

| | |
|-------------|--|
| Title | プログラミング演習 Python 2019 |
| Author(s) | 喜多, 一 |
| Citation | (2020): 1-200 |
| Issue Date | 2020-02-13 |
| URL | http://hdl.handle.net/2433/245698 |
| Right | 本書はCC-BY-NC-NDライセンスによって許諾されています。ライセンスの内容を知りたい方は https://creativecommons.org/licenses/by-nc-nd/4.0/deed.ja でご確認ください。 |
| Type | Learning Material |
| Textversion | publisher |

プログラミング演習 Python 2019

Python の予約語、薄い字のものは本書では扱いません

京都大学 国際高等教育院 喜多 一

Version 2020/02/13

0. まえがき

本書は京都大学の全学共通科目として実施されるプログラミング演習（Python）の教科書として作成されたものです。

0.1 目的と到達目標

この授業の目的・到達目標は以下のように定めています。

0.1.1 目的

プログラミング言語 Python は初学者にも学びやすい言語である一方で、さまざまな応用も可能である。近年では学術研究にも利用が広がっている。本授業ではプログラミングの初学者を対象に Python を用いたプログラミングを演習方式で学ぶ。

0.1.2 到達目標

- Python によるプログラムの実行についての基本操作ができるようになる。
- Python プログラムを構成する基本的要素の機能と書式について説明し、例題を用いて実行例を構成できるようになる。
- Python を用いて簡単なプログラムを自ら設計、実装、テストできるようになる。

0.2 屋上屋を重ねる理由

本書は 2018 年度の授業実践に基づいて執筆しました。Python についてはすでに多くの入門書が刊行されているのですが、わざわざ屋上屋を重ねるように教科書を作成する理由は以下のようなことからです。

- この授業は Python というプログラミング言語を紹介するのではなく、Python というプログラミング言語で実際にプログラムを書く（書けるようになる）ことを目的にしています。多くの解説書がプログラミング言語の紹介に終始しがちです。
- 初学者にとってプログラミング言語の学習はさまざまな躓きを乗り越えることです。初学者にとって深刻な躓きでも、ある程度、プログラミングの経験を積

むと躓いたことさえ忘れてしまいます。この教科書は授業実践を通じて初学者が躓く点やそれへの手助けを意識して記述しています。

- 上記とも関連しますが、プログラミング言語の学習は実際に手を動かしてプログラムを書き、実行することで成り立ちます。本書では実際の演習への指示を示しています。

本書は学習の方向付けとして Python を用いたプログラミングの基本を解説していますが、プログラミング言語の仕様を網羅的に紹介するものではありません。別途、Python についての解説書などを用意して受講されることをお勧めします。

0.3 本書の構成について

本書は 2018 年度の実践にもとづいて構成されています。このため、前から順に演習していただけるように構成しています。また授業に関連はするが、少し横道にそれる話題については「コラム」としてとりまとめてあります。

0.4 本書での表記

Python については簡単な命令は Python Shell という対話的な環境で 1 行ずつ試してみる一方で、まとめたプログラム（スクリプト）をエディタ上で作成し、一括実行するという 2 通りの方法を使い分けて学習を進めることができます。

Python Shell で試してほしい機能については、文中で

a = 1 + 2

と Arial フォントにより赤字で示しました。それを実行した結果については青字で

3

のように示しています。実際に試しながら学習してください。

一方、まとめたプログラムについては、3 列からなる表形式で

| 行 | ソースコード | 説明 |
|---|-----------|----|
| 1 | a = 1 + 2 | |
| 2 | print(a) | |

のように示しています。ソースコードの部分をエディタで入力し、実行してください。

0.5 コピペに注意

本書の掲載されているソースコードは Word でのフォーマッティングと PDF への変換を行っているため、自動で文字が変わっている場合があり PDF からコピーペーストしてもプログラムとしてはエラーになる場合があります。

謝辞

本書の改訂にあたり、macOS での利用に関わる部分など、京都大学情報環境機構、森村吉貴准教授に情報の提供、利用手順の確認などを頂きました。

本書は CC-BY-NC-ND ライセンスによって許諾されています。ライセンスの内容を知りたい方は <https://creativecommons.org/licenses/by-nc-nd/4.0/deed.ja> でご確認ください。



目次

| | |
|--|-----------|
| 0. まえがき | 2 |
| 0.1 目的と到達目標 | 2 |
| 0.2 屋上屋を重ねる理由 | 2 |
| 0.3 本書の構成について | 3 |
| 0.4 本書での表記 | 3 |
| 0.5 コピペに注意 | 4 |
| 謝辞 | 4 |
| 目次 | 5 |
| 1. コンピュータとプログラミング | 9 |
| 1.1 この章の目的 | 9 |
| 1.2 コンピュータとプログラム | 9 |
| 1.3 コンピュータの仕組み | 12 |
| 1.4 プログラミング言語 | 13 |
| 1.5 プログラミング言語 Python | 16 |
| 1.6 さまざまな応用 | 17 |
| 1.7 プログラミングの学び方 | 18 |
| 1.8 プログラムを構成する基礎的な概念 | 24 |
| 1.9 プログラムの「どこ」を作るか | 24 |
| 参考文献 | 25 |
| 2. Python の実行環境と使い方 | 26 |
| 2.1 本章の学習の目標 | 26 |
| 2.2 学習環境の想定 | 26 |
| 2.3 準備 | 26 |
| 2.4 IDLE の起動 | 27 |
| 2.5 Python シェル | 28 |
| 2.6 スクリプトの作成と実行 | 30 |
| 2.7 Anaconda Prompt での作業フォルダの設定 | 32 |
| 2.8 IDLE のキー操作など | 34 |
| 2.9 Python コマンドの実行 | 34 |
| 2.10 Python を学ぶ環境づくり | 36 |
| 2.11 Mac ユーザへ | 37 |
| 参考文献 | 39 |
| 3. 変数と演算、代入 | 40 |
| 3.1 本章の学習の目標 | 40 |
| 3.2 プログラムの実行の流れと情報の流れ | 40 |
| 3.3 変数の名前 | 41 |
| 3.4 変数への代入と値の評価 | 43 |
| 3.5 代入演算子 | 44 |
| 3.6 Python で使えるデータ型 | 45 |

| | | |
|-----------|--|-----------|
| 3.7 | Python の変数のより正しい理解 | 47 |
| 3.8 | 例題：平方根を求める | 48 |
| 3.9 | 読み易い式の表記 | 50 |
| | 参考文献 | 50 |
| 4. | 制御構造 | 51 |
| 4.1 | 本章の学習の目標 | 51 |
| 4.2 | for 文と range() 関数を用いた一定回数の繰り返し | 51 |
| 4.3 | for 文の書き方 .. | 52 |
| 4.4 | Python でのブロック | 53 |
| 4.5 | for 文内での処理の制御 | 54 |
| 4.6 | range() 関数 | 55 |
| 4.7 | for 文の入れ子 | 56 |
| 4.8 | while 文による繰り返し | 57 |
| 4.9 | if 文による分岐 | 59 |
| 4.10 | 条件式の書き方 | 61 |
| 4.11 | if 文の入れ子 | 62 |
| 4.12 | 端末からの入力 | 63 |
| 4.13 | エラーへの対処 | 64 |
| 4.14 | Python での数学関数 | 65 |
| 4.15 | 数値を表示する際のフォーマット指定 | 66 |
| 4.16 | 力試し | 67 |
| 5. | 関数を使った処理のカプセル化 | 68 |
| 5.1 | 本章の学習の目標 | 68 |
| 5.2 | 前章の例題から | 68 |
| 5.3 | 関数 square_root() を実装する | 69 |
| 5.4 | 関数定義の書式 | 70 |
| 5.5 | 仮引数と実引数 | 71 |
| 5.6 | 関数内の変数の扱い | 72 |
| 5.7 | 関数の利用パターン | 72 |
| 5.8 | 関数の呼び出しと関数オブジェクトの引き渡し | 74 |
| 5.9 | デフォルト引数値とキーワード引数 | 74 |
| 6. | Turtle で遊ぶ | 76 |
| 6.1 | 本章の学習の目標 | 76 |
| 6.2 | Turtle –由緒正しき亀さん | 76 |
| 6.3 | Python の Turtle モジュール | 77 |
| 6.4 | 使ってみよう | 77 |
| 6.5 | Turtle モジュールの主な関数 | 79 |
| 6.6 | 複数のタートルを動かす | 79 |
| 6.7 | 作品作りのためのヒント | 81 |
| 6.8 | Turtle Demo | 83 |
| 6.9 | 課題 Turtle の作品制作 | 84 |
| | 参考文献 | 84 |
| 7. | Tkinter で作る GUI アプリケーション(1) | 89 |
| 7.1 | 本章の学習の目標 | 89 |
| 7.2 | GUI とイベント駆動型プログラミング | 89 |

| | | |
|------------|---|------------|
| 7.3 | モデルとユーザーインターフェイスの分離 | 90 |
| 7.4 | tkinter..... | 91 |
| 7.5 | tkinter の例題(<code>tkdemo-2term.py</code>) | 92 |
| 7.6 | tkinter を用いたプログラムの基本構成 | 96 |
| 7.7 | gird によるレイアウト | 97 |
| 7.8 | lambda (λ) 表現を使った Call Back 関数の記述..... | 98 |
| 7.9 | ウィジェットの体裁の調整 | 102 |
| 7.10 | tkinter の終わり方 | 103 |
| 7.11 | Frame クラスを拡張する方式での実装法..... | 103 |
| | 参考文献 | 107 |
| 8. | Tkinter で作る GUI アプリケーション(2) | 108 |
| 8.1 | 本章の学習の目標 | 108 |
| 8.2 | 自律的に動作するプログラムと GUI との衝突 | 108 |
| 8.3 | モジュール | 109 |
| 8.4 | tkinter を用いたアナログ時計プログラム | 109 |
| 9. | クラス | 117 |
| 9.1 | 本章の学習の目標 | 117 |
| 9.2 | オブジェクト指向プログラミング | 117 |
| 9.3 | Python でのクラスの書き方、使い方 | 118 |
| 9.4 | クラスの変数とアクセスの制限 | 120 |
| 9.5 | 継承 | 123 |
| 9.6 | インスタンスを起点にクラスを設計する | 123 |
| 10. | リスト | 125 |
| 10.1 | 本章の学習の目標 | 125 |
| 10.2 | Python Shell を用いた学習 | 125 |
| 10.3 | リストとは | 126 |
| 10.4 | リストの生成 | 126 |
| 10.5 | リストの要素へのアクセス | 127 |
| 10.6 | リストを操作する <code>for</code> 文 | 128 |
| 10.7 | 負の添え字とスライス | 129 |
| 10.8 | リストへの追加、結合 | 130 |
| 10.9 | リストのリスト | 131 |
| 10.10 | 内包表記 | 132 |
| 10.11 | リストの代入と複製 | 132 |
| 10.12 | イミュータブルとミュータブル | 134 |
| 10.13 | 浅いコピー、深いコピー | 135 |
| 11. | ファイル入出力 | 137 |
| 11.1 | 本章の学習の目標 | 137 |
| 11.2 | データを永続的に利用するには | 137 |
| 11.3 | ファイルについて | 137 |
| 11.4 | まずは動かしてみよう | 140 |
| 11.5 | Python でのファイルの読み書き | 141 |
| 11.6 | 例題 1 波の近似 | 143 |
| 11.7 | 例題 2 | 148 |

| | | |
|------------|-------------------------------|------------|
| 12. | 三目並べで学ぶプログラム開発 | 151 |
| 12.1 | 本章の学習の目標 | 151 |
| 12.2 | プログラムを開発するということ | 151 |
| 12.3 | 設計手順—コンピュータを使う前にすることがある | 151 |
| 12.4 | 三目並べを例にしたプログラムの設計 | 152 |
| 12.5 | プログラムの実装 | 157 |
| 12.6 | 力試し | 171 |
| 12.7 | プログラムの開発に関連するいくつかの話題 | 172 |
| 13. | Python の学術利用 | 174 |
| 13.1 | 本章の学習の目標 | 174 |
| 13.2 | import 時の別名 | 174 |
| 13.3 | NumPy..... | 175 |
| 13.4 | Matplotlib..... | 178 |
| 13.5 | pandas | 187 |
| 13.6 | 課題 | 194 |
| | 参考文献 | 196 |
| 14. | 振り返りとこれから | 197 |
| 14.1 | 本章の学習の目標 | 197 |
| 14.2 | 振り返り | 197 |
| 14.3 | Python の利用環境..... | 197 |
| 14.4 | モジュール等の追加 | 197 |
| 14.5 | 本書で紹介しなかった話題 | 198 |
| 14.6 | 感謝と恩返し—学んだことをどう活かすのか | 198 |
| 15. | IDLE Python 便利帳 | 199 |
| 15.1 | Python 便利メモ | 199 |
| 15.2 | ファイル名に注意 | 199 |
| 15.3 | IDLE メモ—Python シェルのキー操作 | 199 |
| 15.4 | IDLE メモ—エディタ | 200 |

1. コンピュータとプログラミング

1.1 この章の目的

- プログラミングの対象となるコンピュータの動作の概略とそこで実行されるプログラムの役割について知る。
- プログラミングにおけるプログラミング言語の役割について知る。
- プログラムを作成するさまざまな対象、応用について知る。
- プログラミングの学び方について知る。

演習 1. この授業の受講動機

以下の質問に答えてください。

1. なぜこの授業に参加しようと思いましたか？
2. プログラミングを学びたい理由は何ですか？
3. なぜ Python 言語を学びたいのですか？
4. プログラミング（の学習）の経験があれば教えてください。有無、どれぐらい？
5. プログラミング（の学習）の経験がある方は、使ったプログラミング言語をお教えください。

1.2 コンピュータとプログラム

1.2.1 プログラムで動く機械

「ジャカード織機」という機械の名前を歴史の授業で聞いたことがあると思います。織物は縦糸の間に横糸を通して織り上げていきますが、どの縦糸を横糸の上にして、どの縦糸を下にするかを変えることで「柄」を織ることができます。縦糸の上下についての指示穴の開いた厚紙（パンチカード、「紋紙」と呼ばれています）で指示できるようにしたものがジャカード織機です。沢山のパンチカードを紐で閉じて順に送ることで複雑な模様を実現します。紋紙をかけ替えることで、別の模様を織りだすことが可能です。

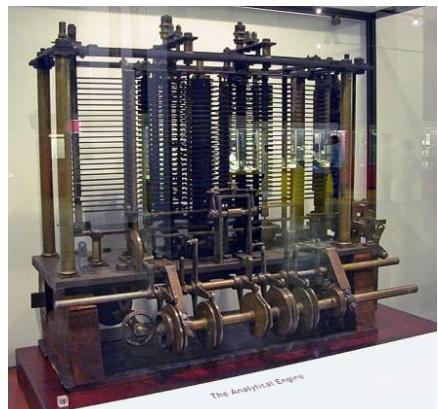
京都の西陣は機織りが地場産業ですが、ジャカード織機がまだ現役で稼働してい

ました。図 1

英国の C. バベッジ (1791–1871) は機械仕掛けで動く計算する装置を開発しようとした人ですが、階差数列から数表を自動生成する機械（階差エンジン）に続いて、ジャカード織機にヒントを得てプログラムで動く機械仕掛けの計算機（解析エンジン）を作ろうとしました。残念ながら、完成には至りませんでしたが、プログラムで計算する機械の先駆けとされています。



ジャカード織機と紋紙
京都、フクオカ機業にて撮影



バベッジの解析エンジン

図 1 ジャカード織機と解析エンジン

https://commons.wikimedia.org/wiki/File:AnalyticalMachine_Babbage_London.jpg

1.2.2 コンピュータの構成要素は電気で動かせる「スイッチ」

バベッジの時には複雑な動作の実現には歯車などの機械仕掛けだけが利用可能でした。その後、「電気」で別の「電気スイッチ」を取り切りすることで機械を構成するようになりました。その一つが電磁石で機械的なスイッチを動かす「**継電器（リレー）**」です。実際、「継電器」でコンピュータを構成することも試みました。しかしながら、この方式では電気では動くのですが機械的動作を伴うため動作が遅いという欠点がありました。

その後、真空中の電極（陰極、陽極）間を流れる電子をその間においた別の電極に印加する電圧で制御する**真空管**が発明されコンピュータの構成に応用されました。真空管は電子的動作で速度が速いという利点がありましたが、陰極から電子を放出させるための加熱用にフィラメントを用いていたことから寿命があるという欠点が

ありました。

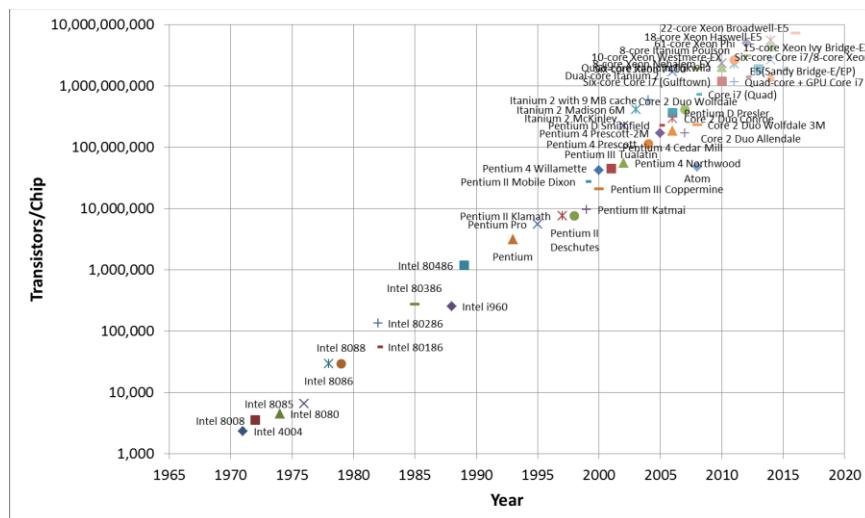
真空管のような動作を半導体の固体内で実現する素子としてトランジスタが発明され、寿命が長く、小型で消費電力も小さいという特性があり、多数の個別部品のトランジスタを配線する形でコンピュータが構成されるようになります。

さらに、1つの半導体チップ上に多数のトランジスタやその間の配線などを印刷技術で実現する集積回路が開発され、小型化、低価格化を実現します。集積回路技術によりコンピュータの主要部分を1つのチップ上に実現したマイクロプロセッサが開発され、これが現代のパーソナルコンピュータやスマートフォンなど誰でも手にすることができる小型で安価なコンピュータを実現させます。



図 2 論理素子の進歩

その後、1つの半導体チップ上に集積できるトランジスタ数（集積度）が40年で100万倍という飛躍的な進歩を見せます。性能が何桁も変わる技術革新は記憶装置の容量、通信速度でも達成されています。このような技術革新のおかげで現在ではスマートフォンでYouTubeなどの動画を楽しめるようになりました。



1.3 コンピュータの仕組み

1.3.1 プログラム内蔵方式

現代のコンピュータは複雑な情報処理をどのように実行しているのでしょうか。

コンピュータ（のハードウェア）が一度にできる動作は単純なもので、情報処理の複雑な仕事は、単純な動作の組み合わせをプログラムとして示すことで実現しています。

プログラムはメモリ上に格納され、高速に読み出し、実行できるようになっています。このようなコンピュータの構成方式を「プログラム内蔵方式」¹と呼び、家電製品に用いられる小さなマイコンからスーパーコンピューターまですべてこの方式が採用されています。

させたい仕事に応じてプログラムを切り替えて実行することで同じ（ハードウェアの）コンピュータがさまざまな仕事を行えます。

プログラミングとは実現したい情報処理をプログラムとして記述することです。

1.3.2 コンピュータの構成と動作

コンピュータのハードウェアの主要な構成要素はCPUとメモリです。

CPU内には以下のようないくつかの要素があります。

- メモリから命令を取り込み、命令を解析する仕組み
- 実行中のプログラムのメモリ上の番地を示すカウンタ
- データを保持する仕掛け（レジスタ）
- その値に算術や論理演算などをほどこす計算機能（ALU）

コンピュータの基本動作は以下のように単純なものです。

- メモリ上にプログラムが（何らかの手段で）配置されています
- CPUにプログラムの実行開始場所を与えます
- 以下を繰り返します
 1. CPUはメモリ上のプログラムから1ステップ分を読み出し、指示に従って計算、データの転送などを行います。

¹ 最初期のコンピュータである ENIAC では、計算の設定は「プログラム内蔵方式」ではなく、ケーブルでの配線変更で行われましたが、後継機の EDVAC の開発にあたってこの考え方が提案されました。提案のレポートがフォン・ノイマンによって提出されたことから「ノイマン型コンピュータ」とも呼ばれます。

- ✧ 計算結果をメモリに書き出す場合もあります。
 - ✧ 入出力を行う場合もあります。
2. CPU は実行の対象を次に進めます。
- ✧ 実行場所はプログラムによって変化する場合があります。

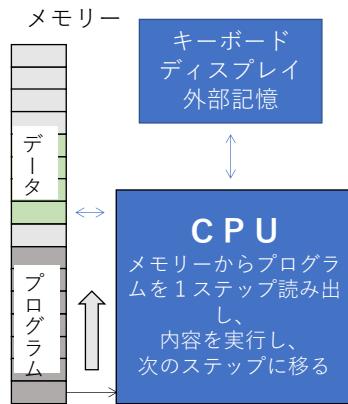


図 4 コンピュータ（ハードウェア）の構成

実際のコンピュータのハードウェア(CPU+メモリ)が実行できる命令はハードウェアで高速に処理することを簡単に実現するために極めて単純な命令群として構成されます。このような命令を「機械語」と呼びます。

1.4 プログラミング言語

機械語で複雑なコンピュータの応用をプログラミングすることは大変難しい作業です。この問題を解決するために考案されたものがプログラミング言語です。プログラミング言語は次の2つの考え方で成り立っています。

- 機械語よりも数式などに近い形で人間により分かりやすいプログラムを書くためにルールを定める（プログラミング言語の仕様の策定）
- そのルールにしたがったプログラム（ソースコードと呼ばれます）を実行するためのプログラムを作成する（プログラミング言語の処理系の構成）

すなわち、プログラミング言語でプログラムを書き、書かれたプログラムを処理系を使って実行するという仕組みです。これは見方を変えれば、「処理系+実際の計算機」で「プログラミング言語で書かれたプログラムを実行できる仮想の計算機」を実現している、ということもできます。

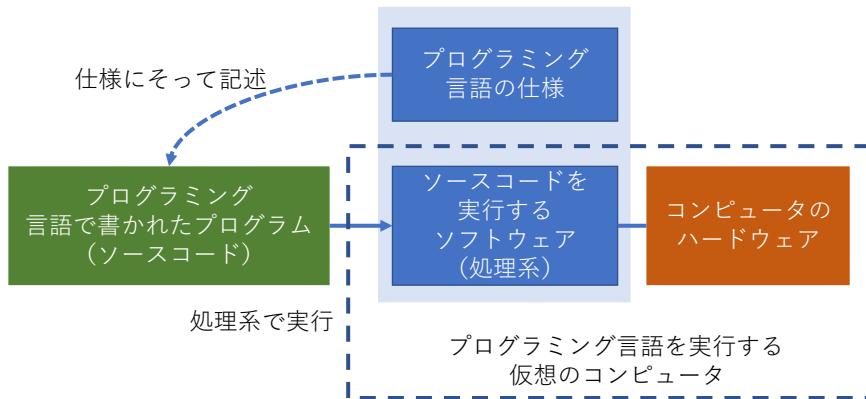


図 5 プログラミング言語と処理系

1.4.1 さまざまなプログラミング言語

以下の表のようにさまざまなプログラミング言語が開発され利用されています。

| FORTRAN | COBOL | ALGOL | Pascal | PL/I |
|---------|-----------|--------|------------|---------|
| BASIC | C, ++, C# | Java | Go | swift |
| Perl | Ruby | Python | JavaScript | LISP |
| Haskell | R | Matlab | ProLog | Scratch |

また、プログラミング言語に類似したものとして、Web ページ記述する HTML やページのスタイルを記述する CSS, データを記述する XML や JSON, データベースへの問い合わせを記述する SQL などが併用されることも多いです。

なぜ、これほど多くのプログラミング言語が考案され、利用されているのでしょうか。なぜ、1つの言語に統一されないのでしょうか。

コンピュータ技術の進展は、開発するプログラムの高度化も求めます。それにともなって、プログラムを効果的に記述するための考え方とそれに基づくプログラミング言語が開発されてきました。また、プログラムをより簡単に書きたい、より高速に、より安全に動かしたいという要望が常に存在しています。特定の用途に適したプログラミング言語へのニーズもあります。

これらのことから、新しいプログラミング言語が開発されたり、特定のプログラミング言語の仕様や処理系が改訂されたりしています。他方で、特定のプログラミング言語で開発されたソフトウェア資産や、その言語での開発を望む技術者はそのプログラミング言語の継続的な利用を求めます。古い言語も、それを捨てることは難しいのです。実際 FORTRAN と呼ばれる科学計算用のプログラミング言語は言語としては最長老ですが、さまざまな改訂も行われつつ現役の言語として使用されて

います。

プログラミング言語の開発はソフトウェア企業が行う場合や、個人が発案し、コミュニティで発展させる場合などがあります。企業での開発については、プログラミング言語の処理系そのものを有償で販売することを目的とする場合に加え、自社のニーズから開発するが、処理系の利用を無償にしたり、ソースコードを公開したりするなどの場合もあります。

