

スツキリわかる Python

国本大悟／須藤秋良・著
株式会社 フレアリンク・監修

入門



はじめての人・つまづいた人・納得したい人のための入門書
サクサク進めてしくみもバッチリ!
プログラミングの「本質」を
身に付け未来への
道を切り拓こう!



業務自動化、データサイエンス、人工知能、
アプリ開発プログラマへの第一歩!

インプレス

スツキリわかる Python

入門

国本大悟／須藤秋良・著
株式会社 フレアリンク・監修



はじめての人・つまずいた人・納得したい人のための入門書
サクサク進めてしくみもバッチリ!
プログラミングの「本質」を
身に付け未来への
道を切り拓こう!



業務自動化、データサイエンス、人工知能、
アプリ開発プログラマへの第一歩!

インプレス

本書の内容については正確な記述につとめましたが、著者、株式会社インプレスは本書の内容に一切責任を負いかねますので、あらかじめご了承ください。

本書に掲載している会社名や製品名、サービス名は、各社の商標または登録商標です。本文中に、TM および® は明記していません。

インプレスの書籍ホームページ

書籍の新刊や正誤表など最新情報を随時更新しております。

<https://book.impress.co.jp/>

まえがき

著者の2人は、新入社員からベテランまで多くのエンジニアの学習を、研修を通じてお手伝いしています。従来、プログラミング言語研修といえばJavaが多かったのですが、近年はデータサイエンティストの需要の高まりを受け、Python研修の希望が増えています。研修後に初心者によく相談されるのが「今後ひとりで学習するのに、お勧めの入門書は？」ということです。そこで、数あるPython入門書に目を通してお勧めを探しましたが、初心者がひとりで学習するには難しすぎるか、逆に簡単すぎて今後につながらないものが多く、Pythonエンジニアを本気で目指す人の1冊目として自信を持って勧められる本を見つけられず苦悩しました。ないならば作ってしまおうと生まれたのが本書です。執筆に際しては、特に次の点を意識しています。

1. 今後に活かせる「基礎」を学べる

Pythonの利用分野は幅広く、また文法は多岐にわたるため、すべてを1冊の本にまとめることは非常に困難です。そこで、本書では初心者が利用する機会が少ない文法は思い切って割愛しました。プログラミング未経験者が基礎をしっかりと学べ、機械学習やWebアプリケーションといった専門分野の学習に進めることを目指しています。

2. 初心者でも「楽しく」学べる

Pythonの文法はシンプルでわかりやすいと言われますが、プログラミング初心者にとっては簡単ではありません。本書は、「スッキリわかる」シリーズで好評の親しみやすいイラストと柔らかい文章で仕上げています。初心者がつまずきやすい部分も、楽しくマスターできるでしょう。

3. 「ひとり」でも学べる

筆者はこれまでの研修を通じて、プログラミング言語学習の難しさは文法ではなく、トラブルシューティングにあると感じています。研修ならエラーが発生しても講師に質問して解決することができます。しかし、本での独習ではそうはいきません。そこで本書では、多くの若手エンジニアがよく起こしてしまうエラーやトラブルをできるだけ多く盛り込み、ひとりでも解決できるようにしました。

本書を通じて、読者のみなさまがPython並びにプログラミングの面白さに出会い、ひいてはエンジニアへの第一歩を踏み出すお手伝いできれば、著者としてこれ以上の喜びはありません。

著者

【謝辞】

本書の企画から発売まで多くのアドバイスとご支援をいただいた株式会社フレアリンクの中山清喬様、飯田理恵子様、インプレス編集部、イラストを担当してくださった高田様、私に教え方を教えてくれた教え子のみなさん、応援してくれた家族、その他この本に直接的、間接的に関わったすべてのみなさまに心より感謝申し上げます。

本書の見方

本書には、理解の助けとなるさまざまな用意があります。押さえるべき重要なポイントや覚えておく便利なトピックなどを要所要所に楽しいデザインで盛り込みました。読み進める際にぜひ活用してください。

本文中の色文字:

本文中、重要な用語や特に注意すべき部分に色をつけました。

1.3 データ型

1.3.1 データ型とは

これまで私たちは、数値と文字列の2種類の値を使ってきました。数値や文字列といった値の種類のことを**データ型** (data type) または単に**型**といいます。Pythonでは、この2つの種類以外に、表1-6のようなデータ型を使うことができます。

コード 6-5 RPGの勇者を表すクラスの定義と利用

```
class Hero:
    name = '松田'
    hp = 100
    def sleep(self, hours):
        print('{}は{}時間寝た!'.format(self.name, hours))
        self.hp += hours
# ゲーム開始
print('スッキリファンタジーXII ~金色の理想郷~')
h = Hero()
h.sleep(3)
print('{}のHPは現在{}'.format(h.name, h.hp))
```

データとしてnameとhp、関数としてsleepを持つオリジナル設計Heroを定義

HP100の勇者松田がオブジェクトとして誕生

予約語:

予約語 (P049) は色つきで表します。

注目コード:

注目すべきコードには、吹き出しに説明を入れて示しています。

コメント:

グレーの文字の部分は**コメント** (P026) です。

アイコン:

各アイコンの示す内容については、このページの下 (アイコンの種類) で確認してください。

リストから指定した値を削除

リスト.remove(リストから削除したい値)

※削除した要素の後ろの要素は前に詰められる。

コード 2-9 リストから要素を削除

```
members = ['工藤', '松田', '浅木']
members.remove('松田')
print(members)
```

実行結果

```
['工藤', '浅木']
```



何で僕を削除しちゃうんですか。



松田のいた位置には私が詰めておくから安心してよね。ところで、追加と削除ができるなら、変更もできるんですか？



もちろん!!

リスト内の特定の要素の内容を変更するには、添え字を指定して代入します (コード 2-10)。

吹き出し会話:

みなさんと一緒に学ぶ仲間たち (P013) が繰り広げる会話です。学びの場や開発現場でありがちな疑問点やひらめき、さらには重要なヒントが含まれていることも。ぜひ、お見逃しなく!

各章のまとめ:

その章で学んだことをまとめています。内容を正しく理解できているか確認し、達成度を測るチェック表として活用してください。

3.5 第3章のまとめ

この章では、次のことを学びました。

文と制御構造

- 1行に記述された1つの処理の実行単位であり、1つの文である。
- 文の実行順序は制御構造によってコントロールすることができ、主に順次・分岐・繰り返し (ループ) の3つがある。

条件分岐

- if文は、ある条件に当てはまる場合に処理を実行させることができる。
- if文は、条件が成立したらifブロックを、不成立だったらelseブロックを実行する。
- ブロックは、複数の文をひとまとまりとして扱う (文は1つでもよい)。
- ブロックはインデントによって定義する。

4.5 練習問題

練習 4-1
次の各コードについて、繰り返しが行われる回数を求めてください。

```
(1) count = 0
while count < 5:
    count += 1

(2) count = 1
while count <= 5:
    count += 1

(3) data = [10, 20, 30, 40, 50]
count = 0
```

各章の練習問題:

各章の章末には練習問題があり、理解度を確認できます。理解できていない場合には、その章を読み返しましょう。

アイコンの種類



構文紹介:

構文の記述ルールと文法上の留意点などを紹介します。



Column:

本書では詳細は取り上げないものの、知っておくと重宝する補足知識やトリビアなどを紹介します。



ポイント紹介:

本文における解説で、特に重要なポイントをまとめています。

CONTENTS

まえがき	003
本書の見方	004
第0章 ようこそ Python の世界へ	011
0.1 ようこそ Python の世界へ	012
0.1.1 Python を使ってできること	012
0.1.2 一緒に Python を学ぶ仲間たち	013
0.2 はじめてのプログラミング	014
0.2.1 はじめてのプログラミング	014
0.2.2 エラーと上手に付き合う	016
0.2.3 セルとノートブック	018
0.3 Python プログラミングの基礎知識	020
0.3.1 開発の流れ	020
0.3.2 統合開発環境	022
0.3.3 プログラムの書き方	024
0.3.4 プログラミング体験を終えて	026
第I部 Python の基礎を学ぼう	031
第1章 変数とデータ型	033
1.1 式と演算	034
1.1.1 数値の演算	034
1.1.2 文字列の演算	036
1.1.3 エスケープシーケンス	039
1.1.4 式と評価	040
1.2 変数	044
1.2.1 変数の利用	044
1.2.2 変数名のルール	048
1.2.3 変数の上書き	050
1.2.4 まとめて代入 (アンパック代入)	052
1.2.5 自分自身への代入	053
1.2.6 複合代入演算子	056
1.2.7 キーボード入力値の代入	057
1.3 データ型	061
1.3.1 データ型とは	061
1.3.2 データ型の変換	065
1.3.3 文字列の中に数値を埋め込む	068

CONTENTS

1.4 第1章のまとめ	073
1.5 練習問題	074
1.6 練習問題の解答	075
第2章 コレクション	077
2.1 データの集まり	078
2.1.1 変数を持つ不便さ	078
2.2 リスト	080
2.2.1 リストの特徴	080
2.2.2 リストの作成	081
2.2.3 リストの要素を参照	082
2.2.4 リスト要素の合計と要素数の取得	084
2.2.5 リスト要素の追加・削除・変更	086
2.2.6 高度な要素の指定	089
2.3 ディクショナリ	092
2.3.1 ディクショナリの特徴	092
2.3.2 ディクショナリの作成	093
2.3.3 ディクショナリ要素の参照	094
2.3.4 ディクショナリ要素の追加と変更	095
2.3.5 ディクショナリ要素の削除	096
2.3.6 ディクショナリとリストの比較	097
2.4 タプルとセット	100
2.4.1 タプル	100
2.4.2 セット	104
2.5 コレクションの応用	108
2.5.1 コレクションの相互変換	108
2.5.2 コレクションのネスト	110
2.5.3 集合演算	112
2.6 第2章のまとめ	116
2.7 練習問題	117
2.8 練習問題の解答	118
第3章 条件分岐	119
3.1 プログラムの流れ	120
3.1.1 文と制御構造	120
3.2 条件分岐の基本構造	123
3.2.1 if文	123

3.2.2	ブロックとインデント	127
3.3	条件式	131
3.3.1	比較演算子	131
3.3.2	in 演算子	132
3.3.3	真偽値	136
3.3.4	論理演算子	138
3.4	分岐構文のバリエーション	143
3.4.1	3 種類の if 文	143
3.4.2	if-else 構文	143
3.4.3	if のみの構文	144
3.4.4	if-elif 構文	147
3.4.5	if 文のネスト	150
3.5	第 3 章のまとめ	153
3.6	練習問題	154
3.7	練習問題の解答	156

第 4 章 繰り返し **159**

4.1	繰り返しの基本構造	160
4.1.1	while 文	160
4.1.2	無限ループ	164
4.1.3	状態による繰り返し	166
4.1.4	繰り返しによるリストの作成	167
4.1.5	繰り返しによるリスト要素の利用	169
4.2	for 文	171
4.2.1	for 文による繰り返し	171
4.2.2	for 文の基本構造	172
4.2.3	for 文による決まった回数の繰り返し	173
4.2.4	while 文と for 文の使い分け	175
4.3	繰り返しの制御	177
4.3.1	繰り返しの強制終了	177
4.3.2	繰り返しのスキップ	179
4.3.3	break 文と continue 文	181
4.4	第 4 章のまとめ	183
4.5	練習問題	184
4.6	練習問題の解答	187

CONTENTS

第Ⅱ部 Python で部品を組み上げよう 191

第5章 関数 193

5.1 オリジナルの関数	194
5.1.1 関数の必要性和メリット	194
5.1.2 関数を使うための2ステップ	199
5.1.3 関数定義と呼び出し	200
5.1.4 ローカル変数と独立性	202
5.2 引数と戻り値	206
5.2.1 引数	206
5.2.2 複数の引数を渡す	208
5.2.3 戻り値	211
5.2.4 関数呼び出しの正体	214
5.2.5 関数の連携	217
5.3 関数の応用テクニック	220
5.3.1 暗黙のタプルによる複数の戻り値	220
5.3.2 デフォルト引数	221
5.3.3 引数のキーワード指定	225
5.3.4 可変長引数	226
5.4 独立性の破れ	230
5.4.1 グローバル変数	230
5.4.2 引数と戻り値の存在価値	233
5.5 第5章のまとめ	236
5.6 練習問題	237
5.7 練習問題の解答	241

第6章 オブジェクト 243

6.1 「値」の正体	244
6.1.1 format 関数の謎	244
6.1.2 オブジェクトの型	247
6.1.3 文字列オブジェクトが持つメソッド	248
6.2 オブジェクトの設計図	251
6.2.1 オブジェクトの姿を決定づける設計図	251
6.2.2 オリジナルの設計図を作る	254
6.3 オブジェクトの落とし穴	257
6.3.1 オブジェクトの identity	257
6.3.2 参照	259

6.3.3 参照による副作用	263
6.3.4 防御的コピー	265
6.3.5 不変オブジェクト	268
6.4 第6章のまとめ	274
6.5 練習問題	275
6.6 練習問題の解答	277

第7章 モジュール 279

7.1 部品を使おう	280
7.1.1 Python で使える部品たち	280
7.2 組み込み関数	281
7.2.1 組み込み関数とは	281
7.2.2 ファイル入出力	282
7.3 モジュールの利用	287
7.3.1 モジュールとは	287
7.3.2 標準ライブラリ	289
7.3.3 モジュールの取り込み	289
7.3.4 特定の変数や関数だけを取り込む	292
7.3.5 ワイルドカードインポート	295
7.3.6 モジュール取り込みのまとめ	297
7.4 パッケージの利用	299
7.4.1 パッケージとは	299
7.4.2 パッケージ内のモジュールを取り込む	300
7.5 外部ライブラリの利用	304
7.5.1 外部ライブラリとは	304
7.5.2 外部ライブラリの準備	305
7.5.3 matplotlib	306
7.5.4 requests	309
7.6 第7章のまとめ	313
7.7 練習問題	314
7.8 練習問題の解答	316

第8章 まだまだ広がる Python の世界 319

8.1 Python の可能性	320
8.1.1 まだまだ広がる Python の世界	320
8.1.2 ルーチンワークの自動化	321
8.1.3 データベースの操作	322
8.1.4 ウィンドウアプリケーションの作成	324

CONTENTS

8.1.5 Web アプリケーションの作成	326
8.1.6 IoT アプリケーションの作成	328
8.1.7 データ分析・機械学習	331
8.2 Python の基礎を学び終えて	334
8.2.1 終わりに	334
付録 A sukkiri.jp について	337
A.1 sukkiri.jp について	338
付録 B エラー解決・虎の巻	339
B.1 エラーとの上手な付き合い方	340
B.1.1 エラーを解決できるようになる 3 つのコツ	340
B.1.2 エラーメッセージの読み方	341
B.1.3 スタックトレース	342
B.2 エラー虎の巻	345
B.2.1 構文エラーが発生した	345
B.2.2 実行時エラーが発生した	348
B.3 例外処理	361
B.3.1 例外処理とは	361
B.3.2 エラーの内容に応じて対応する	363
索引	366



Column

Python の由来	017	真偽値に評価されない条件式	141
Python 3 系を使おう	019	空ブロックの作り方	146
そのほかの開発環境	030	三項条件演算子	158
円記号とバックスラッシュ	040	無限ループを止める方法	165
代入演算子の特殊性	048	「空っぽ」を意味する None	217
Python の予約語	051	ディクショナリを用いた可変長引数	229
複数の単語から作る識別子の命名規則	052	関数定義と呼び出しのコーディング	235
暗黙の型変換	068	関数さえオブジェクト	250
2 つのタイプの関数	071	コレクション変換関数の正体	254
f-string	072	「箱」より「名札」に近い Python の変数	262
ディクショナリ要素の順序	099	捨てられた不変オブジェクトの行方	273
ディクショナリの合計	099	破壊的な関数	273
コレクションたちの別名	107	不変オブジェクトの再利用	278
ディクショナリへの変換	109	ストリーム	286
1 行 ≠ 1 つの文とならない書き方	122	車輪の再発明	288
チャットボットと AI	127	組み込み関数の正体	303
タブ文字	130	外部ライブラリのインストール	306
文字列の大小比較	135	MicroPython	331
範囲指定の条件式	140	R 言語	333
論理演算子の名前の由来	141	sukkiri.jp	338

第 0 章

ようこそ Pythonの世界へ

Python の世界への入り口となる第 0 章では、
はじめてのプログラムを動かすための基礎知識を紹介します。
本書を通じて一緒に学ぶ仲間とともに、
新しい世界へ旅立つ第一歩を踏み出しましょう。

CONTENTS

- 0.1 ようこそ Python の世界へ
- 0.2 はじめてのプログラミング
- 0.3 Python プログラミングの基礎知識

0.1

ようこそ Python の世界へ

0.1.1 Python を使ってできること

Python とは、プログラムを作るために利用するプログラミング言語の 1 つです。特にデータ分析、AI(機械学習、深層学習)の分野で注目されていますが、ほかにも Web アプリケーション、IoT など幅広い分野で利用することができます。

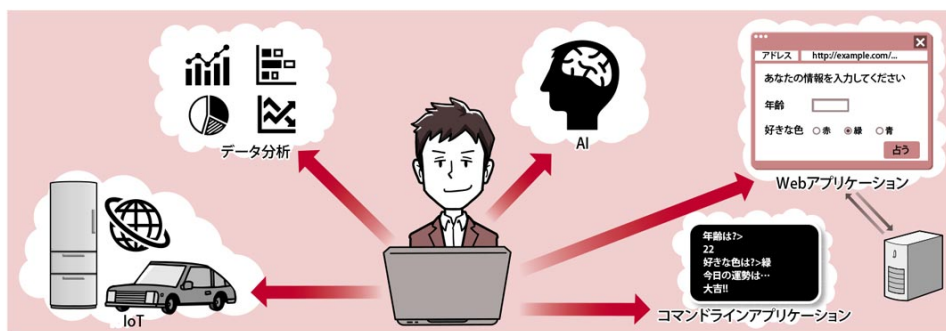


図 0-1 Python を使ってできること

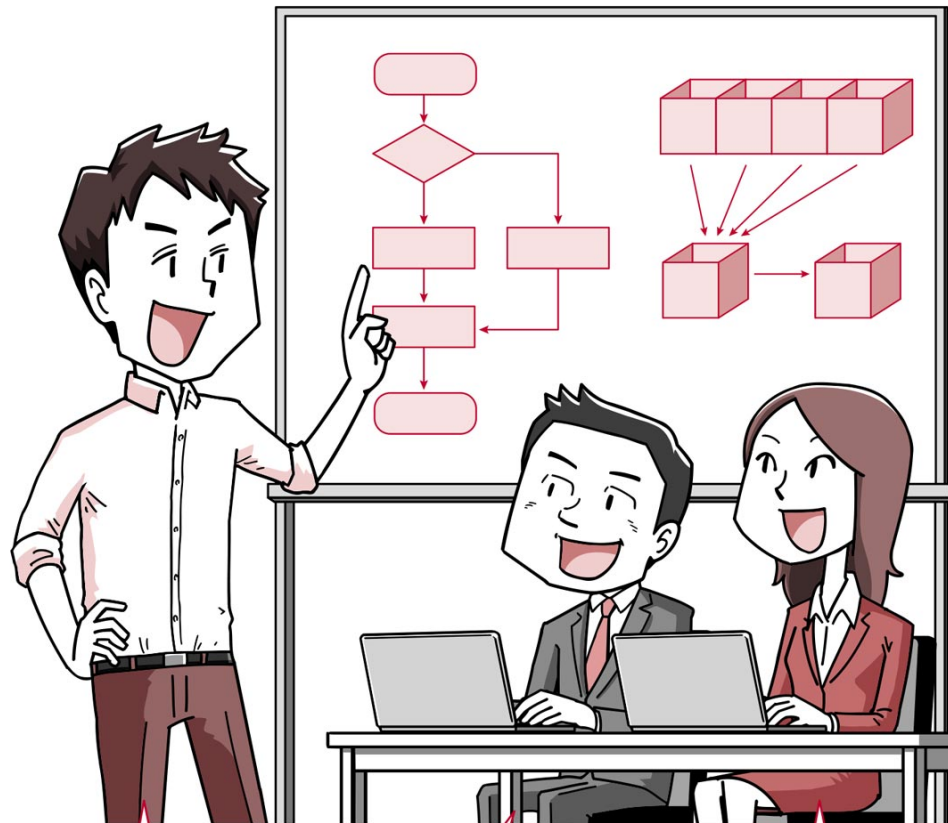
その汎用性の高さに加えて、次のような特徴を持つことから、Python は人気が高く、現在最も注目されているプログラミング言語の 1 つです。

- 基本文法がシンプルで学びやすい。
- 簡潔で読みやすいプログラムを書くことができる。
- 便利な命令が豊富に備わっており、すぐに使い始めることができる。

このような特徴を持つ Python を使って、プログラムを組めるようになりたいと思う人のために、この本は生まれました。はじめてプログラミング言語に触れるという人も、スッキリ理解できて楽しく読み進められるように構成されています。ぜひ実際に手を動かしてプログラミングをしながら、Python をマスターしていきましょう。

0.1.2 一緒に Python を学ぶ仲間たち

この本でみなさんと一緒に Python を学んでいく 3 人を紹介しましょう。

**工藤 慎平 (30)**

(株)ミヤビリンク勤務。システム開発部所属。若いながらも、AI・データサイエンスのパイオニアとして社内で一目置かれている。忙しい業務の合間を縫って、松田と浅木の育成を担当する。三度の飯より数学が好きで、語り出すと止まらなくなる。

松田 光太 (22)

システム開発部所属の新入社員。大学は文系でプログラミングは未経験。上司から、Pythonが流行っているからとりあえず勉強してこいと言われて、工藤に師事。おっちょこちょいで脳天気だが、根性と体力は人一倍ある。とにかくカレーが大好き。

浅木 薫 (24)

マーケティング部門への異動に伴い、データサイエンスの勉強に取り組むことになった新卒3年目。プログラミングは未経験だが、大学時代から機械学習に興味があり独学で学んでいる。要領はよいが、何事もまず原理の理解が大事と考え、ときに細部までこだわってしまうことがある。松田とは大学の陸上部でともに活動していた。

図 0-2 一緒に Python を学ぶ仲間たち

0.2

はじめてのプログラミング

0.2.1

はじめてのプログラミング



それでは、さっそく Python プログラミングを体験してみよう。

はい！



Python のプログラミングを始めるには、開発環境を準備する必要があります。方法はいくつかありますが、本書では **Anaconda** と **JupyterLab** というツールを組み合わせで使用します。付録 A (P337) を参考にしてこれらのツールをセットアップして、JupyterLab を起動しましょう。JupyterLab は、Python のプログラムを入力して実行できる開発環境であり、Anaconda によって準備されたさまざまな Python の機能を利用できるようになります。



JupyterLab が起動したら、四角の枠にこれを書いてごらん。

JupyterLab では、四角の枠にプログラムを入力します。早速プログラムのコードを入力してみましょう (コード 0-1)。

コード 0-1 はじめての Python プログラム

```
1 print('Hello, World')
```

コード 0-1 のコードを入力したら、図 0-3 にある右向きの三角形の▶ ボタンをクリックしてプログラムを実行してみましょう。

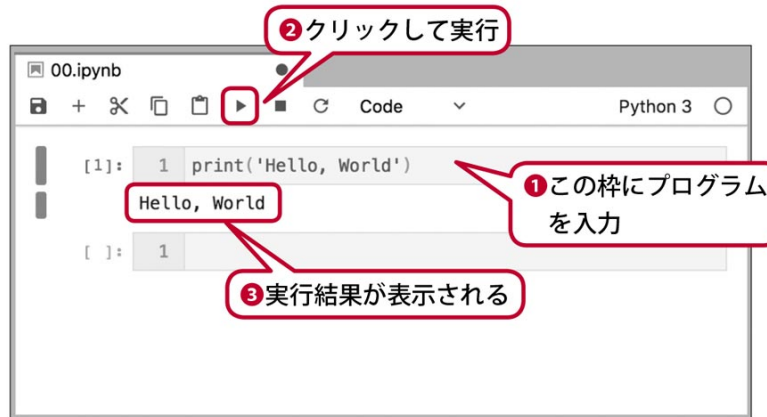


図 0-3 JupyterLab での実行

このプログラムは画面に「Hello, World」という文字列を表示するという単純なものです。現時点でそのしくみを理解する必要はありません。



このプログラムでこれが出るってことは、ひょっとしてここを書き換えれば…。

浅木さんはプログラムを書き替えてみました(コード 0-2)。

コード 0-2 浅木さんが書き換えたプログラム

```
1 print('浅木薫、がんばります!!')
```

実行結果

浅木薫、がんばります!!

「print」の直後にある () の中を書き換えることで、表示される内容を変更する

ことができました。この「print()」のことを「print 関数」と呼びます。関数についてはこれから順を追って解説していきますので、今の段階ではざっくりと、**関数とは命令みたいなもの**だととらえておくといよいでしょう。print 関数を使うことで、画面に文字列を表示させることができましたが、Python にはほかにもさまざまな関数が用意されています。

0.2.2 エラーと上手に付き合う



先輩やりますね！ じゃ、僕も…。

松田くんもプログラムを書き換えてみました(コード 0-3)。

コード 0-3 松田くんが書き換えたプログラム(エラー)

```
1 print('松田くん、カッコいい！ 最高！')
```

実行結果

File "<ipython-input-1-01deb68e97ba>", line 1

```
print('松田くん、カッコいい！ 最高！')
```

^

SyntaxError: EOL while scanning string literal



惜しいね。これだと動かないよ。

松田くんは、カッコよくも最高でもないから…。





そこは関係ないじゃないですか！ あ、「最高！」の後ろに'記号がないからかな？

コード 0-3 は、松田くんが気づいたとおり、文字列の最後に'(シングルクォーテーション)がないため、エラーが発生してしまいます。



たったこれだけのことでエラーになっちゃうのね。ミスしないように書けるか、ちょっと心配だな…。

プログラミングにミスはつきものです。どんなにベテランの開発者であってもミスをしてエラーを起こすことからは逃れられません。大事なのは、**エラーを恐れるのではなく、エラーとの上手な付き合い方を身に付ける**ことです。

そのためには、エラーメッセージを面倒がらずにきちんと読めるようになることが最も近道です。Python のエラーメッセージの読み方と、よくあるエラーの対処方法を付録 B (P339) にまとめてあります。ぜひ活用してください。



Column

Python の由来

Python は、オランダ人のガイド・ヴァンロッサムによって、1989 年のクリスマス前後の暇つぶしとして開発され始めました。Python という名前は、ガイドが熱烈なファンであったコメディ番組『空飛ぶモンティ・パイソン』に由来します。この英単語は「ニシキヘビ」という意味を持つことから、マスコットやアイコンにはヘビがモチーフとして使われています。

0.2.3

セルとノートブック



それじゃ、ほかの文字列も表示してみよう。

1つ目の枠内を書き換えるのではなく、2つ目の枠に次のコード 0-4 を入力してみてください。このコードでは、画面に複数の文字列を表示しています。

なお、プログラムを入力する四角い枠のことを**セル**といいます。2つ目のセルが選択されている（セルの枠線が青色になっている）ことが確認できたら、先ほどと同じように実行してみてください。

コード 0-4 複数の文章を表示する

```
1 print('工藤 慎平')
2 print('30歳')
3 print('三度の飯より数学が好き')
```

実行結果

工藤 慎平

30歳

三度の飯より数学が好き



1 行目の文字列から順番に出てきましたね。

松田くんが言うように、基本的に、プログラムは上から下へと実行されます。

本書で使用している JupyterLab は、セルごとにプログラムを実行することができます。あるセルを実行した結果は、ほかのセルで引き継ぐことになります。そのため、プログラムを複数のセルに分割して個別に実行するなど、分量のあるプログラムでも動作を細かく確認することができます。また、セルをドラッグ・

アンド・ドロップで移動して実行順序を変えることも可能です。

セルをまとめたファイルのことを、**ノートブック** (ipynb ファイル) と呼びます。本書では、1つの章ごとに1つのノートブックを作成し、1つのサンプルコードは1つのセルに記述することを前提としています。

**0
章**

本書での JupyterLab の使い方

- 1つの章で1つのノートブックを作成する。
- 1つのコードは1つのセルに記述する。



Column

Python 3 系を使おう

Python にはバージョン 2 系と 3 系が存在しますが、本書では 3 系を扱っていきます。2 系は 2020 年でサポートが終了する予定になっており、現在は 3 系が主流となっています。また、2 系と 3 系では文法が異なるため、インターネットなどで書き方を調べる際には、対象としているバージョンに注意してください。

0.3 Python プログラミングの基礎知識

0.3.1 開発の流れ



はじめてのプログラミング体験はどうだったかな？ 次に、Python プログラムを開発していく流れを紹介するよ。

Python で作ったプログラムをコンピュータで実行するには、次の手順で行います(図 0-4)。

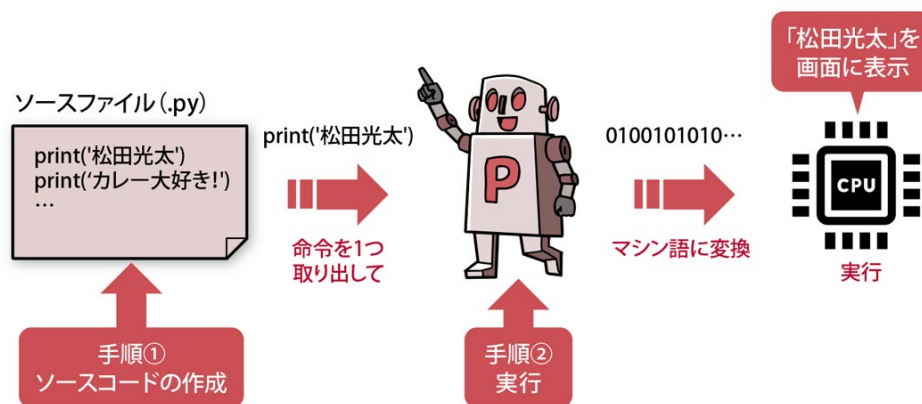


図 0-4 Python プログラムの開発の流れ

それぞれの手順の内容を詳しく見ていきましょう。

手順① ソースコードの作成

Python が定める文法に従って、計算などのコンピュータへのさまざまな命令を記述していきます。このとき、プログラムは「print('Hello, World')」や「100 + 20」のような人間が読んで意味のわかる状態となっています。このようなプログラムを**ソースコード**(source code)、または単にソースやコードといいます。

記述したソースコードはファイルとしてコンピュータに保存します。このファイルを**ソースファイル**(source file)といい、Python のソースファイルには「～.py」という名前を指定するのが習慣となっています。

手順② 実行

ソースファイルとして保存しただけでは、プログラムを動作させることはできません。なぜなら、コンピュータの心臓部である CPU は、**マシン語**(machine code)と呼ばれる言語で記述されたプログラムしか実行できないからです。そこで、**Python インタプリタ**というソフトウェアを使って、ソースコードをマシン語に変換します。

Python インタプリタは、まずソースコードに文法上の誤りがないかをチェックします。もし誤りがあった場合は**構文エラー**(SyntaxError)を表示して変換を中止します(コード 0-3 の実行結果、P016)。

文法上の誤りがなければ、ソースファイルに書かれている命令を 1 つずつマシン語に変換しながら実行していきます(図 0-5)。命令を実行した際に問題が起これば、そこでもエラーが発生します。Python では実行時に起きたエラーを**例外**(Exception)と呼びますが、Python インタプリタは例外が起きたらその内容を表示して実行をストップします。

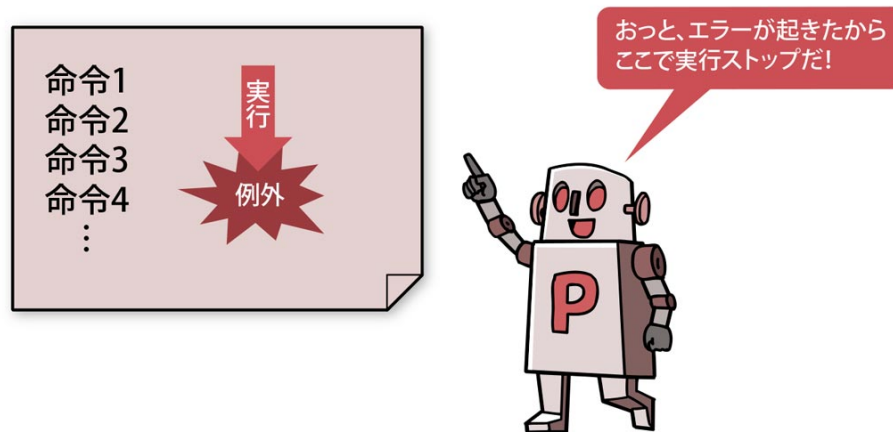


図 0-5 命令は 1 つずつ実行され、例外が起きたら実行は止まる



2 種類のエラー

- 構文エラー (SyntaxError)
文法の誤りによるエラー。発生したら実行されない。
- 例外 (Exception)
実行時に発生したエラー。発生したら実行は中止される。

0.3.2

統合開発環境



でも、さっきの体験では、「～.py」という名前のファイルを作った覚えはないですよ。

Python インタプリタというのも使ってないです。



JupyterLab を使用している場合、前項で紹介した作業を開発者が行う必要はありません。実行ボタンを押した際に、JupyterLab がそのような工程を開発者に代わって行ってくれているのです。

このような開発を補助してくれるソフトウェアを**統合開発環境** (IDE: Integrated Development Environment) といいます。統合開発環境は、プログラムの開発に必要なエディタやインタプリタ、プログラムのバグを検出するデバッガなどの多種多様なツール群を 1 つの画面で利用できるようにしたソフトウェアです。

統合開発環境を使わない場合は、ソースファイルの作成はメモ帳などのエディタ機能のあるソフトウェア、Python インタプリタの実行はターミナルソフトウェアを利用します。



JupyterLab を使わなくても、Python のプログラムを作ったり実行したりできるんですね。

でも慣れないうちは、いろいろなソフトウェアを使いこなしながらプログラムの勉強をするのは不安だわ…。



浅木さんのように感じるプログラミング初心者にも学びやすく、効率的な開発が行えるよう、本書ではすべての章を通して JupyterLab を使用していきます。



JupyterLab は手軽に実行できるというだけじゃないんだ。プログラムをセルに分けて実行したり、グラフを表示できたりといった、便利な機能を数多く備えているから、入門者だけでなく現場のエンジニアにも人気があるんだよ。

JupyterLab を使用するうえで、1 つだけ注意点があります。JupyterLab で作成したファイルはノートブック（～.ipynb）という形式になり、通常の Python のソースファイル（～.py）とは異なります。そのため、ノートブックを直接 Python インタプリタで変換して実行することはできません。



あれ？ さっきインストールしたのは Anaconda でしたよね？
Anaconda が JupyterLab なんですか？

Anaconda は Python プログラム開発のための便利な環境を提供してくれるソフトウェアです。Anaconda をインストールすると、開発に不可欠な Python インタプリタや、JupyterLab のような統合開発環境、外部ライブラリ（第 7 章で解説）がまとめてインストールされます。



トッピング全部乗せカレーみたいなものですね！

0.3.3

プログラムの書き方



よし、環境は整ったかな。ここからはソースコードを書くコツを紹介しておくよ。

ソースコードを記述するにあたっては、意識すべき大切な点が3つあります。

意識① 正確に記述する

ソースコードには、さまざまな文字や数字、記号が登場します。見た目が似ていても、間違った文字を入力するとプログラムは正常に動作しません。特に、次の点に気をつけましょう。

- 英数字は基本的に半角で入力し、大文字／小文字の違いを意識する。
- o (英字のオー) と 0 (数字のゼロ)、l (英字のエール) と 1 (数字のイチ)、;(セミコロン) と : (コロン)、.(ピリオド) と ,(カンマ) を間違えない。
- () (丸カッコ)、{} (波カッコ)、[] (角カッコ) を間違えない。
- ' (シングルクォーテーション) と " (ダブルクォーテーション) の引用符の種類を間違えない。
- カッコと引用符は、必ず閉じる必要がある。

たとえば、次のように記述するとエラーになります。

```
Print('Hello, World')
print{'Hello, World'}
print('Hello, World)
```

— P が大文字
— 波カッコになっている (丸カッコが正しい)
— 引用符を閉じていない



JupyterLab は、' や (を入力したら、対応する ' や) を自動的に書いてくれるよ。

意識② 読みやすいソースコードを記述する

文法に誤りがなくても、人間が読みにくい煩雑なコードや複雑すぎて内容の理解に時間がかかるコードは、修正や改良が難しくなります。特に業務でプログラムを作成する場合、チームメンバーや取引先にソースコードを見てもらうことがあるため、誰が見てもわかりやすい記述をするように心がけましょう。



具体的に、どういう工夫をしたら読みやすいソースコードが書けるようになりますか？

いい質問だね。まずはコメントを活用するといいね。



コメント (comment) とは、ソースコードの中に書き込むことのできる解説文です。プログラムの実行の際には無視され、動作にはまったく影響しません。人間が読むためだけに書くものですから、日本語での記述も可能です(コード 0-5)。

コード 0-5 コメントを入れたプログラム

```
1  """
2  自己紹介プログラム
3  作成者: 工藤 慎平
4  作成日: 20XX年XX月XX日
5  """
6
7  # 名前と特技を表示
8  print('僕の名前は工藤 慎平')
9  print('三度の飯より数学が好き')
```

""" から """ までの複数行のコメント

から行末までの単一行のコメント

1 行のみのコメントとするか、複数行とするかで構文が異なります。



コメント文(単一行)

コメント本文 (行末まで)



コメント文(複数行)

```
"""
```

コメント本文 (複数行の記述が可能)

```
:
```

```
"""
```

※ " の代わりに ' を使用してもよい。

0.3.4

プログラミング体験を終えて

はじめてのプログラミング体験はここまでですが、もっと複雑で高度なプログラムをすることももちろん可能です。Python という言語は、今話題の AI と強く結び付くイメージがあるかもしれませんが、AI 以外にもさまざまな分野で Python のプログラムは動いています。

たとえば、YouTube、Instagram の開発の一部にも Python が使われていますし、ゲームでさえも作ることができます。みなさんが思いつくプログラムの多くを Python で開発することが可能です。ぜひ、「いつか作ってみたいプログラム」を自由に想像してみてください。それが Python を学習するための推進力となることでしょう。



「作りたいプログラムがあること」も上達の近道なんだよ。

私は機械学習を学んで、マーケティングの仕事に活かしたいです。



僕は上司に「流行りだから勉強してこい」って言われてきただけで、まだ何も思いつかないや。これじゃあ上達できないかな…。

そんな顔をしないでいいよ。一番大事なのは「楽しむ」ことだ。



松田くんのように、Python を通してプログラミングの勉強をとりあえずやってみたいという理由で本書を手にとった人は、それが立派な目的になります。文法がシンプルな Python は、プログラミングを最初に学ぶための教育用言語として高く評価されています。マサチューセッツ工科大学やカリフォルニア大学バークレー校をはじめとする多くの名門大学が、プログラミング入門コースの教材に Python を採用しています。



よし、エラーでも何でもどんとこいだ！ 楽しんでやるぞ！！

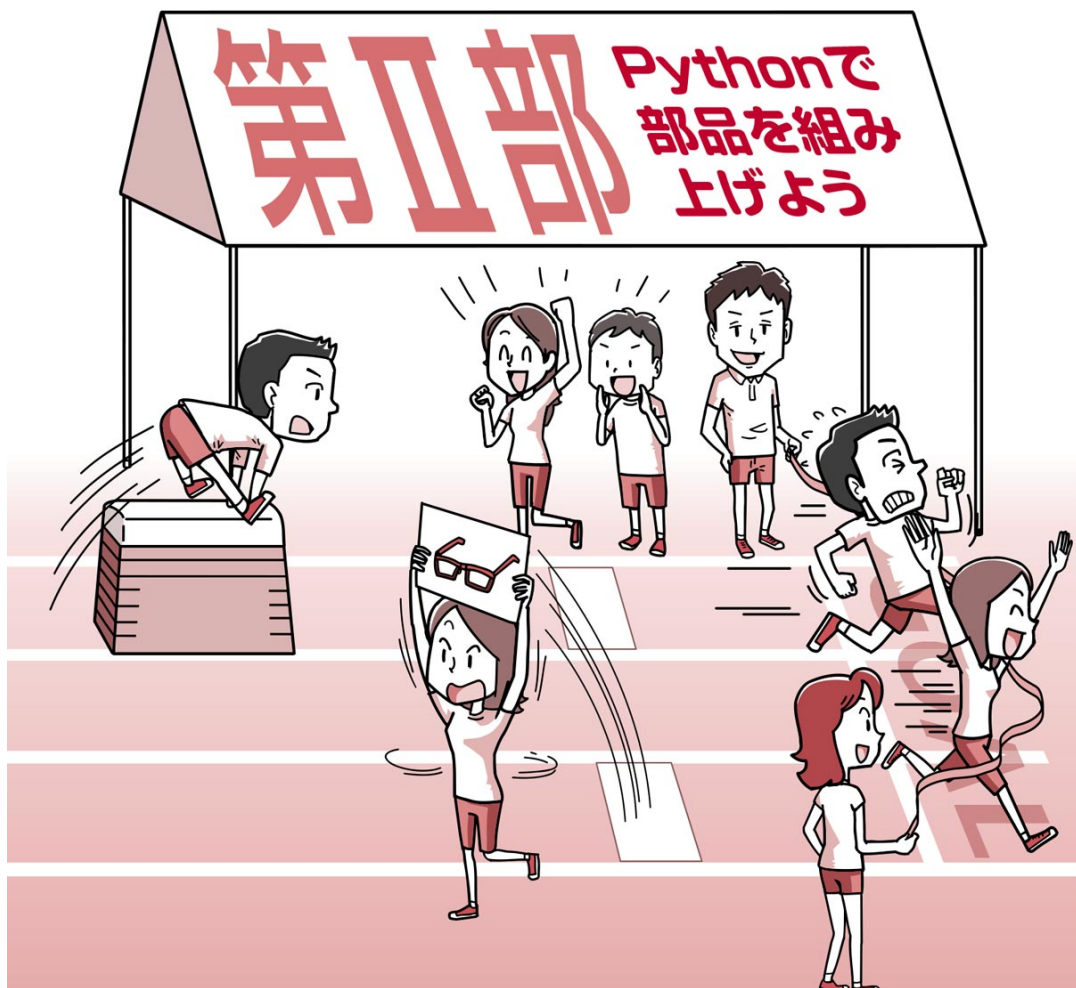
そうそう、それが松田くんらしくていいね！



上達への最も手近な方法は「プログラミングを楽しむ」ことです。次の章からいよいよ Python の学習に入っていきます。エラーが出てもガックリせず、楽しみながら学習を進めていきましょう。



図 0-6 本書による学習のロードマップ





Column

その他の開発環境

Anaconda には、JupyterLab とよく似た Jupyter Notebook というソフトウェアも含まれています。Jupyter Notebook は JupyterLab の前身で、歴史が古く、長らく使用されてきました。しかし、Jupyter Notebook はすでに開発が終了しており、JupyterLab への移行が進んでいることから、本書では JupyterLab を使用します。基本的な特徴は共通ですから、Jupyter Notebook に慣れている場合はそちらを使用してもかまいません。

なお、Python の統合開発環境には、JupyterLab や Jupyter Notebook 以外にも、PyCharm、Spyder などがあります。

第 I 部

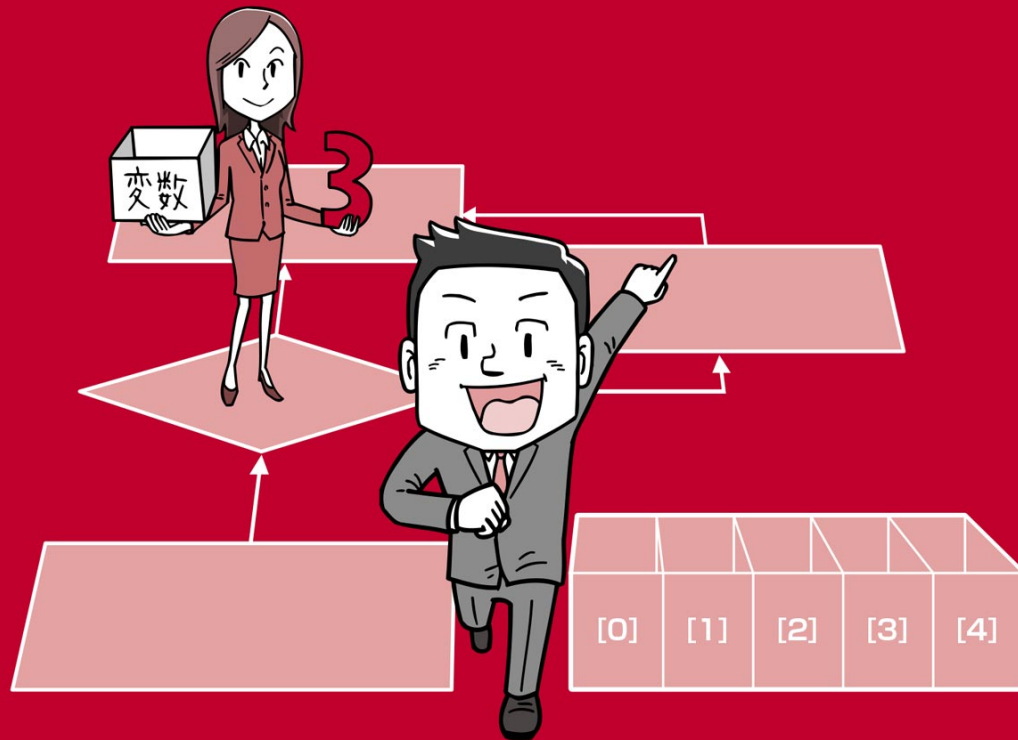
Python の 基礎を学ぼう

第 1 章 変数とデータ型

第 2 章 コレクション

第 3 章 条件分岐

第 4 章 繰り返し



Python プログラミングことはじめ



「Python を楽しむぞ！」って決めたのはいいけど、知識ゼロなんだよね。

なーに、心配することはないさ。誰だってはじめは知識ゼロからのスタートだ。



そうですね。でも、まずは何から学べばいいのかしら…。

Python の場合は、難しくかまえずに、どんどんコードを書いて、動かして、改造して、また動かして…と、試行錯誤を繰り返すのが近道なんだ。



なるほど…。習うより慣れろ、というスタンスですね。

そうだね。実際に手を動かしながらのほうが、文法やしぐみも理解しやすいはずだよ！



さあ、いよいよ本格的に Python を学ぶ旅を始めましょう。知識や経験がゼロでも、過去に挫折してしまったことがあっても大丈夫。入門者にやさしいといわれる Python の基本構文を 1 つずつ楽しみながら学んでいきましょう。

第 1 章

変数とデータ型

Python では、数値や文字列といったさまざまな種類の値を、手軽に効率よく扱うことができます。

このような特徴は、データ分析や機械学習の分野で、Python が広く使われている理由の 1 つとも言えるでしょう。

本章では、データ処理の基本である式のしくみと、変数およびデータ型について説明します。

CONTENTS

- 1.1 式と演算
- 1.2 変数
- 1.3 データ型
- 1.4 第 1 章のまとめ
- 1.5 練習問題
- 1.6 練習問題の解答

1.1

式と演算



それじゃあ、まずは Python でどのような演算をすることができるか、見ていこう！

はい！



1.1.1 数値の演算

第 0 章では print 関数という命令を使って「Hello, World」という文字列を表示しました。実は、この print 関数の () の中には、文字列だけでなく、数値も書くことができます(コード 1-1)。

コード 1-1 print 関数に数値を書く

```
1 print(1)
2 print(10)
```

実行結果

```
1
10
```



数値を書けるということは、計算もできるんですか？ Python でいろいろな計算をやりたいです。

それは Python の最も得意とするところだよ！ それなら、次は計算の基礎の基礎、足し算と引き算を見てみよう。

**1章****コード 1-2 加算と減算**

```
1 print(1 + 1)
2 print(10 - 2)
```

実行結果

```
2
8
```

Python のコード中で、コンピュータに計算をさせるために用いる記号のことを**演算子** (operator) といいます。特に、コード 1-2 に登場した「+」や「-」などの数値の四則演算 (足す・引く・掛ける・割る) をさせるための演算子を**算術演算子**といい、表 1-1 のようなものが存在します。ぜひ、いくつかの演算子を使って、その動作を確認してみてください。

表 1-1 算術演算子の種類

演算子	説明	例	例の表示結果
+	足し算 (加算)	print(10 + 10)	20
-	引き算 (減算)	print(10 - 1)	9
*	掛け算 (乗算)	print(2 * 3)	6
/	割り算 (除算) ※結果は小数	print(7 / 4)	1.75
//	割り算の商 ※結果は整数	print(7 // 4)	1
%	割り算の余り	print(7 % 4)	3
**	べき乗 (るい乗)	print(2 ** 3)	8

1.1.2 文字列の演算



じゃあ次に、コード 1-1 の 1 行目「print(1)」を「print('1)」に書き換えて実行してごらん。

特に結果は変わりませんでしたけど…。' 記号で囲んでも囲まなくても、Python では同じってことかな？



松田くんと同様の疑問を感じたみなさんは、続いて、コード 1-3 のようなコードを実行してみてください。

コード 1-3 文字列の演算

```
1 print('1' + '1')
```

実行結果

11



あれ？ 結果が 11 って表示されました。どうして？

ふふふ、ヒントは'で囲んでいることだよ。この謎、解けるかな？



プログラムのコードに書き込んだ「1」や「25」などの具体的な値のことを、**リテラル** (literal) といいます。特に、コード 1-2 (P035) の「1」「10」「2」のように、数字がそのままの姿で書かれたものを**数値リテラル**といい、その名のとおり数値情報として扱われます。整数のほかにも「3.14」のような小数も記述できます。

一方、プログラムのコードにシングルクォーテーション(')またはダブルクォーテーション(")で囲んで記述された値は**文字列リテラル**といい、文字情報として

扱われます。コード 0-1(P014)に登場した「Hello, World」も文字列リテラルだったというわけですね。



文字列と数値

- 数値リテラル
整数や小数などの数値情報を ' や " で囲まずに記述する。
- 文字列リテラル
文字の並びとしての情報を ' や " で囲んで記述する。



文字列リテラルを囲む記号は、' と " のどちらを使っても意味は同じだよ。ただし、必ず同じ種類の記号で囲む、これがルールだ。

コード 1-3 に登場した「1」は、**数値としての1ではなく、文字としての1を意味するもの**だったことがわかります。そして Python では、文字列情報に対しても一部の算術演算子を用いることができ、**数値情報に対して利用する場合とは異なる動きをします** (表 1-2)。

表 1-2 文字列における算術演算子

演算子	説明	例	例の表示結果
+	文字列の連結 (文字列 + 文字列のとき)	print('1' + '1')	11
*	文字列の反復 (文字列 * 数値または数値 * 文字列のとき)	print(' オラ' * 3)	オラオラオラ



なるほど。これで「11」の謎は、すべて解けましたよ！

これを応用すれば、コード 1-4 のようなプログラムを記述することもできます。

コード 1-4 文字列の演算の応用

```

1 print('Python' + 'の世界へようこそ')
2 print('Pythonは' + 'とっても' * 3 + '楽しいですよ')

```

文字列の連結

文字列の反復

実行結果

Pythonの世界へようこそ

Pythonはとってもとってもとっても楽しいですよ



数だけでなく文字に対しても計算ができるんですね！

コード 1-3 では、「print('1' + '1)」の結果が「11」となりました。これは「1」と「1」の文字をつなげた「11」という 2 文字の文字列であることを意味しているのです。図 1-1 に示すように、人間の目から見れば同じ 1 でも、**プログラムの中では、「数値の 1」なのか「文字列の 1」なのかを厳密に区別**しています。なお、1 文字でも文字列とみなします。

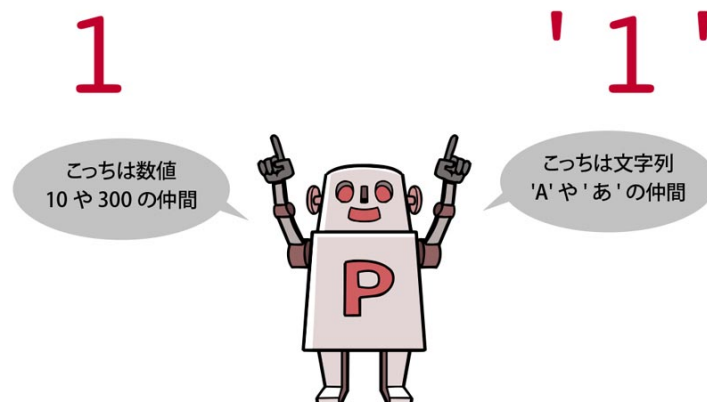


図 1-1 数値の 1 と文字列の 1



情報は、数値なのか文字列なのか、ちゃんと区別して使い分けなきゃいけないのね。

1.1.3 エスケープシーケンス

文字列リテラルを記述する際にしばしば用いられるのが、**エスケープシーケンス**(escape sequence)と呼ばれる特殊な記号です。これは、表 1-3 のように、\ (バックスラッシュ) とそれに続く文字からなる表記で、それぞれ特殊な文字を意味します。

表 1-3 代表的なエスケープシーケンス

表記	意味
\n	改行を表す制御文字
\\	\ (バックスラッシュ)
\'	' (シングルクォーテーション)
\"	" (ダブルクォーテーション)



何ですかこれ？ 何でこんなものが必要なんですか？

文字列を途中で改行したいとき (コード 1-5) や記号を表示したいときに使うんだよ。



コード 1-5 文字列の途中で改行する

```
1 print('はじめまして松田です身体を動かすのが好きです')
2 print('はじめまして\n松田です\n身体を動かすのが好きです')
3 print('引用符には、\'と\"があります')
```

実行結果

```
はじめまして松田です身体を動かすのが好きです
はじめまして
松田です
身体を動かすのが好きです
引用符には、'と"があります
```



Column

円記号とバックスラッシュ

本書では、エスケープシーケンスに用いる記号として\ (バックスラッシュ記号) を紹介しています。しかし、国内の PC や書籍などでは、¥ (円記号) が使われているものを見かけるかもしれません。これは、日本国内でコンピュータが広がり始めた頃、\ 記号のキーに通貨単位である ¥ 記号が割り当てられ、広まった名残りです。

PC の環境やフォントによっては、「\」の入力で「¥」と表示されることもあります。コンピュータ内部ではこの 2 つは同じ文字として扱われるため問題ありません。ただし、Linux の一部や macOS では「¥」と「\」を明確に区別しています。「¥」でエラーになってしまうときは、\ 記号 (macOS では `option` + `¥` キーで入力できます) を使ってみてください。

1.1.4 式と評価

ここまで、数値情報や文字列情報を、演算子を使って Python に計算させる方法を紹介してきました。コード 1-1 ~ コード 1-4 で `print` 関数内に記述した部分は、正式には**式** (expression) といいます。



確かにコード 1-2 の「10 - 2」とかは、数式ですもんね。

でも、コード 1-4 の場合は全然数式っぽくないような…。これも式なんですか？

**1
章**

プログラムの世界でいう「式」は、数学の世界の式にたまたま似ているものもありますが、根本的に異なるものです。プログラムの世界では、コードの中に登場する次のような定義による記述部分を「式」と呼んでいます。

**式とは**

「計算を指示するための記号」である演算子と「その演算子によって計算される情報」であるオペランドが並んでいるもの。

オペランド (operand) とは、「演算子 (operator) によって計算されるもの」という意味で、実際には数値や文字列などの値です。そしてオペランドは多くの場合、演算子の左右に書くことになっています。コード 1-2 の「10 - 2」やコード 1-4 の「'Python' + 'の世界へようこそ」は、その決まりに従って書かれている立派な式なのです。



そして、式に含まれる演算子には「周囲を巻き添えにして、化ける」という特性があるんだよ。

Python におけるすべての式は、実際にプログラムとしてその部分が実行されると、図 1-2 のように「式に含まれる演算子が 1 つずつ、周囲のオペランドを巻き添えにしながら次々と置き換わっていく」という処理がなされ、最終的に 1 つの計算結果となります。この式が処理されていく過程のことを、式の**評価** (evaluation) といいます。

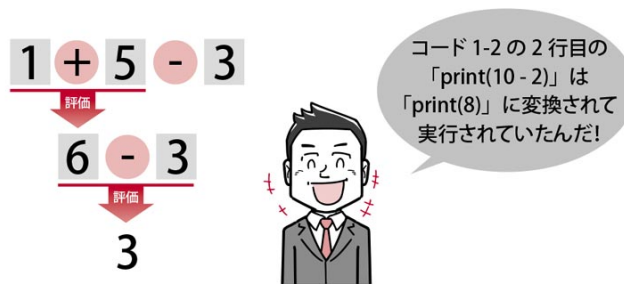


図 1-2 式の評価



でもコード 1-4 の 2 行目って、前から順に評価されていないんじゃない？

浅木さんは、コード 1-4 (P038) の 2 行目の式は、図 1-3 のように評価されるのではないかと考えたようです。

'Pythonは'+ 'ととても' * 3 + '楽しいですよ'
↓ 評価
'Pythonはととても' * 3 + '楽しいですよ'
↓ 評価
'PythonはととてもPythonはととてもPythonはととても' + '楽しいですよ'
↓ 評価
'PythonはととてもPythonはととてもPythonはととても楽しいですよ'



これは間違いだ。
押しが強く
暑苦しいね

図 1-3 コード 1-4 (2 行目) の式の評価 (誤った予想)

実際にこのような実行結果とはならないのは、すべての演算子には**優先順位**というものが定められているからです。



式の評価と演算子の優先順位

1章

- ある式に複数の演算子が含まれる場合、優先順位が高いものから順に処理される。
- 同じ優先順位である複数の演算子がある場合、左にある演算子から処理される。

Python で利用可能なすべての演算子には優先順位が定められています。その詳細は Python の公式ドキュメントで説明されていますが、まずは表 1-4 に示す 3 つのレベルを覚えておけば大丈夫です。

表 1-4 演算子の優先順位（抜粋）

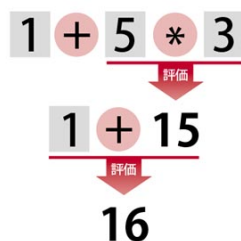
優先順位	演算子
高	**
中	*, /, %
低	+, -



「足し算引き算より、掛け算割り算が優先」って、数学と同じなのね!

