Pythonの黒魔術

カウプラン機関
極東支部

技術書典8
（2020年春）

技術書典8
（2020年春）

Pythonの黒魔術

カウプラン機関極東支部

Kauplan Press

はじめに

黒魔術は滅びぬ、何度でもよみがえるさ！
黒魔術の力こそ人類の夢だからだ！

本書の目的

本書はタイトルの通り、Python の黒魔術を説明した本です。ここでいう「黒魔術」とは、高度でマニアックな技術のことを指します。本書では、Python の黒魔術を理解して使いこなせるようになること、また黒魔術が使われたソースコードが読めるようになることを目的とします。

なお本書では Python3 を対象とします。またサンプルプログラムはすべて Python 3.8 で動作確認しています。

本書の背景

Python の人気フレームワークや有名ライブラリでは、黒魔術が使われています。

- Flask では「Thread Local Storage」という黒魔術が、
- SQLAlchemy では「演算子オーバーロード」という黒魔術が、
- pytest では「構文木変換」という黒魔術が使われています。

しかし Python 界隈では「黒魔術は使うべからず」とされているせいで、これらの技術や知識を学ぶ機会は限られています。そのため、若者がフレームワークのソースコードを読んでも理解ができないという事態が発生しています。

考えてみたらおかしなことだと思いませんか？ 大人はみんな「黒魔術は使うな」と言う、けど現実のフレームワークやライブラリでは黒魔術が使われている、そのせいで若者がソースコードを読んでもよく分からない。

まるで、学生には「髪を染めるな、お化粧をするな」と言っておきながら、いざ社会に出ると「お化粧ぐらいできるようになっておけ」と言い出す世間のようです。言いつけを守っていた真面目な子より、親に隠れてこっそり化粧していた子のほうが卒業後は高評価されるという矛盾した現実。

こんな現実はおかしいです。現実のソースコードでは黒魔術が使われていることを認め、きちんとその技術や知識を教える機会が必要ではないでしょうか。

本書はそのための一冊です。

本書の内容

本書では、数多くの黒魔術を説明するだけでなく、必ず実用例も紹介しています。実用例を数多く知ることで、黒魔術は役に立つのだと実感してもらえたらと思います。

本書の内容は次の通りです。

- 第 1 章「スタックフレーム」(p.1) では、関数の呼び出し元を調べる方法を説明します。実用例：f-文字列のシミュレートや、`@property` の改良など。
- 第 2 章「特殊メソッド」(p.19) では、オブジェクトを辞書のようにアクセスしたり、辞書をオブジェクトのようにアクセスする方法を説明します。実用例：自動拡張される辞書や、リストと辞書を兼ねたようなデータなど。
- 第 3 章「演算子」(p.35) では、演算子を上書きする方法を説明します。実用例：日付操作の簡単化や、パースせずに構文木を作る方法など。
- 第 4 章「デコレータ」(p.81) では、関数デコレータの高度な使い方を紹介します。実用例：高速化のためのメモ化や、ベンチマーク用デコレータなど。
- 第 5 章「関数とコードオブジェクト」(p.115) では、関数のさまざまなメタ情報を引き出す方法を説明します。実用例：クロージャの高速化や、ユニットテストのフィクスチャを DI 化する方法など。
- 第 6 章「抽象構文木」(p.151) では、抽象構文木を書き換える方法を説明します。実用例：assert 文のメッセージを自動設定。
- 第 7 章「オブジェクトとメソッド」(p.161) では、Python でのオブジェクトやクラスやメソッドの仕組みを説明します。実用例：特異メソッド、ミックスイン。
- 第 8 章「ディスクリプタ」(p.197) では、メソッド呼び出しやプロパティの仕組みを支えるディスクリプタ機能について説明します。実用例：プロパティ値のキャッシュなど。
- 第 9 章「メタクラス」(p.217) では、クラスオブジェクトを操作する方法を説明します。実用例：O/R マッパーなど。
- 第 10 章「for 文と with 文」(p.239) では、with 文と for 文のしくみを説明します。実用例：キャッシュ用 DSL、ベンチマークツール。
- 第 11 章「例外とトレースバック」(p.273) では、例外に関するテクニックを説明します。実用例：トレースバックを間引いたり、カラーで表示する。
- 第 12 章「グローバル変数とローカル変数」(p.291) は、グローバル変数とローカル変数のテクニックです。実用例：細かい条件でトレースバックを間引く。
- 第 13 章「モジュール」(p.303) では、モジュールに関するテクニックを説明します。実用例：1 ファイルで複数のモジュールに分割。

各章は独立しているので、どの章から読んでも構いません。興味のある章から読むといいでしょう。

本書で扱わないこと

本書では、次のことは扱いません。

- The Zen of Python[*1]
- Pythonic Way[*2]

対象読者

本書は、Python をある程度使いこなしている中級者以上を対象にしています。Python を始めたばかりの初心者や、入門書を終えたばかりの初級者は、申し訳ありませんが本書の対象読者ではありません。

具体的には次のようなレベルの人を対象としています。

- 「`[x for x in range(1, 11) if x % 2]`」の文法が分かる
- 「`(lambda x, y: x + y)(3, 4)`」の結果が予想できる
- 「`x=1`」と「`fn(x=1)`」の違いが説明できる
- 「`return`」と「`yield`」を使い分けられる
- 「`@property`」が何なのか分かる

サポートサイト

本書のサポートサイトで正誤表を公開しています。またサンプルプログラム[*3]もダウンロードできます。

- https://kauplan.org/books/pythonmagic/

謝辞

本書は次の方にレビューしていただきました（順不同）。ありがとうございます。

- ゴリラ氏 (https://twitter.com/gorilla0513)
- @74th 氏 (https://twitter.com/74th/)
- @it_ojisan4321 氏 (https://twitter.com/it_ojisan4321)

特に@74th 氏にはいくつか重大な間違いを指摘いただきました。その深い知識に感謝します。

なおこの本に何か不備や問題点があったとしても、その責任はあくまで著者にあり、氏らにはありません。氏らに迷惑のかかるようなまねはしないようお願いします。

[*1] 「The Zen of Python って何？」と思った人はおそらく本書の対象読者ではありません。

[*2] 「Pythonic Way って何？」と思った人はおそらく本書の対象読者ではありません。

[*3] サンプルプログラムはすべて Python 3.8 で動作確認しています。

表記ルール

本書では次のような表記を使います。

- 背景色がグレーだと、プログラムを表す。
- 背景色が黒色だと、ターミナル操作を表す。
- **太字**は強調か、または前回からの追加を表す。
- ~~取り消し線~~は前回からの削除を表す。
- グレーの「←」はプログラムのコメントを表す。
- グレーの「⇒」は出力結果か、または発生する例外クラスを表す。

いくつか例を示します。

▼背景色がグレーなのでプログラムを表す

```
def f(x):
    return x + 1      ← 太字なので強調を表す
print(f(10))          ⇒ 11
```

▼上のプログラムとの差分を表示する

```
def f(x, y=1):        ← 太字は追加を表す
    return x + 1      ← 取り消し線は削除を表す
    return x + y      ← 太字は追加を表す
print(f(10, 2))       ⇒ 12
```

▼背景色が黒色なのでターミナル操作を表す

```
$ python3
>>> 1 + 1
2
```

目次

はじめに **i**

表記ルール iv

第 1 章 スタックフレーム **1**

1.1 スタックフレームとは 2
1.2 呼び出し元のスタックフレームにアクセスする 6
1.3 呼び出し元の情報を取得する 7
1.4 呼び出し元のローカル変数にアクセスする 8
1.5 `inspect` モジュール 11
1.6 実用例：f-文字列をシミュレート 12
1.7 実用例：現在行のファイル名と行番号 13
1.8 実用例：`getter` と `setter` 14

第 2 章 特殊メソッド **19**

2.1 添字アクセス 20
2.2 属性アクセス 23
2.3 より強力な属性アクセス 25
2.4 関数呼び出し 27
2.5 その他の特殊メソッド 30
2.6 実用例：AutoDict 31
2.7 実用例：HTML 要素 32

第 3 章 演算子 **35**

3.1 演算子に対応した特殊メソッド 36
3.2 演算子の左右を入れ替えた特殊メソッド 38
3.3 累算代入演算子 41
3.4 単項演算子 42
3.5 実用例：f-文字列をシミュレート（続） 45
3.6 実用例：日付 46
♣ 日付を進める 46
♣ 「n 日間」を生成する 47
3.7 実用例：HTTP クライアント 48
3.8 実用例：構文木の構築 51
3.9 実用例：アサーション 54
♣ assert 文について 54