1.4.2 プログラミング言語の処理系の構成

プログラミング言語で書かれたプログラム（ソースコード）を実際に処理し、実行するためのプログラムの構成方式には以下のようにいくつかのものがあります。

1) コンパイラ方式

ソースコードを一旦、その内容を実行する機械語に翻訳（コンパイル）し、翻訳された機械語を実行する方式です。コンパイルには手間がかかるがコンパイルされた実行可能なプログラムは高速に実行が可能です。

2) インタープリタ方式

ソースコードを逐行的に解釈し、動作を模擬する方式。ソースコードの解釈を行うために実行速度は遅いが、対話的な利用など柔軟性が高い。

3) 中間コード方式

コンパイラ方式とインタープリタ方式の中間的な方法として、実在する CPU の機械語ではなく、その言語用に想定した仮想の計算機の機械語（中間コード）用にソースコードをコンパイルし中間コードをインタープリタ方式実行するプログラム（仮想マシン）で実行する方式です。Java や Python で採用されています。

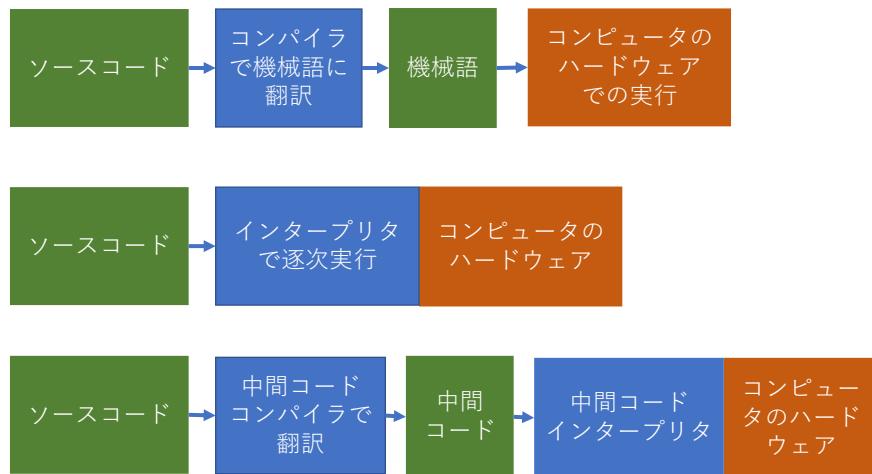


図 6 プログラミング言語処理系の構成方式

1.5 プログラミング言語 Python

1.5.1 Python の歴史

1989 年に Guido van Rossum により実装が始まりました。2000 年に Version 2.0 が公開され、2008 年に Version 3.0 が公開されています。

注意 Python の Version 3 は Version 2 と上位互換でない (Version 3 が Version 2 の仕様を含んでいない)ため、Version 2 で書かれたプログラムを稼働させるため、現在は両方が併用されています。

注意 Mac や Redhat Linux では標準で Python が導入されていますが Version 2 がインストールされている場合があるので Version 3 を利用する際には Version 3 の処理系を別途インストールするとともに、どちらの処理系を利用しているのかを確認する必要があります。

1.5.2 Python の特徴

- 初心者にも学びやすく、他方で高度なプログラミングも可能です。
- 多様な応用が可能です。
- 科学計算などのライブラリ (numpy, scipy, matplotlib, pandas など) が多くの人により開発されています。
- とりわけ、近年のデータ科学、人工知能（機械学習）技術への関心から、この

面でのライブラリが豊富な Python が人気を集めています。

1.5.3 Python の配布パッケージ

Python の処理系はいくつか開発されており、これに開発環境やライブラリなどを組み合わせた配布パッケージも複数あります。本授業に関係するものとして、以下の 2つを挙げておきます。

- Python : Python の設計者による配布パッケージ、Python の処理系としては C 言語で記述された CPython が使われています。
- Anaconda: CPython に科学計算用のモジュールなどを一括したパッケージ。本授業ではこのパッケージの利用を想定しています。

1.6 さまざまな応用

皆さんが Python を学びたい理由として、具体的な応用を想定している人もいるかと思います。Python のさまざまな利用シーンについてみてみましょう。

1.6.1 パーソナルコンピュータのアプリケーション

パーソナルコンピュータ上で動くプログラムはその動作環境から大きく 2 つに分かれます。

- CUI 型の応用プログラム。Windows のコマンドプロンプトなどで動かすプログラムで、キーボードから文字入力を受け取り、画面に文字出力を行う形式のプログラムです。入出力が単純なため比較的学びやすいですが、利用する側には使い勝手が悪くなります。
- GUI 型の応用プログラム。ウィンドウやその上でのボタンなどで操作するプログラムです。利用者には使い慣れた操作を提供できますし、画像などを扱うことも可能ですが、プログラミングすべき事項は多くなります。

ファイルの読み書きやネットワークの操作などは CUI 型でも GUI 型でも共通の方法でのプログラミングが必要です。

アプリケーションの用途としては、以下のようなものが考えられます。

- 科学計算や数値シミュレーション
- 数値、文字列、画像などのデータの加工、分析
- ゲームやグラフィクスの作品
- Web サイトからの情報の自動抽出（Web スクレイピングと呼ばれています）

1.6.2 その他のアプリケーション

- スマートフォンのアプリケーション
- Web サーバなどのネットワーク上のサーバで稼働するプログラム
- 電子回路と連携して動くプログラム。Raspberry PI はこのために開発された Linux や Python が稼働する小型のコンピュータです。

1.7 プログラミングの学び方

1.7.1 プログラミングが難しい理由

コンピュータのプログラミングはさまざまな理由で難しさを抱えています。どこがなぜ難しいのかを把握しておくことは学習の助けになるでしょう。

1) プログラミング言語を構成している概念が分かりにくい

我々が日常、使用している自然言語でも、複雑なことを表現するために文法や語法などのさまざまなルールで運用されています。プログラミング言語も複雑なプログラムをうまく表現するためにさまざまな概念、仕組みが取り入れられています。これらを一度に理解する必要はありません。やさしいものから徐々にステップアップしてゆくことができます。

2) エラーへの対応ができない

プログラムではタイプミスや考え違いなどできまざまなエラー（プログラムに喰う虫に例えてバグと呼ばれます）が発生します。バグは出て当たり前だというぐらいいの気持ちで取り組むことが大事ですが、

- ソースコードの文法にそった正確なタイピングが必要であることを理解する必要があります。
- 文法エラーのほか、プログラムについての考え方により、予期した動作をしないこともあります。
- エラーの大半はプログラミングをしている人間に原因があります。どういうエラーかをしっかりと理解し、対処する経験値を上げることが求められます。
- エラー対応は「エラーが生じたという結果」から「エラーを生じさせている原因」を探る、いわば「逆方向」の推論です。結果に適合すると思われる原因についての仮説をさまざまに想定し、実際にそれが原因なのかを検証して行くこ

とが求められます。

- エラーについて、コンピュータの応答（コンピュータ処理できなくなった状況）とプログラムの実際の誤り箇所は異なっていることがあります。

3) 実現したい機能をプログラムに展開できない

プログラミング言語を構成する要素を学んでも、それをどのように組み合わせればやりたいことが実現できるのかはなかなか分かりません。鋸が金槌を使えるようになっても、家がどのような構成になっているのかを知らなければ家を建てることはできないのと同じです。

やりたいことを普通の言葉でしっかりと分析し、手順を明確にしたうえで、プログラムを書くことが必要です。比較的簡単な応用例を通して学んでいけばいいでしょう。

4) プログラムが複雑になって分からなくなる

プログラムが長くなると、複雑さが増して、何をプログラミングしているのか理解することが急に難しくなります。初学者にとっては 100 行のプログラムでも大変に感じるようですが、長くてもわかりやすいプログラムを書く方法を身に付けながら、100 行程度のプログラムを書くことを目標にするといいでしょう。

5) 大きなプログラムには開発手法がある

世の中の巨大なプログラムはソースコードの行数で数億行にもなるそうです。もちろん一人で書くことはできませんし、すべてを頭に入れることもできません。大きなプログラムを書くには、それなりの方法と道具があります。例えば以下のことです。

- プログラム全体をしっかりと設計する
- 分業できるようにモジュールに分割する
- モジュール単位で実装し、テストをする。
- 全体の組み上げテストをする。

100 行程度のプログラムが書けるようになれば、これらのことと意識して、より大きなプログラムの作成に挑戦されるといいでしょう。

1.7.2 プログラミングの学び方

どのようなことでも「学び方を学ぶ」ことは重要です。

- 外国語はどうやって身に着けますか
- 数学はどうやって身に着けますか

プログラミングの学習は実際に手を動かしてプログラムを書く技能ですので、これらと似た面があります。

1) 動機付け：興味の持てる課題に取り組むこと。

- 学習の動機は学習を続けてゆくうえでとても大事です。難易度の程度はありますが、ご自身が興味の持てる課題・応用に取り組むことで学習の動機を維持しましょう。

2) プログラミングの学習はたくさん読む、書くが基本です。

- 例題をたくさんタイピングして実行する。
 - 「理解して実行」ではなく、「実行を通じて理解」する
 - 単語、記号、表記のパターンを覚える。よく使うパターンを全体として身に着けることも重要です。
 - ソースコードのタイピングをより速く、より正確にすることで学習効率が向上します。

3) 音読、訳読

- 声に出して読む、ソースコードの記号を音読することや、日本語で意味を訳して読むことは、指導を受けたりする際の**コミュニケーションを円滑にする**意味からも重要です。

4) ティンカリング：例題のプログラムをいじくって遊ぶ。

- 例題を少し変えて、どのようなことが可能かを探る。
- 複数の例題を組み合わせることに取り組むことで、プログラムを組み合わせるにはどのような調整が必要かを理解できます。

5) トレース

- プログラムがどのように実行されるのか自分で解釈しながら追いかける（トレースと言います）

6) エラーに対処できるようになること

- プログラミングではエラーへの対処は常に生じます、それ自体が重要な学習目

標です。

- 実際のプログラミングでは予期せぬエラーへの対処が求められますが、意図的にエラーのあるプログラムを書いてみて、何が起きるのかを知ることで経験値を高めることも効果的です。

7) 情報探索

- 以下のような、内容や方法でプログラミングのための情報探索ができること
 - プログラミングのためのより高度な概念やライブラリの使い方を学ぶこと
 - プログラミングを支えるツールを知ること。
 - 書籍やネット上の情報などを探せること
 - 分からないときは人に聞く、これは聞くことができるだけのコミュニケーション力も必要になります。

1.7.3 プログラムで使う文字

私たちはプログラムの中で日本語を含めた文字を扱いたいですが、他方で多くのプログラミング言語が英語を基礎に設計されているため、文字と文字コードに対するある程度の理解と注意が求められます。

- Python (を含めた多くのプログラミング言語)の基本は半角の英数字です。
- 全角文字は「文字列としてのデータ」と「コメント」だけで使います。
- Python では大文字と小文字を区別します(Case Sensitive)。
- プログラミングでは、さまざまな記号も使います。
 - キーボードの位置だけでなく、
 - 他の方とのコミュニケーションのために「呼び方」も覚えることが必要です。
- このほか、Ctrl キーを押しながら、例えば C というキーを押すなどのキー操作も求められます。この場合 Ctrl-C などと記述します。Alt キーについても同様です。



図 7 JIS キーボードの配置、
英文用キーボード(ASCII 配置)では記号などの配置が異なっています。

以下の表に主な記号やその読み方、使用上の注意をまとめおきます。この表は文献[1]から著者の了解を得たうえで、一部補足して作成したものです。

表 1 プログラミングで用いる記号と読み

| 記号 | 読み方 | 注意 |
|----|--------------------------------|---|
| _ | 空白, スペース | 記号欄では分かりやすくするために_と表記しました。 プログラムには半角スペースを用います。日本語文字列として用いる以外の全角スペースは見分けにくいエラーになるので注意。 |
| ! | 感嘆符, エクスクラメーションマーク | |
| " | 二重引用符, ダブルクオーテーション, ダブルクオート | Python では文字列を “ もしくは ‘ で囲みます。どちらでも構いませんが 開始と終了に同じ引用符を使ってください。 |
| ' | 一重引用符, アポストロフィ, シングルクオーテーション | |
| # | シャープ, ナンバー | |
| \$ | ドル, ダラー | |
| % | パーセント | |
| & | アンド, アンパサンド | |

| | | |
|---|--------------------|--|
| * | アステリスク、アスタリスク | |
| + | プラス | |
| , | カンマ、コンマ | |
| - | マイナス、ハイフン | |
| . | ピリオド、ドット | |
| / | 斜線、スラッシュ | |
| : | コロン | これらの違いに注意 |
| ; | セミコロン | |
| < | 小なり、左不等 | |
| > | 大なり、右不等号 | |
| = | イコール、等号 | |
| ? | クエスチョンマーク、疑問符 | |
| @ | アット、アットマーク、単価記号 | |
| ¥ | 円記号 | JIS コードでは \ と同じコードに¥ を割り付けています。Python で用いる unicode (UTF-8) ではこれらは異なるコードになっていますが、Windows の多くのフォントでは\ の代わりに ¥ の字体が収められています。mac ユーザはすなおに逆スラッシュを使ってください。 |
| \ | 逆スラッシュ | |
| ^ | やま、アクサンシルコンフレックス | |
| _ | アンダーライン、アンダースコア、下線 | Python ではアンダースコアを 2 文字続けて__のように使いことが多いです。 |
| | 縦線、縦棒 | |
| ~ | なみ、チルド、チルダ | |
| [| 大括弧、各括弧（開く） | Python を含む多くのプログラミング言語では、さまざまな意味で括弧を使い分けます。タイプミスが発生しやすいです。 |
|] | 〃（閉じる） | |
| { | 中括弧（開く） | |
| } | 〃（閉じる） | |
| (| 小括弧、丸括弧、括弧（開く） | |
|) | 〃（閉じる） | |

| | | |
|--------------------|----------|--------|
| <code><=</code> | 小なりイコール | 2 文字です |
| <code>>=</code> | 大なりイコール | 2 文字です |
| <code>!=</code> | ノットイコール | 2 文字です |
| <code>==</code> | イコールイコール | 2 文字です |

演習 2. プログラミングで使う記号のキーボードでの配置と読みを確認してください。

1.8 プログラムを構成する基礎的な概念

Python によらず広くプログラミング言語を学習する際には以下のようない事項がプログラムを構成する基礎的な概念になります。

- 算術、文字列、論理（真偽）の演算
- 変数、変数への代入、変数の値の評価（代入されている値を使うこと）
- 条件判断によるプログラムの実行の切り替え（分岐）
- プログラムの特定箇所の繰り返し実行
- 定型動作の記述と呼び出し（関数の定義と呼び出し）
- 複合的なデータの取り扱い
- 入出力（端末、GUI、ファイル、ネットワーク）

1.9 プログラムの「どこ」を作るか

現代ではアプリケーションプログラムをすべて自身で作成することはめったにありません。以下の 2 つの中間に位置する部分を用途に応じて作成するのだと理解してください。

- フレームワーク：パーソナルコンピュータ上の GUI を用いるアプリケーションや Web サーバ上のアプリケーションでは、GUI 全体や Web サーバそのものは事前に用意されているものを使います。このようなプログラム全体の大枠をフレームワークと呼びます。
- ライブラリ：他方でプログラムの中で、`sin` や `cos` といった数学関数などのように誰もが部品として使いたいものはライブラリとして用意されているものを使います。

すなわちフレームワーク上で、アプリケーション固有の動きを、ニーズにあったライブラリを活用しながら自分でプログラミングすることになります。



図 8 フレームワークとライブラリ

参考文献

- [1] 喜多 一、岡本雅子、藤岡健史、吉川直人：写経型学習による C 言語プログラミングワークブック、共立出版 (2012)

2. Python の実行環境と使い方

2.1 本章の学習の目標

- Python の統合環境 IDLE を起動できる。
- IDLE 上での Python シェルを操作できる。
- IDLE で Python のプログラム（スクリプト）を編集するエディタを操作できる。
- Python プログラムのエラーを体験する。

2.2 学習環境の想定

本書では京都大学の教育用コンピュータシステムの固定端末での学習を前提に以下の環境での Python の学習を想定しています。

- 基本ソフトウェアとして Windows 10
- Python の配布パッケージとして Python バージョン 3 で構成された Anaconda
- Python の統合開発環境として Anaconda に含まれている IDLE

個人所有の PC についてはそれぞれ Anaconda をインストールしてください。なお、11 章で紹介するする数値計算などのモジュール numpy, matplotlib, pandas を用いない場合は、本家の Python パッケージでもかまいません。

Python の開発環境としては外にも Jupyter Notebook や Spyder などさまざまなものがあります。本書で統合開発環境として IDLE を用いる理由は機能が限定されていて初学者には分かりやすいことと、例として取り上げるタートルグラフィックスを稼働させやすいことが理由です。

2.3 準備

この授業で作成する Python のプログラム（スクリプト）を保存するフォルダを作成してください。例えばドキュメントの下に「Python Scripts」というフォルダを作成します¹。

¹ 教育用コンピュータシステムでは NextCloud というサービスとして稼働させている N: ドライブにフォルダを作成すれば、自宅の PC など外部からもアクセス可能です。

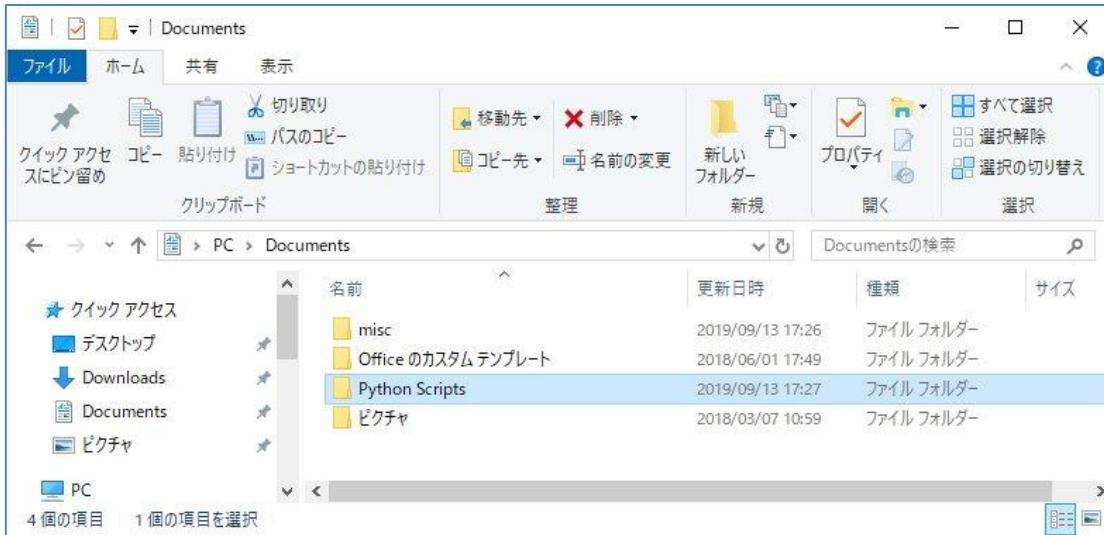


図 9 プログラム保存用フォルダの作成

2.4 IDLE の起動

スタートメニューから Anaconda3 というフォルダの中の Anaconda Prompt を選んでダブルクリックで起動してください。起動したらこのウィンドウ内で `idle`（大文字、小文字は問いません）と入力し ENTER キーを押すことで IDLE が起動します。

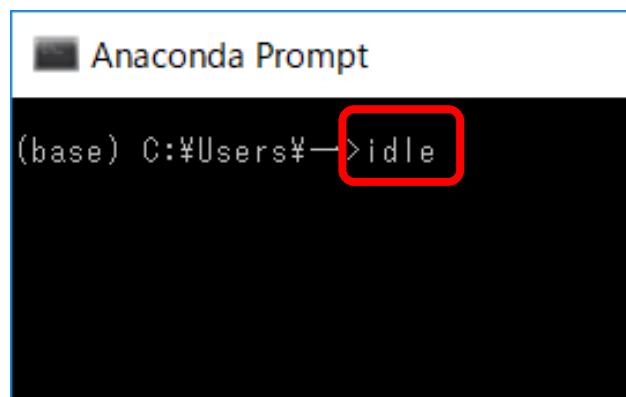


図 10 Anaconda Prompt からの idle の起動

2.5 Python シェル

2.5.1 起動の確認

IDLE を起動すると下図のような Python シェルが現れます。これは Python を対話的に実行する環境です。次の 2 点を確認してください。

- ウィンドウのタイトルの確認。ウィンドウのタイトルに実行している Python のバージョン（ここでは Python 3.6.4 となっています。）が表示されます。複数の Python のバージョンがインストールされている場合、誤ったバージョン（Python バージョン 2 とか）が起動されことがあります。その場合、IDLE の起動方法などを確認してください
- プロンプトの確認。ウィンドウの中の「>>>」が入力を促進する記号（プロンプト）です。これに続けて Python の命令をキーボードから入力できます。

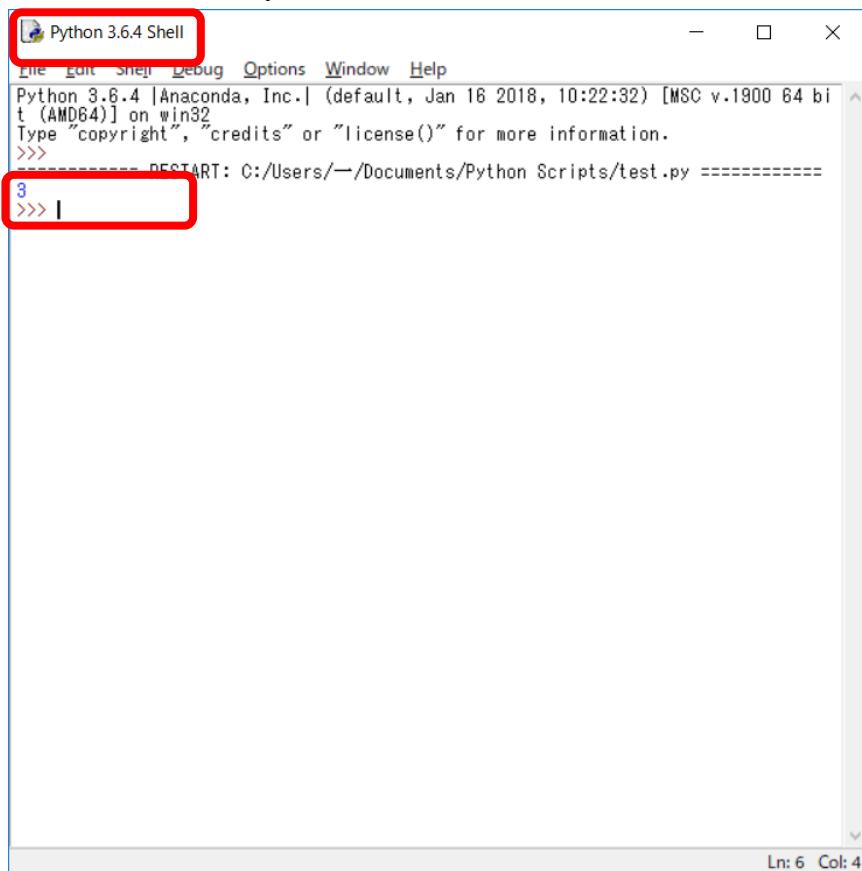


図 11 IDLE の Python シェル

2.5.1 Python の命令の実行

Python シェルのプロンプトに続けて

1+2

と入力し、ENTER キーを押してみてください。以下、**シェルから入力するものは赤字**で書きます。これは「1+2 という計算をする」立派な Python のプログラムです。シェルはこれを実行して

3

と答えてくれるはずです。以下、**実行結果は青字**で書きます。

Python での算術演算は下の表のようになっています。掛け算には「*」、割り算には「/」を用います。数学での計算と同様、掛け算、割り算は足し算、引き算より優先されます。また計算順序を優先するために () が使えます。

表 2 Python の算術演算

| 演算子 | 演算 | 備考 |
|-----|-------|----------------|
| + | 足し算 | |
| - | 引き算 | |
| * | 掛け算 | |
| / | 割り算 | 整数商を求める場合は「//」 |
| ** | べき乗 | 2 文字です。 |
| () | 演算の優先 | 他の括弧は使えません。 |

演習 3. 算術演算の確認

Python シェルで算術演算を練習してください。

次に以下の 2 行を（一行ずつ）入力してみてください。左図参照

a = 1 + 2

a

1 行目は等「=」の左辺の「a」 という変数に右辺の「 1+2 」という式の計算結果を代入しなさい、という命令です。この行の実行ではシェルは何も表示せず次の入力を要求します。

2 行目の実行で変数 a の値を確認しています。

```
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a = 1 + 2
>>> a
3
>>> print(a)
3
>>> |
```