なお、式中の一部を丸カッコ () で囲むことで、その部分の優先順位を最高レベルまで引き上げることもできます (図 1-4)。

+ より * のほうが優先順位が高い演算子なので...



() で囲んだ範囲は先に評価される

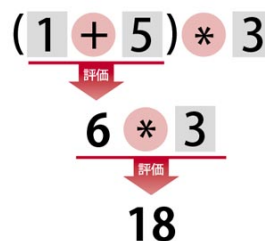


図 1-4 優先順位が高い演算子から評価される

1.2

変数

1.2.1

変数の利用



これでいろいろな計算ができるわね。微分や積分もできるのかしら。

いきなり攻めるね。微積はともかく、本格的な計算処理をやっていくには、今までの知識だけじゃちょっと大変だよ。



これまでは `print` 関数の中に計算式を書いていましたが、その計算の結果は、`print` 関数内でのみ有効です。そのため、たとえばプログラム中で何度も同じような計算をする場合には、その都度同じ式を書く必要がありました(コード 1-6)。

コード 1-6 同じ式が何度も登場してしまう

```
1 print('半径が3cmの円の直径は、')
2 print(3 * 2)
3 print('その円の円周の長さは、直径×円周率で求まるため、')
4 print(3 * 2 * 3.14)
```

直径を再び計算

実行結果

半径が3cmの円の直径は、
6

その円の円周の長さは、直径×円周率で求まるため、
18.84



直径を求めるために「3 * 2」を何度も計算させなくても、一度求めた直径を保存しておけばいいのに。

Pythonに限らず、一般的なプログラミング言語には、計算結果などの値を一次的に保存したり、必要に応じてあとで取り出したりするための道具として**変数** (variable) というしくみが準備されています (図 1-5)。

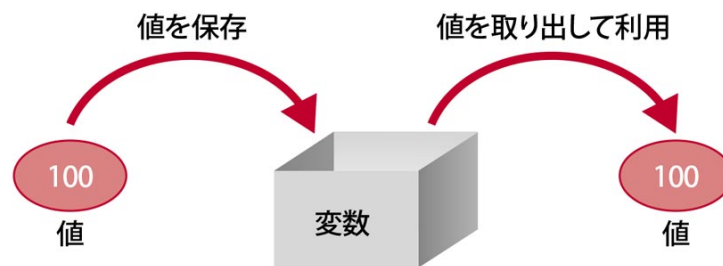


図 1-5 変数

では、変数を利用する例を見てみましょう (コード 1-7)。

コード 1-7 変数の利用

```
1 name = '松田'
2 age = 22
3 print(name)
4 print(age)
```

name という変数 (箱) を用意して、文字列 '松田' を入れる

age という変数 (箱) を用意して、数値 22 を入れる

箱の内容を表示

実行結果

松田

22

コード 1-7 では、変数 `name` に '松田'、変数 `age` に 22 という値を入れ、それぞれの変数の内容を画面に表示しています。変数に値を保存することを「**代入**する」、変数の中の値を取り出して利用することを「**参照**する」ともいいます。



変数の代入

変数名 = 値

※ = (イコール) は「右辺の値を左辺の変数に代入する」ことを意味する。

コンピュータ内に変数を準備して利用可能な状態にすることを、変数を「**定義**する」といいます。変数を定義するために独自の記述が必要なプログラミング言語もありますが、Python では、前述の構文に従って代入を行えば、まだその変数名が存在しない場合には自動的に変数が定義されます。



変数の参照

変数名

※変数名を書くだけで、その中身を取り出せる。



ちょっとマニアックなことを言うと、変数名は評価されて「中身の値」に化けるんだ。「`print(age)`」は、実は「`print(22)`」に変換されてるんだよ。

なお、「箱に入れたボールを取り出すと、箱の中からはボールがなくなる」とのは違い、変数に入れた値を参照しても、中身は変わりません。変数の値は繰り返し参照することができます。

それでは、ここまでの知識を使ってコード 1-6 (P044) を改善してみましょう (コード 1-8)。

コード 1-8 変数を利用してコード 1-6 を改善

```
1 print('半径が3cmの円の直径は、')
2 dia = 3 * 2 # diaはdiameter (直径) の略
3 print(dia)
4 print('その円の円周の長さは、')
5 print(dia * 3.14)
```

計算結果を変数
dia に代入

計算結果を利用

計算結果を再利用

実行結果

```
半径が3cmの円の直径は、
6
その円の円周の長さは、
18.84
```

今回の例では、「 $3 * 2$ 」という簡単な掛け算の重複を回避できたただけですが、計算だけでなく、データを再利用する場面はたくさんあります。変数はさまざまな処理に役立ちますから、変数を上手に操れるか否かが、最も基礎的かつ重要な鍵と言えるでしょう。



数学と同じで、変数を上手に使えることが大事なのね。でも…
うーん、「age = 22」を見ると、どうしても「age と 22 は等しい」って一瞬思っちゃうのよね。

Python では、「=」には「代入する」という意味しかないんだ。数学との違いには気をつけてほしい。



代入に使用する = 記号のことを**代入演算子**といいます。「左辺と右辺が等しい」という意味では使用しないので注意してください（「等しい」という意味を持つ記号は別に存在します。第 3 章で紹介します）。



Column

代入演算子の特殊性

変数に値を代入するために用いられる = 演算子は、Python に限らず多くのプログラミング言語に存在し、代入演算子と呼ばれています。実は Python の厳密な仕様では、= は代入構文の一部に用いられる記号として定められており、式を構成する演算子ではありませんが、本書ではより理解しやすいよう演算子とみなして紹介します。なお、原則として式中では左の演算子から評価が行われると紹介しましたが (P043)、代入演算子のみ例外で、「右から評価」されます。また、代入演算子の優先順位は、全演算子の中で最も低く設定されています。

1.2.2 変数名のルール



よし、変数をいろいろ使ってみるぞ!! あれっ…、日本の人口を変数「japan」に、世界の人口を変数「global」に入れようとしたら、エラーが出ちゃった…。

おっと、大切なことを伝え忘れていたよ。



変数などの名前として使う文字や数字の並びのことを**識別子** (identifier) といいます。識別子は原則として自由に決めることができます。「name」や「age」はもちろん、「ゴール数」「標高」「 α 」などの英数字以外の文字を使った名前も可能ですが、次のようなルールや慣習を覚えておきましょう。

①予約語は使用できない

Python には、識別子として使用できない単語が 35 個ほどあり、これらは**予約語** (keyword) といいます (P051 参照)。たとえば if や for、global や with など は予約語であり、これらを変数名として利用することはできません。

②先頭の文字は数字であってはならない

識別子は数字で始まることは許されていません。

③先頭に `_` を 2 つ付けた名前は原則として使用しない

先頭に `_` (アンダースコア) 記号を 2 つ付けた名前は、Python 自体のために予約された名前であり、特別な用途で使用するになっています。原則として予約語と同様に利用を避けてください。

④大文字／小文字、全角／半角は区別される

大文字／小文字、全角／半角の違いは完全に区別されます。たとえば変数 `name` と `Name` は別のものとして扱われます。

⑤小文字で始まるわかりやすい名前が望ましい

Python では、変数名には小文字で始まる名詞形の名前を付ける慣習があります。また、その変数に格納される情報の内容を誰もが想像しやすくするために、具体的な名前とするのが望ましいでしょう。一部例外はありますが、a や s のような 1 文字の変数名、data や flag などの抽象的で内容がわかりにくい変数名は避けるようにしましょう。なお、本書では紙面の都合により短い変数名を用いることがあります。



そうか、「global」は予約語だから変数名に使えないんだね。

1.2.3 変数の上書き

変数には値を何度でも代入することができます。変数にいったん値を代入したあと、別の値を代入したらどのようなになるか見てみましょう (コード 1-9)。

コード 1-9 変数の上書き

```
1 age = 20 変数 age に 20 を代入
2 print('浅木の年齢は')
3 print(age)
4 age = 24 今度は変数 age に 24 を代入
5 print('うそ。本当は')
6 print(age)
```

実行結果

```
浅木の年齢は
20
うそ。本当は
24
```

すでに定義されている変数に代入を行うと、新しい変数が定義されるのではなく、その変数の値を上書きします。つまり、一度使ってもう利用しなくなった変数に、別の数値を代入して使い回す (再利用する) ことが可能なのです。

コード 1-9 のように、「浅木さんの年齢」という 1 つのデータを入れるために変数 `age` を再利用するのは問題になりにくいのですが、まったく別のデータを代入するために、1 つの変数を使い回すことは危険が伴います。

たとえば、変数 `x` に身長 の値を入れて処理をしたあと、次は体重の値を入れて別の処理をするコードも書けます。しかし、このようなコードは、変数 `x` に現在どのような値が入っているかが不明確になり、思わぬ不具合の原因になりやすいことが知られています。原則として**変数の再利用は避ける**ことをお勧めします。



変数は再利用しない

予期せぬ不具合を避けるため、原則として、変数は使い回さない。

1
章



P049 で紹介した「具体的な名前を付ける」ことを心がけていれば、自然に再利用は減るはずだ。

そうかもしれないですが、自信ないなあ…。何かいい策はないんですか？



システムの安全装置じゃないんだが、Python プログラマがよくやる手があるよ。

Python では、中身を絶対に書き換えられたくない変数は、目立つように大文字の変数名を付ける慣習があります（「TAX_RATE」など）。みなさんもこのような大文字の変数名を見かけたら、その変数には代入しないように注意しましょう。



Column

Python の予約語

バージョン 3.7 の時点では、35 個の単語が予約語とされていますが、バージョンアップにより変更される可能性があります。利用している環境での予約語は次のコードで確認できます。

```
1 import keyword
2 print(keyword.kwlist)
```

なお、本書掲載のコードでは、予約語を `global` のように表しています。



Column

複数の単語から作る識別子の命名規則

複数の単語をつなげて変数名を作るには、いくつかの方法があります。

- アッパーキャメルケース : MyAge、UploadData
- ロワーキャメルケース : myAge、uploadData
- スネークケース : my_age、UPLOAD_DATA
- チェインケース : my-age、UPLOAD-DATA

どの方法にも一長一短があり、どれを選ぶかは基本的に開発者の自由ですが、同じ種類の識別子に対して複数の方法を混在させずに統一して用いることが重要です。なお、Python の標準コーディング規約である **PEP8** では、変数名にはスネークケースを推奨しています。

1.2.4

まとめて代入 (アンパック代入)



それにしても、変数がたくさん出てくると、それだけでプログラムが長くなっちゃいそうですね。

たくさんの変数を使いこなすようになってくると、変数への代入だけで数十行を費やす可能性があります。そのため、できるだけソースコードをシンプルに記述したいと思うような場面も出てくるかもしれません。そのようなときは、複数の変数定義を 1 行でまとめて書くことができます。コード 1-10 を見てみましょう。

コード 1-10 複数の変数をまとめて定義

```
1 name, age = '浅木', 24
```

変数と値をそれぞれ「,」(カンマ)でつなげて書きます。この代入方法のことを**アンパック代入**と呼びます(図 1-6)。

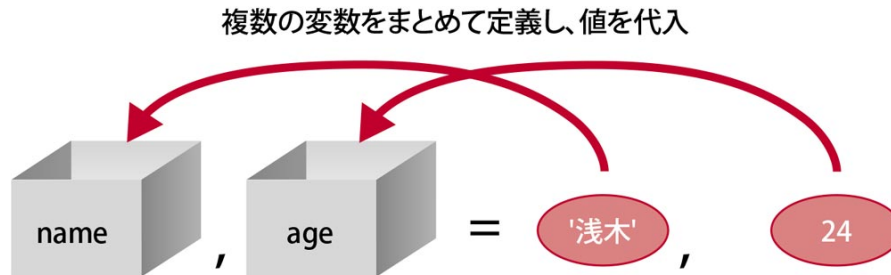


図 1-6 アンパック代入

アンパック代入を用いることで、プログラムの行数を減らすことができます。ただし、あまり多くの変数定義を 1 行にまとめると、どの変数にどの値が代入されているのかわかりづらくなったり、変数と値の数が一致せずにエラー (ValueError) が起こりやすくなったりするので注意しましょう。

1.2.5 自分自身への代入



浅木先輩の年齢を予測するプログラムを作ってみましたよ。

いったい何に使うプログラムなんだい。しかも、結果がおかしいじゃないか。



松田くんが作ったプログラムを見てみましょう(コード 1-11)。

コード 1-11 浅木さんの年齢を予測するプログラム

```
1 age = 24
2 print('浅木先輩の今年の年齢は...')
```

```
3 print(age)
4 age + 1
5 print('来年は…')
6 print(age)
7 age + 1
8 print('再来年は…')
9 print(age)
```

実行結果

浅木先輩の今年の年齢は…

24

来年は…

24

再来年は…

24



「永遠の 24 歳」なので問題ありません。

いやあ、いくら先輩が化け物でも、いくら何でも…。あ、ハイ、何でもありません。



コード 1-11 が世の中の一般的な常識では異常な動作となってしまう原因は、4 行目と 7 行目です。それぞれ「age + 1」という記述がありますが、これは「変数 age の値と 1 を足す」という処理をただけです。その計算結果はどこかに保存されることはなくその場で消えてしまいます。もちろん変数 age の内容が書き換わることもありません。

変数 age の値を増やしていくためには、この 2 箇所をコード 1-12 のように修正します。

コード 1-12 浅木さんの年齢を予測するプログラム(修正版)

```
1 age = 24
2 print('浅木先輩の今年の年齢は...')
3 print(age)
4 age = age + 1
5 print('来年は...')
6 print(age)
7 age = age + 1
8 print('再来年は...')
9 print(age)
```

変数 age の内容 (24) に 1 を加えた結果を変数 age に代入

変数 age の内容 (25) に 1 を加えた結果を変数 age に代入

実行結果

浅木先輩の今年の年齢は…

24

来年は…

25

再来年は…

26



先輩が化け物から人間に戻った！ よかった～。

もう!! でもそれより「age = age + 1」って、すごく気持ち悪いんですけど…。



この数式に独特の違和感を持つ人も少なくないでしょう。しかし、これまでに登場した次の3つのPythonのルールを思い出しながら、ゆっくり整理していけ

ば、きっとスッキリ納得できるはずです。

- ・ 式は評価され、計算結果に置き換わる (P041)。
- ・ 代入演算子は優先順位が一番低く、右から順に評価される (P048)。
- ・ 変数名は中身の値に評価される (P046)。

これらのルールを踏まえると、「age = age + 1」は図 1-7 のように処理されることになります。

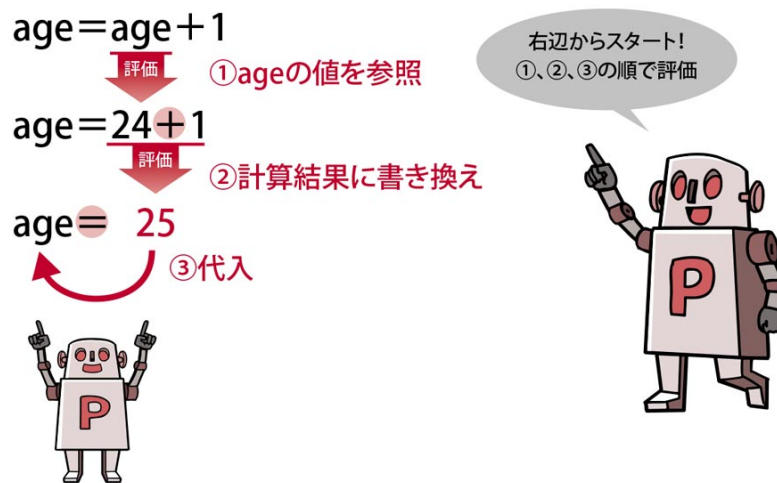


図 1-7 「age = age + 1」の評価過程



なるほど。ここでも Python は「評価」っていうシンプルな原理に従って動いてるのね…。何か美しさを感じるわ…。

ははは。気持ちはわかるけど、松田くんがちょっと引いてるぞ。



1.2.6 複合代入演算子

前項の「age = age + 1」のように、ある変数の現在の値に対して計算を行いたいことはよくあります。たとえば「変数 price の中身を 1.5 倍にする」場合も、「price

= price * 1.5」と記述することになるでしょう。

このように、「変数の現在の値に加減乗除する場合」は、コード 1-13 のように書くことも可能です。

コード 1-13 変数の現在の値に加減乗除する場合

```
1 age += 1
2 price *= 1.5
```

「age = age + 1」と同じ


「price = price * 1.5」と同じ

このように算術演算子と =(代入演算子)を組み合わせたものを**複合代入演算子**といいます。Python では、表 1-5 のものを利用できます。

表 1-5 主な複合代入演算子

演算子	説明
+=	右辺の値と左辺の変数の値を足し算して変数に代入
-=	右辺の値と左辺の変数の値を引き算して変数に代入
*=	右辺の値と左辺の変数の値を掛け算して変数に代入
/=	右辺の値と左辺の変数の値を割り算して変数に代入

1.2.7 キーボード入力値の代入



おめでとう。変数については、ここまで理解できればまずは合格と言っていいだろう。ご褒美に、ちょっと「楽しい代入」を紹介しよう。

これまで私たちは、コード内で決まった値だけを変数に代入してきました。しかし、次の構文を使うことで、「プログラムの実行時にユーザーが入力した値を変数に代入する」ことが可能になります。



キーボードからの入力を変数に代入

変数名 = input(文字列)

※文字列には、'名前を入力してください'のように、ユーザーに入力を促すための文字列を記述する。

この構文の右辺に使われているのは、**input 関数**という Python の代表的な命令です。この命令は、実行されると () 内に指定された文字列を画面に表示して、ユーザーのキーボード入力を待ちます。そして、いざ入力があると、input(文字列) の部分が入力内容に「化ける」という動きをします。



ちなみに「化ける」とあるように、命令実行も実は「評価」なんだ。

少し複雑な動作をする関数ですが、まずは理屈抜きでコード 1-14 を実行してみましょう。動きを実際に体験することが納得への近道です。

コード 1-14 キーボードから値を入力する

```
1 name = input('あなたの名前を入力してください >>')
2 print('おお' + name + 'よ、そなたがくるのを待っておったぞ!')
```

実行結果

あなたの名前を入力してください >>

まずinput関数の()の中に指定された文字列が表示され、あたかも実行が止まっているかのように見えます。しかし実は、実行が止まるのではなく、プログラムはユーザーがキーボードから何かを入力してくれるのを待ち続けているのです。



試しに、適当に文字列を入力して、`[Enter]` キーを押してごらん。

実行結果の続き

あなたの名前を入力してください >>浅木
おお浅木よ、そなたがくるのを待っておったぞ！



すごい！ プログラムと対話してるみたい！

工藤さん、さっそく `input` 関数を使って、割り勘計算のプログラムを作っちゃいましたよ。これで、来週末の合コンはモテモテだ〜♪



松田くんの作ったプログラムを見てみましょう（コード 1-15）。

コード 1-15 割り勘計算プログラム

```
1 price = input('料金を入力 >>')
2 number = input('人数を入力 >>')
3 payment = price / number
4 print('お支払いは' + payment + '円です')
```



見ててくださいよー。15,000 円を 4 人で分けて…っと。

実行結果

料金を入力 >>15000

人数を入力 >>4

```
-----
TypeError Traceback (most recent call last)
<ipython-input-20-5bc4cf8de530> in <module>
      1 price = input("料金を入力 >>")
      2 number = input("人数を入力 >>")
----> 3 payment = price / number
      4 print("お支払いは" + payment + "円です")

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

TypeErrorが発生

TypeErrorが発生した行

エラーの原因



あれ？ TypeError…。何だこれは!?

Type というのは「データ型」のことだよ。「データ型」は、変数を使いこなすためには絶対に欠かせないんだ。次節からはこの「データ型」について紹介しよう。



とほほ、モテモテへの道は、まだ遠いか…。

1.3

データ型

1章

1.3.1 データ型とは

これまで私たちは、数値と文字列の2種類の値を扱ってきました。数値や文字列といった値の種類のことを**データ型**(data type)または単に**型**といいます。Pythonでは、この2つの種類以外に、表1-6のようなデータ型を扱うことができます。

表 1-6 主なデータ型

データ型の名称	説明	例
int	整数	100、-100
float	小数	3.14、-0.5
str	文字列	"Hello"、'カレー'
bool	真偽値	True、False



これまで「数値」と呼んでいたものは、正確には「整数」と「小数」の2種類に分かれるのね。

数値と文字列は今まで使ってきたものですね。それと、真偽値？
何ですかこれは？



表1-6の最後の真偽値に関しては第3章で説明するから、今は気にしないでいいよ。

Pythonの変数には、どのようなデータ型の値でも代入することができます(図1-8)。また、1つの変数に何度も代入を行う場合、その都度異なるデータ型の値を入れることができます(コード1-16)。

コード 1-16 変数には異なるデータ型の値を代入できる

```

1 x = '松田' # 名前
2 print(x)
3 x = 23     # 年齢
4 print(x)
5 x = 175.6  # 身長
6 print(x)

```

str 型の値を代入

int 型の値を代入

float 型の値を代入

実行結果

松田

23

175.6

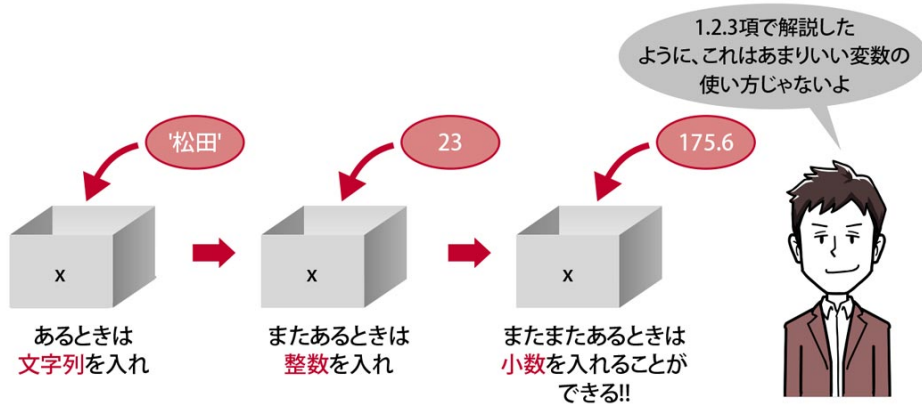


図 1-8 変数とデータ型

プログラミング言語によっては、ある変数を定義する際に、変数自体の型を決める必要があります。たとえば Java では、int 型の変数 age を定義した場合には、この変数 age には int 型の値しか代入できなくなります。一見すると不便に思うかもしれませんが、変数 age に文字列などの int 型以外の値が格納されてしまう可能性は万が一にもありませんので、安心して四則演算などの計算に使うことが

できます。

一方、Python の場合、変数自体に型の定めはなく、どのような種類の値でも格納できます。便利な半面、どのような型の情報が格納されているかがわからなくなったり、期待している種類とは異なる型の値が格納されてしまったりすることもあります。



値と変数とデータ型

値にはデータ型の定めがあるが、変数はデータ型を持たない。

そこで利用されるのが、**type 関数**という Python が備える命令です。この命令は、ある変数にどのような型の値が格納されているかを調べることができます(コード 1-17)。

コード 1-17 格納されている値のデータ型を調べる

```
1 x = 10
2 print(type(x))
```

変数 x に格納されている値のデータ型を調べて表示

実行結果

```
<class 'int'>
```

実行結果を見ると「int」と出力されています。つまり、変数 x には現在、int 型(整数型)の値が代入されていることがわかります。



地味だけど、データ型の確認は、トラブルシューティングを行ううえで重要になっていくから、しっかり覚えておこう。



type 関数

type(変数名)

※変数に代入されている値のデータ型を調べることができる。

※ print 関数内で使用することで、調べた結果を表示できる。

type 関数を使って、松田くんの「割り勘計算プログラム」(P059 のコード 1-15) の問題点を探ってみましょう。まずはキーボードから入力された値のデータ型を調べてみます(コード 1-18)。

コード 1-18 割り勘計算プログラムの問題点

```
1 price = input('料金を入力 >>')
2 print(type(price))
```

実行結果

```
料金を入力 >>15000
<class 'str'>
```



ああっ！ 文字列になってる！ だから、割り算ができなかったのか。

input 関数でキーボードから入力した値は、文字列として扱われます。そのため、「10」と入力しても変数に代入されるのは文字列の「'10'」です。文字列の値は、四則演算に用いることができないため、エラーが発生していたのです。



じゃあキーボードから入力した値は計算に使えないってことですか？

それだと困るだろう？ そんなときは、データ型を変換すれば大丈夫なんだ。



1.3.2 データ型の変換

Python には、ある値のデータ型を別のデータ型に変換するために、表 1-7 のような関数が準備されています。

表 1-7 データ型変換のための命令

関数名	例	説明
int 関数	int(x)	変数 x の値を int 型に変換 (x が小数の場合は小数点以下を切り捨て。数値として解釈できない文字列の場合はエラー)
float 関数	float(x)	変数 x の値を float 型に変換 (数値として解釈できない文字列の場合はエラー)
str 関数	str(x)	変数 x の値を str 型に変換
bool 関数	bool(x)	変数 x の値を bool 型に変換

データ型を変換する例を見てみましょう (コード 1-19)。

コード 1-19 データ型の変換

```

1  x = 3.14
2  y = int(x)
3  print(y)          # 変換結果を表示
4  print(type(y))    # 変換後のデータ型を表示
5  z = str(x)
6  print(z)          # 変換結果を表示
7  print(type(z))    # 変換後のデータ型を表示

```

変数 x の値を int (整数) 型に変換した結果を変数 y に代入

変数 x の値を str (文字列) 型に変換した結果を変数 z に代入


```
8 print(z * 2)
```

実行結果

3 **int 型に変換したことで、小数点以下が切り捨てられる**

```
<class 'int'>
```

```
3.14
```

```
<class 'str'>
```

3.143.14 **文字列なので「*」で「3.14」を反復する**

データ型によって、できることとできないことに違いがあります。たとえば、**int 型は str 型のように「+」による連結ができませんし、str 型は四則演算をすることができません**。また、同じ演算子を使っても、データ型によって結果が異なることもあります。たとえば、「*」は整数や小数に用いると掛け算になり、文字列に用いれば反復になります (P037)。

現在取り扱っている値のデータ型をしっかりと把握し、必要に応じて変換することで、必要な処理を行えるようになるのです (図 1-9)。

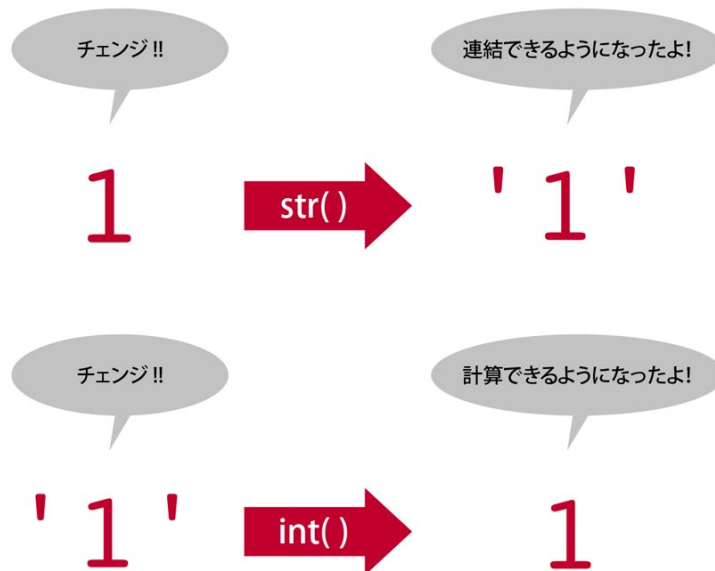


図 1-9 値のデータ型を変換



ということは、僕の割り勘計算プログラムは、こうすればいいんですね！

コード 1-20 割り勘計算プログラムの修正 (未完成)

```

1 price = input('料金を入力 >>')    # キーボード入力結果はstr型
2 price = int(price)    str から int へ変換
3 number = input('人数を入力 >>')    # キーボード入力結果はstr型
4 number = int(number)    str から int へ変換
5 payment = price / number    # 割り算の結果はfloat型
6 payment = int(payment)    float から int へ変換 (小数点以下切り捨て)
7 print('お支払いは' + payment + '円です')
```

実行結果

```

-----
TypeError                                Traceback (most recent call last)
<ipython-input-24-981aed201826> in <module>
      5 payment = price / number
      6 payment = int(payment)
----> 7 print('お支払いは' + payment + '円です')

エラーが発生した行
(5行目の割り算では発生していない)

TypeError: can only concatenate str (not "int") to str
```



惜しい。もう1箇所「型の変換」が必要な場所があるんだ。エラーメッセージをよく見てごらん。

はっ！ まさか、コレですか!?



みなさんは、コード 1-20 のどこを直すべきか想像できましたか。正解は 7 行目です。

```
7 print('お支払いは' + str(payment) + '円です')
```

文字列と数値とを + 演算子で連結できるプログラミング言語も存在しますが、Python の場合はできません。**演算は、文字列同士か数値同士で**。それが原則です。



Column

暗黙の型変換

代入や演算に際して、自動的に型変換が行われるしくみを**暗黙の型変換**といいます。Python では数値（整数や小数）と文字列の間では暗黙の型変換が行われないため、str 関数などで**明示的な型変換**を行う必要があることは、本文で紹介したとおりです。

なお、Python でも、オペランドが「整数と小数の数値同士の組み合わせ」の場合、暗黙の型変換により小数に揃えられてから演算されることとなっています（例： $1 + 2.2 \rightarrow 1.0 + 2.2 \rightarrow 3.2$ ）。

1.3.3

文字列の中に数値を埋め込む



工藤さん、変数を使って自己紹介を書いてみたんですが、コードが読みにくくて…。

数値は数値同士、文字列は文字列同士でしか演算ができないという原則は、頭では理解できても、実際にプログラムを作成しようとするとき面倒だと感じる人もいるでしょう。たとえば、次のように文字列の中に数値を埋め込むプログラムでは、ソースコードが少々読みにくくなってしまいます（コード 1-21）。

コード 1-21 文字列に数値を埋め込む

```
1 name = '松田光太'
2 age = 23
3 height = 175.6
4 print('私の名前は' + name + 'で、年齢は' + str(age) +
      '歳で、身長は' + str(height) + 'cmです')
```

実行結果

私の名前は松田光太で、年齢は23歳で、身長は175.6cmです



うーん、確かに型の変換や「+」がいっぱいで見にくいわね…。

そこで！ 第1章の締めくくりに、とっておきの便利な命令を紹介しよう！



先ほどのコード 1-21 は、コード 1-22 のように書き換えることができます。

コード 1-22 format 関数で文字列に数値を埋め込む

```
1 name = '松田光太'
2 age = 23
3 height = 175.6
4 print('私の名前は{}で、年齢は{}歳で、身長は{}cmです'
      .format(name, age, height))
```

埋め込む値

値を埋め込む場所

実行結果

私の名前は松田光太で、年齢は23歳で、身長は175.6cmです



こっちのほうが、断然見やすいですね！

コード 1-22 では、「ある文字列の中のさまざまな場所に、値を逐次埋め込む」ことができるように、次の **format 関数** の構文を活用してエレガントな表記を実現しています。

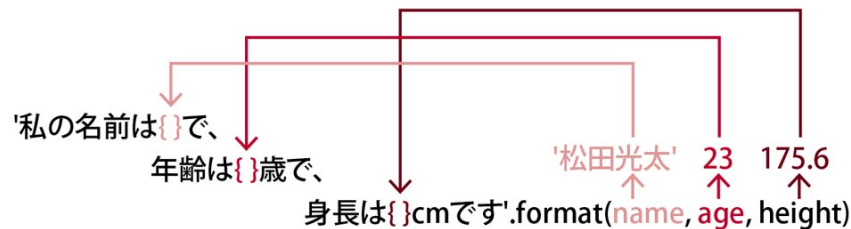


文字列の中に値を埋め込む

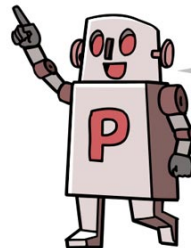
'{' を含む文字列 '.format(埋め込む値 1, 埋め込む値 2, ...)

※文字列中の値を埋め込みたい場所に「{'」を書く（複数可）。

※ format の（と）の間に、埋め込みたい値をカンマ区切りで並べる。



format の直前に
ピリオド (.) 記号を付け忘れ
ないように注意してほしい



順番に値を
埋め込むよ

図 1-10 format 関数の利用

文字列の中に記述した{}を、**プレースホルダー**といいます。format のカッコの中に並べた値は、左から順番にプレースホルダーに埋め込まれていきます(図 1-10)。このとき、値は自動的に文字列型に変換されます。すべての値を埋め込み終わると、この構文全体が 1 つの文字列に化けます。



Column

2つのタイプの関数

この章では、print 関数、input 関数、type 関数、そして format 関数など、たくさんの命令を学びました。しかし、最後の format 関数だけは少し使い方が異なることに気づいた人もいるでしょう。

Python は私たち開発者のために、ほかにもたくさんの関数を準備してくれていますが、実は呼び出し方の違いによって、2 つのタイプに分類できます。

- ①「関数名(～)」で呼び出すもの(例: print 関数、input 関数)
- ②「値. 関数名(～)」で呼び出すもの(例: format 関数)

どちらの呼び出し方をするかは関数によって決まっていますが、現時点では丸暗記しておけば十分です。詳細は第 6 章で紹介します。



よし！ これで割り勘計算プログラムが完成したぞ！！

割り勘計算プログラムの完成版を見てみましょう(コード 1-23)。

コード 1-23 割り勘計算プログラム(完成版)

```
1 price = int(input('料金を入力 >>'))
2 number = int(input('人数を入力 >>'))
```

```
3 payment = int(price / number)
4 print('お支払いは{}円です'.format(payment))
```

実行結果

料金を入力 >>15000

人数を入力 >>4

お支払いは3750円です



うん。今まで学習したことをうまく融合させているね。



Column

f-string

コード 1-22 の 4 行目は、Python 3.6 で導入された f-string という機能を使用すると、より簡潔に書くことができます。

```
print(f'私の名前は{name}で、年齢は{age}歳で、身長は{height}cmです')
```

文字列の直前に「f」を付けると、数値が格納された変数名を{}の中に直接指定することができ、より直感的に理解しやすいコードが書けます。また{}の中には式を記述することも許されており、実行時にはその評価結果が埋め込まれます。

1.4

第 1 章のまとめ

1
章

この章では、次のことを学びました。

式と演算子

- ・ プログラムは、さまざまな部分に式を含むことができる。
- ・ 式は、演算子とオペランドによって構成される。
- ・ 算術演算子を使うと、数値の四則計算や文字列の連結が行える。
- ・ 式は実行されると評価され、式に含まれる各演算子が周囲のオペランドを巻き添えにしながら計算結果に置き換わっていく。
- ・ 優先順位が高い演算子から順に、同じ順位の場合は左から評価される。

リテラルとデータ型

- ・ プログラム中に書き込まれた具体的な数値や文字列をリテラルという。
- ・ 情報の種類のことをデータ型といい、`int`・`float`・`str`・`bool` が代表的である。
- ・ 数値と文字列の組み合わせでは演算できないため、型変換を要する。

変数

- ・ 変数には代入演算子を用いて情報を保存（代入）することができる。
- ・ コード中に変数名を記述すると、箱（変数）の中の値を利用（参照）できる。
- ・ 変数の命名は原則として自由だが、予約語など注意すべきルールが存在する。
- ・ 複合代入演算子を用いると、変数自身を書き換える処理を簡潔に書ける。

1.5

練習問題

練習 1-1

次の各式が評価されていく過程と結果を、図 1-2 に準じた形で表記してください。途中でエラーが発生するものは、その箇所で「エラー」と表記してください。

- (1) `2 + 10 * 5`
- (2) `'7' * (3 + 4)`
- (3) `'version {}'.format(3 + 2 * 0.1 + 9 * 0.01)`
- (4) `4 * 'num' + ' 回目の TypeError'`

練習 1-2

次の各代入文で代入された各変数に格納されているデータ型を答えてください。なお、変数 `num` には `int` 型の整数 2 が格納されているものとします。また、途中でエラーが発生するものはその旨を答えてください。

- (1) `var = 35 + num`
- (2) `num += '5'`
- (3) `GLOBAL = '世界' + str(num) + ' 箇国'`
- (4) `check_code = num * (9 / 3)`

練習 1-3

キーボードから身長 (cm) と体重 (kg) の入力を受け付け、その人の BMI を算出して表示するプログラムを作ってください。なお、BMI とは人の肥満度を表す体格指数のことで、次の式で求められます。

$$\text{BMI} = \text{体重 (kg)} \div \text{身長 (m)} \div \text{身長 (m)}$$

1.6

練習問題の解答

1
章

練習 1-1

- (1) $2 + 10 * 5 \Rightarrow 2 + 50 \Rightarrow 52$
- (2) $'7' * (3 + 4) \Rightarrow '7' * 7 \Rightarrow '7777777'$
- (3) `'version {}'.format(3 + 2 * 0.1 + 9 * 0.01)`
 \Rightarrow `'version {}'.format(3 + 0.2 + 9 * 0.01)`
 \Rightarrow `'version {}'.format(3 + 0.2 + 0.09)`
 \Rightarrow `'version {}'.format(3.2 + 0.09)`
 \Rightarrow `'version {}'.format(3.29) \Rightarrow 'version 3.29'`
- (4) $4 * 'num' + ' 回目の TypeError'$
 \Rightarrow `'numnumnumnum' + ' 回目の TypeError'`
 \Rightarrow `'numnumnumnum 回目の TypeError'`

練習 1-2

- (1) int 型
- (2) エラーとなる (int 型の変数 num に str 型の '5' を加算しようとするため)
- (3) str 型 (「global」は予約語だが、大文字は予約されていないためエラーにならない)
- (4) float 型 (9 / 3 は 3.0 という float 型になるため)

練習 1-3

```
1 h = int(input('身長 (cm) は? >>')) / 100
2 w = float(input('体重 (kg) は? >>'))
3 bmi = w / h / h
4 print('BMIは{}です'.format(bmi))
```


(別解) アンパック代入と f-string の応用構文を用いた例

```
1 h, w = int(input('身長 (cm) は? >>')) / 100, \
      float(input('体重 (kg) は? >>'))
2 print(f'BMIは{ w / h ** 2 }です')
```

※ 1 行目の最後の \ (バックスラッシュ) は、1 行目のコードが途中で折り返していることを示している。

第2章

コレクション

プログラムではさまざまなデータを扱いますが、
同じ意味を持つデータは個別に扱うより、
まとめて扱うほうが便利な場合が少なくありません。
本章では、データをまとめて扱うしくみである
コレクションの基本を学びます。

CONTENTS

- 2.1 データの集まり
- 2.2 リスト
- 2.3 ディクショナリ
- 2.4 タプルとセット
- 2.5 コレクションの応用
- 2.6 第2章のまとめ
- 2.7 練習問題
- 2.8 練習問題の解答

2.1 データの集まり

2.1.1 変数を持つ不便さ



工藤さん、前章で学んだ変数を使って、この前受けた社内試験の点数を集計するプログラムを作ってみました。我ながらいいですよ！

よくできているね。でも、もっとラクにできる方法があるよ。いい題材だから、改良できるところがないか、考えてみよう。



第 1 章では、数値や文字列などを格納して扱う変数のしくみを学びました。変数だけでもプログラムを書くことはできますが、それだけでは少し不便なこともあります。浅木さんが作成した、試験の点数を管理するプログラムで考えてみましょう(コード 2-1)。

コード 2-1 点数管理プログラム

```
1 network = 88      ネットワークは 88 点
2 database = 95     データベースは 95 点
3 security = 90     セキュリティは 90 点
4 total = network + database + security    合計を計算
5 avg = total / 3    平均を計算
6 print('合計点:{}'.format(total))
7 print('平均点:{}'.format(avg))
```

一見、問題はなさそうですが、このコードには不便なことが2つあります。

①試験科目が増えるたびに、コードに追加しなければならない

もし新しい試験科目が増えた場合には、新しい変数を準備して、さらに合計と平均の計算に書き加える必要があります。

②まとめて処理できない

たとえば、点数の高い科目から順に並べて表示するなどの処理を行いたい場合、コードが長く、複雑になってしまいます。

これらの原因は、3つの試験科目の変数を「個々の独立したデータ」として扱っていることにあります。私たち人間は、この3つの変数は「各科目の点数を格納している変数で、合計の算出や並び替えなど、1組のものとして処理することがある」と無意識に考えています。しかし、コンピュータにとって個々の変数は「何の関係もないバラバラの箱」でしかなく、1組のものとしては扱えないのです。

そこで、ほとんどのプログラミング言語では、「関連するデータをグループにして、まとめて1つの変数として扱える」しくみが用意されています(図2-1)。

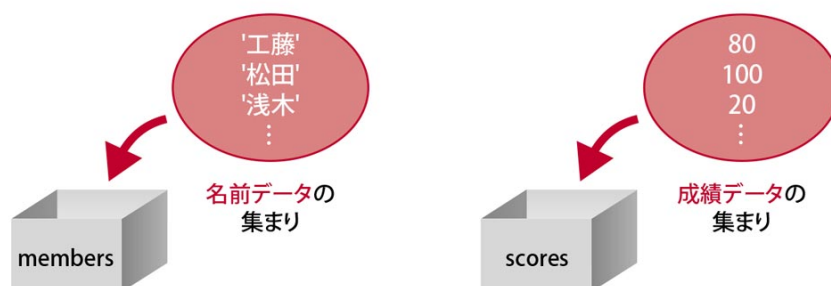


図 2-1 データをまとめて管理

このようなしくみを**データ構造** (data structure) といい、Python では**コレクション** (collection) または**コンテナ** (container) と呼びます。コレクションにはいくつかの種類があり、「リスト」「ディクショナリ」「タプル」「セット」の4つが代表的です。まずは最もよく使う「リスト」から学習していきましょう。

2.2

リスト

2.2.1 リストの特徴

リスト (list) とは、複数の値を 1 列に並べて管理するためのコレクションです。リストに格納されているそれぞれの値を**要素** (element) といい、先頭から順に**添え字** または **インデックス** (index) と呼ばれる管理番号が自動で振られます (図 2-2)。なお、添え字は **0 から始まる** ことに注意してください。

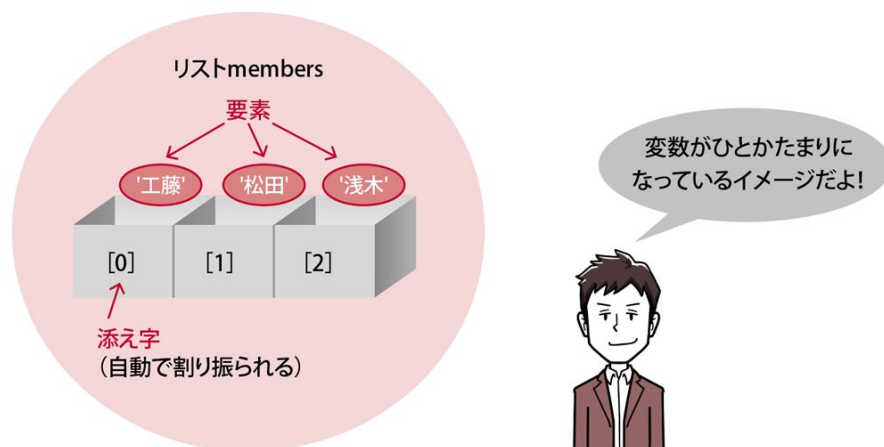


図 2-2 リスト

あるリストに値を代入したり、リストに格納されている値を参照したりしたい場合、どの要素を使用するかは添え字を使って指定します。たとえば、図 2-2 のようなリスト `members` が存在する場合、「`members` の 0 番目」と指定すれば「`工藤`」を、「`members` の 2 番目」と指定すれば「`浅木`」を参照することができます。



添え字は 0 から始まる

最初の要素を示す添え字は 0 であり、最後の添え字は要素の数より 1 つ少ない数になる。

2.2.2 リストの作成

それでは実際に、リストを体験してみましょう。コード 2-2 は本書に登場する 3 人の名前を、リストを使ってまとめたものです。

コード 2-2 リストを作成して参照

```
1 members = ['工藤', '松田', '浅木']
2 print(members)
```

リストでまとめる

リストの内容を表示する

実行結果

```
['工藤', '松田', '浅木']
```

1 行目の右辺では、`[]` (角カッコ) の中にカンマ区切りで値を並べることで、3 つの値を含んだ 1 つのリストを生み出しています。そして、「変数 = リスト」と記述することで、**リストを変数に代入**できます (図 2-3)。



リストの定義

変数 = [要素 1, 要素 2, 要素 3, ...]

※要素には、数値や文字列などを指定できる。

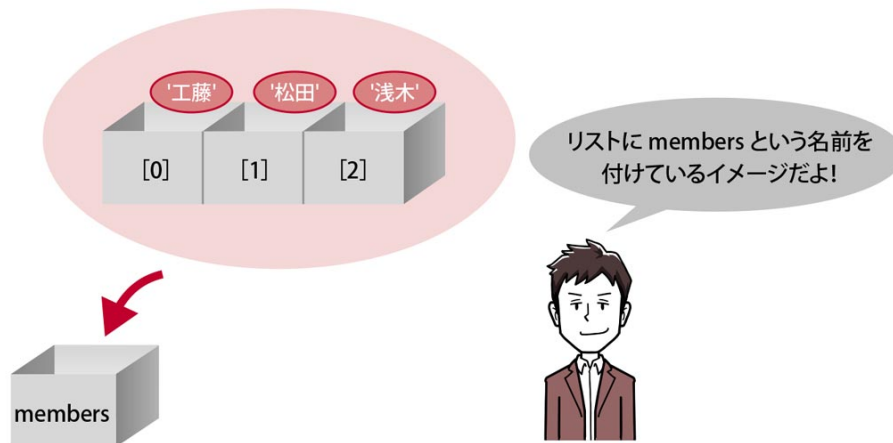


図 2-3 リストを変数に代入

なお、変数が型を持たないように (P063)、リストにも型の定めはありません。したがって、1つのリストに対して、文字列と数値など、異なるデータ型の値を格納することが可能です。

2.2.3 リストの要素を参照

リストを代入した変数を参照すると、リスト全体を使用することができます。コード 2-2 の 2 行目では変数 `members` を `print` 関数に引き渡すことで、リストに格納されているすべての要素が表示されています。

もしリスト全体ではなく、リスト内の特定の要素を参照したい場合は、変数名の直後に `[]` (角カッコ) で囲んで添え字を記述します。たとえば、先頭の要素だけを表示したい場合には、コード 2-3 のように指定します。

コード 2-3 リストの要素を参照

```
1 members = ['工藤', '松田', '浅木']
2 print(members[0])
```

0 番目の要素を参照

実行結果

工藤

2章



リストの要素を参照

リスト [添え字]

※割り振られていない添え字を指定するとエラーになる。



あれ？ 前から3つ目にある私の名前を表示しようとしたらエラーになっちゃいました(コード2-4)。

コード2-4 リストの要素を参照(エラー)

```
1 members = ['工藤', '松田', '浅木']
2 print(members[3])
```

3 番目 (4 つ目) の要素は存在しないのでエラー

実行結果

```
IndexError                                Traceback (most recent call last)
<ipython-input-7-cd7724b6c85f> in <module>()
      1 members = ['工藤', '松田', '浅木']
----> 2 print(members[3])

IndexError: list index out of range
```

浅木さんは、「前から3つ目」という意識に引っ張られて、コード2-4では添え字として3を指定してしまったようです。しかし、**添え字は0から始まる**こ

とを思い出してください(P080)。members に割り振られている添え字は 0・1・2 であって、3 を指定するとエラー (IndexError) が発生します。



慣れるまでは本当によくやっちゃうミスなんだ。気をつけよう。

2.2.4 リスト要素の合計と要素数の取得



さっそく、リストを使って得点をまとめてみました。あとは合計点と平均点を求めたいんだけど…。

合計や平均を求めるために、コード 2-5 のようなコードを思いつく人もいるでしょう。しかし、試験の数が増えていくに従って、total を求めるために足さなければならない要素が増えていくため、コーディングが大変になっていくことは想像に難くありません。

コード 2-5 試験の合計と平均を求める

```
1 # ネットワーク、データベース、セキュリティ試験の点数
2 scores = [88, 90, 95]
3 total = scores[0] + scores[1] + scores[2]
4 print('合計{}点'.format(total))
```

実行結果

合計273点

Python には、リストなどのデータの集まりに対して、合計値を簡単に求めることができる sum 関数という命令が準備されています。



sum 関数

sum(リスト)

※リストのすべての要素を合計した値に置き換わる。

※文字列を格納しているリストには使えない。

※後述するタプルやセットに対しても使用できる。

sum 関数を用いれば、コード 2-5 はコード 2-6 のように書き換えることができます。

コード 2-6 sum 関数を用いて合計を求める

```
1 scores = [88, 90, 95]
2 total = sum(scores)
3 print('合計{}点'.format(total))
```

リスト内の全要素の合計を求める

実行結果

合計273点



これなら試験が増えても大丈夫ですね。

合計の sum があるなら平均の average もあったりして…。



残念ながら average という関数はないんだ。でも、工夫すれば簡単に平均を求めることができるよ。

リストの平均値は、「要素の合計」を「要素の個数」で割れば求めることができます。前者は `sum` 関数で求めますが、後者は `len` 関数で求めることができます。



len 関数

len(リスト)

※リストの要素数に置き換わる。

※後述するディクショナリ、タプル、セットに対しても使用できる。

それでは、`sum` 関数と `len` 関数を組み合わせて、リストの合計値と平均値を出すプログラムを完成させましょう（コード 2-7）。

コード 2-7 リストの合計値と平均値を求める

```
1 scores = [88, 90, 95]
2 total = sum(scores)
3 avg = total / len(scores)
4 print('合計{}点、平均{}点'.format(total, avg))
```

リストの要素数を求める

実行結果

合計273点、平均91.0点

2.2.5 リスト要素の追加・削除・変更



次はリストの要素を操作してみよう。まずは追加からだ。

一度定義したリストに要素を追加することができます。コード 2-8 を見てみましょう。

コード 2-8 リストに要素を追加

2章

```
1 members = ['工藤', '松田', '浅木']
2 members.append('菅原')
3 members.append('湊')
4 members.append('朝香')
5 print(members)
```

リストに要素を追加

実行結果

```
['工藤', '松田', '浅木', '菅原', '湊', '朝香']
```

`append` 関数を使用することで、指定した要素をリストに追加できます。追加される位置はリストの末尾です。なお、この関数は、「リスト.append()」の形で使用するタイプの関数ですので注意しましょう (P071)。



リストの末尾に値を追加

リスト .append(リストに追加したい値)

※指定した値が、リストの末尾に新たな要素として追加される。

また、`remove` 関数を使うと指定した要素をリスト内から削除することができます (コード 2-9)。この関数も「リスト.remove()」の形で使います。



リストから指定した値を削除

リスト `.remove()` (リストから削除したい値)

※削除した要素の後ろの要素は前に詰められる。

コード 2-9 リストから要素を削除

```
1 members = ['工藤', '松田', '浅木']
2 members.remove('松田')
3 print(members)
```

リストから要素を削除

実行結果

['工藤', '浅木']



何で僕を削除しちゃうんですか。

松田のいた位置には私が詰めておくから安心してよね。ところで、追加と削除ができるなら、変更もできるんですか？



もちろん!!

リスト内の特定の要素の内容を変更するには、添え字を指定して代入します(コード 2-10)。

コード 2-10 リストの要素を変更

```
1 members = ['工藤', '松田', '浅木']  
2 members[0] = '菅原' ) リストの要素を変更  
3 print(members)
```

実行結果

```
['菅原', '松田', '浅木']
```



リストの要素を変更

リスト[変更要素の添え字] = 変更後の値



こらこら、僕を書き換えるんじゃない！ 菅原さんのほうがいいなら、今からでも交代するぞ。

冗談です。工藤さんのほうがいいです!! たぶん。



2.2.6

高度な要素の指定



リストの基本的な使い方については、こんなもんでいいだろう。あとは、うーん…まあ一応紹介しておくか。

この項では、知っておくと便利な要素の指定方法を2つ紹介しておきましょう。

なお、ここで紹介する内容は、入門段階からマスターすることが必須というわけではありません。余裕のある人はぜひチャレンジしてみてください。

①スライスによる範囲指定

リストで要素を指定する際、**スライス** (slice) という構文を用いることで、連続した範囲にある要素を参照することができます (コード 2-11)。



スライスによる範囲指定

リスト変数 [A:B]

※添え字が A 以上 B 未満の要素を参照する部分リストに評価される。

※「A:」と指定すると、添え字が A 以上のすべての要素を参照する。

※「:B」と指定すると、添え字が B 未満のすべての要素を参照する。

※「:」だけを指定すると、すべての要素を参照する。

コード 2-11 スライスによる範囲指定

```
1 a = [10, 20, 30, 40, 50]
2 print(a[1:3]) # 添え字が1以上3未満の要素
3 print(a[2:]) # 添え字が2以上のすべての要素
4 print(a[:3]) # 添え字が3未満のすべての要素
```

実行結果

