start=0,
   end=100):
   ...     for value in data[start:end]:
   ...         print(value)
   ...
42 >>> print_data(['a', 'b', 'c', 'd', 'e',
   'f'], start=2)
   c
   d
   e
   f
43 >>> print_data(['a', 'b', 'c', 'd', 'e',
   'f'], 2)
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   TypeError: print_data() takes 1 positional
   argument but 2 were given
```

`*`以降はキーワード専用引数にする設定。

位置引数として書くことはできない。

```
44 >>> def print_data(data, start=0, end=100):
   ...     for value in data[start:end]:
   ...         print(value)
   ...
45 >>> print_data(['a', 'b', 'c', 'd', 'e',
   'f'], start=2)
   c
   d
   e
   f
46 >>> print_data(['a', 'b', 'c', 'd', 'e',
   'f'], 2)
   c
   d
   e
   f
```

復習ついでに通常の形を復習。

仮引数における*での縛りがなければ位置引数、キーワード引数どちらでも受け付けられる。

Pythonにおいて「*」の語法は色々あるのでしっかり区別が必要。

オブジェクトとしての関数、関数内関数

```
>>> def say_hello(name):
...     print(f"Hello, {name}!")
...
>>> say_hello('Ami')
Hello, Ami!
>>> greet = say_hello
>>> greet('Ami')
Hello, Ami!
>>> def apply(x):
...     result = square(x)
...     return result
...
>>> def square(x):
...     return x * x
...
>>> apply(5)
25
>>> def apply(func, x):
...     result = func(x)
...     return result
...
>>> def square(x):
...     return x * x
...
```

```
>>> apply(square, 5)
25
>>> apply(greet, 'Ami')
Hello, Ami!
>>> def apply(func, x):
...     return func(x)
...
>>> apply(square, 5)
25
>>> apply(greet, 'Ami')
Hello, Ami!
>>> def outer(a, b):
...     def inner(c, d):
...         return c * d
...     return inner(a, b)
...
>>> outer(3, 1)
3
```

関数内関数の定義。

関数内関数の呼び出し。

クロージャ、無名関数（ラムダ関数）

```
>>> def outer2(a, b):
...     def closure():
...         return a * b
...     return closure
...
>>> outer2(3, 1)
<function outer2.<locals>.closure at
0x104ae3e20>
>>> closure1 = outer2(3, 1)
>>> closure2 = outer2(4, 1)
>>> closure1
<function outer2.<locals>.closure at
0x104ae3f60>
>>> closure2
<function outer2.<locals>.closure at
0x104ae3ce0>
>>> closure1()
3
>>> closure2()
4
```

関数内関数は外の関数の値を覚えられる。

値を動的に変えて関数を生成できる。

```
>>> numbers = (1, 2, 3)
>>> def edit(data, func):
...     for datum in data:
...         print(func(datum))
...
>>> def double(number):
...     return number * 2
...
>>> edit(numbers, double)
2
4
6
>>> edit(numbers, lambda num: num * 2)
2
4
6
>>> def circle(r):
...     return r * r * 3.14
...
>>> circle(3)
28.26
>>> circle = lambda r: r * r * 3.14
>>> circle(3)
28.26
```

ラムダ文は無名関数にすることが目的だが、あえて変数に代入して名前をつけても良い。この場合は単に記述量の削減が目的となる。

ジェネレータ関数

```
>>> def my_generator():
...     yield 1
...     yield 2
...     yield 3
...
>>> gen = my_generator()
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
3
>>> def my_range(first=0, last=10, step=1):
...     number = first
...     while number < last:
...         yield number
...         number += step
...
>>> my_range
<function my_range at 0x104ae8360>
>>> for x in my_range(1, 4):
...     print(x)
...
1
2
3
>>> genobj = (num for num in range(1, 4))
>>> genobj
<generator object <genexpr> at 0x10484d780>
>>> for x in genobj:
...     print(x)
...
1
2
3
>>> def my_range(first=0, last=10, step=1):
...     number = first
...     while number < last:
...         return number
...         number += step
...
>>> my_range(1, 4)
1
```

値を一度に全て生成しない。これでメモリに収まる以上の大きさのシークエンスでも作成でき、効率的に処理できる。組み込みのrange関数はジェネレータの一種である。
ジェネレータはイテレーションを通じて要素を生成し、必要な値だけをメモリに保持する。これにより、大規模なデータセットや無限のシーケンスを効率的に扱うことができる。ジェネレータは、for ループを始めとする反復処理に適している。

yield文を使うのが特徴。

タプルの内包表記かのように見えるこれは、ジェネレータ内包表記。

return文を使うと関数が終了する。

デコレータ、再帰関数

```
93 >>> def my_decorator(func):
   ...     def wrapper():
   ...         print("Pre-processing")
   ...         func()
   ...         print("Post-processing")
   ...     return wrapper
   ...
94 >>> @my_decorator
   ... def say_hello():
   ...     print("Hello!")
   ...
95 >>> say_hello()
   Pre-processing
   Hello!
   Post-processing
```

```
fac(5)
5 * fac(4)
5 * (4 * fac(3))
5 * (4 * (3 * fac(2)))
5 * (4 * (3 * (2 * fac(1))))
5 * (4 * (3 * (2 * (1 * fac(0)))))
5 * (4 * (3 * (2 * (1 * 1))))
5 * (4 * (3 * (2 * 1)))
5 * (4 * (3 * 2))
5 * (4 * 6)
5 * 24
120
```

```
96 >>> def dive():
   ...     return dive()
   ...
97 >>> dive()
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
     File "<stdin>", line 2, in dive
     File "<stdin>", line 2, in dive
     File "<stdin>", line 2, in dive
     [Previous line repeated 996 more times]
   RecursionError: maximum recursion depth
   exceeded
98 >>> def fac(n):
   ...     if n == 0:
   ...         return 1
   ...     else:
   ...         return n * fac(n - 1)
   ...
99 >>> print(fac(5))
   120
```

まとめ

- エラーには**構文エラー**と**例外**の２種類ある。
- 構文エラーは**プログラムの構文解析時に検出**されるものであり、例外は**プログラムの実行時に検出**されるもの。
- 例外が発生するとプログラムが強制終了してしまうので、対処が必要。それを**例外処理**と呼ぶ。
- 例外処理の基本的な構文に、**try-except文**がある。**as節**、**else節**、**finally節**は必要に応じて併用できるオプション。
- 関数の引数の様々な指定方法：**位置引数**、**キーワード引数**、**デフォルト引数**、**可変長引数**（*でタプルに展開、**で辞書に展開）
- 基本のデータ型と同様に**関数もオブジェクト**の一種。
- 関数の様々な定義方法：関数内関数、クロージャ、**無名関数（ラムダ関数）**、ジェネレータ関数、デコレータ、**再帰関数**（深さに制限あり）

次回予告　**オブジェクト指向**