図 12 Python Shell での操作

3

という結果が表示されているはずです。次に

`print(a)`

という命令を入れてみてください。`print()` は () 内の式を文字としてシェルに出力する関数です。やはり

3

と表示されます。

2.6 スクリプトの作成と実行

次に複数行の Python の命令を書いて、まとめて実行する方法を学びます。このためには命令を編集する IDLE Editor を使います。

2.6.1 新規ファイルの作成

新しいプログラムを作成するのでシェルウィンドウの「File」メニューから「New File」を選択します。IDLE Editor が起動されます。

2.6.2 IDLE Editor の確認

IDLE Editor と Python Shell はよく似ています。間違わないように以下の 3 点を確認してください。

- Window のタイトルは編集しているファイル名になります。New File を選んだ場合は「Untitled」になっています。
- Window のメニューは Python Shell と異なっています。「run」というメニューあることを確認してください。
- Window 内は空白です。シェルのプロンプト「>>>」は表示されません。
- Windows の右下にカーソルのある行と列が表示されます。

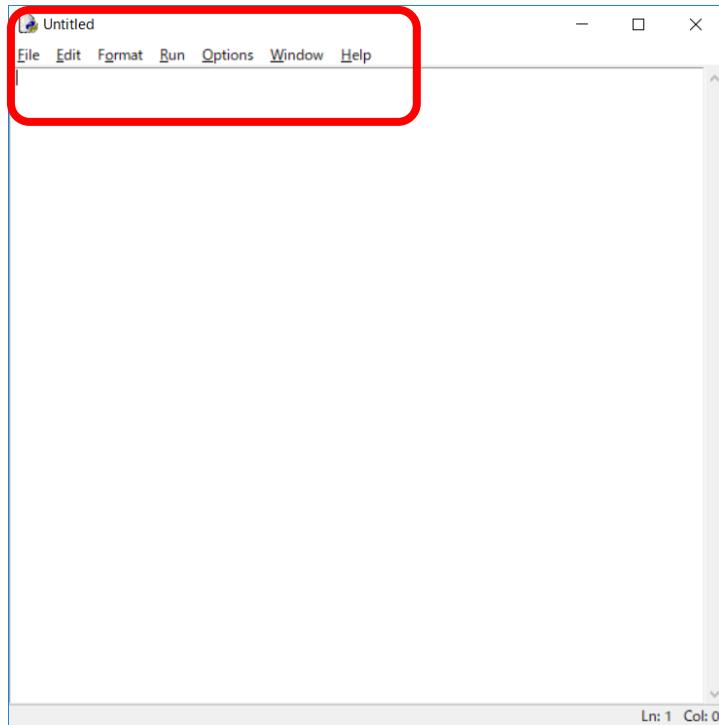


図 13 IDLE Editor, Python シェルとの違いに注意

2.6.3 Python プログラムの記述、保存、実行

2行だけのプログラムですが IDLE エディタで下の表の黄色の部分を入力してください。

プログラム 1 (ex0.py)

| 行 | ソースコード | 備考 |
|---|-----------|-------------------------------|
| 1 | a = 1 + 2 | 右辺 $1+2$ の計算結果を左辺の変数 a に代入する。 |
| 2 | print(a) | 変数 a の値を画面に出力する。 |

タイプミスがないことを確認して、メニューの「Run」から「Run Module」を選びます。新規のプログラムなので保存する必要があります。Python のプログラムを置くフォルダに ex0.py (.py は Python プログラムの拡張子)という名称で保存を指示すると、フォルダにプログラムを保存した後、Python Shell 上で実行され、実行結果が表示されます。

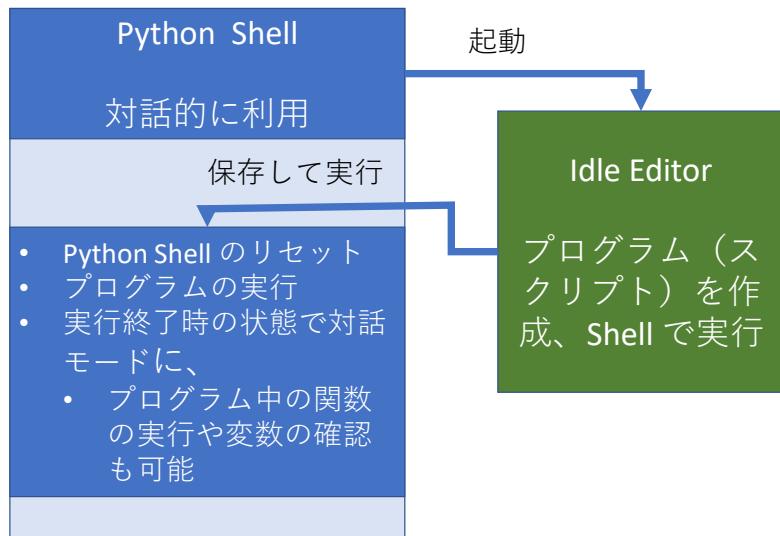


図 14 IDLE での Shell とエディタの連携

Idle Editor から実行を指示すると、プログラムがファイルに保存され、Python Shell がリセットされてプログラムが実行されます¹。実行が終了すると Python Shell は終了した状態でキーボードから入力を受け付ける対話モードになります。これにより、プログラム中で使った変数の値を確認したり、プログラム中の関数を呼び出したりすることができます。

演習 4. `ex0.py` の実行が終了した時点で以下のコマンドを実行して変数 `a` の値を確認してください。

```
print(a)
```

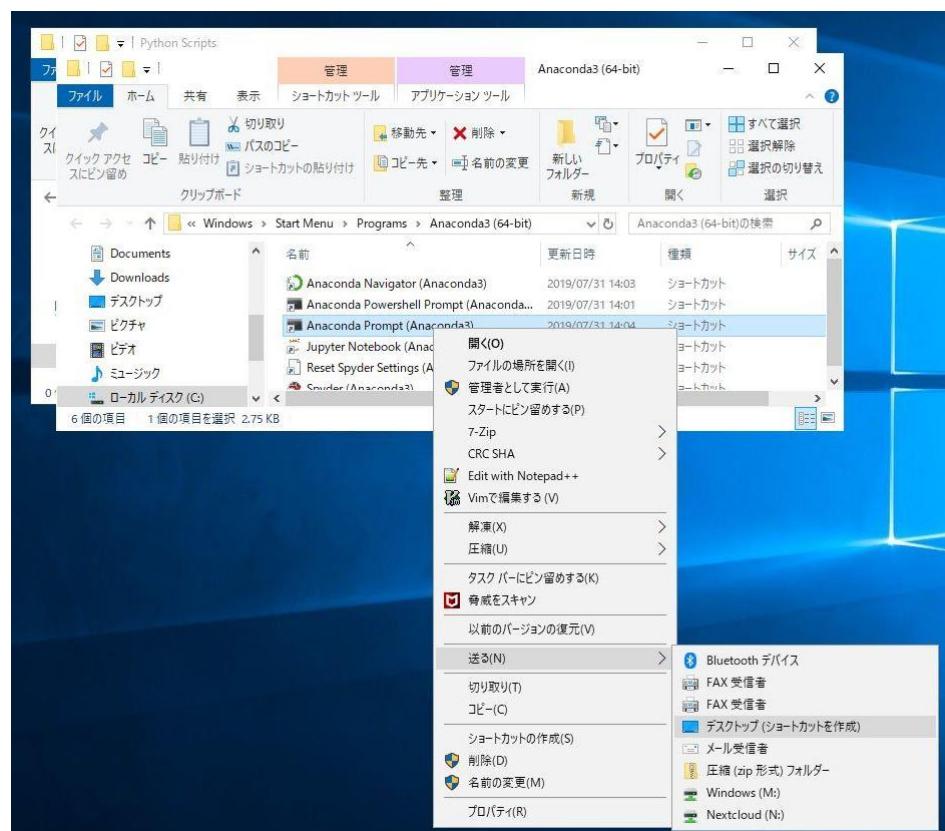
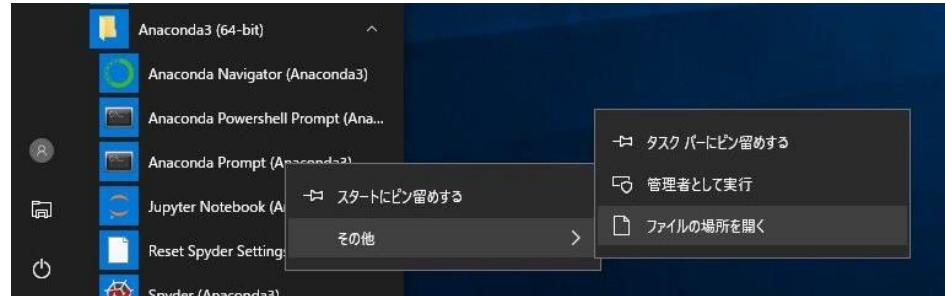
2.7 Anaconda Prompt での作業フォルダの設定

Python のプログラムを入れるフォルダを作成したのでこのフォルダを最初に開くように設定します。IDLE では作業フォルダを指定することができないので IDLE を起動する Anaconda Prompt の作業フォルダを指定することにします。以下の手順で設定してください。

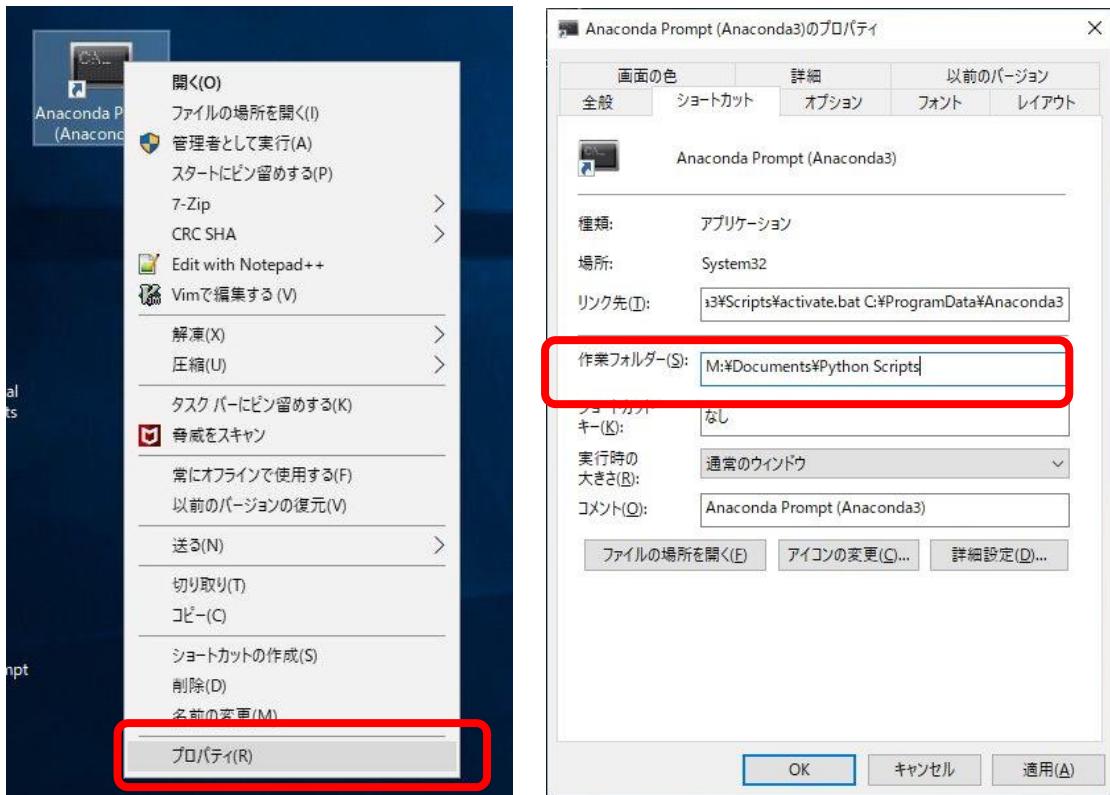
1. デスクトップに Anaconda Prompt のショートカットを作成
 1. スタートメニュー Anaconda Prompt で右クリック
 2. 「その他」 → 「ファイルの場所を開く」を選択

¹ Shell のリセットが必要なのは、それまでに対話的に使用してきた変数などが残されているので、その影響をなくすためです。

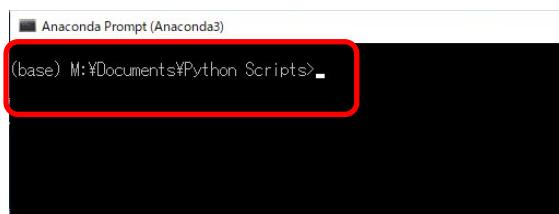
3. エクスプローラの Anaconda Prompt アイコンで右クリック
4. 「送る」→「デスクトップ（ショートカットの作成）」を選択。



2. デスクトップ上の Anaconda Prompt ショートカットに作業フォルダを設定
 1. デスクトップの Anaconda Prompt アイコンを右クリック
 2. 「プロパティ」を選択
 3. 「作業フォルダ」に Python のスクリプトを置くフォルダを設定
 4. 「OK」ボタンをクリック



以後、デスクトップのアイコンをダブルクリックすれば Anaconda Prompt や idle が設定された作業フォルダで動きます。



2.8 IDLE のキー操作など

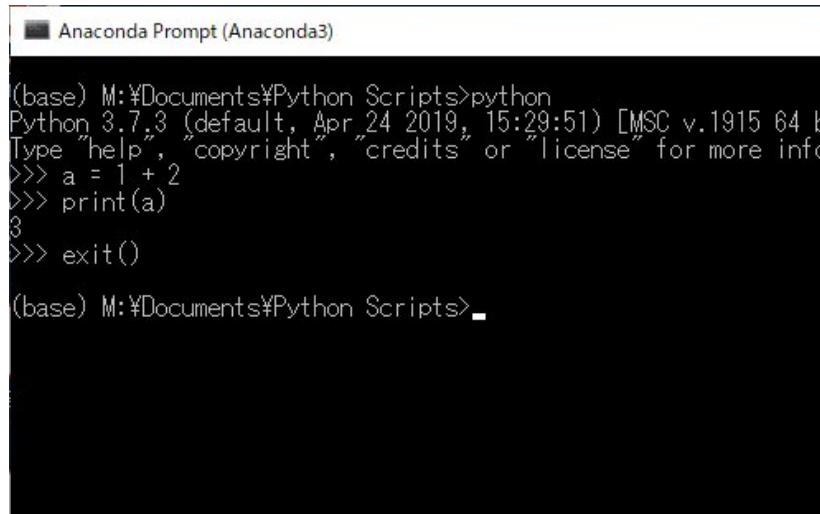
IDLE の Python Shell や IDLE Editor はシンプルなものですですが、いくつか便利なキー操作が設定されています。IDLE のオンラインマニュアルでも読めますが、よく使うものを 15 章の「IDLE Python 便利帳」にまとめておきました。

2.9 Python コマンドの実行

Python のプログラムは Anaconda Prompt 上で直接、実行することも可能です。以下のことを確認してください。

- すでに開いている IDLE や Anaconda Prompt は一旦、終了してください。

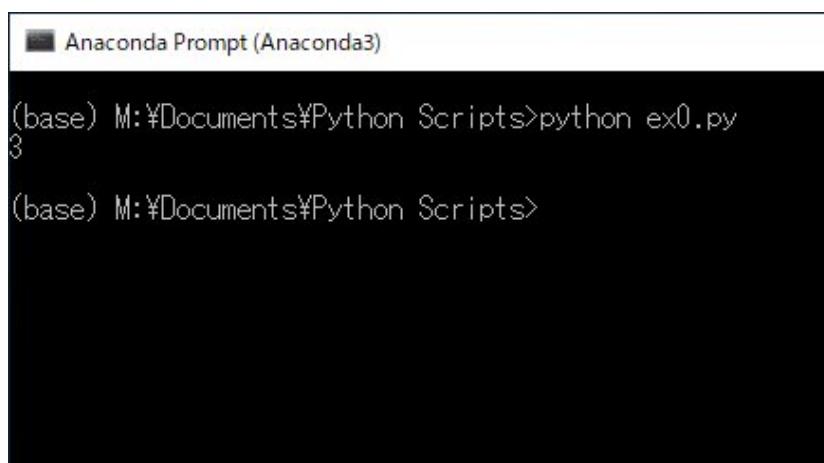
2. 前節で示した作業用フォルダの設定を行います。
3. デスクトップ上のショートカットから Anaconda Prompt を起動します。
4. cd と入力して ENTER キーを押し、作業用フォルダが正しく設定されていることを確認してください。
5. dir と入力し、ENTER キーを押すとフォルダにあるファイルの一覧が表示されます。ex0.py が含まれていることを確認してください。
6. python と入力すると anaonda prompt で python シェルが起動することを確認してください。今は起動を確認するだけでよいので exit() と入力するか、CTRL キーを押しながら C を入力(Ctrl-C)してシェルを終了してください。
7. python ex0.py と Python プログラム（スクリプト）名を指定して python コマンドを実行するとプログラムが実行されます。実行結果を確認してください。
8. python -i ex0.py と -i オプションを指定するとプログラム実行後に対話モードを継続します（IDLE での実行と同様）



```
(base) M:¥Documents¥Python Scripts>python
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit]
Type "help", "copyright", "credits" or "license" for more information
>>> a = 1 + 2
>>> print(a)
3
>>> exit()

(base) M:¥Documents¥Python Scripts>
```

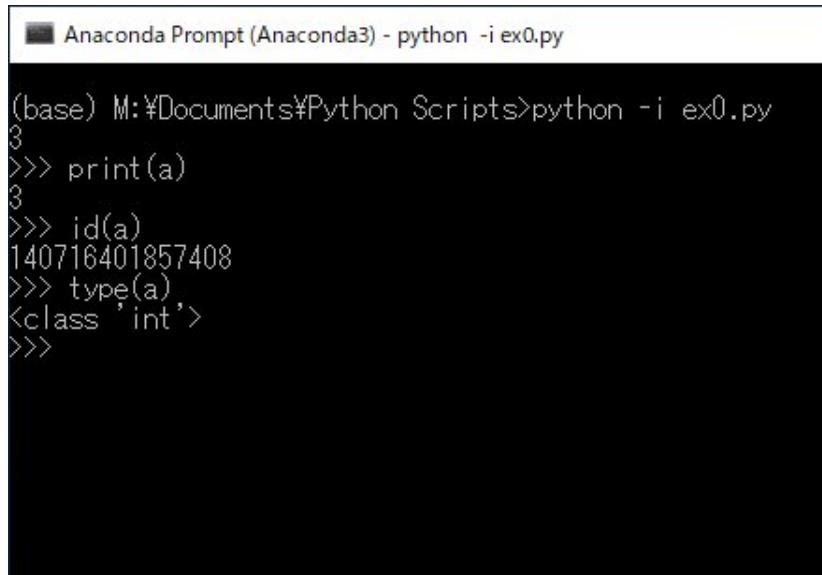
図 15 対話モードでの Python の起動



```
(base) M:¥Documents¥Python Scripts>python ex0.py
3

(base) M:¥Documents¥Python Scripts>
```

図 16 スクリプトを指定した Python の実行



The screenshot shows a terminal window titled "Anaconda Prompt (Anaconda3) - python -i ex0.py". The command "python -i ex0.py" is entered at the prompt. The script "ex0.py" contains three lines of code: "print(a)", "id(a)", and "type(a)". The output shows the value 3 for print(a), the memory address 140716401857408 for id(a), and <class 'int'> for type(a). The prompt then returns to the user.

```
(base) M:¥Documents¥Python Scripts>python -i ex0.py
3
>>> print(a)
3
>>> id(a)
140716401857408
>>> type(a)
<class 'int'>
>>>
```

図 17 -i オプションでスクリプト実行後に対話モードを継続

2.10 Python を学ぶ環境づくり

- Python の処理系：自分の PC に Anaconda をインストールする。
- Python のリファレンスマニュアルの確認
- Python の本（自分に合いそうなものを、1冊は手元に）
- 英語の辞書（資料の確認、変数や関数の命名）
- ノート、筆記具（PC 上のツールでも可）：気づいたことを書き留める

演習 5. あなた自身が Python を学ぶ環境を用意し、報告してください。

演習 6. 本日の演習内容をご自身の学習環境で再確認してください。

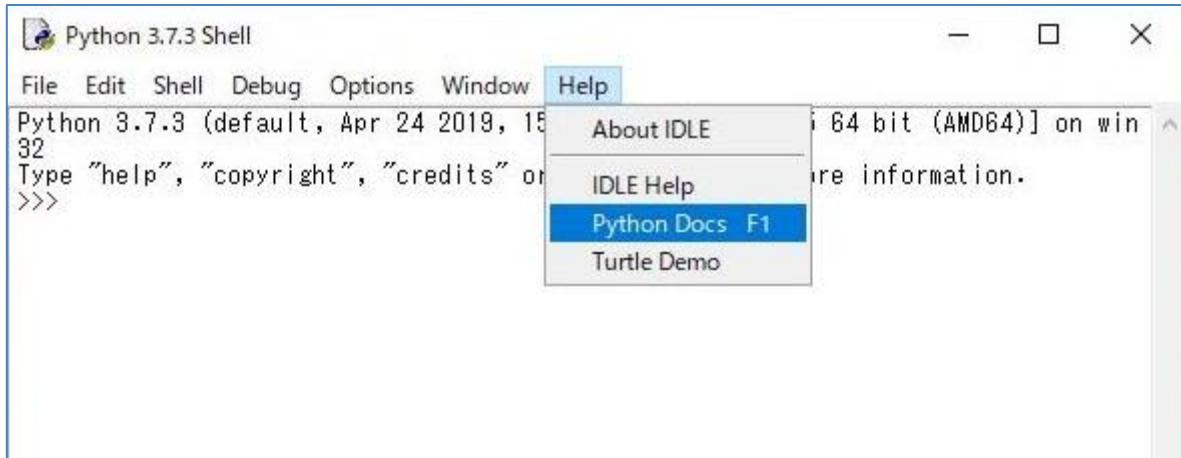


図 18 IDLE からオンラインマニュアルの起動

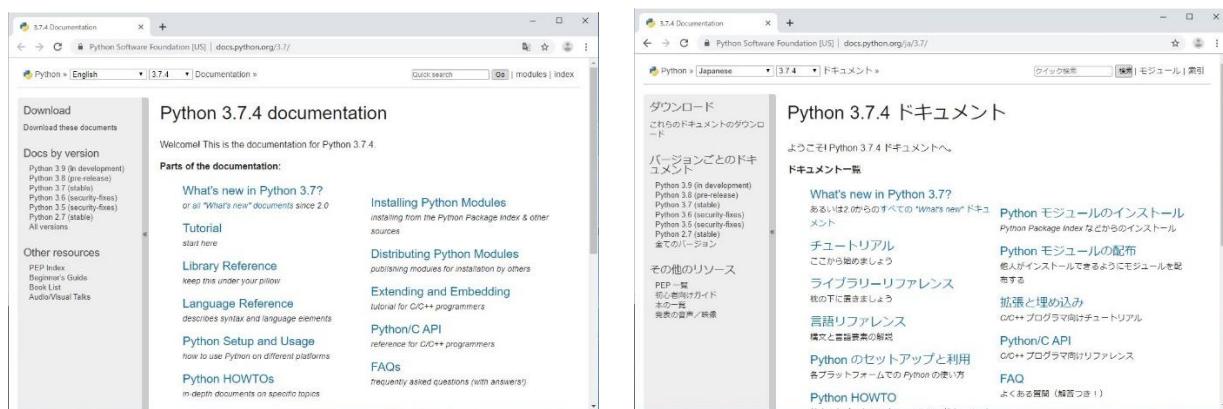


図 19 Python のオンラインマニュアル（右は日本語を指定した場合）

2.11 Mac ユーザへ

本書では Windows 環境での Python の利用を説明しますが、Mac での利用にはいくつか課題があります。以下を参考にしてください。

2.11.1 Mac での Python の導入と IDLE の起動

1) Anaconda のインストール

以下サイトで画面中央の「Mac OS」をクリックし、Python 3.7 version (左側) の「Download」ボタンを押す。

<https://www.anaconda.com/distribution/>

ダウンロードしたパッケージファイルを実行し、インストールする。

2) IDLE の起動

Windows のコマンドプロンプトにあたるのはターミナルです。IDLE はターミナルから起動します。

Finder の「アプリケーション」→「ユーティリティ」→「ターミナル」で起動できます。

Anaconda をインストールした状態であれば、「idle3」とキーボードで入力しエンターキーを押せば Python 3 用の IDLE を起動できます。(※IDLE 起動時に Mac の画面が落ちた場合は下記注参照)

3) IDLE での日本語入力

日本語入力をする場合、「変換中の文字が直接画面に表示されない」問題がありますが、変換候補は選べ、確定も可能です（ただし、確定にリターンキーを一回多く押す必要）。日本語については、他のテキストエディタからのコピー＆ペーストすることも可能です。

コメント等の動作に影響しない部分の文章については、最初から日本語を使わないので一つの手段です。

4) IDLE でのバックスラッシュの入力

この教科書では 11, 12 章でプログラム中に円記号「¥」を入力する箇所がありますが、これは Windows を想定して書いたもので Mac ではバックスラッシュ「\」を入力してください。

バックスラッシュ記号 「\」の入力方法ですが、Mac では 左下の 「option」 キーを押しながら 「¥」 キーを押すと入力できます。

5) IDLE の終了

idle を終了したら、ターミナルのウィンドウを閉じても大丈夫です。

(注：IDLE 起動で Mac の画面が落ちた時)

IDLE は tk というグラフィクスパッケージを使っています。mac では tk のバージョンによって正しく動作しないことがあるようです。以下の手順でバージョン 8.6.7 に設定変更してご利用ください。