```
[20, 30]
[30, 40, 50]
[10, 20, 30]
```

②負の数による指定

リストの要素は、先頭からだけでなく、末尾からの順番で指定することもできます(図 2-4)。リストの末尾の要素は添え字「-1」で参照し、その1つ前の要素は「-2」で参照します(コード 2-12)。

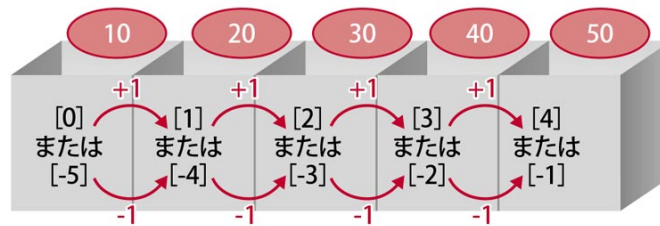


図 2-4 負の数による指定

コード 2-12 負の数による指定

```
1 a = [10, 20, 30, 40, 50]
2 print(a[-1])    # 末尾の要素を参照
3 print(a[-2])    # 末尾から2番目の要素を参照
```

実行結果

50

40

2.3 ディクショナリ

2.3.1 ディクショナリの特徴



どうしたんだい。松田くん。難しい顔をして。

さっきの試験得点をまとめたリストなんですけど…。あれだと、各要素が何の試験の得点か、わからなくなりますか。



それぞれの要素に意味を持たせたいんだね。それならディクショナリを使うといいよ。

リストは 0 から始まる整数、つまり添え字で要素を管理します。基本的に、Python が自動で添え字を割り振るなどの管理してくれるので、開発者は格納するデータだけを意識すればよいというメリットがあります。その一方で、「0 番目が何のデータなのか?」「使いたいデータは何番目にあるのか?」というような、各要素のデータが持つ具体的な意味合いがわかりにくくなってしまうという懸念もあります。

このような悩みを解決するためには、**ディクショナリ** (dictionary) というもう 1 つのコレクションでデータを管理しましょう。ディクショナリもリストと同様に、複数のデータをひとまとまりに集めて管理するための道具という意味ではよく似ています。しかし、ディクショナリはそれぞれの要素に対して**キー** (key) という見出し情報を付けることができます (図 2-5)。

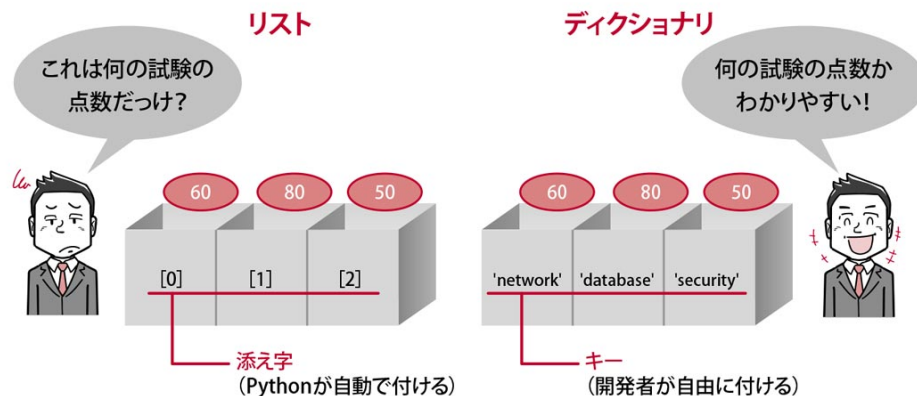


図 2-5 リストとディクショナリの比較

2.3.2 ディクショナリの作成

では、ディクショナリを使って試験の点数をまとめてみましょう(コード 2-13)。

コード 2-13 ディクショナリの作成

```
1 scores = {'network':60, 'database':80, 'security':50}
2 print(scores)
```

ディクショナリを作成

ディクショナリの全要素を表示

実行結果

```
{'network': 60, 'database': 80, 'security': 50}
```

1 行目の右辺で、**{}**(波カッコ)によってディクショナリを生み出しています。リストの生成とは用いるカッコの種類が異なるので注意しましょう。また、ディクショナリに格納する各要素を、「キー: 値」の形式で指定している点がポイントです。今回の場合、試験名をキーにすることで、それぞれのデータが何の試験の点数かがわかるようにしています



ディクショナリの定義

変数 = { キー 1: 値 1, キー 2: 値 2, ... }

※キーと値の対応を、:(コロン)を使って指定する。

なお、各要素に付けるキーは、次のようなルールに従って開発者が自由に決めることができます。

- キーには、文字列のほか数値型など、さまざまな型のデータを指定できる。
- キーのデータ型は要素ごとに異なってもよい。
- キーの重複も許される(ただし、最後に指定したもの以外は無視される)。



キーが重なってもエラーにはならないけれど、基本的にキーは「重複不可」と考えて利用することをお勧めするよ。

2.3.3 ディクショナリ要素の参照

ディクショナリの要素を指定するには、要素に設定したキーを使用します(コード 2-14)。

コード 2-14 ディクショナリの要素を参照

```
1 scores = {'network':60, 'database':80, 'security':50}
2 print(scores['database'])
```

ディクショナリの要素を指定

実行結果

80

リストよりも直感的でわかりやすいですが、正しい要素を参照するためには、各要素のキーをしっかりと把握しておく必要がありますので注意しましょう。



キーを1文字でも間違えるとエラー (KeyError) が発生してしまうから、スペルミスにも注意しよう。

2章



ディクショナリの要素を指定

ディクショナリ[キー]

※キーは大文字／小文字を区別する。

2.3.4

ディクショナリ要素の追加と変更



ディクショナリの構造が理解できたら、要素を操作してみよう。

松田くんが60点未満で不合格となったセキュリティの試験を再受験し、かつ、新たにプログラミングの試験も受験してきたとしましょう。その場合、次のようにディクショナリ要素の追加と変更を行います(コード 2-15)。

コード 2-15 ディクショナリの要素の追加と変更

```
1 scores = {'network':60, 'database':80, 'security':50}
2 scores['programming'] = 65
3 scores['security'] = 55
4 print(scores)
```

ディクショナリの要素を追加

ディクショナリの要素を変更

実行結果

```
{'network': 60, 'database': 80, 'security': 55, 'programming': 65}
```



追加も変更も同じ書き方なんですね。

指定したキーが、ディクショナリ内の要素にすでに使用されていた場合は変更、使用されていない場合は追加になります。このように、ディクショナリの追加と変更はまったく同じ構文を使います。キーをしっかり把握しておかないと、追加したつもりが変更になってしまうことがあるので注意しましょう。



ディクショナリの要素を追加

ディクショナリ[新しいキー] = 新しい値



ディクショナリの既存の要素を変更

ディクショナリ[変更したい要素のキー] = 変更後の値

2.3.5

ディクショナリ要素の削除



うーん。合格する自信がないから、セキュリティの点数は削除したいなあ。

ディクショナリの要素を削除するには、**del 文**という特殊な構文を使います(コード 2-16)。

コード 2-16 ディクショナリの要素を削除

```
1 scores = {'network':60, 'database':80, 'security':55}
2 del scores['security']
3 print(scores)
```

ディクショナリの要素を削除

実行結果

```
{'network': 60, 'database': 80}
```



ディクショナリの要素を削除

```
del ディクショナリ [ 削除したい要素のキー ]
```

2.3.6

ディクショナリとリストの比較



何だか、ディクショナリってリストより便利ですね。これなら、リストなんていらんんじゃないですか？

そんなことはない。リストのほうが便利なこともあるよ。



ディクショナリには人間にとって意味のあるキーを付けられるため、使い勝手がよいと感じる人も少なくないでしょう。一方のリストは、たとえば0番目がどのようなデータなのかわからないというデメリットがありますが、裏を返せば、**0番目がどのようなデータなのか考える必要がない**とも言えます。いちいち自分で要素にキーを振らなくてもよいため、どのようなキーを振ったか覚えておく必

要もないのです。

たとえば、無作為に抽出した 10,000 件の家賃データの合計や平均を求めるような場面を想像してみましょう。このような場合、10,000 個のデータさえあれば目的を達成することができるので、リストで十分なのです。もし、ディクショナリに格納するとなると、必要のないキーを 10,000 個ものデータに設定しなければならず、大変な手間となります。



うへえ。10,000 個もキーを考えるくらいなら、単純に 1 から順に振っちゃいそうです…。

ははは。それならリストの添え字とほぼ同じになるね。



また、リストの添え字は整数なので、0 番目の次は 1 番目というように、それぞれの要素には順序があります。一方、ディクショナリのキーは任意に振ることができるため、データの順序を表すことができません。ディクショナリ内の各要素には順序はなく、追加した順にデータが入っているという保証はありません(詳細は次ページの Column を参照)。



リストとディクショナリの使いどころ

- 複数のデータを単にまとめて管理したい場合や、順序を持つ複数のデータを管理したい場合にはリストを使う。
- 順序を持たない複数のデータに見出しを付けて管理したい場合には、ディクショナリを使う。



データの性質に合ったまとめ方を選択しないとイケないのね。



Column

ディクショナリ要素の順序

本書では、ディクショナリの特徴として順序を持たないことを紹介しましたが、2018年6月にリリースされた Python 3.7 では順序性が保証されたため、追加した順に要素を取り出すことが可能になりました。しかし、Python 3.6 以前は順序性が保証されないため、基本的にはディクショナリの要素は順序を持たないと覚えておきましょう。



Column

ディクショナリの合計

リストと違い、ディクショナリの場合は `sum` 関数を使用して格納している値の合計を求めることができません。ディクショナリで合計を求めるには、次のように記述します。

```
scores = {'network':60, 'database':80, 'security':50}
total = sum(scores.values())
print(total)
```

「ディクショナリ.values()」と記述することで、ディクショナリのキーを除いた、値だけからなる集まり (60、80、50) を取得することができます。この集まりは、本章で紹介するリスト、セット、タプルとも違う、また別の種類のコレクションですが、`sum` 関数を使用することができます。

2.4

タプルとセット



Python には、リストやディクショナリ以外にもデータをまとめる方法があるんだ。

いっぱいあるんですね…。全部覚えるのは大変そう。



大丈夫。これから教えるタプルやセットは、利用頻度は低いからポイントだけ押さえていこう。いざ使おうというときに、おさらいしてくれたらいいよ。

2.4.1 タプル

タプル (tuple) とは、リストとほぼ同じ特徴を持つコレクションです。唯一、「要素の追加、変更、削除ができない」という点だけが異なります。概ね「中身が変更できないリスト」と考えて差し支えありません。構文とコード 2-17 を見ながら、タプルの特徴を確認しましょう。



タプルの定義

変数 = (値 1, 値 2, 値 3, …)

※要素を追加、変更、削除することはできない。

コード 2-17 タプルの利用

```
1 scores = (70, 80, 55)
2 print(scores)
3 print(scores[0])
4 print('要素数は{}'.format(len(scores)))
5 print('合計は{}'.format(sum(scores)))
```

タプルの要素を添え字で指定

実行結果

```
(70, 80, 55)
70
要素数は3
合計は205
```

タプルはリストと違い、`()` (丸カッコ) で定義します。リストと同様に、各要素には、0 から始まる添え字が自動で設定されますが、定義後に要素を変更したり、追加したり、削除したりすることはできません。実際、0 番目の要素を変更しようとする、コード 2-18 のようにエラーとなります。

コード 2-18 タプルの要素を変更 (エラー)

```
1 scores = (70, 80, 55)
2 scores[0] = 80
```

タプルの要素を変更

実行結果

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-16-b6466b1d25ff> in <module>
      1 scores = (70, 80, 55)
```

```
----> 2 scores[0] = 80
```

```
TypeError: 'tuple' object does not support item assignment
```



うわっ！ 本当にエラーになった。要素が変更できないコレクションを使うメリットってあるんですか？

「データを変更できない」という不便さは、「**データが書き換えられていないことを保証できる**」という**メリットと表裏一体**です。たとえば本格的に業務で使用するプログラムともなると、ソースコードは数百～数千行に及ぶのは珍しくありません。

当然、プログラムは複数の人間によって手分けして開発することになりますが、開発者の数が増えるほど、データをうっかり書き換えてしまうようなミスを誰かが犯すリスクは増していきます。そのような場合、データをリストではなくタプルで管理することで、意図しない書き換えによるリスクを抑えることができるのです。



リストとタプルの使いどころ

- ・書き換える可能性のある複数のデータを単にまとめて管理したい場合は、リストを使う。
- ・書き換える可能性のない複数のデータを単にまとめて管理したい場合は、タプルを使う。

なお、非常によく似ているリストとタプルは、**シーケンス** (sequence) と総称されることもあります。シーケンスに対しては `sum` 関数・`len` 関数などが共通して使えるほか、スライス (P090) を使って部分シーケンスを取り出したり、`+` 演算子を使ってシーケンス同士を連結したりすることもできます。



まあ、「書き換えられずに安全なリスト」ってことですね。タプルは楽勝！

…と言いたいところだが、タプルには特有の使用上の注意点が1つあるんだ。



2章

コレクションではまれに、要素数が1となるようなデータを扱うことがあります。たとえば、コード 2-3 (P082) でメンバーがまだ自分ひとりしかいない場合や、コード 2-13 (P093) でまだ1科目しか試験を受けていない場合では、コード 2-19 のようなコードになるでしょう。

コード 2-19 要素数が1つのリストとディクショナリ

```
1 members = ['松田']      # 要素数1のリスト
2 scores = {'network':82} # 要素数1のディクショナリ
```

1 行目のリスト members と同様のことをタプルでもやってみましょう (コード 2-20)。

コード 2-20 要素数1のタプル(のつもり)

```
1 members = ('松田')      # 要素数1のタプルを定義 (したつもり)
2 print(type(members))    membersの型を調べる
```

実行結果

```
<class 'str'>
```



文字列になっちゃってる…。何でだろう…。

コード 2-20 の 1 行目をよく眺めてみると、「文字列 '松田' を、優先順位を引き上げるための丸カッコ (P043) で囲んでいる」とも読めます。よって、この式は「members = '松田'」と同じものと解釈され、Python は members に文字列を代入してしまいます。

要素数 1 のタプルを生成するには、タプル定義に記述する値の後ろにカンマを付けます (コード 2-21)。

コード 2-21 要素数 1 のタプル

```
1 members = ('松田', )      # 要素数1のタプルの正しい定義
2 print(type(members))
```

実行結果

```
<class 'tuple'>
```

実は、Python の言語仕様では、タプルを生成する記号は丸カッコではなく、カンマであると定められています。つまり、丸カッコを省略して、「members = '松田', '浅木'」のように記述しても、members というタプルが作られるのです。



でも、丸カッコの省略は、何をしているコードなのか紛らわしいからお勧めはしないよ。

2.4.2 セット

最後に紹介する **セット** (set) もリストと似たコレクションですが、次の 4 つの点でリストとは異なります。

- ① 重複した値を格納することができない。
- ② 添え字やキーという概念がなく、特定の要素に対して代入・参照する方法が存在しない。
- ③ 添え字がないため、要素は順序を持たない。
- ④ `append` 関数の代わりに `add` 関数で要素を追加する。



添え字もキーもないなんて…。これいったい何のために使うんですか？

たとえば、「信号の色を挙げてください」と聞かれたら、みなさんはどう答えますか。「赤・黄・青」と答える人がいるかもしれませんが、「赤・青・黄」と答える人もいるかもしれませんが、どちらも正解です。挙げる順序には意味がないからです。また、信号に同じ色は2回以上登場しませんから、「赤・赤・黄・青」と答える人もいないでしょう。

このように、あるものの「種類」をデータとして管理するような場合に、セットは活躍します。



セットは種類を管理する

セットは順序を持たず、その要素は重複しないため、「種類」の管理に向いている。



セットはディクショナリと同じ、波カッコで生み出すんだ。もちろんキーは書かないよ。



セットの定義

変数 = { 値 1, 値 2, ... }

※重複する値は取り除かれる。

それでは、コード 2-22 を実行してセットの特徴を確認していきましょう。

コード 2-22 セットの利用

```
1 scores = {70, 80, 55, 80}
2 scores.add(80);
3 print(scores)
4 print('要素数は{}'.format(len(scores)))
5 print('合計は{}'.format(sum(scores)))
```

実行結果

{80, 70, 55}

要素数は3

合計は205



この結果からわかるセットの特徴を順に解説していくよ。

まず、セットは重複する値を格納できないという特徴がありました(特徴①)。そのため、1 行目や 2 行目で 80 を重複して格納することを試みていますが、表示結果を見てわかるように 80 は 1 つしか格納されていません。

また、添え字やキーが存在しない(特徴②)ため、たとえば「0 番目の要素を表示する」などのように、要素の指定はできません。しかし、3 行目のように全要

素を丸ごと表示したり、5行目のように全要素の合計値を求めたりすることはできます。

さらに、セットの要素には順序がありません(特徴③)。そのため、1行目で記述した順番で格納される保証はなく、事実、3行目の実行結果ではそれとは異なる順で表示されています。

最後に、セットへの値の追加には、2行目のように `add` 関数を使います(特徴④)。リストの場合は要素に順序があるため、「末尾に加える」という意味を持つ `append()` という関数名でした。セットには末尾がないため、単に「加える」という意味の `add()` という関数名になっています。



Column

コレクションたちの別名

本章で紹介した各コレクションは、本書で紹介したものとは別の名前と呼ばれることもあります。

表 2-1 コレクションの別名

本書で紹介したコレクション	別名	説明
リスト	配列 (array)	ほかのプログラミング言語では「配列」が多い
ディクショナリ	辞書、マップ (map)	マップは対応表という意味
セット	集合	集合も順序がないことを示す
タプル	—	別名は特になし

2.5

コレクションの応用



お疲れさま。ここまでで、主要な 4 つのコレクションは、基本的な使い方ができるようになったはずだよ。最後にいくつか、応用テクニックを紹介しておこう。

2.5.1

コレクションの相互変換

この章を通して、私たちは複数のデータをひとまとまりにして取り扱う方法を学びました。特に、そのデータの性質や使い方に合わせて、4 つのコレクションを使い分けることが重要でした。

本格的なプログラムの開発では、コレクションの使い分けもさることながら、プログラムの実行中に違う種類のコレクションへと変換する必要に迫られる場面も多く見られます。

そこで、Python では、各コレクション間での相互変換を可能にするため、表 2-2 のような関数を準備しています。いずれも、「関数名(～)」の形で呼び出すことができ、引き渡したコレクションを目的のコレクションに変換してくれます(コード 2-23)。

表 2-2 リスト・タプル・セットへの変換

関数	説明
list 関数	渡されたものをリストに変換する ^{※1}
tuple 関数	渡されたものをタプルに変換する ^{※1}
set 関数	渡されたものをセットに変換する ^{※2}

※1 もともと順序関係がないディクショナリやセットからは、変換の際にどのような順序で要素が並べられるかは保証されない。

※2 もともと重複が許されるリストやタプルからは、変換の際に重複が排除される。

※3 「list()」のように、何も指定せずにこれらの関数を呼び出すと、空のコレクションを作成することができる。

なお、表 2-2 の関数にディクショナリを渡すと、キーだけが使われます。ディクショナリの値だけを使ってリストやタプルに変換したい場合は、「ディクショナリ.values()」(P099)の結果を関数に渡すようにしてください。

2章

コード 2-23 コレクションの相互変換

```
1 scores = {'network':60, 'database':80, 'security':60}
2 members = ['松田', '浅木', '工藤']
3 print(tuple(members))    # リストmembersをタプルに変換して表示
4 print(list(scores))      # scoresのキーをリストに変換して表示
5 print(set(scores.values())) # scoresの値をセットに変換して表示
```

実行結果

```
('松田', '浅木', '工藤')
['network', 'database', 'security']
{80, 60}
```



Column

ディクショナリへの変換

ディクショナリへの変換は少し複雑です。ディクショナリはキーと値の2種類の情報をペアで管理するコレクションであるため、単一のリストやセットからは変換できません。「キーを格納したリスト」と「値を格納したリスト」の2つが存在する場合、「dict(zip(キーのリスト, 値のリスト))」という構文で1つのディクショナリに変換できます。

2.5.2 コレクションのネスト

これまで、コレクションには、文字列や数値のデータを格納してきました。しかし、コレクションの中に別のコレクションを格納することもできます。たとえば、メンバーごとの試験結果を管理したい場合、コード 2-24 のように、ディクショナリの中にディクショナリを入れた二重構造で管理することができます。

コード 2-24 ディクショナリの中にディクショナリをネスト

```
1 matsuda_scores = {'network':60, 'database':80, 'security':50}
2 asagi_scores = {'network':80, 'database':75, 'security':92}
3 member_scores = {
4     '松田': matsuda_scores,
5     '浅木': asagi_scores
6 }
```

メンバーごとの趣味一覧を管理したいなら、ディクショナリの中にセットを入れた二重構造が便利です (コード 2-25)。

コード 2-25 ディクショナリの中にセットをネスト

```
1 member_hobbies = {
2     '松田': {'SNS', '麻雀', '自転車'},
3     '浅木': {'麻雀', '食べ歩き', '数学', '数学', '数学'}
4 }
5 # 全員の趣味一覧を表示する
6 print(member_hobbies)
7 # 松田くんの趣味一覧を表示する
```

数学は 1 つのみ登録される

```
8 print(member_hobbies['松田'])
9 # 浅木さんの趣味一覧を表示する
10 print(member_hobbies['浅木'])
```

実行結果

```
{'松田': {'自転車', 'SNS', '麻雀'}, '浅木': {'麻雀', '食べ歩き', '数
学'}}
{'自転車', 'SNS', '麻雀'}
{'麻雀', '食べ歩き', '数学'}
```

また、リストの中にリストを入れた構造は特に **2次元リスト**ともいわれ、表の構造を持つデータ管理などに使われます。コード 2-26 と併せて、図 2-6 のような構造をイメージしてみてください。

コード 2-26 2次元リストの例

```
1 a = [1, 2, 3]
2 b = [4, 5, 6]
3 c = [a, b]      # aを0番目、bを1番目とする2次元リストcを定義
4
5 print(c)        # リストc全体を参照
6 print(c[0])     # リストcの0番目（リストa）だけを参照
7 print(c[1][2])  # リストcの1番目（リストb）の2番目だけを参照
```

実行結果

```
[[1, 2, 3], [4, 5, 6]]
[1, 2, 3]
6
```

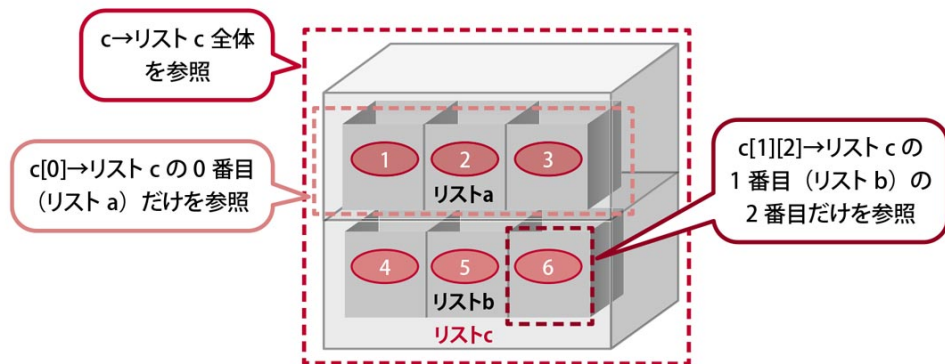



図 2-6 2次元リストの構造

二重構造だけではなく、三重構造以上のコレクションも作ることができます。このような多重構造のことを**ネスト**や**入れ子**といいます。本格的なデータ分析プログラムでは、ネストしたコレクションの活用は欠かせません。



ちなみに、ディクショナリのキーには、タプル以外の3つのコレクションは使えないというルールがある。そもそもタプルを使うことは多くないだろうから、「キーにコレクションは使わない」と覚えておけばいいだろう。

2.5.3

集合演算



最後に紹介するのは、「セット」だけに与えられた特別な機能だよ。

順序関係がなく、重複も許されず、添え字もキーもないから要素を指定して値を出し入れできない。そのような特徴を持つセットは、「ほかのコレクションに比べて、イマイチ使えないヤツだ」と考えてしまう人もいるでしょう。

確かに、特有の不便さもあり、一般的にリストやディクショナリほどの頻度では用いられないセットですが、その特徴ゆえに与えられた特別な機能が**集合演算**です。



しゅうごうえんぜん…？ 難しいっていう予感しかないんですけど…。

なーに、たいしたことはないよ。な、浅木さん？



2章

集合演算とは、ある2つのセットの「共通点」や「違い」を探す処理のことです。たとえば、コード 2-25 (P110) では、松田くんの趣味を格納したセット (member_hobbies['松田']) と浅木さんの趣味を格納したセット (member_hobbies['浅木']) が登場しました。もしこれらのセットの共通項がわかれば、2人の相性もわかると思いませんか。

Python では、2つのセットの共通項を求めるための道具として、「& 演算子」を準備しています。構文を確認して、2人の趣味の共通点を探してみましょう (コード 2-27)。



セットの & 演算

セット 1 & セット 2

※セット 1 とセット 2 の両方に含まれる要素からなるセットに評価される。

コード 2-27 セットの & 演算

```
1 member_hobbies = {
2     '松田': {'SNS', '麻雀', '自転車'},
3     '浅木': {'麻雀', '食べ歩き', '数学', '数学', '数学'}
4 }
5 common_hobbies = member_hobbies['松田'] & member_hobbies['浅木']
```

```
6 print(common_hobbies) # 2人に共通する趣味一覧を表示する
```

実行結果

```
{'麻雀'}
```



浅木先輩も麻雀が好きだったんですね！ 今度一緒に卓を囲みましょう！！

いいけど、私、勝つまでやめないからね♥



...

2つのセットの共通項として求められたセットのことを、数学の世界では「積集合」といいます。数学の世界には、ほかにも図 2-7 に示すようなセット同士の演算方法が存在しますが、Python でもそれぞれの集合演算子を用いて簡単に実現することができます。

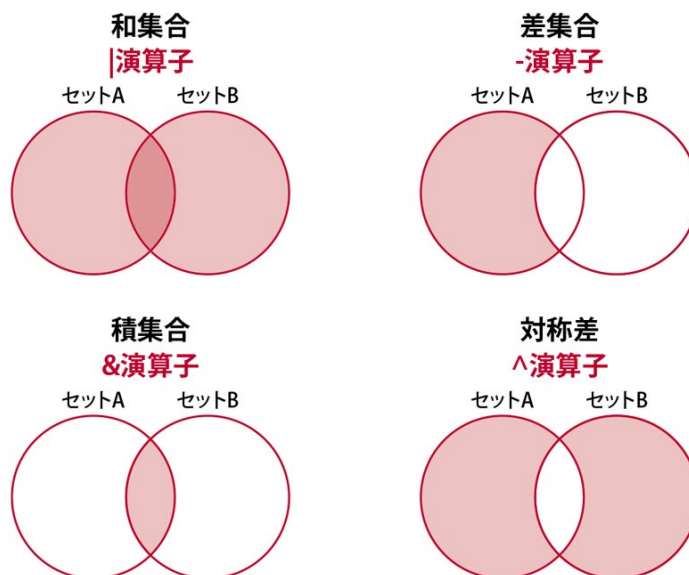


図 2-7 集合演算の種類

4つの集合演算が動作する様子を実際のコードで確認してみましょう(コード2-28)。

コード 2-28 4つの集合演算

2章

```
1 A = {1, 2, 3, 4}
2 B = {2, 3, 4, 5}
3 print(A | B)
4 print(A & B)
5 print(A - B)
6 print(A ^ B)
```

和集合

積集合

差集合

対称差

実行結果

{1, 2, 3, 4, 5}

{2, 3, 4}

{1}

{1, 5}



集合演算はデータ分析に役立つよ。興味がある人はぜひマスターしておこう！

2.6

第 2 章のまとめ

この章では、次のことを学びました。

コレクション

- 複数の値をまとめて扱うコレクションというしくみがある。
- 代表的なコレクションとして、4 つの種類があり、データの特性や管理目的に合わせて適切なコレクションを用いる必要がある。
- コレクションには数値や文字列を混在して格納することができる。
- コレクションは、変数と同じく決まったデータ型を持たない。
- リストやタプルで用いる添え字は 0 から始まる。

特徴	リスト	ディクショナリ	タプル	セット
格納内容	複数の値	複数の「キーと値のペア」	複数の値	複数の値
定義方法	[値 1, 値 2, ...]	{ キー 1: 値 1, キー 2: 値 2, ... }	(値 1, 値 2, ...)	{ 値 1, 値 2, ... }
個別要素の参照	変数名 [添え字]	変数名 [キー]	変数名 [添え字]	困難 [※]
要素の追加	変数名.append(値)	変数名 [キー] = 値	不可	変数名.add(値)
要素の変更	変数名 [添え字] = 値	変数名 [キー] = 値	不可	困難 [※]
要素の削除	変数名.remove(値)	del 変数名 [キー]	不可	変数名.remove(値)
要素の順序関係	あり	なし	あり	なし
重複値の格納	可	キーは不可 値は可	可	不可
集合演算	不可	不可	不可	可
len 関数による 要素数の取得	可	可	可	可
sum 関数による 合計の取得	可	不可	可	可
スライスの利用	可	不可	可	不可
+ 演算子による 連結	可	不可	可	不可

※キーや添え字を用いた操作は行えない。第 4 章で紹介する繰り返しによる操作は可能。

2.7

練習問題

2
章

練習 2-1

次の各要件のために用いるコレクションとして、一般的に最も妥当と思われるものを、リスト、セット、ディクショナリから選んでください。ただし、コレクションはネストせず 1 つのみを利用するものとします。

- (1) 47 都道府県の「都道府県名と人口」
- (2) 解析に用いるための過去 28 日間分の「1 日あたりの Web サイトアクセス数」
- (3) 「北」や「南」といった 4 つの方角
- (4) この世に存在する「メジャーなプログラミング言語の名称」(Python や Ruby など)
- (5) ある航空機の 200 座席の予約状態 (0 なら空き、1 なら予約済み)

練習 2-2

キーボードから国語・算数・理科・社会・英語の 5 つの試験結果の点数を入力させ、その合計点と平均点を表示するプログラムを作成してください。なお、各教科の点数については表示する必要はありません。

練習 2-3

次のような仕様の相性診断プログラムを作成してください。

- (1) 1 人目の趣味を 5 つ格納したセットを準備する。
- (2) 2 人目の趣味を 5 つ格納したセットを準備する。
- (3) 「心の準備ができたなら Enter キーを押してください」と表示して入力を待つ。
- (4) 2 人の趣味の「積集合の要素数 ÷ 和集合の要素数」を計算し、0 ~ 100% の「相性度」として表示する。

2.8

練習問題の解答

練習 2-1

- (1)ディクショナリ (2)リスト (3)セット
(4)セット (5)ディクショナリ(座席番号をキーとする)

練習 2-2

```
1 scores = []
2 scores.append(int(input('国語の点数 >>')))
3 scores.append(int(input('算数の点数 >>')))
4 scores.append(int(input('理科の点数 >>')))
5 scores.append(int(input('社会の点数 >>')))
6 scores.append(int(input('英語の点数 >>')))
7 print(f'合計 {sum(scores)}点 / 平均 {sum(scores) / len(scores)}点')
```

練習 2-3

```
1 player1 = {'読書', '昼寝', '映画鑑賞', '散歩', '料理'}
2 player2 = {'テニス', '将棋', '料理', '読書', '旅行'}
3 input('心の準備ができたならEnterキーを押してください')
4 common = player1 & player2
5 total = player1 | player2
6 compatibility_rate = len(common) / len(total) * 100
7 print(f'相性度は{compatibility_rate}パーセントでした')
```

※ 1 行目と 2 行目でセットに格納した趣味は一例である。

第3章

条件分岐

私たちは日常生活の中で、もし晴れていたら外出し、雨が降っていたら家で寛ぐというように、さまざまな条件に基づいて行動しています。プログラムでも同様に、状況に応じて実行する処理を変えることができます。この章では、条件に基づいて処理の流れを制御する方法について学びましょう。

CONTENTS

- 3.1 プログラムの流れ
- 3.2 条件分岐の基本構造
- 3.3 条件式
- 3.4 分岐構文のバリエーション
- 3.5 第3章のまとめ
- 3.6 練習問題
- 3.7 練習問題の解答

3.1 プログラムの流れ

3.1.1 文と制御構造

これまでの章では、変数や演算子を使った計算や、`print()` などの関数を使った命令、コレクションの操作などを紹介してきました。そして、Python でそれらの処理を実行するには、図 3-1 のように 1 行ごとに記述するのでしたね。

```
name = ' 松田 '
year = 2019 - 23
print(name + ' くんは ' + str(year) + ' 年生まれ ')
```

1行に処理を1つ書く

1行=1つの実行単位=文

図 3-1 1 行に 1 つの処理を書く

Python では、このような 1 行の実行単位を文(statement)といいます。これまで登場した文はすべて、プログラムの上から順に 1 行ずつ実行されていました。実は、条件によって実行する文を変えたり、同じ文を繰り返し実行したりすることによって、実行順序を変えることができます(図 3-2)。

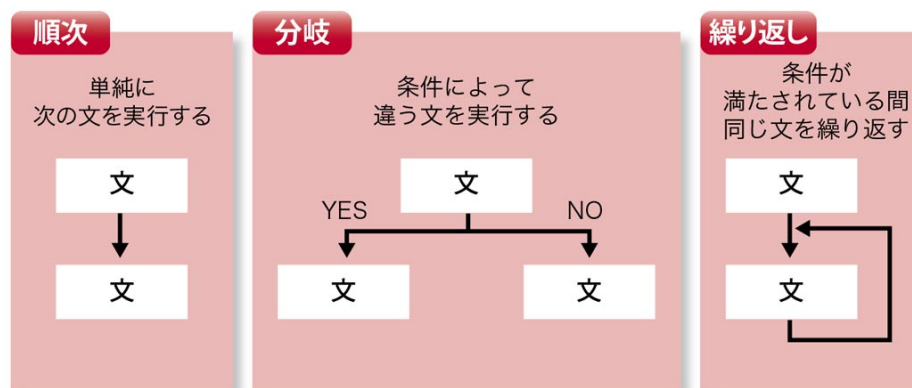


図 3-2 代表的な制御構造



これまで上から順に1行ずつ実行されていたのは、順次という構造だったからなんですね。

そのとおり。特に何も指定しなければ、プログラムは順次の流れによって実行されていくんだ。



このように、文の実行順をコントロールするプログラムの構成を**制御構造**といい、代表的な構造が図3-2に挙げた**順次・分岐・繰り返し(ループ)**です。



実は、この3つだけであらゆるプログラムが作成できるんだよ。

なるほど！ プログラムをマスターするには、順次・分岐・繰り返しを覚えれば何とかなるってことですね！



この3つの制御構造を組み合わせることで、文の実行の流れを自由にコントロールでき、しかも、ありとあらゆるプログラムの作成が可能であることが研究で明らかになっています。これを**構造化定理**といいます。



構造化定理

順次・分岐・繰り返しの3つの制御構造を組み合わせることで、どんなに複雑なプログラムでも作成できることが理論上可能である。



Column

1 行≠ 1 つの文とならない書き方

本文では、1 行が実行の単位であり、1 つの文であると紹介しましたが、文の終わりに ; (セミコロン) を入れることによって、1 行に複数の文を記述することもできます。

```
name = '松田'; print('僕の名前は{ }です'.format(name))
```

また、1 行を複数行に分けて記述するには、\ (バックスラッシュ) を行末に記述します。

```
1  n = 1 + 2 \  
2  + 3  
3  print(f'答えは{n}\  
4  改行しても足し算できていますね')
```

単純な処理で使う場合には全体のボリュームが下がるなどのメリットもありますが、あまり多用しすぎるとコードが読みにくくなり、バグの温床ともなりかねません。本書では、紙面の都合で使うこともありますが、原則、特に自分が入門レベルであると自覚する間は、用いないようにしましょう。

3.2

条件分岐の基本構造

3.2.1

if 文

3
章

この章では、まずは分岐について紹介していくよ。
松田くん、この前作っていたチャットボットのプログラムを使わせてくれるかな？

松田くんが作ったチャットボットを見てみましょう（コード 3-1）。

コード 3-1 いつも同じことを言うチャットボット

```
1 name = input('あなたの名前を教えてください >>')
2 print('{}さん、こんにちは'.format(name))
3 food = input('{}さんの好きな食べ物を教えてください >>'.format(name))
4 print('私も{}が好きですよ'.format(food))
```

実行結果

あなたの名前を教えてください >>松田

松田さん、こんにちは

松田さんの好きな食べ物を教えてください >>カレー

私もカレーが好きですよ



このプログラム、いつも同じ返事をするから面白くないんですよね…。

そういうときこそ、分岐を使えば違う返事をするようにできるんだよ！



構文はのちほど紹介しますので、まずは分岐を体験してみましょう。4 行目と 6 行目の: (コロン) を忘れやすいので注意してください。また、5 行目と 7 行目は、半角スペース 4 個分によって字下げされていることに着目してください。

コード 3-2 のプログラムは、入力した食べ物によって異なる返事をします。まず、「カレー」と入力してみてください。

コード 3-2 答えが分岐するチャットボット

```

1 name = input('あなたの名前を教えてください >>')
2 print('{}さん、こんにちは'.format(name))
3 food = input('{}さんの好きな食べ物を教えてください >>'
               .format(name))
4 if food == 'カレー': 変数 food が 'カレー' だったら
5     print('素敵です。カレーは最高ですね!!')
6 else:
7     print('私も{}が好きですよ'.format(food))

```

実行結果

あなたの名前を教えてください >>松田

松田さん、こんにちは

松田さんの好きな食べ物を教えてください >>カレー

素敵です。カレーは最高ですね!! 5 行目が実行された

次は、「カレー」以外の食べ物を入力してみましょう。

実行結果

あなたの名前を教えてください >>松田
 松田さん、こんにちは
 松田さんの好きな食べ物を教えてください >>ラーメン
 私もラーメンが好きですよ

7行目が実行された



すごい！ ちゃんと違う答えが返ってきましたよ！

コード 3-2 の処理の流れを図で表すと、図 3-3 のようになります。このように、処理の順序を箱や矢印で表した図を**フローチャート** (flowchart) または**流れ図**といいます。

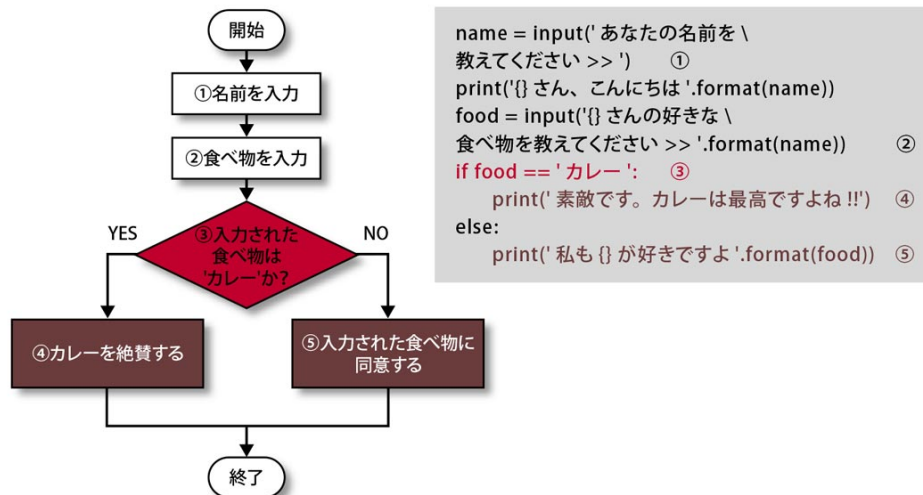


図 3-3 分岐のフローチャート



コードとフローチャートを見比べてみると、次のことが読み取れるだろう。

- if の後ろに処理が分岐する条件を書く。
- 条件が成立していたら、if から else の前までの文が実行される。
- 条件が成立していなかったら、else 以降の文が実行される。

if の後ろに記述する、分岐する条件のことを**条件式**といい、条件式を使って分岐させる文を**if 文**といいます。

if は「もし～ならば」という意味の英単語なので、「if food == 'カレー」は「もし変数 food が 'カレー' ならば」と読むことができ、まさにそのとおりに動作します。条件が成立していた場合に実行される文は複数記述することができ、これを**if ブロック**と呼びます(図 3-4)。

if ブロックの後ろにある else は「そうでなければ」という意味を持つ英単語なので、「変数 food が 'カレー' でなければ」と読むことができます。条件が成立していなかった場合に実行される文も複数記述することができ、これを**else ブロック**といいます。

このように、「複数の文がまとまった部分」のことを**ブロック**と呼びます。

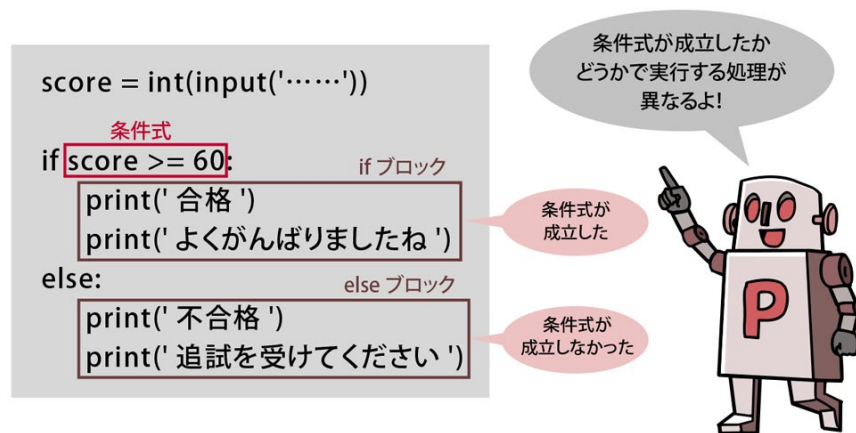


図 3-4 分岐は条件式とブロックから構成される



英単語の意味と関連付けると覚えやすそう! 「もし○○ならば A を実行する、そうでなければ B を実行する」ということね。



Column

チャットボットと AI

チャットボット (chatbot) とは、テキストや音声を通じて、会話を自動的に行うプログラムです。自動的とはいっても、あらかじめ想定されたパターンに沿った表層的な対話を目的としているため、「人工無脳」とも呼ばれます。これに対して、人間の思考を再現し、文脈を理解したうえで対話することを目的としたものを一般的に人工知能 (AI: Artificial Intelligence) と位置付けています。

3章

3.2.2 ブロックとインデント



あれれ？ 今度は社内試験の結果を判定するプログラムを作ってみたんですが、同じように書いているのに、おかしい結果になっちゃいました。

松田くんが書いたプログラムを見てみましょう (コード 3-3)。

コード 3-3 常に追試を受けることになる判定プログラム

```
1 score = int(input('試験の点数を入力してください >>'))
2 if score >= 60: 変数 score が 60 以上だったら
3     print('合格！')
4     print('よくがんばりましたね！')
5 else:
6     print('残念ながら不合格です！')
7 print('追試を受けてください')
```

実行結果

試験の点数を入力してください >>80

合格！

よくがんばりましたね

追試を受けてください

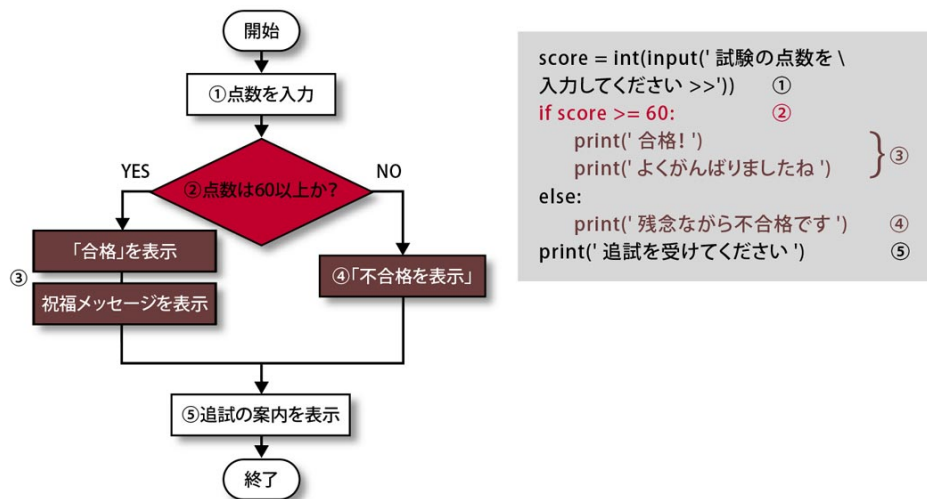


合格なのに、追試を受けろって言われちゃったのね。やっぱり日頃の行いのせいじゃない？

いいミスをしてくれたね！ 原因は日頃の行いじゃなくて、インデントだよ。



Python では、ブロックの範囲を**インデント**で示す必要がありますが、松田くんの書いたコード 3-3 をよく見ると、最終行の文の先頭が字下げされていません。そのため、else ブロックの範囲は 6 行目の「print(' 残念ながら不合格です)」という 1 つの文のみと解釈されてしまいました。最終行は else ブロックとはみなされないため、条件の成立／不成立にかかわらず、必ず実行されることになるのです(図 3-5)。





インデントでブロックを指定する

ブロックの範囲はインデントによって指定する。

3章



一般的に、インデントはコードを見やすくするもので、インデントするかどうかは自由というプログラミング言語が多いんだ。だが、Python では重要な意味を持っているので、プログラミング経験者は十分に注意しよう。

また、インデントの入れ方にもルールがあるので注意しましょう。「何個分の半角スペースをインデントとするか」には特に決まりはありません。しかし、1つのブロックの中では個数を揃える必要があり、揃っていない場合はエラー (IndentationError) が発生します。

```
1  if x >= 100:
2      print('hoge')
3      print('hoge')
4  else:
5      # 省略
```

同じブロック内で半角スペースの個数が揃っていないのでエラー

一般的にはインデントは、半角スペースを2個、4個、8個分のいずれかにすることが多いようです。チームや企業内で決まりがあればそれに従いましょう。もし、そのようなルールがない場合は、Python の標準コーディング規約である **PEP8** で推奨されている4個にすることをお勧めします。なお、本書でも基本的に4個で紹介しています。



JupyterLab でのインデント

JupyterLab では、: (コロン) を入力したあとに改行すると、以降の行の先頭に半角スペースを 4 個分入れて自動でインデントが行われる。

※うまくインデントされない場合は、自分でインデントを行う必要がある。

分岐の構造を紹介したこの節の最後に、if 文の構文を確認しておきましょう。



if 文の基本構文

if 条件式 :

条件が成立したときの処理 (if ブロック)

else:

条件が成立しなかったときの処理 (else ブロック)

※ブロックはインデント (字下げ) によって指定する。



Column

タブ文字

インデントを [Tab] キーで入れる場合、何が入力されるかは使用するソフトウェアによって異なります。JupyterLab では、半角スペースが 4 個分入力されますが、設定によっては異なる個数の半角スペースやタブ文字という特殊文字が入る場合もあります。Python 3 以降では、同じブロック内でタブ文字とスペースが混在するとエラー (TabError) が発生します。もし [Tab] キーでコーディングしたい場合には、設定をしっかりと確認し、動作を確認したうえで使いましょう。

3.3

条件式

3.3.1

比較演算子

3
章

工藤さん、条件式の中に「==」とか「>=」が出てきましたけど、これで条件を判定しているんですよね？

そうだよ！ 次は条件式について詳しく見ていこう。



条件式を作るには、これまで登場した「==」や「>=」などの**比較演算子**（または**関係演算子**）を使います。比較演算子には、表 3-1 のような種類があります。

表 3-1 比較演算子の種類と意味

演算子	意味
==	左辺と右辺は等しい
!=	左辺と右辺は等しくない
>	左辺は右辺より大きい
<	左辺は右辺より小さい
>=	左辺は右辺より大きい か等しい
<=	左辺は右辺より小さい か等しい

比較演算子を使うと、たとえば、次のような条件式を作ることができます。

- `score == 100` : 変数 `score` が 100 なら
- `name == '浅木'` : 変数 `name` が '浅木' なら
- `passowrd != '1a#s5S'` : 変数 `password` が '1a#s5S' でなければ
- `temperature < 0` : 変数 `temperature` が 0 より小さければ

特に、等しいことを表現する比較演算子は `==`（イコール記号を 2 つ記述）であ

ることに注意してください。誤ってイコール記号を 1 つしか書かないと構文エラー (SyntaxError) が発生します。



うっかり「if score = 100:」とか書きちゃいそうです。

初心者がやってしまうミスの筆頭だからね。もし if 文に構文エラーが出たら、まずはこれを疑ってみるといいよ。



3.3.2 in 演算子



工藤さん、僕のチャットボット、カツカレーでもシーフードカレーでも、とにかくカレーを入力したら「最高です！」って、言っ
てほしいんですけど…。

どんだけカレーが好きなんだい。まあ、ちょうどいい比較演算子があるよ。



どのような演算子を使うのか見てみましょう (コード 3-4)。

コード 3-4 どんなカレーでも絶賛するチャットボット

```
1 name = input('あなたの名前を教えてください >>')
2 print('{}さん、こんにちは'.format(name))
3 food = input('{}さんの好きな食べ物を教えてください >>'.format(name))
4 if 'カレー' in food:
5     print('素敵です。カレーは最高ですね!!')
```

変数 food に 'カレー' が含まれているか

```
6 else:
7     print('私も{}が好きですよ'.format(food))
```

実行結果

あなたの名前を教えてください >>松田

松田さん、こんにちは

松田さんの好きな食べ物を教えてください >>カツカレー

素敵です。カレーは最高ですよね!! **if ブロックが実行された**

in 演算子は、右辺に左辺の値が含まれているかを判定することができる比較演算子です。コード 3-4 の場合、変数 food に 'カレー' という文字列が含まれていれば、条件が成立したと判定されます。



この演算子、調べたい変数名を右側を書くんだね。

それはきっと「in」だからよね。カレーが food の中にあるかどうかを尋ねたいわけだから、ね。



なお、in 演算子の右辺には、第 2 章で紹介したコレクションを使用することもできます(コード 3-5)。

コード 3-5 100 点があるかどうかを調べる

```
1 scores = [80, 100, 20, 60] 試験結果のリスト
2 if 100 in scores:
3     print('100点満点の試験があったんですね。おめでとう!')
4 else:
5     print('次はどれか1つでも100点満点をとろう')
```

実行結果

100点満点の試験があったんですね。おめでとう！

これを利用して、右辺にディクショナリを指定すると、そのディクショナリの中でキーが使用されているかをチェックすることができます(コード 3-6)。

コード 3-6 ディクショナリのキーをチェックする

```
1 scores = {'network': 60, 'database': 80, 'security': 50}
2 key = input('追加する科目名を入力してください >>')
3 if key in scores:
4     print('すでに登録済みです')
5 else:
6     data = int(input('得点を入力してください >>'))
7     scores[key] = data
8 print(scores)
```

試験結果のディクショナリ

実行結果 (まだ登録していないキーを入力した場合)

追加する科目名を入力してください >>python

得点を入力してください >>80

{'network': 60, 'database': 80, 'security': 50, 'python': 80}

実行結果 (すでに登録したキーを入力した場合)

追加する科目名を入力してください >>network

すでに登録済みです

{'network': 60, 'database': 80, 'security': 50}

すでにあるディクショナリの要素を変更されたくない場合(2.3.4 項)には、in

演算子による入力チェックを行うことで、要素の不用意な変更を防ぐことができます。



ディクショナリのキーの存在を調べる

キー in ディクショナリ

※キーがディクショナリに存在する場合は条件成立、存在しない場合は条件不成立となる。

3章



Column

文字列の大小比較

値の大小は、文字列でも比較することができます。

```
1 name = '松田'
2 if name < '浅木':
3     print('条件が成立')
4 else:
5     print('条件が成立しない')
```

実行結果

条件が成立

文字列で大小比較を行うと、Python の内部では、その文字に対応した文字コードによって比較されます。文字コードには、代表的なものとして UTF-8 や Shift_JIS、ASCII などがありますが、Python 3 では一般的に UTF-8 を文字コードとして使用しています。

3.3.3 真偽値

この節で紹介した比較演算子は、第 1 章で紹介した算術演算子と同じく、演算子の仲間です。ここで、演算子を使用したときのルールをもう一度思い出してみましょう。

式に含まれる演算子は、1 つずつ、周囲のオペランドを巻き添えにしながら置き換わっていき、最終的に 1 つの計算結果となります。このような過程を式の評価というのでした(1.1.4 項)。たとえば、算術演算子を用いた「 $3 + 2 * 5$ 」という計算式は、評価されて計算結果の「13」に置き換わります。



それじゃ、比較演算子を用いた条件式は、評価されると何に置き換わるか、わかるかな？

比較演算子も演算子ですから、評価されると別のものに置き換わります。比較演算子を用いた条件式の場合、条件が成立したら True、成立しなければ False に置き換わります(図 3-6)。True と False は真偽値といい、第 1 章で紹介したデータ型のうちの 1 つ、bool 型です(1.3.1 項)。

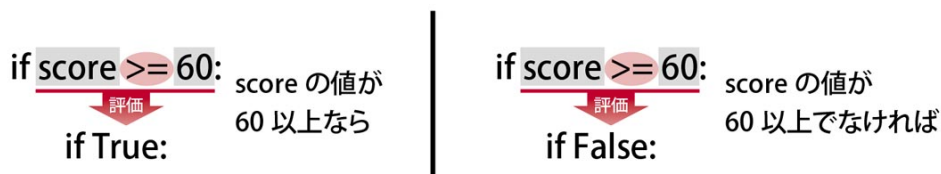


図 3-6 条件式は真偽値に評価される



True と False って、文字列なんですか？

いや、True と False は文字列じゃない。だから ' や " は付けないぞ。あくまでも、True と False という値なんだ。



条件式が真偽値に置き換わる様子は、コードで実際に確認することができます (コード 3-7)。

コード 3-7 条件式の評価結果を確認する

```
1 score = int(input('試験の点数を入力 >>'))  
2 print(score >= 60)
```

実行結果

試験の点数を入力 >>60
True

実行結果

試験の点数を入力 >>50
False

3章

以上のことを踏まえると、**if 文は条件式の評価結果が True ならば if ブロックを、False ならば else ブロックを実行する文**ととらえることができます (図 3-7)。

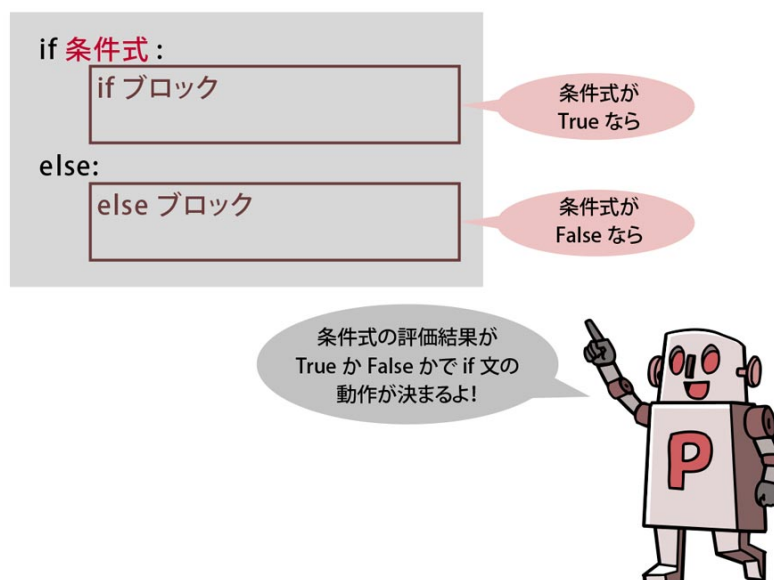


図 3-7 条件式の評価結果によって if 文の動作は決定する

3.3.4 論理演算子



工藤さん、社内試験は 60 点以上が合格ですが、「score >= 60」だと 100 より大きい点も合格になっちゃって…。

判定の条件が 2 つ以上あるときには、論理演算子を使うといいよ。



これまで登場した条件式は、すべて 1 つの条件だけで判定してきましたが、複数の条件を組み合わせた条件式も作ることができます。たとえば、試験結果を判定する条件として、60 ～ 100 点の範囲のみを合格としたい場合は、次のように記述します。

```
if score >= 60 and score <= 100: 60 点以上かつ 100 点以下だったら
```