♣ Step 1. ヘルパー関数とヘルパークラスを用意 55
♣ Step 2. 「==」以外の演算子にも対応する 57
♣ Step 3. 上書きできない演算子にも対応する 57
♣ Step 4. 演算子に関係なく便利なメソッドを追加する 58
3.10 実用例：SQL ビルダー 61
♣ Step 1. Python の式を SQL 文字列に変換 61
♣ Step 2. 「`== None`」を「`is null`」に変換 62
♣ Step 3. 式クラスとビルダークラスを導入 64
♣ Step 4. select 文を生成 66
3.11 実用例：経路定義用 DSL 68
♣ Step 1. `Route` クラスと `Point` クラスを用意する 68
♣ Step 2. 「`>`」演算子を上書きする 69
♣ Step 3. 単項演算子の「`+`」を上書きする 72
3.12 実用例：図形のような DSL 74
♣ 矢印の長さに意味を持たせる 74
♣ Step 1. ランナーと単項演算子を定義する 75
♣ Step 2. レースと「`<`」演算子を実装する 76
♣ Step 3. レース終了時に相互参照を解消する 79

第 4 章 デコレータ 81
4.1 関数定義と高階関数について 82
4.2 関数デコレータとは 83
4.3 引数や戻り値に対応する 85
4.4 関数デコレータに引数を与える 87
4.5 引数を省略できる関数デコレータ 89
4.6 関数の名前と Docstring をコピーする 91
4.7 JavaScript の即時関数をまねる 94
4.8 クラスデコレータ 98
4.9 実用例：実行時間の計測 101
4.10 実用例：メモ化 102
4.11 実用例：タスクランナー 103
♣ 指定されたタスクを実行する 103
♣ タスク名を指定可能にする 105
♣ 共通する前処理は 1 度しか実行しない 106
4.12 実用例：Django 108
♣ Django のサンプルプログラム 108
♣ 応答可能なリクエストメソッドを限定する 110
♣ 1 つの URL パスで複数のリクエストメソッドに応対する 112

第 5 章 関数とコードオブジェクト 115
5.1 関数オブジェクト 116
5.2 コードオブジェクト 118
5.3 関数の引数を調べる 120
5.4 引数のデフォルト値を調べる 122
5.5 関数の種類を調べる 123
5.6 関数名とファイル名と開始行番号 125
5.7 クロージャ 126
5.8 Python プログラムをコンパイルする 130
♣ 複数文をコンパイルする 130
♣ 単一文をコンパイルする 131
♣ 式をコンパイルする 131
♣ グローバル変数とローカル変数を指定する 131
♣ 注意点 133
5.9 実用例：ユニットテストでのテスト順 134
♣ メソッドの一覧を定義順に並べる 134
♣ テストメソッドを定義順に実行する 136
5.10 実用例：フィクスチャ用 DI 137
5.11 実用例：テンプレートエンジン 141
5.12 実用例：再帰関数を高速化 145

第 6 章 抽象構文木 151
6.1 プログラムを構文解析する 152
6.2 抽象構文木をたどる 153
6.3 抽象構文木をコンパイルする 156
6.4 実用例：assert 文の書き換え 156

第 7 章 オブジェクトとメソッド 161
7.1 オブジェクトの作成 162
7.2 `__dict__`属性 164
7.3 `__slots__`変数 166
7.4 `getattr()` と `setattr()` 168
7.5 `__class__`属性 168
7.6 クラスオブジェクト 170
7.7 クラス変数 171
7.8 クラス継承 172
7.9 属性へのアクセス 174
7.10 インスタンスメソッド 176

7.11 メソッド呼び出し . . . 177
7.12 __class__特殊変数 . . . 181
7.13 super() . . . 183
7.14 実用例：特異メソッド . . . 186
♣ 属性に関数を設定する . . . 186
♣ 属性にクロージャを設定する . . . 187
♣ メソッドオブジェクトを作成する . . . 187
♣ 所属するクラスを変更する . . . 189
7.15 実用例：ミックスイン . . . 190
♣ ミックスインについて . . . 190
♣ ミックスインが便利なケース . . . 191
♣ 所属クラスを子クラスに切り替える . . . 191
♣ 親クラスを継承しない . . . 193
♣ 切り替えた所属クラスを元に戻す . . . 194

第 8 章 ディスクリプタ 197
8.1 ディスクリプタとは . . . 198
8.2 __get__() メソッド . . . 199
8.3 __set__() メソッド . . . 200
8.4 __del__() メソッド . . . 201
8.5 ディスクリプタの種類 . . . 202
8.6 プロパティ . . . 204
8.7 関数とインスタンスメソッド . . . 205
8.8 クラスメソッド . . . 206
8.9 static メソッド . . . 208
8.10 実用例：部分適用 . . . 210
♣ クロージャを使った部分適用 . . . 211
♣ ディスクリプタ用の__get__() を使った部分適用 . . . 211
8.11 実用例：プロパティ値をキャッシュ . . . 212
♣ getter を定義 . . . 212
♣ プロパティの値をキャッシュして高速化 . . . 213
♣ 自動的にキャッシュするプロパティクラス . . . 213
♣ よりシンプルで無駄のないキャッシュ . . . 215

第 9 章 メタクラス 217
9.1 メタクラスとは . . . 218
9.2 クラスオブジェクトの生成 . . . 219
9.3 メタクラスを定義する . . . 221
9.4 __new__() メソッド . . . 223

9.5 本物のクラスメソッド . . . 225
9.6 クラスデコレータとの比較 . . . 227
9.7 __init_subclass__() メソッド . . . 229
9.8 実用例：定義中のクラスを参照する . . . 230
9.9 実用例：クラスオブジェクトに演算子を定義 . . . 231
♣「@classmethod」を使う . . . 231
♣ メタクラスを使う . . . 232
♣ 同じ名前のインスタンスメソッドとクラスメソッド . . . 233
9.10 実用例：O/R マッパー . . . 234
♣ クラス名をもとにテーブル名を自動設定 . . . 234
♣ 変数名をもとにカラムオブジェクトに名前を自動設定 . . . 236

第 10 章 for 文と with 文 239

10.1 for 文とは . . . 240
10.2 __iter__() と__next__() . . . 240
10.3 for 文の仕組み . . . 241
10.4 ジェネレータ . . . 242
♣ ジェネレータの動作 . . . 242
♣ yield 文の引数 . . . 244
♣ ジェネレータのサンプル . . . 245
10.5 ジェネレータのより高度な機能 . . . 246
♣ send() . . . 246
♣ throw() . . . 247
♣ yield from . . . 249
10.6 ジェネレータを for 文で使う . . . 249
10.7 with 文とは . . . 250
10.8 __enter__() と__exit__() . . . 251
10.9 with 文の仕組み . . . 252
10.10 ジェネレータ関数を with 文で使う . . . 253
10.11 ジェネレータ関数を with 文で使う利点 . . . 255
10.12 @contextmanager の仕組み . . . 258
10.13 for 文で with 文をまねる . . . 259
10.14 for 文で if 文をまねる . . . 261
10.15 実用例：キャッシュ用 DSL . . . 262
♣ if 文を使う . . . 262
♣ キャッシュ用のクラスを作る . . . 263
♣ ブロック引数を使う (Ruby) . . . 264
♣ for 文を使う . . . 265
10.16 実用例：ベンチマークツール . . . 267

♣ 実行時間を計測する 267
♣ ランキングを表示する 268
♣ ランキングを自動的に表示する 269
♣ 変数のスコープを制限する 271

第 11 章 例外とトレースバック 273

11.1 例外オブジェクト 274
11.2 トレースバック 275
11.3 例外の表示を変更する 276
11.4 トレースバックを変更する 278
11.5 SyntaxError クラス 281
11.6 実用例：例外ハンドラをカラー化 281
♣ 例外ハンドラを自作する 282
♣ 連鎖して発生した例外に対応する 283
♣ 表示をカラー化する 284
11.7 実用例：JSON パーサのエラー表示 286
♣ JSON のパースエラー 286
♣ 例外のトレースバックを間引く 287
♣ JSON ファイルのエラー箇所を表示する 288

第 12 章 グローバル変数とローカル変数 291

12.1 グローバル変数の読み書き 292
12.2 グローバル変数の一覧 293
12.3 真のグローバル変数 294
12.4 外側の関数のローカル変数 296
12.5 ローカル変数の一覧 297
12.6 呼び出し元のローカル変数 298
12.7 実用例：例外のトレースバックを間引く 299

第 13 章 モジュール 303

13.1 モジュールオブジェクトを生成 304
13.2 モジュール辞書 305
13.3 モジュールオブジェクトを登録 306
13.4 モジュール名 307
13.5 モジュールファイルを実行したときの落とし穴 308
13.6 実用例：1 ファイルで複数のモジュール 310
♣ 1 ファイルで複数モジュールにする背景 310
♣ モジュールを分割する 311

参考文献 313

第1章

スタックフレーム

　スタックフレームとは、関数を呼び出すごとに用意されるメモリ上のデータのことです。スタックフレームをたどることで、関数の呼び出し元の情報にアクセスできます。このような機能は主にデバッガで使われていますが、Python ではデバッガでなくてもスタックフレームにアクセスできます。

　この章では、スタックフレームをたどることでできる黒魔術を紹介します。

1.1　スタックフレームとは 2
1.2　呼び出し元のスタックフレームにアクセスする 6
1.3　呼び出し元の情報を取得する 7
1.4　呼び出し元のローカル変数にアクセスする 8
1.5　`inspect` モジュール 11
1.6　実用例：f-文字列をシミュレート 12
1.7　実用例：現在行のファイル名と行番号 13
1.8　実用例：`getter` と `setter` 14