上記手順でターミナルを起動した後、

- ターミナル上で conda install tk=8.6.7 と入力してリターン
- しばらく時間がかかるので待つ
- Proceed ([y]/n)? と表示されるので、 y と入力してリターン

上記の手順を実行した後、ターミナルから idle3 を起動すると問題は解消されます。

2.11.2 Mac での Tkinter の問題

上記のほか GUI 環境の Tkinter は Tcl/Tk というパッケージを利用しておおり、これに伴う問題や、グラフ描画モジュール matplotlib での日本語フォントの取り扱いなど OS に依存する部分の操作の Windows との違いがあります。。

参考文献

Python に関する書籍は近年、数多く出版されていて、どれを買っていいのか迷うかと思います。以下、いくつか挙げておきます。文献[2]～[6]は Python の入門書です。文献[7]は名称からは分かりにくいですが Python を解説しながら独学でプログラマーになった著者の経験なども紹介しています。文献[8]はプログラミングの周辺で入門書にはあまり書かれない事項を解説しています。文献[9]、[10]は応用を意識した本です。文献[11]、[12]は本書で少し触れる数値計算など学術利用関連のライブラリである NumPy, matplotlib, pandas の解説書です。このほかの書籍を図書館などで探される際には Python Version 3 を扱っていることを確認してください。

- [2] Bill Lubanovic 著、斎藤 康毅 監訳、長尾 高弘 訳：入門 Python3、オライリー・ジャパン (2015)
- [3] 柴田淳：みんなの Python 第4版、SB クリエイティブ (2017)
- [4] 大津真：基礎 Python、インプレス (2016)
- [5] 松浦健一郎、司ゆき：はじめての Python エンジニア入門編、秀和システム (2019)
- [6] 大澤文孝：いちばんやさしい Python 入門教室、ソーテック社 (2017)
- [7] コーリー・アルソフ著、清水川貴之訳：独学プログラマー、日経 BP (2018)
- [8] 増井敏克：基礎からのプログラミングリテラシー、技術評論社 (2019)
- [9] 日経ソフトウェア編：いろいろ作りながら学ぶ！Python 入門、日経 BP (2019)
- [10] Al Sweigart 著、相川愛三訳：退屈なことは Python にやらせよう、オライリー・ジャパン (2017)
- [11] Python によるデータ分析入門、オライリー・ジャパン
- [12] Jake VanderPlas 著、菊池彰訳：Python データサイエンスハンドブック、オライリー・ジャパン (2018)

3. 変数と演算、代入

3.1 本章の学習の目標

- Python のプログラムでの実行流れと情報の流れを理解し、順次実行について知る。
- プログラムでの変数の命名と代入、評価について知る。
- Python の基本的なデータ型を知る。
- データ（オブジェクト）の型を調べる `type()` 関数とオブジェクトの所在を調べる `id()` 関数について知る。

3.2 プログラムの実行の流れと情報の流れ

3.2.1 順次実行

前章の例

```
a = 1 + 2
```

```
print(a)
```

ではプログラムは上から順に 1 行ずつ実行されて行きます。これは「順次実行」といい、プログラムの基本です。このほか、

- 条件によって実行する箇所を切り替える分岐
- 同じ処理の繰り返し
- 関数を呼び出すことで、処理を関数の定義に移すこと

などがあり、これらについては後の章で説明します。Python プログラムのソースコードは「実行の流れ」に沿って書かれています。

演習 7. プログラムと楽譜

コンピュータのプログラムは音楽の楽譜と似た面があります。音楽の楽譜も基本は前から順に音符を演奏することです。このほか、演奏箇所を切り替えたり、繰り返したりする記法があることを確認してみてください。

3.2.2 変数を通じた情報の流れ

一方、プログラムでは情報をプログラムの各ステップで加工して行きますが、情報は変数に代入された数値や文字列として扱われます。このため、「実行の流れ」と比較して「情報の流れ」は同じ変数への代入や参照を通じて行なわれるため「分かりにくい」、ということを意識しておいてください。例えば上の例で 1 行目で設定された変数 `a` の値が 2 行目の `print` 関数で使われています。

では、以下のような例ではどうでしょうか

```
a = 1 + 2  
a = 3 + 4  
print(a)
```

このプログラムでは 1 行目の変数 `a` への代入は、2 行目で同じ変数が上書きされるため、3 行目の `print(a)` に対して無意味であるということを変数 `a` について追いかけることで初めて分かります。

3.3 変数の名前

3.3.1 プログラムでは複数文字の変数名も使う

先の例では変数名として「`a`」を用いました。数学では変数には 1 文字のアルファベット（やギリシャ文字）を用いることは多いですが、さまざまなデータを取り扱うプログラミング言語では変数名としてより長い名前が使えます。例えば

```
a  
x  
x2  
root  
square_root  
などです。
```

3.3.2 変数の命名ルール

以下のルールを覚えておいてください。

- 英大文字、英小文字、数字、アンダースコア(`_`)を使う。
- 大文字と小文字は区別される。

- 数字を先頭に使ってはいけない。
- Python の文法で使用する予約語(例えば if など、IDLE Editor では予約語は赤字で表示されます)は使えない。

日本語（漢字など）の変数名も利用可能ですが、あまり使われていません。

3.3.3 分かりやすい変数名を使う

1) 数学での変数名を例に

適切な名称の使用は思考やコミュニケーションを円滑にします。例えば数学でも一次関数を

$$y = ax + b$$

と書けば、y は x の一次関数で傾きが a 、切片が b だとすぐに分かります。これは x や y を変数に、y は x の関数として、そして a や b はパラメータとして使うという慣習があるからです。

$$a = xb + y$$

は a と b を x と y に入れ替えただけの式ですが、なんだか急に分かりにくくなります。

2) Python プログラムでの変数名の付け方

プログラムでも変数の命名はプログラムを分かりやすくする重要なものです。以下のようなことに心がけるとよいでしょう。

- プログラムでの意味を表す命名をしましょう。¹
- 1 文字などの短い変数名はできるだけ狭い範囲だけで有効な変数として使いましょう。特に l, o, O （小文字のエル、小文字のオー、大文字のオー）などは数字と紛らわしいため使用しないようにします。
- 大文字ではなく、小文字を使いましょう。大文字は値を変化させない定数を表すことに使われることが多いです。
- 複数語の変数名は単語の間をアンダースコア「_」で区切れます。
- できれば英語を使う。プログラムは当初の意図とはズれて、想定外に成長し多くの方に使われる場合があります。海外の方に使われる場合もあるので、予め

¹ 「命名する」という行為は日常生活ではありません、子供やペットに名前をつけるぐらいでしょう。しかし、コンピュータ（情報）を利用する場合にはファイルやフォルダ名など、命名するという行為が大変重要になります。プログラミングはそれが最もよく現れる行為の一つだと言えます。「命名する」というスキルが必要だということを意識してみてはいかがでしょうか。

英語で命名しておくといいでしよう。¹

変数名以外にも、どのようにプログラムを書くと分かりやすいかは重要な点です。Python では PEP8[13] というプログラムを書く際のコーディング規約が提唱されています。

演習 8. さまざまな変数名を利用する練習。

ex0.py で示したプログラムについてシェル上の実行で構いませんから、変数名をさまざまに代えて練習してみてください。複数語をアンダースコアで接続した変数名も試すこと。また、先頭に数字を用いた場合、予約語を用いた場合などにどのようなエラーが生じるかも併せて試みてください。

3.4 変数への代入と値の評価

以下のプログラムを Python シェルで実行してみてください。

```
a = 1
print(a)
a = a + 1
print(a)
```

1 行目では変数 a に 1 を代入しています。

3 行目は読み方に注意が必要です。このプログラムについて、Python では

- まず代入演算子(=)の右辺を計算します。

- すでに変数 a については 1 が代入されていますので、右辺は「a の値を評価した結果」の 1 を使って、1+1 となり、2 という計算結果が得られます。

- つぎにこの結果が左辺の変数 a に代入（上書き）されます。

変数については「名前のついた箱」というイメージで理解するとよいでしょう。

¹ 以前行っていた研究でドイツの先生が書いた FORTRAN のプログラムを C に書き直したことがあります。古い仕様の FORTRAN は変数名の長さが制限されていて、これがドイツ語の単語を短くしたものであったため、まったく意味が分からませんでした。その先生のプログラムは丁寧な英語での注釈がついていたので、なんとか目的を果たすことができました。

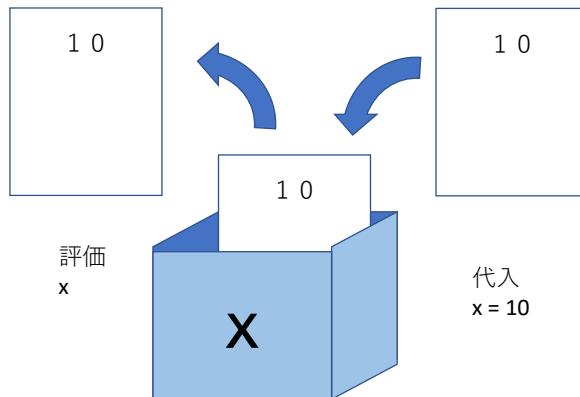


図 20 変数についてのイメージ

演習 9. 変数の動作の説明

以下は 1000 円の商品の 15% 引きを計算するプログラムです。

- このプログラムには 1箇所誤りがあり、実行するとエラーになります。どのような誤りがあるかを説明してください。
- 誤りを修正したうえでプログラムの動作を説明してください。

```
kakaku = 1000
nebikiritsu= 15
kakaku = Kakaku*(100-nebikiritsu)/100
print(kakaku)
```

3.5 代入演算子

プログラムではある変数に一定数（例えば 1）を加える、一定数を差し引く、という計算をしばしば行います。このような計算を便利に行うため、代入演算子として “=” 以外に以下のようないものも利用可能になっています。

表 3 Python の代入演算子

| 演算子 | 例 | 意味 |
|-----------------|---------------------|-------------|
| <code>+=</code> | <code>a += b</code> | $a = a + b$ |
| <code>-=</code> | <code>a -= b</code> | $a = a - b$ |
| <code>*=</code> | <code>a *= b</code> | $a = a * b$ |

| | | |
|----|--------|---------|
| /= | a /= b | a = a/b |
|----|--------|---------|

なお、C 言語などでよく使われる「++」や「--」という演算子は Python にはありません。

3.6 Python で使えるデータ型

先の例ではデータとして整数を扱いましたが、このほか基本的なデータの型として Python では下の表のように小数点以下の値も扱える浮動小数点数、文字列、論理値(真と偽)などがあります。Python の特徴として、整数の桁数に制限を設けていない(コンピュータのメモリ容量や計算速度の問題はありますが、)ということがあります。例えば

2**100

は

1606938044258990275541962092341162602522202993782792835301376

と計算してくれます。

数値型としてこのほか、複素数も扱えます。

表 4 Python で使えるデータ型

| 型 変換のための 関数 | 説明 | 定数 (リテラル) の表記例 | 備考 |
|-------------------|-------------------|---------------------|---|
| 整数 int() | | 12345 | Python では桁数に制限はありません |
| 浮動小数点数 float() | 小数点 以下を 含む数 | 1.0 2.99792458E8 | 大きさ(有効数字の桁数と表現できる範囲)に制限があります。E8 という表記は×10 ⁸ という意味です。 |
| 文字列 str() | 文字の 並び | 'aaa' “日本語” | シングルクオートかダブルクオートで 文字列を囲みます |

| | | | |
|----------------------------|---------------|---------------|-------------|
| 論理値 bool() ¹ | 条件判断に使 います | True False | 定数は先頭が大文字です |
|----------------------------|---------------|---------------|-------------|

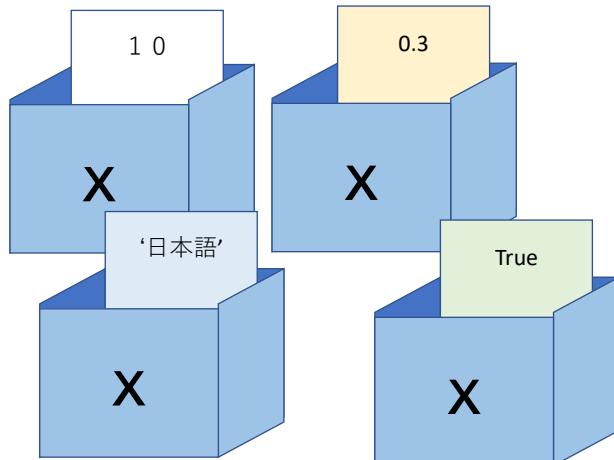
なお、Python を含む多くのプログラミング言語で、浮動小数点数の演算は 2 進法で行なわれます。我々は小数も 10 進法で扱っているのですが、 $1/3$ は 10 進法の小数では正しく表現できないのと同じ現象として $1/10$ が 2 進法の浮動小数点数では正しく表現できません。詳しくはコラム「float って」を参照ください。

演習 10. Python シェルで以下を実行してください。

```
a = 1
b = 1/2
c = "ABC"
print(a)
print(b)
print(c)
print(type(a))
print(type(b))
print(type(c))
```

Python ではデータ（一般にオブジェクトと呼びます）には、「型」がありますが、変数にはどのような型のデータでも代入することが可能です。

変数に代入されているオブジェクトの型を知るには `type()` 関数を使います。



¹ 論理演算の代数としての理論を考案した George Boole にちなんでいます。

図 21 Python では変数の内容の型は自由である

3.7 Python の変数のより正しい理解

実際には Python では変数が直接、データ（オブジェクト）を持っているのではなく、「オブジェクトがどこにあるか」という所在の情報（参照）を持っています。今の時点でこのことをあまり意識する必要はありませんが、後により複雑なリストなどのデータの扱いを正しく理解する上では重要になります。

変数がもつデータの所在を特定する情報は `id()` という関数で調べることができます。

```
a = 1  
b = 2  
print(id(a), id(b))
```

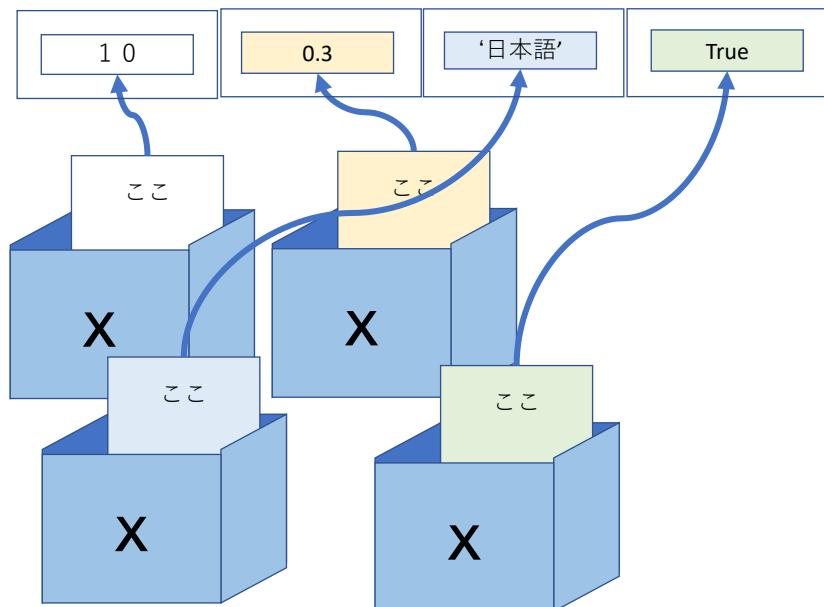


図 22 Python の変数はデータ（オブジェクト）の所在情報を持つ

3.8 例題：平方根を求める

順次実行と変数の扱いを使った例題として与えられた数値の平方根の近似値を求めてみましょう。以下の計算方法は単純ですが、桁数の多い割り算を使います。このため手で計算するのは面倒ですがコンピュータを使うと簡単に実現できます。

3.8.1 計算手順

1. 平方根を求める数値を $x (> 0)$ とします。
2. 平方根の近似値の初期値 $r_{\text{new}} (> 0)$ を定めます。ここでは初期値として x と同じ値を設定します。
3. r_1 に r_{new} の値を代入します。
4. もう一つの近似値として $r_2 = x/r_1$ を考えます。
 r_1 が x の平方根であれば r_2 もまた x の平方根になりますが、異なっていれば、どちらかが真の値より大きく、どちらかは真の値よりは小さくなり、真の値は r_1 と r_2 の間にあります。
5. そこで新しい近似値として r_1 と r_2 の平均 $r_{\text{new}} = (r_1 + r_2)/2$ を考えます。
6. ステップ 3. ~ 5. を適当な回数繰り返します。

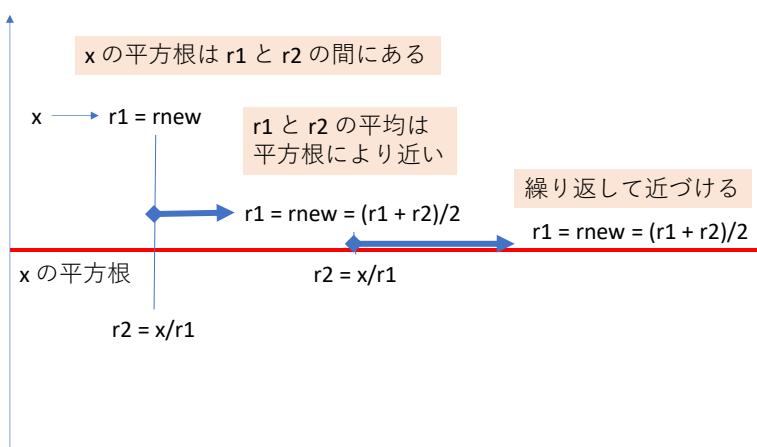


図 23 平方根の近似計算の直感的説明

3.8.2 Python プログラム

演習 11. 次の表のソースコードの部分を IDLE エディタで入力し, `ex1.py` という名で保存して実行してみてください。

プログラム 2 平方根を求めるプログラム（その 1、ex1.py）

| 行 | ソースコード | 説明 |
|----|---------------------|-------------------------|
| 1 | # x の平方根を求める | # で始まる部分は注釈 |
| 2 | x = 2 | |
| 3 | # | |
| 4 | rnew = x | 最初の近似値の想定 |
| 5 | # | |
| 6 | r1 = rnew | |
| 7 | r2 = x/r1 | |
| 8 | rnew = (r1 + r2)/2 | |
| 9 | print(r1, rnew, r2) | |
| 10 | # | 以下は赤字の部分を 3 回繰り返しているだけ。 |
| 11 | r1 = rnew | |
| 12 | r2 = x/r1 | |
| 13 | rnew = (r1 + r2)/2 | |
| 14 | print(r1, rnew, r2) | |
| 15 | # | |
| 16 | r1 = rnew | |
| 17 | r2 = x/r1 | |
| 18 | rnew = (r1 + r2)/2 | |
| 19 | print(r1, rnew, r2) | |
| 20 | # | |
| 21 | r1 = rnew | |
| 22 | r2 = x/r1 | |
| 23 | rnew = (r1 + r2)/2 | |
| 24 | print(r1, rnew, r2) | |

以下のような結果が得られれば成功です。

```
===== RESTART: C:/Users/一/Documents/Python Scripts/ex1.py =====
2 1.5 1.0
1.5 1.4166666666666665 1.3333333333333333
1.4166666666666665 1.4142156862745097 1.411764705882353
```

| | | |
|------------------------------------|------------------------------------|----------------------------------|
| 1.4142156862745097 | 1.4142135623746899 | 1.41421143847487 |
|------------------------------------|------------------------------------|----------------------------------|

4回の繰り返しで近似値として 1.4142135623746899 が得られました、別途、

$2^{**}(1/2)$

を求めてみると

[1.4142135623730951](#)

が得られます。小数点以下 11 桁目まで正しいことが分かります。

上では計算方法について直感的な説明を行いましたが、これはニュートン法と呼ばれる数値計算手法の平方根を求める場合への適用例です。詳しくはコラムの「ニュートン法」を参照ください。

演習 12. 他の数値の平方根を求める。

1. ex1.py を変更して、他の正の数値の平方根を求めてください。
2. また、このプログラムで 0 の平方根を求めようとすると何が生じるか確認してください。単にエラーのメッセージを見るだけでなく、実際にプログラムをご自身で追いかけて（トレースすると言います）、どこで問題が生じるかを考えてください。

3.9 読み易い式の表記

プログラム ex1.py の 8 行目

`rnew = (r1 + r2)/2`

では、代入演算子 `=`、足し算 `+`、割り算 `/`、優先順位を変える `()` が使われていますが、式を読みやすくするために `=` と `+` の前後には空白を入れています。他方で `()` の内側や `/` の前後には空白を入れていません。試しにすべての空白を抜くと

`rnew=(r1+r2)/2`

とかなり詰まった感じになり、読みにくくなります。優先順位の高い演算である `*` や `/` の前後はつめ、低い演算である `+` や `-`, `=` の前後を開けるようにするとよいでしょう。

参考文献

- [13] PEP 8 -- Style Guide for Python Code, <https://www.python.org/dev/peps/pep-0008/>
(2020/2/12 アクセス)

4. 制御構造

4.1 本章の学習の目標

本章ではプログラムの実行を制御する以下の方法を学びます。

1. for 文, while 文による繰り返し処理と range() 関数
2. if 文による分岐
3. try 文によるエラー処理

また、これに関連して以下も学びます

4. 条件式の書き方
5. input() 関数によるキーボードからの入力
6. Python の数学関数
7. 文字出力におけるフォーマット指定

4.2 for 文と range() 関数を用いた一定回数の繰り返し

プログラミングによりコンピュータが威力を発揮するのは多くの処理を速い速度で実施できることですが、順次実行では実行するステップ数だけプログラムを書かなければなりません。前章での平方根を求めるプログラムでは全く同じ記述が 4 回繰り返されています。ここでは、for 文と range() 関数を用いて繰り返しを自動化することを学びます。

演習 13. 次の表に示すプログラムを作成し、実行してください。

プログラム 3 平方根を求めるプログラム（その 2、ex2.py）

| 行 | ソースコード | 説明 |
|---|---------------------|-----------------|
| 1 | # x の平方根を求める | # で始まる部分は注釈 |
| 2 | x = 2 | |
| 3 | # | |
| 4 | rnew = x | |
| 5 | # | |
| 6 | for i in range(10): | i を 0 から 9 まで変え |

| | | |
|----|---------------------|-------------------------------------|
| 7 | r1 = rnew | ながら以下を繰り返します。6 行目最後はコロン「:」があることに注意。 |
| 8 | r2 = x/r1 | |
| 9 | rnew = (r1 + r2)/2 | |
| 10 | print(r1, rnew, r2) | |

Python では繰り返す範囲（ブロック）を字下げ（推奨は空白 4 文字）します。

IDLE エディタでは領域を選んで、Ctrl キーを押しながら] キーを押す(Ctrl-]と表記します)一括して字下げできます。逆の操作は Ctrl-] です。

4.3 for 文の書き方¹

For という英単語はいろいろな意味がありますが、ここでは「～のために」と考えると分からなくなります。「～について」ぐらいの意味で読み取って下さい。Python の for 文は以下のように構成します。

for 目標変数 in 繰り返しの範囲 :

 繰り返すブロック

上の ex2.py の例では「目標変数」は i で、「繰り返しの範囲」は range(10)、ブロックは（4 文字）字下げされた 7~10 行です。

range(10) という関数は 0 から添え字の値-1 の 9 までの 10 個の値を生成する関数で、for 文は生成された値を変数 i に入れて、ブロックの部分を繰り返します。

この for 文を「日本語に訳読する」なら以下のようにようになるでしょう。

「繰り返しの範囲」の各値をいれた「目標変数」について「繰り返すブロック」を繰り返す。

以下、Python の文法的事項の説明は上の例のように枠で囲って示します。固定的な表現は赤字で、内容によって変化するものは黒字で書きます。

演習 14. ブロックの確認

¹ Python の for 文は後述のリストなどと組み合わせて効果的な書き方が可能ですが、ここでは range() 関数との組み合わせを紹介します。

先の例(ex2.py) の 10 行目を次のように左につめてブロックから外し、動作を確認し説明してください。

プログラム 4 平方根を求めるプログラム（その 2、ex2_2.py）

| 行 | ソースコード | 説明 |
|----|---------------------|---------------|
| 1 | # x の平方根を求める | # で始まる部分は注釈 |
| 2 | x = 2 | |
| 3 | # | |
| 4 | rnew = x | |
| 5 | # | |
| 6 | for i in range(10): | |
| 7 | r1 = rnew | |
| 8 | r2 = x/r1 | |
| 9 | rnew = (r1 + r2)/2 | |
| 10 | print(r1, rnew, r2) | この行をブロックから外す。 |

演習 15. イタズラ

上のプログラム(ex2_2.py)は端末への出力を for 文の繰り返しから外したので、繰り返し部分は高速に実行できます。6 行目の range() 関数の添え字を 10 から 100, 1000, 10000, 100000, 1000000, 10000000 と変化させてどの程度の時間がかかるか試してみてください。¹

4.4 Python でのブロック

複数行のプログラムを一括して扱うブロックはプログラミングでの重要な考え方です。

- Python ではブロックを「同じ深さの字下げ」で表記します。これは Python の特徴の一つです。
- Python のプログラムでは字下げが重要な意味をもつため、空白 4 文字など統一した記法が望まれます。

¹ 現代のパーソナルコンピュータは GHz 程度のクロックで動いており、1 命令の実行を 1 クロックで行うなどとても高速な処理を行います。しかしながら、中間コード方式で実行される Python はプログラミング言語の中では処理が遅いものです。