この条件式は、「score >= 60」と「score <= 100」の 2 つの条件から成り立っています。このように、2 つ以上の条件を組み合わせるには、**論理演算子**を使用します(表 3-2)。

表 3-2 論理演算子とその意味

演算子	意味
and	かつ
or	または
not	でなければ(否定)

60 ～ 100 点の範囲以外を不合格とする条件は、or を使って次のように表すことができます。

```
if score < 60 or score > 100: 60 点未満または 100 点より大きかったら
```

論理演算子を使うと、条件式は図 3-8 のように評価されます。

andの例：変数scoreの値が110のとき
もしscoreが60以上で かつ 100以下なら

if score \geq 60 and score \leq 100 :



if True and False :



if False :

orの例：変数scoreの値が50のとき
もし60未満 または 100より大きかったら

if score < 60 or score > 100 :



if True or False :



if True :



andは「かつ」つまり左辺と右辺の両方が成立したときTrueになる。
それに対してorはどちらか一方が成立すればTrueだ。この違いを理解しよう

図 3-8 論理演算子を使った条件式の評価



論理演算子は難しく考えずに、英単語の意味そのままと思えばよさそうね。

not は、「not 条件式」という形で使用します。条件式が成立しなければ True、成立したら False というように、通常とは逆の真偽値に置き換えるという演算を行います。また、「if not 条件式:」とすることで、もし条件式が成立しなければという条件を作ることができます。

```
if not (score < 60 or score > 100):
```

「60 点未満または 100 点より大きい」でなければ



「60 点未満または 100 点より大きい」でなければ、ということは…つまり合格ってことね！ 最初の条件と同じじゃない！

うーん、わかりにくいよ…。not ってどんなときに使えばいいんだろう？



not は and や or ほど利用する場面は多くありませんが、in 演算子を使った条件式では、not を使うと、逆の条件をシンプルに記述することができます。

```
if not 'カレー' in food: food に 'カレー' が含まれていなければ
```

not 演算子を活用した例は、次の第 4 章でも登場しますので、まずは and と or を優先的に覚えておきましょう。



Column

範囲指定の条件式

ある 1 つの変数を取り得る値の範囲を判定する条件は、次のように記述することもできます。

```
if 60 <= score <= 100: score >= 60 and score <= 100:
```



数学と同じ書き方なのね。なじみがあってわかりやすいわ。

人によっては直感的でとてもわかりやすい表現ですが、Python 以外のほとんどのプログラミング言語ではサポートしていない書き方なので注意してください。



Column

論理演算子の名前の由来

論理演算子の左右に記述する条件式は、評価されると True または False に置き換わるので、論理演算子は True または False に対する演算をしていると言えます。True と False は真偽値以外にも真理値や論理値とも呼ばれます。論理値に対する演算をするので、論理演算子という名前が付いています。



Column

真偽値に評価されない条件式

本文で紹介している条件式は、すべて真偽値に評価されるものです。しかし、Python では次のような条件式を書くことも許されています。

```
1 score = 0
2 if score:
3     # 処理
4 else:
5     # 処理
```



えっ？ これってどういう意味？

条件式の結果が真偽値にならない場合、Python は 0 か 0 以外かによって、実行するブロックを決定します。結果が 0 以外なら True と解釈して if ブロックを、結果が 0 なら False と解釈して else ブロックを実行します。先ほどのコードをこのルールに則って読み解くと、変数 score は

0 なので、else ブロックが実行されることになります。



何だ。結局「score != 0」と同じってことか。

なお、次に挙げるものは、0 と同様にすべて False と解釈される値です。通常、このような条件式をあえて用いる必要はありませんが、開発現場などでは見かける可能性もありますので、知識として知っておくとよいでしょう。

False None 0 0.0 " "" [] {} ()

※ "、"" は空の文字列、[]・{}・() は空のコレクションを表す。



これはあくまでも参考情報だよ。自分がコードを書く場合には、分岐の意味が明確になる True か False で評価される条件式を使おう。

3.4

分岐構文のバリエーション

3.4.1

3 種類の if 文

3
章

さあ、分岐についてはいよいよ大詰めだ。if 文にはいろいろな構文があるから、それを整理して紹介するよ。

if 文には 3 つのバリエーションがあります。この章でこれまで紹介してきた if 文は、基本形の if-else 構文です。これ以外にもあと 2 つの構文があり、それらは条件式の結果が False の場合に、基本形とは処理の流れが変わってきます (図 3-9)。

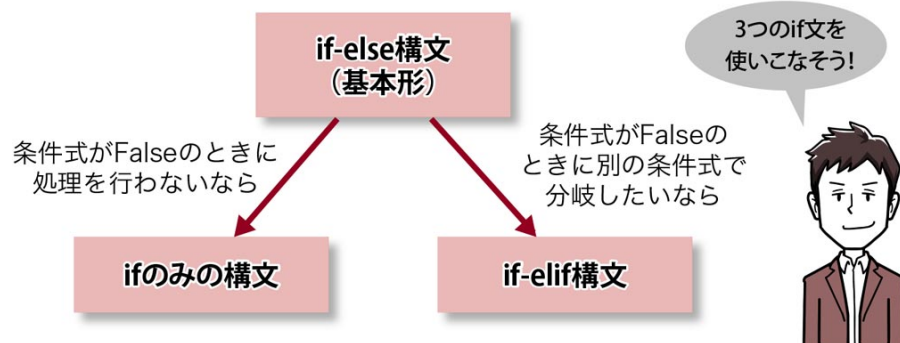


図 3-9 if 文のバリエーション

これらの構文の違いを理解して、if 文を使いこなしていきましょう。

3.4.2

if-else 構文

まずは基本形となる if-else 構文を確認しておきましょう。条件式が成立したときと、成立しなかったときで処理を分けることができます (図 3-10)。



if-else 構文

if 条件式 :

条件式が成立したときの処理

else:

条件式が成立しなかったときの処理

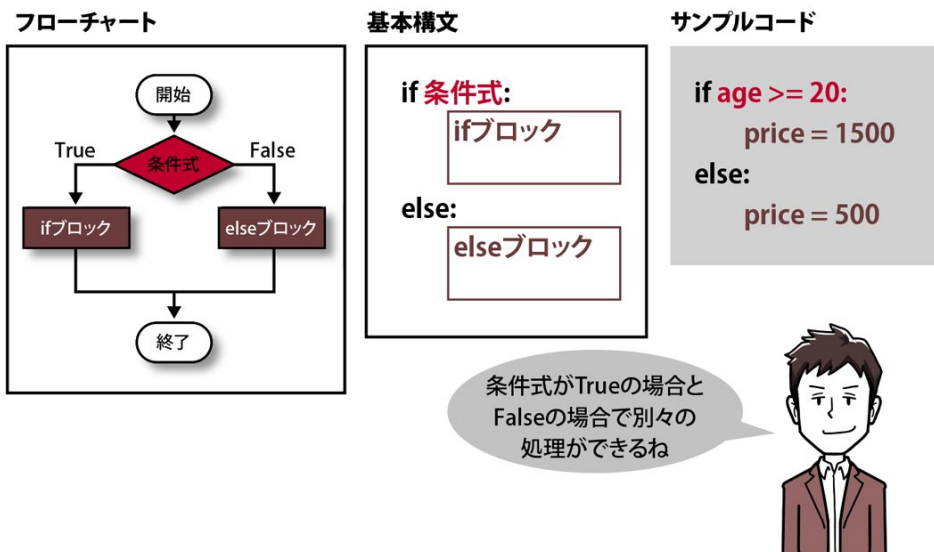
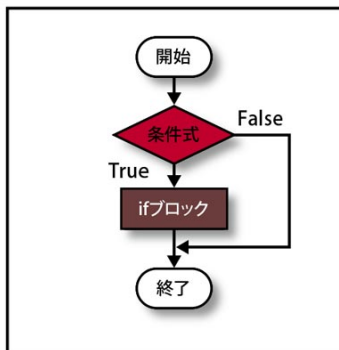


図 3-10 if-else 構文

3.4.3 if のみの構文

条件式が成立しなかったときには何もしない場合、else ブロックは空になります。このようなときには、else ブロックを省略することができます。これが if のみの構文となります(図 3-11)。

フローチャート



基本構文

```
if 条件式:
    ifブロック
```

サンプルコード

```
if age >= 20:
    price = 1500
```

elseを省略しただけね



3
章

図 3-11 if のみの構文



if のみの構文

if 条件式:

条件式が成立したときの処理

if のみの構文の例を見てみましょう (コード 3-8)。

コード 3-8 else ブロックのない分岐

```

1 name = input('あなたの名前を教えてください >>')
2 print('{}さん、こんにちは'.format(name))
3 if name == '松田':
4     print('松田さんに会えてうれしいです')
5 food = input('{}さんの好きな食べ物を教えてください >>')
   .format(name))
```

name が '松田' ではないときの処理はない

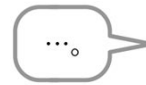
```
6 if 'カレー' in food:
7     print('素敵です。とにかくカレーは最高ですよ!!')
8 else:
9     print('私も{}が大好きですよ'.format(food))
```

実行結果

あなたの名前を教えてください >>松田
松田さん、こんにちは
松田さんに会えてうれしいです
松田さんの好きな食べ物を教えてください >>カレー
素敵です。とにかくカレーは最高ですよ!!



むふふ。僕だけに特別な挨拶をするようにできたぞ。



Column

空ブロックの作り方

Python では空のブロックを禁じているため、if のみの構文を使用せずに、何も処理をしない空の else ブロックを書くとエラーになります。ブロックの中を空にしたい場合は、**pass** とだけ書いて、何もしないことを表明する必要があります (図 3-12)。

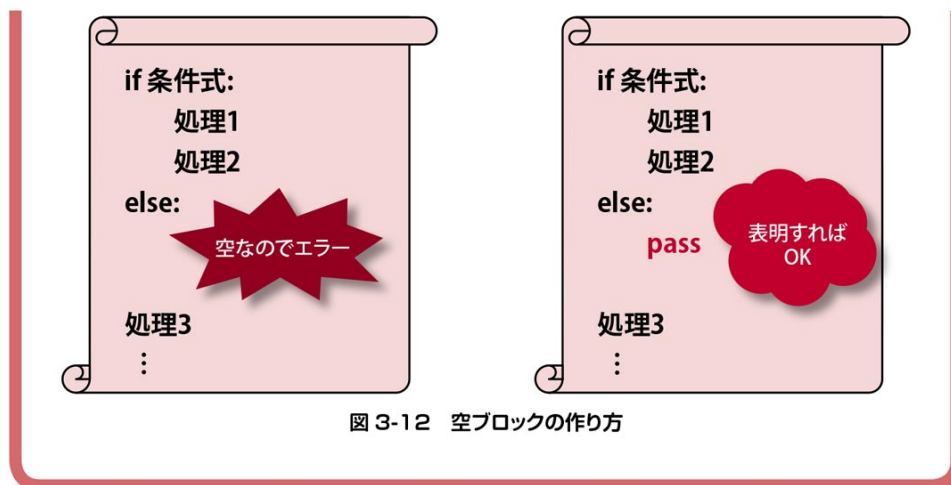


図 3-12 空ブロックの作り方

3.4.4 if-elif 構文

条件式が成立しなかったときには別の条件式で判定したい場合は、if ブロックのあとに **elif ブロック**を追加した if-elif 構文を使用します(図 3-13)。

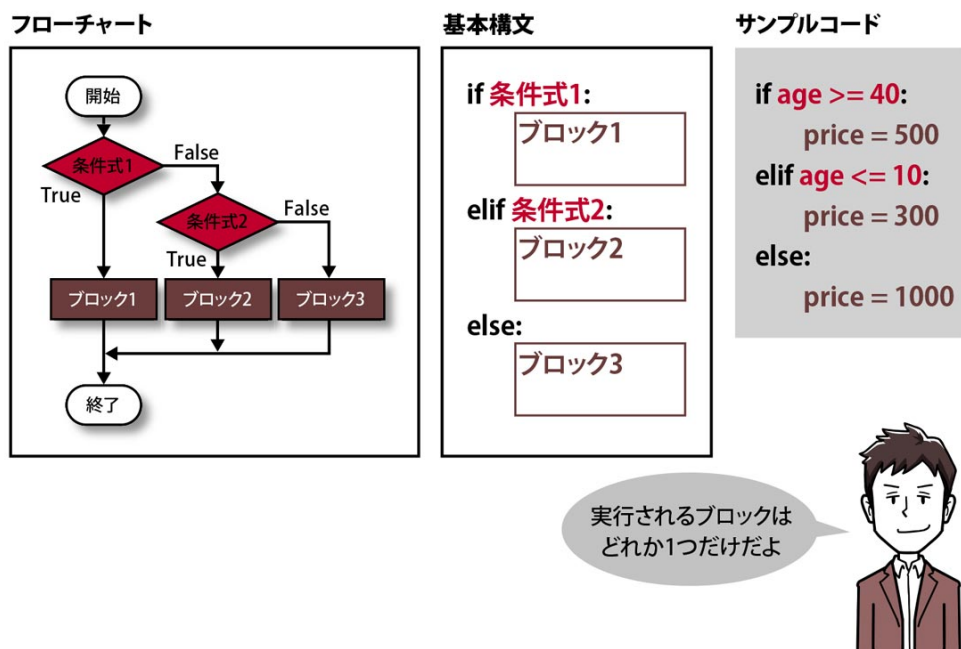


図 3-13 if-elif 構文

図 3-13 では elif は 1 つだけですが、elif の数には制限はなく、必要なだけ書くことができます。条件式を先頭から順に判定していき、**最初に** True になった条件式のブロックが実行されます。elif とは「else if」を省略したものですから、「if A ～ elif B ～」は、「もし A ならば～、そうでなく B ならば～」という意味となります。

また、すべての条件式が False だったときに何もする必要がなければ、最後の else ブロックを省略することができます。



if-elif 構文

if 条件式 1:

条件式 1 が成立したときの処理

elif 条件式 2:

条件式 1 が成立せず、条件式 2 が成立したときの処理

⋮

elif 条件式 n:

上記の条件式がすべて成立せず、条件式 n が成立したときの処理

else:

すべての条件式が成立しなかったときの処理

※ else ブロックは省略可能 (「else:」自体の記述が不要)。



断っても断っても、どんどん条件を変えてご飯に誘ってくる人みたいね。

if-elif 構文の例を見てみましょう (コード 3-9)。

コード 3-9 多分岐する if 文

```
1 score = int(input('試験の点数を入力してください >>'))
2 if score < 0 or score > 100: ) 条件式 1
3     print('異常な得点です')
4     print('入力し直してください')
5 elif score >= 60: ) 条件式 2
6     print('合格!')
7     print('よくがんばりましたね')
8 else:
9     print('残念ながら不合格です')
10    print('追試を受けてください') ] これまでの条件式がすべて False だったら実行される
```

実行結果 (条件式 1 が True の場合)

試験の点数を入力してください >>120

異常な得点です

入力し直してください

実行結果 (条件式 1 が False、条件式 2 が True の場合)

試験の点数を入力してください >>70

合格!

よくがんばりましたね

実行結果 (条件式 1 が False、条件式 2 も False の場合)

試験の点数を入力してください >>50

残念ながら不合格です

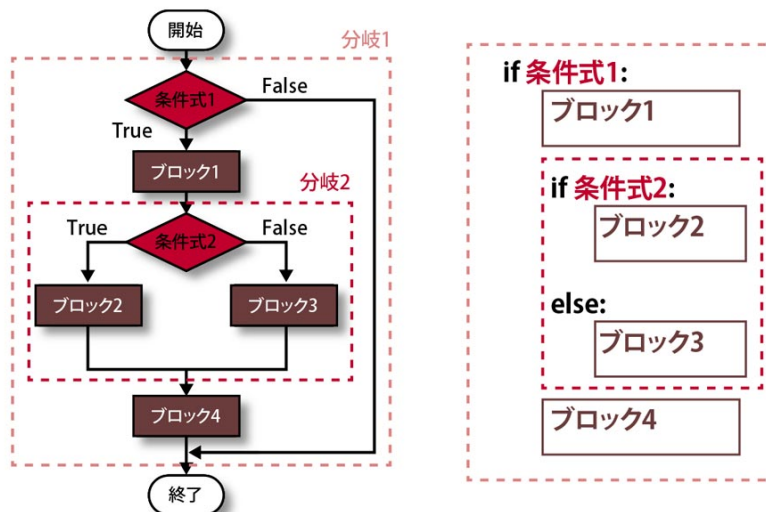
追試を受けてください

3.4.5 if 文のネスト



if 文のバリエーションは以上だよ。最後にこれらの構文を組み合わせる方法を紹介しておくよ。

これまで解説してきた if 文のブロックに、また別の if 文を入れることもできます。このような多重構造を、コレクションのときと同様に、ネストや入れ子といいます(図 3-14)。if 文をネストすることで、複雑な分岐を作ることができます。



ブロックを指示するのはインデントだということに再度注意を払ってほしい。if 文をネストする場合は意図とインデントが合致しているか、十分検証しよう



図 3-14 ネストした if 文



if 文をネストさせて、僕の晩ご飯をお勧めしてくれるチャットボットを作ってみましたよ。

松田くんが新しいチャットボットを作りました(コード 3-10)。

コード 3-10 晩ご飯をRecommendするチャットボット

```
1 print('すべての質問に y または n で教えてください')
2 okane_aruka = input('お金に余裕はありますか? >>')
3 if okane_aruka == 'y': ) if-else 構文
4     onaka_suiteruka = input('お腹がすごく空いてますか? >>')
5     nomitai_kibunka = input('ビールを飲みたいですか? >>')
6     if onaka_suiteruka == 'y' and nomitai_kibunka == 'y':
7         print('焼き肉はいかがですか')
8     elif onaka_suiteruka == 'y': ) if-elif 構文
9         print('カレーはいかがですか')
10    elif nomitai_kibunka == 'y': ) if-elif 構文
11        print('焼き鳥はいかがですか')
12    else: ) if-elif 構文
13        print('パスタはいかがですか')
14    yashoku_iruka = input('夜食は必要ですか? >>')
15    if yashoku_iruka == 'y': ) if のみの構文
16        print('コンビニのチキンはいかがですか')
17 else: ) if-else 構文
18    print('家で食べましょう')
```

実行結果

すべての質問に y または n で教えてください
お金に余裕はありますか? >>y
お腹がすごく空いてますか? >>y
ビールを飲みたいですか? >>n
カレーはいかがですか

夜食は必要ですか？ >>y
コンビニのチキンはいかがですか



すごいじゃないか！ ここまでできたら分岐処理はもうマスターしているよ。

コード 3-10 は、本章で学習した if の構文がすべて入っています。コードを眺めるだけでは、全体の構造を把握するのは難しいかもしれません。何度も実行して質問に答え、その結果とコードを見比べながらじっくり理解してみてください。



僕たちの日常生活には分岐する場面がたくさんあるよ。その一部を切り取って、プログラムにしてみるといい勉強になるかもしれないね。

やってみます！



3.5

第 3 章のまとめ

この章では、次のことを学びました。

3
章**文と制御構造**

- 1 行に記述された 1 つの処理が実行単位であり、1 つの文である。
- 文の実行順序は制御構造によってコントロールすることができ、主に順次・分岐・繰り返し (ループ) の 3 つがある。

条件分岐

- if 文は、ある条件に従って処理を分岐させることができる。
- if 文は、条件が成立したら if ブロックを、不成立だったら else ブロックを実行する。
- ブロックは、複数の文をひとまとまりとして扱う (文は 1 つでもよい)。
- ブロックはインデントによって指定する。

条件式

- 比較演算子や論理演算子を用いて分岐する条件を記述する。
- 条件式は評価されると真偽値に置き換わる。

分岐構文のバリエーション

- if-else 構文は処理を 2 つに分岐させることができる。
- if のみの構文は、ある処理を実行するか、実行しないかに分岐させることができる。
- if-elif 構文は処理を 3 つ以上に分岐させることができる。
- if 文はネストできる。

3.6

練習問題

練習 3-1

次の各条件式が評価している内容を日本語で答えてください。もし条件式として適当でない場合は、×を答えてください。

- (1) `price * 1.1 <= 300000`
- (2) `n = 0`
- (3) `'gihu' in kansai`
- (4) `a + b > 60 and day == 3`
- (5) `False`

練習 3-2

次のような判定ができる条件式をそれぞれ記述してください。

- (1) 変数 `initial` の値は「K」と等しいか
- (2) 変数 `point` の値は 80 以上かつ 256 未満か
- (3) 変数 `bmi` の値が 20 より小さいかまたは 25 より大きい
- (4) 変数 `year` の値は 4 で割り切れるか
- (5) 変数 `day` の値は 28・30・31 のいずれにも当てはまらないか

練習 3-3

if 文を使って、次のような動作をするプログラムをそれぞれ作成してください。

- (1) 変数 `isError` が `False` かつ変数 `n` が 100 未満の場合のみ、画面表示を行う（表示内容は問わない）。
- (2) 入力された数値について、偶数か奇数かを判定してその結果を表示する。
- (3) 入力された次の文字列に応じて、挨拶を表示する。
 - ・ こんにちは→ようこそ！
 - ・ 景気は？ →ぼちぼちです
 - ・ さようなら→お元気で！

- ・上記以外は、「どうしました?」を表示。

練習 3-4

次のプログラムを実行したとき、①～③のブロックが実行される入力値をそれぞれ教えてください。また、そのときに表示される内容を教えてください。

```
1 month = int(input('今は何月ですか? (数字を入力) >>'))
2 if month in [1, 3, 5, 7, 8, 10, 12]:
3     print('31日までありますね') )— ブロック①
4 else:
5     if month != 2:
6         print('30日までですね') )— ブロック②
7     else:
8         print('1年で一番寒い月ですね') )— ブロック③
9     print('年が明けてから')
10 print('{}箇月が過ぎました'.format(month))
```

3.7

練習問題の解答

練習 3-1

- (1)変数 price の値に 1.1 を掛けた値は 300000 以下か
- (2)×
- (3)変数 kansai に「'gihu'」は含まれるか
- (4)変数 a と b の合計が 60 よりも大きく、かつ変数 day の値は 3 と等しいか
- (5)False かどうか

(2)n = 0

= 記号 1 つは代入演算子を意味します。等しいかどうかを判定するには、比較演算子の == を使います。

(5)False

条件式は bool 型に評価されるので、False そのものを記述することもできます。この条件式は常に else ブロックのみを実行します。

練習 3-2

- (1)initial == 'K'
- (2)point >= 80 and point < 256 (別解)80 <= point < 256
- (3)bmi < 20 or bmi > 25
- (4)year % 4 == 0
- (5)not(day in [28, 30, 31])

練習 3-3

(1)

```
1  isError = False
2  n = 99
3  if isError == False and n < 100:
```

```
4 print('正解です')
```

(2)

```
1 number = int(input('数値を入力してください >>'))
2 if number % 2 == 0:
3     print('偶数です')
4 else:
5     print('奇数です')
```

3章

(3)

```
1 greeting = input('挨拶をどうぞ >>')
2 if greeting == 'こんにちは':
3     print('ようこそ!')
4 elif greeting == '景気は?':
5     print('ぼちぼちです')
6 elif greeting == 'さようなら':
7     print('お元気で!')
8 else:
9     print('どうしました?')
```

練習 3-4

ブロック①: 入力値が 1 (または 3、5、7、8、10、12) の場合に実行される。

実行結果 (1 を入力した場合)

31日までありますね

1箇月が過ぎました

ブロック②:入力値が 4 (または 6、9、11)の場合に実行される。

実行結果 (4 を入力した場合)

30日までありますね
年が明けてから
4箇月が過ぎました

ブロック③:入力値が 2 の場合に実行される。

実行結果 (2 を入力した場合)

1年で一番寒い月ですね
年が明けてから
2箇月が過ぎました



Column

三項条件演算子

練習 3-2 (2) の解答 2 ~ 5 行目は、**三項条件演算子**または**三項演算子**という構文を用いると、次のように簡潔に表現できます。

```
1 div = '偶数' if number % 2 == 0 else '奇数'  
2 print('{}です'.format(div))
```



三項条件演算子

値 1 if 条件式 else 値 2

※条件式が成立すれば値 1 に、そうでなければ値 2 に全体が「化ける」。

第 4 章

繰り返し

前章では、if 文による処理の分岐を学びました。
この章で紹介する繰り返しでは、
適切な条件を設定することによって
同じような処理を何度も実行できます。
3 つの制御構造をマスターして、
プログラムの流れを自由自在に操れるようになりましょう。

CONTENTS

- 4.1 繰り返しの基本構造
- 4.2 for 文
- 4.3 繰り返しの制御
- 4.4 第 4 章のまとめ
- 4.5 練習問題
- 4.6 練習問題の解答

4.1

繰り返しの基本構造

4.1.1 while 文



工藤さんのアドバイスに従って、自分の生活の一部をプログラムにしてみたんですが…。イマイチなんです。

浅木さんのプログラムを見てみましょう (コード 4-1)。

コード 4-1 ひつじを数えて眠る

```
1 print('さあ、寝ようかしら')
2 count = 0    # ひつじの数
3 count += 1
4 print('ひつじが{}匹'.format(count))
5 count += 1
6 print('ひつじが{}匹'.format(count))
7 count += 1
8 print('ひつじが{}匹'.format(count))
9 print('おやすみなさい…')
```

ひつじの数を 1 匹
ずつ増やして表示

実行結果

```
さあ、寝ようかしら
ひつじが1匹
ひつじが2匹
ひつじが3匹
```

おやすみなさい…



先輩、寝つくのめっちゃ早いんですね！ 僕はこう見えても繊細だから、100匹は数えないと。

あら、人は見かけによらないわね。同じ処理を何度も書くのは面倒だけど、がんばってコピペするしかないんですか？



4章

コード 4-1 の 3～8 行目では、変数 `count` に 1 を足して、その内容を表示する処理を 3 回繰り返しています。変数 `count` の値は表示の都度変わりますが、処理の本質は同じです。このように、同じような処理を繰り返したい場面はプログラミングにおいて非常によく登場します。しかし、実行したい分だけ同じ処理を書いていくのは現実的ではありません。



今回はたったの 3 回だけれど、松田くんみたいに 100 回も 1,000 回も必要になったら大変だよね。こういうときは「繰り返し」を使えばいいんだよ。

繰り返しは、第 3 章で紹介した分岐と同じく、代表的な制御構造のうちの 1 つです (P120)。分岐は条件によって実行する処理を変えることができましたが、繰り返しは条件によって何度も処理を繰り返すことができます。

さっそく、次のコードを書いて実行してみましょう (コード 4-2)。

コード 4-2 ひつじを数えるのを 3 回繰り返す

```
1 count = 0
2 while count < 3:
```

```

3     count += 1
4     print('ひつじが{}匹'.format(count))
5     print('おやすみなさい...')

```

インデント

実行結果

ひつじが1匹
ひつじが2匹
ひつじが3匹
おやすみなさい…



わっ!? print 関数を 1 回しか書いてないのに、ちゃんと 3 回表示されてるわ!

コード 4-2 の処理の流れをフローチャートで表すと、図 4-1 のようになります。

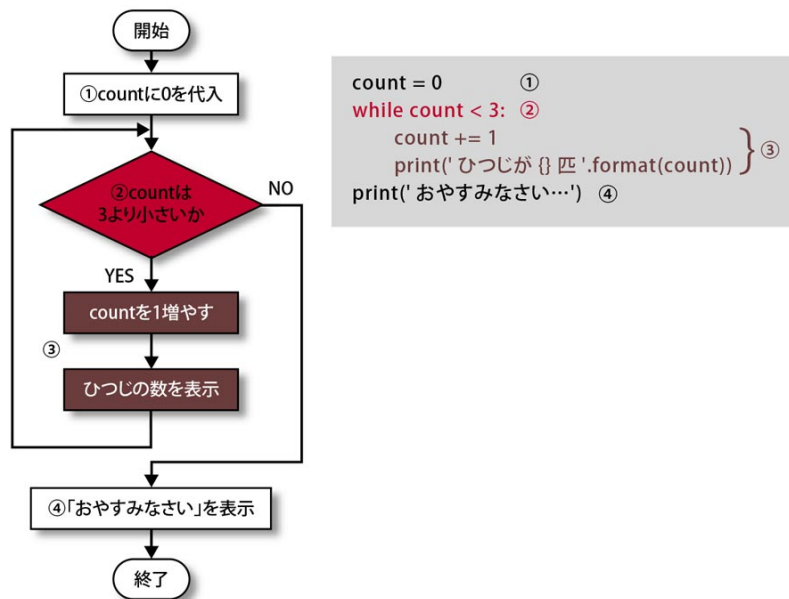


図 4-1 繰り返しのフローチャート



if 文のときと同じように、コードとフローチャートを見比べてみよう。次のことが読み取れるかな？

- while の後ろに処理を繰り返す条件式を書く。
- 条件が成立している間は、直後のブロックが繰り返し実行される。
- 条件が成立していなかったら、繰り返しが終わる。

条件によって処理の繰り返しを指示する文を **while 文** といい、後ろに条件式を記述するのは if 文と同じです。while は「～の間」という意味の英単語ですから、「while count < 3」は「変数 count が 3 より小さい間は」ととらえることができます。そしてその文章のとおり、変数 count の値が 3 より小さいという条件が成立している間は、直後のブロックを何度でも繰り返し実行します。

このように、繰り返し実行されるブロックを **while ブロック** といい、コード 4-2 では 3～4 行目がこれに当たります。

4章



if 文と同じで、while 文も英単語の意味と関連付けるとわかりやすいね。

while ブロックを実行したあとは、そのまま下へ進まずに条件式まで戻るといった流れになるのが if 文とは違うわね。



while 文の条件式は、繰り返すたびに判定の内容が変わっていくことに注目してください。変数 count の値は最初は 0 ですが、while ブロックを実行すると 1 増えるため、条件式の内容は「0 < 3」「1 < 3」「2 < 3」「3 < 3」と変化します。「3 < 3」のときに条件式の評価結果が False となり、そこで繰り返しが終了します。結果的には 3 回繰り返したことになりますね。

変数 count のように、繰り返すたびにその値が変化し、繰り返しの条件となる変数のことを **カウンタ変数** または **ループ変数** (**ループカウンタ**) といいます。



繰り返しでは、判定内容が変化するのがポイントだよ。



while 文

while 条件式 :

条件が成立したときの処理 (while ブロック)

※ブロックはインデント (字下げ) によって指定する。

4.1.2 無限ループ



く、工藤さああん！ プログラムが止まらなくなっちゃいました！ 助けてえええ！！

あー、さっそくやっちゃったか！



いったい何が起きたのでしょうか (コード 4-3)。

コード 4-3 無限ループ

```
1 count = 0
2 while count < 3:
3     print('ひつじが{}匹'.format(count))
4     print('スヤスヤ...zzz')
```

実行結果

ひつじが0匹
ひつじが0匹
ひつじが0匹
⋮

実行している間、無限に表示される

松田くんの作ったコード 4-3 は、while ブロックの中で**変数 count を更新していません**。変数 count の値を変えなければ、繰り返し条件である「count < 3」は常に成立しますから、繰り返しは決して終了しません。このようないつまでも続く繰り返しのことを**無限ループ**と呼びます。



ふー、直りました。

経験者でもうっかり無限ループしちゃうことはまああるからね。無限ループの止め方は覚えておくといいよ。



4章



Column

無限ループを止める方法

JupyterLab で意図せず無限ループに陥ってしまった場合は、次のように解決してみてください。まず、図 4-2 の①を確認しましょう。「*」が表示されていれば、セル内のプログラムがまだ実行中であることを表しています。

無限ループ以外にも、大規模データを扱うなど、処理に時間がかかっている場合にも同じ表示になることがありますが、長時間この状態のままなら無限ループの可能性もあります。②の■ボタンを押すと、実行中のプログラムを強制終了して無限ループを解消することができます。

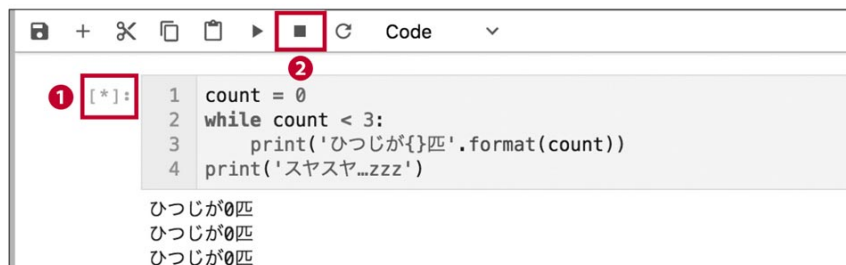


図 4-2 無限ループが発生した JupyterLab の画面

4.1.3 状態による繰り返し

繰り返しの条件を回数ではなく、あるものの状態によって判定することも可能です。たとえば、回数ではなく、眠るまでひつじを数えるのを繰り返す場合には、次のようになるでしょう（コード 4-4）。

コード 4-4 ひつじを数えるのを眠るまで繰り返す

```
1 is_awesome = True
2 count = 0
3 while is_awesome == True:
4     count += 1
5     print('ひつじが{}匹...'.format(count))
6     key = input('もう眠りそうですか? (y/n) >>')
7     if key == 'y':
8         is_awesome = False
9 print('おやすみなさい...')
```

起きている状態ならば

眠ったのでフラグを False にする

実行結果

```
ひつじが1匹...
もう眠りそうですか? (y/n) >>n
ひつじが2匹...
もう眠りそうですか? (y/n) >>n
ひつじが3匹...
もう眠りそうですか? (y/n) >>n
:
ひつじが1024匹...
もう眠りそうですか? (y/n) >>y
おやすみなさい...
```

ポイントは変数 `is_awake` です。この変数は起きている、または、眠っているという状態を表しており、起きている状態ならば `True`、眠った状態になったら（6行目の問いに「y」と答えたら）、`False` が代入されます。このような二者択一の状態を表す情報を**フラグ** (flag) といい、`bool` 型がよく使われます。



bool 型でフラグを表す

ある事柄や状態を二者択一で表すには、`bool` 型を利用する。フラグとなる変数名は、「`is_xxx`」とするのが一般的。

4章



while ブロックの中に if 文があるから、インデントが2段階になっていることにも注意しよう。

4.1.4

繰り返しによるリストの作成



繰り返しの基本が理解できたら、今度はもう少し実践的な使い方を紹介しよう。第2章で出てきたリストを使うよ。

これまでの解説で繰り返しに慣れたら、繰り返しの定石とも言えるプログラムを見ていきましょう。第2章で紹介したコレクション (P078) は、通常、繰り返しと組み合わせて使われます。

まずは繰り返しを利用したリストの作成です (コード 4-5)。

コード 4-5 繰り返しを使って得点リストを作成する

```
1 count = 0 # カウンタ変数
2 student_num = int(input('学生の数を入力 >>')) # 学生の数
```

```

3 score_list = list()  # 空のリストを準備 # 得点リスト
4 while count < student_num:
5     count += 1
6     score = int(input('{} 人目の試験の得点を入力 >>'
                        .format(count)))
7     score_list.append(score)  # 入力された得点を得点リスト
8 print(score_list)            # 入力された学生の数より
9 total = sum(score_list)      # 小さければ繰り返す
10 print('平均点は{}点です'.format(total / student_num))

```

実行結果

```

学生の数を入力 >>3
1人目の試験の得点を入力 >>80
2人目の試験の得点を入力 >>85
3人目の試験の得点を入力 >>75
[80, 85, 75]
平均点は80.0点です

```



3 行目の「score_list = list()」って、今まで見たことのない書き方ですね。

カッコの中に何も値を書かないと、中身のない空のリストが作成されるんだよ (2.5.1 項)。箱だけを準備したって感じかな。



4 行目で、入力された学生の数だけ繰り返す条件となっていることを読み取れるでしょうか。while ブロックの中では、入力された得点をリストへ順番に追加しているので、繰り返しが終了した時点で得点リストが作成されているというわけです。このように、繰り返しを利用して、リストにデータを追加するという処理は非常によく見かけます。

4.1.5 繰り返しによるリスト要素の利用

次のコードは、リストに格納された得点を1つひとつ調べ、合否判定をするプログラムです(コード 4-6)。60 点以上ならば合格、60 点未満だと不合格と表示されます。

コード 4-6 リストの全要素を繰り返し参照する

```
1 scores = [80, 20, 75, 60]
2 count = 0
3 while count < len(scores):
4     if scores[count] >= 60:
5         print('合格')
6     else:
7         print('不合格')
8     count += 1
```

リストの要素数より小さければ
繰り返す

実行結果

合格
不合格
合格
合格

今回のコードでは、リストの要素数だけ繰り返す条件となっています(3 行目)。while ブロックでは、リストの 0 番目から順にアクセスし、その値によって処理が分岐します。



4 行目の if 文なんですが、リストの添え字に変数を使えるんですか？

いいところに気づいたね！ 繰り返し処理の中でリストの要素を参照するには、添え字にカウンタ変数を使うのがポイントだ。



while ブロックの中でリストの添え字としてカウンタ変数を指定すると、繰り返しのたびに自動的に添え字が変化していくため、リストの要素に順番にアクセスすることができます(図 4-3)。このように、繰り返しを利用してリストの先頭から末尾までを参照する処理は「リストを回す」ともいい、非常によく行うので、構文で覚えてしまうとよいでしょう。

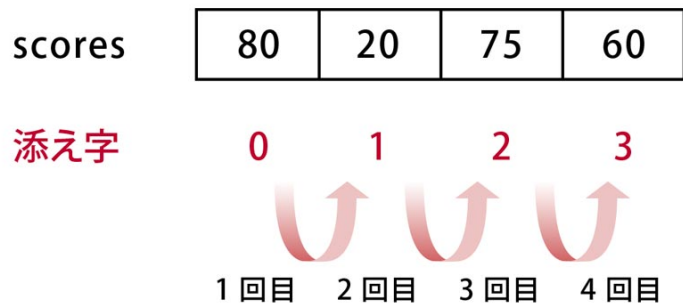


図 4-3 カウンタ変数を利用すると添え字は自動的に変化する



繰り返しを利用してリストの要素を参照する

カウンタ変数 = 0

while カウンタ変数 < len(リスト):

 リスト[カウンタ変数]を使った処理

 カウンタ変数 += 1



少し複雑に思えるかもしれないけれど、これまで紹介した基本的な内容を組み合わせているだけだから、慌てずに。1行ずつ、何をしている処理なのかを復習しながら理解していこう。

4.2 for 文

4.2.1 for 文による繰り返し



リストを while 文で回すのって、難しいなあ。ついつっかりカウンタ変数を増やすのを忘れて、無限ループを起こしちゃうよ。

私は条件式の比較演算子を「<=」にしてしまって、IndexError になっちゃうわ…。



4章

前節の最後では、while 文を使ってリストの要素を順に参照する構文を紹介しました。しかし、紹介した方法は、繰り返しの条件やカウンタ変数の操作を明示的にコーディングする必要があり、無限ループや IndexError を起こしやすいというリスクがあります。

そこで、リストなどのデータの集まりについて、先頭から末尾まで順に参照する場合には、より安全かつエレガントな記述ができる **for 文** がよく利用されます。

コード 4-6 (P169) を for 文で書き直したのが、次のコードです (コード 4-7)。

コード 4-7 for 文でリストの全要素を参照する

```
1 scores = [80, 20, 75, 60]
2 for data in scores:
3     if data >= 60:
4         print('合格')
5     else:
6         print('不合格')
```




ひゃー！ 何かよくわからないけど、繰り返し条件やカウンタ変数を書かなくていいんですか!?

for 文では、繰り返し条件を記述しなくても、自動的にリストの先頭から末尾まで順に繰り返しが行われていきます。また、カウンタ変数を指定する必要もありません。繰り返し条件やカウンタ変数を書く必要がないということは、無限ループや間違った条件での繰り返しを心配する必要もありません。



こんな便利なものがあるなら、何で最初から教えてくれなかったんですか!?

まあまあ落ち着いて。それにはちゃんと理由があるんだ。



for 文を使えば、繰り返しそのものに関わる単純なコーディングミスを減らすことはできます。しかし、繰り返す処理の内容によっては、for 文よりも while 文のほうが向いている場合もあります。while 文で練習したことはムダにならないので安心してください。

4.2.2 for 文の基本構造

for 文も while 文と同様に、直後のブロックを繰り返し実行します。このブロックを **for ブロック** といいます。繰り返す回数は、リストの要素数で決まります。コード 4-7 では、リスト scores の要素数が 4 なので、繰り返しは 4 回行われます。そして、繰り返すたびにリストの要素を先頭から順に取得し、「for」の直後に記述した変数に代入します (図 4-4)。

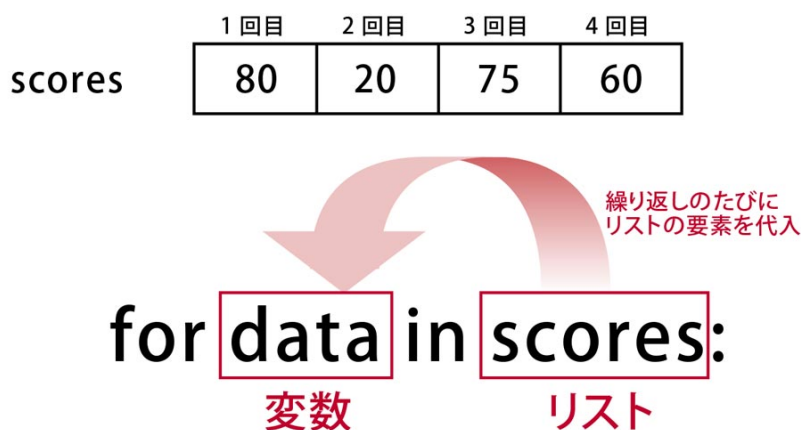


図 4-4 繰り返しのたびにリストの要素が先頭から順に代入される

**for 文でリストの全要素を参照する**

for 変数 in リスト :
 繰り返し処理

リストだけでなく、タプルやディクショナリ、セットなどのコレクションでも for 文は利用可能です。なお、セットのように順序を持たないコレクションの場合、代入される要素の順序は保証されません。

4.2.3 for 文による決まった回数の繰り返し

実は、前節で紹介した while 文の例（コード 4-2、コード 4-5、コード 4-6）のように、決まった回数を繰り返す場合も、for 文を利用するとよりラクに書くことができます（コード 4-8）。

コード 4-8 for 文で決まった回数を繰り返す

```
1 for num in range(3):  
2     print('Pythonは楽しい')
```

3 回繰り返す

実行結果

Pythonは楽しい
Pythonは楽しい
Pythonは楽しい



「range(3)」？ この3のところで回数を指定しているのかしら？

range() はあらかじめ用意された関数で、0 から指定した数より 1 小さい整数までの要素を持つ整数列を作ることができます。たとえば、「range(3)」とすると「0、1、2」という整数からなる整数列が作られます。



ここでいう整数列は、リストのようなものと考えておけば OK だよ。range 関数によって、連番の要素を持つリストが自動的に作られるイメージだ。



range 関数

range(n)

※ 0 以上 n 未満までの範囲の整数列に評価される。

range 関数で作成される整数列もデータの集まりですから、for 文による繰り返しが可能です。なお、コード 4-8 では代入用の変数として num を準備していますが、for ブロックでは使用していません。このように、用意した変数を for ブロックの中で使うかどうかは任意です。



for 文で決まった回数を繰り返す

for 変数 in range(n):

繰り返し処理

※繰り返しは n 回実行される。

4章

4.2.4

while 文と for 文の使い分け



うーん。何だか、for 文があれば while 文はいらない気がしてきました。while 文と for 文ってどうやって使い分けるんですか？

そうだね。ここでちょっとそれぞれの特徴を整理しておこうか。



while 文は開発者が条件式を記述するため、基本的にどのような繰り返しでも表現することができます。しかし、次のような繰り返しでは、for 文のほうがシンプルかつ直感的にわかりやすいコードを書くことができますので、積極的に使っていきましょう。

- データの集まりから要素を取り出してそれに対する処理を繰り返す（コード 4-7）。
- 決まった回数だけ繰り返す（コード 4-8）。

これらは、どちらも繰り返す回数にある程度の目処が立つパターンと言えるでしょう（1 つ目は要素の数、2 つ目は指定した数）。一方 while 文は、コード 4-4 (P166) のように、いつまで繰り返すべきか予測できない場合にその威力を発揮します。繰り返し回数の目処が立つときは for 文、目処が立たないときは while 文を使うと考えるとよいでしょう。



while 文と for 文の使い分け

- while 文 : 繰り返す回数の目処が立たないときに使う。
- for 文 : 繰り返す回数の目処が立つときに使う。



特にリストと for 文の組み合わせによる繰り返しは、非常に高い頻度で利用するよ。章末の練習問題でしっかりと身に付けておこう。

4.3 繰り返しの制御

4.3.1 繰り返しの強制終了



工藤さん、頼まれていたサンプル抽出のプログラムの試作ができました！ これで私もデータサイエンティストとしての一步を踏み出しましたね！

4章

うん、前節までの内容をよく理解して使えているね。本番で使うために、あとひと工夫してみよう。



浅木さんは、大量の年齢データの中から、サンプルとして20代だけを一定数抽出するプログラムを依頼され、次のコードを作成したようです(コード4-9)。

コード4-9 データのまとまりからサンプルを抽出する

```
1 ages = [28, 50, 8, 20, 78, 25, 22, 10, 27, 33] # 対象データ
2 num = 5                                     # 目標の抽出数
3 samples = list()                           # サンプルデータを格納するリスト
4 for age in ages:
5     if 20 <= age < 30:
6         if len(samples) < num:
7             samples.append(age)
8 print(samples)
```

抽出数が目標に達していなければ、リストに追加

実行結果

```
[28, 20, 25, 22, 27]
```

リスト `ages` には、抽出対象となる年齢データが入っており、そこから 20 以上 30 未満のデータを目標の数だけ抽出して、リスト `samples` に格納しています。まだ試作の段階なので、年齢データの数は 10 件で、目標の抽出数は 5 件としています。



結果も問題ないようですが…。まだ改良の余地があるんですか？

コード 4-9 は試作の段階なのでデータ数は 10 件と非常に少ないですが、実際のデータ分析で扱うデータは数万件以上になることも珍しくありません。現在のコードでは、繰り返しの途中で抽出数が目標に達しても、リスト `ages` の末尾まで繰り返しを続けてしまうため、条件が決して成立することのない `if` 文 (6 行目) を延々と実行することになります。



不要な繰り返しはリソースのムダ使いだ。コンピュータに多大な負荷をかけてしまうことになるんだよ。

実は、繰り返しは **break 文** によって途中で強制的に終了させることができます。実際に使用した例を見てみましょう (コード 4-10)。

コード 4-10 目標数に達したら繰り返いを終了する

```
1 ages = [28, 50, 8, 20, 78, 25, 22, 10, 27, 33] # 対象データ
2 num = 5                                     # 目標の抽出数
3 samples = list()                           # サンプルデータを格納するリスト
```

```
4 for data in ages:
5     if 20 <= data < 30:
6         samples.append(data)
7         if len(samples) == num:
8             break
9 print(samples)
```

抽出数が目標に達したので for 文を強制終了

実行結果

[28, 20, 25, 22, 27]

4章

リスト `samples` の要素数が抽出の目標である 5 になれば `break` 文が実行され、`for` ブロックを抜けて繰り返しが終了します。これにより余計な繰り返しをなくすことができます。

4.3.2 繰り返しのスキップ

前項で紹介した `break` 文は、繰り返しを途中で中止するという命令でした。しかし、繰り返しそのものではなく、現在の回のみ処理をスキップして次の回のループを継続したいという場合もあります。それには `continue` 文を使います(コード 4-11)。

コード 4-11 不要な回のループをスキップする

```
1 ages = [28, 50, 'ひみつ', 20, 78, 25, 22, 10, '無回答', 33]
2 samples = list() # サンプルデータを格納するリスト
3 for data in ages:
4     if not isinstance(data, int):
5         continue
6     if data < 20 or data >= 30:
```

数値でないデータはスキップ

目的の条件に合致しないデータはスキップ

```
7         continue
8     samples.append(data)
9     print(samples)
```

実行結果

```
[28, 20, 25, 22]
```

今度の年齢データには、文字列が混在しており、このままでは 20 代のデータだけを抽出することができません。そこでまず、関数でデータが整数かどうか確認し、整数でない場合には `continue` 文が実行され、その回の以降の処理がスキップされます (4 ~ 5 行目)。また、整数データのうち、20 代に合致しないデータはやはり `continue` 文によってはじかれます (6 ~ 7 行目)。

整数かつ 20 代に該当するデータだった場合に、ようやく 8 行目まで処理が到達し、リスト `samples` に追加されるというわけです。



いろいろな条件をクリアしないと目的の処理までたどり着けないですね。たくさんの敵を倒しながらお姫さまを助けに行くゲームみたいです。

そうだね。`continue` を使わずにこの処理を実現しようとする、インデントが深くなってわかりにくいコードになってしまうんだよ。



図 4-5 の左図のように、インデントの深いコードは読みにくいだけでなく、分岐の対応関係を把握しづらいため、処理の内容を理解することが難しくなってしまいます。可読性の低いコードはメンテナンスの難易度が高くなり、不具合を起こしやすくなります。

一方、`continue` 文を使用した図 4-5 の右図は見た目もスッキリし、ひと目で構造を把握することができます。

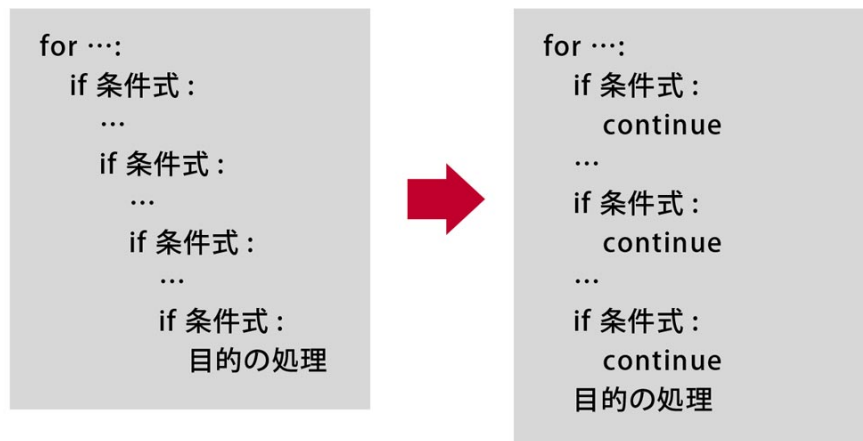


図 4-5 深いインデントは不具合を誘発しやすい



isinstance 関数

isinstance(データ , データ型)

※データがデータ型と一致したら True に置き換わる。

※データ型には int、str、bool などが使用できる (1.3.1 項)。

4.3.3 break 文と continue 文



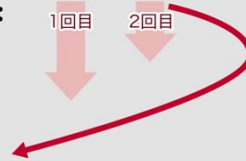
break 文と continue 文は混同しやすいから、最後に整理しておこう。

break 文と continue 文は、何を終了させるかが異なります (図 4-6)。この 2 つの文は、必須というわけではないですが、うまく使うことでコードをきれいにスッキリさせることができます。ぜひ、使い方を身に付けておきましょう。

break 文

(繰り返し自体を中断)

```
data_list = [1, 2, 3]
for num in data_list:
    if num == 2:
        break
    print(num)
```



continue 文

(現在の回だけを中断し、次の回へ)

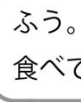
```
data_list = [1, 2, 3]
for num in data_list:
    if num == 2:
        continue
    print(num)
```



図 4-6 2 種類の中断方法



2つの章にわたって学んできた制御構造の解説もこれで終わりでしょ。長い時間お疲れさま。



ふう。いろんなことを教わったなあ。今日はカレーを繰り返し食べてしまいそうですよ。



無限ループにならないように、終了条件はしっかり決めておかなきゃね！

4.4

第 4 章のまとめ

この章では、次のことを学びました。

繰り返し

4
章

- while 文は、条件に従って while ブロック内の処理を繰り返すことができる。
- while 文は、繰り返し回数の目処が立たない繰り返しの適している。
- for 文は、指定したコレクションが持つ要素の数だけ、ブロック内の処理を繰り返すことができる。
- for 文はデータの集まりを順に参照する繰り返しや、一定回数の繰り返しの適している。

繰り返しの中断

- break 文は、繰り返しを強制的に終了する。
- continue 文は、その回のループを終了し、次の回を継続する。

繰り返しのヒント

- 永久に終わることのない繰り返しを無限ループという。
- range 関数で定数回の繰り返しを作成できる。
- 繰り返しによるコレクションの操作は、プログラムの定石である。
- 繰り返しと分岐を組み合わせることで、より実践的なコードを記述することができる。

4.5

練習問題

練習 4-1

次の各コードについて、繰り返しが行われる回数を教えてください。

- (1)

```
1 count = 0
2 while count < 5:
3     count += 1
```
- (2)

```
1 count = 1
2 while count <= 5:
3     count += 1
```
- (3)

```
1 data = [88, 21, 65, 160, 57]
2 count = 0
3 while count < len(data):
4     count += 1
```
- (4)

```
1 for num in range(5):
2     print(num)
```
- (5)

```
1 for item in [88, 21, 65, 160, 57]:
2     print(item)
```
- (6)

```
1 data = [88, 21, 65, 160, 57]
2 for item in data:
3     print(item)
```

(7)

```
1 for item in [88, 21, 65, 160, 57]:
2     if item >= 100:
3         break
4     print(item)
```

(8)

```
1 for item in [88, 21, 65, 160, 57]:
2     if item >= 100:
3         continue
4     print(item)
```

練習 4-2

(1)～(6)のように動作するプログラムを while 文を用いて作成してください。

- (1) 変数 count を任意の値で初期化する。
- (2) 画面に「カレーを召し上がれ」と表示する。
- (3) 画面に「○皿のカレーを食べました」と表示する（○には食べた皿数が入る）。
- (4) 画面に「おかわりはいかがですか？ (y/n)>>」と表示する。
- (5) y が入力されたら、変数 count の値を 1 増やして (3) へ戻る。
- (6) n が入力されたら、「ごちそうさまでした」と表示して終了する。

練習 4-3

「10、9、8、……、2、1、Lift off !」のようなカウントダウンを行うプログラムを for 文と range 関数を用いて作成してください。なお、print 関数に次のような end オプションを付けると、改行せずに文字列を表示できます。

```
1 print('Hello', end='')
2 print('Python')
```

改行しないことを指示するオプション

実行結果

HelloPython

練習 4-4

次の(1)～(3)の内容のプログラムをそれぞれ作成してください。

- (1) 九九の計算をするプログラムを、for 文を用いて作成する。
- (2) (1)のプログラムについて、奇数の段のみ計算するように continue 文を用いて変更する。
- (3) (2)のプログラムについて、掛け算の答えが 50 を超えたらその段の計算を中止し、次の段の計算へ進むように変更する。

練習 4-5

次の(1)～(4)の内容のプログラムを、for 文を用いて作成してください。

- (1) 次の表は、ある日の 8 時から 17 時までの気温を記録したものである。これらの気温をリスト temp に 1 件ずつ入力する。

8 時	9 時	10 時	11 時	12 時	13 時	14 時	15 時	16 時	17 時
7.8	9.1	10.2	11.0	12.5	12.4	14.3	13.8	12.9	12.4