1.1 スタックフレームとは

スタックフレーム (Stack frame) とは、関数を呼び出すごとに用意されるメモリ上のデータのことです。主には関数のローカル変数が格納されます（引数もローカル変数の一種です）。

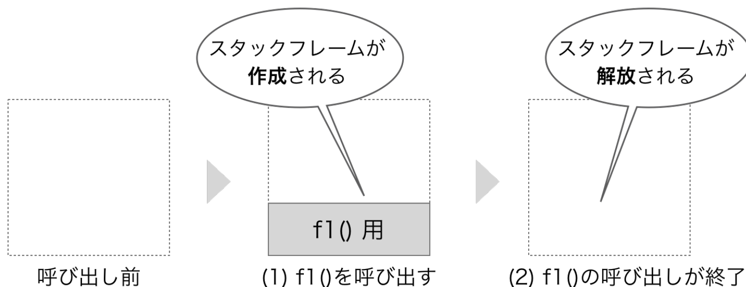
スタックフレームは、関数を呼び出すと作成され、呼び出しが終了すると解放されます。たとえば次のサンプルプログラムを見てください。

▼関数 f1() を呼び出す

```
def f1(x):
    y = x + 1
    print(y)

f1()
```

これを実行すると、内部では次のような動作になります（図 1.1）。

(1) 関数 f1() を呼び出すと、f1() 用のスタックフレームが**作成**される。
(2) 関数 f1() の呼び出しが終了すると、そのスタックフレームが**解放**される。



▲図 1.1　スタックフレームの作成と解放

関数呼び出しが終了するとローカル変数が使えなくなる理由

すでに説明したように、スタックフレームには主にローカル変数が格納されます（引数もローカル変数の一種です）。そのため関数呼び出しが終了してスタックフレームが解放されると、ローカル変数も使えなくなります。

そして再び関数が呼び出されると、新しいスタックフレームが作成されてローカル変数が使えるようになります。

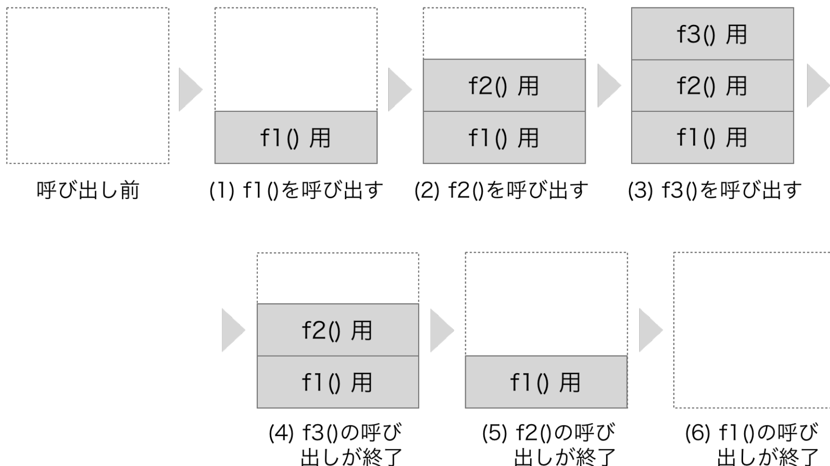
関数呼び出しが入れ子になっている場合（ある関数から別の関数を呼び出した場合）も、同じように動作します。たとえば次のサンプルプログラムを見てください。

▼関数 f1() が f2() を、f2() が f3() を呼び出す

```
def f1():
    x = 10
    f2(x)        ← f1()がf2()を呼び出す

def f2(y):
    f3(y+1)      ← f2()がf3()を呼び出す

def f3(z):
    print(f"z={z}")

f1()             ← f1()を呼び出す
```

これを実行すると、内部では次のような動作をします（図 1.2）。

(1) f1() を呼び出すと、f1() 用のスタックフレームが作成される。
(2) f1() が f2() を呼び出すと、f2() 用のスタックフレームが作成される。
(3) f2() が f3() を呼び出すと、f3() 用のスタックフレームが作成される。
(4) f3() の呼び出しが終了すると、f3() 用のスタックフレームが解放される。
(5) f2() の呼び出しが終了すると、f2() 用のスタックフレームが解放される。
(6) f1() の呼び出しが終了すると、f1() 用のスタックフレームが解放される。



▲図 1.2　スタックフレームは作成されたのとは逆順に解放される

このように、スタックフレームは作成されたのとは逆の順番で解放されます。これは関数呼び出しの動作と同じです（あとから呼んだ関数のほうが先に終了する）。

スタックフレームの大きさ

スタックフレームの大きさ（メモリサイズ）は、関数ごとに異なります。図 1.2 では f1() と f2() と f3() のスタックフレームがどれも同じ大きさで描かれていますが、通常は関数が違えばスタックフレームの大きさも異なります。

関数を再帰呼び出ししても動作は同じです。つまり作成されたのとは逆順にスタックフレームが解放されます。

たとえば次のサンプルプログラムを見てください。

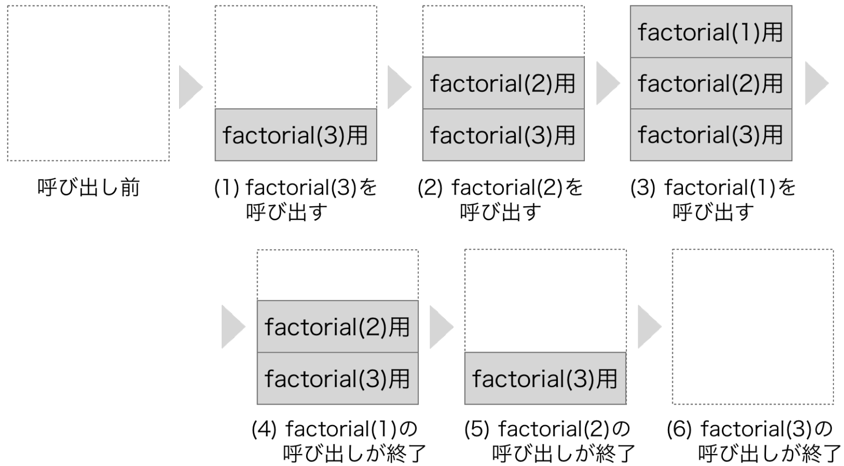
▼ factorial()：階乗を計算する

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)

val = factorial(3)     ← 3 * 2 * 1 を計算する
print(val)             ⇒ 6
```

これを実行すると、内部では次のような動作をします（図 1.3）。

(1) factorial(3) を呼び出すと、新しいスタックフレームが作成される。
(2) factorial(2) を呼び出すと、新しいスタックフレームが作成される。
(3) factorial(1) を呼び出すと、新しいスタックフレームが作成される。
(4) factorial(1) の呼び出しが終了すると、(3) で作成したスタックフレームが解放される。
(5) factorial(2) の呼び出しが終了すると、(2) で作成したスタックフレームが解放される。
(6) factorial(3) の呼び出しが終了すると、(1) で作成したスタックフレームが解放される。

このように、再帰関数でも動作は同じであることが分かります。



▲図 1.3　関数の再帰呼び出しでも動作は同じ

再帰呼び出しでもローカル変数は上書きされない

スタックフレームのおかげで、ローカル変数は関数呼び出しごとに別のメモリ領域に格納されます。そのため、再帰呼び出しをしても同じ名前のローカル変数が上書きされることはありません。これは初級者が勘違いしやすい点なので注意してください。

たとえば「`factorial(3)`」を呼び出すと、引数「`n`」の値は 3 です。その呼び出し中に「`factorial(2)`」を呼び出すと引数「`n`」の値が 2 になるので、上書きされたように見えるかもしれません。

しかし「`factorial(3)`」の呼び出しと「`factorial(2)`」の呼び出しではスタックフレームが別なので、それぞれの引数「`n`」はメモリ領域が別です。そのおかげで、引数「`n`」の値は上書きされないのです。

実行フレーム

Python のマニュアル[*1]では、スタックフレームのことを「実行フレーム」(Execution frame) と呼んでいます。しかしそう呼んでいるのが Python のマニュアルぐらいしかなく、他の資料や他のプログラミング言語だと「スタックフレーム」と呼んでいるため、本書でもそう呼びます。

[*1] https://docs.python.org/ja/3/reference/executionmodel.html

1.2 呼び出し元のスタックフレームにアクセスする

Python では、「sys._getframe()」を使うと関数呼び出し元のスタックフレームにアクセスできます。

- 「sys._getframe(0)」で、現在のスタックフレーム
- 「sys._getframe(1)」で、呼び出し元のスタックフレーム
- 「sys._getframe(2)」で、呼び出し元の呼び出し元のスタックフレーム
- 「sys._getframe(3)」で、呼び出し元の呼び出し元の呼び出し元のスタックフレーム
- …

次のサンプルプログラムを見てください。f1() の中から f2() を呼び出しているので、f2() の呼び出し元は f1() です。そして f2() の中から、呼び出し元である f1() のスタックフレームにアクセスしています。

▼ファイル「ex-stackframe1.py」