- IDLE Editor では for 文などブロックを伴う行を入力すると続く行を自動的に字下げしてくれます。
- 字下げに全角の空白を入れてしまうとエラーになります。また TAB キーはエディタの設定にもありますが、TAB コードがそのまま入力される場合もエラーになります。
- 他の言語では例えば C ではブロックは {} で囲みます。他の言語に慣れた人は記法に注意が必要です。

4.5 for 文内の処理の制御

for 文内の処理を打ち切ったり、特定の繰り返しでは処理をスキップしたりするために break と continue という命令が用意されています。

- break: for 文の繰り返しから脱出します。
- continue: for 文の繰り返しのブロックの残りの部分をスキップして次の繰り返しに移ります。

これらは、後述の if 文での条件分岐と組み合わせて用いられます。

プログラム 5 continue と break

| 行 | ソースコード | 備考 |
|---|---------------------|------------------|
| 1 | for i in range(10): | |
| 2 | if i == 1: | i が 1 なら次に移る |
| 3 | continue | |
| 4 | if i== 8: | i が 8 なら繰り返しから脱出 |
| 5 | break | |
| 6 | print(i) | |

このプログラムの実行結果は以下のようになります。

```
===== RESTART: C:/Users/一/Documents/Python Scripts/test.py =====
0
2
3
4
5
```

6

7

演習 16. 上記の実行結果についてソースコード用いて説明しなさい。

4.6 range() 関数

正確には `range` は関数ではなく「クラス」として実装されているのですが、関数のような使い方が主ですので、ここでは簡単のため `range()` 関数と呼びます。`range()` は一定間隔での数値の並びを生成してくれます。ただし、実際にどのような値が生成されるかは `range()` 関数の呼び出しだけでは分からないので、生成される値からリスト（データの並びを持つ構造）を生成して確認します。Python シェルで

`list(range(10))`

と入力すると

`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

が得られます。

`range()` 関数の使い方としては引数の与え方で以下の 3 通りがあります。

- 終了値を与える。先に見たように

`range(終了値)`

といった使い方です。0 で始まり、終了値より手前の整数で終わります。

プログラミング言語になじみのない方は 0 で始まることや終了値が含まれないことを奇異に思われるかもしれません。これについてはコラム「0 始まり」も参照ください。生成される個数は終了値と一致するので実際にはそれほど分かりにくいものではありません。このような範囲の取り扱いは Python では標準的に行われます。

- 開始値と終了値の 2 つを与える。

`range(開始値, 終了値)`

開始値は含まれ、終了値は含まれません。2 つの引数の間にはカンマ「,」を入れてください。カンマの後に（半角の）スペースを入れると読みやすくなります。

- 開始値と終了値とステップ幅の 3 つを与える。

`range(開始値, 終了値, ステップ幅)`

演習 17. `range()` 関数

上で述べたように `list()` と組み合わせて `range()` 関数の 3 通りの使い方を

Python シェルで練習してください。

4.7 for 文の入れ子

行と列のように 2 方向に広がる表の各要素の生成は for 文のブロック中にさらに for 文を書くことで実現できます。例えば

プログラム 6 for 文の入れ子

| 行 | ソースコード | 説明 |
|---|--------------------|----|
| 1 | for i in range(3): | |
| 2 | for j in range(3): | |
| 3 | print(i,j) | |

を実行すると以下のように出力されます

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

外側の for 文と内側の for 文で「目標変数」を変えていることに注意してください。

演習 18. 2 行目の range() 関数の引数に変数 i を使う (range(i)とする) とどうなるか、試してみてください。

ここでは詳しくは説明しませんが for 文は range() 関数の代わりにリストを与えるとリストの要素を順に見てくれます。以下の例を試してみてください。半角と全角文字が混じるので注意してください。地名以外は空白を含めて半角文字であることに注意してください。

プログラム 7

| 行 | ソースコード | 説明 |
|---|-------------------------------|--------------|
| 1 | for i in ["三条", "四条", "五条"]: | 全角文字は赤字の部分だけ |
| 2 | for j in ["河原町", "烏丸", "堀川"]: | |
| 3 | cross = i+j | |
| 4 | print(cross) | |

結果は以下のようになります¹

三条河原町

三条烏丸

三条堀川

四条河原町

四条烏丸

四条堀川

五条河原町

五条烏丸

五条堀川

4.8 while 文による繰り返し

4.8.1 精度を指定した平方根の計算

平方根の計算を一定の精度を要求して計算するようにしましょう。r1 と r2 は真の値を挟んでいますので、その差の絶対値は計算の精度として捉えることができます。精度を 10^{-6} 以下として指定してプログラムを構成します。

このような精度は数値が小数点以下何桁かを指定する絶対精度ですが、科学の計算では、有効数字の桁数を指定するほうが使いやすいという面もあります。これについてはコラムの「相対精度」を参照してください。

プログラム 8 平方根を計算するプログラム（その 3、ex3.py）

| 行 | ソースコード | 説明 |
|---|--------|----|
|---|--------|----|

¹ 京都の交差点の名称は「四条河原町」のように東西の通りを南北の通りより先に言う場合と「東山三条」のように後に言う場合があります。ここでは機械的に生成しているので、実際の呼称とは一致しません。

| | | |
|----|------------------------|---------------------------------|
| 1 | # x の平方根を求める | |
| 2 | x = 2 | |
| 3 | # | |
| 4 | rnew = x | |
| 5 | # | 2つの近似値 (x と 1 (=x/x)) の差をとる |
| 6 | diff = rnew - x/rnew | 負なら符号を反転。 |
| 7 | if (diff < 0): | |
| 8 | diff = -diff | |
| 9 | while (diff > 1.0E-6): | 差が 10^{-6} より大きければ |
| 10 | r1 = rnew | 繰り返す |
| 11 | r2 = x/r1 | |
| 12 | rnew = (r1 + r2)/2 | |
| 13 | print(r1, rnew, r2) | 差の再計算 |
| 14 | diff = r1 - r2 | |
| 15 | if (diff < 0): | |
| 16 | diff = -diff | |

演習 19. 上のプログラムを作成して実行してください。

以下のような実行結果が得られるはずです

2 1.5 1.0

1.5 1.4166666666666665 1.3333333333333333
1.4166666666666665 1.4142156862745097 1.411764705882353
1.4142156862745097 1.4142135623746899 1.41421143847487
1.4142135623746899 1.414213562373095 1.4142135623715002

4.8.2 while 文の構成

条件が成立している間、ブロック内の処理を繰り返す while 文の書式は以下のようになっています。

while 条件式:

 実行ブロック

条件は実行ブロックに入る「前」にチェックされます。このため先の例では diff の値を while 文に入る前と実行ブロック内の両方で計算しています。条件式の詳しい

書き方についてはこの先で紹介します。

また for 文と同様、繰り返しを脱出する break や次の繰り返しに移る continue が使えます。

4.8.3 無限ループ⁹

while 文の使い方として以下のような例に出会うことも少なくありません。

while True:

break 文を含む実行ブロック

この例では条件式は真であることを表す定数 True ですので、常に成り立ち、while 文そのものは無限にループする仕組みとなっています。そのため実行ブロック内で脱出のための条件を確認し、break 文で脱出する必要があります。

実際に脱出のための条件が成り立たない場合は Python の実行を強制的に止めなければなりません。キー操作「Ctrl-C」で停止させてください。

4.9 if 文による分岐

4.9.1 無限ループ型での平方根の計算

無限ループ型で構成した平方根を計算するプログラムを次に示します。break 文で脱出するために if 文で計算精度を判定しています。ex3.py との違いは以下の 2 点です。

- ループを開始する際に終了条件を判定していません。
- while 文の条件は継続を判定するためのものであるが、break 文を発動する条件は終了を判定するためのものであり、逆になります。

プログラム 9 平方根を求めるプログラム（無限ループ型、ex3_2.py）

| 行 | ソースコード | 説明 |
|---|--------------|-------------|
| 1 | # x の平方根を求める | |
| 2 | x = 2 | |
| 3 | # | |
| 4 | rnew = x | |
| 5 | # | |
| 6 | while True: | 無限ループ型の繰り返し |

```

7   r1 = rnew
8   r2 = x/r1
9   rnew = (r1 + r2)/2
10  print(r1, rnew, r2)
11  diff = r1 - r2
12  if (diff < 0):
13      diff = -diff
14  if diff <= 1.0E-6:
15      break

```

差の絶対値が 10^{-6} 以下なら break で終了

4.9.2 if 文の構成

if 文は条件が成立している場合のみ与えられたブロックを実行するもののほか、成立しない場合に実行するブロックを指定するもの、成立しない場合にさらに別の条件を検査するものなど、いくつかのパターンがあります。

if 条件式:

成立した場合に実行するブロック

if 条件式:

成立した場合に実行するブロック

else:

成立しない場合に実行するブロック

if 条件式 1:

条件式 1 が成立した場合に実行するブロック

elif 条件式 2:

条件式 1 は成立しないが条件式 2 が成立する場合に実行するブロック

else:

いずれの条件も成立しない場合に実行するブロック

なお、C 言語の switch 文のように検査する値によって 3つ以上のブロックの実行を切り替える命令はありません。elif を多数回使うことで同様の分岐ができます。

4.10 条件式の書き方

4.10.1 比較演算

1) 数値の比較

条件の代表的なものは数値の比較です。比較するための演算子として次のものが用意されています。

表 5 Python の比較演算子

| 演算子 | 意味 | 備考 |
|--------------------|------------|------------------------------------|
| <code>==</code> | 等しい | 等号は 2 つです。これは 1 文字の場合を代入演算に使うためです。 |
| <code>!=</code> | 等しくない | 2 文字です |
| <code>></code> | 左辺が右辺より大きい | |
| <code><</code> | 左辺が右辺より小さい | |
| <code>>=</code> | 左辺が右辺以上で | 2 文字です |
| <code><=</code> | 左辺が右辺以下 | 2 文字です |

浮動小数点数(float)は多くの場合近似値を扱いますので、「等しい」という比較は予期せぬ振る舞いをする可能性があります。使用を避け、不等号で判断するようにしてください。

2) 文字列の比較

文字列についても上記の演算が使えます。ただし、大小関係は文字コード(unicode)としての番号で行われますので使用は慎重に行ってください。

また、上記に加えて「in」で左辺が右辺の文字列に含まれているかどうかを調べることができます。例えば

'a' in 'abc'

に対して

True

が得られます。

4.10.2 論理演算

複数の条件を合成するために論理演算子 「and」、「or」、「not」 が用意されています。

4.10.3 () による演算の優先

Python では演算子に優先順位が定義されており、

- 算術演算は比較演算よりも優先、
- 比較演算は論理演算よりも優先

されますが、プログラムを読みやすくするために () で明示的に優先順位を示した方がいいでしょう。例えば

```
a == 1 and b != 0
```

より以下のほうが読みやすいでしょう

```
(a == 1) and (b != 0)
```

4.11 if 文の入れ子

for 文を入れ子にして利用したように if 文を入れ子にして利用することもしばしば行われます。次の 2 つのプログラムは同じ判定を異なる書き方で書いたものです。

プログラム 10 複合的な条件を用いた分岐

| 行 | ソースコード | 説明 |
|---|------------------------------|------------|
| 1 | a = 1 | |
| 2 | b = 0 | |
| 3 | if (a == 1) and (b == 0): | |
| 4 | print("YES a==1 and b == 0") | 複合的な条件での分岐 |

プログラム 11 if 文を入れ子にした分岐

| 行 | ソースコード | 説明 |
|---|------------|---------------|
| 1 | a = 1 | |
| 2 | b = 0 | |
| 3 | if a == 1: | |
| 4 | if b==0: | if 文を入れ子にした分岐 |

5

print("YES a==1 and b == 0")

4.12 端末からの入力

これまで平方根を計算したい数値をプログラムに埋め込んで計算してきました。端末から入力する方法を考えましょう。以下は Python シェルの画面です。赤字は入力、青字は出力です。

| Python Shell の画面 | 備考 |
|-----------------------|---|
| >>> a = input("*** ") | “***” を入力促進の文字列にして input 関数で入力を得て a に代入 |
| *** sss | sss と入力 |
| >>> a | a の値を評価 |
| 'sss' | |
| >>> type(a) | type 関数で a のデータ型を調査 |
| <class 'str'> | 文字列型(str)であると表示 |
| >>> | |

input 関数の引数は端末に表示する文字列です。また、返り値のデータ型は文字列です。

数値データを得るにはこれを int() や float() で適した型に変換します。

先の平方根を求めるプログラムでは x の値を設定している箇所を

```
x = input("平方根を求める数 ")
```

```
x = float(x)
```

に入れ替えるか、あるいは一括して

```
x = float(input("平方根を求める数 "))
```

と書きかえることで変数 x に端末から入力された数値を得ることができます。

演習 20. ex3.py を改造して端末から平方根を求める数値を入力するようにしなさい。

ex3.py もしくは ex3_2.py を開き、File メニューの Save As で ex3_3.py として保存したうえで改造すること。

4.13 エラーへの対処

関数 `float()` や `int()` は引数として与えられた文字列が数値として解釈できない場合 `ValueError` という種類のエラーを発生します。特にその場合の処理を指定しないなければ Python はそこで処理を中断します。プログラム内でエラーを処理するには `try` 文を使います。

次のプログラムは継続的に入力を受け取りエラー処理を行い、正の数値の場合のみ `print(x)` で値を出力します。停止するためには Ctrl-C を入力してください。

プログラム 12 入力を得て検査するプログラム(`inputcheck.py`)

| 行 | ソースコード | 説明 |
|----|--|------------------------------|
| 1 | <code>while True:</code> | 無限ループ |
| 2 | <code>x = input("正の数値を入力してください ")</code> | |
| 3 | <code>try:</code> | エラーが生じる箇所を try ブロックに入れる |
| 4 | <code>x = float(x)</code> | <code>ValueError</code> への対応 |
| 5 | <code>except ValueError:</code> | 大文字小文字に注意 |
| 6 | <code>print(x, "は数値に変換できません")</code> | |
| 7 | <code>continue</code> | |
| 8 | <code>except:</code> | 他のエラーへの対応 |
| 9 | <code>print("予期していないエラーです")</code> | 終了する、try ブロックは ここまで。 |
| 10 | <code>exit()</code> | |
| 11 | <code>if (x <=0):</code> | 正の値であることの検査 |
| 12 | <code>print(x, "は正の数値ではありません")</code> | |
| 13 | <code>continue</code> | |
| 14 | <code># 以下は正しい入力が得られた時の処理</code> | while ブロックのつづき |
| 15 | <code>print(x)</code> | |

演習 21. エラー処理の確認

上のプログラムを実行し、さまざまな入力で動作を確認してください。

4.13.1 try 文の構成

try 文では例外を生じるブロックを try 文の中に入れ、catch 文で例外を指定して処理するブロックを書きます。例外が指定されていない catch 文は（その上で処理されるものを除いて）すべての例外に対して機能します。

try:

例外処理の対象とするブロック

catch 例外:

その例外が生じた際の処理を行うブロック

catch:

上記で指定した以外のすべての例外に対応するブロック

4.13.2 外部からの入力は疑え

外部から与えられる入力はプログラマーには統制できません。正しい入力だけを想定して書かれたプログラムは想定外の入力に正しく応答できず、場合によっては間違った結果を出してしまう恐れもあります。外部からの入力については「疑ってかかる」ことが重要で、値の妥当性を検査したり、生じえるエラーの処理を的確に行ったりすることが望まれます。

4.14 Python での数学関数

これまでの例では誤差 diff の絶対値を

```
if diff < 0:  
    diff = -diff
```

と明示的に計算していたが Python では絶対値関数 `abs()` が利用可能であり、上の計算は

```
diff = abs(diff)
```

を書けます。

また Python には数学関数を利用するためのライブラリ（モジュール） `math` が提供されています。これを用いるためには利用に先立って

```
import math
```

という命令でモジュールを導入します。ここで定義されている定数や関数は

`math.pi`

`math.sqrt(2)`

のようにモジュール名の後に「.」で呼び出したい定数や関数を書きます。上の例は円周率と平方根です。

演習 22. 上の例に従って Python Shell で `math` モジュールを使ってみてください。

4.15 数値を表示する際のフォーマット指定

Python で `print()` 関数で数値を表示すると、与えられた数値に合った桁数などが自動で選ばれます、利用者が表示する書式を指定することも可能です。具体的な例として

```
c = 2.99792458E8
na = 6.02214076E23
form = '光速は{0:12.8g} m/s, アボガドロ数は {1:12.8g} mol**(-1) です'
print(form.format(c, na))
```

といった感じで利用します¹。

3 行目は書式を指定する文字列で {} で囲まれた箇所が数値を変換する書式です。例えば `{0:12.8g}` は後で述べる `format` メソッドの引数の 0 番目の要素に対して、最小 12 桁とて、小数点以下 8 桁まで `g` 形式（浮動小数点数を表示する形式の一つ、整数を 10 進数で表示する場合は'd'、浮動小数点数を指数表記で表すには 'e'、固定小数点表記で表すには 'f' が用いられます。両者を値によって切り替えるのが 'g' 形式です）で変換するということを意味します。

4 行目の `form.format(c, na)` は文字列 `form` を書式に変数 `c` と `na` を変換した文字列を生成することを意味します。文字列変数（ここでは `form`）に付随するメソッド（関数のようなもの）`format` を呼び出して実施するのですが、メソッドは対象となる変数に「.」で続けて指定しています。

¹ 国際単位系(SI)では 2018 年の改定で 2019 年 5 月 20 日から「国際キログラム原器」が廃され、単位系を定義として与えられた物理定数で組み立てられるようになりました。ここで例として挙げた光速やアボガドロ数も計測して定めるものではなく、確定された量として定義されています。

4.16 力試し

演習 23. `inputcheck.py`, `ex3.py` を組み合わせて以下の条件を満たす平方根を求めるプログラム作成しなさい。

1. 平方根を求める数を繰り返し端末から入力できるようにすること。
2. 平方根を求める数の入力が数値に変換できない場合は、その旨を示して、次の入力を求めること。
3. 平方根を求める数が 0 以下の場合は、その旨を示して、次の入力を求めること。
4. 絶対値の計算には `abs()` 関数を用いること。

できれば以下にも挑戦すること

5. 端末からの入力が “end” という文字列なら終了すること。
6. 計算精度を絶対精度ではなく、相対精度で 10^{-6} とすること。これについて大きな数や小さな数（例えば 10^{10} や 10^{-10} ）の平方根を求め、結果を確認すること。

5. 関数を使った処理のカプセル化

5.1 本章の学習の目標

1. 本章では前章の例題を使って、まとまった処理を関数として定義して使用することを学びます。

5.2 前章の例題から

前章では平方根を求めるプログラムと、求めたい数を端末から入力する方法を学び、これらを組み合わせ、繰り返し端末から入力された値について平方根を求めるプログラムの構成に取り組みました。繰り返し端末から入力するプログラムの構成方法としては以下の図のような2通りが考えられます。

1. 端末から入力を得て、正しい入力が得られた場合にだけ平方根を計算し、入力を繰り返す。
2. 端末から正しい入力を得ることと、その入力についての平方根を求める直列に実行することを繰り返す。

これらをプログラミングするなら、上記の記述のように1. の場合は「平方根を求める」、2. なら「端末から正しい入力を得る」「平方根を求める」ということを素直に表現したいところです。それらをそれぞれ `get_positive_numeral()`, `square_root()` という関数で書けるものと想定するとプログラムはそれぞれ以下のように書けるはずです。

1. の場合

```
while True:  
    端末から入力 x を得て正の数値かどうかを検査する  
    if 正の数値:  
        r = square_root(x)  
        print(r)
```

2. の場合

```
while True:  
    x = get_positive_numeral()  
    r = square_root(x)
```

```
print(r)
```

このように関数として一定の処理を取りまとめることでその処理をカプセル化することができます。関数によるカプセル化のメリットは

- 呼び出す側のプログラムが短く分かりやすくなる。
- 同じ処理をプログラムの別の場所でも使える。
- 関数を利用する側のプログラムと定義する側のプログラムを分離することで、定義の修正などが行いやすくなる。

といったことです。

5.3 関数 `square_root()` を実装する

それでは上のうち、平方根の計算プログラム `ex3.py` を関数にすることで `square_root()` を実装してみましょう。

プログラム 13 関数 `square_root()` の実装

| 行 | ソースコード | 説明 |
|----|----------------------------------|---|
| 1 | # | |
| 2 | <code>def square_root(x):</code> | 引数 x を取る関数 |
| 3 | '引数 x の平方根を求める' | 説明文字列(docstring) |
| 4 | rnew = x | 以下、17 行目までが関数を定義するブロック、 <code>ex3.py</code> と同じだが、インデントに注意。 |
| 5 | # | |
| 6 | diff = rnew - x/rnew | |
| 7 | if (diff < 0): | |
| 8 | diff = -diff | |
| 9 | while (diff > 1.0E-6): | |
| 10 | r1 = rnew | |
| 11 | r2 = x/r1 | |
| 12 | rnew = (r1 + r2)/2 | |
| 13 | print(r1, rnew, r2) | |
| 14 | diff = r1 - r2 | |
| 15 | if (diff < 0): | |
| 16 | diff = -diff | |
| 17 | return rnew | 値を返す |

```

18 # ここからメインプログラム
19 v = 2
20 r = square_root(v)
21 print("結果は ", r)

```

計算した値を返り値で戻す

実行結果は以下のようになります。

Python では関数の定義に文字列(docstring)を入れておくとこの関数名を引数に help 関数を呼び出すことで説明が表示されます。

```

===== RESTART: C:/Users/一/Documents/Python Scripts/ex3_func.py =====
2 1.5 1.0
1.5 1.4166666666666665 1.3333333333333333
1.4166666666666665 1.4142156862745097 1.411764705882353
1.4142156862745097 1.4142135623746899 1.41421143847487
1.4142135623746899 1.414213562373095 1.4142135623715002
結果は 1.414213562373095

>>> help(square_root)
Help on function square_root in module __main__:

square_root(x)
    引数 x の平方根を求める

>>>

```

演習 24. 繰り返し平方根を求めるプログラムを関数 `square_root()` を定義して利用する形に書き換えなさい。

演習 25. 関数 `get_positive_numeral()` も構成し、繰り返し平方根を求めるプログラムをこれと `square_root()` を利用する形に書き換えなさい。

5.4 関数定義の書式

関数定義の書式は以下のようになります。

```

def 関数名(引数):
    関数として実行するブロック
    return 戻り値としてもどす値

```

- 関数名は変数名と同様なルールで定めます（一般に識別子）と呼びます。
- 引数は受け取りたい値を関数内で使う変数名（仮引数）で書きます。複数の引数を受け取るには仮引数を「,」で区切ります¹。
- 引数がない場合でも def 文の関数名のあとの()は省略できません。引数のない関数を呼び出すときも () は必要です。
- return 文は特定の条件が成立した場合の if 文のブロック内など、関数定義の最後以外の場所に書いてかまいません。
- 返り値が不要な関数なら return 文は要りません。

5.5 仮引数と実引数

関数を呼び出す側の引数を実引数、呼び出される側の引数を仮引数と呼びます。

- 実引数は変数ではなく、式や関数呼び出しでもかまいません。まず式として実引数を評価して、その結果を関数に渡します。
- 実引数と仮引数は同じ名前であることは要求しません。
- 数値や文字列などが仮引数で関数に渡された場合は関数内で仮引数に別の値を代入しても実引数の値には影響しません²。
- Python では複数の値を「,」で区切って return 文で返すことが可能です。呼び出す側は複数の変数を「,」で区切って受け取ります。1つの変数で受け取ると、この変数は複数の返り値の値で構成された「タプル」というデータ形式になります。

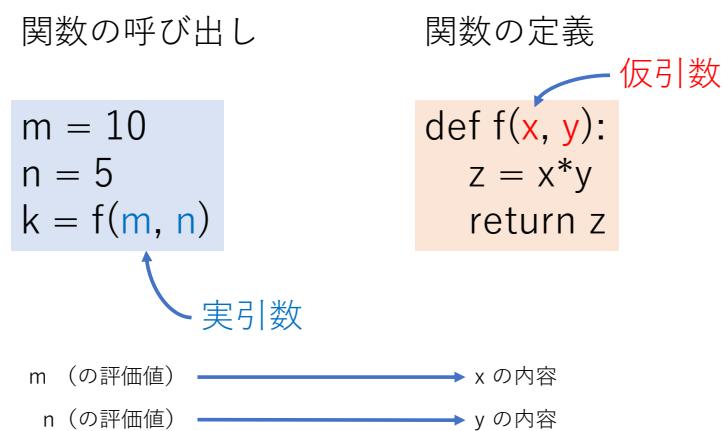


図 24 実引数と仮引数

¹ Python にはこれ以外の引き数の扱い方もあるのですが、ここでは省略します。

² リストの場合は書き換えが可能です。これについてはリストを紹介する際に改めて説明します。

5.6 関数内の変数の扱い

Python では関数内の変数は以下のように扱います。

- **ローカル変数**：関数内で定義された（代入された）変数は関数内でのみ利用可能で、関数の（毎回の）実行ごとに関数の実行が終了すれば失われます。
- **グローバル変数**：関数外で定義されている変数は値を読み取ることのみ可能です。
- **グローバル変数への代入**：関数内で `global` 宣言された変数のみ、グローバル変数に代入可能です。