- (2) リスト temp について、1 件ずつ気温を取り出して画面に表示する。
- (3) リスト temp のうち、13 時のデータは計測機器の障害で不正確なことが判明した。データの内容を「N/A」として新しいリスト temp_new に登録し、両方のリストを表示して登録されている内容を比較する。
- (4) リスト temp_new を使って、その日の平均気温を表示する。

練習 4-6

数値 1 の要素を 2 つだけ持つリスト numbers があります。このリストについて、繰り返しによって次の処理を実現するプログラムをそれぞれ作成してください。

- (1) 前の 2 つの要素を足した数値が次の要素の値となるように、numbers に要素を追加していく。ただし、追加する値は 1000 を超過しないものとする。
- (2) (1)のリスト numbers について、要素の値 ÷ 1 つ前の要素の値を要素とした新しいリスト ratios を作成する。
- (3) (2)のリスト ratios について、各要素の値が小数点以下第 3 位までの値になるよう更新する。
(ヒント) 0.12 は 10 倍して int 型に変換後、10 で割ると 0.1 になる。

4.6

練習問題の解答

練習 4-1

(7)以外は 5 回。(7)は 4 回。

4
章

練習 4-2

```
1 count = 1
2 ans = True
3 print('カレーを召し上がれ')
4 while ans == True:
5     print('{}皿のカレーを食べました'.format(count))
6     key = input('おかわりはいかがですか？ (y/n) >>')
7     if key == 'y':
8         count += 1
9     else:
10        ans = False
11 print('ごちそうさまでした')
```

練習 4-3

```
1 for n in range(10):
2     print('{}、'.format(10 - n), end='')
3 print('Lift off !')
```

練習 4-4

(1)

```
1 for i in range(9):
2     for j in range(9):
3         print('{}×{}={} '.format(i+1, j+1, (i+1)*(j+1)))
```

(2)

```
1 for i in range(9):
2     if (i+1) % 2 == 0:
3         continue
4     for j in range(9):
5         print('{}×{}={} '.format(i+1, j+1, (i+1)*(j+1)))
```

(3)

```
1 for i in range(9):
2     if (i+1) % 2 == 0:
3         continue
4     for j in range(9):
5         if (i+1)*(j+1) > 50:
6             break
7         print('{}×{}={} '.format(i+1, j+1, (i+1)*(j+1)))
```

練習 4-5

(1)

```
1 temp = list()
2 for n in range(10):
3     data = float(input('{}個目のデータを入力 >> '.format(n+1)))
4     temp.append(data)
```

(2) ※リスト temp にデータが登録されていることを前提とする。

```
1 for count in range(len(temp)):
2     print('{}時 {}'.format(count+8, temp[count]))
```

(3) ※リスト temp にデータが登録されていることを前提とする。

```
1 temp_new = list()
2 for count in range(len(temp)):
3     if count == 5:
4         temp_new.append('N/A')
5     else:
6         temp_new.append(temp[count])
7 print(temp)
8 print(temp_new)
```

4
章

(4) ※リスト temp_new にデータが登録されていることを前提とする。

```
1 total = 0
2 for data in temp_new:
3     if isinstance(data, float):
4         total = total + data
5 print(total / (len(temp_new) - 1))
```


練習 4-6

(1)

```
1 numbers = [1, 1]
2 data = sum(numbers)
3 count = 2
4 while data <= 1000:
5     numbers.append(data)
6     data = data + numbers[count-1]
7     count += 1
8 print(numbers)
```

最初に追加する値を算出

次に追加する値は、今追加した値と 1 つ前の値との合計

(2) ※(1)の処理を前提とする。

```
1 ratios = list()
2 for count in range(len(numbers)):
3     if count == len(numbers) - 1:
4         break
5     ratios.append(numbers[count+1] / numbers[count])
6 print(ratios)
```

最後の要素には次の要素がないので終了

(3) ※(2)の処理を前提とする。

```
1 for count in range(len(ratios)):
2     ratios[count] = int(ratios[count] * 1000) / 1000
3 print(ratios)
```

第Ⅱ部

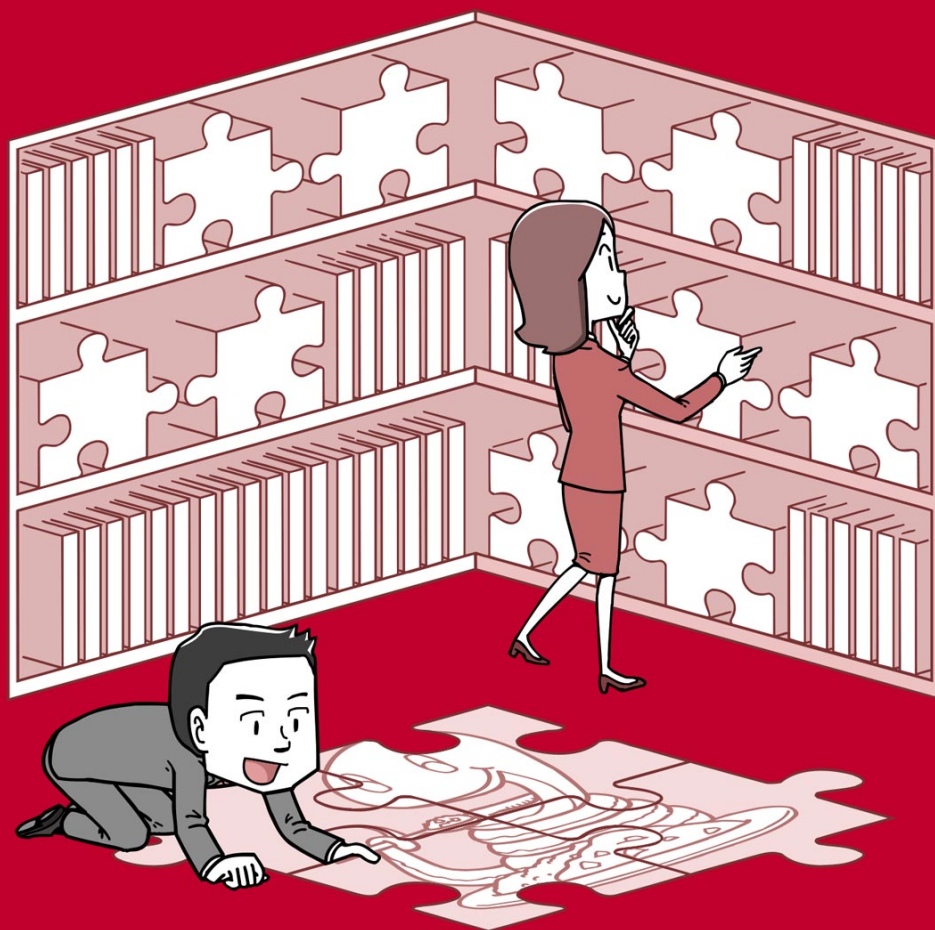
Pythonで部品を組み上げよう

第5章 関数

第6章 オブジェクト

第7章 モジュール

第8章 まだまだ広がる Python の世界



応用構文を学ぼう



うふっ…ふふふっ…うふふふっ…♪

ど、どうしたんですか先輩。気持ち悪い笑い方して…。



松田くん、あなたまだ気づかないの？ 順次・分岐・繰り返しをマスターした今、私たちに不可能はないの。つまり卒業したのよ、Python をッ！

確かにいろいろ勉強したけど…。でも、ちゃんとしたプログラムを作ろうとするなら、まだまだ大変なんじゃないですかね？



ま、そこで見えていなさい。この Python マスター浅木が、統計ソフトだろうが人工知能だろうがチャチャッと作ってみせるわよ。うふふ♪

私たちは第Ⅰ部で Python の基礎となる文法を学びました。今後は、これらの基礎的な道具立てをいかに効率よく上手に応用していくかが鍵となります。第Ⅱ部では、実用的な開発に欠かせない「部品」という考え方とその使い方を学んでいきましょう。

第5章

関数

これまで私たちは、`print` 関数や `input` 関数など、`Python` が備えるさまざまな関数を利用してきました。このようなあらかじめ用意された既存の関数を利用するだけでなく、開発者が自らオリジナルの関数を作ることもできます。この章では、本格的なプログラム開発に欠かせない関数の作り方と使い方について学んでいきましょう。

CONTENTS

- 5.1 オリジナルの関数
- 5.2 引数と戻り値
- 5.3 関数の応用テクニック
- 5.4 独立性の破れ
- 5.5 第5章のまとめ
- 5.6 練習問題
- 5.7 練習問題の解答

5.1

オリジナルの関数

5.1.1 関数の必要性とメリット



おや浅木さん。どうしたんだい？ そんな難しい顔をして。

あ、工藤さん。繰り返しの復習をしようと思って、試験の得点を管理するプログラムを作っていたんですが、だんだんどこで何をやっているのかわからなくなってきてしまって…。



浅木さんの作ったプログラムを見てみましょう（コード 5-1）。

コード 5-1 見通しの悪いプログラム

```
1 student_list = ['浅木', '松田']
2 count = 0
3 for student in student_list:
4     print('{}さんの試験結果を入力してください'.format(student))
5     network = int(input('ネットワークの得点？ >>'))
6     database = int(input('データベースの得点？ >>'))
7     security = int(input('セキュリティの得点？ >>'))
8     if student == '浅木':
9         asagi_scores = [network, database, security]
10        asagi_avg = sum(asagi_scores) / len(asagi_scores)
```

```
11     else:
12         matsuda_scores = [network, database, security]
13         matsuda_avg = sum(matsuda_scores) / len(matsuda_scores)
14     print('浅木さんの平均点は{}です'.format(asagi_avg))
15     print('松田さんの平均点は{}です'.format(matsuda_avg))
```

実行結果

浅木さんの試験結果を入力してください

ネットワークの得点? >>90

データベースの得点? >>88

セキュリティの得点? >>92

松田さんの試験結果を入力してください

ネットワークの得点? >>50

データベースの得点? >>40

セキュリティの得点? >>60

浅木さんの平均点は90.0です

松田さんの平均点は50.0です



やりたいことが増えてきたら、プログラムが複雑になってしまっ
て、何が何だかわからなくなっちゃいました。エラーも次
から次に出てくるし…。ぐすん。

ちゃんと動いているじゃないか。がんばって作ったんだね。でも、
実際の開発現場ではもっと複雑なプログラムも作るんだよ。



コード 5-1 は、これまで本書に登場したものと比較すると多少複雑ですが、業務で開発するプログラムは、もっと込み入った作りになります。

たとえば、入力されたデータを厳しくチェックしたり、出力結果の見栄えをよくしたり、さらには平均値の計算より高度なデータ分析を行ったりするでしょう。そうなれば、分岐や繰り返しの構造が入り組んだ複雑なプログラムになることは

想像に難くありません。プログラムの行数も、数千～数万行に及ぶでしょう。

そのような巨大なプログラムをこれまでどおりの作り方で作成するとどうなるでしょうか。「計算がおかしいので直してほしい」「入力チェックのルールをもっと細かくしてほしい」という要望に応えるために、どこを修正したらよいか探すだけでも大変な作業になります。



ちょっと待ってください。そんなこと言われても、プログラムが長くなるのは仕方ないじゃないですか。

たとえ長くなっても、どこでどんな処理をしているのか、見通しをよくすることはできるんだよ。工夫次第で、こんな感じにスッキリ書けるんだ。



スッキリと、見通しをよくしたプログラムを見てみましょう(コード 5-2)。

コード 5-2 見通しがよくなったプログラム

```
1 # 得点を入力
2 asagi_scores = input_scores('浅木')
3 matsuda_scores = input_scores('松田')
4 # 平均点を計算
5 asagi_avg = calc_average(asagi_scores)
6 matsuda_avg = calc_average(matsuda_scores)
7 # 結果を出力
8 output_result('浅木', asagi_avg)
9 output_result('松田', matsuda_avg)
```



何これ、めちゃくちゃスッキリしてて、意味もわかりやすい！
なるほど、input_scores や calc_average なんていう関数も
Python にはあったんですね！ やだなあ…あるなら早く教えて
くださいよお。

ははは。残念ながらそんな関数はない。でも、「ないなら作れ
ばいい」んだよ。



これまで私たちは、関数とは「Python が準備してくれるもので、開発者はそ
れを呼び出すだけ」と考えてきました。しかし、実は私たちも創意工夫次第で、
オリジナルの関数を作ることができます。もし、「キーボードから点数を入力さ
せる input_scores 関数みたいなものがあっていいな」と思ったら、自分の手で
作れるのです。

5
章**関数は使うだけでなく、作れる**

- これまでのイメージ
関数は Python が準備してくれるもの。
- これからのイメージ
関数は Python も準備してくれているが、自分たちでも作れる。

そして、自分でも関数を作ることができるようになると、プログラムを複数の
部品に分けることができるようになります。たとえば、図 5-1 のように、1 つの
長い処理を機能によってある程度分割して複数の関数に分け、それぞれの処理を
担当させることが可能になります。

このように、1 つのプログラムを複数の部品に分けることを**部品化**といいます。

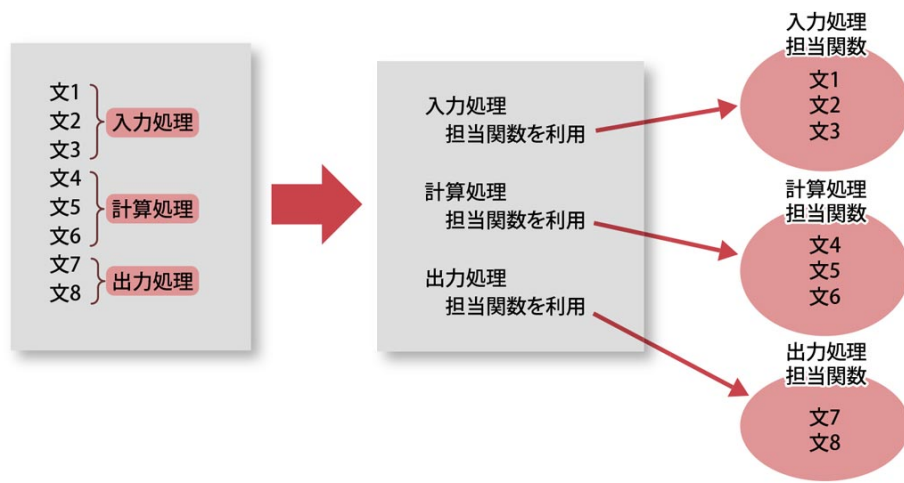


図 5-1 関数による部品化

プログラムを部品化することによって、どの処理がどのような機能を担っているのか、プログラムをスッキリと見通せるようになり、全体を把握するのがラクになります。

また、「結果の出力がおかしい」という不具合や、「計算方法を変更したい」という要望にも、その機能を担当する関数だけを修正すれば対応できるので、ソースコードを変更する範囲を限定することができます。

さらに、関数は何度でも呼び出すことができます。何度も行う処理は関数にしておくことで、全体のコード行数を減らし、プログラムをコンパクトにすることが可能になります。



部品化のメリット

- ・ プログラム全体の見通しがよくなり、処理を把握しやすくなる（プログラムの可読性が向上する）。
- ・ 機能ごとに関数を記述するため、修正範囲を限定できる（プログラムの保守性が向上する）。
- ・ 何度も使う機能を関数にまとめることで、プログラミングの作業効率上がる。

5.1.2 関数を使うための 2 ステップ



世界に 1 つだけの「My 関数」を作れちゃうなんてワクワクしますね。

それじゃあ、まずは簡単な関数を作って動かす体験をしてみよう。



まずは、とてもシンプルな関数を作成して動かしてみましょう。次のコード 5-3 は、呼び出すと画面に「こんにちは。工藤です。」という文字列を表示するだけの関数です。入力して、実行してみてください。

5 章

コード 5-3 hello 関数の定義

```
1 def hello():  
2     print('こんにちは。工藤です。')
```

インデント



あれ？ 何も表示されませんね？

この 2 行は「こういう内容の hello 関数を作って利用可能にして」と Python に指示しただけだからね。「hello 関数を動かして」という指示は別にする必要があるんだ。



それでは、利用可能になった hello 関数をさっそく使ってみましょう。次のコード 5-4 を入力して実行してみてください。

コード 5-4 hello 関数の呼び出し

```
1 hello()
```

実行結果

こんにちは。工藤です。

このように、私たちが作成したオリジナルの関数を使うためには、コード 5-3 で記述したような**定義** (definition) と、コード 5-4 で行ったような**呼び出し** (call) の 2 つのステップが必要です。



関数の定義と呼び出し

- ステップ 1: 関数の定義
呼び出されたらどのような動作を行うかを記述し、名前を付ける。
- ステップ 2: 関数の呼び出し
関数の名前を記述して関数を呼び出す。定義された動作が実行される。

5.1.3

関数定義と呼び出し



シンプルな関数定義の構文を紹介しておこう。「def」文は英語の「define (定義する)」に由来しているんだ。

if や while と同じく、英語の意味そのままと思えば OK なのね。





シンプルな関数の定義

`def 関数名 ():`

処理

※処理はインデントして記述する。

関数名の右にある () の部分は、より複雑な関数を定義するときに使います。現時点では、単に「関数名の右には () を書く」と丸暗記してかまいません。

構文の 2 行目以降は、この関数が呼び出されたときに実行する処理を記述します。この部分を**関数ブロック** (function block) といい、if 文や for 文のブロックと同様、字下げ(インデント)してブロックの範囲を定めます。コード 5-3 では、print 関数による表示処理が 1 行だけのブロックでしたが (P199)、複数行を記述したり、条件分岐や繰り返しなどのブロックをネストさせたりすることももちろん可能です。

また、hello 関数のようなシンプルな関数の場合は、次のような構文で呼び出すことができます。

5
章



シンプルな関数の呼び出し

関数名 ()



オリジナルの関数を作るのって、思ってたより簡単ですね！

そうだね。でも 1 点だけ大事な注意点があるんだ。



Python では、ある関数がすでに存在している状態で、まったく同じ名前の関数を定義することが許されています。このように、関数などの名前が重複することを一般的に**名前の衝突**といいます。**関数名が衝突すると、関数の定義が上書きされます**。たとえば、うっかり `input` という名前の関数を自分で定義してしまうと、Python が標準で提供している `input` 関数が使えなくなってしまうのです。



えっ…。それってかなりマズくないですか。エラーとかにはならないんですか？

残念ながらエラーにはならないんだ。だからこの「名前の衝突は怖い」という感覚は大事にしておいてくれ。



関数名の衝突による上書き

すでに定義されている関数と同じ名前を付けると、以前の関数は呼び出せなくなる。

5.1.4 ローカル変数と独立性



よし！ 試験の得点を入力する関数を作ってみよう！

```
1 def input_scores():
2     print('浅木さんの試験結果を入力してください')
```

ここで浅木さんの手が止まってしまった



まず、メッセージを表示して、と…。あれ？ これじゃ、私の得点しか入力できないじゃない。

浅木さんが作りかけた関数では、「浅木さんの試験結果を…」と表示しているので、浅木さん 1 人の得点入力にしか使えません。もし得点を入力したい人が 1,000 人いたとしたら、似たような関数を 1,000 個も作る必要があります。



うーん、こうすればいいんじゃないかな？

松田くんが書いたコードを見てみましょう (コード 5-5、コード 5-6)。

コード 5-5 名前を表示するつものの input_scores 関数

5章

```
1 def input_scores():  
2     name = ''  
3     print('{}の試験結果を入力してください'.format(name))
```

空の変数 name を準備

コード 5-6 input_scores 関数内の変数 name に代入するつもり

```
1 name = '浅木'  
2 input_scores()  
3 name = '松田'  
4 input_scores()
```

input_scores 関数内の変数 name に代入したつもり

input_scores 関数内の変数 name に代入したつもり

実行結果

の試験結果を入力してください

の試験結果を入力してください



あれ？ 名前が表示されないぞ。

松田くんは、関数の中で変数 name を準備しておき、呼び出し側でその値を変更するというアイデアを思いつきましたが、残念ながらうまくいきませんでした。なぜなら、**関数内で準備された変数は、その関数の中でしか読み書きできない**というルールがあるからです。このような変数の性質を**ローカル変数の独立性**といいます。



ローカル変数の独立性

- ・ 関数の中で定義された変数は、その関数の中でしか使えない。
- ・ その関数の外やほかの関数の中に偶然同じ名前の変数があったとしても、まったく無関係な別の存在として扱われる。

このルールは、「1 つひとつの関数は独立した 1 つの世界である」という考えに基づいています。関数を呼び出す側は、「関数とは呼び出せばきちんと仕事をしてくれるもの」という前提で関数を呼び出します。関数がどのような変数を準備して、どのように処理をするかまでは気にしません。

これは裏を返せば、外の世界から関数内の変数に直接アクセスする権限がないことを意味します。もし、外側から内部の変数に対して不用意にアクセスされてしまうと、関数は任された仕事を正しく処理するという責任を果たすことが難しくなってしまうからです。関数は独立しているからこそ、関数を呼び出す側は安心してその仕事を任せることができるのです (図 5-2)。



いったん仕事を任せたら、あれこれムダな口は挟まない工藤さんと同じですね。

キミたちを信頼している証拠だよ。…というか、僕自身が細かく指示されるのは苦手だからね！



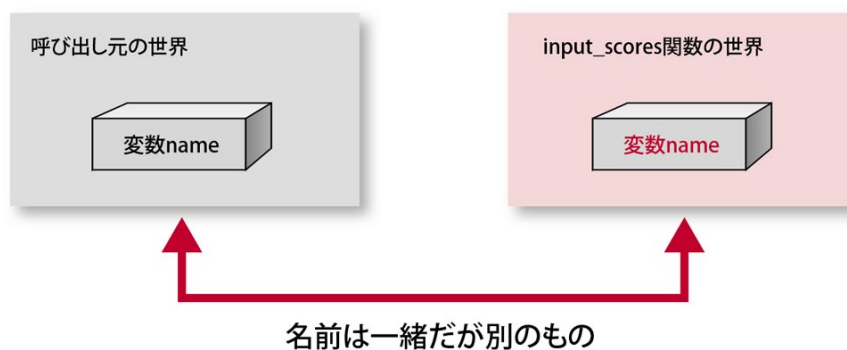


図 5-2 呼び出し元と呼び出し先は別の世界

5章

なお、関数の中で定義された変数のことを**ローカル変数**といいます。ローカル (local) とは、「特定の、地域限定の」という意味を持つ英単語です。



関数同士は「越えられない壁」で仕切られているから、外の世界からはローカル変数を触れないということか。

実は、その壁を越えて情報をやりとりする方法が2つ存在するんだ。次節から順に紹介していこう。



5.2

引数と戻り値

5.2.1 引数

前節で紹介したように、関数は「独立した 1 つの世界」ですから、その内部にあるローカル変数を関数の外部から読み書きすることは原則としてできません。

しかし、関数に仕事をしてもらうためには、必要なデータをやりとりしなければならない場合もあります。そのようなときには、ある関数を呼び出す際に外部からデータを送り込む方法として、^{ひきすう}引数 (argument) を使うことができます (図 5-3)。

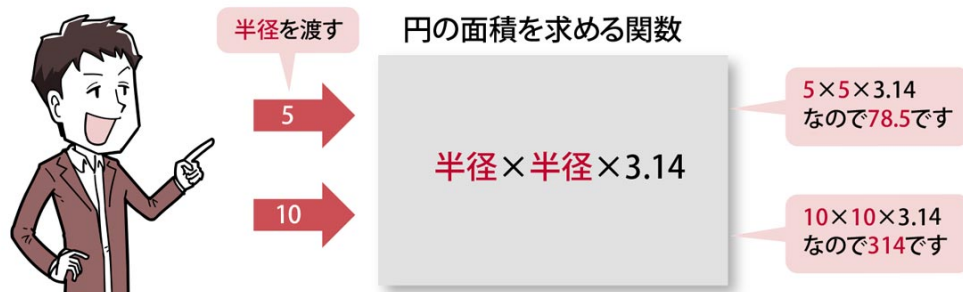


図 5-3 関数を呼び出すときに外からデータを送り込むことができる

引数を利用するには、前節で紹介した関数の定義と呼び出しの方法を少し変更する必要があります。実際に、コード 5-3 の hello 関数 (P199) を変更してみましょう (コード 5-7)。

コード 5-7 引数を受け取る hello 関数

```
1 def hello(name):
2     print('こんにちは。{}です。'.format(name))
```

呼び出し時に渡されるデータを受け取る変数

コード 5-8 引数を渡しながら hello 関数を呼び出す

```
1 hello('浅木')
2 hello('松田')
```

呼び出しと同時に '浅木' を渡す

呼び出しと同時に '松田' を渡す

実行結果

こんにちは。浅木です。
こんにちは。松田です。

5章



おおっ！ ちゃんと名前も表示されましたよ。

まず hello 関数の定義に注目してください(コード 5-7)。関数名に続くカッコの中に、「name」と書かれています。これは、この関数が呼び出された際、渡されたデータを name という引数で受け取り、関数内で扱っていくことを表明しています。



引数もローカル変数の1つだから、関数を呼び出すとき以外にはもちろんアクセスできないよ。

次に関数の呼び出し側も見てみましょう(コード 5-8)。これまでは単に「hello()」とだけ記述すれば呼び出すことができましたが、新しい定義では、hello 関数は引数の要求を表明していますから、関数名の後ろのカッコ内で「浅木」「松田」というデータを渡しています。

このように、関数で引数を使うには、関数定義でデータを受け取ること、呼び出し側ではデータを渡すことをきちんと表明する必要があります(図 5-4)。



引数という受付係を通せば、関数はデータを受け取ってくれるのね。

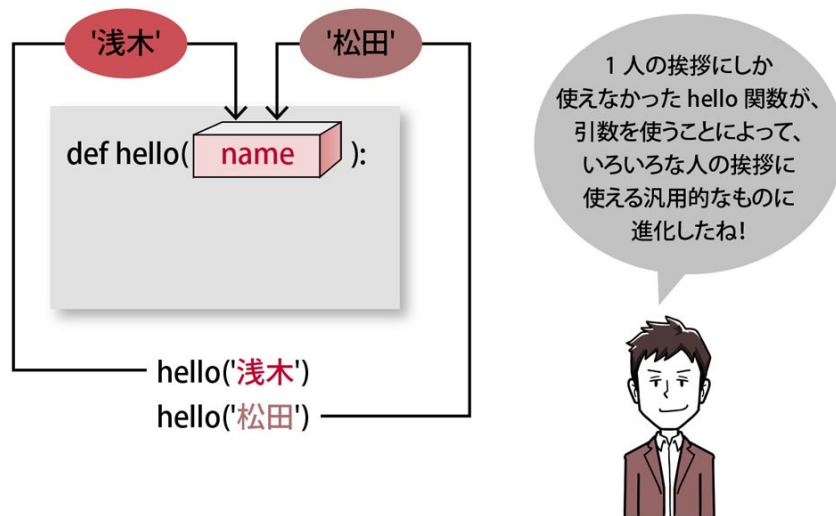


図 5-4 関数に引数を渡す

なお、呼び出し側で渡すデータ（「浅木」「松田」など）も、それを受け取る関数側のローカル変数（name など）も、どちらも引数と総称されます。両方を厳密に区別したい場合は、渡すデータを**実引数**、そのデータを受け取る変数を**仮引数**と呼び分けます。



具体的な実際のデータだから「実引数」なのね。

5.2.2 複数の引数を渡す

複数の引数を渡すことも可能です。次のコード 5-9 とコード 5-10 は、3 つの引数を受け取る profile 関数を定義し、呼び出している例です。

コード 5-9 複数の引数を受け取る profile 関数

```
1 def profile(name, age, hobby):
2     print('私の名前は{}です。'.format(name))
```

```
3 print('年齢は{}歳です.'.format(age))
4 print('趣味は{}です.'.format(hobby))
```

コード 5-10 複数の引数を渡しながら profile 関数を呼び出す

```
1 profile('浅木', 24, 'カフェ巡り')
```

実行結果

私の名前は浅木です。

年齢は24歳です。

趣味はカフェ巡りです。

5章

profile関数の呼び出し時に指定した、「浅木」「24」「カフェ巡り」の3つのデータがそれぞれ変数 name、age、hobby に引き渡されます(コード 5-10)。引数は、呼び出し時に記述した順に従って受け取られるため、渡す順番を間違えないように注意しましょう(図 5-5)。たとえば、「profile('カフェ巡り','浅木',24)」と書いてしまうと、関数が意図する動作とは異なる動きをしたり、エラーになったりする可能性があります。

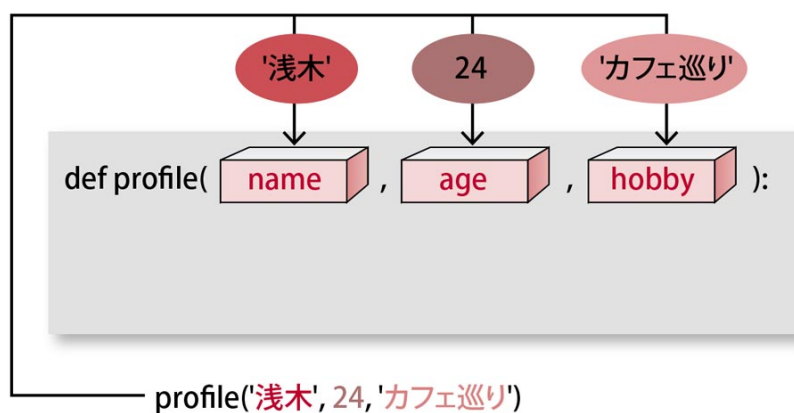


図 5-5 複数の引数を渡す



これで引数に関しては概ねマスターだ。念のため構文を確認しておこう。



引数を利用する関数の定義

```
def 関数名 ( 引数 1, 引数 2, ...):  
    処理
```



引数を利用する関数の呼び出し

```
関数名 ( 引数 1, 引数 2, ...)
```



工藤さん見てください！ 引数を使って calc_average 関数を作ってみました！ コード 5-2 (P196) から呼び出せますよね。

うん、よくできてるじゃないか。引数 scores はリストを前提として、何科目分でも受け取れるように工夫したんだね。



浅木さんが作った関数を見てみましょう (コード 5-11)。

コード 5-11 リストの平均を求める calc_average 関数

```
1 def calc_average(scores):  
2     avg = sum(scores) / len(scores)  
3     print('平均点は{}です'.format(avg))
```

リストを受け取る引数

受け取ったリストに格納されている得点の平均を求める



引数として引き渡せるデータ

引数には、数値や文字列はもちろん、コレクションも引き渡すことができる。

5.2.3

戻り値



calc_average 関数は無事作れたんですが、input_scores 関数がうまくいなくて…。

5
章

浅木さんが作った、得点を入力する input_scores 関数と、平均を求める calc_average 関数を併せて見てみましょう(コード 5-12)。

コード 5-12 input_scores 関数と calc_average 関数

```

1  def input_scores(name): 得点入力を担当する関数
2      print('{}さんの試験結果を入力してください'.format(name))
3      network = int(input('ネットワークの得点? >>'))
4      database = int(input('データベースの得点? >>'))
5      security = int(input('セキュリティの得点? >>'))
6      scores = [network, database, security]
7      入力された得点をローカル変数 scores にリストとして代入
8  def calc_average(scores): 平均算出を担当する関数
9      avg = sum(scores) / len(scores)
10     print('平均点は{}です'.format(avg))

```

これらの関数を次のコード 5-13 によって呼び出すと、次のようなエラーが出てしまいます。

コード 5-13 input_scores 関数と calc_average 関数を呼び出す

```
1 input_scores('浅木')
2 calc_average(scores) ) ここでエラーが発生
```

実行結果

NameError: name 'scores' is not defined



うーん、input_scores 関数の中になら変数 scores はあるけど、呼び出し元にはないですもんね。

このエラーの原因も、ローカル変数の独立性 (P204) です。input_scores 関数では、ユーザーがキーボードで入力した点数をローカル変数 scores にリストとして代入しています (コード 5-12 の 6 行目)。しかし、その情報はローカル変数であるがゆえに input_scores 関数の外からは手が出せず、呼び出し元や calc_average 関数の中では使用することができないのです。



引数とは逆に、「関数の中から外に情報を渡す」ことができれば、何とかなりそうなんですけど…。

そこで「壁」を越える第 2 の方法、戻り値の登場だよ。



引数は、「呼び出し元から関数へデータを渡す」しくみでした。それとは逆に、「関数から呼び出し元へデータを渡す」しくみが**戻り値** (return value) です (図 5-6)。

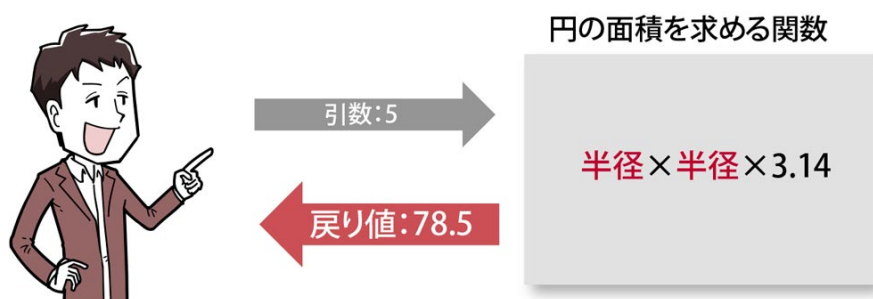


図 5-6 戻り値により呼び出し元へデータを渡す

戻り値を返す関数を定義するには、次の構文を用います。

5章



引数と戻り値を利用する関数の定義

```
def 関数名 ( 引数 1, 引数 2, ...):
```

```
    処理
```

```
    return 戻り値
```

※引数は複数の指定が可能だが、戻り値は1つのみ。

関数定義の最終行に登場しているのは **return 文** といいます。その関数の実行を終了するとともに、return の後ろに記述された変数や値を呼び出し元に返す役割を持っています。戻り値には、数値や文字列のほか、リストやディクショナリなどのコレクションを指定することが可能です。

ただし、引数は複数の利用が許されていますが、**戻り値として返せる値は1つだけ**ということに注意してください。



まずは簡単な例で動きを確認してみよう。引数で渡された値を足し算して結果を返す関数だよ。

引数を足し算して戻り値として返す関数を `plus` という名前で定義します (コード 5-14)。

コード 5-14 足し算の結果を返す `plus` 関数

```
1 def plus(x, y):  
2     answer = x + y  
3     return answer
```

ローカル変数 `answer` の値を返す

この関数を呼び出して戻り値を受け取るには、次のコード 5-15 のように記述します。

コード 5-15 `plus` 関数の呼び出し

```
1 answer = plus(100, 50)  
2 print('足し算の答えは{}です'.format(answer))
```

変数 `answer` に戻り値が代入される

実行結果

足し算の答えは150です

5.2.4 関数呼び出しの正体



あれっ？ これだと、変数 `answer` に関数が入っちゃうんじゃないですか？

松田くんは、コード 5-15 の 1 行目、`plus` 関数の呼び出し方について疑問があるようです。



不思議な感じがするのはわかるよ。この謎を解くヒントは第1章にあるんだ。

私たちは第1章で、式や演算子による評価について学びました。その際、「input 関数などの命令実行も、実は評価されて実行結果に置き換わる」と紹介したことを思い出してください(P058)。実は、関数呼び出しの際に関数名の右側に記述する () は、**関数呼び出し演算子**という演算子であり、次のような働きをすることが定められています。

5章



関数呼び出し演算子の働き

- ・評価されると、左カッコの直前に記述された関数を呼び出す。その際、左カッコと右カッコの間に記述された引数を呼び出し先に引き渡し、実行完了を待つ。
- ・呼び出した関数の実行が完了すると、返ってきた戻り値に「化ける」。



えーっ！ 関数呼び出しに書くカッコって、演算子だったんですか!? 演算子って、+ や / みたいな1文字や2文字の記号だけかと思ってましたよ。

周囲の関数名や引数を巻き込んで戻り値に化ける。立派な演算子さ。



このルールに従い、コード 5-15 の「plus(100, 50)」の部分は、関数の実行後に戻り値である 150 に置き換わり、「answer = 150」という代入の文になります。結果として、変数 answer に戻り値が代入されることになるのです(図 5-7)。

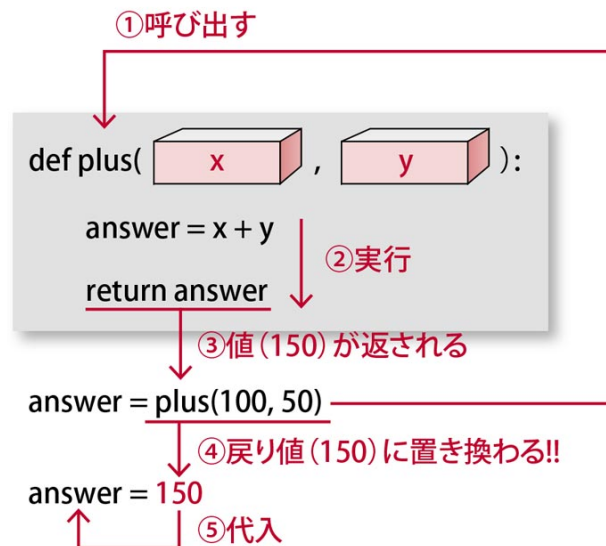


図 5-7 関数呼び出しが評価され、戻り値に「化ける」



引数と戻り値を利用する関数の呼び出し

戻り値を受け取る変数名 = 関数名 (引数 1, 引数 2, ...)



それともう 1 点、コード 5-14 の変数 answer とコード 5-15 の変数 answer は、まったく違う別の変数だということが読み取れているかな？

plus 関数内で使っている変数(コード 5-14、P214)と、それを受け取る変数(コード 5-15、P214)の名前がともに answer となっています。前節で「ローカル変数の独立性」を学んだみなさんは、これらが同じ変数ではなく、それぞれ独立した別の変数であることを理解できるでしょう。

今回の例では同じ変数名ですが、戻り値の変数と受け取る変数の名前を揃える必要はありません。ただし、変数名は格納している値の意味が想像できるわかり

やすい名称が好ましいとされます (P049)。そのため、計算結果を格納する変数が呼び出し元でも呼び出し先でも同じ `answer` と名付けられることは自然なことと言えるでしょう。



Column

「空っぽ」を意味する None

関数定義の末尾に明示的な `return` 文が登場しない場合は、「`return None`」という記述をしたものとみなされます。**None** は、Python の世界では「何もない状態」「空っぽ」を意味するために準備された特殊な値です。`print` 関数など、返すべき戻り値が特にない関数は `None` を返しています。

**5
章**

5.2.5 関数の連携



…よしっ。試験結果を計算するプログラム、無事完成しました！

浅木さんのプログラムはどのようになったのでしょうか (コード 5-16、コード 5-17)。

コード 5-16 さまざまな機能を担当する関数の定義

```
1 def input_scores(name): 得点入力を担当する関数
2     print('{}さんの試験結果を入力してください'.format(name))
3     network = int(input('ネットワークの得点? >>'))
4     database = int(input('データベースの得点? >>'))
5     security = int(input('セキュリティの得点? >>'))
```

```

6     scores = [network, database, security]
7     return scores
8
9     def calc_average(scores): 平均算出を担当する関数
10         avg = sum(scores) / len(scores)
11         return avg
12
13     def output_result(name, avg): 平均点の出力を担当する関数
14         print('{}さんの平均点は{}'.format(name, avg))

```

コード 5-17 試験結果を入力して平均点を出す

```

1     # 浅木と松田の得点入力
2     asagi_scores = input_scores('浅木')
3     matsuda_scores = input_scores('松田')
4     # 平均点を計算
5     asagi_avg = calc_average(asagi_scores)
6     matsuda_avg = calc_average(matsuda_scores)
7     # 結果を出力
8     output_result('浅木', asagi_avg)
9     output_result('松田', matsuda_avg)

```

input_scores 関数の
戻り値を引数に渡す

calc_average 関数の戻
り値を引数に渡す

完成した浅木さんの計算プログラムのように、「ある関数を呼び出して得られた結果を、引数として別関数に渡して処理させる」というパターンは非常に多く見られます。また、関数の中でさらに別の関数を呼び出すということも頻繁にあるでしょう。



えっ？ 関数の中で、別の関数を呼び出したりもできるんですか？

もちろん！ 今までだって、関数の中で print 関数を呼び出していただろう（コード 5-16 など）？



実務などである程度の規模のプログラムを作ろうとすると、どのような関数をどのくらい作べきか、どのように引数と戻り値をやりとりすべきかといったプログラム全体の作り、設計を考える必要が出てきます。慣れるまでは少し大変と感じるかもしれませんが、試行錯誤を繰り返すことで無意識に実践できるようになります。

5章



そのためにも、シンプルなものを手始めに、これからも着実に練習を繰り返していこう。

はい！



5.3

関数の応用テクニック

5.3.1

暗黙のタプルによる複数の戻り値



おめでとう。定義と呼び出し、そして引数と戻り値をマスターした今、関数についてはほとんど制覇したも同然だよ。ここからは、いくつかの便利な応用テクニックを紹介していこう。

まずは、次のコード 5-18 の `plus_and_minus` 関数が返す戻り値に着目してください。

コード 5-18 2つの戻り値を返す？

```
1 def plus_and_minus(a, b):  
2     return a + b, a - b  
3  
4 next, prev = plus_and_minus(1978, 1)
```

和と差の2つの値が戻り値のように見える



なるほど！ `return` 文の後ろには、カンマ区切りで複数の戻り値を指定できるんですね！

まあ、そう見えちゃうだろうけど、`return` が返す戻り値は常に1つなんだ (P213)。



この return 文が返しているものの正体を見破るには、第2章で学んだタプルについて思い出す必要があります。タプルを定義するための丸カッコは、省略可能というルールがあるため(P104)、カンマで区切られた複数の値はタプルとして扱われます。結果として、コード5-18は、次のコード5-19のように、「1つのタプルを return 文で戻して、アンパック代入(P052)している」にすぎません。

コード5-19 戻り値のタプルをアンパック代入

```
1 def plus_and_minus(a, b):
```

```
2     return (a + b, a - b)
```

```
3
```

```
4 (next, prev) = plus_and_minus(1978, 1)
```

要素数2のタプルを1つ返しているだけ

返ってきたタプルをアンパック代入しているだけ

5
章



アンパック代入…？ ああ、変数をカンマで区切ってまとめて代入する方法ですね。

そう。カンマで区切られた値はタプルを意味するから、アンパック代入とはつまり、タプル同士の代入にすぎないんだよ。



タプルの丸カッコを省略すると、処理の意味が紛らわしくなるため原則的には推奨しませんが(P104)、関数の戻り値として記述する場合にのみ限定すると、省略した記法を用いるのが一般的です。

5.3.2 デフォルト引数



ところで、松田くんの食生活を独自にリサーチした結果をプログラムで表現してみたんだが…。

工藤さんのプログラムを見てみましょう(コード 5-20、コード 5-21)。

コード 5-20 指定された 3 食を表示する eat 関数

```
1 def eat(breakfast, lunch, dinner):  
2     print('朝は{}を食べました'.format(breakfast))  
3     print('昼は{}を食べました'.format(lunch))  
4     print('晩は{}を食べました'.format(dinner))
```

コード 5-21 松田くんの食生活を出力する

```
1 print('8月1日')  
2 eat('トースト', 'おにぎり', 'カレー')  
3 print('8月2日')  
4 eat('納豆ごはん', 'ラーメン', 'カレー')  
5 print('8月3日')  
6 eat('バナナ', 'そば', '焼肉')  
7 print('8月4日')  
8 eat('サンドウィッチ', 'しゅうまい弁当', 'カレー')
```



何これ!? 晩ご飯はカレーばかりじゃない!

カレーは無敵だからね。今月は週に 6 日はカレー食べてるし、もう、eat 関数を呼び出すとき、最後に「カレー」って書くのが面倒なくらいだよ。



食生活として好ましいか否かは別として、どうやら eat 関数の仮引数 dinner には、ほとんどのケースで「カレー」が指定されるようです。このように、ある仮引数に指定される値が概ね想定される場合、次のような **デフォルト引数** (default argument) というしくみを使って関数を定義するとよいでしょう。



引数にデフォルト値を指定する関数の定義

```
def 関数名 ( 仮引数名 = デフォルト値 , ...):
```

処理

```
return 戻り値
```

※関数呼び出しで実引数が指定されない場合は、デフォルト値が指定されたものとみなされる。

※デフォルト値を指定した以降の仮引数もデフォルト値の指定が必須となる。

5
章

これを利用して、先ほどの eat 関数を呼び出す手間を低減してみましょう (コード 5-22、コード 5-23)。

コード 5-22 指定された 3 食を表示する eat 関数 (デフォルト値を利用)

引数 dinner にデフォルト値を設定

```
1 def eat(breakfast, lunch, dinner='カレー'):  
2     print('朝は{ }を食べました'.format(breakfast))  
3     print('昼は{ }を食べました'.format(lunch))  
4     print('晩は{ }を食べました'.format(dinner))
```

コード 5-23 松田くんの言うまでもない食生活を実出力する

```

1 print('8月1日')
2 eat('トースト', 'おにぎり')
3 print('8月2日')
4 eat('納豆ごはん', 'ラーメン')
5 print('8月3日')
6 eat('バナナ', 'そば', '焼肉')
7 print('8月4日')
8 eat('サンドウィッチ', 'しゅうまい弁当')

```

引数 dinner の実引数を省略

引数 dinner の実引数を省略

引数 dinner の実引数を省略



あら便利ね♪ 私の場合、朝は必ずトーストだから、こんな eat 関数にすればいいわね。

```

1 def eat(breakfast='トースト', lunch, dinner):

```

第 1 引数にだけデフォルト値を設定したいが…

しかし残念ながら、Python のデフォルト引数には次のような制約があるため、浅木さんの eat 関数を定義したこのコードはエラーになってしまいます。



デフォルト引数の制約

デフォルト引数が指定された仮引数より後ろに、デフォルト引数がない仮引数を定義してはならない。

デフォルト引数を利用する場合は、必ず一番後ろの引数から順にデフォルト値を指定するようにしましょう。

5.3.3 引数のキーワード指定



僕の eat 関数の定義をより実情に合わせてみました！ ただ、夜がカレーじゃない日は、関数の呼び出しがしっくりなくて。

松田くんは何に悩んでいるのでしょうか（コード 5-24、コード 5-25）。

5
章

コード 5-24 松田くんの基本的な eat 関数

```
1 def eat(breakfast, lunch='ラーメン', dinner='カレー'):  
2     print('朝は{}を食べました'.format(breakfast))  
3     print('昼は{}を食べました'.format(lunch))  
4     print('晩は{}を食べました'.format(dinner))
```

夜は基本カレー

昼は基本ラーメン

コード 5-25 夜がカレーじゃない日の eat 関数の呼び出し

```
1 eat('納豆ごはん', 'ラーメン', 'カレーうどん')
```

昼は基本ラーメンなので、
本当は省略したいが…

昼はいつも基本的にラーメンなので、これを省略して「eat('納豆ごはん', 'カレーうどん)」のように呼び出したいところですが、実際にこのような記述をすると、「朝は納豆ごはん、昼はカレーうどん、夜はカレー」と解釈されてしまいます。そのため、松田くんは仕方なく第2引数を省略せずに記述しているようです。

このようなケースでは、**引数のキーワード指定**という構文を用いると便利です。



引数にキーワードを指定した関数呼び出し

関数名 (仮引数名 1=実引数 1, 仮引数名 2=実引数 2, …)

※実引数として列挙された順番にかかわらず、値は指定された仮引数に引き渡される。

キーワード指定を使うと、松田くんの悩みは次のように解決できます (コード 5-26)。

コード 5-26 夜がカレーじゃない日の eat 関数の呼び出し (キーワード指定)

```

1 eat (breakfast='納豆ごはん', dinner='カレーうどん')    # ①
2 eat (dinner='カレーうどん', breakfast='納豆ごはん')    # ②
3 eat ('納豆ごはん', dinner='カレーうどん')              # ③

```

キーワードを指定した実引数は、記述した順序にかかわらず指定した仮引数に渡されます (コード 5-26 ①、②)。キーワードを指定しない実引数は、これまでどおり前から順に仮引数に渡されます (コード 5-26 ③の「納豆ごはん」)。したがって、コード 5-26 の 3 つの関数呼び出しはどれも同じ意味になります。

5.3.4 可変長引数



さらに食生活のリサーチを進めたんだが、何と 1 日に 2 回もおやつを食べている日もあることが発覚した。

(ど、どこでその情報を…) ちっ、違いますよ！ 先週の水曜日はおやつ4回でしたからっ！



日に三度の食事以外でおやつを何回か食べる場合、まず思いつくのは次のような eat 関数の定義でしょう(コード 5-27、コード 5-28)。

コード 5-27 おやつも出力できる eat 関数

```
1 def eat(breakfast, lunch, dinner='カレー', desserts=()):
2     print('朝は{}を食べました'.format(breakfast))
3     print('昼は{}を食べました'.format(lunch))
4     print('晩は{}を食べました'.format(dinner))
5     for d in desserts:
6         print('おやつに{}を食べました'.format(d))
```

要素数0のタプルをデフォルト引数として指定

5章

コード 5-28 おやつも食べた日の eat 関数の呼び出し

```
1 eat('トースト', 'パスタ', 'カレー', ('アイス', 'チョコ', 'パフェ'))
```

おやつの部分は丸カッコで囲んでタプルにする

この関数は、第4引数として複数のおやつが入ったタプルを受け取ることを表明しています。ただし、呼び出しの際は、**おやつ部分を丸カッコで囲んでタプルにすることを忘れてはなりません。**



うーん、まあ気をつければいだけかもしれないけど、朝・昼・晩・おやつ・おやつ・おやつって気軽に書けるといいんだけどなあ。

松田くんのささやかな願いを叶える道具が、**可変長引数**というしくみです。



可変長引数を利用した関数定義

def 関数名 (仮引数名 1, 仮引数名 2, ..., * 仮引数名 n):

※呼び出し時に n 個以上の実引数を指定することができる。

※第 n 引数以降に指定した実引数は、1 つのタプルとして受け取る。

※第 n 実引数の指定が省略された場合、関数は空のタプルを受け取る。

※可変長引数は、末尾の仮引数にしか指定することができない。

このしくみを用いて、より呼び出しやすい eat 関数を作ってみましょう (コード 5-29、コード 5-30)。

コード 5-29 おやつも出力できる eat 関数 (可変長引数を利用)

```
1 def eat(breakfast, lunch, dinner='カレー', *desserts):
2     print('朝は{}を食べました'.format(breakfast))
3     print('昼は{}を食べました'.format(lunch))
4     print('晩は{}を食べました'.format(dinner))
5     for d in desserts:
6         print('おやつに{}を食べました'.format(d))
```

desserts は可変
長引数であるこ
とを表明

コード 5-30 おやつも食べた日の eat 関数の呼び出し (可変長引数を利用)

```
1 eat('トースト', 'パスタ', 'カレー', 'アイス', 'チョコ', 'カレー')
```

この部分が 1 つのタプルとして
仮引数 desserts に引き渡される



便利なのはわかったけど…、6つ目の引数も「カレー」って…。

なお、これまで何度も用いてきた format 関数も、呼び出し時に指定する引数の数を変えられる関数でした。これは format 関数が可変長引数を使って定義されているからです。



Column

ディクショナリを用いた可変長引数

可変長引数にタプルではなくディクショナリを用いることもできます。仮引数の前に付ける「*」を2つにすると、実引数をディクショナリとして受け取ることができます。

```
1 def eat(**kwargs):
2     for key in kwargs:
3         print('{}に{}を食べました'
              .format(key, kwargs[key]))
```

for 文にディクショナリを指定するとキーが順番に取り出される

```
1 eat(朝食='納豆', 遅めの昼食='パスタ',
     夕方のおやつ='カレーパン')
```

実行結果

朝食に納豆を食べました

遅めの昼食にパスタを食べました

夕方のおやつにカレーパンを食べました

この部分が1つのディクショナリとして引数 kwargs に渡される

可変長引数がタプルの場合は仮引数名を「*args」、ディクショナリの場合は「**kwargs」とする慣習があります。

5.4 独立性の破れ

5.4.1 グローバル変数



工藤さん、大変です！ 「独立性の壁」が崩壊しちゃいました！

松田くんが慌てて持ち込んだ次のコードを見てみましょう（コード 5-31、コード 5-32）。

コード 5-31 引数を使わないで変数 name の値を受け渡している

```
1 name = '松田'
2 def hello():
3     print('こんにちは' + name + 'さん')
```

関数の外にある変数 name を使っている

コード 5-32 hello 関数の呼び出し

```
1 hello()
```

実行結果

こんにちは松田さん

「関数は外の世界からは独立しており、引数や戻り値を使わなければ外の世界とデータのやりとりはできない」。それが関数の原則でした(P204)。しかし、コー

ド 5-31 では、関数の外側(1 行目)で定義した変数 `name` を関数の中(3 行目)で使うことができています。



ありや、見つかった？ 参ったなあ…。まあ一応、教えておこうか。

実は、「変数の独立性」が崩壊したかのように見える例外的なケースがいくつか存在します。コード 5-31 はそのうちの 1 つであり、このケースでは、変数 `name` はどの関数にも属さない場所(関数定義の外側)で定義された**グローバル変数**(global variable)と呼ばれる特殊な変数であることに起因しています。

5章



グローバル変数とその性質

すべての関数から参照してデータを利用することができる。ただし、同名のローカル変数がある場合、ローカル変数が優先して利用される。

コード 5-31 では、`hello` 関数の中で変数 `name` を利用していますが、ローカル変数 `name` が定義されておらず存在しないため、グローバル変数である `name` が参照されたというわけです。

次に、これとは逆に、関数の中からグローバル変数への代入を試みてみましょう(コード 5-33、コード 5-34)。

コード 5-33 グローバル変数に代入する？

```
1 name = '松田'
2 def change_name():
3     name = '浅木'
4 def hello():
```

グローバル変数 `name` に代入しているつもり


```
5 print('こんにちは' + name + 'さん')
```

コード 5-34 関数の中からグローバル変数への代入を試みる

```
1 change_name()
2 hello()
```

実行結果

こんにちは松田さん



エラーにはならなかったけど…、名前が変わってないわ。

コード 5-33 の 3 行目は、「change_name 関数の中でローカル変数 name を定義する」と解釈されてしまいます。そして、ローカル変数の独立性の原則に従って、ローカル変数である name に「浅木」を代入しても、グローバル変数 name の値には影響しないのです。

もしどうしても関数の中からグローバル変数を書き換えたい場合には、**global 文**を使って、指定した変数がグローバル変数であることを明示的に宣言する必要があります(コード 5-35)。



global 文

global 変数名

コード 5-35 global 文を用いてグローバル変数に代入する

```
1 name = '松田'
2 def change_name():
3     global name
4     name = '浅木'
5 def hello():
6     print('こんにちは' + name + 'さん')
```

この関数における「name」はローカル変数ではなくグローバル変数を意味することを宣言

グローバル変数 name へ代入している

コード 5-35 で定義している関数を、コード 5-34 と同様に呼び出して実行してみると、「こんにちは浅木さん」という結果が表示されます。グローバル変数 name の値が書き換わったことが確認できるでしょう。

このように、グローバル変数 name は change_name 関数と hello 関数の 2 つの関数から読み書きされており、引数も戻り値も使わずに関数を連携させることが可能になります。グローバル変数とは、複数の関数が 1 つの変数を共有する手段とも言えるのです。

5.4.2 引数と戻り値の存在価値



なーんだ、じゃあ全部グローバル変数を使えば、引数も戻り値もいらなないじゃないですか。global 文を使えば、すべての関数からいつでも自由に書き換えができるんだし。