```
import sys

def f1():                  ← f1()の中から
    f2()                   ← f2()を呼び出す（つまりf1()が呼び出し元）

def f2():
    frame0 = sys._getframe(0)  ← f2()のスタックフレーム
    frame1 = sys._getframe(1)  ← f1()のスタックフレーム
    frame2 = sys._getframe(2)  ← f1()の呼び出し元のスタックフレーム
    print(frame0)          ← 現在、つまりf2()のスタックフレーム
    print(frame1)          ← 呼び出し元であるf1()のスタックフレーム
    print(frame2)          ← 呼び出し元の呼び出し元のスタックフレーム
    del frame0, frame1, frame2  ← スタックフレームへの参照を消す

f1()
```

▼実行結果：右端の関数名に注目

```
$ python3 ex-stackframe1.py
<frame at 0x1100565e8, file 'ex-stackframe1.py', line 10, code f2>
<frame at 0x7f9b53508668, file 'ex-stackframe1.py', line 4, code f1>
<frame at 0x10fffa9f8, file 'ex-stackframe1.py', line 15, code <modu
le>>
```

なおこのサンプルプログラムの場合、「sys._getframe(2)」までしかスタックフレームをたどれないので、「sys._getframe(3)」を実行すると「ValueError: call

stack is not deep enough」というエラーが発生します。

スタックフレームへの参照を消す

すでに説明したように、スタックフレームにはローカル変数のデータが含まれています。そのため、ローカル変数がスタックフレームを参照すると、循環参照になります[*2]。

- スタックフレームはローカル変数（の辞書）を参照し、
- ローカル変数がスタックフレームを参照するので、循環参照になる。

Python では循環参照が発生してもガーベージコレクション (Garbage collection, GC) が回収してくれますが、GC に余計な負荷がかかります。そのため、「del frame0」や「frame0 = None」のようにスタックフレームへの参照を明示的に消すのがいいでしょう。

また例外が発生してもスタックフレームへの参照を確実に消すには、try-finally 構文を使ってください。

```
frame = sys._getframe(1)
try:
    ... frameを使った作業 ...
finally:
    del frame
```

1.3 呼び出し元の情報を取得する

呼び出し元のスタックフレームにアクセスできると、呼び出し元の次のような情報を取得できます。

```
frame.f_back              ← 呼び出し元フレーム
frame.f_globals           ← グローバル変数の一覧
frame.f_locals            ← ローカル変数の一覧
frame.f_lineno            ← 行番号
frame.f_code              ← コードオブジェクト
frame.f_code.co_filename  ← ファイル名
```

たとえば次の例では、呼び出し元を次々とたどってファイル名と行番号を表示しています。

[*2] 参考：https://docs.python.org/ja/3/library/inspect.html#the-interpreter-stack

▼ファイル「ex-stackframe2.py」

```
import sys, os

def f1():
    f2()

def f2():
    f3()

def f3():
    frame = sys._getframe(0)     ← 現在のスタックフレーム
    while frame is not None:
        file = os.path.basename(frame.f_code.co_filename)
        line = frame.f_lineno
        print(f"{file}:{line}") ← ファイル名と行番号を表示
        frame = frame.f_back     ← 読み出し元をたどる

f1()
```

▼実行例

```
$ python3 ex-stackframe2.py
ex-stackframe2.py:13
ex-stackframe2.py:7
ex-stackframe2.py:4
ex-stackframe2.py:17
```

1.4 呼び出し元のローカル変数にアクセスする

呼び出し元のスタックフレームにアクセスできると、呼び出し元のローカル変数にもアクセスできます。

次の例を見てください。f1() から f2() を呼び出しているので、f2() の呼び出し元は f1() です。そして f2() の中から f1() のスタックフレーム経由で f1() のローカル変数を取り出しています。

▼ファイル「ex-stackframe3.py」

```
import sys

def f1(a, b):
    t = a + b
    f2(t)
```

```
def f2(x):
    frame = sys._getframe(1)    ← f1()のスタックフレーム
    print(frame.f_locals)       ← f1()のローカル変数
    print(type(frame.f_locals))   ⇒ <class 'dict'>
    del frame

f1(3, 4)
```

▼実行例

```
{'a': 3, 'b': 4, 't': 7}     ← f1()のすべてのローカル変数
<class 'dict'>               ← frame.f_localsは辞書
```

この結果を見ると、呼び出し元である `f1()` のローカル変数を辞書オブジェクトで取り出していますね。そしてこの辞書を書き換えたら呼び出し元のローカル変数も変更できそうですが、そうはなりませんでした。

▼ファイル「ex-stackframe4.py」

```
import sys

def f1():
    x = 1
    a = [0]
    print(f"before: x={x}, a={a}")   ⇒ x=1, a=[0]
    f2()
    print(f"after:  x={x}, a={a}")   ⇒ x=1, a=[9]

def f2():
    frame = sys._getframe(1)      ← f1()のスタックフレーム
    frame.f_locals['x'] = 9       ← f1()のローカル変数 x を変更
    frame.f_locals['a'][0] = 9    ← f1()のローカル変数 a（リスト）
                                    の内容を変更
    del frame

f1()
```

▼実行例

```
$ python3 ex-stackframe4.py
before: x=1, a=[0]
after:  x=1, a=[9]   ← xは変更されず、aの中身だけ変更
```

この実行結果を見ると、`frame.f_locals` の辞書を変更しても `f1()` のローカル変数 x が変更されていないことが分かります。

このように、スタックフレームを経由して呼び出し元のローカル変数にアクセスできます。また呼び出し元のローカル変数そのものは変更できませんが、ローカル変数が参照しているオブジェクトの内容は変更できます。

変数が参照しているオブジェクトを変更する

「変数の値を変更する」ことと、「変数が参照しているオブジェクトを変更する」ことは違います。先ほどのサンプルプログラムでいうと、辞書を変更してもローカル変数 x や a の値は変更されませんが、a が参照してるリストオブジェクトの中身は変更できてしまいます。

詳しくは『オブジェクト指向言語解体新書[*3]』の第１章を参照してください。

クラス定義では呼び出し元のローカル変数を変更できる

さきほど「呼び出し元のローカル変数そのものは変更できません」と説明しましたが、実はクラス定義の中ではこれができてしまいます。

▼ファイル「ex-stackframe5.py」

```
import sys

def f3():
    frame = sys._getframe(1)
    frame.f_locals['x'] = 20
    del frame

class Dummy:
    x = 10
    print(f"x={x}")     ⇒ x=10
    f3()
    print(f"x={x}")     ⇒ x=20
```

▼実行例

```
$ python3 ex-stackframe5.py
x=10
x=20        ← ローカル変数の値が変更されている！
```

このような仕様になっている理由（つまりクラス定義の中でだけ変更可能である理由）は不明です[*4]。ご存知の方がいたら教えていただければと思います。

*3 https://kauplan.org/books/oopl/
*4 関連：第 12.5 節「ローカル変数の一覧」(p.297)

1.5 inspect モジュール

Python の標準ライブラリ「inspect」には、スタックフレームへアクセスする関数が定義されています。これらの関数は内部で sys._getframe() を呼び出しています。

たとえば現在のスタックフレームを返す「inspect.currentframe()」は次のように定義されています。sys._getframe() を呼び出していることが分かりますね。

▼ inspect.currentframe() の定義

```
def currentframe():
    """Return the frame of the caller or None if this is not possibl›
›e."""
    return sys._getframe(1) if hasattr(sys, "_getframe") else None
```

inspect モジュールには次のような関数があります。currentframe() 以外の関数は、スタックフレームオブジェクトを Traceback オブジェクト[*5]に変換するので、呼び出し箇所のソースコードを取得するときに便利です。詳しくはマニュアル[*6]を参照してください。

inspect.currentframe()
: 現在のスタックフレームを返す。sys._getframe(0) と同じ。

inspect.getframeinfo(frame, context=1)
: スタックフレームを渡すと、Traceback オブジェクトに変換して返す。第 2 引数はソースコードの行を前後何行取り出すかを示す。

inspect.getouterframes(frame, context=1)
: 引数に渡したスタックフレームの呼び出し元をたどり、Traceback オブジェクトのリストにして返す。

inspect.getinnerframes(frame, context=1)
: 引数に渡したスタックフレームから呼び出されたスタックフレームを、Traceback オブジェクトのリストにして返す。

inspect.stack(context=1)
: 現在のスタックフレームの呼び出し元をたどり、Traceback オブジェクトのリストにして返す。

inspect.trace(context=1)
: 例外が発生した箇所のスタックフレームから呼び出されたスタックフレームを、Traceback オブジェクトのリストにして返す。

[*5] 例外発生時に表示されるデータ。詳しくは第 11.2 節「トレースバック」(p.275) を参照してください。
[*6] https://docs.python.org/ja/3/library/inspect.html

1.6 実用例：f-文字列をシミュレート

呼び出し元のローカル変数やグローバル変数にアクセスできることを利用して、f-文字列をシミュレートしてみましょう。

Python の「f-文字列」(f-strings) とは、文字列の中に任意の式を埋め込める文字列リテラルのことです。

▼ Python：f-文字列の例