Python で関数内の変数をこのように扱うことにより

- 関数内で一時的に必要な変数は他への影響を考えずに自由に使えます。
- グローバル変数の操作はプログラムが長くなるとプログラムを分かりにくくする要因になるのですが、Python では比較的安全な読み取りを許可する一方で、書き込みは `global` 宣言で明示的にプログラムに示すことで安全性と利便性のバランスをとっています。

| | | | |
|---|---------------|--|--|
| <code>a = 10</code> | a, b はグローバル変数 | | |
| <code>def f():</code> <code>global b</code> <code>c = a*a</code> <code>b= c</code> | 関数定義 | b をグローバル宣言 グローバル変数 a は参照のみ可能 c はローカル変数 グローバル宣言した変数は代入可能 | |
| <code>f()</code> | メインプログラム | | |

図 25 グローバル変数とローカル変数

5.7 関数の利用パターン

数学の関数とは異なり Python の関数の利用パターンはいくつかあります。関数に () 内に記述して与える情報を「引数」、返される値を「返り値」と呼びます。

- 引数を与え、返り値を使う。数学の関数と同様な使い方です。

```
y = math.fabs(-2.0)
```

- 引数も与えず、返り値も使わない。定型的な命令を実行する場合に使います。

例えば次章の `turtle` の `up()`

- 引数は与えるが返り値は使わない。可変な値を含む命令の実行に使います。

例えば次章の turtle の `forward(100)`

- 引数は与えないが返り値を使う。対象の状態を知るのに使います。

例えば次章の turtle の `p = pos()`

このほか、「グローバル変数を読み書き」する関数や「リストなど書き換え可能な引数を通じて情報をやりとり」する使い方もありますが、このような使い方は関数の「副作用」と呼ばれ、ソースコード（特に関数を呼び出す側）からは明示化されにくいため、プログラムを分かりにくくするという弊害もあります。

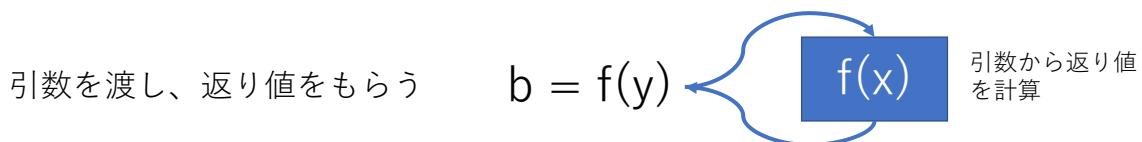
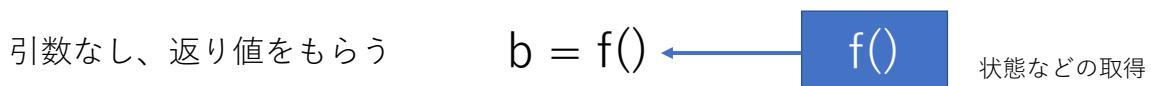


図 26 関数の利用パターン

| | |
|---|---|
| <code>a = 0</code> | グローバル変数 |
| <code>def f(): global a a = a+1</code> | グローバル変数を操作する関数 |
| <code>def g(x): x[0] = 0</code> | リスト型の引数の内容を操作する関数 |
| <code>f() print(a) b = [1,2,3] g(b) print(b)</code> | メインプログラム 関数の呼び出しにより グローバル変数の内容が変化 引数のリストの内容が変化 |

図 27 関数呼び出しと「副作用」

5.8 関数の呼び出しと関数オブジェクトの引き渡し

`def` 文で定義された関数を実行する際には引数の有無にかかわらず `f()` のようにカッコを付けます。これに対し、後の章で出てくるタートルグラフィクスでのマウスクリック時に実行する関数の指定や、`tkinter` での GUI プログラムでボタンが押された際に実行する関数の指定などでは関数名だけを表記します。このような表記により、関数をその場で実行するのではなく「後で実行する」関数そのものをオブジェクトとして引き渡すことができます。以下の例では関数 `f` を関数 `F` の引数として渡して `F` の中に `f` を実行しています。

```
def f():
    print("f says Hello")
# 関数を引数でもらって実行する関数
def F(y):
    print("In F, ", end="")
    y()
# f を実行
f()
f says Hello
# f を F に渡して F を実行
F(f)
In F, f says Hello
```

5.9 デフォルト引数値とキーワード引数

Python の関数の定義では引数名のあとに `=` で値を指定することで、引数が与えられない場合に暗黙で使用する値（デフォルト値）を指定することができます。また、関数を呼び出す側では、前から順に引数を渡すほかに「引数名=値」という形式（キーワード引数）で、特定の引数だけ値を渡すことができます。

```
def f(a, b=2, c=3):
    return a + b + c

f(1,1,1)
```

3

f(1)

6

f(1, c=2)

5

tkinter のプログラミングでは、数多くの引数があり、キーワード引数で必要なものだけを指定する使い方をします。

6. Turtle で遊ぶ

6.1 本章の学習の目標

1. Turtle を通じて Python でのモジュールの使用法を知る。
2. Turtle を通じて Python のクラス型オブジェクトの使用法を知る。
3. Turtle を使ったグラフィクスの作品作りを通じて、これまでに学んだことを確認するとともに、作品に必要なライブラリの使用などを主体的に学ぶ。

6.2 Turtle —由緒正しき亀さん

タートルグラフィクスは画面上の亀（ロボット）に前進や回転などの命令を与え、その軌跡としてグラフィクスを作成するものです。

これはマサチューセッツ工科大学(MIT)で開発された LOGO という言語に盛り込まれたグラフィクス機能でプログラムの動きを視覚化して学ぶことを意図しています。LOGO の開発者の一人であるシーモア・パパートは子供たちへのプログラミング教育に取り組んだことで有名ですが、彼の名言に

今日多くの学校では、「コンピューターによる学習」というと、コンピューターに子供を教えさせるということを意味する。コンピューターが子供をプログラムするのに使われていると言ってもよい。私の描く世界では、子供がコンピューターをプログラムし、そうする過程で最も進んだ強力な科学技術の産物を統御するという実感を得るとともに、科学、数学そして知性のモデルを作る学問などからくる深遠な理念と密接な関係を確立するのである。

シーモア・パパート[10]、太字は著者

があります。日本では戸塚さんという先生が LOGO を処理するプログラムを自ら作成し小学校での教育に取り組みました[11]。子供でも扱いやすいプログラミング言語として、アラン・ケイらによって開発された Squeak や、MIT で開発された Scratch¹ がありますが、Squeak や Scratch はタートルグラフィクスの機能を持っています。Scratch の導入としてマスコットの猫を走らせるプログラミングの例を用いますがタートルグラフィクスの流れを汲んだものです。Python でも Turtle というモジュールを利用することでタートルグラフィクスが楽しめます。

¹ Scratch を開発している MIT Media Lab のグループは「Lifelong Kindergarten」（生涯の幼稚園）と名乗っています。素敵なネーミングだと思いませんか？

数値ばかりを扱っていても退屈なので本章ではタートルグラフィクスを通じてこれまで学んだことのおさらいをします。

6.3 Python の Turtle モジュール

- Python のタートルグラフィクスは `turtle` モジュールとして提供されています。
- GUI 環境として `tkinter` を基盤にしています¹。
- 1つのタートルを関数呼び出しで操作する手続き指向と複数のタートルを扱えるオブジェクト指向の2種類の使い方が可能です。
- 注意：モジュール名と同じファイル名で（`turtle.py`）で Python プログラムを作らないこと。Python が正しいモジュールを検索できなくなります。

6.4 使ってみよう

次の表のプログラムを作成して実行してみてください。

プログラム 14 `turtle` を使う例
(`turtle.py` という名前で保存してはいけない)

| 行 | ソースコード | 説明 |
|----|-----------------------------------|--|
| 1 | <code>from turtle import *</code> | <code>turtle</code> で定義されている関数を呼べるようにする。 |
| 2 | <code>forward(100)</code> | |
| 3 | <code>left(90)</code> | |
| 4 | <code>forward(100)</code> | |
| 5 | <code>left(90)</code> | |
| 6 | <code>forward(100)</code> | |
| 7 | <code>left(90)</code> | |
| 8 | <code>forward(100)</code> | |
| 9 | <code>left(90)</code> | |
| 10 | <code>done()</code> | 終了 |

`forward()` はタートルを前進させる、`left()` は左に回す関数です。

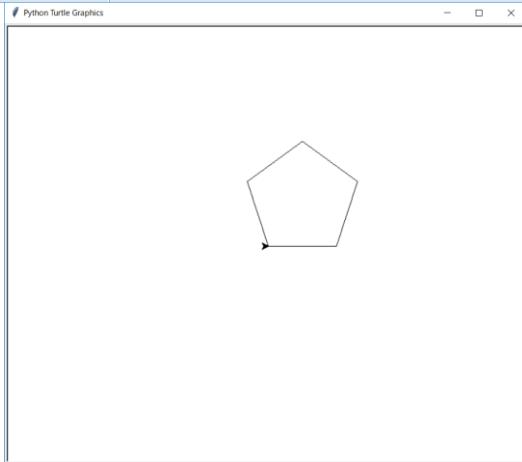
¹ 本授業で使用している IDLE や次に紹介する GUI 環境と共に通です。`tkinter` は Tcl/Tk という GUI ライブリを用いますが、Mac では予め導入されているバージョンに問題があるようです。

タートルはペンを持っており、ペンが降りている（デフォルト）では、移動した軌跡が残されます。

演習 26. 以下のプログラムを完成させて正 n 角形を書くプログラムを作成してください。

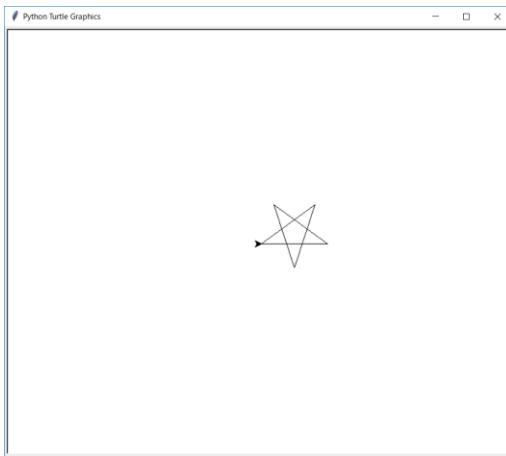
プログラム 15 n 角形を描くプログラム（未完成）

| 行 | ソースコード | 説明 |
|---|----------------------|-----------------------------|
| 1 | from turtle import * | turtle で定義されている関数を呼べるようにする。 |
| 2 | n = 5 | 正 5 角形を描く |
| 3 | for i in range(n): | n 回繰り返す |
| | done() | 終了 |



演習 27. 星形はどうやって描けばいいでしょうか

ヒント n 角形では turtle の向きが一巡で丁度 1 回転します。星形ではどうでしょう。



6.5 Turtle モジュールの主な関数

以下のような関数が使えます。詳しくは Python ドキュメントの

24.1. turtle --- タートルグラフィックス

を参照してください。

- forward(d): d だけ前進。fd(d) も同じ。
- back(d), bk(d), backward(d): 後退
- right(a), rt(a): 右へ a 度回転
- left(a), lt(a): 左へ a 度回転
- goto(x, y), setpos(x, y), setposition(x, y): 座標 x, y に移動
- setheading(a): 向きを a 度に設定
- pendown(), pd(), down(): 軌跡を描くペンを下ろします。
- penup(), pu(), up(): ペンを上げます
- position(), pos(): タートルの位置を 2 次元ベクトルとして返します。以下のように 2 つの変数で返り値を受け取ります。

x, y = pos()

- heading(): タートルの向きを返します
- isdown(): ペンが降りていれば True を、上がっていれば False を返します。

6.6 複数のタートルを動かす

6.6.1 プログラム例

| 行 | ソースコード | 説明 |
|---|--------|----|
|---|--------|----|

| | | |
|----|----------------------|-----------------------------|
| 1 | from turtle import * | turtle で定義されている関数を呼べるようにする。 |
| 2 | t1 = Turtle() | 1 つ目のタートル t1 を作る |
| 3 | t2 = Turtle() | 2 つ目のタートル t2 を作る |
| 4 | t1.color('red') | t1 の色を赤に |
| 5 | t2.color('blue') | t2 の色を青に |
| 6 | for i in range(180): | |
| 7 | t1.forward(5) | t1 は 5 ステップ前進 |
| 8 | t2.forward(3) | t2 は 3 ステップ前進 |
| 9 | t1.left(2) | それぞれ 2 度回転 |
| 10 | t2.left(2) | |
| 11 | done() | 終了 |

6.6.2 クラスオブジェクトの利用

この例ではそれぞれのタートルは Turtle クラスのオブジェクトとして生成されます。

1つのタートルには「居る場所（座標）」、「向いている方向」、「ペンが上がっているか、降りているか」、「ペンの色」などの状態をもっています。タートルをプログラミングするためには、これらの状態を知ったり、あるいは状態を変更したりすることが求められます。

クラス型のオブジェクトはこのような操作を記述するのに適した方式です。複数のタートルを使うことで、クラス型オブジェクトの使い方のポイントが理解できると思います。

1) タートルを作る

Turtle クラスのオブジェクトは Turtle() という命令で作成します。クラス型のオブジェクトを生成する関数を特に「コンストラクタ」と呼びます。下の例では生成されたオブジェクトを変数 t に代入することで、以後、変数 t でこのタートルを指定できます。

```
t = Turtle()
```

2) タートルを操作する

このタートルの状態を知ったり、状態を変えたりする方法として「メソッド」の呼び出しを行います。メソッドはオブジェクトの変数に「.」とメソッド名（と引数）

を書くことで行います。対象がそのオブジェクトであることを除けば関数呼び出しと同じようなものだと考えてください。

```
t.forward(10)
```

3) タートルの状態を知る

タートルの状態を知るには状態を返すメソッドを呼び出して結果を適当な変数に代入するなどすればいいでしょう。

```
x, y = t1.pos()
```

6.7 作品作りのためのヒント

6.7.1 マウスクリックに応答する

プログラム 16 タートルグラフィクスでのマウスクリックへの応答

| 行 | ソースコード | 説明 |
|---|-----------------------------|---|
| 1 | from turtle import * | turtle で定義されている関数を呼べるようにする。 |
| 2 | def come(x,y): | マウスがクリックされたときに実行される関数を定義、引数はクリックしたときのマウスカーソルの位置 |
| 3 | (xx,yy) = pos() | |
| 4 | newxy = ((xx+x)/2,(yy+y)/2) | |
| 5 | print(x,y) | 関数の定義はここまで |
| 6 | goto(newxy) | |
| 7 | onscreenclick(come) | マウスがクリックされたときに呼び出す関数を設定 (come の後に () が無いことに注意 |
| 8 | done() | |

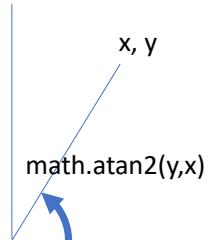
6.7.2 座標を角度に変換する

x, y 座標からその方向への角度を求めるために Python では math モジュールに atan2 という関数を用意しています¹。引数は縦の座標が前です。返り値はラジアン

¹ 角度から座標を計算するには三角関数 cos や sin を使えばいいのですが、三角関数の逆関数として適当なものがないため、逆正接関数(atan) の拡張として atan2 が導入されています。

なので、角度を「度」で設定する `turtle` で使うには、例えば以下のように変換します。

```
import math  
y = 2  
x = 1  
angle = math.atan2(y, x)*180/math.pi
```



6.7.3 亂数を使う

ランダムな動きも可視化すると面白いものの一つです。

- Python で乱数を使うには `random` モジュールを使います。
- コラムー乱数 を参照ください
- 亂数を使ったタートルグラフィクスの例を用意しました。`random_turtle.py` 参照ください。

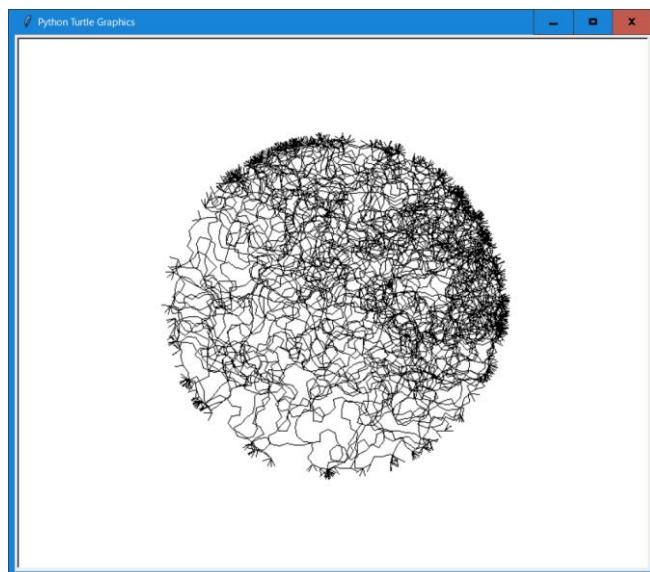


図 28 `random_turtle.py` の実行結果

6.7.4 フラクタル図形を描く

一部が全体と相似な図形を「フラクタル」と呼びます。フラクタルな図形は描かれたものも、それを描くアルゴリズムも興味深いものです。

- 関数の中で自分自身を呼び出す（再帰）を使う
- コラム一再帰 を参照ください
- detour.py, turtle-tree.py を参照ください。

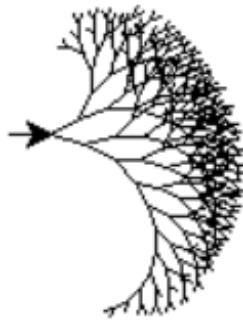


図 29 turtle-tree.py の実行結果

6.8 Turtle Demo

Python にはタートルグラフィクスのデモプログラムが用意されています。IDLE のメニューから呼び出せます。

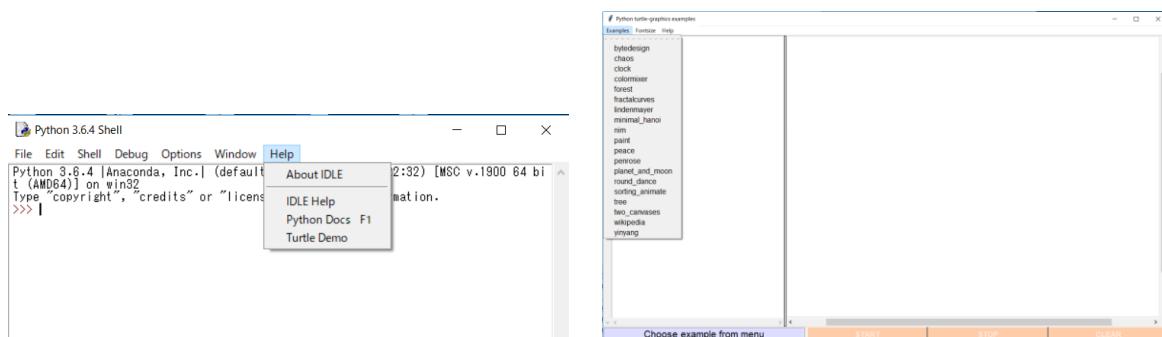


図 30 Turtle Demo の実行方法

6.9 課題 Turtle の作品制作

ここまで学習した Turtle を使った作品を制作してください。

- プログラムとスクリーンショット、作成メモを提出すること。
 - Windows で特定のウィンドウのスクリーンショットを得るには、そのウィンドウを選択しておいて ALT キーをおしながら PRTSC というキーを押してください。この操作でスクリーンショットがコピーされていますので、これに続けてペイントなど、画像を操作できるプログラムでスクリーンショットを張り付けることができます。
- プログラムは自身でプログラミングすること。
 - ただし、アドバイスを受けることは OK です。作成メモにはその旨、謝辞として記載してください。
- 作成メモには以下を記載すること。
 - 氏名、所属
 - 作品の説明
 - 自身で学習した Python や Turtle グラフィクスの機能
 - 参考にした情報があれば、その書誌情報、Web サイトなら、サイトのタイトル、URL、アクセス日時
 - 支援を得た人がいれば謝辞に支援を得た人と支援内容を書く

参考文献

- [14] シーモア・パパート著；奥村貴世子訳：マインドストーム：子供、コンピューター、そして強力なアイデア、未来社（1995）
- [15] 戸塚滝登著：コンピュータ教育の銀河、晚成書房（1995）

プログラム 17 random_turtle.py

| 行 | ソースコード |
|----|--|
| 1 | from turtle import * |
| 2 | import random |
| 3 | # 亂数を使うので random モジュールもインポート |
| 4 | |
| 5 | # 実行を停止するための変数（フラッグ） |
| 6 | stop_flag = False |
| 7 | |
| 8 | # マウスがクリックされたときの関数、引数 x, y をとるように |
| 9 | # しないといけないが、使わない |
| 10 | # 実行停止フラグを True にする |
| 11 | |
| 12 | def clicked(x,y): |
| 13 | global stop_flag |
| 14 | stop_flag = True |
| 15 | |
| 16 | # |
| 17 | # マウスがクリックされたときの動作を指定、clicked 関数を |
| 18 | # 呼び出す |
| 19 | # |
| 20 | onscreenclick(clicked) |
| 21 | |
| 22 | speed(0) |
| 23 | while(not stop_flag): |
| 24 | # -90 度から 90 度の範囲でランダムに向きを変える |
| 25 | left(random.randint(-90,90)) |
| 26 | forward(10) |
| 27 | # タートルの位置が原点から一定の距離を超えるれば、戻る |
| 28 | if (position()[0]**2+position()[1]**2 > 200**2): |
| 29 | forward(-10) |

プログラム 18 detour.py

| 行 | ソースコード |
|----|----------------------|
| 1 | from turtle import * |
| 2 | def detour(l): |
| 3 | if L < 10: |
| 4 | forward(L) |
| 5 | else: |
| 6 | LL = L/3 |
| 7 | detour(LL) |
| 8 | left(60) |
| 9 | detour(LL) |
| 10 | right(120) |
| 11 | detour(LL) |
| 12 | left(60) |
| 13 | detour(LL) |
| 14 | |
| 15 | for i in range(6): |
| 16 | detour(100) |
| 17 | left(60) |

プログラム 19 turtle-tree.py

| 行 | ソースコード |
|----|----------------------|
| 1 | from turtle import * |
| 2 | |
| 3 | # 再帰的に木を描く |
| 4 | def tree(n): |
| 5 | # 引数が 1 以下なら 5 歩すすむ |
| 6 | if n<=1: |
| 7 | forward(5) |
| 8 | else: |
| 9 | # 引数は 1 より大きいとき |
| 10 | # 引数の値に応じて前進（幹） |
| 11 | forward(5*(1.1**n)) |
| 12 | # 今の位置と向きを記録 |
| 13 | xx = pos() |
| 14 | h = heading() |
| 15 | # 左へ 30 度回転 |
| 16 | left(30) |
| 17 | # 大きさ n-2 で木を描く（左の枝） |
| 18 | tree(n-2) |
| 19 | # ペンを挙げて軌跡を残さない |
| 20 | up() |
| 21 | # 先に記録した位置（幹の先端）に戻る |
| 22 | setpos(xx) |
| 23 | setheading(h) |
| 24 | # ペンを降ろす |
| 25 | down() |
| 26 | # 右へ 15 度 |
| 27 | right(15) |
| 28 | # 大きさ n-1 で木を描く（右の枝） |
| 29 | tree(n-1) |
| 30 | # ペンを上げてもどる |
| 31 | up() |
| 32 | setpos(xx) |

```
33      setheading(h)
34      # ペンを降ろす
35      down()
36
37      # 時間がかかるので最も早い描画
38      speed(0)
39
40      # 大きさ 12 の木を描く
41      tree(12)
```

7. Tkinter で作る GUI アプリケーション(1)

7.1 本章の学習の目標

この章では tkinter を用いた GUI 型のアプリケーションの作成を通じて

- GUI アプリケーションにおけるフレームワークの役割とイベント駆動型プログラミングを理解する。
- GUI アプリケーションにおける MVC アーキテクチャを理解する。
- tkinter でのコールバック関数の実装を通じて Python での関数の定義方法をしる。
- 時計のプログラミングを通じてイベント駆動型プログラミングにおけるプログラムの自律的な動作の実現方法を知る。

7.2 GUI とイベント駆動型プログラミング

GUI 型のアプリケーションでは、メニュー やボタンなどによるさまざまな操作をユーザ自身が選択して利用します。そして、操作に対してコンピュータが適切に応答することを期待します。

- このようなユーザの操作を「イベント」と呼びます。
- 多くの GUI 型のアプリケーションは GUI 用の「フレームワーク」を利用します。
- フレームワークはマウスやキーボードの操作を監視してイベントを検出し、プログラマーによって設定されたイベント処理用のプログラムを呼び出します。