そうくると思ったよ。でも、その方法を本格的なプログラム開発でやろうとすると破綻してしまうんだよ。



本書でこれまで紹介してきたような数十行程度の小さなプログラムならば、松田くんの提案する「全変数をグローバルに定義して、各関数から共有して利用す

る方法」も合理的かもしれません。すべての関数の定義や呼び出し方法は、自分ひとりが理解していればそれでよいからです。

しかし、業務のためにプロジェクトチームで開発するプログラムや、機械学習を活用するような複雑なプログラムを作る場合は、グローバル変数の副作用による問題が噴出するでしょう。「○○関数を呼び出すときには、事前に変数△△と変数××をグローバル変数として定義して値を入れておく必要がある」「変数□□は**関数で使っているから、別の関数からは絶対に書き換えてはならない」といったような暗黙のルールが無数に登場するため、大勢の開発者に混乱を招き、結果、バグや障害が頻発することになります。



グローバル変数を濫用するリスク

グローバル変数は便利だが、開発者の混乱やミスを招きやすいという副作用がある。自分ひとりだけの開発では問題になりにくいですが、チームでの開発や中規模以上の開発には向かない。

グローバル変数に頼らずに、引数や戻り値を使ってデータをやりとりする方法ならば、この副作用をかなり抑え込むことができます。呼び出そうとする関数の def 文を見れば、どのようなデータをいくつ引き渡すべきかが明確に表明されていますし、ローカル変数の独立性があるおかげで、誤ってほかの関数が使っている変数を書き換えることもありません。

本格的な開発では、「自分が作った関数と、他人が作った関数を連携させて 1 つのプログラムを作る」という道を避けて通ることはできません。逆に、自分はある関数を作る役割を担い、まったく別の他人がそれを呼び出すプログラムを作ることもあるでしょう。

関数とは、他人と力を合わせて目的のプログラム開発を実現する、「分業のための道具」でもあります。関数を作る開発者には、「他人が作った関数と一緒に使っても副作用が出ることなく、安心して組み合わせることのできる関数を作る」ことが求められるのです。



わかりました！ 引数や戻り値をちゃんと使って、安心して利用できる関数を作っていきます！

引数や戻り値にはきちんと存在理由があったんですね。理屈がわかってスッキリしました。これでもう関数はどんとこいですよ！



頼もしいね。でも、関数にはまだ隠された謎が存在するんだ。次章では、その謎を解くために、新しい概念を学んでいこう。

5章



Column

関数定義と呼び出しのコーディング

本章では、関数と呼び出す側のコードを明確に分けて紹介しました。しかし、必ずしもこの書き方でなくてはならないということではありません。def で始まる関数の定義が事前に実行されていれば、その関数を呼び出すことが可能です。

以降では、関数定義と関数呼び出しを同じコードに記述する形式も登場します。

5.5

第 5 章のまとめ

この章では、次のことを学びました。

関数

- 関数はあらかじめ準備されたものだけでなく、自分で作ることができる。
- 関数を用いて処理を部品化することで、可読性や保守性、作業効率が向上する。
- 関数は複数人で分業するための道具である。
- 定義された関数は、() 演算子を用いて何度でも呼び出すことができる。

引数

- 関数は仮引数を表明することで、実行時にデータを受け取ることができる。
- 仮引数にはデフォルト値や可変長引数を定めることができる。

戻り値

- return 文で関数の実行を終了し、呼び出し元に戻り値を返すことができる。
- 戻り値には、整数や文字列、コレクションを 1 つのみ用いることができる。
- 呼び出し元では、代入文によって戻り値を変数に受け取ることができる。

変数の独立性

- 関数の外で定義された変数をグローバル変数、関数の中で定義された変数をその関数のローカル変数という。
- ローカル変数は、原則として関数の外から読み書きすることができない。
- グローバル変数の参照に制限はないが、代入には global 文が必要である。
- グローバル変数は自身で小さなプログラムを作る目的以外では推奨されない。

5.6

練習問題

5章

練習 5-1

次の各関数について、定義の 1 行目に記述する内容と、戻り値がある場合は望ましいデータ型を答えてください。なお、データ型は、表 1-6 (P061) の 4 つの型、およびリスト・ディクショナリ・タプル・セットから選ぶものとします。

- (1) 呼び出すと、「今日は晴れです」という文字列を画面に表示する `weather` 関数
- (2) 円の半径を渡すとその円の面積を返す `calc_circle_area` 関数
- (3) 呼び出すと現在時刻を調べて、「18 時 25 分 30 秒」のようなデータを返す `nowstr` 関数
- (4) 呼び出すと現在時刻を調べて、時分秒を表す 3 つの数値を返す `nowint` 関数
- (5) 西暦を渡すと、うるう年かどうかを判定する `is_leapyear` 関数

練習 5-2

練習 5-1 の (5) の `is_leapyear` 関数について、次の判定方法を参考にして関数定義を完成させてください。

[うるう年の判定方法]

- ・ 400 で割り切れる年はうるう年である。
- ・ 4 で割り切れる年はうるう年だが、100 で割り切れる年はうるう年ではない。

また、キーボードから現在の西暦を入力させたうえでこの関数を呼び出し、次のように表示するプログラムを作成してください。

- ・ うるう年だった場合 : 西暦〇〇年は、うるう年です
- ・ うるう年でなかった場合: 西暦〇〇年は、うるう年ではありません

練習 5-3

次のコードについて、以下の問いに答えてください。


```
1 def take_bus():
2     print('バスに乗ります')
3 def run():
4     print('走ります！')
5 def walk():
6     print('ちょっと歩きます')
7
8 print('行ってきます！')
9 walk(); take_bus(); run(); run()
10 print('ただいま')
```

このプログラムを、走ったあとは必ず歩くようにするために修正すべき点を 2 つ挙げてください。ただし、関数でない部分(最後の 3 行)は変更しないものとします。

練習 5-4

次のような試験の得点を分析する関数があります。

```
1 def analyze_scores(sansu, kokugo, rika, syakai, eigo=None, *others):
2     # 処理内容は省略
3     return [max_score, min_score, avg_score]
```

この関数について述べた(1)～(5)の文について、正しいものは○、誤っているものは×を答えてください。また、×としたものについては、その理由を説明してください。

- (1) この関数を呼び出すときに、少なくとも算数・国語・理科・社会・英語の 5 教科の点数を引数として与えなければならない。
- (2) 算数・国語・理科・社会・英語がすべて 80 点、音楽と体育と美術がすべて 70 点の場合、次のようなコードでこの関数を呼び出すことができる。

```
analyze_scores(80, 80, 80, 80, 80, 70, 70, 70)
```

- (3) 次のコードでこの関数を呼び出したとき、仮引数 `others` には `int` 型の `80` ではなく、要素数 1 のリスト `[80]` が渡される。

```
analyze_scores(80, 80, 80, 80, 80, 80)
```

- (4) 次のコードでこの関数を呼び出すと、仮引数の定義順とは異なる任意の順で実引数を指定することができる

```
analyze_scores(eigo=80, kokugo=20, rika=30, syakai=40, sansu=70)
```

- (5) この関数は次の 2 つの方法で呼び出すことができる。

```
result = analyze_scores(1, 2, 3, 4, 5)
[x, n, g] = analyze_scores(1, 2, 3, 4, 5)
```

練習 5-5

次のようなルールに基づいて割り勘を計算するプログラムがあります。

- ・ 1 人あたりの支払額は支払総額を参加人数で割った金額とする。
- ・ 支払いの単位は 100 円とし、100 円未満の金額がある場合は切り上げる。
- ・ 支払額を超過した分は、幹事が受け取ることができる。

```
1 # 計算データの入力
2 amount = int(input('支払総額を入力してください >>'))
3 people = int(input('参加人数を入力してください >>'))
4
5 # 割り勘の計算
6 dnum = amount / people      # 総額を人数で割る（端数も保持）
7 pay = dnum // 100 * 100     # 100円未満を切り捨てる
8 if dnum > pay:               # 元の値と比較して、
```

```

9      pay = int(pay + 100) # 小さければ100円未満があったので上乗せ
10
11     # 幹事の支払額の計算
12     payorg = amount - pay * (people - 1)
13
14     # 結果の表示
15     print('*** 支払額 ***')
16     print('1人あたり{}円({}人)、幹事は{}円です'
            .format(pay, people - 1, payorg))

```

次の(1)～(3)の機能について、それぞれの仕様に従って関数に部品化し、利用するようにプログラム全体を修正してください。

(1) int_input 関数

機能	画面に入力を促すメッセージを表示し、入力結果を数値に変換して返す メッセージの例)〇〇を入力してください>>
引数	入力を促す項目を示す文字列
戻り値	入力された数値

(2) calc_payment 関数

機能	割り勘の額を計算する。ただし、幹事以外の支払額は100円単位に丸めて切り上げる 例)813 → 900、1370 → 1400
引数	支払総額、参加人数(省略時は2とする)
戻り値	1人あたりの支払額(100円単位)と幹事支払額

(3) show_payment 関数

機能	渡された引数を見やすく表示する 例)*** 支払額 *** 1人あたり〇円(〇人)、幹事は〇円です
引数	支払額、幹事支払額、参加人数(省略時は2とする)
戻り値	なし

5.7

練習問題の解答

5章

練習 5-1

項番	定義	戻り値の型
(1)	def weather():	なし
(2)	def calc_circle_area(dia):	float 型
(3)	def nowstr():	str 型
(4)	def nowint():	リストまたはタプルまたはディクショナリ
(5)	def is_leapyear(y):	bool 型

※仮引数の名称は、上記解答と異なってもかまわない。

練習 5-2

```

1  def is_leapyear(y):
2      return (y % 400 == 0 or (y % 4 == 0 and y % 100 != 0))
3
4  current_year = int(input('現在の西暦を入力してください >>'))
5  if is_leapyear(current_year):
6      print('西暦{}年は、うるう年です'.format(current_year))
7  else:
8      print('西暦{}年は、うるう年ではありません'.format(current_year))

```

練習 5-3

(1) 4 行目と 5 行目の間に次の 1 行を挿入する。

```
walk()
```

(2) walk 関数の定義を run 関数の定義より前に移動する。

練習 5-4

- (1)×: 仮引数 `eigo` にはデフォルト引数が指定されているため、算数・国語・理科・社会の 4 教科の点数だけを指定して呼び出すことが可能。
- (2)○
- (3)×: 仮引数 `others` には要素数 1 のタプル (80,) が渡される。
- (4)○
- (5)○

練習 5-5

```
1 def int_input(msg):
2     return int(input('{}を入力してください >>'.format(msg)))
3 def calc_payment(amount, people=2):
4     dnum = amount / people      # 総額を人数で割る (端数も保持)
5     pay = dnum // 100 * 100     # 100円未満を切り捨てる
6     if dnum > pay:              # 元の値と比較して、
7         pay = pay + 100        # 小さければ100円未満があったので上乗せ
8     payorg = amount - pay * (people - 1)
9     return [int(pay), int(payorg)]
10 def show_payment(pay, payorg, people=2):
11     print('*** 支払額 ***')
12     print('1人あたり{}円({}人)、幹事は{}円です'
13           .format(pay, people-1, payorg))
14
15 # 計算データの入力
16 amount = int_input('支払総額'); people = int_input('参加人数')
17 # 割り勘の計算
18 [pay, payorg] = calc_payment(amount, people)
19 # 結果の表示
20 show_payment(pay, payorg, people)
```

第6章

オブジェクト

前章で関数を学んだ私たちは、大きなプログラムも部品に分けて開発することができるようになりました。しかし、Python における関数には、重大な落とし穴が存在します。この章では、その鍵となる「オブジェクト」という概念を学び、より安全で合理的に関数を使えるようになります。

CONTENTS

- 6.1 「値」の正体
- 6.2 オブジェクトの設計図
- 6.3 オブジェクトの落とし穴
- 6.4 第6章のまとめ
- 6.5 練習問題
- 6.6 練習問題の解答

6.1 「値」の正体

6.1.1 format 関数の謎



print() もただの関数で、似たようなものを自分でも作れるって知ったときは、ちょっと感動しちゃいましたよ。

そうね。でも、関数のことがわかったら、format() の構文が気になっちゃって。



前章で私たちは、オリジナルの関数を作る方法を学びました。これまで用いてきた print 関数や input 関数も、「Python があらかじめ定義してくれていた関数」にすぎません。そのため、私たちはこれらの関数も自分で作ったオリジナル関数も、「関数名 (引数)」という共通の構文で同様に呼び出すことができたのでした。

しかし、私たちが第 1 章で出会った format 関数 (P070) や、第 2 章で出会った append 関数 (P087) は、通常の関数とは異なる呼び出し方法で記述していたことを浅木さんは思い出したようです (コード 6-1)。

コード 6-1 append 関数や format 関数の呼び出し

```
1 tpl = '3人目は{}さん'
2 names = ['松田', '浅木']
3 names.append('工藤')
4 message = tpl.format(names[2])
5 print(message)
```

値・関数名 () で呼び出す



append() や format() も関数の一種であることには違いないんだが、「値に所属している」という意味で、実は少しだけ特殊な存在なんだ。

えっ!? 関数が、値に所属…?



その疑問を解くためには、今まで描いてきた「値」のイメージを新たにしなければなりません。たとえばこれまで、コード 6-1 の 1 行目に登場する変数 `tpl` には、単に「3 人目は {} さん」という文字列のデータが値として入っている様子をイメージしてきました。しかし、実際には、図 6-1 のような姿をしています。

6
章

今までの「値」のイメージ

これからの「値」のイメージ

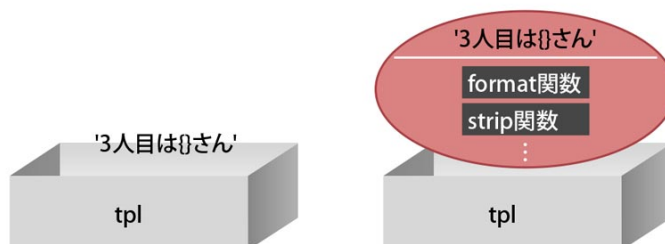


図 6-1 「値」の従来イメージと新しいイメージ



文字列のデータと一緒に、関数もセットになって入っている、ということですか？

ご名答。「関数なんて一緒に入れた覚えはない！」と思うかもしれないが、Python では、文字列データは `format()` や `strip()` といった関数がセットにされて 1 つの値として扱う、ということに決まってるんだ。



ただのデータととらえていたものが、実は関数を従えているというのは、文字

列だけではありません。これまで当然のように変数に入れて扱ってきた整数、真偽値、リストなどそれぞれが、データに加えて、いくつかの決められた関数を従えているのです(コード 6-2)。

コード 6-2 整数もリストもデータと関数をセットで持つ

1 num = 10 2 print(num.bit_length()) 3 names = ['松田', '浅木'] 4 names.append('工藤')	}	10 というデータと bit_length 関数が セットになっている
	}	'松田'・'浅木' というデータと append 関数などがセットになっている

ここまで見てきたように、あるデータと、そのデータに関する処理を行う関数がひとかたまりとなっているものを**オブジェクト** (object) といいます。Python では、**あらゆる値はオブジェクトとして扱う**決まりになっています。



すべての値はオブジェクト

- ・「データ」と「そのデータに関する処理を行う関数」をひとかたまりにしたものをオブジェクトという。
- ・Python におけるあらゆる値はオブジェクトである。



あらゆるものをオブジェクトとして扱うのが、Python の特徴の 1 つなんだ。

オブジェクトに所属する関数は、特に**メソッド** (method) ともいわれ、次の構文で呼び出す決まりになっています。



メソッドの呼び出し

オブジェクト.メソッド名(引数…)

※オブジェクトには、リテラル、オブジェクトが格納された変数、オブジェクトを戻り値として返す関数呼び出しを指定することができる。

6.1.2

オブジェクトの型



今までただの数として扱ってきた3や10も、関数を従えたオブジェクトだったのか。

今まで見ていた世界が急に変わってしまった感じね。それにしても、どのオブジェクトがどんな関数を持っているのかしら。



6章



オブジェクトが従える関数を決定するもの

オブジェクトが従える関数は、「型」によって決まる。

すべてのオブジェクトには、必ずその種類を表す「型」があります。これは、オブジェクトの型を調べるという機能を持った `type` 関数によって知ることができます(P064)。たとえば、3や10といった数値リテラルのオブジェクトは `int` 型、`input` 関数が返すキーボードからの入力データは `str` 型でした。

そして、あるオブジェクトがどんな関数を従えているかは、通常、そのオブジェクトの型によって決定されます。たとえば、整数型(int 型)のオブジェクトならば、`bit_length` 関数を必ず従えていますし、リスト型(list 型)のオブジェクトならば `append` 関数を必ず従えています。

Python には標準としてどのような種類の型があって、それぞれの型のオブジェクトがどのようなメソッドを持つかは、Python の公式サイトにドキュメントとして公開されています(図 6-2、<https://docs.python.org/ja/3/library/stdtypes.html>)。

整数型における追加のメソッド

整数型は `numbers.Integral` 抽象基底クラス (abstract base class) を実装します。さらに、追加のメソッドをいくつか提供します:

`int.bit_length()`

整数を、符号と先頭の 0 は除いて二進法で表すために必要なビットの数を返します:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

図 6-2 Python の公式ドキュメントにおける int 型のメソッドの紹介

6.1.3 文字列オブジェクトが持つメソッド



工藤さん！ 公式ドキュメントを見ていたら、str 型ってたくさんメソッド持ってるじゃないですか！こんなに、どうやって使うんですか？

もう見つけたのかい。仕方ない、ちょっとだけ紹介するか。



文字列型(str 型)は、Python が標準で提供する型の中でも特に多くの便利なメソッドを持っています。これらのメソッドを使うことによって、文字列を効率的に操作することができます(表 6-1)。

表 6-1 str 型オブジェクトが備える代表的なメソッド

メソッド名	機能
capitalize()	先頭文字だけ大文字に、残りを小文字にする
lower()	すべてを小文字にする
upper()	すべてを大文字にする
title()	単語の先頭だけを大文字に、残りを小文字にする
strip()	文字列の前後の空白を取り除く
split(●)	文字列●で区切り、各要素をリストで返す
replace(●, ■)	文字列中の●部分を■に置き換えた結果を返す
count(●)	文字列●が登場する回数を返す

次のコード 6-3 は、これらのメソッドを用いた例です。

6章

コード 6-3 str オブジェクトのメソッドを活用した血液型占い

```

1 userinfo = input('名前と血液型をカンマで区切って1行で入力 >>')
2 [name, blood] = userinfo.split(',')
3 blood = blood.upper().strip()
4 print('{}さんは{}型なので大吉です'.format(name, blood))

```

実行結果

```

名前と血液型をカンマで区切って1行で入力 >>工藤,b
工藤さんはB型なので大吉です

```



ま、あくまでも一例だよ。興味がわいたら、いろいろと試してみるといいだろう。



Column

関数さえオブジェクト

Python では、文字列や整数だけでなく関数も、「呼び出されたら動作する内容を保持する値」という考えのもとに function 型オブジェクトとして扱います。第 5 章で学んだ `def` による関数定義 (P201) とは、関数名と同じ名前の変数を準備し、そこに関数オブジェクトを代入する行為なのです。また、「関数名 ()」による関数の呼び出しは、変数内のオブジェクトを取り出し、処理を起動して、戻り値に化ける動作です。関数さえもオブジェクトとして扱う Python の特徴を、次のコードで体験してみてください。

```
1 def add(x, y):  
2     return x + y
```

関数オブジェクトを定義し、
変数 `add` に代入

```
3
```

```
4 type(add)
```

変数 `add` の型を調べる

```
5 newadd = add
```

変数に代入されたオブジェクトは別の変数にコピー可能

```
6 print(newadd(4, 5))
```

コピーされた関数オブジェクトを起動

6.2

オブジェクトの設計図

6.2.1

オブジェクトの姿を決定づける設計図



型によってメソッドは決まっていると紹介したけど、これは型ごとの「設計図」に書いてあるからなんだ。

6章

Python のそれぞれの型には、「この型のオブジェクトなら、この関数をメソッドとして従えるべきである」というルールがあることを紹介しました (P247)。これは、オブジェクトの設計図のようなものが Python 内部にあるため、正式にはこの設計図を**クラス** (class) といいます。すべてのオブジェクトはこのクラス定義に基づいて誕生し、どのクラスから生まれたかによってその型と持つメソッドが決まります (図 6-3)。

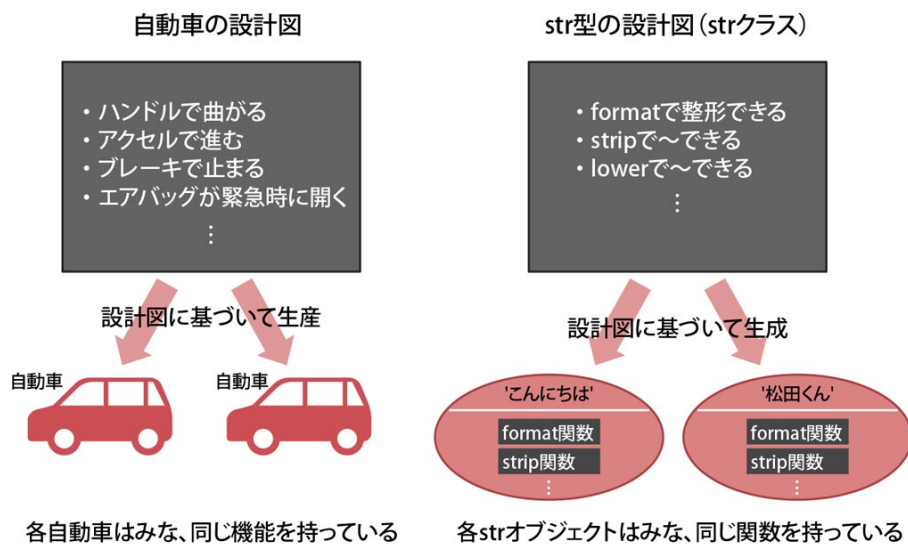


図 6-3 設計図から生まれるオブジェクト



じゃあ、僕たちがフツーに使ってきた文字列も、この設計図を使って生まれてきた文字列オブジェクトってことなんですか？

そうだよ。松田くんは「str 型の設計図をもとに生まれてこい」なんて指示した覚えはないだろうけどね。



これまで何気なくソースコード上に書いてきたリテラルには、実は、「クラスに基づいてオブジェクトを生み出し、そのオブジェクトに化けろ」という意味があります。より具体的には、各リテラルは次のようなオブジェクト生成の指示と解釈されます。



リテラルによるオブジェクト生成の指示(一部)

- ・ 小数点を含まない数字 : int クラスを用いてオブジェクトを生成。
- ・ 小数点を含む数字 : float クラスを用いてオブジェクトを生成。
- ・ ' や " で囲まれた文字 : str クラスを用いてオブジェクトを生成。
- ・ [] で囲まれた文字 : list クラスを用いてオブジェクトを生成。
- ・ {} で囲まれた文字 : { ~ : ~ } 形式なら dict クラス、{ ~ } 形式なら set クラスを用いてオブジェクトを生成。



私たちが軽い気持ちでコードに書いていた数字や文字列を使って、Python はすごい仕事をこなしていたのね。

なお、設計図からオブジェクトを生み出す手段は、前述のようなリテラルを使う方法だけではありません。Python では、次の構文を使うことで、あらゆる設計図からオブジェクトを生み出すことができます。



クラス名関数によるオブジェクト生成

変数名 = クラス名 ()

※クラスによっては、引数としてさまざまな設定や初期値を渡すことができる（詳細は Python 公式ドキュメントを参照）。

たとえば、int や list オブジェクトも、クラス名と同じ名前の関数を使うことで生成することが可能です（コード 6-4）。

6章

コード 6-4 リテラルやクラス名関数を用いたオブジェクトの生成

```
1 int_value1 = 0
2 int_value2 = int()
3 int_value3 = int(9)
4 list_value1 = []
5 list_value2 = list()
6 list_value3 = list(('松田', '浅木'))
```

int オブジェクト (中身のデータは 0) を生成

int オブジェクト (中身のデータは 9) を生成

空のリストオブジェクトを生成

2つの要素を持つリストオブジェクトを生成



オブジェクトの生成方法

クラスに基づきオブジェクトを生成する方法は 2 つ存在する。

- ①リテラルを用いる（使われるクラスは記述に応じて自動的に決まる）。
- ②クラス名と同名の関数を呼び出す。



Column

コレクション変換関数の正体

コード 6-4 の最後の行に着目してください。引数として渡されている「('松田', '浅木')」の部分は、実はタプルとして解釈されます (P100)。その結果、list 関数に 1 つのタプルを渡していることになります。これは、第 2 章で紹介したコレクションを相互変換する関数 (P108) と同じものです。

第 2 章では、引数に渡したコレクションを目的のコレクションに変換してくれる関数として紹介しましたが、より正確には、それぞれのコレクションクラスからオブジェクトを生み出す関数だったのです。

6.2.2 オリジナルの設計図を作る



実は、設計図であるクラスも、関数と同じように自分たちで作ることができるんだ。

ええっ。設計図が作れるってことは、自分で「型」が作れるってことじゃないですか！



Python では、私たちがオリジナルの設計図を作って、そのオブジェクトを利用することも許されており、そのための専用の構文が存在します。すべての人に必要となる知識ではないため紹介のみに留めますが、コード 6-5 でその雰囲気に触れておきましょう (構文やしくみを厳密に理解する必要はありません)。



なお、sleep メソッドの仮引数 self (コード 6-5 の 4 行目) だが、現段階では、メソッドの第 1 引数に書く決まり文句と思ってくれていいよ。これはオブジェクト自身を表す変数なんだ。

コード 6-5 RPG の勇者を表すクラスの定義と利用

```

1 class Hero:
2     name = '松田'
3     hp = 100
4     def sleep(self, hours):
5         print('{}は{}時間寝た！'.format(self.name, hours))
6         self.hp += hours
7
8 # ゲーム開始
9 print('スッキリファンタジーXII ～金色の理想郷～')
10 h = Hero()
11 h.sleep(3)
12 print('{}のHPは現在{}です'.format(h.name, h.hp))

```

データとして name と hp、関数として sleep を持つオリジナル設計図 Hero を定義

HP100 の勇者松田がオブジェクトとして誕生

実行結果

スッキリファンタジーXII ～金色の理想郷～

松田は3時間寝た！

松田のHPは現在103です



ゲームが始まって、僕、すぐに寝ちゃってるじゃないですか！
まあ、よく寝ますけど…。

リアルでいいじゃないか。それにこのオブジェクト、HP データを持っていたり眠れたり、まるで本物の「勇者」みたいだろ？



コード 6-5 では、Hero という設計図からオブジェクトが生み出されています (10 行目)。このオブジェクトは、sleep 関数に加え、name や hp といった複数

のデータも持っているようですね。オブジェクトに属する関数をメソッドと呼ぶように、オブジェクトに属する個々のデータは、**属性** (attribute) と呼びます。

今後、この勇者の設計図に、攻撃力の属性や攻撃するなどのメソッドを追加すれば、より本格的なゲームに利用可能であると想像できます。このように、オブジェクトにさまざまな属性やメソッドを持たせて活用していく考え方を、**オブジェクト指向プログラミング** (object oriented programming) といいます。本書では詳しく紹介しませんが、オブジェクトを上手に活用すると、ラクに・楽しく・複雑なプログラムを作ることができるということをぜひ知っておいてください。



結局、クラスって金型みたいなものですね？ 金型を一度作っておけば、同じ形のたい焼きがどんどん作れるみたいな。

相変わらずハイカロリーな発想だが理解自体は合っているよ。
そしていろんな金型を揃えているのが Python の強みなんだ。



もし仮に、さまざまなステータスの属性や行動のメソッドを持つ Hero クラスが準備されているとしたら、あなたはたった 1 行「h = Hero()」と書くだけで、豊富な属性やメソッドを持つ勇者を生み出すことができます。

残念ながら Hero クラスは存在しませんが、ファイル操作や OS 制御、通信、データ統計などの各分野で役に立つたくさんの金型を、Python は標準で準備してくれています。私たちは、それらの準備されたクラスからオブジェクトを生み出し、属性やメソッドを使うだけで、データベースアクセスやネット通信などをはじめとする非常に高度で複雑な処理を簡単に実現できるのです。



Python の標準クラス

Python は標準で、種類豊富なクラスを準備してくれている。

私たちは、その金型からオブジェクトを生み出して自由に利用できる。

6.3

オブジェクトの落とし穴

6.3.1 オブジェクトの identity



Python が準備してくれたクラス、マーケティングにも使えるものがあるのかなあ。早く使ってみたいです！

そうだね！ ただ、クラスをうまく使いこなすためには、落とし穴をしっかりと知っておいてほしいんだ。

6
章

前節までに紹介したように、豊富に準備されているさまざまなクラスやオブジェクトを効果的に使いこなせるか否かが、Python 活用の成否を握ると言えるでしょう。しかし、Python のオブジェクトには、1 つ大きな落とし穴が潜んでいることが知られています。落とし穴のからくりを見破り、回避するための鍵となるのが **identity** という概念です。

identity とは、すべてのオブジェクトに自動的に割り振られる管理用の番号です。Python では、あるオブジェクトが生み出されると、自動的にほかと重複しない整数の値が identity として付与されます。そして、そのオブジェクトが消滅するまで変わることはありません。

あるオブジェクトの identity 値は、id 関数を使って調べることも可能です(コード 6-6)。

コード 6-6 オブジェクトの identity の確認

```
1 scores1 = [80, 40, 50]
2 scores2 = [80, 40, 50]
3 print('scores1のidentity: {}'.format(id(scores1)))
```

```
4 print('scores2のidentity: {}'.format(id(scores2)))
5
6 if scores1 == scores2:
7     print('scores1とscores2は同じ内容です')
8 else:
9     print('scores1とscores2は違う内容です')
10
11 if id(scores1) == id(scores2):
12     print('scores1とscores2は同じ存在です')
13 else:
14     print('scores1とscores2は違う存在です')
```

実行結果

```
scores1のidentity: 4451042816
scores2のidentity: 4451043136
scores1とscores2は同じ内容です
scores1とscores2は違う存在です
```

この値は実行するたびに変化する

2つのリストがそれぞれ異なる identity 値を持つことを確認したうえで 6 行目と 11 行目に着目してください。6 行目ではオブジェクトの「内容が等しいか」を判定している一方で、11 行目ではオブジェクトの identity 値が等しいか（同一のオブジェクトか）を判定しています。プログラミングの世界では前者のような判定を**等価判定**、後者を**等値判定**といいます。



等価と等値

内容が等しければ「等価」、存在が等しければ「等値」である。



scores1 と scores2 は、同じ内容を持つ、違うオブジェクトなんですね。

正解！ そしてこの identity が、変数に入っている「本当のもの」を知る鍵なんだ。



6.3.2 参照

これまで私たちは、変数にオブジェクトが格納されることを、図 6-4 の左図のような姿で理解してきました。しかし、このイメージは厳密には正しくありません。図 6-4 の右図のように、実は変数にはオブジェクト自体は入っておらず、その identity 値が格納されているのです。

6
章

scores1 = [80, 40, 50]を実行した場合のイメージ

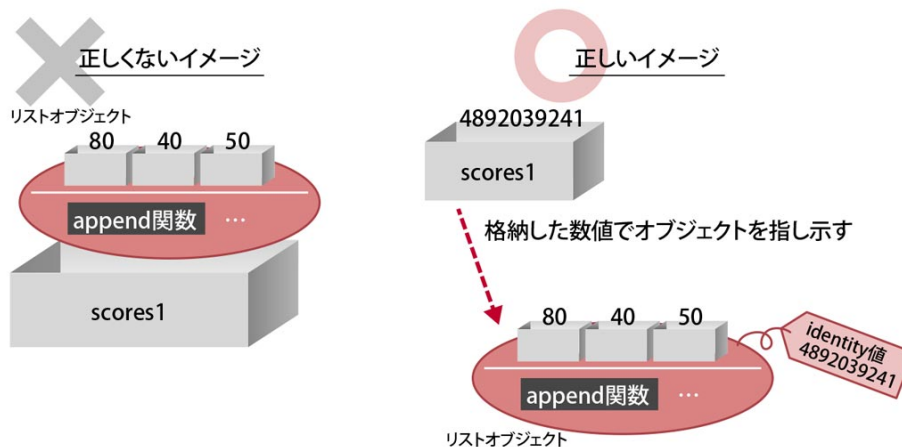


図 6-4 変数へのオブジェクト格納のイメージ



オブジェクトは別の場所にあって、変数に入ってるのは identity 値、つまりただの数値なんだ。

変数 `scores1` の中には、オブジェクト自体は入っていません。その代わりに、「この変数の内容の詳細は、この `identity` 値のオブジェクトを見てください」というように、オブジェクトを指し示す数値が入っています。このように、別のところにある実体を指し示すための数値を、プログラミングの世界では**参照** (reference) といいます。



このことを頭の片隅に入れたうえで、次のコードの不思議な動きについて考えてみよう。

次のコード 6-7 は、実行すると一見不可解とも思える結果が表示されます。

コード 6-7 リストオブジェクトのコピーによる不可解な動作

```
1 scores1 = [80, 40, 50]
2 scores2 = [80, 40, 50]
3 print('scores1の先頭要素は{}'.format(scores1[0]))
4 print('scores2の先頭要素は{}'.format(scores2[0]))
5
6 print('変数scores2の中身を変数scores1に代入（コピー）します')
7 scores1 = scores2
8
9 print('scores1の先頭要素を90に書き換えます')
10 scores1[0] = 90
11
12 print('90を代入したscores1の先頭要素は{}'.format(scores1[0]))
13 print('90を代入していないscores2の先頭要素は{}'.format(scores2[0]))
```


実行結果

scores1の先頭要素は80

scores2の先頭要素は80

変数scores2の中身を変数scores1に代入（コピー）します

scores1の先頭要素を90に書き換えます

90を代入したscores1の先頭要素は90

90を代入していないscores2の先頭要素は90

90を代入していないのに90になっている



おかしい。変数 scores2 には 90 を入れてないのに 90 になっている。ってことはつまり…。

6章

そう。ポイントは7行目でコピーしている「もの」だ。



7行目では「scores2の中身をscores1に代入」していますが、**ここでコピーされるものは、identity というただの数値**であることを思い出す必要があります。つまり、リストをコピーしているわけではなく、参照(identity値)をコピーしているにすぎないのです(図6-5)。そのため、scores2とscores1とは実質的に同じオブジェクトを指し示すことになるのです。

scores1 = scores2を実行した場合のイメージ

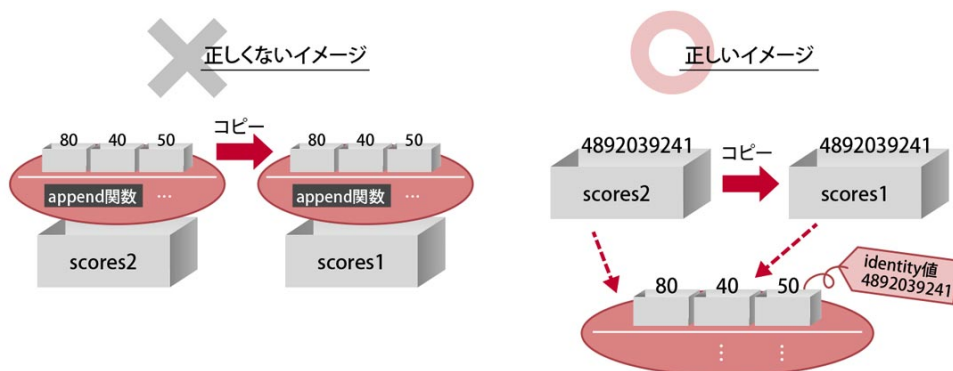


図 6-5 リストオブジェクト同士の代入に関するイメージ



結局、代入によって、scores2 も scores1 も同じオブジェクトを指し示してしまうってことなんですね。

そのとおり。もはや scores1 と scores2 は、同一のリストオブジェクトに付けられた 2 つの別の名前にすぎないんだ。リスト 6-7 の 10 行目の「scores1[0] = 90」は、「scores2[0] = 90」と書いても実質同じ意味になるんだよ。



代入文でコピーされるもの

ある変数を代入文でコピーすると、オブジェクトではなく参照がコピーされる。そのため、同じオブジェクトを複数の異なる名前でアクセスできるようになる。



Column

「箱」より「名札」に近い Python の変数

ほぼすべてのプログラミング言語には変数のしくみがあり、よく「データを入れる箱」という例えで紹介されます。本書でも冒頭から、変数を箱だと例えてきました。

しかし、図 6-5 のような構図を考えると、Python における変数とは、「identity 値に、わかりやすい名前を付けたもの」と考えることもできます。実際、Python の公式ドキュメントでは、変数への代入を名前付け (naming) と表現しています。

6.3.3 参照による副作用



参照は、オブジェクトを指し示すためのしくみなんですね。

でも、これのどこが落とし穴なんですか？



参照自体は落とし穴じゃないよ。ただ、参照と関数を組み合わせたとき、やっかいなトラップが発動することがあるんだ。

6
章

第5章で私たちは、変数の独立性について学びました(P202)。各関数で定義された変数(引数を含む)は、基本的に独立していて外部からはアクセスできず、仮に同じ名前の変数が関数の外に存在したとしても、まったくの別物として解釈されるというものでした。

このため、関数同士でデータをやりとりするためには、わざわざ引数や戻り値といったしくみを使う必要がありました。しかしこの独立性が保証されているおかげで、「ある関数を呼び出したら、自分が使っていた変数と同名の変数を偶然使っていて、変数の中身を壊されてしまった」などということは起こらず、一定の安心感を得ることができたのです。



本格的な開発では、赤の他人が作った関数を呼び出すこともざらにある。だから、他人の関数を使っても、自分が作った部分には悪影響が出ないという保証は、分業には絶対に欠かせない重要な基盤なんだ。

しかし、引数や戻り値を利用していても、ある条件のもとでは呼び出した関数に変数を破壊されてしまうという事故が起こり得るのです。実際に、次のコードで確認してみましょう(コード 6-8)。

コード 6-8 関数に渡すと変数の内容を書き換えられてしまう

```

1 def add_suffix(names):
2     for i in range(len(names)):
3         names[i] = names[i] + 'さん'
4     return names
5
6 before_names = ['松田', '浅木', '工藤']
7 after_names = add_suffix(before_names)
8 print('さん付け後:' + after_names[0])
9 print('さん付け前:' + before_names[0])

```

渡されたリスト内の名前に「さん」を付ける関数

add_suffix から返されたリスト内の先頭の要素

add_suffix に渡したリスト内の先頭の要素

実行結果

さん付け後:松田さん

さん付け前:松田さん

なぜか「さん」が付いてしまっている



あれれ。before_names の要素にはさん付けを指示してないはずなのに。

これが、ほかならぬ「参照による副作用」なんだ。



コード 6-8 の 7 行目では、引数として add_suffix 関数にリスト before_names を渡しています。しかし、ここで呼び出し先に渡されるのは、リストオブジェクトそのものではなく、before_names を指す参照 (identity 値) です。これにより、add_suffix 関数の names は、before_names と同じものを指し示すことになります。つまり、add_suffix 関数内でリスト names の中身を変更してしまうと、呼び出し元の before_names の中身も変更されてしまうのです。

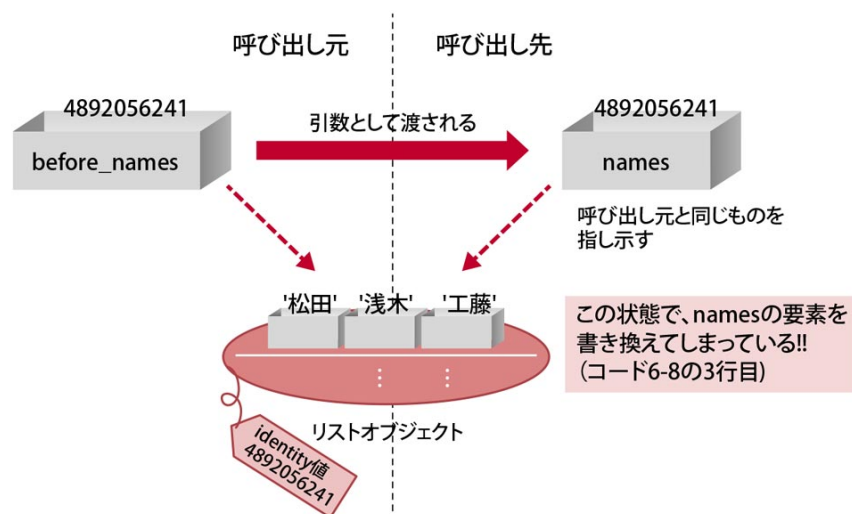


図 6-6 引数として参照を引き渡すことによる悪影響の発現

このように、関数を呼び出す際に引数や戻り値として参照をやりとりすると、**変数の独立性が崩れる**ことは、さまざまなプログラミング言語に共通の落とし穴としてよく知られています。特に、すべての値がオブジェクトとして扱われる Python では、引数や戻り値は常に参照でやりとりするため、このような副作用が発生する恐れと隣り合わせなのです。



Python の関数にはリスクが常に伴う

Python では、関数に引数を渡すと、その内容が破壊されてしまうリスクが必ず付きまとう。

6.3.4 防御的コピー



自分の変数が壊される可能性があるなんて、困ります！

そうですよ。そんなリスクを考えたら、Python がいくら便利な関数やクラスを準備してくれても、怖くて呼び出せないですよ。



Python が準備している関数やクラスは、プロが十分に検討して作ったものなので、みなさんの変数を破壊してしまうリスクは小さいと言えます。しかし、コード 6-8 のような意図しない破壊を確実に避けたい場合に用いるテクニックとして、**防御的コピー**があります。大事なデータは関数にそのまま渡すのではなく、複製したものを渡すことで、万が一破壊されても影響がないようにするというテクニックです。

コード 6-8 を防御的コピーを使って安全にしたものが、コード 6-9 です。

コード 6-9 防御的コピーを用いて悪影響を防ぐ

```

1  def add_suffix(names):
2      for i in range(len(names)):
3          names[i] = names[i] + 'さん'
4      return names
5
6  before_names = ['松田', '浅木', '工藤']
7  copied_names = list()
8  for n in before_names:
9      copied_names.append(n)
10 after_names = add_suffix(copied_names)
11 print('さん付け後:' + after_names[0])
12 print('さん付け前:' + before_names[0])

```

渡されたリスト内の名前に「さん」を付ける関数

リストを複製する

複製したリストを関数に渡す

add_suffixから返されたリスト内の先頭の要素

実行結果

さん付け後:松田さん

さん付け前:松田

ポイントは7～9行目です。before_names が指すリストオブジェクトを丸ごと複製して copied_names という別のリストオブジェクトを作り、add_suffix 関数に引き渡しています。この方法であれば、add_suffix 関数による動作が before_names に影響を及ぼすことはありません(図 6-7)。

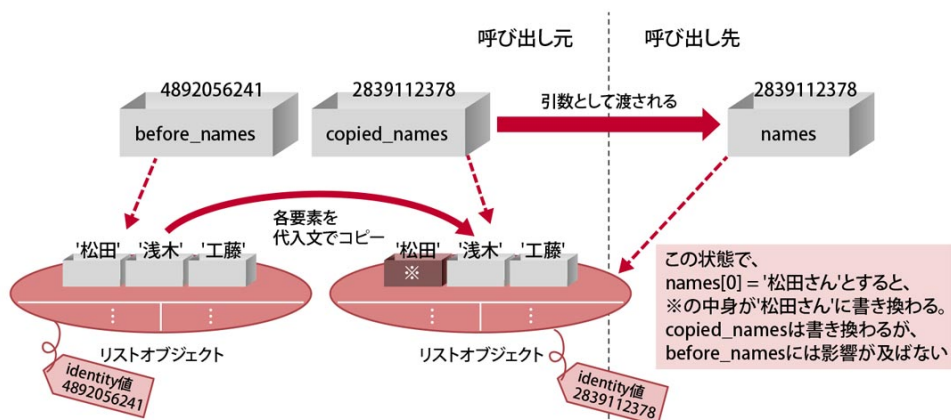


図 6-7 防御的コピーを用いて悪影響を防ぐ



防御的コピー

変数を複製してから関数に引き渡せば、万が一壊されても元の変数には影響が及ばない。



ちなみに、コード 6-9 の7～9行目は、copy メソッドを使った「before_names.copy()」やスライス記法(P090)を用いた「before_names[:]」でも実現できる。現場ではそのほうがよく見かけるだろう。

list オブジェクトは、自身の中身を複製した別のリストオブジェクトを返す copy メソッドを持っていますが、すべての型のオブジェクトが copy メソッドを持つとは限りません。もし持たない場合は、コード 6-9 の7～9行目のように、

開発者が内容を 1 つずつコピーするか、Python が提供する copy モジュールを利用して複製します (モジュールについては第 7 章で紹介)。

6.3.5 不変オブジェクト



工藤さん！ なぜか独立性が崩れないケースを見つけちゃいました！

松田くんは、次のコード 6-10 で、なぜか変数の独立性が崩れないことを発見したようです。先ほどのコード 6-8 (P264) とほぼ同じですが、add_suffix 関数に渡す引数が、リストではなく文字列に変わっています。

コード 6-10 文字列を引き渡しても悪影響が起きない

```

1 def add_suffix(name):
2     name = name + 'さん'
3     return name
4
5 before_name = '松田'
6 after_name = add_suffix(before_name)
7 print('さん付け後:' + after_name)
8 print('さん付け前:' + before_name)
```

渡された名前に「さん」を付ける関数

文字列を渡している

add_suffix から返された名前

add_suffix に渡した名前

実行結果

さん付け後: 松田さん

さん付け前: 松田

防御的コピーをしていないのに「さん」が付かない

確かにこのケースでは、防御的コピーはしておらず、add_suffix 関数の中で引数 name を書き換えているため、同じオブジェクトを指す before_name も書き

換えられてしまうと予測できます。しかし実行結果を見る限り、`add_suffix` 関数で行った書き換えは呼び出し元の `before_name` には影響を及ぼしておらず、独立性が保たれています。



どうして？ 防御的コピーをしないと、てっきり呼び出し先にデータを壊されてしまうと思っていたのに…。

これこそ、この章で最後に学ぶ、「落とし穴と見せかけて落とし穴じゃない」という特殊なパターンなんだ。



Python に登場するすべてのオブジェクトは、書き換えができるかできないかによって、次の 2 種類に分類することができます。

6 章



不変性によるオブジェクトの分類

- **可変** (mutable) オブジェクト : 中身の値や属性を書き換えられる。
- **不変** (immutable) オブジェクト : 中身の値や属性を書き換えられない。

あるオブジェクトが可変か不変かは、そのオブジェクトの型によって決まります。ほとんどの型は基本的に可変ですが、一部は不変な型として作られています。



オブジェクトが不変となる代表的な型

- `int` 型、`str` 型、`bool` 型などの一部の標準的な型
- `tuple` 型 (コレクションのタプル)



え？ 私、文字列型の変数とか普通に書き換えてますよ。これのどこが不変なんですか？

浅木さんと同じ疑問を感じた人は、次のコード 6-11 を実行して確かめてみてください。

コード 6-11 リストと文字列による書き換え時の identity 値の変化

```

1 names = list() # リストの場合
2 print('変更前のlistのidentity: {}'.format(id(names)))
3 names.append('松田')
4 print('変更後のlistのidentity: {}'.format(id(names)))
5
6 name = '松田' # 文字列の場合
7 print('変更前のstrのidentity: {}'.format(id(name)))
8 name = 'スーパー' + name
9 print('変更後のstrのidentity: {}'.format(id(name)))

```

実行結果

変更前のlistのidentity: 4462142594

変更後のlistのidentity: 4462142594) identity は変わっていない

変更前のstrのidentity: 4462142720

変更後のstrのidentity: 4464511792) identity が変わっている



リストは要素を追加しても identity は変わっていないのに、文字列のほうは変わっています！ ということは…、別のオブジェクトになっちゃったってことじゃないですか！

そうなんだ。「松田」が「スーパー松田」に変わったんじゃない。
「松田」とは違う、別の新しい「スーパー松田」が生まれたんだ。



int 型や str 型のような不変オブジェクトは、一度オブジェクトが生まれたら、その後は絶対に内容を書き換えられないように設計されています。そのため、コード 6-11 のように無理矢理その内容を書き換えようとすると、書き換え後の内容を持つ別のオブジェクトがその場で生み出されることになっています (図 6-8)。

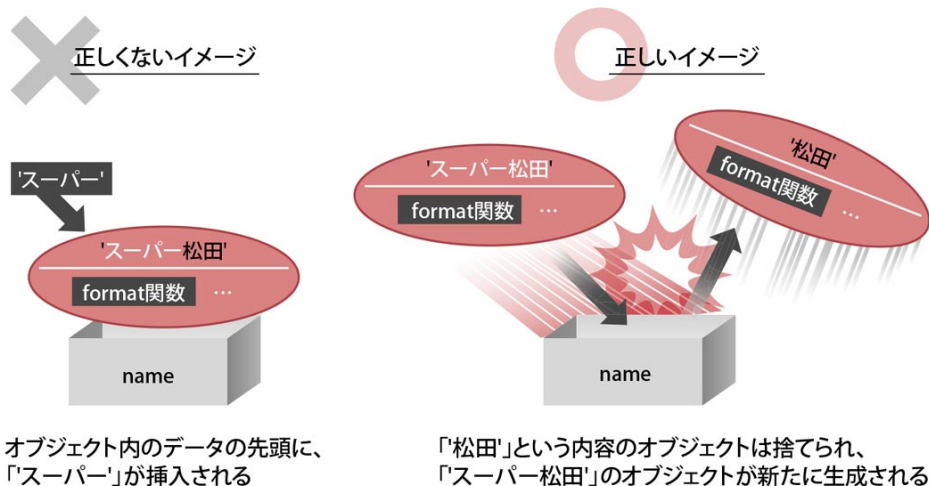


図 6-8 不変オブジェクトに対する書き換え



不変オブジェクトを書き換えたときの動作

- ・ 不変オブジェクトを書き換えようとすると、別のオブジェクトとして生まれ変わる。
- ・ 元のオブジェクトは変化することなく、捨てられる。

不変オブジェクトのこのような特性を理解すれば、なぜコード 6-10 で独立性が崩れなかったかは、次のように説明がつけます。

- 1 行目で受け取った引数 `name` には、呼び出し元の `before_name` と同じ `identity` 値 (例:1111) が格納され、「松田」という内容の文字列オブジェクトを指している。
- 2 行目で `name` を書き換えようとした際、文字列型は不変オブジェクトであるため、「スーパー松田」という別のオブジェクトが生まれ、その `identity` 値 (例:2222) が `name` に代入される。
- `before_name` が指すオブジェクト (例:1111) の内容は、「松田」のまま変化していない。



不変オブジェクトは、開発者が防御的コピーをしなくても勝手に増殖してくれるから悪影響が起きないんだよ。

なるほど。int や str は気軽に関数に渡してもいいけど、タプル以外のコレクションは要注意ってことですね。



不変オブジェクトは安全に使用できる

不変オブジェクトは、関数に引き渡しても、呼び出し先で書き換えられる心配がない。



これでやっと、関数に関する謎はすべて解けましたね！

そうだね、おめでとう。次章では、関数を本格的に連携させて実用的なプログラムを開発するための方法を紹介するよ。





Column

捨てられた不変オブジェクトの行方

内容が書き換わったために捨てられた不変オブジェクトは、その identity 値を格納した変数が存在しなくなるため、オブジェクトへアクセスできる一切の手段が失われます。使われなくなったオブジェクトは、ガベージコレクションというしくみによって、そのオブジェクトが占有していたメモリ領域が自動的に解放され、消滅します。



Column

破壊的な関数

受け取った引数の内容を内部で書き換えてしまう関数を破壊的な関数（メソッド）といいます。破壊的な関数に可変オブジェクトを渡すと、本章で紹介した副作用が生じます。ある関数が破壊的かそうでないかは、関数名だけではわからないことが多いため、重要なプログラムで利用する場合はドキュメントで調べる必要があります。ドキュメントが手に入らない場合は、念のため防御的コピーで対処します。

6.4

第 6 章のまとめ

この章では、次のことを学びました。

オブジェクトの基本構造

- Python では、整数や文字列などの値はすべてオブジェクトとして扱われる。
- オブジェクトは、データと、そのデータを処理するための関数を持っている。
- オブジェクトに属するデータを属性、関数をメソッドという。

クラスと型

- オブジェクトは基本的に設計図であるクラスから生み出される。
- クラス名と同名の関数を用いることで、オブジェクトを生み出すことができる。
- リテラルは、その表記法によって特定のクラスからオブジェクトを生み出す指示である。
- オリジナルのクラスを定義して利用することもできる。

identity と参照

- すべてのオブジェクトは自動的に与えられた一意の数値 (identity) を持つ。
- 変数は、オブジェクト自体ではなく、その identity を内部に格納する。
- 引数や戻り値に identity が受け渡されることによって、変数の独立性が崩れる可能性がある。

オブジェクトの不変性

- int 型、str 型、tuple 型などの不変オブジェクトの内容は決して変化しない。
- 不変オブジェクトへの書き換えは、別のオブジェクトの生成として実現される。
- 不変オブジェクトを関数へ引き渡しても、変数の独立性は崩れない。

6.5

練習問題

練習 6-1

次のコードに登場する変数 a、b、c に格納されるオブジェクトについて、そのオブジェクトの「型」、「可変」または「不変」オブジェクトのどちらであることを答えてください。

```
1 a = 'Python'
2 b = [1, 3, 5]
3 class MyClass:
4     def hello(self):
5         print('Hello' + a)
6 c = MyClass()
7 c.hello()
```

6
章

練習 6-2

次のコードを実行し、キーボードから ABC と入力した結果、画面に表示される内容を答えてください。

```
1 x = ['ABC']
2 y = [input()]
3 print(x[0] == y[0])
4 print(id(x[0]) == id(y[0]))
5 y = x
6 y[0] = 'XYZ'
7 print(x[0])
```

練習 6-3

次のコードは意図したとおりに動作しません。現れる症状とその原因を説明したうえで、どのように修正すればよいかを教えてください。なお、welcome 関数の内容は変更できないものとします。

```
1  def welcome(u):
2      print('ようこそ{}さん'.format(u['name']))
3      u['age'] = u['age'] + 1
4      print('あなたは来年{}歳だから大吉です!'.format(u['age']))
5
6  username = input('名前を入力してください >>')
7  userage = int(input('年齢を入力してください >>'))
8  user = {'name': username, 'age': userage}
9  welcome(user)
10 print('{}歳の{}さん、またプレイしてくださいね'
      .format(user['age'], user['name']))
```

6.6

練習問題の解答

練習 6-1

a:str 型(不変) b:list 型(可変) c:MyClass 型(可変)

練習 6-2

実行結果

True

False

XYZ

6
章

練習 6-3

問題文のコードでは、可変オブジェクトであるディクショナリを `welcome` 関数の引数として渡しているため、参照を渡すことによる独立性の崩壊が発生します。具体的には、3 行目でディクショナリの中身を変更しているため、その影響が呼び出し元の変数 `user` にも及び、7 行目で入力した年齢が書き換えられてしまいます。この影響を回避するためには、防御的コピーを活用するように、9 行目の「`welcome(user)`」を次の 2 行に置き換えます。

```
copied_user = user.copy()
welcome(copied_user)
```



Column

不変オブジェクトの再利用

近年の Python は、同じ内容の不変オブジェクトが生成されようとすると、できるだけ同じオブジェクトを再利用することを試み、コンピュータのメモリを節約しようとしています。たとえば、「a = 3」に続いて「b = 1 + 2」という文を実行すると、b のために新たに int 型オブジェクトは生成されず、a が指しているオブジェクトを b も指すようになります。

このような節約術で実害が出ないのも、「決して書き換えられず、万が一書き換えられそうになったら別の存在として増殖する」という不変オブジェクトの特性によるものです。

第7章

モジュール

Python で実用的なプログラムを作るには、
開発を手助けしてくれるさまざまな部品を
積極的に活用することが欠かせません。

この章では、Python で利用できる部品の種類と
その使い方を紹介していきます。

CONTENTS

- 7.1 部品を使おう
- 7.2 組み込み関数
- 7.3 モジュールの利用
- 7.4 パッケージの利用
- 7.5 外部ライブラリの利用
- 7.6 第7章のまとめ
- 7.7 練習問題
- 7.8 練習問題の解答