```
w = 2560
h = 1600
print(f"width={w}, height={h}")   ← 先頭に f がついていることに注意
```

この f-文字列は大変便利な機能ですが、残念ながら Python 3.6 からでしか使えません。

そこで、f-文字列と似たようなことを行う関数を作成してみましょう。

▼ ファイル「ex-fstrings1.py」

```
import sys, re

def f(string):
    return _rexp.sub(_replace, string) ← 第1引数に関数を指定

_rexp = re.compile(r'\{(\w+)\}')

def _replace(m):       ← 引数の「m」は正規表現のマッチデータ
    key = m.group(1)
    frame = sys._getframe(2)      ← f()の呼び出し元のスタックフレーム
    try:
        if key in frame.f_locals:    ← ローカル変数を検索し、
            return str(frame.f_locals[key])  ← 置換する。
        if key in frame.f_globals:  ← グローバル変数を検索し、
            return str(frame.f_globals[key]) ← 置換する。
        return "{%s}" % key         ← なければ置換しない。
    finally:
        del frame

## 使い方
w = 2560
h = 1600
print(f("width={w}, height={h}"))
```

▼ 実行結果

```
$ python3 ex-fstrings1.py
width=2560, height=1600
```

Python の f-文字列は変数だけでなく任意の式が埋め込めますが、ここで定義した f() はローカル変数またはグローバル変数しか使えないので注意してください。

なお第 3 章 (p.35) で説明する演算子の上書き機能を使うと、見た目が f-文字列に近づきます。詳しくは第 3.5 節「実用例：f-文字列をシミュレート（続）」(p.45) を参照してください。

1.7 実用例：現在行のファイル名と行番号

プログラムコードにおいて現在行のファイル名と行番号を取得するには、スタックフレームの情報を使います。

▼ ファイル「ex-location.py」

```
import sys

def location():
    """現在行のファイル名と行番号を返す"""
    frame = sys._getframe(1)
    filename = frame.f_code.co_filename    ← ファイル名
    lineno   = frame.f_lineno              ← 行番号
    del frame
    return filename, lineno

## 使い方
filename, lineno = location()
print(f"filename={filename}, lineno={lineno}")
```

▼ 実行例

```
$ python3 ex-location.py
filename=ex-location.py, lineno=11
```

あるいは、現在のファイル名と行番号つきでメッセージを表示するようなデバッグ用関数は、次のようにすれば実現できます。

▼ ファイル「ex-debugprint.py」

```
import sys
```

```
def debug(message):
    """ファイル名と行番号つきでメッセージをstderrに表示"""
    frame = sys._getframe(1)
    filename = frame.f_code.co_filename   ← ファイル名
    lineno   = frame.f_lineno             ← 行番号
    del frame
    sys.stderr.write(f"** [DEBUG] {filename}:{lineno}: {message}\n")

## 使い方
debug("something wrong")
```

▼実行例

```
$ python3 ex-debugprint.py
** [DEBUG] ex-debugprint.py:11: something wrong
```

1.8 実用例：getter と setter

Python ではオブジェクトの getter や setter[*7]を定義するとき、関数デコレータ「@property」を使って次のようにします（詳細は省略します）。

▼ファイル「ex-getter1.py」

```
class Movie(object):

    def __init__(self, title):
        self._title = title

    @property          ← タイトルのgetterを定義
    def title(self):
        return self._title

    @title.setter      ← タイトルのsetterを定義
    def title(self, newtitle):
        self._title = newtitle

katasumi = Movie(u"この世界の片隅に")
print(katasumi.title)     ⇒ この世界の片隅に
```

[*7] getter とは、オブジェクトの属性にアクセスしてその値を返すメソッドのこと。setter とは、オブジェクトの属性に値を設定するメソッドのこと。どちらもオブジェクト指向言語で一般的な用語であり、Python 独自の用語ではありません。

```
katasumi.title = u"この世界の（さらにいくつもの）片隅に"
print(katasumi.title)    ⇒ この世界の（さらにいくつもの）片隅に
```

▼実行例

```
$ python3 ex-getter1.py
この世界の片隅に
この世界の（さらにいくつもの）片隅に
```

ポイントは次の点です。

- getter は「`@property`」を使う。
- setter は「`@`*propertyname*`.setter`」を使う。

これを見ると、getter と setter の定義が非対称的ですね。もっと分かりやすく、対照的にはできないでしょうか。

実は呼び出し元のローカル変数にアクセスできると、改善できます。次のサンプルプログラムを見てください。

▼ファイル「ex-getter2.py」

```
import sys

def getter(func):
    return property(func)          ← 「@property」と実質同じ

def setter(func):
    name = func.__name__           ← 関数名
    vars = sys._getframe(1).f_locals ← 呼び出し元（クラス定義）のローカル変数の一覧（辞書）
    prop = vars[name]              ← getterで定義したプロパティ
    return prop.setter(func)       ← 「@prop.setter」と実質同じ

## 使用例

class Movie(object):

    def __init__(self, title):
        self._title = title

    @property
    @getter             ← タイトルのgetterを定義
    def title(self):
        return self._title

    @title.setter
```

```
    @setter             ← タイトルのsetterを定義
    def title(self, newtitle):
        self._title = newtitle

katasumi = Movie(u"この世界の片隅に")
print(katasumi.title)    ⇒ この世界の片隅に
katasumi.title = u"この世界の（さらにいくつもの）片隅に"
print(katasumi.title)    ⇒ この世界の（さらにいくつもの）片隅に
```

▼実行例

```
$ python3 ex-getter1.py
この世界の片隅に
この世界の（さらにいくつもの）片隅に
```

ポイントは次の点です。

- 「@property」のかわりに「@getter」を使う。
- 「@title.setter」のかわりに「@setter」を使う。

定義方法がとても対象的になり、直感的で分かりやすくなりました。
ただしこのままだといくつか問題があります。

- getter がないと setter を定義できない。setter ではあらかじめ getter で property 化されていることを仮定しているため、定義する順番に依存性がある。
- もし setter が先に定義できたとして、あとから getter を定義すると setter が消えてしまう（上書きされてしまうため）。

これらの点を改善し、より実用的に使えるようにしてみましょう。

▼ファイル「ex-getter3.py」

```
import sys

def getter(func):
    name = func.__name__                ← 関数名
    vars = sys._getframe(1).f_locals ← 呼び出し元（クラス定義）のローカ
ル変数の一覧（辞書）
    prop = vars.get(name, None)     ← 定義済みのプロパティ
    if prop is None:                ← 定義済みのプロパティがなければ
        return property(func)       ←     「@property」と実質同じ
    return prop.getter(func)        ← 「@prop.getter」と実質同じ

def setter(func):
    name = func.__name__                ← 関数名
    vars = sys._getframe(1).f_locals ← 呼び出し元（クラス定義）のローカ
ル変数の一覧（辞書）
```

```
    prop = vars.get(name, None)    ← 定義済みのプロパティ
    if prop is None:               ← 定義済みのプロパティがなければ
        return property(None, func)  ← setterだけを持つプロパティを生成
    return prop.setter(func)       ← 「@prop.setter」と実質同じ

## 使用例（getterよりsetterを先に定義）

class Movie(object):

    def __init__(self, title):
        self._title = title

    @setter            ← setterを先に定義してみる
    def title(self, newtitle):
        self._title = newtitle

    @getter            ← getterを後から定義してみる
    def title(self):
        return self._title

katasumi = Movie(u"この世界の片隅に")
print(katasumi.title)    ⇒ この世界の片隅に
katasumi.title = u"この世界の（さらにいくつもの）片隅に"
print(katasumi.title)    ⇒ この世界の（さらにいくつもの）片隅に
```

これで問題点は解決されましたが、コードを見ると重複した部分が多いですね。重複をなくすようリファクタリングし、ついでに deleter の定義も加えてみましょう。

▼ファイル「ex-getter4.py」

```
import sys

def _getprop(func):
    name = func.__name__               ← 関数名
    vars = sys._getframe(2).f_locals ← 呼び出し元（クラス定義）のローカル変数の一覧（辞書）
    prop = vars.get(name, None)    ← 定義済みのプロパティ
    return prop                    ← 定義済みのプロパティを返す

def getter(func):
    prop = _getprop(func)          ← 定義済みのプロパティを取得
    if prop is None:               ← 定義済みのプロパティがなければ
        return property(func)      ←     「@property」と実質同じ
    return prop.getter(func)       ← 「@prop.getter」と実質同じ
```

```
def setter(func):
    prop = _getprop(func)          ← 定義済みのプロパティを取得
    if prop is None:               ← 定義済みのプロパティがなければ
        return property(None, func)  ← setterだけを持つプロパティを生成
    return prop.setter(func)       ← 「@prop.setter」と実質同じ

def deleter(func):
    prop = _getprop(func)          ← 定義済みのプロパティを取得
    if prop is None:               ← 定義済みのプロパティがなければ
        return property(None, None, func)  ← deleterだけを持つプロパティを生成
    return prop.setter(func)       ← 「@prop.deleter」と実質同じ