フレームワークを用いた GUI 型のアプリケーションではプログラマーは主として以下のようないくつかのプログラミングを担います。

- GUI アプリケーションとしてボタンなどを配置する画面の構成
- イベントが発生した際に行う処理の定義

このようなプログラミングをイベントに対する応答を主に記述することからイベント駆動型 (event-driven) プログラミングと呼びます。

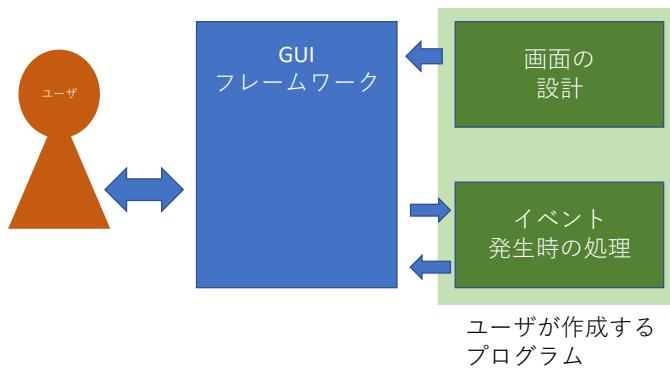


図 31 イベント駆動型プログラミングの枠組み

7.3 モデルとユーザーインターフェイスの分離

例えば加減乗除を扱う電卓のプログラムを作成することを考えましょう。ユーザがコンピュータに委ねたい仕事は加減乗除が2つの数値に対する演算（二項演算）ですから、

- 「第1項目の数値」と「第2項目の数値」と「適用する演算」を「設定し」、
- 設定された演算を「適用し」、
- 「適用結果」を「得る」

ことです。このうち青色で書いた言葉は数値や演算など操作の対象となる「もの」で「名詞（句）」で示されています。他方で、赤色で書いた言葉は「操作」で、サ变动詞を含む動詞で示されています。これらが「電卓」という仕事の「モデル」を構成します。

一方、このモデルに対して人が関わるために、具体的に人が操作するユーザーインターフェイスが必要になります。パソコン用コンピュータやスマートフォンの電卓アプリケーションのユーザーインターフェイスは具体例の一つですし、Pythonシェルのように、式をキーボードから入力することも考えられます。また、視覚に障害のある方のためには、音声や点字などのインターフェイスが求められるでしょう。

「電卓」という応用を中心に考えると、「モデル」は共通で、ユーザーインターフェイスはいろいろ変わるということが言えます。GUIのプログラミングにおいて、このような考え方を表すことばとして「MVCアーキテクチャー」があり、次の M, V, C を分けて考えようというものです。

- M: Model, アプリケーションの骨格を与える計算対象のモデル、基本的に GUI とは独立

- V: View、Model で得られた結果をユーザに示すプログラム、GUI が担う
- C: Control、モデルに対するユーザの操作のためのプログラム、GUI が担う

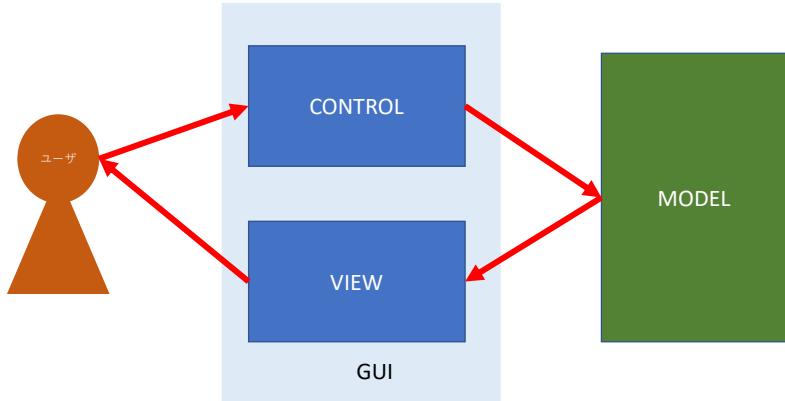


図 32 Model-View-Control アーキテクチャ

7.4 tkinter

パソコン用コンピュータの基本ソフトウェアとウィンドウ環境としては Windows, macOS, Linux/X-Window など複数のものが使われており、それぞれにウィンドウの描画などは異なった方法で行われます。これらの OS/ウィンドウ環境の差異を吸収し、共通に使える GUI 用のフレームワークとして Tk/Tk があります。

tkinter はこの Tk/Tk を Python から利用できるようにしたパッケージです。



図 33 Tkinter のシステム構成

7.4.1 tkinter の用語

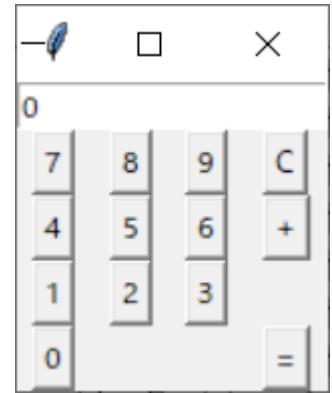
- **widget (ウィジェット)** : GUI を構成するボタンなどの部品の総称です。

- ・ コンテナ : widget (群) を格納する入れ物です。
- ・ レイアウト・マネージャ／ジオメトリ・マネージャ : widget の幾何学的配置を調整する仕組みです。
- ・ コールバック (Call back) 関数 : widge が操作されたときに必要な処理を行うために呼び出す関数

7.5 tkinter の例題(**tkdemo-2term.py**)

以下のような加算のみの電卓を考えます。

- ・ 0～9 の 10 キーと、C(クリア)、+(足し算) =(計算) の 13 個のボタンと
- ・ 数値を示す 1 行の文字入出力で構成されます。
- ・ 0～9 は電卓と同様に入力中の数字の最小桁を挿入します。(入力中の数字を 10 倍して押されたキーの数字を加える)
- ・ C は数値を 0 にします。
- ・ + キーは入力された数値を 2 項演算の第 1 項に登録し、入力中の数字を 0 にします。
- ・ = キーは入力中の数字を 2 項演算の第 2 項に登録し、足し算を実行し、その結果を表示するとともに入力中の数字を 0 します。



プログラム 20 加算のみの電卓(**tkdemo-2term.py**)

| 行 | ソースコード | 説明 |
|---|-------------------------|-----------------------------------|
| 1 | import tkinter as tk | tkinter を短縮形 tk で参照する形でインポート |
| 2 | # 計算機能のための変数とイベント用の関数定義 | |
| 3 | # 2 項演算のモデル | 二項演算を処理するための変数、関数、GUI に依存しないことに注意 |
| 4 | # 入力中の数字 | |
| 5 | current_number = 0 | |
| 6 | # 第一項 | |
| 7 | first_term = 0 | |
| 8 | # 第二項 | |
| 9 | second_term = 0 | |

```
12 # 結果
13 result = 0
14
15 def do_plus():
16     "+ キーが押されたときの計算動作、第一項の設定と入
力中の数字のクリア"
17     global current_number
18     global first_term
19
20     first_term = current_number
21     current_number = 0
22
23 def do_eq():
24     "= キーが押されたときの計算動作、第二項の設定、加
算の実施、入力中の数字のクリア"
25     global second_term
26     global result
27     global current_number
28     second_term = current_number
29     result = first_term + second_term
30     current_number = 0
31
32
33 # 数字キーの Call Back 関数
34 def key1():
35     key(1)
36
37 def key2():
38     key(2)
39
40 def key3():
41     key(3)
42
43 def key4():
44     key(4)
```

ここからはウィジェットの
Call Back 関数の定義

| | |
|----|--|
| 45 | |
| 46 | def key5(): |
| 47 | key(5) |
| 48 | |
| 49 | def key6(): |
| 50 | key(6) |
| 51 | |
| 52 | def key7(): |
| 53 | key(7) |
| 54 | |
| 55 | def key8(): |
| 56 | key(8) |
| 57 | |
| 58 | def key9(): |
| 59 | key(9) |
| 60 | |
| 61 | def key0(): |
| 62 | key(0) |
| 63 | |
| 64 | # 数字キーを一括処理する関数 |
| 65 | def key(n): |
| 66 | global current_number |
| 67 | current_number = current_number * 10 + n |
| 68 | show_number(current_number) |
| 69 | |
| 70 | def clear(): |
| 71 | global current_number |
| 72 | current_number = 0 |
| 73 | show_number(current_number) |
| 74 | |
| 75 | def plus(): |
| 76 | do_plus() |
| 77 | show_number(current_number) |
| 78 | |

| | | |
|-----|---|------------------------------------|
| 79 | def eq(): | |
| 80 | do_eq() | |
| 81 | show_number(result) | 数値をエントリーに表示する 関数 |
| 82 | | |
| 83 | def show_number(num): | |
| 84 | e.delete(0,tk.END) | Tk() でウインドウ作成 |
| 85 | e.insert(0,str(num)) | Frame コンテナ作成 |
| 86 | | Frame を割り付け |
| 87 | # tkinter での画面の構成 | |
| 88 | | |
| 89 | root = tk.Tk() | ボタンを表示テキストと Call back 関数を指定して生成 |
| 90 | f = tk.Frame(root) | |
| 91 | f.grid() | |
| 92 | | |
| 93 | # ウィジェットの作成 | |
| 94 | | |
| 95 | b1 = tk.Button(f,text='1', command=key1) | |
| 96 | b2 = tk.Button(f,text='2', command=key2) | |
| 97 | b3 = tk.Button(f,text='3', command=key3) | |
| 98 | b4 = tk.Button(f,text='4', command=key4) | |
| 99 | b5 = tk.Button(f,text='5', command=key5) | |
| 100 | b6 = tk.Button(f,text='6', command=key6) | |
| 101 | b7 = tk.Button(f,text='7', command=key7) | |
| 102 | b8 = tk.Button(f,text='8', command=key8) | |
| 103 | b9 = tk.Button(f,text='9', command=key9) | |
| 104 | b0 = tk.Button(f,text='0', command=key0) | |
| 105 | bc = tk.Button(f,text='C', command=clear) | |
| 106 | bp = tk.Button(f,text='+', command=plus) | |
| 107 | be = tk.Button(f,text="=", command= eq) | |
| 108 | | |
| 109 | # Grid 型ジオメトリマネージャによるウィジェットの # 割付 | |
| 110 | | ボタンを grid で位置を指定 して Frame に割り付け |
| 111 | b1.grid(row=3,column=0) | |

| | | |
|-----|-------------------------------------|--------------------------------------|
| 112 | b2.grid(row=3,column=1) | |
| 113 | b3.grid(row=3,column=2) | |
| 114 | b4.grid(row=2,column=0) | |
| 115 | b5.grid(row=2,column=1) | |
| 116 | b6.grid(row=2,column=2) | |
| 117 | b7.grid(row=1,column=0) | |
| 118 | b8.grid(row=1,column=1) | |
| 119 | b9.grid(row=1,column=2) | |
| 120 | b0.grid(row=4,column=0) | |
| 121 | bc.grid(row=1,column=3) | |
| 122 | be.grid(row=4,column=3) | |
| 123 | bp.grid(row=2,column=3) | |
| 124 | | |
| 125 | | 文字入力用の Entry ウィジェットを数値表示用に生成、横長に割り付け |
| 126 | # 数値を表示するウィジェット | |
| 127 | e = tk.Entry(f) | |
| 128 | e.grid(row=0,column=0,columnspan=4) | |
| 129 | clear() | |
| 130 | | mainloop メソッドで GUI の処理に移る |
| 131 | # ここから GUI がスタート | |
| 132 | root.mainloop() | |
| 133 | | |

7.6 tkinter を用いたプログラムの基本構成

1. モジュールのインポート

```
import tkinter as tk #短い名称で使えるようにする
```

2. Call back 関数の定義

```
def key1():
    key1 の内容
```

3. ウィンドウの作成

```
root = tk.Tk()
```

4. フレームの作成と割り付け。フレームはそのなかにウィジェットを格納するコンテナの一種です。

```
f = tk.Frame(root)      # root を親に Frame を作成し、
```

```
f.grid()                # grid() で割り付け
```

5. ウィジェットの作成（ボタン）

```
b1 = tk.Button(f, text='1', command=key1)
```

```
# f を親に、ボタンを作成、表示文字列は '1'、実行するコマンドは key1
```

6. ウィジェットの作成（エントリー、文字を表示する）

```
e = tk.Entry(f)
```

7. レイアウトの指定

```
b1.grid(row=3, column=0)
```

8. GUI の実行

```
root.mainloop()
```

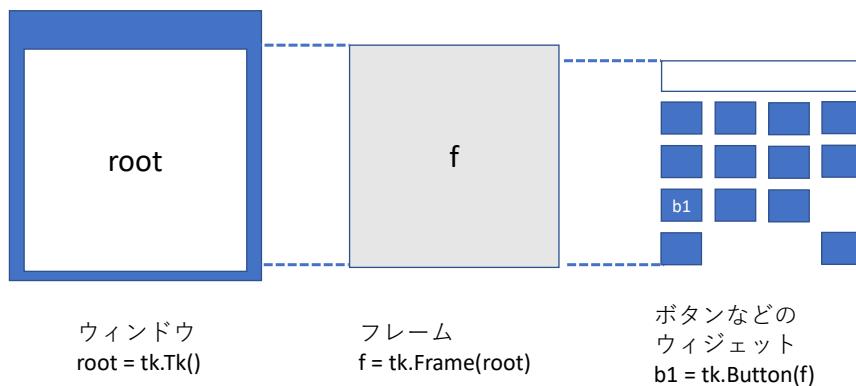


図 34 tkinter でのオブジェクトの関係

7.7 grid によるレイアウト

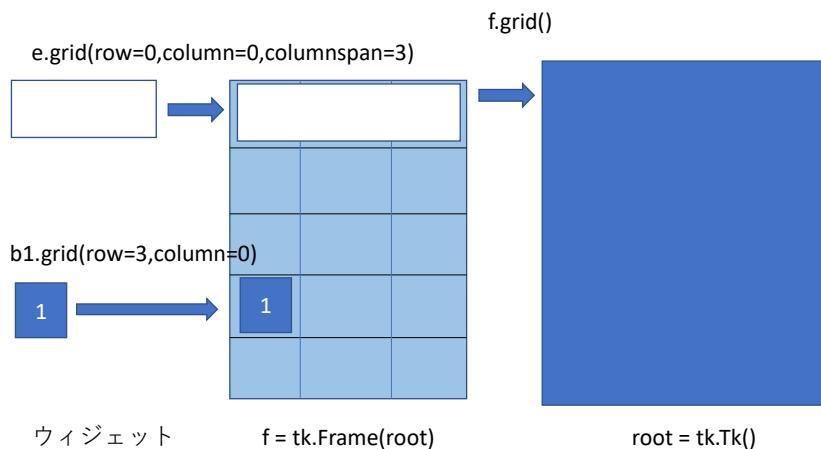
tkinter のウィジェットはレイアウトを管理するレイアウト・マネージャを指定して初めてウィンドウやコンテナに割り付けられます。レイアウト・マネージャには幾通りかありますが、簡単なものとして格子状の位置を与える `grid` があり、以下のように使います。

- 格子状のレイアウトを位置を指定して行う

```
b1.grid(row=3,column=0)
```

- いくつかのコラムをまたがる指定も可能

```
e.grid(row=0,column=0,columnspan=4)
```

図 35 `grid` によるレイアウト

7.8 lambda (λ) 表現を使った Call Back 関数の記述

先の例では関数 `key0()` ~ `key9()` の中身は引数を変えた関数 `key()` の呼び出しです。これは `widge` の定義において、

```
b1 = tk.Button(f,text='1', command=key1)
```

の中で引数 `command=key1` の右辺は「関数オブジェクト」でなければならず

```
b1 = tk.Button(f,text='1', command=key(1)) #これはまちがい
```

と書いてしまうと関数 `key()` を引数 1 で呼び出した結果の「返り値」が代入されてしまい、関数 `key()` を引数 1 も指定して call-back 関数として呼び出してもらうことになります。他方、`key1()` という関数はボタン `b1` の call-back 専用の関数であり、他には使われることはありません。その意味で、関数 `key1()` の定義とボタン `b1` の call-back 関数の指定を直接結びつけられると便利で `key1` という名前は不要になります。

これを実現する記法として Python では `lambda` 表現¹というものがあり、その場で関数名をつけずに関数を定義して、変数に代入する手法で、`b1` の例では

```
b1 = tk.Button(f,text='1', command=lambda:key(1))
```

と書けます。`lambda:` に続く `key(1)` がその場限りの関数定義の内容で、実質的に関数 `key1 ()` と同じ内容になります。

¹ 計算機科学で λ 計算と呼ばれている理論モデルが名称の由来です。

プログラム 21 lambda 式を使った引数付きコールバック関数の設定

| 行 | ソースコード | 説明 |
|----|--|----|
| 1 | import tkinter as tk | |
| 2 | | |
| 3 | # 計算機能のための変数とイベント用の関数定義 | |
| 4 | | |
| 5 | # 2 項演算のモデル | |
| 6 | # 入力中の数字 | |
| 7 | current_number = 0 | |
| 8 | # 第一項 | |
| 9 | first_term = 0 | |
| 10 | # 第二項 | |
| 11 | second_term = 0 | |
| 12 | # 結果 | |
| 13 | result = 0 | |
| 14 | | |
| 15 | def do_plus(): | |
| 16 | "+ キーが押されたときの計算動作、第一項の設定と入 力中の数字のクリア" | |
| 17 | global current_number | |
| 18 | global first_term | |
| 19 | | |
| 20 | first_term = current_number | |
| 21 | current_number = 0 | |
| 22 | | |
| 23 | def do_eq(): | |
| 24 | "= キーが押されたときの計算動作、第二項の設定、加 算の実施、入力中の数字のクリア" | |
| 26 | global second_term | |
| 27 | global result | |
| 28 | global current_number | |
| 29 | second_term = current_number | |
| 30 | result = first_term + second_term | |
| 31 | current_number = 0 | |

```
32
33 # 数字キーを一括処理する関数
34 def key(n):
35     global current_number
36     current_number = current_number * 10 + n
37     show_number(current_number)
38
39 def clear():
40     global current_number
41     current_number = 0
42     show_number(current_number)
43
44 def plus():
45     do_plus()
46     show_number(current_number)
47
48 def eq():
49     do_eq()
50     show_number(result)
51
52 def show_number(num):
53     e.delete(0,tk.END)
54     e.insert(0,str(num))
55
56 # tkinter での画面の構成
57
58 root = tk.Tk()
59 f = tk.Frame(root)
60 f.grid()
61
62 # ウィジェットの作成
63
64 b1 = tk.Button(f,text='1', command=lambda:key(1))
65 b2 = tk.Button(f,text='2', command= lambda:key(2))
```

```
66 b3 = tk.Button(f,text='3', command= lambda:key(3))
67 b4 = tk.Button(f,text='4', command= lambda:key(4))
68 b5 = tk.Button(f,text='5', command= lambda:key(5))
69 b6 = tk.Button(f,text='6', command= lambda:key(6))
70 b7 = tk.Button(f,text='7', command= lambda:key(7))
71 b8 = tk.Button(f,text='8', command= lambda:key(8))
72 b9 = tk.Button(f,text='9', command= lambda:key(9))
73 b0 = tk.Button(f,text='0', command= lambda:key(0))
74 bc = tk.Button(f,text='C', command=clear)
75 bp = tk.Button(f,text='+', command=plus)
76 be = tk.Button(f,text="=", command= eq)
77
78 # Grid 型ジオメトリマネージャによるウィジェットの
79 # 割付
80
81 b1.grid(row=3,column=0)
82 b2.grid(row=3,column=1)
83 b3.grid(row=3,column=2)
84 b4.grid(row=2,column=0)
85 b5.grid(row=2,column=1)
86 b6.grid(row=2,column=2)
87 b7.grid(row=1,column=0)
88 b8.grid(row=1,column=1)
89 b9.grid(row=1,column=2)
90 b0.grid(row=4,column=0)
91 bc.grid(row=1,column=3)
92 be.grid(row=4,column=3)
93 bp.grid(row=2,column=3)
94
95
96 # 数値を表示するウィジェット
97 e = tk.Entry(f)
98 e.grid(row=0,column=0,columnspan=4)
99 clear()
```

```

100
101 # ここから GUI がスタート
102 root.mainloop()

```

7.9 ウィジェットの体裁の調整

ボタンなどのウィジェットの体裁はウィジェット生成する際の引数で指定します。

- `font=('Helvetica', 14)` 文字フォントと大きさ
- `width=2` ウィジェットの大きさ
(ボタンでは文字数)
- `bg = '#ffffc0'` 背景色の指定、RGB でそれぞれ 16 進数 2 桁
00 暗い、ff 明るい

なお、mac では `bg` の設定で色が変わらないという報告があり、代わりに `highlightbackground` という引数を設定するとよいようです。

表 6 tkinter での色指定

| 表記 | Red | Green | Blue | 色 |
|-----------|-----|-------|------|---|
| '#ffffff' | ff | ff | ff | 白 |
| '#000000' | 00 | 00 | 00 | 黒 |
| '#ff0000' | ff | 00 | 00 | 赤 |
| '#00ff00' | 00 | ff | 00 | 緑 |
| '#0000ff' | 00 | 00 | ff | 青 |

演習 28. ウィジェットの体裁の調整

- 足し算電卓のフォントサイズ、ウィジェットの色を以下のように設定してください。背景色を Frame は'#ffffc0'（薄黄色）、数字キーは白、クリアキーは赤、+、= キーは緑にする。
- ボタンの大きさは 2 （文字分）にする。
- ボタンとエントリーのフォントとサイズは('Helvetica', 14)にする。

演習 29. 電卓の四則演算への拡張

足し算電卓を 4 則演算が可能なように拡張してください。ただし以下に留意する

こと。

- ボタンの配置は適宜検討すること。
- 割り算は 0 で割るエラーが発生する可能性があるので、第 2 項の数値が 0 の場合は何もしない。
- 割り算の小数点以下は切り捨てる。Python で整数商を求める演算子は「//」です。

ヒント：電卓では + などの演算キーが押されたときにはその演算は行われず、 = キーが押されたときに実行されます。このため、加減乗除の演算キーが押されたときには、どの演算を行うべきかを実行の際まで変数に記憶しておき、 = キーが押されたときに、記憶していた演算に応じて動作を変える必要があります。

演習 30. 実際の電卓との差異

作成したプログラムと実際の電卓（や電卓アプリ）との動作の違いを探ってください。例えば = キーの代わりに + などの演算キーを押した場合の動作など。

7.10 tkinter の終わり方

tkinter を用いたアプリケーションでは mainloop() を呼び出すと、ユーザの操作を待って、call back 関数を呼び出す無限ループになります。ウィンドウの終了ボタンで終了する以外の終了方法は以下のようになります。

- mainloop() から脱出するには、何かの call back 関数の中で tk.Tk() で作成したオブジェクト（例えば root）の quit() メソッドか destroy() メソッドを呼び出します。これらの動作の違いは以下のようになります。
 - quit(): ループは脱出しますが、ウィンドウやウィジエットは残ります。
 - destroy(): ループを脱出し、ウィンドウやメソッドそのものをなくします。

7.11 Frame クラスを拡張する方式での実装法

先のプログラムでは Frame と Button などのウィジエットは別に構成していましたが、tkinter の実装例では Frame を拡張したクラスとして、その初期化の中でウィジエットを生成するプログラムをしばしば見かけます。ここでは、この方式での実装例を示します。

プログラム 22 Frame クラスを拡張する tkinter の実装法

| 行 | ソースコード | 説明 |
|----|-------------------------------------|----|
| 1 | import tkinter as tk | |
| 2 | | |
| 3 | # 計算機能のための変数とイベント用の関数定義 | |
| 4 | # Frame のサブクラスを使った実装例 | |
| 5 | | |
| 6 | # 2 項演算のモデル | |
| 7 | # 入力中の数字 | |
| 8 | current_number = 0 | |
| 9 | # 第一項 | |
| 10 | first_term = 0 | |
| 11 | # 第二項 | |
| 12 | second_term = 0 | |
| 13 | # 結果 | |
| 14 | result = 0 | |
| 15 | | |
| 16 | def do_plus(): | |
| 17 | "+ キーが押されたときの計算動作、第一項の設定と入力中の数字のクリア | |
| 18 | " | |
| 19 | global current_number | |
| 20 | global first_term | |
| 21 | first_term = current_number | |
| 22 | current_number = 0 | |
| 23 | def do_eq(): | |
| 24 | "= キーが押されたときの計算動作、第二項の設定、加算の実施、入力中の | |
| 25 | 数字のクリア" | |
| 26 | global second_term | |
| 27 | global result | |
| 28 | global current_number | |
| 29 | second_term = current_number | |
| 30 | result = first_term + second_term | |
| | current_number = 0 | |

```
31 #  
32 # tk.Frame を継承した MyFrame というクラスを作り  
33 # その中でウィジェットやコールバック関数（メソッド）を  
34 # 設定する。tkinter をつかう定番  
35 #  
36 class MyFrame(tk.Frame):  
37 #  
38 #     __init__ はクラスオブジェクトを作る際の初期化メソッド  
39 # アンダースコアは前後それおれ2つづつ  
40     def __init__(self, master = None):  
41         super().__init__(master)  
42 # あとで参照しないウィジェットの作成、ローカル変数  
43         b1 = tk.Button(self,text='1', command=lambda:self.key(1))  
44         b2 = tk.Button(self,text='2', command=lambda:self.key(2))  
45         b3 = tk.Button(self,text='3', command=lambda:self.key(3))  
46         b4 = tk.Button(self,text='4', command=lambda:self.key(4))  
47         b5 = tk.Button(self,text='5', command=lambda:self.key(5))  
48         b6 = tk.Button(self,text='6', command=lambda:self.key(6))  
49         b7 = tk.Button(self,text='7', command=lambda:self.key(7))  
50         b8 = tk.Button(self,text='8', command=lambda:self.key(8))  
51         b9 = tk.Button(self,text='9', command=lambda:self.key(9))  
52         b0 = tk.Button(self,text='0', command=lambda:self.key(0))  
53         bc = tk.Button(self,text='C', command=self.clear)  
54         bp = tk.Button(self,text='+', command=self.plus)  
55         be = tk.Button(self,text="=", command=self.eq)  
56  
57 # Grid 型ジオメトリマネージャによるウィジェット割付  
58         b1.grid(row=3,column=0)  
59         b2.grid(row=3,column=1)  
60         b3.grid(row=3,column=2)  
61         b4.grid(row=2,column=0)  
62         b5.grid(row=2,column=1)  
63         b6.grid(row=2,column=2)  
64         b7.grid(row=1,column=0)
```

```
65     b8.grid(row=1,column=1)
66     b9.grid(row=1,column=2)
67     b0.grid(row=4,column=0)
68     bc.grid(row=1,column=3)
69     be.grid(row=4,column=3)
70     bp.grid(row=2,column=3)
71
72 # 他のメソッドで参照する数値を表示するウィジェット、クラスオブジェクトの
73 # 変数として作成、頭に self. がつく
74     self.e = tk.Entry(self)
75     self.e.grid(row=0,column=0,columnspan=4)
76 # クラスの定義では
77 # メソッドの最初の引数は self, 中でクラスオブジェクトの変数、
78 # メソッドは self をつけて参照
79 #
80     def key(self,n):
81         global current_number
82         current_number = current_number * 10 + n
83         self.show_number(current_number)
84
85     def clear(self):
86         global current_number
87         current_number = 0
88         self.show_number(current_number)
89
90     def plus(self):
91         do_plus()
92         self.show_number(current_number)
93
94     def eq(self):
95         do_eq()
96         self.show_number(result)
97
98     def show_number(self, num):
```

| | | |
|-----|---------------------------|--------------------|
| 99 | self.e.delete(0,tk.END) | |
| 100 | self.e.insert(0,str(num)) | |
| 101 | | |
| 102 | # | |
| 103 | # ここからメインプログラム | |
| 104 | # | |
| 105 | root = tk.Tk() | |
| 106 | f = MyFrame(root) | |
| 107 | f.pack() | 拡張した クラスを 使う |
| 108 | f.mainloop() | |
| 109 | | |