7.1 部品を使おう

7.1.1 Python で使える部品たち

通常、プログラミング言語には、開発を助けてくれるさまざまな部品が用意されています。私たちは、このような部品を積極的に用いることで、開発効率を大きく引き上げることができます。



自分で関数を作ることを部品化と説明したけど (P197)、この章で紹介する部品は、関数が集まったさらに大きな部品ととらえれば OK だよ。

Python で利用できる部品は、その所属している場所によって、図 7-1 のように俯瞰できます。それぞれの特徴と基本的な使い方を順に紹介していきます。

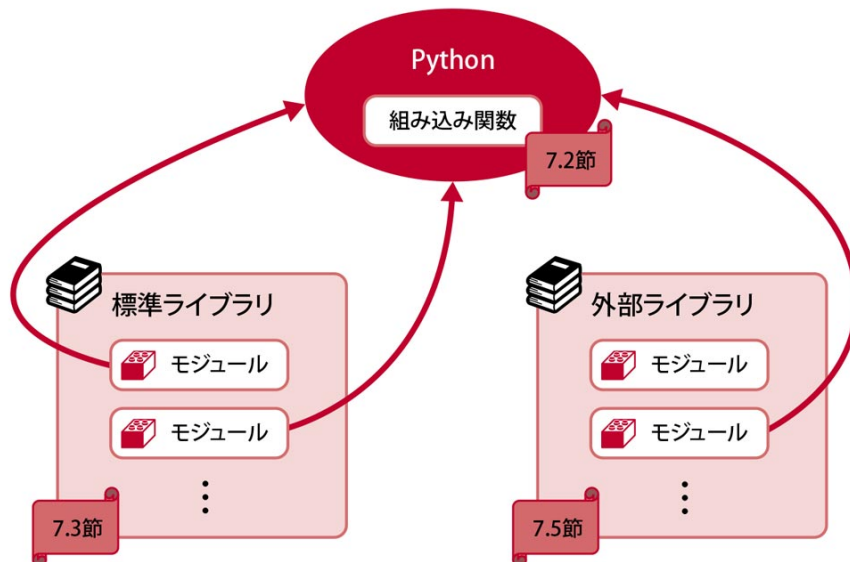


図 7-1 Python には便利な部品がたくさん用意されている

7.2

組み込み関数

7.2.1

組み込み関数とは



まずは組み込み関数からいくよ。今までにもよく使ってきた、おなじみの関数だね。

組み込み関数 (built-in functions) とは、Python 自体に組み込まれており、いつでも自由に呼び出せる関数のことです。print 関数や input 関数など、どのようなプログラムを開発するにも必要となる基本的な機能が組み込み関数として提供されています(表 7-1)。

7章

表 7-1 主な組み込み関数

分類	組み込み関数名	機能
入出力	print 関数	引数を標準出力*に出力する
	input 関数	標準入力*から 1 行読み込んだ結果を返す
データ型	type 関数	引数のデータ型を返す
	int 関数	引数を整数に変換した結果を返す
	float 関数	引数を小数に変換した結果を返す
	str 関数	引数を文字列に変換した結果を返す
コレクション	list 関数	引数でリストを作成して返す
	dict 関数	引数でディクショナリを作成して返す
	tuple 関数	引数でタプルを作成して返す
	len 関数	引数の長さ(要素数)を返す
	sum 関数	引数の合計値を返す
	max 関数	引数の最大値を返す
	min 関数	引数の最小値を返す
計算	abs 関数	引数の絶対値を返す
	round 関数	引数を四捨五入した値を返す
ファイル	open 関数	引数で指定したファイルを開く

※通常、標準入力とはキーボード、標準出力はディスプレイ(画面)を指す。

Python には、表 7-1 に記載したもの以外にも、70 近くもの組み込み関数が用意されています。より詳細な情報は、付録 A で紹介しているリファレンスや公式サイトを参照してください。

7.2.2 ファイル入出力



それじゃあ、新しい組み込み関数を 1 つ使ってみようか。open 関数を使ってファイルにデータを書き込もう。

Python に限らず、プログラムからファイルの読み書きをするには、通常、次の手順で行います。

手順① ファイルを開く

手順② ファイルに書き込む、またはファイルから読み込む

手順③ ファイルを閉じる



僕たちがパソコンでファイルを使用する手順とまったく同じですね。

コード 7-1 は、入力された内容をファイルに書き込みます。1 行の日記など、簡単なメモを記録しておくイメージのプログラムです。

コード 7-1 1 行日記を記録する

```
1 text = input('何を記録しますか? >>')
2 file = open('diary.txt', 'a')
3 file.write(text + '\n')
4 file.close()
```

手順① ファイルを開く (追記モード)

手順② ファイルに書き込む

手順③ ファイルを閉じる

実行結果 (1 回目)

何を記録しますか？ >>7月15日は、朝香先輩ともつ鍋パーティー

実行結果 (2 回目)

何を記録しますか？ >>7月16日は、ひとり焼き鳥&ボーリング



先輩、焼き鳥はともかく、ボーリングもひとりで行ったんですか…？

あら、社会人のたしなみよ。点数を素数で揃えていくの、楽しいんだから。



7章

このプログラムを実行すると、同じフォルダにテキストファイル「diary.txt」が作成されます。ファイルを開くと、次のように書き込まれていることが確認できるでしょう (JupyterLab では、ノートブックの左側に同じフォルダ内の一覧が表示されています)。

diary.txt の内容

7月15日は、朝香先輩ともつ鍋パーティー	1 回目の実行結果
7月16日は、ひとり焼き鳥&ボーリング	2 回目の実行結果

「手順① ファイルを開く」では、組み込み関数である open 関数を使用しています (コード 7-1 の 2 行目)。第 1 引数として開くファイルの名前、第 2 引数としてモードを指定します。

モードは、開いたファイルをどのように操作するかを示す文字です。コード 7-1 のように「a」 (追記モード) を指定すると、操作対象のファイルが存在しない場合には、ファイルを新しく作成して開き、ファイルの末尾にデータを書き込んでいきます。



このプログラムのポイントは変数 `file` に代入している `open` 関数の戻り値だ。これは、第 6 章で紹介したオブジェクトだよ。

`open` 関数は、戻り値として開いたファイルを表すファイルオブジェクトを返します。ファイルオブジェクトは、開いたファイルに関するさまざまな属性や、操作を行うためのメソッドを持っています。

「手順② ファイルに書き込む」では、ファイルオブジェクトが持つ `write` メソッドを使って、開いたファイルにデータを書き込んでいます(コード 7-1 の 3 行目)。入力された内容に改行を表すエスケープシーケンス「`\n`」(P039)を付けているため、書き込むたびに改行されるというわけです。もし「`\n`」を付けないと、「7 月 15 日は、朝香先輩ともつ鍋パーティー 7 月 16 日は、ひとり焼き鳥&ボーリング」のように、改行されずにファイルに書き込まれます。



ファイルを開く

`open(ファイル名, モード)`

※ファイル名に開くファイルを指定する。

※モードにはどのようにファイルを操作するかを指定する。

`r` : 読み込み (指定したファイルが存在しない場合はエラー)

`w` : 書き込み (指定したファイルが存在しない場合は新規作成)

`a` : 追記 (指定したファイルが存在しない場合は新規作成)

※戻り値としてファイルオブジェクトが返される。



ファイルに書き込む

ファイルオブジェクト `.write(書き込む内容)`



ファイルを閉じる

ファイルオブジェクト .close()

開いたファイルを閉じるには、ファイルオブジェクトの `close` メソッドを使用します。ファイルをプログラム内で閉じなかった場合は Python が自動的に閉じますが、そのタイミングは Python に任されており、また必ず閉じるという保証也没有ありません。必要なファイル操作が終わったら、すぐに閉じる処理が行われるようにしたほうがよいでしょう。



そう言われても、うっかり閉じるのを忘れちゃいそうだな…。

そんな松田くんのために、いい方法があるよ。

**7
章**

開いたファイルを確実に閉じたい場合は、**with 文**を使いましょう。with 文でファイルを開き、with ブロック内で開いたファイルに関する操作を行います。そして、with ブロックが終了すると、自動的にファイルを閉じてくれるといった具合です(コード 7-2)。

コード 7-2 用が済んだらすぐに閉じる

```
1 text = input('今日は何をした? >>')
2 with open('diary.txt', 'a') as file:
3     file.write(text + '\n')
```

ファイルオブジェクトを
代入する変数

ファイルを開く処理



with 文

with ファイルを開く処理 as 変数 : ファイル进行操作する処理

※開く処理で返されるファイルオブジェクトが変数に代入される。

※ with ブロックの終了時にファイルを自動的に閉じる処理が行われる。

※処理はインデントして記述する。



Column

ストリーム

プログラムは通常、キーボードや画面、ファイルなどとデータをやりとりして動作しますが、そのようなデータの流れを**ストリーム** (stream) といいます。大規模なシステムでは、さらにデータベースやネットワークといった外部資源ともストリームでつながることがあります。

ストリームは、不必要に開き続けると、ほかのプログラムやアプリケーションからその資源へアクセスできなくなる、メモリなどのリソースを使い果たしてシステムダウンするなどの問題が発生することがあります。「開いたら必ず閉じるべきもの」をプログラム内で扱う場合には、with 文を用いて確実に閉じるようにしましょう。

7.3 モジュールの利用

7.3.1 モジュールとは

前節では、Python に最初から組み込まれている関数について紹介しました。これらの関数は、いつでもどこでも好きなときに呼び出せることが特徴でした。このような関数以外にも、明示的に宣言することによって、Python やほかの開発者が用意してくれた変数や関数、クラスを自由に自分のプログラムに取り入れることができます。



えっ、ほかの人が作ったプログラムを自分のプログラムに入れるんですか？

そうだよ。すでに開発された安全で性能の高いプログラムを活用して、さらに便利で面白いプログラムを効率的に作っていかないか。



7章

モジュール (module) は、別のプログラムに取り込んで使うことを前提として、ある機能をひとまとめにした 1 つのファイルを指す言葉です。関連した機能ごとに、変数や関数などの細かな部品を集めた 1 つの大きな部品ととらえることができます。



モジュールの役割

モジュールは、変数や関数 (または、それらをまとめたクラス) を提供する。

モジュールは、それを開発した人によって 2 種類のライブラリに属しています。
ライブラリ (library) とは、複数のモジュールがまとまったものと考えとよいでしょう。いわばライブラリは、さまざまな道具が詰まった便利な工具箱です。



モジュールが属する 2 種類のライブラリ

- ・ **標準ライブラリ** : Python が公式に用意したモジュールのまとめ
- ・ **外部ライブラリ** : 別の組織や個人が用意したモジュールのまとめ

この節では、まずは標準ライブラリに属するモジュールの使い方について紹介していきます。



大きなプロジェクトの場合、共通して使う機能をモジュールとして開発しておき、チームで共有して使っていくことが一般的だよ。



Column

車輪の再発明

他人が用意したプログラムを利用することに抵抗を覚える人もいるかもしれません。しかし、すでに存在しているプログラムを再び一から作ることを「車輪の再発明」といい、IT 業界では避けるべきこととされています。なぜなら、稼働実績のあるプログラムはバグが少なく、その性能も改良を重ねていることが多いので、初心者が同レベルのものを業務目的で作成すると、時間とお金のムダになってしまうからです。ただし、技術習得などの学習目的のために、あえて再発明をすることはムダではなく、むしろよい訓練となるでしょう。

7.3.2 標準ライブラリ

Python が用意した標準ライブラリでは、主に表 7-2 のようなモジュールが提供されており、各モジュールには、特定の用途ごとに役立つ変数や関数、クラスが定義されています。たとえば math モジュールの場合は数学計算に関する変数や関数、datetime モジュールには日付と時間の処理に関するクラスが定義されているといった具合です。

表 7-2 標準ライブラリに含まれる主なモジュール

モジュール	用途
math モジュール	数学計算に関する処理
random モジュール	乱数に関する処理
datetime モジュール	日付と時間に関する処理
email モジュール	電子メールに関する処理
csv モジュール	CSV ファイルに関する処理
json モジュール	JSON ファイルに関する処理
os モジュール	OS 操作に関する処理

目的に合ったモジュールを取り込んで、定義されている関数などを利用することで、専門的な知識がなくても高度な処理を実現できるようになります。たとえば、三角関数の計算をしたいなら、その計算方法をいちいち記述しなくても、math モジュールを取り込んで sin 関数や cos 関数を呼び出すだけでよいのです。

7.3.3 モジュールの取り込み



あらっ、数学計算ができるモジュールもあるんですね！ 早く使ってみましょうよ！

まあまあ、慌てない慌てない。モジュールを使うには 1 ステップ必要なんだ。



浅木さんの期待に応えて、実際に math モジュールを例に基本的な使い方を紹

介していきましょう。モジュールを自分のプログラムに取り込むには、最初にそのモジュールを使うことを明示的に宣言する必要があります(コード 7-3)。

コード 7-3 math モジュールを利用する

```
1 import math
```

math モジュールを取り込むことを宣言

```
2 print('円周率は{}です'.format(math.pi))
```

math モジュールの
変数 pi を参照

```
3 print('小数点以下を切り捨てれば{}です'  
      .format(math.floor(math.pi)))
```

math モジュールの
floor 関数を呼び出す

```
4 print('小数点以下を切り上げれば{}です'  
      .format(math.ceil(math.pi)))
```

math モジュールの
ceil 関数を呼び出す

実行結果

円周率は3.141592653589793です

小数点以下を切り捨てれば3です

小数点以下を切り上げれば4です

math モジュールを取り込むことを、1 行目に記述した **import 文** で宣言しています。続けて 2 行目で、モジュールが提供する変数 pi を参照しています。この変数にはその名のとおりに、あらかじめ円周率が代入されています。3、4 行目では、math モジュールが提供する関数を呼び出しています。floor 関数は引数を小数点以下で切り捨てた結果、ceil 関数は小数点以下で切り上げた結果を返します。

このように、モジュールに属する変数や関数は、その名前の前に「モジュール名 .」を付けることで参照したり呼び出したりすることができます。



モジュールを取り込む

import モジュール名

**モジュール内の変数を参照**

モジュール名・変数名

**モジュール内の関数を呼び出す**

モジュール名・関数名 (引数, ...)



組み込み関数と違って、モジュールを使うためには import 文が必要なんですね。

**7
章**

取り込んだモジュールに別の名前を付けることもできます。次のコードは、取り込んだ math モジュールに「m」という別名を付けている例です (コード 7-4)。

コード 7-4 math モジュールに別名を付けて利用する

```
1 import math as m
2 print('円周率は{ }です'.format(m.pi))
3 print('小数点以下を切り捨てれば{ }です'.format(m.floor(m.pi)))
4 print('小数点以下を切り上げれば{ }です'.format(m.ceil(m.pi)))
```

m (math モジュール)
の変数 pi を参照

math モジュールを m として取り込む

m (math モジュール) の
ceil 関数を呼び出す

m (math モジュール) の
floor 関数を呼び出す



別名を付けたモジュールの利用

import モジュール名 as 別名

※モジュール内の変数や関数は、別名を付けて参照する。



覚えにくい名前や長い名前のモジュールでも、別名を付けておけばシンプルなコードになりそうだな♪

7.3.4 特定の変数や関数だけを取り込む

モジュール全体ではなく、モジュールから特定の変数や関数だけを取り込むこともできます。コード 7-5 は、math モジュールから変数 pi と floor 関数だけを取り込んでいます。

コード 7-5 特定の変数や関数だけを利用する

```

1 from math import pi
2 from math import floor
3 print('円周率は{}'.format(pi))
4 print('小数点以下を切り捨てれば{}'.format(floor(pi)))

```

Diagram annotations for the code block:

- Line 1: `from math import pi` → math モジュールから変数 pi を取り込む
- Line 2: `from math import floor` → math モジュールから floor 関数を取り込む
- Line 3: `format(pi)` → 取り込んだ変数 pi を参照
- Line 4: `format(floor(pi))` → 取り込んだ floor 関数を呼び出す

実行結果

円周率は3.141592653589793

小数点以下を切り捨てれば3です

3、4 行目に記述しているとおり、取り込んだ変数や関数はモジュール名を付けずに使用することができます。また、取り込んでいない変数や関数は使用できないので、ceil 関数を呼び出すことはできません。

**特定の変数や関数だけを取り込む****from モジュール名 import 変数名または関数名**

※取り込んだ変数や関数はその名前だけで参照できる。



これは便利ですよ！ 名前だけで使用できるなんて、自作の関数や組み込み関数を使うのと変わらないじゃないですか。

そうだね。でも、注意しないといけないことがあるんだ。

**7
章**

次のコード 7-6 を見てください。1 行目で math モジュールから対数を求める log 関数を取り込んでいます。しかし、2～3 行目で同じ名前の関数を定義しています。このように、名前が重複した場合には、あとで実行した関数定義が優先されるため、math モジュールの log 関数が呼び出せなくなってしまいます。

コード 7-6 関数名が重複すると…

```
1 from math import log
2 def log(msg):
3     print('{ }を記録します'.format(msg))
4 log(10)
```

math モジュールから log 関数を取り込む
ログ出力を行う自作の log 関数を定義
対数を求めるつもりが、ログが出力される

実行結果

10を記録します

このような事態を避けるためには、取り込んだ変数や関数をきちんと把握しておく必要があります。



えっ、僕そんなの自信ありません。



言うと思ったよ。だからこんな工夫をするんだ。

import 文は、モジュール内の変数や関数を利用する前ならば、どこにでも書くことができますが、一般的には、プログラムの先頭（最初のセル）にまとめて書くことが推奨されています。そうすることで、取り込んでいる変数や関数を把握しやすくなるからです（図 7-2）。

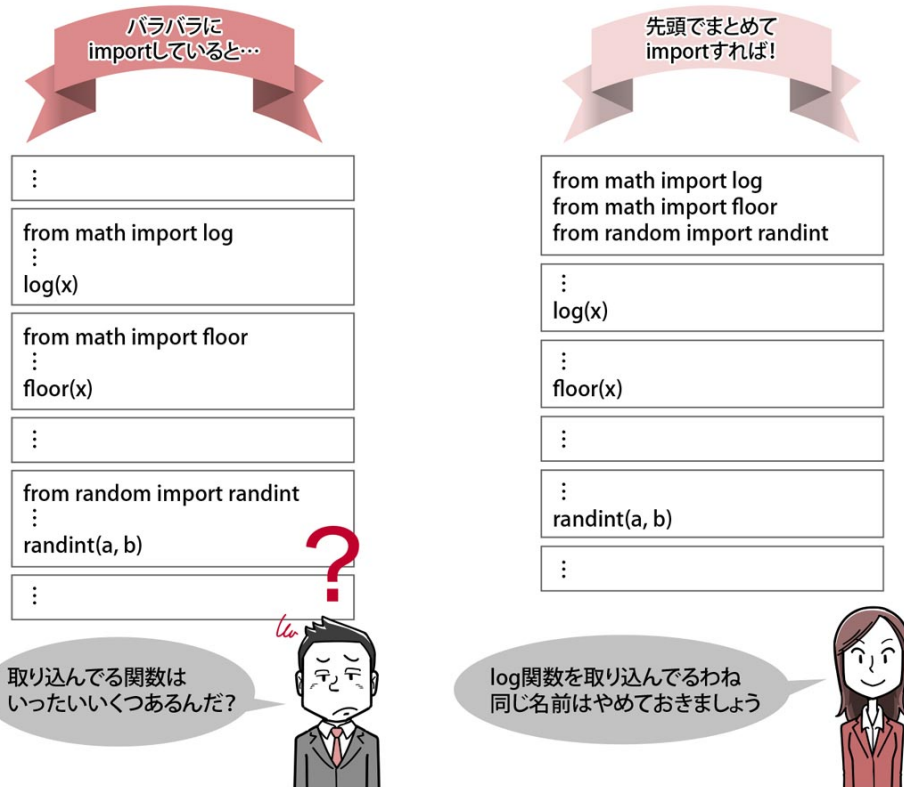


図 7-2 import 文はプログラムの先頭（最初のセル）に記述する

また、モジュールと同様に、「as」を使用して取り込んだ変数や関数に別名を付けることができます（コード 7-7）。これによって変数名や関数名の重複を避ける

ことができますが、別名があまりにも多すぎると、標準と異なる名前の変数や関数が増えてしまい、逆に混乱することがあるので注意しましょう。

コード 7-7 特定の変数や関数だけを別名を付けて利用する

```
1 from math import pi as ensyuritsu
2 from math import floor as kirisute
3 print('円周率は{}'.format(ensyuritsu))
4 print('小数点以下を切り捨てれば{}です'
      .format(kirisute(ensyuritsu)))
```

7章



特定の変数や関数だけを別名を付けて取り込む

from モジュール名 import 変数名または関数名 as 別名

※取り込んだ変数や関数は、別名だけで参照できる。



ハンドルネームで呼び合っていて、本名がわからなくなっちゃう状態みたいなものかしら。

7.3.5

ワイルドカードインポート



最後に、特殊な取り込み方法を紹介しておこう。ただし、これはあまりお勧めしないから、参考程度に知っておくだけでいいよ。

コード 7-8 では、* 記号を使ってモジュールの取り込みを宣言しています。このような取り込み方法を、**ワイルドカードインポート**と呼びます。

コード 7-8 ワイルドカードインポートを使ってモジュールを利用する

```

1 from math import *
2 print('円周率は{ }です'.format(pi))
3 print('小数点以下を切り捨てれば{ }です'.format(floor(pi)))
4 print('小数点以下を切り上げれば{ }です'.format(ceil(pi)))

```

math モジュールのすべての変数と関数を取り込む

よく見ると、特定の変数や関数だけを取り込む構文(7.3.4 項)と同じですから、指定したモジュール内のすべての変数と関数をその名前だけで使用できるようになります。とても便利な機能ではありますが、取り込んだ変数や関数の把握が難しくなり、意図せぬ名前の衝突が起きやすくなります。そのため、ワイルドカードインポートの使用は推奨されていません。



ワイルドカードインポート

from モジュール名 import *

※指定したモジュールのすべての変数と関数を取り込む。

※取り込んだ変数や関数はその名前だけで参照できる。



どんな機能があるかわからないのに全部取り込んじゃうなんて、ワイルドだろ～。

…ちなみに、* 記号のことをワイルドカードって呼ぶのよ。



7.3.6 モジュール取り込みのまとめ



うーん。いろんな取り込み方があって、だんだん混乱してきました…。

うん。ちょっと整理しておこうか。



この節で紹介したモジュールの取り込み方は次の5つであり、それぞれの方法によって、取り込まれる範囲や、取り込んだ変数や関数の参照方法が異なります(表 7-3)。

7章

方法① モジュールを取り込む

```
import モジュール名
```

方法② モジュールに別名を付けて取り込む

```
import モジュール名 as 別名
```

方法③ 特定の変数や関数だけを取り込む

```
from モジュール名 import 変数名または関数名
```

方法④ 特定の変数や関数だけを別名を付けて取り込む

```
from モジュール名 import 変数名または関数名 as 別名
```

方法⑤ ワイルドカードインポート (非推奨)

```
from モジュール名 import *
```


表 7-3 モジュール取り込みのまとめ

	取り込む範囲	取り込んだ変数の参照	取り込んだ関数の呼び出し
方法①	モジュール全体	モジュール名 . 変数名	モジュール名 . 関数名 ()
方法②	モジュール全体	別名 . 変数名	別名 . 関数名 ()
方法③	特定の変数または関数	変数名	関数名 ()
方法④	特定の変数または関数	別名	別名 ()
方法⑤	モジュール全体	変数名	関数名 ()



よく使うのは方法①と方法③だよ。まずはこれを優先的に覚えよう。

7.4

パッケージの利用

7.4.1 パッケージとは



外部ライブラリの紹介に入る前に、モジュールについてもう1つだけ補足しておこう。

モジュールは、ある機能をひとまとめにした1つのファイルと紹介しましたが(7.3.1 項)、いくつかのモジュールをまとめて**パッケージ**(package)を作ることができます。モジュールの実体はPython のプログラムファイルですが、パッケージの実体はフォルダです。

本書ではパッケージの作成方法を取り扱いませんが、業務システムの開発などで大量のモジュールを作成するような場合、関連するモジュールをパッケージにまとめることで、モジュールの管理が行いやすくなります。

また、標準ライブラリの中には、パッケージにまとめられているモジュールもあります。たとえば、http パッケージには、Web 通信に関するモジュールがまとめられています(図 7-3)。

7章

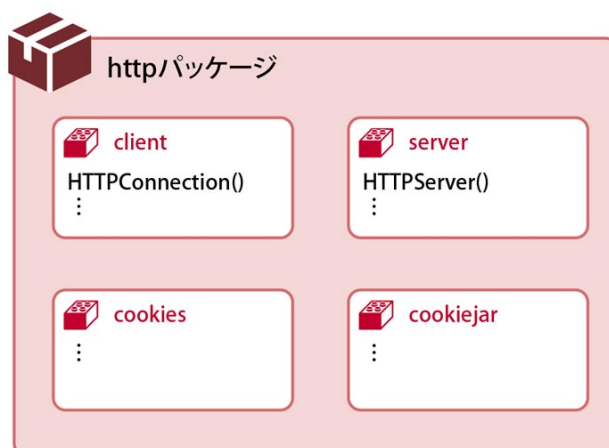


図 7-3 http パッケージ (標準ライブラリ)

7.4.2 パッケージ内のモジュールを取り込む

次のコード 7-9 ～コード 7-11 では、http パッケージ内の client モジュールが提供する HTTPConnection 関数を呼び出しています。この関数は、引数で渡された URL の Web ページにアクセスを行います。それぞれの import 文の違いで関数の呼び出し方が変わることに着目してください。

コード 7-9 http パッケージの client モジュールを取り込む

```
1 import http.client
2 conn = http.client.HTTPConnection('www.python.org')
3 :
```

コード 7-10 http パッケージの client モジュールを取り込む (from 利用)

```
1 from http import client
2 conn = client.HTTPConnection('www.python.org')
3 :
```

コード 7-9 とコード 7-10 は、client モジュール全体を取り込むことに変わりはありません。しかし、コード 7-9 での関数の呼び出しには、パッケージ名とモジュール名の両方の指定が必要です。コード 7-10 ではパッケージ名は不要となり、モジュール名を付けるだけでよくなります。



関数だけを取り込むことももちろん可能だ。

コード 7-11 http パッケージの client モジュールから関数だけを取り込む

```
1 from http.client import HTTPConnection
2 conn = HTTPConnection('www.python.org')
3 :
```

特定の関数だけを取り込んだ場合には、前節で紹介した方法③ (7.3.6 項) と同様に、関数名だけの指定で呼び出すことが可能になります。



パッケージ内のモジュールの取り込み／変数参照／関数呼び出し

`import パッケージ名 . モジュール名`

※ as で別名を付けることも可能。

`パッケージ名 . モジュール名 . 変数名`

`パッケージ名 . モジュール名 . 関数名 (引数 , ...)`



パッケージ内のモジュールの取り込み／変数参照／関数呼び出し (from 利用)

`from パッケージ名 import モジュール名`

※ as で別名を付けることも可能。

`モジュール名 . 変数名`

`モジュール名 . 関数名 (引数 , ...)`



パッケージ内のモジュールから特定の変数や関数だけを取り込む

`from` パッケージ名 . モジュール名 `import` 変数名または関数名

※取り込んだ変数や関数はその名前だけで参照できる。

※ `as` で別名を付けることも可能。



`import` 文の書き方によって関数の呼び出し方が変わるから混乱しちゃいます。何だか目が回ってきた…。

いい方法を紹介しよう。`import` の右に注目するんだ。



実は、関数を呼び出すときに指定するものは、`import` 文の右に書かれた内容と一致しています(図 7-4)。コード 7-9 ~コード 7-11 を見比べてみてください。

コード7-9

```
import http.client
```



```
conn = http.client.HTTPConnection(...)
```

コード7-10

```
from http import client
```



```
conn = client.HTTPConnection(...)
```

コード7-11

```
from http.client import HTTPConnection
```



```
conn = HTTPConnection(...)
```

実は、このルールは
パッケージを利用しないときでも
当てはまるんだ(表7-3参照)。
ただし、`as`を用いたときは
例外だよ



図 7-4 `import` 文と関数呼び出しの関係



標準ライブラリの基礎はこれで終わりだよ。自分の好きなモジュールをリファレンスで調べて、どんどん使ってみよう。

標準ライブラリには、非常にたくさんの数のモジュールが用意されています。いくつかのモジュールについては付録 A で紹介しているリファレンスを参考にしてください。標準ライブラリー覧や、各モジュールの詳細について知りたい場合は、Python の公式サイトを参照してください。



Column

組み込み関数の正体

組み込み関数は、「`__builtin__`」という名前のモジュールに定義された関数です。このモジュールは、プログラム実行時に自動的に取り込まれるため、定義された関数はモジュール名を指定しなくても呼び出せるというわけです。このようなことから、実は組み込み関数は標準ライブラリの一部ととらえることができます。

7章

7.5

外部ライブラリの利用

7.5.1

外部ライブラリとは



外部ライブラリって、標準ライブラリと何が違うんですか？
作った人が違うというのは以前教えてもらいましたけど。

誤解を恐れずに言うと、外部モジュールを使えるようになれば、
何だって作れるようになる。可能性が無限大に広がるんだ。



Python の世界では、個人や組織が作成したモジュールが公開されており、それらを取り込んで利用することもできます。これらのモジュールは外部ライブラリに属しています(7.3.1 項)。



標準ライブラリが純正パーツなら、外部ライブラリはサード
パーティーが作った追加パーツって感じですね。

うん、うまい例えだね。それこそ、数え切れないほど存在するよ。



外部ライブラリは膨大な数が存在しますが、その多くは標準ライブラリよりもさらに用途を絞り込んだものとなっています。いわば、その道に熟練した職人のための道具とも言えるでしょう。そのため、利用する場面は限定的ですが、標準ライブラリに用意されたモジュールよりも高度な処理を簡単に行うことができるという特徴を持っています(表 7-4)。

表 7-4 代表的な外部ライブラリ

モジュール名 (パッケージ名)	主な用途
matplotlib	データの可視化
Pandas	データ解析
NumPy	ベクトル・行列計算
SciPy	科学技術計算
SymPy	代数計算
scikit-learn	機械学習
TensorFlow	深層学習
Pygame	グラフィックス、音声
dateutil	日付・時間
simplejson	JSON ファイル
pyYAML	YAML ファイル
requests	Web アクセス

外部ライブラリは提供している機能も盛りだくさんです。1冊丸ごと1つの外部ライブラリだけを取り扱った専門書があるものも珍しくありません。したがって、外部ライブラリは標準ライブラリに比べて、学習コストが高い傾向にあります。専門的に特化した職人の道具を自由自在に使えるようになるためには、それなりの訓練の時間が必要なのです。

また、どの外部ライブラリを用いるかは、プログラムの目的に大きく左右されます。たとえば、機械学習をしないのであれば scikit-learn の使い方を学習しても意味がありません。高い学習コストをムダにしないためにも、やみくもに手を出すのではなく、実現したいプログラムが明確になってから、目的の外部ライブラリを学んでも遅くはありません。

本節では、有名な2つの外部ライブラリの体験を通して、外部ライブラリの特徴と使い方のコツを紹介します。

7.5.2 外部ライブラリの準備

外部ライブラリを使うには、開発環境に事前にインストールしておく必要があります。インストールする方法はいくつかありますが、開発環境に沿った方法で行いましょう。もし開発環境に合っていない方法で外部ライブラリのインストールを行うと、開発環境が壊れてしまう可能性もあるので注意してください。



ええっ。うまくやれるかな。

大丈夫！ Anaconda を使っていればその心配はほとんどないよ。



本書を通して使用している Anaconda には、著名な外部ライブラリが同梱されています。そのため、Anaconda をインストールしていれば、それらの外部ライブラリは標準ライブラリのように取り込んで使用することができます。Anaconda に組み込まれている外部ライブラリは、「conda list」コマンドで確認することができます。



Column

外部ライブラリのインストール

外部ライブラリをインストールするには、一般的には「pip install モジュール名」コマンドを使用します。たとえば、NumPy という外部ライブラリをインストールする場合は、「pip install numpy」を実行します。

ただし、開発環境として Anaconda を使っている場合は、pip コマンドを使用すると環境が壊れてしまうことがあります。Anaconda に付属する「conda install パッケージ名」コマンドなら、安全に外部ライブラリをインストールすることができます。

7.5.3 matplotlib

1 つ目に紹介する外部ライブラリは matplotlib です (<https://matplotlib.org/>)。matplotlib は、データの可視化に関する関数を提供しています。

次のコード 7-12 は、リストに格納された松田くんの 1 年間の体重データを折れ線グラフで表すものです。

コード 7-12 matplotlib でリストのデータを可視化する

```

1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 weight = [68.4, 68.0, 69.5, 68.4, 68.6, 70.2, 71.4, 70.8,
            68.5, 68.6, 68.3, 68.4]
4 plt.plot(weight)

```

matplotlib パッケージ内の pyplot モジュールを別名 plt として取り込む

取り込んだ plt の plot 関数を呼び出す

1 行目は、matplotlib が作成したグラフを JupyterLab で表示するために必要な記述です。matplotlib を使用するときの「お約束」だと割り切ってしまったほうがいいでしょう。

外部ライブラリも標準ライブラリと同様に、import 文で取り込みます(2 行目)。実は、matplotlib は複数のモジュールをまとめたパッケージです。matplotlib に限らず、多くの外部ライブラリはパッケージで提供されています。

7 章



外部ライブラリはパッケージで提供されていることが多い

著名な外部ライブラリの多くはパッケージとして提供されているため、パッケージ名とモジュール名を指定して取り込む必要がある。

なお、matplotlib を利用する場合、pyplot モジュールに plt という別名を付けて取り込むことが、Python 開発者の間で慣習になっています。



慣習なんて、何だか古くさくていやだな。僕は、元の名前のままでいきます。その場合は、「pyplot.plot()」でいいんだな。

私はもっと短く「p」にしよう。となると、「p.plot()」ね。



もちろんそれでも動作するが、慣習には従っておいたほうがいいぞ。

もし、matplotlib を本格的に学ぶとしたら、公式サイトだけでなく解説サイトや専門書も調べるでしょう。これらのサイトや書籍のほとんどは、慣習に合わせた import 文を記述しています。もし、自分だけが慣習と異なる import 文を記述していると、サイトや書籍に掲載されたコードをそのまま使うことができず、いちいち修正しなければなりません。

前述したように、外部ライブラリは学習コストが高いので(7.5.1 項)、少しでも効率が上がるように、慣習に合わせた書き方をお勧めします。原則として、外部ライブラリ制作者の記法が慣習となっていることが多いので、公式サイトのマニュアルやサンプルコードを確認するとよいでしょう。



外部ライブラリの取り込みの書き方

外部ライブラリを取り込む import 文の記法は、慣習(制作者の記法)に従うことで、より効率的な開発や学習に結び付く。



「郷に入っては郷に従え」ですね。それで損するわけでもないし、ここは従っておきますか。

pyplot モジュールの plot 関数を使用することで、折れ線グラフを作成することができます(4 行目)。コード 7-12 を実行すると、図 7-5 のようなグラフが表示されます。なお、グラフ上の 0 は、1 月を表していることに注意してください。

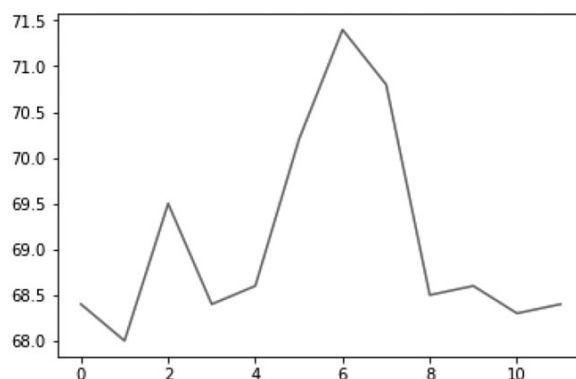


図 7-5 コード 7-12 の実行結果



何これ!? 6～8月の間だけ、体重が急増してるじゃない!

7章

いやー、夏はカレーがおいすぎて、ついつい食べすぎちゃうんですよね。



このように、データを可視化することで、データの大きさや変化の傾向を直感的にとらえることができるようになります。matplotlib では、折れ線グラフのほかにも、棒グラフ、円グラフ、ヒストグラム、散布図など、さまざまな可視化の方法を提供しています。Python を使ってデータ分析を行う場合、matplotlib はとても役に立つでしょう。



標準ライブラリには、こんなグラフを作成するモジュールはないからね。外部ライブラリの便利さが際立つ例と言えるだろう。

7.5.4 requests

2 つ目に紹介する外部ライブラリは requests です (<http://docs.python-requests.org/en/master/>)。requests は、Web ページへのアクセスに関する関数を提供し

ています。

次のコード 7-13 は、Python の公式サイト (<https://www.python.org/>) から、「Download」ページの内容を取得します。

コード 7-13 requests で Python の公式サイトを取得する

```
1 import requests  requests パッケージを取り込む
2 response = requests.get('https://www.python.org/downloads/')
3 text = response.text
4 print(text)
```

実行結果

```
<!doctype html>
<!--[if lt IE 7]>    <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">
<![endif]-->
<!--[if IE 7]>      <html class="no-js ie7 lt-ie8 lt-ie9">
<![endif]-->
<!--[if IE 8]>      <html class="no-js ie8 lt-ie9">
<![endif]-->
<!--[if gt IE 8]><!--><html class="no-js" lang="en" dir="ltr">
<!--<![endif]-->

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
(以下省略)
```

コード 7-13 の 1 行目でモジュールを取り込んでいますが、requests も正確にはパッケージです。



それなら、「import requests. モジュール名」にしないといけないんじゃないんですか？

実は、パッケージの作り方によっては、モジュール名を付けなくてもいい場合があるんだ。



パッケージの作り方によっては、import 文にパッケージ名だけを指定することで必要なモジュールを取り込むことができます。そのような記述が可能かどうかは、パッケージの公式サイトや解説書などで確認する必要があります。

2 行目では、requests が提供する get 関数を呼び出して、引数で指定した URL の Web ページにアクセスしています。この関数は、アクセスした Web ページに関するデータを格納した Response オブジェクトを返します。

7
章



オブジェクト…ってことは、アクセスした Web ページを表す属性やメソッドを持っているんですね。

3 行目で、Response オブジェクトの属性 text を参照しています。この属性は、アクセスした Web ページの内容 (HTML で書かれたテキスト) を持っており、4 行目でその内容を画面に表示しているというわけです。

requests を活用すれば、Web ページを巡回して情報を収集する「Web スクレイピング」というプログラムを作ったり、インターネット上で公開されている「WebAPI」というプログラムを利用したりすることができるようになります。



WebAPI を使うと、さまざまな Web サービスと自分のプログラムを連携させることができる。WebAPI はあらゆる分野で提供されているよ。

同様のことは、標準ライブラリの http.client モジュールや、urllib.request モジュールでも実現できます。次のコード 7-14 は、http.client モジュールを使っ

でコード 7-13 とまったく同じことを行っています。

コード 7-14 標準ライブラリを利用した Web ページの取得

```
1 import http.client
2 conn = http.client.HTTPSConnection('www.python.org')
3 conn.request('GET', '/downloads/')
4 response = conn.getresponse()
5 text = response.read().decode('UTF-8')
6 print(text)
7 conn.close()
```



うーん、何だか requests を使ったコードのほうがシンプルでわかりやすい気がします。

そうよね。もしもっと複雑な Web アクセス処理を作るとしたら、差は歴然たるものでしょうね。



このように、外部ライブラリの中には標準ライブラリと同じことをより簡単に実現できるようにしたものもあるんだよ。

7.6

第 7 章のまとめ

この章では、次のことを学びました。

組み込み関数

- 組み込み関数は、Python 自体に組み込まれており、特別な手続きをすることなく呼び出せる。
- print 関数や input 関数など、主にプログラムの基本的な機能が提供されている。

標準ライブラリ

- Python が用意したモジュールのまとまりを標準ライブラリという。
- 標準ライブラリに含まれるモジュールを取り込むには、import 文を記述する。

外部ライブラリ

- 第三者が用意したモジュールのまとまりを外部ライブラリという。
- 外部ライブラリは専門分野に特化した機能を提供しており、使いこなすには相当量の学習が必要である。
- 外部ライブラリを使うには、事前にインストールを行う。
- 外部ライブラリに含まれるモジュールを取り込むには、import 文を記述する。

パッケージ

- パッケージは複数のモジュールのまとまりであり、その実体はフォルダである。
- ライブラリの中には、パッケージとして提供されているものがある。

7.7

練習問題

練習 7-1

次の各文を読んで、正しいものは○、間違っているものは×を教えてください。
また、×としたものについては、その理由を説明してください。

- (1) 組み込み関数を利用するには、import 文を記述しなければならない。
- (2) モジュールとは、変数や関数、クラスをまとめたファイルである。
- (3) 標準ライブラリのモジュールから特定の関数だけを取り込むことはできない。
- (4) 外部ライブラリを利用するには、必ず事前にインストールが必要である。
- (5) 外部ライブラリは高度な機能を提供しているが、誰でも手軽に扱える。

練習 7-2

あるモジュール A に func 関数が定義されています。A を次のように import したとき、func 関数を呼び出すにはそれぞれどのような記述をしたらよいか教えてください。なお、func 関数は引数を受け取らないものとします。

- (1) import A
- (2) import A as B

練習 7-3

練習 7-2 の func 関数を呼び出す際、モジュール名を付けずに関数名単体で呼び出すには、モジュール A をどのように import すればよいか教えてください。

練習 7-4

次の機能を持つプログラムを、組み込み関数を使ってそれぞれ作成してください。ただし、いずれも必ず指定の数値が入力されることを前提とします。

- (1) 入力された 3 つの整数のうち、大きい値を表示する。
- (2) 円周率 3.141519 について、小数点以下第 1 位から第 5 位を四捨五入した値をそれぞれ表示する。

練習 7-5

open 関数を使ってファイルを読み込むには、次のように記述します。

```
1 file = open('sample.txt', 'r')
2 for line in file:
3     print(line)
4 file.close()
```

読み込んだファイルを 1 行ずつ表示する

これを参考にして、ファイルをコピーするプログラムを作成してください。

練習 7-6

標準ライブラリに含まれる random モジュールの randint 関数は、第 1 引数と第 2 引数に渡した整数の範囲内でランダムな整数を返します。この関数を使って、(1)～(7)のように動作する数当てゲームのプログラムを作成してください。

- (1) 画面に「数当てゲームを始めます。3 桁の数を当ててください!」と表示する。
- (2) リスト answer を準備して、0 ～ 9 のランダムな整数を 3 つ格納する。
- (3) 画面に「○桁目の予想を入力 (0 ～ 9) >>」と 3 回表示し、それぞれ入力された数をリスト prediction に格納する。
- (4) リスト answer と prediction を比較し、位置と数値が一致する要素と、位置は異なるが数値が一致する要素を数え、それぞれの結果を「○ヒット! ○ボール!」と画面に表示する。
- (5) 3 ヒットなら続けて画面に「正解です!」と表示してゲームを終了する。
- (6) それ以外の場合は、「続けますか? 1:続ける 2:終了 >>」を表示する。
- (7) 1 を入力されたら (3) に戻る。2 が入力されたら正解を表示してゲームを終了する。

7.8

練習問題の解答

練習 7-1

- (1)×：組み込み関数は import 文を記述しなくてもいつでも呼び出せる。
- (2)○
- (3)×：特定の変数や関数、クラスだけを取り込んで利用することができる。
- (4)○
- (5)×：外部ライブラリは高度な機能を提供するゆえに、それ相応の学習が必要。

練習 7-2

- (1)A.func() (2)B.func()

練習 7-3

from A import func

または

from A import *

練習 7-4

- (1)

```
1  nums = list()
2  for n in range(3):
3      data = int(input('{}個目の整数を入力してください >>'
                        .format(n + 1)))
4      nums.append(data)
5  print(max(nums))
```

(2)

```
1 pi = 3.141519
2 print(round(pi))
3 for n in range(4):
4     print(round(pi, n + 1))
```

練習 7-5

```
1 file_r = open('sample.txt', 'r')
2 file_w = open('copy.txt', 'w')
3 for line in file_r:
4     file_w.write(line)
5 file_r.close()
6 file_w.close()
```

読み込んだファイルを1行ずつ新しい
ファイルに書き込む

**7
章****練習 7-6**

```
1 # randomモジュールのrandint関数を取り込む
2 from random import randint
3 print('数当てゲームを始めます。3桁の数を当ててください！')
4
5 # 正解を作成
6 answer = list()
7 for n in range(3):
8     answer.append(randint(0, 9))
9
10 is_continue = True
11 while is_continue == True:
12     # 予想の入力
```

```
13     prediction = list()
14     for n in range(3):
15         data = int(input('{}桁目の予想入力 (0～9) >>'
16                             .format(n + 1)))
17
18         prediction.append(data)
19
20     # 答え合わせ
21     hit = 0
22     blow = 0
23     for n in range(3):
24         if prediction[n] == answer[n]:
25             hit += 1
26         else:
27             for m in range(3):
28                 if prediction[n] == answer[m] and n != m:
29                     blow += 1
30
31     # 結果発表
32     print('{}ヒット！{}ボール！'.format(hit, blow))
33     if hit == 3:
34         print('正解です！')
35         is_continue = False
36     else:
37         if int(input('続けますか？ 1:続ける 2:終了 >>')) == 2:
38             print('正解は{}{}{}でした'
39                     .format(answer[0], answer[1], answer[2]))
39         is_continue = False
```

※ 2行目を「import random」とした場合は、8行目の関数の呼び出しを「random.randint」とする。

第8章

まだまだ広がる Python の世界

第II部で学んできたモジュールやライブラリは、
Python という世界の広さを感じさせるものでした。
現状でも広く感じられる Python 活用の場は、
今、さまざまな分野へと急速に広がっています。
みなさんが本書を通して切り拓いた道のさらに先へと
広がる世界を眺め、最終章を締めくくりましょう。

CONTENTS

- 8.1 Python の可能性
- 8.2 Python の基礎を学び終えて

8.1

Python の可能性

8.1.1

まだまだ広がる Python の世界



2 人ともお疲れさま。Python 入門の旅はどうだったかな？

Python ならいろいろなことができそうで、ワクワクしています。早く続きを勉強したいです！



僕はまだ、コレっていうものは見つかってないけど、チャットボットっぽいものが作れて楽しかったです。

Python 入門の旅もいよいよ終わりに近づいてきました。画面に「Hello, World」と表示したあの頃がずいぶんと昔のように感じられたとしたら、それはみなさん自身が大いに成長したからにほかなりません。

本書の最終章となるこの章では、まだやりたいことが決まっていない松田くんのためにも、まだまだ広がる Python の世界を少しずつ巡って、入門の旅を終えることにしましょう。



よーし、興味を持てそうなものがあるか、ヒントを探してみます！

8.1.2 ルーチンワークの自動化

私たちの日々の行動には、いわゆるルーチンワークという決まり切った作業も少なくありません。たとえば、「毎月 1 日に、月次売上集計表を上司にメールする」「毎朝、自社に関わる重要なセキュリティのニュースがないかネットで検索して調べる」といったように、難しくはないけれど面倒に感じる作業です。

Python をうまく利用すると、そのような作業を自動化することができます。次のコードは、「Python 公式ブログのトップページを調べ、脆弱性やセキュリティに関する記述がないか調べる」という作業をコマンド 1 つで実現するプログラムです(コード 8-1)。

コード 8-1 Python 公式ブログからセキュリティ関連記述の有無を調べる

```
1 import urllib.request
2
3 url = 'https://blog.python.org/'
4 req = urllib.request.Request(url)
5 with urllib.request.urlopen(req) as res:
6     body = str(res.read())
7
8 if 'security' in body or 'vulnerability' in body:
9     print('セキュリティに関する記述があります')
10    print('https://blog.python.org/を確認してください')
11 else:
12    print('調査対象のセキュリティ用語はありませんでした')
```




こういった自動化プログラムは、Python 以外の言語でももちろん作れるけど、Python は文法が簡単で手軽だし、多くの OS で同じように動くから便利なんだよ。

このような自動化は、クラウド環境でシステムを運用する場面でも近年広く活用されています。たとえば、ショッピングサイトのシステムでは、アクセスが増え始める 20 時頃に「サーバ台数を数倍に増やす」というプログラムを、翌 2 時頃に「サーバ台数を元に戻す」というプログラムを自動的に実行させることで、ピークに沿ったサーバ運用を実現しています。



日々の仕事も人手をかけることなくラクにできるのね。IT 業界に関わるなら勉強しておいて損はないですね。

よーし、さっそく自動化できるルーチンワークがないか、探してみよう。そうすれば、僕のボーナス査定はきっと…。むふっ、むふふ。



8.1.3 データベースの操作

データベースとは、データを整理して格納したり、効率的に取り出したりするためのソフトウェアとデータの集合体をいいます。多数のユーザーが同時にアクセスしても、データの整合性を保ちつつ高速に処理できることが大きな特長です。

一般的なデータベースは複数の表の形式でデータを保存し、その値を取得したり、書き換えたりして利用します。データを読み書きするには、**SQL** と呼ばれるデータベースを操作するための専用の言語でデータベースに指示を出します。



データベースならいずみ先輩に少し聞いたことがあります。Python でもデータベースを操作できるんですね。

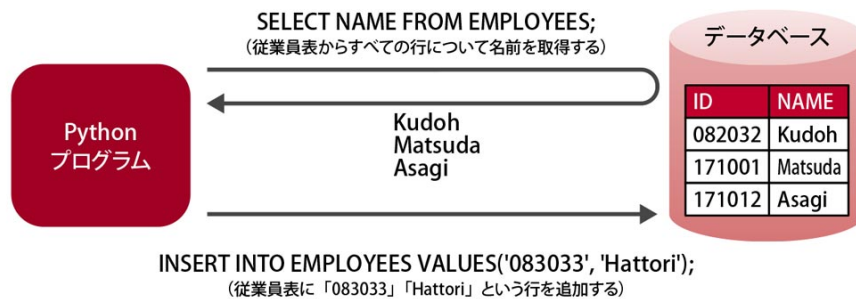


図 8-1 データベースは SQL で操作する



「SELECT 〜」と「INSERT 〜」というのが SQL 文だよ。詳しくは『スッキリわかる SQL 入門』などを参考にしてほしい。

ここでは、Python の標準モジュール sqlite3 に含まれている SQLite というデータベースを使った例を紹介します。次のコードは、従業員の ID と名前が登録された EMPLOYEES という表から、データを取り出して表示します(コード 8-2)。

8
章

コード 8-2 SQLite を使ったデータベース検索

```
1 import sqlite3
2 with sqlite3.connect('sample.db') as conn:
3     cursor = conn.cursor()
4     cursor.execute('SELECT ID, NAME FROM EMPLOYEES')
5     for row in cursor.fetchall():
6         print(row[0])
7         print(row[1])
```



こんな少しのコードでデータベースを操作できるなんて意外だなあ。

データベースは本格的なアプリケーションを作るには欠かせないから、興味を持ったなら調べてみるといいよ。



8.1.4 ウィンドウアプリケーションの作成

物理的に最も人間に近く、人間とコンピュータの接点となる部分を**ユーザーインタフェース** (UI: User Interface) といい、主にコンピュータの操作性や処理された情報の表示方法を指します。

本書でこれまで行ってきたキーボードによる入力のみでコンピュータを操作する方法を **CUI** (Character User Interface) といいます。一方、macOS や Windows などのグラフィカルなウィンドウ表示とマウスやタッチパッドなどによってコンピュータを操作する方法を **GUI** (Graphical User Interface) と呼びます。GUI を備えたプログラムのことをウィンドウアプリケーションとも呼びます。

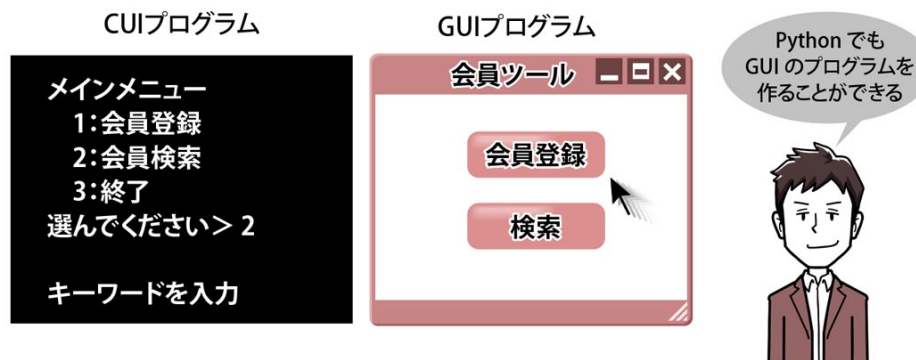


図 8-2 CUI と GUI



へえー、Python でもウィンドウが表示できるんですか。今までの実行結果は文字ばかりだったから、何だか意外だな。

Python には、手軽に GUI を作成できるライブラリが充実しています。ここでは、標準ライブラリの tkinter を使って「Hello, World」というボタンを持つウィンドウを表示する例を紹介します (コード 8-3)。

コード 8-3 tkinter を使ってボタンのあるウィンドウを作成する

```
1 import tkinter as tk
2 root = tk.Tk()
3 root.geometry('240x240')
4 root.title('GUI Sample')
5 button = tk.Button(root, text='Hello, World')
6 button.pack()
7 root.mainloop()
```

実行結果



8 章

Python でウィンドウアプリケーションを作るなら、tkinter だけでなく、Kivy や PyQt、wxPython といった外部ライブラリも有名です。



外部ライブラリを上手に利用すれば、自分でゲームを作ること
もできるんだよ。

自分が遊ぶばかりじゃなくて、ほかの人に自分のプログラム
で遊んでもらうのも楽しそうですね！



ウィンドウアプリケーションの分野では Python は主流とは言えませんが、手

軽にグラフィカルなアプリケーションを作成できますし、見た目にも楽しいので、初学者の勉強の題材にはお勧めです。

8.1.5 Web アプリケーションの作成

インターネット上のショッピングサイトでは、商品カテゴリーや検索ワードを指定して検索することで、店のデータベースからさまざまな商品の情報や在庫情報を取得して閲覧することができます。さらに、商品を選択して必要な情報を入力し、購入ボタンをクリックすれば、購入記録がデータベースに登録され、実際に商品を購入することができます(図 8-3)。

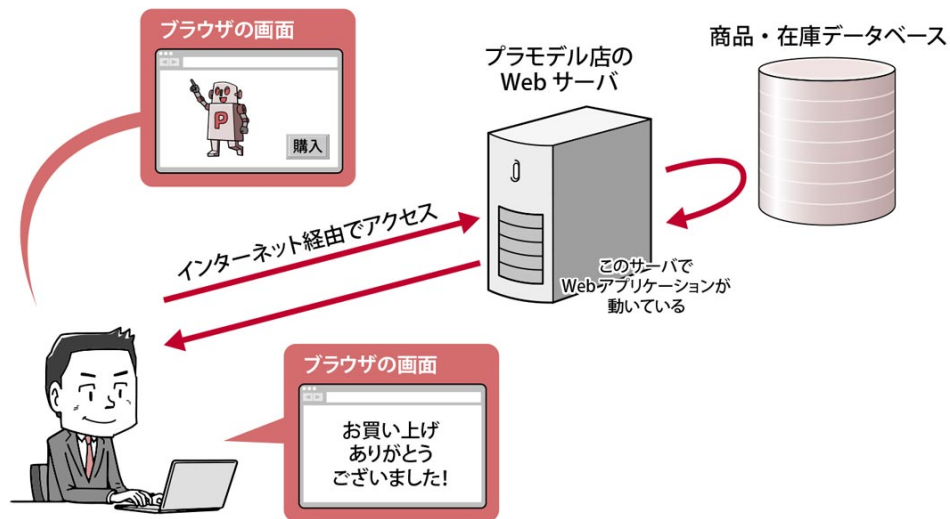


図 8-3 ショッピングサイトの Web アプリケーション

このような、利用者がブラウザから入力した情報をサーバ側のプログラムで処理するしくみを備えた Web サイトを **Web アプリケーション** といいます。ショッピングサイトだけでなく、検索サイトや、旅行や映画などの予約サイト、あるいは SNS サイトなど、みなさんも数多くの Web アプリケーションを利用していることでしょう。

Web アプリケーションにはさまざまな機能が必要となるため、一般的にはゼロから開発するのではなく、**Web フレームワーク** と呼ばれるライブラリを使用します。Web フレームワークは効率的な開発を実現するための専用の関数や、

Web アプリケーションの骨組みまで用意されているため、少ないコードで開発することが可能となります。Python では、外部ライブラリの Django や Flask といった Web フレームワークが有名です。

次のコードは、アクセスされたら現在時刻を表示する時報のような動作を行う Web アプリケーションを、Flask を使用して作成した例です（コード 8-4）。

コード 8-4 Flask を使って現在時刻を表示する

```
1 from flask import Flask
2 import datetime
3 app = Flask(__name__)
4 @app.route('/')
5 def hello():
6     dt = datetime.datetime.now()
7     html = '<!DOCTYPE html>'
8     html += '<html>'
9     html += '<head><title>Flask Sample</title></head>'
10    html += '<body>'
11    html += '{}年{}月{}日 {}時{}分{}秒です'.format(
12        dt.year, dt.month, dt.day, dt.hour, dt.minute,
13        dt.second)
14    html += '</body></html>'
15    return html
16 if __name__ == '__main__':
17     app.run()
```

8章

本来、Web アプリケーションを動作させるには、Web サーバを準備して、プログラムをサーバ上の適切な場所に配置するという作業が必要になります。Flask には標準で簡易的な Web サーバが付属しているので、別途 Web サーバを

準備しなくても動作確認を行うことができます。

コード 8-4 を実行すると、自分のコンピュータが Web サーバとして動作するので、ブラウザで「http://127.0.0.1:5000」という URL パターンにアクセスすれば動作を確認できます (図 8-4)。

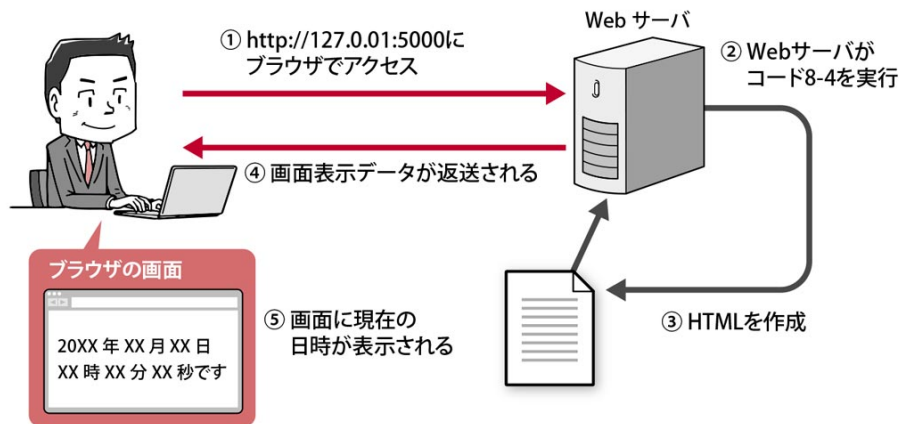


図 8-4 コード 8-4 の実行の流れ

8.1.6 IoT アプリケーションの作成

従来、インターネットに接続してデータのやりとりをしていたのは、主にパソコンやサーバなどのコンピュータでした。しかし近年では、あらゆる「モノ」がインターネットを通じてつながり、互いの情報や機能を利用し合っています。このようなしくみを **IoT** (Internet of Things) といいます。テレビ、家電、車、衣服、ドア、鍵、マンホール、犬…。あらゆるモノが IoT でつながることが可能です。



い、犬!? 犬もネットにつながるんですか!?