## 使用例
#....(省略)....
```

これで完成です。

スタックフレームへのアクセスは遅い？

Python ではスタックフレームにアクセスできますが、必ずしも高速にアクセスできるとは限りません。CPython ではそんなに気にするほどではありませんが、JIT コンパイラを搭載していることで有名な PyPy[*8]という処理系だと非常に遅くなります。どうやら、スタックフレームへのアクセスには JIT コンパイラによる高速化が使えないからのようです。

スタックフレームへのアクセスには、多かれ少なかれ動作コストがかかることに注意してください。

[*8] https://pypy.org/

第 2 章

特殊メソッド

Python には特殊な機能を持ったメソッドが用意されています。それらを上書きすると、たとえば「`obj.attr`」や「`obj[key]`」の動作を変更できます。

この章では、特殊な機能を持ったメソッドをいくつか紹介します。

2.1 添字アクセス 20
2.2 属性アクセス 23
2.3 より強力な属性アクセス 25
2.4 関数呼び出し 27
2.5 その他の特殊メソッド 30
2.6 実用例：AutoDict 31
2.7 実用例：HTML 要素 32

2.1 添字アクセス

Python では、添字でのアクセスを「[]」演算子で行います。そして「[]」演算子に相当する特殊なメソッドがあります。

- 「obj.__getitem__(*key*)」は「obj[*key*]」に相当。
- 「obj.__setitem__(*key*, *value*)」は「obj[*key*]=*value*」に相当。
- 「obj.__delitem__(*key*)」は「del obj[*key*]」に相当。

これらの特殊なメソッドを使うと、オブジェクトをあたかも辞書のようにアクセスできます。

▼ ファイル「ex-special01.py」

```
class Soldier(object):

    def __init__(self, name, height):
        self.name   = name
        self.height = height

    def __getitem__(self, key):          ← obj[key] に相当
        if key == 'name':   return self.name
        if key == 'height': return self.height
        raise KeyError(key)

    def __setitem__(self, key, value):   ← obj[key] = value 相当
        if   key == 'name':   self.name   = value
        elif key == 'height': self.height = value
        else:
            raise KeyError(key)

    def __delitem__(self, key):          ← del obj[key] に相当
        if   key == 'name':   self.name   = None
        elif key == 'height': self.height = None
        else:
            raise KeyError(key)

## 使用例
eren = Soldier(u"エレン", 170)
print(eren.name)                ⇒ エレン
print(eren['name'])             ⇒ エレン
eren['name'] = u"Eren Yeager"
print(eren.name)                ⇒ Eren Yeager
print(eren['name'])             ⇒ Eren Yeager
```

第 3 章

演算子

Python では「+」や「-」や「*」や「/」といった演算子ごとに特殊メソッドが用意されており、それらを定義することで演算子の挙動を上書きできます。
この章では演算子を上書きする方法とその応用を説明します。

3.1 演算子に対応した特殊メソッド 36
3.2 演算子の左右を入れ替えた特殊メソッド 38
3.3 累算代入演算子 41
3.4 単項演算子 42
3.5 実用例：f-文字列をシミュレート（続） 45
3.6 実用例：日付 46
3.7 実用例：HTTP クライアント 48
3.8 実用例：構文木の構築 51
3.9 実用例：アサーション 54
3.10 実用例：SQL ビルダー 61
3.11 実用例：経路定義用 DSL 68
3.12 実用例：図形のような DSL 74

3.1 演算子に対応した特殊メソッド

Python には、演算子と対応した特殊メソッドが用意されています[*1]。次ページの表 3.1 に一覧を載せているので見てください[*2]。

そしてこれらの特殊メソッドを上書きすると、演算子の挙動を変更できます[*3]。

たとえば次の例では、左シフト演算子「<<」でリストに要素を追加できるよう、特殊メソッド「`__lshift__()`」を上書きしています。

▼ファイル「ex-operator01.py」

```
## Pythonでは組み込みクラス「list」を変更できないので、
## かわりに「list」を継承したクラスを作る
class MyList(list):

    def __lshift__(self, arg):     ← 「<<」に対応する特殊メソッド
        list.append(self, arg)     ← 親クラスのメソッドを呼び出す

## 使い方
list1 = MyList(['A', 'B'])
list1 << 'C'       ← list1.__lshift__('C') と同じ
list1 << 'D'       ← list1.__lshift__('D') と同じ
print(list1)      ⇒ ['A', 'B', 'C', 'D']
```

▼実行例

```
$ python3 ex-operator01.py
['A', 'B', 'C', 'D']
```

「`__lshift__()`」の戻り値をリスト自身にすると、「<<」を連続して使えます。

▼ファイル「ex-operator02.py」

```
class MyList(list):

    def __lshift__(self, arg):     ← 「<<」に対応する特殊メソッド
        list.append(self, arg)     ← 親クラスのメソッドを呼び出す
        return self                ← リスト自身を返す
```

[*1] 参考：https://docs.python.org/ja/3/reference/datamodel.html

[*2] このうち、添字指定の「x[y]」と対応した特殊メソッド「`__getitem__()`」については第 2.1 節 (p.20) を参照してください。

[*3] 演算子を上書きすることを、プログラミング用語で演算子オーバーロード (Operator overload) といいます。

第 4 章

デコレータ

「デコレータ」とは、一言でいうと「関数やクラスをあれこれいじる機能」です。デコレータを使うと、たとえば「関数に前処理や後処理を追加する」「関数の引数や戻り値を改変する」「クラス定義時にクラス変数を追加する」といったことができます。まさに黒魔術向きですね。

この章では、まず関数デコレータについて基礎から応用まで説明したあと、クラスデコレータについて説明します。

4.1 関数定義と高階関数について 82
4.2 関数デコレータとは 83
4.3 引数や戻り値に対応する 85
4.4 関数デコレータに引数を与える 87
4.5 引数を省略できる関数デコレータ 89
4.6 関数の名前と Docstring をコピーする 91
4.7 JavaScript の即時関数をまねる 94
4.8 クラスデコレータ 98
4.9 実用例：実行時間の計測 101
4.10 実用例：メモ化 102
4.11 実用例：タスクランナー 103
4.12 実用例：Django 108

4.1 関数定義と高階関数について

デコレータについて説明するまえに、関数定義と高階関数について説明します。

Python における関数定義は、実際は関数オブジェクトを変数に代入しているだけです。そのため、ある関数を別の変数に代入すると、その変数を使って関数呼び出しができます。

```
## このような関数定義は、
def sum(x, y, z):
    return x + y + z

## このような代入と同じ（細かい点を除けば）
sum = lambda x, y, z: x + y + z

## そのため、関数を別の変数に代入して呼び出すことが可能
total = sum
print(total(1, 2, 3))     ⇒ 6
```

このコードを見ると、関数オブジェクトがただのデータであることが分かるでしょう。つまり文字列や整数のようなデータを変数に代入するのと同じように、関数オブジェクトもまたデータとして変数に代入できるのです。

そして高階関数 (Higher-order function) とは、関数をデータとして扱うような関数のことです。具体的には、関数を引数として受け取るような関数や、関数を返すような関数のことです。先ほど説明したように Python では関数をデータとして扱えるので、高階関数も定義できます。

次のサンプルプログラムを見てください。関数「debug()」は引数に関数を受け取り、内部で新しい関数を作成して返しています。そのため、関数「debug()」は高階関数です。

▼ファイル「ex-deco1.py」

```
## 高階関数
def debug(func):          ← 引数に関数を受け取る
    def newfunc():        ← 新しい関数を作成
        print(f"** {func.__name__}(): enter")   ← 前処理
        func()            ← もとの関数を呼び出す
        print(f"** {func.__name__}(): exit")    ← 後処理
    return newfunc        ← 作成した新しい関数を返す

## こんな関数があったとして、
```

第 5 章

関数とコードオブジェクト

Python では、関数はオブジェクトです。関数についてのさまざまな情報は、関数オブジェクトを調べれば得られます。

また「コードオブジェクト」とは、Python のバイトコードをオブジェクト化したものです。関数のコードオブジェクトを調べれば、関数オブジェクトについてより細かい情報が得られます。

この章では関数とコードオブジェクトを使ってどのようなことができるか、紹介します。

5.1 関数オブジェクト 116
5.2 コードオブジェクト 118
5.3 関数の引数を調べる 120
5.4 引数のデフォルト値を調べる 122
5.5 関数の種類を調べる 123
5.6 関数名とファイル名と開始行番号 125
5.7 クロージャ 126
5.8 Python プログラムをコンパイルする 130
5.9 実用例：ユニットテストでのテスト順 134
5.10 実用例：フィクスチャ用 DI 137
5.11 実用例：テンプレートエンジン 141
5.12 実用例：再帰関数を高速化 145

5.1 関数オブジェクト

Python では、関数はオブジェクトです。関数オブジェクトの属性を調べると、関数についての情報が得られます。

***func*.__name__**
: 関数の名前。変更が可能。

***func*.__qualname__**
: クラス名を含めた関数の名前。

***func*.__doc__**
: 関数の Docstring。

***func*.__code__**
: 関数のコードオブジェクト（次節で説明）。ここには関数のより細かい情報が格納されている。

***func*.__defaults__**
: 位置引数のデフォルト値（のタプル）。

***func*.__kwdefaults__**
: キーワード専用引数のデフォルト値（の辞書）。

***func*.__closure__**
: 関数がクロージャの場合、参照している外側の変数（の辞書）。

***func*.__globals__**
: 関数から参照できるグローバル変数（の辞書）。

▼ファイル「ex-funcobj1.py」

```
def f1(a, b=1, c=[], *, d=None, e={}):
    """Sample function"""
    pass

print(f1.__name__)            ⇒ f1
print(f1.__qualname__)        ⇒ f1
print(f1.__doc__)             ⇒ Sample function
print(type(f1.__code__))      ⇒ <class 'code'>
print(f1.__defaults__)        ⇒ (1, [])
print(f1.__kwdefaults__)      ⇒ {'d': None, 'e': {}}
print(f1.__closure__)         ⇒ None
print(type(f1.__globals__))   ⇒ <class 'dict'>

## __name__と__qualname__の違い
```

第 6 章

抽象構文木

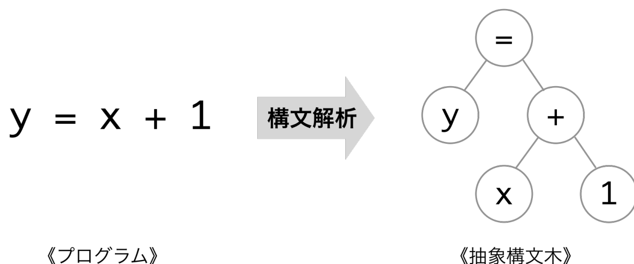
「抽象構文木」とは、プログラムを構文解析して得られる木構造のデータです。Python ではそのためのライブラリが標準で用意されています。これを使うと、プログラムを構文解析して抽象構文木にし、それを好きなように改変してから実行できます。

この章では抽象構文木を扱う方法について説明します。

6.1　プログラムを構文解析する 152
6.2　抽象構文木をたどる 153
6.3　抽象構文木をコンパイルする 156
6.4　実用例：assert 文の書き換え 156

6.1 プログラムを構文解析する

構文解析 (Parsing) とは、文法に従ってプログラムを解析することです。構文解析の結果として得られるデータを抽象構文木 (Abstract syntax tree, AST) といいます（図 6.1)。抽象構文木はその名の通り木構造をしています。



▲図 6.1　プログラムを構文解析すると抽象構文木 (AST) が得られる

Python には、Python プログラムの構文解析をするライブラリが標準で付属しています。これを使ってみましょう。

▼ファイル「ex-ast1.py」

```
import ast   ← astはabstract syntax treeの略

program = "y = x + 1"        ← Pythonプログラムを、
node = ast.parse(program)    ← 構文解析して抽象構文木を得る
print(ast.dump(node))
```

▼実行例

```
$ python3 ex-ast1.py
Module(body=[Assign(targets=[Name(id='y', ctx=Store())], value=BinOp
(left=Name(id='x', ctx=Load()), op=Add(), right=Constant(value=1, ki
nd=None)), type_comment=None)], type_ignores=[])
```

構文解析はできましたが、抽象構文木の表示にインデントがなく、とても分かりづらいですね。インデントつきで表示する方法は次の節で紹介します。

抽象構文木には、行番号やカラム番号のようなより細かい情報も含まれています。それらも表示してみましょう。

第 7 章

オブジェクトとメソッド

Python では、オブジェクトやメソッドの仕組みが隠されておらず、ユーザが自由に触ることができます。このような仕組みを公開していることは、ソフトウェア一般ではよくない（情報隠ぺいすべき）とされていますが、理解したうえで仕組みを利用するのは決して悪いことではありません。

この章では Python のオブジェクトやメソッドの仕組みを説明し、どのような黒魔術に利用できるかを紹介します。

7.1 オブジェクトの作成 162
7.2 __dict__属性 164
7.3 __slots__変数 166
7.4 getattr() と setattr() 168
7.5 __class__属性 168
7.6 クラスオブジェクト 170
7.7 クラス変数 171
7.8 クラス継承 172
7.9 属性へのアクセス 174
7.10 インスタンスメソッド 176
7.11 メソッド呼び出し 177
7.12 __class__特殊変数 181
7.13 super() 183
7.14 実用例：特異メソッド 186
7.15 実用例：ミックスイン 190

7.1 オブジェクトの作成

たとえば Hello クラスのオブジェクトを新規作成するには、Python では「Hello("world")」のようにします。このとき、内部の処理はメモリの「割り当て」(Allocation) とオブジェクトの「初期化」(Initialization) の 2 段階に分かれています[*1]。

▼ファイル「ex-new1.py」

```
## このようなクラスがあったとして、
class Hello(object):
    def __init__(self, name):
        self.name = name

## 「Hello("world")」がやっているのはだいたい次と同じ
obj = Hello.__new__()         ← オブジェクト用にメモリを割り当て
if isinstance(obj, Hello):
    Hello.__init__(obj, "world")  ← オブジェクトを初期化
```

「__new__()」メソッドを上書きすれば、既存のオブジェクトを使い回したり、子クラスのオブジェクトを返すことができます。たとえば次のサンプルプログラムでは、引数に指定された名前によって、異なる子クラスのオブジェクトを作成しています。

▼ファイル「ex-new2.py」

```
import re

class Hello(object):            ← 親クラス

    def __new__(cls, name):
        ## 名前がアルファベットか判定し、子クラスのオブジェクトを生成する
        if re.match(r'^[a-zA-Z0-9 ]+$', name):
            return super().__new__(EnglishHello)  ← 英語用
        else:
            return super().__new__(JapaneseHello) ← 日本語用

    def __init__(self, name):
        self.name = name

class EnglishHello(Hello):      ← 子クラス（英語用）
    def hello(self):
```

[*1] このような仕組みは他の言語でも見られます。たとえば Objective-C では「[[*Class* alloc] init]」、Ruby では「obj = *Class*.allocate(); obj.initialize()」がオブジェクト生成時に行われます。

第 8 章

ディスクリプタ

「ディスクリプタ」とは、属性の読み書きを制御するための機能です。他のプログラミング言語では見かけない、Python 独特の方法です。

「`@property`」による getter や setter の実装、あるいは前章で説明したインスタンスメソッドやクラスメソッドの呼び出しに、ディスクリプタが関わっています。なんだか黒魔術にも使えそうですよね？

この章では、Python の重要な機能を下から支えるディスクリプタについて説明します。

8.1	ディスクリプタとは	198
8.2	`__get__()` メソッド	199
8.3	`__set__()` メソッド	200
8.4	`__del__()` メソッド	201
8.5	ディスクリプタの種類	202
8.6	プロパティ	204
8.7	関数とインスタンスメソッド	205
8.8	クラスメソッド	206
8.9	static メソッド	208
8.10	実用例：部分適用	210
8.11	実用例：プロパティ値をキャッシュ	212

8.1 ディスクリプタとは

ディスクリプタ (Descriptor) とは、属性への読み書きを制御するための機能です。他のプログラミング言語では見かけることのない、Python 独特の方法です。

ディスクリプタの実体は、「__get__()」「__set__()」「__delete__()」のどれか（または全部）のメソッドを持っているオブジェクトのことです。特別なクラスを継承している必要はなく、3 つのメソッドのうちどれかがあれば、ディスクリプタとして扱われます。

- 属性から値を読み出すとき、「__get__()」が実行されます。
- 属性に値を設定するとき、「__set__()」が実行されます。
- del 文で属性を削除するとき、「__delete__()」が実行されます。

「__get__()」メソッドを使ったサンプルプログラムを見てみましょう。

▼ファイル「ex-descriptor1.py」

```
class Hello:
    def __init__(self, name):
        self.name = name

    ## __get__()があるので、Helloクラスのインスタンスオブジェクトは
    ## ディスクリプタとして扱われる。
    def __get__(self, *args):
        return f"Hello, {self.name}!"

class Dummy:

    ## クラス変数にHelloオブジェクトを設定
    attr1 = Hello("world")   ← ディスクリプタ

## オブジェクトのattr1属性にアクセスすると、
## Helloオブジェクトが返されるかと思いきや、
## Helloオブジェクトの__get__()が実行される。
print(Dummy().attr1)      ⇒ Hello, world!
print(Dummy.attr1)        ⇒ Hello, world!
```

▼実行例

```
$ python3 ex-descriptor1.py
Hello, world!     ← Dummy().attr1 （インスタンスオブジェクトの属性にアクセス）
Hello, world!     ← Dummy.attr1   （クラスオブジェクトの属性にアクセス）
```

第 9 章

メタクラス

「メタクラス」とは、一般的には「クラスのクラス」という意味です。Python では、クラスオブジェクトを生成するときにメタクラスが使われます。
この章では、Python におけるメタクラスの基礎と応用を説明します。

9.1 メタクラスとは 218
9.2 クラスオブジェクトの生成 219
9.3 メタクラスを定義する 221
9.4 __new__() メソッド 223
9.5 本物のクラスメソッド 225
9.6 クラスデコレータとの比較 227
9.7 __init_subclass__() メソッド 229
9.8 実用例：定義中のクラスを参照する 230
9.9 実用例：クラスオブジェクトに演算子を定義 231
9.10 実用例：O/R マッパー 234

9.1 メタクラスとは

メタクラス (Metaclass)[*1] とは、一般的には「クラスのクラス」（より正確には「クラスオブジェクトのクラス」）を意味します。

次のサンプルプログラムでは、インスタンスオブジェクトのクラスと、クラスオブジェクトのクラスを表示しています。前者は「Hello」クラスで、後者は「type」クラスです。

▼ファイル「ex-metaclass01.py」