正確には、犬につけた首輪だね。



これらのモノには、**マイコン**(マイクロコンピュータ、micro computer)という小型のコンピュータが組み込まれています。これまでマイコンを制御するには、アセンブラやC言語といった専門的なプログラミング言語で書かれた難解なプログラムが必要でした。

しかし近年、Pythonなどの一般的な言語から手軽なマイコンを制御できるキットが販売されています。特に有名なのが ^{ラズベリーパイ}Raspberry Pi ですが、^{アルデュイーノ}Arduino、SDカードサイズの ^{エジソン}Edison など、さまざまなマイコンキットが販売され、人気を博しています(図8-5)。

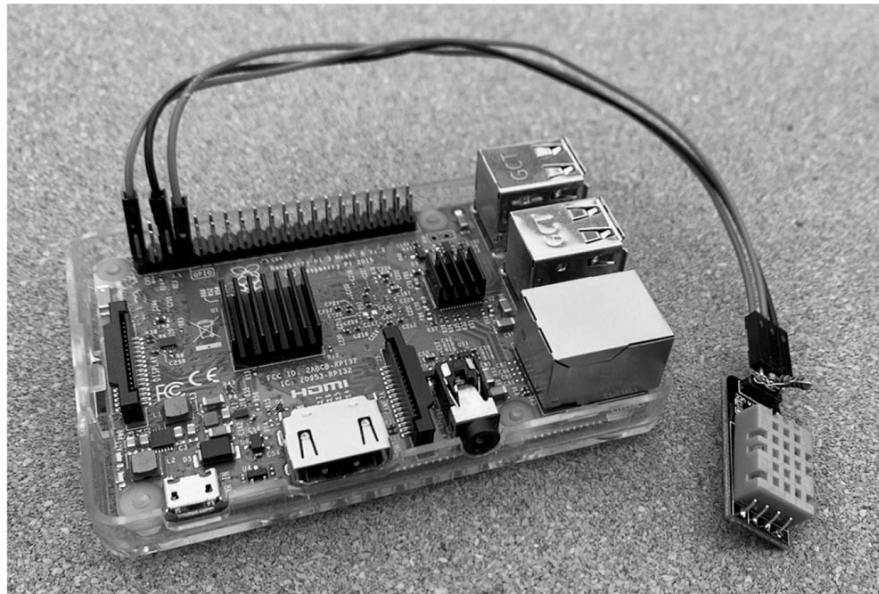


図8-5 温湿度センサー(DHT11)を接続した Raspberry Pi

8章



ちなみに Raspberry Pi の Pi は、Python に由来しているんだ。

この技術を学べば、僕が作ったプログラムで、家電とか鍵とかいろんな装置を制御できちゃうんですね！





何だか危険な思想のような気がするのは私だけ？

次のコード 8-5 は、Raspberry Pi に接続された温湿度センサーから、温度と湿度を 1 秒おきに取得して表示します。このコードを動かすには、外部モジュール RPi.GPIO と DHT11 クラスが必要です。詳細は、専門書を参照してください。

コード 8-5 Raspberry Pi から温湿度を取得して表示する

```
1 import RPi.GPIO as GPIO
2 import dht11
3 import time
4 import datetime
5 GPIO.setwarnings(False)
6 GPIO.setmode(GPIO.BCM)
7 GPIO.cleanup()
8 instance = dht11.DHT11(pin=14)
9 while True:
10     result = instance.read()
11     if result.is_valid():
12         temperature = result.temperature
13         humidity = result.humidity
14         print('温度:{}'.format(temperature))
15         print('湿度:{}'.format(humidity))
16     time.sleep(1)
```



Column

MicroPython

マイコンの中には、小さいゆえにメモリ空間が厳しく制限されているものも存在します。そのような環境でも動作する Python プログラムの開発を目的とした、MicroPython という種類の Python もあります。

8.1.7 データ分析・機械学習



最後に、最も Python らしい分野を紹介して締めくくりにしよう。

8章

近年、**データサイエンス**という言葉が存在感を増しています。データサイエンスとは、データを分析して、新たな価値を創出する活動のことです。従来は単に統計などを行うデータ分析が一般的でしたが、ここ数年、膨大なデータをコンピュータに与え、ルールや規則性を探して人間さながらの予測や分類をさせる**機械学習** (machine learning) の実用化が急速に進んでいます。

たとえばコンビニエンスストアでは、買った商品だけでなく、客の性別や年代などの情報も会計の際に入力されています。全国から集まるそれらの購買情報を集めて人が分析したり、コンピュータに規則性を探させたりすることで、消費者の動向や傾向を見つけ、需要の予測や新商品の開発などに役立てています。



これらの技術は、人工知能 (AI) を支える重要な技術なんだ。

Python は、このようなデータ分析や機械学習に関するライブラリが特に充実していることで知られています。第7章で紹介した matplotlib や Pandas、NumPy、scikit-learn、Tensorflow といったライブラリを組み合わせることで、多種多様なデータ分析や機械学習を短期間で実現することができます。そのため、

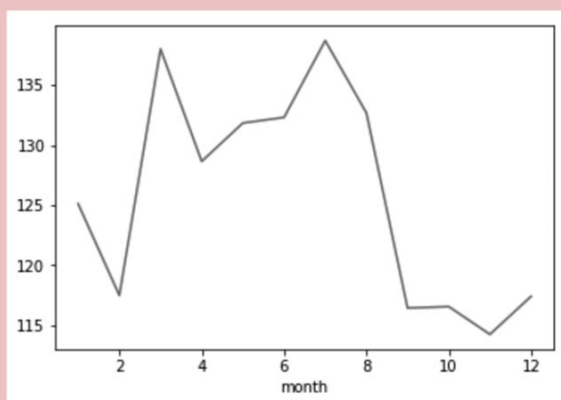
Python はデータサイエンスの分野においてデファクトスタンダードと言えるほど、世界中のデータサイエンティストに使用されています。

次のコードは、日本全国の家庭におけるカレールウに対する支出額の平均を、月別に統計してグラフ化するサンプルです(コード 8-6)。このプログラムが読み込む `curry.csv` というデータファイルには、総務省統計局が公開している「家計調査 家計収支編 (二人以上の世帯)」から抽出したカレールウに対する 10 年分のデータがカンマ区切りで入っています。

コード 8-6 カレールウの月別の支出額をグラフ化する

```
1 import pandas as pd
2 df = pd.read_csv('curry.csv', encoding='Shift_JIS')
3 df['month'] = pd.to_datetime(df['時間軸 (月次)'],
                              format='%Y年%m月').dt.month
4 df = df.groupby('month').mean()
5 df.mean(axis=1)
6 %matplotlib inline
7 df.mean(axis=1).plot() # Pandasがmatplotlibを使って可視化する
```

実行結果





やっぱり夏はカレー！ これで夏の1箇月間カレー祭りを堂々と開催できますよ！ うん、データ分析って楽しいかも！

1年中食べてるくせに…。それにしても、3月もカレーの消費が増えるのね。3月にカレーイベントをしたら儲かるかも♪
マーケティング部の血が騒ぐわ〜。



Column

R 言語

データ分析の分野で、Python と並んでよく使用されるのが R 言語です。Python のように機械学習と組み合わせたり、分析結果をシステムに組み込んだりするのには得意ではありませんが、データ分析に特化しているため、分析の用途では Python よりも手軽に実現できることも多いと言われています。

8 章

8.2

Python の基礎を 学び終えて

8.2.1 終わりに

8 章にわたって Python の基礎を学んできましたが、ここでもう一度、Python の特性を振り返ってみましょう。



Python の特性

- ・ シンプルな文法で、手軽にプログラム開発ができる。
- ・ 機械学習をはじめ、さまざまな分野のライブラリが充実している。

このような特性を持つ Python は 1991 年に公開されて以来、現在に至ってもその人気は衰えることなく、むしろ活躍の場を広げています。そして、データサイエンスや AI といった分野の発展とともに、Python エンジニアの需要は今後も増えていくと予想されています。

本書では、はじめてプログラミングを学ぶという人にも手軽に読み進めてもらえるよう、Python の基礎の部分に集中して解説してきました。しかし、Python はこれまで紹介したもの以外にもたくさんの機能を備えています。本書の内容を復習しつつ、インターネットや多数出版されている技術書などを活用して知識を広げていってください。



そうか、ここまではまだほんの入り口にすぎないのね。

その中では、**オブジェクト指向プログラミング**、**関数型プログラミング**という言葉と出会うことがあるかもしれません。これらはプログラミングのパラダイム

(考え方や手法)を表す言葉で、前者は大規模なプログラムを多人数で手分けして開発するのに役立ちます。後者はバグが混入しにくく、テストがしやすいプログラムを開発するのに役立ちます。それぞれ専門書が書かれるくらいの内容であるため、本書ではあえて紹介せず、基本的な**手続き型プログラミング**をベースに解説しました。

しかし、**オブジェクト指向プログラミングや関数型プログラミングについて詳しく知らなくても、機械学習をはじめとするさまざまな専門分野の学習に進むことはできます**。なぜなら、どのような分野であっても Python にはライブラリが豊富に用意されており、それらのライブラリを駆使していくことが主体となるからです。そしてそのライブラリの基本的な使い方は、本書の知識をしっかりと身に付けておけば、いくらでも応用することが可能です。



オブジェクト指向や関数型プログラミングは、大規模開発に参加したり、ライブラリを作る立場になったりするなど、必要になったときに勉強しても遅くはないんだ。

8章

やりたいことが見つかったら、積極的にチャレンジしてください。やりたいことが見つからなくても、積極的にいろいろな分野に首を突っ込んでみてください。みなさんの行動の先に、きっと道は拓けるでしょう。ぜひ、松田くんや浅木さんとともに、新しい世界を探しに行きましょう。



どの道を選んでも「間違い」はない。選んだ道が正解になると信じて、次の一步を踏み出そう！

はい！



さらなる高みを目指して—— 松田さんと浅木さんの成長の旅は続きます



Python のはじめの一步

Python に関わるすべての人が
知っておきたい内容をサクッ
と速習できる!



AI (人工知能) や機械学習に 興味がある人は

「AI」「機械学習」「データ処理」
などに関する書籍やサイトで
学習しよう

検索キーワード

Python

+

機械学習

Pandas

Numpy

クラスタリング



Python をもっと 使いこなしたい人は

Python のより高度な内容を含
む書籍やサイトで学習しよう

検索キーワード

Python

+

クラス

オブジェクト指向

ラムダ

関数型プログラミング

Flask

Django

付録 A

sukkiri.jp について

本書で Python を学ぶみなさんのために、
学習を手助けするツールをインターネット上に準備しました。
ここでは、そのツールにアクセスするための方法を紹介します。

CONTENTS

A.1 sukkiri.jp について

A.1 sukkiri.jp について

本書では、学習を手助けする各種手順書やリファレンスをインターネット上に準備しました。Anaconda のインストール手順や JupyterLab の基本的な使い方についても、「https://sukkiri.jp/books/sukkiri_python/sukkiri_python_appendix.html」で公開していますので、ぜひ確認してみてください。



Python をとりまく世界は、すごいスピードで変わり続けているからね。

最新の情報を確認できるから、安心ね。



Column

sukkiri.jp

<https://sukkiri.jp/> は、「スッキリわかる」シリーズの著者や制作陣が中心となって、各種情報を 1 箇所に集めたサイトです。書籍に掲載したコード(一部)がダウンロードできるほか、ツール類の導入手順や学び方など、学び手のみなさんのお役に立てる情報をお届けしています。



「スッキリわかる」シリーズ。今後も続刊予定です。

※表紙画像は変更になる場合があります。

付録 B

エラー解決・ 虎の巻

プログラミングをしていると、思いどおりに動かないことやエラーに悩まされることが少なくありません。

幸い、「エラーを解決する方法」にはコツがあります。

この付録では、エラー解決に関するコツとエラーメッセージの読み方を紹介したうえで、困ったときの状況に応じた対応方法を紹介します。

CONTENTS

- B.1 エラーとの上手な付き合い方
- B.2 エラー虎の巻
- B.3 例外処理

B.1

エラーとの上手な付き合い方

B.1.1 エラーを解決できるようになる 3 つのコツ

Python プログラミングを始めて間もないうちは、作成したプログラムが思うように動かないことも多いでしょう。些細なエラーの解決に長い時間を要することもあるかもしれませんが、誰もが通る道ですから自信をなくす必要はありません。

しかし、その「誰もが通る道」を可能な限り効率よく進んで、エラーをすばやく解決できるようになれば理想的です。幸い、エラーをすばやく解決するにはコツがあります。

コツ① 原因を理解したうえで修正する

「なぜエラーが発生したか」という原因を理解しないまま、コードを修正してはいけません。同じエラーに何度も悩まされるよりも、理解に時間がかかったとしても、二度と同じエラーを起こさないほうが合理的と言えるでしょう。特に、原因を理解していなくても表面的にエラーを消せてしまう、開発ツールや統合開発環境の「エラー修正支援機能」には注意が必要です。初心者のうちにはできるだけ、この機能を使わないようにしましょう。

コツ② エラーメッセージから逃げずに読む

エラーが出ると、エラーメッセージをきちんと読まずに思いつきでソースコードを書き換え始める人がいます。しかし、「何が悪いのか、どこが悪いのかという情報」は、**エラーメッセージに書いてあります**。その貴重な手がかりを読まないのは「目隠しをして探し物をする」も同然です。上級者でも難しい「ノーヒント状態でのエラー解決」を、初心者ができるはずがありません。

メッセージが英語、あるいは不親切な日本語であったとしても、エラーメッセージはきちんと読みましょう。特に英語の意味を調べる手間を惜しまないでください。それを調べる数分の時間で、悩む時間が何時間も減ることもあります。

コツ③ エラーと試行錯誤をチャンスと考える

熟練した開発者がすばやくエラーを解決できるのは、Python の文法に精通しているからだけではありません。頭の中に「エラーを起こした失敗経験と、それを解決した成功経験」の記憶の引き出しをたくさん持っているから、つまり、**似たようなエラーで悩んだ経験があるから**なのです。

このことが示すように、エラー解決の上達のためには、「たくさんのエラーに出会い、試行錯誤し、引き出しを 1 つひとつ増やすこと」が不可欠です。つまり、誰もが避けたいと思う**新しいエラーに直面して試行錯誤している時間こそ、自分が一番成長しているとき**なのです。深く悩むときや切羽詰まるときもあるでしょうが、「自分は今、成長している」と考えて、前向きに試行錯誤してください。

これら 3 つのコツの中で、基本であり最も重要なのが、1 つ目の「エラーメッセージをきちんと読むこと」です。しかし、「そもそもエラーメッセージの読み方がわからない」という人もいるでしょう。そこで、次項ではエラーメッセージの読み方を紹介します。

B.1.2 エラーメッセージの読み方

Python のエラーは、プログラム実行前に発生する「構文エラー」と、プログラム実行中に発生する「実行時エラー (例外)」に分けることができます。それぞれ発生すると、エラーに関する情報が表示されます。

構文エラー発生時の例

```
File '<ipython-input-1-126035ba983d>', line 1
    print('hello)
           ^
SyntaxError: EOL while scanning string literal
```

実行時エラー発生時の例

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-1-d9183e048de3> in <module>
----> 1 print(x)
NameError: name 'x' is not defined

```

エラーが発生した場所 (スタックトレース)

エラーメッセージ

エラーの名前。実行時エラーは「SyntaxError」以外になる

構文エラーと実行時エラーでは、表示される内容が異なりますが、主に「エラーが発生した場所」「発生したエラーの名前」「エラーメッセージ」が表示されます。これらの情報をヒントにエラーの原因を推測してソースコードを修正します。

Python のプログラミング経験が少ないうちは、エラー原因を推測するのが難しいので、次節の「エラー虎の巻」を参考にしてください。よく発生するエラーメッセージとその原因を紹介しています。

B.1.3 スタックトレース

実行時エラーが発生すると多くの場合、**スタックトレース** (stacktrace) が表示されます。スタックトレースには、エラーが発生するまでの過程 (関数の呼び出し順序など) が示されています。スタックトレースを読み解くことで、想定外の処理に気づくこともできます。

プログラムの規模が大きくなるにつれ、実行時エラーの要因は多くなるため、エラー解決が難しくなります。その際、とても役に立つのがスタックトレースです。ぜひ、スタックトレースの読み方をマスターしておきましょう。

次のスタックトレースは、コード B-3 (P344) を実行したときに表示されるものです。

スタックトレースの例

```

NameError                                Traceback (most recent call last)
<ipython-input-10-f526d3ecb4bd> in <module>
      1 x = 10
      2 y = 20
----> 3 funcB(10, 20)

<ipython-input-9-7d1a6dbb4892> in funcB(x, y)
      1 def funcB(x, y):
      2     z = x + y
----> 3     funcA(z)

<ipython-input-8-8f50e918c27c> in funcA(z)
      1 def funcA(z):
----> 2     ans = z * a
      3     print(ans)

NameError: name 'a' is not defined

```

funcB を呼び出し (コード B-3)

funcA を呼び出し (コード B-2)

エラー発生!! (コード B-1)

コード B-1 funcA 関数の定義

```

1 def funcA(z):
2     ans = z * a
3     print(ans)

```

コード B-2 funcB 関数の定義

```
1 def funcB(x, y):  
2     z = x + y  
3     funcA(z)
```

コード B-3 funcB 関数の呼び出し

```
1 x = 10  
2 y = 20  
3 funcB(10, 20)
```

行番号の左側に表示される点線の矢印の箇所に注目することで、エラーが発生するまでに、どの関数をどの順番で呼び出しているかを確認することができます。この例の場合、funcB 関数の呼び出し(コード B-3)→func 関数 A の呼び出し(コード B-2)→funcA(コード B-1)の 2 行目で実行時エラーが発生したことを示しています。

エラー解決の手順としては、まず、エラーが発生した関数のソースコードを確認し、エラーがなぜ発生したかが判明すれば、それを修正します。しかし、その関数を見ても問題が見つからないときは呼び出し元の関数を確認します。もし呼び出し元の関数に問題がなければ、「呼び出し方が悪かったのではないかと仮定して、その呼び出し元へとさかのぼっていきます。

今回の例の場合、funcA 関数をまず確認し、問題がなければ funcB 関数にさかのぼります。もし、funcB 関数に問題がなければ、その呼び出し元にさらにさかのぼっていきます。

また、エラーが発生した関数が、ライブラリで提供されている関数である場合があります。ライブラリで提供されている関数にバグがあるとは考えにくいので、そのような場合は呼び出し元の関数から確認します。

B.2

エラー虎の巻

付録
B

B.2.1 構文エラーが発生した

(1) `SyntaxError: EOL while scanning string literal`**原因** 文字列を引用符(' または ")で囲んでいない。**例** 文字列 hello の引用符を閉じていない。

```
1 print('hello)
```

実行結果

```
File '<ipython-input-1-126035ba983d>', line 1
  print('hello)
        ^
```

```
SyntaxError: EOL while scanning string literal
```

(2) `SyntaxError: unexpected EOF while parsing`**原因** 構文が途中で終了している。カッコが正しく閉じられていないことが多い。**例** print 関数の引数の丸カッコを閉じていない。

```
1 print('hello'
```

実行結果

```
File '<ipython-input-7-0b37b907169d>', line 1
  print('hello'
        ^
```

```
SyntaxError: unexpected EOF while parsing
```


例 リストの角カッコを閉じていない。

```
1 my_list = [10, 20, 30
```

実行結果

File '<ipython-input-8-361cfdc223f8>', line 1

```
my_list = [10, 20, 30
```

^

SyntaxError: unexpected EOF while parsing

例 else ブロックの処理を書いていない。

```
1 x = 10
2 if (x < 10) :
3     print('hoge')
4 else:
```

実行結果

File '<ipython-input-31-b6845f3568fa>', line 4

```
else:
```

^

SyntaxError: unexpected EOF while parsing

(3) IndentationError: expected an indented block

原因 インデントが必要な箇所でインデントされていない。

例 else ブロックをインデントしていない。

```
1 x = 10
2 if (x < 10) :
3     print('hoge')
```

```
4 else:
5 print('foo')
```

実行結果

```
File '<ipython-input-32-b6845f3568fa>', line 5
    print('foo')
      ^
IndentationError: expected an indented block
```

(4) SyntaxError: invalid character in identifier

原因 全角スペース(空白)や、全角の記号(引用符やカッコ)を使用している。

例 全角のスペースを使用している。

```
1 x = 10
```

実行結果

```
File '<ipython-input-4-f5fe71432f38>', line 1
    x = 10
      ^
SyntaxError: invalid character in identifier
```

例 全角の丸カッコを使用している。

```
1 print('hello')
```

実行結果

```
File '<ipython-input-5-836f013168c2>', line 1
    print('hello')
      ^
SyntaxError: invalid character in identifier
```

B.2.2 実行時エラーが発生した

(1) `NameError: name 'X' is not defined`

原因 定義していない変数や関数を使用した。

備考 X には変数名や関数名が入る。

例 定義していない変数 x を使用した。

```
1 print(x)
```

実行結果

```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-1-d9183e048de3> in <module>  
----> 1 print(x)  
  
NameError: name 'x' is not defined
```

例 定義していない hello 関数を呼び出した。

```
1 hello()
```

実行結果

```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-2-a75d7781aaeb> in <module>  
----> 1 hello()  
  
NameError: name 'hello' is not defined
```

(2) TypeError: can only concatenate str (not 'int') to str

原因 文字列と整数を「+」で連結した。

例 文字列 hello と整数 10 を連結した。

```
1 x = 10
2 print('hello' + x)
```

実行結果

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-23-13c2189b327d> in <module>
      1 x = 10
----> 2 print('hello' + x)

TypeError: can only concatenate str (not 'int') to str
```

(3) TypeError: unsupported operand type(s) for X

原因 データ型にサポートされていない演算を行った。

備考 X は、演算子によって異なる。

例 文字列 hello を整数 5 で割った。

```
1 print('hello' / 5)
```

実行結果

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-7-4d66ee34113a> in <module>
----> 1 print('hello' / 5)

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

例 文字列のべき乗を求めた。

```
1 print('hello' ** 5)
```

実行結果

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-8-89a89c9a9b29> in <module>
----> 1 print('hello' ** 5)

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

(4) TypeError: X takes Y positional arguments but Z was/were given

原因 関数の仮引数より実引数のほうが多い。

備考 X には関数名、Y には仮引数の数、Z には実引数の数が入る。

例 仮引数がない hello 関数に、実引数を 1 つ指定した。

```
1 def hello():
2     print('Hello')
3 hello('World')
```

実行結果

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-14-ca39a35666b5> in <module>

      1 def hello():
      2     print('Hello')
----> 3 hello('World')

TypeError: hello() takes 0 positional arguments but 1 was given
```

例 仮引数が 2 つの hello 関数に、実引数を 3 つ指定した。

```
1 def hello(x, y):
2     print(x + ', ' + y)
3     hello('Hello', 'Python', 'World')
```

実行結果

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-15-38785fc957fb> in <module>
      1 def hello(x, y):
      2     print(x + ', ' + y)
----> 3     hello('Hello', 'Python', 'World')

TypeError: hello() takes 2 positional arguments but 3 were given
```

(5) TypeError: X missing Y required positional argument(s): 'Z'

原因 呼び出した関数の仮引数に渡す実引数が不足している。

備考 X には関数名、Y には不足している実引数の数、Z には引数名が入る。

例 仮引数が 1 つの hello 関数に実引数を渡さなかった。

```
1 def hello(x):
2     print('Hello,' + x)
3     hello()
```

実行結果

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-17-050bc2c77bac> in <module>
```



```
1 def hello(x):
2     print('Hello,' + x)
----> 3 hello()

TypeError: hello() missing 1 required positional argument: 'x'
```

例 仮引数が 2 つの hello 関数に実引数を渡さなかった。

```
1 def hello(x, y):
2     print(x + ', ' + y)
3     hello()
```

実行結果

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-18-6e38068b3c68> in <module>
      1 def hello(x, y):
      2     print(x + ', ' + y)
----> 3     hello()

TypeError: hello() missing 2 required positional arguments: 'x' and 'y'
```

(6) ZeroDivisionError: division by zero

原因 0 で除算しようとした。

例 10 を 0 で割った。

```
1 x = 10
2 y = 0
3 ans = x / y
```

実行結果

```
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-2-fafe248ef32e> in <module>
      1 x = 10
      2 y = 0
----> 3 ans = x / y

ZeroDivisionError: division by zero
```

(7) IndentationError: unindent does not match any outer indentation level

原因 インデントが正しく行われていない。

例 if-else 構文の「else:」の行に不要なインデントをした。

```
1  if x >= 100:
2      print('hoge')
3      else:
4          print('foo')
```

実行結果

```
File '<tokenize>', line 3

    else:
    ^
IndentationError: unindent does not match any outer indentation level
```

(8) TabError: inconsistent use of tabs and spaces in indentation

原因 1つのブロック内でスペース(空白)とタブ文字を混せてインデントを行っている。

例 インデントにスペース (2 行目) とタブ文字 (3 行目) が混在している。

```
1  if x >= 100:
2      print('hoge')
3      print('foo')
```

実行結果

```
File "<ipython-input-3-a8cbdfb62586>", line 3
```

```
    print('foo')
```

```
        ^
```

```
TabError: inconsistent use of tabs and spaces in indentation
```

(9) IndexError: list index out of range

原因 指定した添え字を持つ要素がリストに存在しない。

例 要素数 3 のリストの 4 番目 ([3]) の要素を指定した。

```
1  my_list = [10, 20, 30]
2  my_list[3]
```

実行結果

```
-----
IndexError                                Traceback (most recent call last)
```

```
<ipython-input-8-d248c2f057bf> in <module>
```

```
    1 my_list = [10, 20, 30]
```

```
----> 2 print(my_list[3])
```

```
IndexError: list index out of range
```

(10) KeyError: 'X'

原因 指定したキーを持つ要素がディクショナリに存在しない。

備考 X には誤って指定したキーが入る。

例 辞書に存在しないキー hoo を指定した (foo の誤り)。

```
1 my_dict = {'hoge':1, 'foo':2 }
2 print(my_dict['hoo'])
```

実行結果

```
-----
KeyError                                Traceback (most recent call last)
<ipython-input-9-a799ecaf9b88> in <module>
      1 my_dict = {'hoge':1, 'foo':2 }
----> 2 print(my_dict['hoo'])

KeyError: 'hoo'
```

(11) ValueError: X

原因 引数に渡した値が、関数が期待しているものと異なる。

備考 用意されている関数を呼び出す際に発生する。X に表示される内容は呼び出した関数によって異なる。

例 引数を整数に変換する int 関数に、整数に変換できない値を渡した。

```
1 x = 'hello'
2 x = int(x)
```

実行結果

```
-----
ValueError                                Traceback (most recent call last)
```

```
<ipython-input-13-cee3ccb3b085> in <module>
      1 x = 'hello'
----> 2 x = int(x)

ValueError: invalid literal for int() with base 10: 'hello'
```

(12) UnboundLocalError: local variable 'X' referenced before assignment

原因 ローカル変数を初期化する前に、値を参照した。

備考 X にはローカル変数の名前が入る。

例 変数 a の値を参照したあとに初期化を行った。

```
1 def hoge():
2     print(a)
3     a = 10
4     hoge()
```

実行結果

```
-----
UnboundLocalError                                Traceback (most recent call last)
<ipython-input-12-33eaa7dc7552> in <module>
      2     print(a)
      3     a = 10
----> 4     hoge()

<ipython-input-12-33eaa7dc7552> in hoge()
      1 def hoge():
----> 2     print(a)
      3     a = 10
      4     hoge()
```

```
UnboundLocalError: local variable 'a' referenced before assignment
```

(13) ModuleNotFoundError: No module named 'X'

原因 指定したモジュールが見つからない。

備考 X には、誤って指定したモジュールの名前が入る。

例 math を指定するつもりで、masu を指定した。

```
1 import masu
```

実行結果

```
-----  
ModuleNotFoundError                                Traceback (most recent call last)  
<ipython-input-20-0d89f4e602f8> in <module>  
----> 1 import masu  
  
ModuleNotFoundError: No module named 'masu'
```

(14) AttributeError: module 'X' has no attribute 'Y'

原因 取り込んだモジュールに存在しない変数や関数を使用した。

備考 X にはモジュール名、Y には変数名または関数名が入る。

例 math モジュールの pow 関数を呼び出すつもりで、誤って power と記述した。

```
1 import math  
2 math.power(10, 2)
```

実行結果


```
AttributeError                                Traceback (most recent call last)
<ipython-input-22-b6373654ee68> in <module>
      1 import math
----> 2 math.power(10, 2)

AttributeError: module 'math' has no attribute 'power'
```

(15) ImportError: cannot import name 'X' from 'Y'

原因 モジュールから指定した変数や関数を取り込めない。

備考 X にはモジュール名、Y には変数名または関数名が入る。

例 math モジュールの pow 関数を取り込むつもりで、誤って power と記述した。

```
1 from math import power
```

実行結果

```
-----
ImportError                                Traceback (most recent call last)
<ipython-input-23-c78644c22643> in <module>
----> 1 from math import power

ImportError: cannot import name 'power' from 'math' (unknown location)
```

(16) KeyboardInterrupt:

原因 プログラムを強制的に終了した。

例 無限ループが発生したので、停止ボタンで終了させた。

```
1 x = 0
2 while x < 10:
    print('Hello')
```

実行結果

```

-----
KeyboardInterrupt                                Traceback (most recent call last)
<ipython-input-24-f459a4229c7e> in <module>

      1 x = 0
      2 while x < 10:
----> 3     print('Hello')

~/anaconda3/lib/python3.7/site-packages/ipykernel/iostream.py in
write(self, string)
    398         is_child = (not self._is_master_process())
    399         # only touch the buffer in the IO thread to avoid races
--> 400         self.pub_thread.schedule(lambda : self._buffer.
write(string))

    401         if is_child:
    402             # newlines imply flush in subprocesses

~/anaconda3/lib/python3.7/site-packages/ipykernel/iostream.py in
schedule(self, f)
    201         self._events.append(f)
    202         # wake event thread (message content is ignored)
--> 203         self._event_pipe.send(b'')
    204     else:
    205         f()

~/anaconda3/lib/python3.7/site-packages/zmq/sugar/socket.py in send(self,
data, flags, copy, track, routing_id, group)
    390                                     copy_threshold=self.copy_
threshold)

```

```

391         data.group = group
--> 392         return super(Socket, self).send(data, flags=flags,
copy=copy, track=track)
393
394     def send_multipart(self, msg_parts, flags=0, copy=True,
track=False, **kwargs):

zmq/backend/cython/socket.pyx in zmq.backend.cython.socket.Socket.send()

zmq/backend/cython/socket.pyx in zmq.backend.cython.socket.Socket.send()

zmq/backend/cython/socket.pyx in zmq.backend.cython.socket._send_copy()

~/anaconda3/lib/python3.7/site-packages/zmq/backend/cython/checkrc.pxd in
zmq.backend.cython.checkrc._check_rc()

KeyboardInterrupt:

```

B.3**例外処理**付録
B**B.3.1 例外処理とは**

コード B-4 は、割り勘を計算するプログラムです。このプログラムを実行して、料金や人数に「hello」などの整数に変換できない文字列を入力すると、`ValueError` が発生してプログラムが途中で終了してしまいます。

コード B-4 ValueError が発生するプログラム

```
1 price = int(input('料金を入力 >>'))
2 number = int(input('人数を入力 >>'))
3 print('1人あたり{}円です'.format(price / number))
4 print('プログラムを終了します')
```

実行結果

料金を入力 >>1000

人数を入力 >>a

ValueError Traceback (most recent call last)

<ipython-input-6-998763f7b1ea> in <module>

1 price = int(input('料金を入力 >>'))

----> 2 number = int(input('人数を入力 >>'))

3 print('1人あたり{}円です'.format(price / number))

4 print('プログラムを終了します')

ValueError: invalid literal for int() with base 10: 'a'

開発中の環境ならプログラムが途中で終了してしまっても問題になることはほとんどありません。しかし、本番運用の環境でプログラムが途中で終了した場合、ユーザーにとって意味のわからないスタックトレースが表示されたり、プログラムを実行しているコンピュータの動作が不安定になったりすることもあるので、あまり好ましくありません。

例外処理を記述しておく、実行時エラーが発生しても処理を途中で終了させずに、継続することができます。例外処理は **try-except** 文で指示します。コード B-5 は、`ValueError` が発生しても処理が継続するように、コード B-4 に例外処理を加えたものです。

コード B-5 例外処理の例

```
1 try:
2     price = int(input('料金を入力 >>'))
3     number = int(input('人数を入力 >>'))
4     print('1人あたり{ }円です'.format(price / number))
5 except ValueError:
6     print('料金または人数は整数を入力してください')
7 print('プログラムを終了します')
```

try ブロック

except ブロック

`try` と `except` という 2 つのブロックがあることに注目してください。`try` ブロックの中で実行時エラーが発生した場合、`try` ブロック内の処理を中断し、`except` ブロック内の処理を実行します。もし、整数に変換できない文字列を人数に入力した場合、次の実行結果になります。

```
料金を入力 >>1000
人数を入力 >>a
料金または人数は整数を入力してください
プログラムを終了します
```

この例の場合、3行目で `ValueError` が発生します。それによって、`try` ブロックの処理は中断され、`except` ブロックが実行されるので、ユーザー向けのエラーメッセージが表示されます。さらに、最終行の `print` 関数まで実行されて、プログラムが正常に終了していることに注目してください。



例外処理でエラーによる強制終了を防ぐ

`try-except` 文によって、実行時エラーが発生しても処理を継続できる。

もし、`try` ブロックで `ValueError` が発生しなかった場合は、`except` ブロックは実行されません。

```
料金を入力 >>1000
人数を入力 >>5
1人あたり200.0円です
プログラムを終了します
```

`except` ブロックでどのような処理を行うかは自由ですが、一般的には、ユーザーにとって意味のわかるエラーメッセージを出力したり、再入力を促したりするなどの回復処理を行うのが一般的です。

B.3.2 エラーの内容に応じて対応する

`try-except` 文を用いても、必ず実行時エラーの発生時に例外処理が行われるわけではないので注意してください。たとえば、コード B-5 の場合、人数に 0 を入力すると 4 行目で `ZeroDivisionError` (P352) が発生します。その場合、`except` ブロックは実行されずプログラムは終了してしまいます。

その理由は、`except` の右に書かれている実行時エラーの名前です。この名前は、対応する実行時エラーを示しています。そのため、コード B-5 では、`try` ブロッ

ク内で `ValueError` が発生した場合にのみ `except` ブロックが実行され、`ZeroDivisionError` などほかのエラーが発生した場合は `except` ブロックが実行されずプログラムが終了します。

もし、`ValueError` と `ZeroDivisionError` の両方に対応したい場合は、コード B-6 のように書きます。

コード B-6 `ValueError` と `ZeroDivisionError` の両方に対応する

```
1 try:
2     price = int(input('料金を入力 >>'))
3     number = int(input('人数を入力 >>'))
4     print('1人あたり{ }円です'.format(price / number))
5 except ValueError:
6     print('料金または人数は整数を入力してください')
7 except ZeroDivisionError:
8     print('人数に0は入力しないでください')
9 print('プログラムを終了します')
```

ValueError 発生時の処理

ZeroDivisionError 発生時の処理

`except` の右には対応する実行時エラーの名前を指定しますが、次のコード B-7 のように省略すると、すべての実行時エラーに対応することができます。ただし、その場合、原因が異なるエラーに対して、同じ対応をすることになってしまうので注意してください。

コード B-7 すべての実行時エラーに対応する

```
1 try:
2     price = int(input('料金を入力 >>'))
3     number = int(input('人数を入力 >>'))
```

```
4     print('1人あたり{}'.format(price / number))
5     except:
6         print('料金と人数に適切な整数を入力してください')
7     print('プログラムを終了します')
```

実行時エラー発生時の処理

付録
B

例外処理

try:

例外処理の対象とする処理

except 実行時エラーの名称:

実行時エラー発生時の処理

※ except は複数記述することができる。

※ 実行時エラーの名称を省略するとすべての実行時エラーに対応する処理となる。

INDEX

記号・数字

-	035, 114
!=	131
%	035
&	114
()	043, 101, 215
*	035, 037
**	035
**kwargs	229
*=	057
*args	229
,	053
/	035
//	035
/=	057
:	094, 130
;	122
[]	081
^	114
_	049
{}	094, 106
	114
\	039, 122
+	035, 037
+=	057
<	131
<=	131
=	046
=	057

==	131
>	131
>=	131
2次元リスト	111

A

abs 関数	281
AI	127
Anaconda	014
and	138
append 関数	087, 244
ASCII	135
as 文	292
AttributeError	357

B

bool 型	061, 136
bool 関数	065
break 文	178

C

capitalize メソッド(str 型オブジェクト)	249
ceil 関数(math モジュール)	291
class 文	255
client モジュール(http パッケージ)	300
close 関数	285
conda list コマンド	306
continue 文	179
count メソッド(str 型オブジェクト)	249

csv モジュール	289
CUI	324

D

datetime モジュール	289
dateutil モジュール	305
def 文	201
del 文	096
dict 関数	109, 281
dict クラス	252

E

elif ブロック	147
else ブロック	126, 346
email モジュール	289
Exception	021

F

False	136
float 型	061
float 関数	065, 281
float クラス	252
floor 関数 (math モジュール)	291
format 関数	070, 244
for ブロック	172
for 文	171
from 文	292
f-string	072
function 型	250

G

get 関数 (requests モジュール)	311
global 文	232
GUI	324

H

HTTPConnection 関数 (client モジュール)	300
http パッケージ	299

I

id 関数	257
IDE	022
identity	257
if-elif 構文	147
if-else 構文	143
if のみの構文	144
if ブロック	126
if 文	126
if 文のネスト	150
import 文	290
ImportError	358
in 演算子	133
IndentationError	129, 353
IndexError	084, 354
input 関数	058, 281
int 型	061
int 関数	065, 281
int クラス	252
IoT	328
ipnb ファイル	019
isinstance 関数	182

J

json モジュール	289
Jupyter Notebook	030
JupyterLab	013

K

KeyboardInterrupt	358
KeyError	355

L

len 関数	086, 281
list 関数	108, 281
list クラス	252
log 関数 (math モジュール)	293
lower メソッド (str 型オブジェクト)	249

M

math モジュール	289
matplotlib モジュール	305, 306
max 関数 (math モジュール)	281
min 関数 (math モジュール)	281
ModuleNotFoundError	357

N

NameError	348
None	217
not	138
NumPy モジュール	305

O

open 関数	281, 284
or	138
os モジュール	289

P

Pandas モジュール	305
pass	146
PEP8	129
pi (math モジュール)	291
pip install コマンド	306
plot 関数 (pyplot モジュール)	307
print 関数	016, 281
py ファイル	023
Pygame モジュール	305
pyplot モジュール	307
Python	012
Python 3 系	019
Python インタプリタ	021
pyYAML モジュール	305

R

randint 関数 (random モジュール)	315
random モジュール	289, 315
range 関数	174
remove 関数	088
replace メソッド (str 型オブジェクト)	249

requests モジュール	305, 309
Response オブジェクト	311
return 文	213
round 関数	281

S

scikit-learn モジュール	305
SciPy モジュール	305
self	254
set 関数	108
set クラス	252
Shift_JIS	135
simplejson モジュール	305
split メソッド (str 型オブジェクト)	249
SQL	322
sqlite3 モジュール	323
str 型	061, 249
str 関数	065, 281
str クラス	252
strip メソッド (str 型オブジェクト)	249
sukkiri.jp	338
sum 関数	085, 281
SymPy モジュール	305
SyntaxError	021

T

TabError	353
TensorFlow モジュール	305
title メソッド (str 型オブジェクト)	249
tkinter モジュール	324
try-except 文	362
try ブロック	362
True	136
tuple 関数	108, 281
type 関数	063, 281
TypeError	060, 349

U

UI	324
UnboundLocalError	356

upper メソッド (str 型オブジェクト)	249
UTF-8	135

V

ValueError	355, 362
values 関数	099

W

WebAPI	311
Web アプリケーション	326
Web フレームワーク	326
while ブロック	163
while 文	163
with ブロック	285
with 文	285
write 関数	284

Z

ZeroDivisionError	352, 363
zip 関数	109

あ

アップパーキャメルケース	052
余り	035
アンダースコア	049
アンパック代入	053, 221
暗黙の型変換	068
暗黙のタプル	220

い

入れ子	112
インデックス	080
インデント	128, 346
引用符	345

う

ウィンドウアプリケーション	324
---------------------	-----

え

エスケープシーケンス	039
------------------	-----

エディタ	022
エラー	016
エラーメッセージ	340
円記号	040
演算子	035

お

オブジェクト	246
オブジェクト指向プログラミング	256, 334
オブジェクトのコピー	262
オブジェクトの生成	252
オペランド	041

か

改行	039
外部ライブラリ	304
カウンタ変数	163
書き込みモード	284
角カッコ	081
掛け算	035
加算	035
型	061
可変オブジェクト	269
可変長引数	228
空のコレクション	142
空の文字列	142
空ブロック	146
仮引数	201
関係演算子	131
関数	016, 197
関数オブジェクト	250
関数型プログラミング	334
関数の定義	200
関数の呼び出し	200, 201
関数の連携	217
関数ブロック	201
関数呼び出し演算子	215
カンマ	053

き

キー	092
キーボード入力	057
機械学習	331
強制終了	165, 358

く

組み込み関数	281
クラス	251
クラスの定義	255
繰り返し	121
グローバル変数	231

け

減算	035
----	-----

こ

構造化定理	121
構文エラー	021, 341
コメント	025
コレクション	079
コレクションの相互変換	108
コレクションのネスト	110
コレクション変換関数	254
コロソ	094
コンテナ	079

さ

差集合	114
算術演算子	035
参照	046, 260

し

シーケンス	102
式	040
識別子	049
辞書	107
実行時エラー	341
実引数	201, 208
集合	107

集合演算	112
順次	121
商	035
条件式	126
乗算	035
小数	061
除算	035
真偽値	061, 136
シングルクォーテーション	017, 036

す

数値リテラル	036
スタックトレース	342
ストリーム	286
スネークケース	052
スライス	090

せ

制御構造	121
制御文字	039
整数	061
積集合	114
セット	104
セットの定義	106
セミコロソ	122
セル	018

そ

添え字	080, 101
ソースコード	020
ソースファイル	021
属性	256

た

対称差	114
代入	046
代入演算子	048
足し算	035
タブ文字	130, 353
タプル	100

タブルの定義	100
ダブルクォーテーション	036

ち

チェインケース	052
チャットボット	127

つ

追記モード	284
-------------	-----

て

定義	046
ディクショナリ	092
ディクショナリの定義	094
ディクショナリの要素の削除	097
ディクショナリの要素の参照	095
ディクショナリの要素の順序	099
ディクショナリの要素の追加	096
ディクショナリの要素の変更	096
データ型	061
データ型の変換	065
データ構造	079
データサイエンス	331
データベース	322
手続き型プログラミング	335
デフォルト引数	221

と

等価判定	258
統合開発環境	022
等値判定	258

な

流れ図	125
名前の衝突	202
波カッコ	094, 106

ね

ネスト	112
-----------	-----

の

ノートブック	019
--------------	-----

は

配列	107
破壊的な関数	273
バックスラッシュ	039, 122
パッケージ	299
反復	037

ひ

比較演算子	131
引き算	035
引数	201, 206
引数のキーワード指定	226
評価	041
標準出力	281
標準入力	281
標準ライブラリ	289

ふ

ファイルオブジェクト	284
ファイルに書き込む	284
ファイルを閉じる	285
ファイルを開く	284
ファイルを読み込む	315
複合代入演算子	057
部品化	197
不変オブジェクト	269
フラグ	167
ブレースホルダー	071
フローチャート	125
ブロック	126
文	120
分岐	121

へ

べき乗	035
別名	292
変数	045

変数名のルール 049

ほ

防御的コピー 267

ま

マイコン 329

マシン語 021

マップ 107

丸カッコ 043, 101, 215

む

無限ループ 164

め

明示的な型変換 068

メソッド 246

も

モード 283

モジュール 287

モジュールの取り込み 289

文字列 061

文字列の大小比較 135

文字列リテラル 036

戻り値 212

ゆ

ユーザーインターフェース 324

優先順位 042

よ

要素 080

読み込みモード 284

予約語 049, 051

ら

ライブラリ 288

り

リスト 080

リストの定義 081

リストの要素数 086

リストの要素の合計 084

リストの要素の削除 088

リストの要素の参照 083

リストの要素の追加 087

リストの要素の変更 089

リテラル 036, 252

る

るい乗 035

ループ 121

ループカウンタ 163

ループ変数 163

れ

例外 021, 341

例外処理 362

連結 037

ろ

ローカル変数 201, 205

ローカル変数の独立性 204

ローワーキャメルケース 052

論理演算子 138

わ

ワイルドカードインポート 295

和集合 114

割り算 035

■著者

国本大悟(くにもと・だいご)

文学部・史学科卒。大学では漢文を読みつつ、IT 系技術を独学。会社でシステム開発やネットワーク・サーバ構築等に携わった後、フリーランスとして独立する。システムの提案、設計から開発を行う一方、プログラミングやネットワーク等の IT 研修に力を入れており、大規模 Sier やインフラ系企業での実績多数。

須藤秋良(すとう・あきよし)

現在の市場では珍しいフリーランスのデータサイエンティスト。
実際にデータ分析を行うことよりも、最近は関東圏で開催されるデータサイエンスの研修講師として登壇することが非常に多く「理論よし！実装よし！それを他人にわかりやすく教えるもよし！」の 3 本柱を掲げている。超入門から上級編までの幅広いセミナーを担当し、セミナー受講者は総計 2,000 人を越える。

■監修・執筆協力

中山清喬(なかやま・きよたか)

株式会社フレアリンク代表取締役。IBM 内の先進技術部隊に所属しシステム構築現場を数多く支援。退職後も研究開発・技術適用支援・教育研修・執筆講演・コンサルティング等を通じ、「技術を味方につける経営」を支援。現役プログラマ。講義スタイルは「ふんわりスパルタ」。

飯田理恵子(いいだ・りえこ)

経営学部 情報管理学科卒。長年、大手金融グループの基幹系システムの開発と保守に SE として携わる。現在は株式会社フレアリンクにて、ソフトウェア開発、コンテンツ制作、経営企画などを通して技術の伝達を支援中。

■イラスト

高田ゲンキ(たかた・げんき)

神奈川県出身／1976 年生。東海大学文学部卒業後、デザイナー職を経て、2004 年よりフリーランス・イラストレーターとして活動。書籍・雑誌・web・広告等で活動中。

ホームページ <http://www.genki119.com>

STAFF	
編集	坂井直美 片元 諭
イラスト	高田ゲンキ
DTP 制作	SeaGrape
カバーデザイン	阿部 修 (G-Co.Inc.)
カバー制作	高橋結花・鈴木 薫
編集長	玉巻秀雄

「スッキリわかる」シリーズ



プログラミングの基礎はこの1冊でマスター！ スッキリわかるC言語入門

定価：本体 ¥2,700+税
著者：中山清喬 著
A5判、752P
ISBN：978-4-295-00368-7

「なぜ?」「どうして?」にしっかりと答えながら解説を進めていく構成によって、「ポインタ」や「文字列操作」など、Cの学習でつまづきやすい部分が楽しく、グングン身に付きます。定番付録「エラー解決・虎の巻」も収録。学習用の開発環境は、複数のOSに対応し手軽に準備できる仮想化による学習環境を提供しています。



Javaの基本からオブジェクト指向までを 「途中で挫折せず」学べる入門書

スッキリわかるJava入門 第3版

定価：本体 ¥2,600+税
著者：中山清喬 / 国本大悟
監修：株式会社フレアリンク
A5判、768P ISBN：978-4-295-00780-7

基礎からオブジェクト指向までの「なぜ?」が必ずわかる決定版入門書。第3版ではJava11を基準に改修し、コレクションを増補。主要コードにはQRコードを配して、仮想環境「dokoJava」にアクセスしやすくなりました。



Javaの開発現場で 即戦力になれる知識が身に付く! スッキリわかるJava入門 実践編 第2版

定価：本体 ¥2,800+税
著者：中山清喬
A5判、628P ISBN：978-4-8443-3677-8

ラムダ式や日付APIの解説を増補してJava8に対応! Javaエンジニアとして最低限、現場で必要とされる周辺知識を易しく・楽しく・スッキリと分かりやすく解説。「Javaはマスターしたけれど、開発現場で必要とされる最低限の知識を身につけておきたい!」という方にお勧めです。





現場で使えるSQLがドリルで グングン身に付く! スッキリわかるSQL入門 第2版 ドリル 222問付き!

定価：本体 ¥2,800+税
著者：中山清喬 / 飯田理恵子
監修：株式会社フレアリンク
A5判、488P ISBN：978-4-295-00509-4

やさしく、楽しくデータベースとSQLに関する実践的な知識が学べる入門書です。手軽に取り組めるクラウド環境「dokoQL」と、徹底的にアウトプットを図る222問の巻末ドリルで、初學者でも現場で使える力がしっかり身に付きます。

FE このマークがある書籍は「基本情報処理技術者試験」(略号:FE)の午後問題対策に有効です。

本書のご感想をぜひお寄せください https://book.impress.co.jp/books/1118101169		
読者登録サービス 	アンケート回答者の中から、抽選で商品券(1万円分)や図書カード(1,000円分)などを毎月プレゼント。 当選は賞品の発送をもって代えさせていただきます。	

■ 商品に関する問い合わせ先

インプレスブックส์のお問い合わせフォームより入力してください。

<https://book.impress.co.jp/info/>

上記フォームがご利用頂けない場合のメールでの問い合わせ先

info@impress.co.jp

- 本書の内容に関するご質問は、お問い合わせフォーム、メールまたは封書にて書名・ISBN・お名前・電話番号と該当するページや具体的な質問内容、お使いの動作環境などを明記のうえ、お問い合わせください。
- 電話やFAX等でのご質問には対応しておりません。なお、本書の範囲を超える質問に関しましてはお答えできませんのでご了承ください。
- インプレスブックส์ (<https://book.impress.co.jp/>) では、本書を含めインプレスの出版物に関するサポート情報などを提供しておりますのでそちらもご覧ください。
- 該当書籍の奥付に記載されている初版発行日から5年が経過した場合、もしくは該当書籍で紹介している製品やサービスについて提供会社によるサポートが終了した場合は、ご質問にお答えしかねる場合があります。

■ 落丁・乱丁本などの問い合わせ先

TEL 03-6837-5016 FAX 03-6837-5023
service@impress.co.jp

(受付時間／10:00-12:00、13:00-17:30 土日、祝祭日を除く)

- 古書店で購入されたものについてはお取り替えできません。

■ 書店／販売店の窓口

株式会社インプレス 受注センター
 TEL 048-449-8040
 FAX 048-449-8041
 株式会社インプレス 出版営業部
 TEL 03-6837-4635

スッキリわかる Python 入門

2019年 6月11日 初版発行

2020年 4月 1日 第1版第3刷発行

監 修 株式会社フレアリンク

著 者 国本大悟、須藤秋良

発行人 小川 亨

編集人 高橋隆志

発行所 株式会社インプレス

〒101-0051 東京都千代田区神田神保町一丁目105番地

ホームページ <https://book.impress.co.jp/>

本書は著作権法上の保護を受けています。本書の一部あるいは全部について（ソフトウェア及びプログラムを含む）、株式会社インプレスから文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

Copyright © 2019 Daigo Kunimoto / Akiyoshi Sutoh. All rights reserved.

印刷所 日経印刷株式会社

ISBN978-4-295-00632-9 C3055

Printed in Japan