```
class Hello(object):
    def __init__(self, name):
        self.name = name

## インスタンスオブジェクトのクラスを調べる
print(type(Hello("world")))  ⇒ <class '__main__.Hello'>

## クラスオブジェクトのクラスを調べる
print(type(Hello))           ⇒ <class 'type'>
```

▼実行例

```
$ python3 ex-metaclass01.py
<class '__main__.Hello'> ← print(type(Hello("world")))
<class 'type'>           ← print(type(Hello))
```

このように、クラスオブジェクトにもクラスが存在します。これがメタオブジェクトです。

Python では、クラスオブジェクトを作るときにメタクラスを使います。メタクラスとクラスオブジェクトの関係は、クラスとインスタンスオブジェクトの関係と同じようなものです。

- クラスからインスタンスオブジェクトが作られるように、
- メタクラスからクラスオブジェクトが作られる。

そのため、メタクラスを理解するにはクラスオブジェクトがどのように作られるかを理解する必要があります。次の節でそれを見てみましょう。

[*1] メタクラスの英語表記は「Metaclass」であって「Meta class」ではないことに注意。

第 10 章

for 文と with 文

for 文は繰り返しのための構文で、with 文は前処理と後処理を追加するための構文です。どちらも実行時に専用のメソッドを呼び出すので、そのメソッドをうまく利用すると様々な処理を行えます。

この章では for 文と with 文の仕組みを説明し、通常とは違った使い方を紹介します。

10.1 for 文とは 240
10.2 `__iter__()` と `__next__()` 240
10.3 for 文の仕組み 241
10.4 ジェネレータ 242
10.5 ジェネレータのより高度な機能 246
10.6 ジェネレータを for 文で使う 249
10.7 with 文とは 250
10.8 `__enter__()` と `__exit__()` 251
10.9 with 文の仕組み 252
10.10 ジェネレータ関数を with 文で使う 253
10.11 ジェネレータ関数を with 文で使う利点 255
10.12 `@contextmanager` の仕組み 258
10.13 for 文で with 文をまねる 259
10.14 for 文で if 文をまねる 261
10.15 実用例：キャッシュ用 DSL 262
10.16 実用例：ベンチマークツール 267

10.1 for 文とは

for 文は、リストやタプルや文字列から 1 つずつ要素を取り出してブロックを実行する構文です。

▼ファイル「ex-for1.py」

```
## たとえばリストに対してfor文を実行する
for x in [100, 200, 300]:
    print(x)

## たとえば文字列に対してfor文を実行する
for s in "ABC":
    print(s)
```

▼実行例

```
$ python3 ex-for1.py
100     ← リストに対するfor文の実行結果
200
300
A       ← 文字列に対するfor文の実行結果
B
C
```

10.2 __iter__() と __next__()

for 文では、内部的に「__iter__()」と「__next__()」が呼ばれます。

- 「__iter__()」でイテレータを作成し、
- イテレータの「__next__()」で要素を取り出す。

ここでイテレータ (Iterator object) とは、要素を順番に取り出す役目を果たすオブジェクトです。

▼ファイル「ex-for2.py」

```
items = [100, 200, 300]

iterator = items.__iter__() ← イテレータを生成
print(type(iterator))       ⇒ <class 'list_iterator'>
```

第 11 章

例外とトレースバック

この節では例外とトレースバックに関するテクニックを紹介します。

11.1 例外オブジェクト . . . 274
11.2 トレースバック . . . 275
11.3 例外の表示を変更する . . . 276
11.4 トレースバックを変更する . . . 278
11.5 SyntaxError クラス . . . 281
11.6 実用例：例外ハンドラをカラー化 . . . 281
11.7 実用例：JSON パーサのエラー表示 . . . 286

11.1 例外オブジェクト

例外オブジェクトには、次のような属性があります。

***exception*.args**
　　例外オブジェクト作成時のすべての引数が入ったタプル。

***exception*.__cause__**
　　例外が連鎖したときに、原因となった例外。

***exception*.__traceback__**
　　例外発生箇所までの関数の呼び出し関係を表すデータ（後述）。

例外の種類によっては、他の属性が追加されていることがあります。

▼ファイル「ex-exception1.py」

```
## NameErrorの場合
try:
    blabla()     ⇒ NameError: name 'blabla' is not defined
except Exception as ex1:
    print(str(ex1))       ⇒ name 'blabla' is not defined
    print(type(ex1))      ⇒ <class 'NameError'>
    print(ex1.args)       ⇒ ("name 'blabla' is not defined",)

print()

## FileNotFoundErrorの場合（args以外にも属性がある）
try:
    open("blabla.html")
except Exception as ex2:
    print(str(ex2))       ⇒ [Errno 2] No such file or directory: 'bl›
›abla.html'
    print(type(ex2))      ⇒ <class 'FileNotFoundError'>
    print(ex2.args)       ⇒ (2, 'No such file or directory')
    print(ex2.errno)      ⇒ 2
    print(ex2.filename)   ⇒ blabla.html
```

▼実行例

```
$ python3 ex-exception1.py
name 'blabla' is not defined         ← print(str(ex))
<class 'NameError'>                  ← print(type(ex1))
("name 'blabla' is not defined",)    ← print(ex1.args)
```

第12章

グローバル変数とローカル変数

この章では、グローバル変数とローカル変数に関するテクニックを紹介します。

12.1 グローバル変数の読み書き 292
12.2 グローバル変数の一覧 293
12.3 真のグローバル変数 294
12.4 外側の関数のローカル変数 296
12.5 ローカル変数の一覧 297
12.6 呼び出し元のローカル変数 298
12.7 実用例：例外のトレースバックを間引く 299

12.1 グローバル変数の読み書き

先に、グローバル変数に関するテクニックを紹介します。

Python のグローバル変数は、関数の中から読み取る（参照する）場合は何も準備はいりませんが、書き込む（変更する）場合は global 宣言が必要です。

▼ファイル「ex-global1.py」

```
x = 10           ← グローバル変数

def f1():
    return x     ← グローバル変数を参照する

def f2():
    global x     ← グローバル宣言
    x = 20       ← グローバル変数を変更する

## グローバル変数の参照
print(f1())     ⇒ 10
## グローバル変数の変更
print(x)        ⇒ 10
f2()            ← xが20に変更される
print(x)        ⇒ 20
```

▼実行例

```
$ python3 ex-global1.py
10          ← print(f1())
10          ← print(x)
20          ← print(x)
```

もし global 宣言がない場合は、ローカル変数への代入と見なされます。

▼ファイル「ex-global2.py」

```
x = 10           ← グローバル変数

def f2():
    global x     ← グローバル宣言がない場合、
    x = 20       ← これはローカル変数への代入になる

print(x)         ⇒ 10
f2()
print(x)         ⇒ 10  ← 変更されてない
```

第 13 章

モジュール

この節ではモジュールに関するテクニックを紹介します。

13.1 モジュールオブジェクトを生成 304
13.2 モジュール辞書 305
13.3 モジュールオブジェクトを登録 306
13.4 モジュール名 307
13.5 モジュールファイルを実行したときの落とし穴 308
13.6 実用例：1 ファイルで複数のモジュール 310

13.1 モジュールオブジェクトを生成

Python ではモジュールを読み込むと、モジュールオブジェクトが生成されます。

▼ファイル「ex-module1.py」

```
import sys     ← モジュールを読み込む
print(sys)                ⇒ <module 'sys' (built-in)>
print(type(sys))          ⇒ <class 'module'>
print(sys.__name__)       ⇒ sys
```

▼実行例

```
$ python3 ex-module1.py
<module 'sys' (built-in)> ← print(sys)
<class 'module'>          ← print(type(sys))
sys                       ← print(sys.__name__)
```

モジュールオブジェクトは、`import` 文を使わなくても作れます。

▼ファイル「ex-module2.py」

```
from types import ModuleType
mod = ModuleType("blabla")
print(mod)                ⇒ <module 'blabla'>
print(type(mod))          ⇒ <class 'module'>
print(mod.__name__)       ⇒ blabla
```

▼実行例

```
$ python3 ex-module2.py
<module 'blabla'>         ← print(mod)
<class 'module'>          ← print(type(mod))
blabla                    ← print(mod.__name__)
```

このやり方でもいいですが、どうせ `sys` モジュールや `re` モジュールを読み込んでいることが多いので、「`type(sys)`」や「`type(re)`」を使ってもいいです。

▼ファイル「ex-module3.py」

```
import sys
mod = type(sys)("blabla")  ← ModuleType("blabla")と同じ
print(mod)                ⇒ <module 'blabla'>
print(type(mod))          ⇒ <class 'module'>
print(mod.__name__)       ⇒ blabla
```

Magic in Python