参考文献

Tkinter については以下の資料のほか、さまざまな解説記事がインターネット上に公開されています。Python 2 と Python 3 で tkinter の使い方が若干異なっています。例えば import するモジュールが Python 2 では Tkinter であるのに対し、Python 3 では tkinter であるなどです。記事などを参照する際には注意してください。

- [16] Tkinter 8.5 reference: a GUI for Python
<https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html>

8. Tkinter で作る GUI アプリケーション(2)

8.1 本章の学習の目標

本章ではアナログ時計を tkinter で作成することを通じて

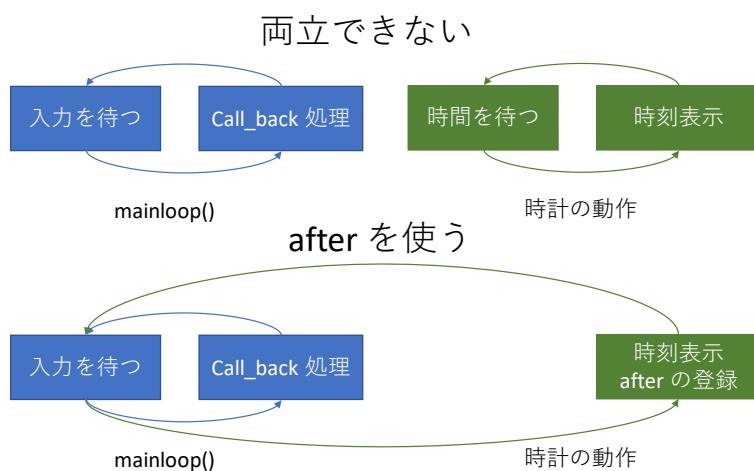
- アニメーションなど自律的に動くプログラムと GUI との共存の方法と
- Canvas ウィジェットを用いたグラフィックスの描画について学びます。

8.2 自律的に動作するプログラムと GUI との衝突

tkinter など GUI のフレームワークはユーザの操作を観測してマウスクリックなどイベントが発生すると設定されたコールバック関数に処理を委ねます。その際、コールバック関数の処理はすみやかに終了することを想定しており、終了をまって再びイベントの発生の観測に戻ります。

他方で、アニメーションなどプログラム自身が継続的動く場合、これをコールバック関数で呼んでしまうと、イベントの観測が停止してしまいます。

tkinter ではこれら両方のニーズを調整する方法として、一定時間後に指定されたコールバック関数を実行する `after` というメソッドが用意されています。アニメーションなどの継続的な処理を一定時間ごとに行う処理としてコールバック関数で実行し、設定した時間後の処理を `after` メソッドで tkinter に登録してコールバック関数を終えることで、GUI のイベント観測ループを長い時間止めないようにするのです。



なお、シミュレーションなど、計算時間そのものを要する応用にはこの方法はあまり適しません。プログラムを並行動作させるスレッド(thread)などの考え方、ライブラリの使用を検討する必要があります。

8.3 モジュール

Python でのさまざまなライブラリは「モジュール」という形で提供されます。モジュールの利用は、Python スクリプトの中で必要なモジュールをインポート(import)する形で行います。インポートの仕方としては以下のような方法があります。

- `import math: math` というモジュールをそのままインポートします。モジュール内の関数や定数などは `math.pi` のようにモジュール名 `pi` とドット ‘.’ を前に付ける形で行います。
- `import tkinter as tk: tkinter` というモジュールを `tk` という別名で利用可能にします。
- `from モジュール名 import * : turtle` の章で行ったようにモジュールの関数などをすべて関数名だけで利用可能にします。乱用は危険です。

注意: モジュール名と同じ名前の Python スクリプトを作業フォルダに作らないことに注意してください。Python がモジュールを探索する際に誤ってしまいます。

8.4 tkinter を用いたアナログ時計プログラム

ここでは図 36 に示すようなアナログ時計を作成します。時針、分針、秒針を表示するほか、日付の表示をボタンでオンオフできるものです。時計の針先位置の計算は図 37 を参考してください。

以下のプログラムは `Frame` クラスを拡張する方法で実装されています。長くなる行がありますが、以下のリストで行番号がついていないところは、長い行を折り返している箇所です。入力の際に注意してください。



図 36 作成するアナログ時計

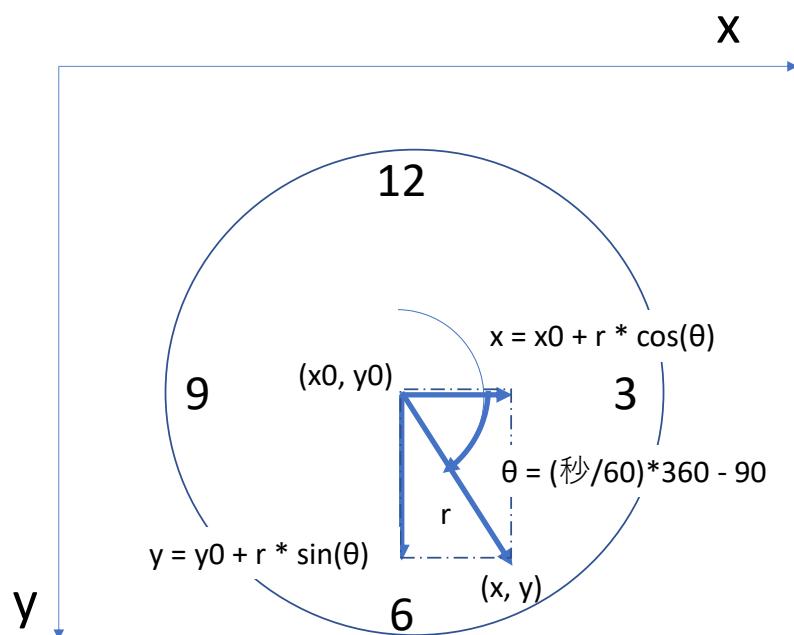


図 37 時計の針先位置の計算

8.4.1 ソースコード

プログラム 23 tkinter でのアナログ時計

| 行 | ソースコード | 説明 |
|----|---|------------|
| 1 | # tkinter canvas を使ったアナログ時計 | |
| 2 | # | |
| 3 | import tkinter as tk | 時刻を扱うため |
| 4 | import math | time をインポー |
| 5 | import time | ト |
| 6 | # | |
| 7 | # | |
| 8 | # Frame を拡張したクラス | |
| 9 | # | |
| 10 | class MyFrame(tk.Frame): | |
| 11 | def __init__(self, master = None): | |
| 12 | super().__init__(master) | |
| 13 | # | |
| 14 | # キャンバスの作成 | 描画用のウィジ |
| 15 | # | エット |
| 16 | self.size = 200 | |
| 17 | self.clock = tk.Canvas(self, width=self.size, height=self.size, | |
| | background="white") | |
| 18 | self.clock.grid(row=0, column=0) | |
| 19 | # | |
| 20 | # 文字盤の描画 | |
| 21 | # | |
| 22 | self.font_size = int(self.size/15) | |
| 23 | for number in range(1,12+1): | |
| 24 | x = self.size/2 + math.cos(math.radians(number*360/12 - | |
| | 90))*self.size/2*0.85 | |
| 25 | y = self.size/2 + math.sin(math.radians(number*360/12 - | |
| | 90))*self.size/2*0.85 | |
| 26 | self.clock.create_text(x,y,text=str(number), fill="black", | |
| | font =("",14)) | |

```
27  #
28  # 日付表示をオンオフするボタンの作成
29  #
30      self.b = tk.Button(self, text="Show Date", font=("",14),
31      command = self.toggle)
32      self.b.grid(row = 1, column = 0)
33  #
34  # 時刻の経過確認などの動作のためのインスタンス変数
35  #
36      self.sec = time.localtime().tm_sec
37      self.sec2 = time.localtime().tm_sec
38      self.min = time.localtime().tm_min
39      self.hour = time.localtime().tm_hour
40      self.start = True
41      self.show_date = False
42      self.toggled = True
43  #
44  # ボタンが押されたときの call back
45  #
46  def toggle(self):
47      if self.show_date:
48          self.b.configure(text="show date")
49      else:
50          self.b.configure(text="hide date")
51      self.show_date = not self.show_date
52      self.toggled = True
53  #
54  #
55  # 変化する画面の描画
56  #
57  def display(self):
58      #
59      # 秒針の描画、最初(start == True) か秒が変わったとき
```

```
60      #
61      if self.sec != time.localtime().tm_sec or self.start:
62          self.sec = time.localtime().tm_sec
63          angle = math.radians(self.sec*360/60 - 90)
64          x0 = self.size/2 - math.cos(angle)*self.size/2*0.1
65          y0 = self.size/2 - math.sin(angle)*self.size/2*0.1
66          x = self.size/2 + math.cos(angle)*self.size/2*0.75
67          y = self.size/2 + math.sin(angle)*self.size/2*0.75
68          #
69          # 前の描画をタグで検索して消してから描画
70          #
71          self.clock.delete("SEC")
72          self.clock.create_line(x0,y0,x,y, width=1, fill="red",
73                                tag="SEC")
74          #
75          # 分針、時針の描画、1分毎、時針は分まで考慮
76          #
77          if self.min != time.localtime().tm_min or self.start:
78              self.min = time.localtime().tm_min
79              x0 = self.size/2
80              y0 = self.size/2
81              angle = math.radians(self.min*360/60 - 90)
82              x = self.size/2 + math.cos(angle)*self.size/2*0.65
83              y = self.size/2 + math.sin(angle)*self.size/2*0.65
84              self.clock.delete("MIN")
85              self.clock.create_line(x0,y0,x,y, width=3, fill="blue",
86                                    tag="MIN")
87              self.hour = time.localtime().tm_hour
88              x0 = self.size/2
89              y0 = self.size/2
90              angle = math.radians((self.hour%12+self.min/60)*360/12 -
91                               90)
92              x = self.size/2 + math.cos(angle)*self.size/2*0.55
93              y = self.size/2 + math.sin(angle)*self.size/2*0.55
```

```
91         self.clock.delete("HOUR")
92         self.clock.create_line(x0,y0,x,y, width=3, fill="green",
93                               tag="HOUR")
94         self.start = False
95         #
96         # 日付の描画、秒が変わるか、ボタンが押されたとき
97         #
98         if self.sec2 != time.localtime().tm_sec or self.toggled:
99             self.sec2 = time.localtime().tm_sec
100            self.toggled = False
101            x = self.size/2
102            y = self.size/2 + 20
103            text = time.strftime('%Y/%m/%d %H:%M:%S')
104            self.clock.delete("TIME")
105            if self.show_date:
106                self.clock.create_text(x, y, text=text, font=(" ",12),
107                                      fill="black", tag="TIME")
108                #
109                # 100 ミリ秒後に再度呼び出す
110                #
111                self.after(100, self.display)
112
113    root = tk.Tk()
114    f = MyFrame(root)
115    f.pack()
116    f.display()
117    root.mainloop()
```

8.4.2 このプログラムのポイント

- モジュールのインポート tkinter のほか、時刻を扱うため time を、三角関数を利用するため math モジュールをインポートしています。(4 ~ 6 行)

- Frame を拡張した MyFrame クラスを定義し、そのなかでウィジェットの作成、割り付け、コールバック関数の定義をしています。
 - MyFrame クラス `__init__()` メソッドはクラスのオブジェクトを生成する際に自動的に呼び出されるメソッドです。この中に必要なウィジェットを作成しています。(12~42 行) アンダースコアは前後、それぞれ 2 つづつ。
 - ✧ 描画に用いる Canvas ウィジェットを生成しています (18 行)
 - ✧ Canvas ウィジェットの `create_text()` メソッドを呼び出す形で文字盤を描画しています。(23~27 行)
 - ✧ 時刻の文字での表示の有無を切り替えるボタンを生成しています。(31,32 行)
 - ✧ 時刻の経過、表示の切り替えなどのためのインスタンス変数を確保し、`time.localtime()` 関数の秒、分、時などで値を設定しています。
 - ボタンが押されたときのコールバック関数の定義。ボタン(b)の表示文字を `b.configure()` メソッドを呼び出して切り替えたり、状態を表す変数を設定したりしています。(47~53 行)
 - 時計の文字盤を描画するメソッド。(58~111 行)
 - ✧ 描画回数を減らすために、初めての描画か、時刻が変化したかを検出して必要な時に描画しています。
 - ✧ Canvas での描画は描画したものに「タグ」をつけておくと、それで後から消去できます。古い描画（時計の針）などをそれではまず消去します。
 - ✧ 時計の針の座標を時刻から三角関数で変換して計算し、`create_line()` メソッドで描画しています。
 - ✧ このメソッドの最後で `after` メソッドで 100 ミリ秒後に、このメソッド自身の呼び出しを設定することで継続的な時計の描画を行います。
- 113 行以降がメインプログラムです。`Tk()` メソッドでウィンドウを作り、MyFrame クラスのオブジェクトを生成し、初回分の描画を `f.display()` で行ったあと、`mainloop()` でプログラムの制御を `tkinter` に渡します。

演習 31. 使用するメソッドなどの確認

このプログラムで呼び出している `time` モジュール、`math` モジュール、`tkinter` の `Canvas` クラスのメソッドなどをリストアップし、メソッドの内容などを Python のオンラインマニュアルなどで確認しなさい。

演習 32. アナログ時計の改造

アナログ時計のプログラムについて以下の改造を加えなさい。

1. 日付の表示について、日付と時刻ではなく、日付と午前、午後を表示するようにしてください
2. ボタンをもう一つ追加し、秒針の表示をする、しない、を切り替えるようにしてください。

ヒント：変数 `self.start`, `self.toggled` の役割を確認して、秒針の表示について同じようなことを行うにはどうすればいいかを考えてください。

9. クラス

9.1 本章の学習の目標

すでにタートルグラフィクスや `tkinter` でもクラス型オブジェクトの利用などを学びました。ここではクラスについて以下を学びます。

- オブジェクト指向プログラミングの考え方を知る。
- クラスを定義して利用する。
- クラスで扱う変数について知る。

9.2 オブジェクト指向プログラミング

タートルグラフィクスで複数のタートルを扱うには以下のようなことを行いました。

- 必要なだけタートルを生成する。
- 個々のタートルに対してメソッドを呼び出す形で、動作を指示したり、状態を照会したりする。

個々のタートルは位置や向き、ペンの色やペンが降りているかどうか、などの状態を持っていました。

このタートルのように、それ自体が内部に状態などを持っていて、外からメソッドを呼び出すことで動作を指示できるようなものを「オブジェクト」と呼び、オブジェクトを使ってプログラミングする方法をオブジェクト指向プログラミングと呼びます。

`tkinter` のアナログ時計の例では、`tkinter` の `Frame` クラスを拡張する形でプログラミングを行いました。「クラス」は独自の状態やメソッドを持つオブジェクトを生成するための「型」の記述です。クラスから生成された個々のオブジェクトを「インスタンス」と呼びます。

簡単にまとめると

- (クラス型の) オブジェクトは独自の状態(変数)とメソッドを有するプログラムの要素、タートルのように命令できるロボットのような存在だと思う。
- クラスとはその型のオブジェクトがどのような変数とメソッドを持つかという記述。オブジェクトを生成するときの型となる。

- インスタンスはクラスを型に生成された個々のオブジェクト
- オブジェクト指向プログラミングはクラスの定義と生成されたインスタンスを用いてプログラムを作成する方法。ロボットを協調動作させるようにプログラムを書く考え方。

9.3 Python でのクラスの書き方、使い方

tkinter の例題と同様、二項演算をする CUI 型のプログラムを作成します。クラスを使って、第 1 項、第 2 項、演算結果、演算子を保持する変数と実際に演算を実行するメソッドをとりまとめます。

9.3.1 ソースコード

プログラム 24 CUI 型電卓プログラム

| 行 | ソースコード | 説明 |
|----|--|-------------|
| 1 | class Dentaku(): | |
| 2 | def __init__(self): | 初期化メソッド |
| 3 | self.first_term = 0 | |
| 4 | self.second_term = 0 | |
| 5 | self.result = 0 | |
| 6 | self.operation = "+" | |
| 7 | | |
| 8 | def do_operation(self): | 計算を実行するメソッド |
| 9 | if self.operation == "+": | |
| 10 | self.result = self.first_term + self.second_term | |
| 11 | elif self.operation == "-": | |
| 12 | self.result = self.first_term - self.second_term | |
| 13 | | |
| 14 | ここからメインプログラム | |
| 15 | dentaku = Dentaku() | オブジェクトの生成 |
| 16 | while True: | |
| 17 | f = int(input("First term ")) | |
| 18 | dentaku.first_term = f | |
| 19 | o = input("Operation ") | |

```

20     dentaku.operation=o
21     s = int(input("Second term "))
22     dentaku.second_term=s
23     dentaku.do_operation()
24     r = dentaku.result
25     print("Result ", r)

```

9.3.2 プログラムの概要

- クラス定義のブロック（1～12行目）、クラスの定義は以下のように行います

class クラス名 ():

メソッドなどの定義

- クラス名（この例では Dentaku）は変数と同様のルールで決めればいいのですが、慣習として先頭を大文字、残りを小文字にします。複数語での命名の場合、各語の先頭を大文字にし、単語間は詰めます。（例えば KansuuDentaku）
- メソッド **__init__(self)** の定義（2～6行）。Python では **_** で始まるメソッドや変数などに特殊な役割を持たせることが多いですが、**__init__()** はクラスのオブジェクトが生成される際に必ず実行されるメソッドです。クラス内で使う変数の初期化などに使います。関数と異なり、クラスのメソッド定義では引数をかならず1つ書かなければならず、通常 **self** という名前で与えます。呼び出しの際にはこの引数の値はシステムが自動的に与え、呼び出す際には第一引数は書く必要はありません。
- インスタンス変数と初期化（3～6行）。**__init__()** メソッド内で行っているのはクラスで使う変数の初期化です。**self.** で始まる変数は「インスタンス変数」と呼ばれ、そのクラスのオブジェクトが生成されるごとにオブジェクト固有で、オブジェクトの中では永続的に使える変数です。これに対し、**self.** を付けない変数は、関数と同様ローカル変数として扱われ、メソッドの処理が終わると捨てられます。
- メソッド **do_operation()** の定義（8～12行）：これは明示的に呼び出して使用するメソッドで、指定された演算を第一項と第二項を対象におこない、結果を書きこみます。引数 **self** が付されていること、処理の中で **self.** を付けてインスタンス変数を操作していることに留意してください。

- メインプログラム(14行目以降)。端末から文字入力を受ける形で実行する電卓プログラムです。無限ループで記述しているので Ctrl-C で脱出します。
- クラス型オブジェクトの生成(15行目)。クラス型オブジェクトはクラス名を関数のように呼び出して、変数に代入することで行います。

変数 = クラス名()

- クラス型オブジェクトのインスタンス変数やメソッドの操作(18~24行)。クラス型オブジェクトの変数名に「.」でインスタンス変数名やメソッド名などを付けて呼び出します。do_operation() メソッドは定義では引数 self が必要ですが、呼び出しの際には不要であることを確認してください。

演習 33. Dentaku クラスのオブジェクトを複数生成して利用するプログラムを作成してみてください。

演習 34. Dentaku クラスを乗算、除算も扱えるように拡張しなさい。ただし、除算は整数商でかまいません。

演習 35. tkinter で作成した電卓プログラムについて、Dentaku クラスを利用するように改造しなさい

9.4 クラスの変数とアクセスの制限

先に Python のプログラムではプログラム全体に有効なグローバル変数と関数内で実行中に限り有効なローカル変数があることを述べましたが、クラスについてはこのほか、クラス変数とインスタンス変数について知っておく必要があります。

- **クラス変数**: クラスの定義でメソッドの定義の外側で宣言される変数はクラスで共通な変数として働きます。クラス名.変数名という形でクラス型オブジェクトを生成しなくても参照できます。
- **インスタンス変数**: 生成されたインスタンスごとに独立した変数として扱われる変数です。メソッドの定義の中で self. を前につけて宣言、参照します。生成されたインスタンスについては、これを代入した変数の「.」と変数名を付けて参照します。

Python はあまり強力な変数の保護機能をもちません。クラス変数、インスタンス変数とも外から参照も書き換えも可能です。クラスの外からのアクセスを制限する方法として「アンダースコア 2つで始まる変数」の利用があります。このような変数はクラス内のメソッドからはアクセスできますが、クラス外から直接、操作はできません。

プログラム 25 クラス変数とインスタンス変数

| 行 | ソースコード | 説明 |
|----|--|----------------------------------|
| 1 | # クラスの練習 | |
| 2 | class MyClass(): | クラス定義 |
| 3 | # 以下はクラス変数 | |
| 4 | a = "マイクラス" | |
| 5 | __b = 0 | __b は アクセス保護される変数 |
| 6 | | |
| 7 | # 以下は生成する際に呼ばれる関数, mydata の初期値を | |
| 8 | # 引数で与える | |
| 9 | def __init__(self, data): | 引数 data をとる初期化メソッド |
| 10 | # __number はインスタンスの通し番号 | |
| 11 | self.__number = MyClass.__b | |
| 12 | self.mydata = data | |
| 13 | print("MyClass Object is created, number: ", | |
| | self.__number) | |
| 14 | # クラス変数を 1 増やす | |
| 15 | MyClass.__b += 1 | |
| 16 | | |
| 17 | # 通し番号を表示するメソッド | |
| 18 | def show_number(self): | |
| 19 | print(self.__number) | |
| 20 | | |
| 21 | # | |
| 22 | # ここからメインプログラム | 24 行の指示によりモジュールでインポートされたときに実行しない |
| 23 | # | |
| 24 | if __name__ == "__main__": | |
| 25 | print("MyClass のクラス変数 a: ", MyClass.a) | |
| 26 | | |
| 27 | instance1 = MyClass(1) | |
| 28 | instance2 = MyClass(10) | |
| 29 | | |

```

30     instance1.show_number()
31     instance2.show_number()
32
33     print("mydata of instance1: ", instance1.mydata)
34     print("mydata of instance2: ", instance2.mydata)
35     instance1.mydata += 1
36     instance2.mydata += 2
37     print("mydata of instance1: ", instance1.mydata)
38     print("mydata of instance2: ", instance2.mydata)

```

このプログラムを実行すると以下を得ます。クラス変数を用いてインスタンスに通し番号が付されていることや、インスタンス変数 `mydata` がインスタンスごとに独立であること、メインプログラムから直接アクセスできることが分かります。

```
===== RESTART: C:/Users/一/Documents/Python Scripts/class_ex.py =====
 MyClass のクラス変数 a: マイクラス
 MyClass Object is created, number: 0
 MyClass Object is created, number: 1
 0
 1
 mydata of instance1: 1
 mydata of instance2: 10
 mydata of instance1: 2
 mydata of instance2: 12
```

またシェルで以下の操作をするとエラーが生じます。「`__`」で始まるインスタンス変数が保護されていることが分かります。

```
>>> print(instance1.__number)
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    print(instance1.__number)
AttributeError: 'MyClass' object has no attribute '__number'
```

なお、ソースコード中の

```
if __name__ == "__main__":
```

という表記はこのソースコードがメインプログラムとして実行された場合について

のみ実行するという指示です。ソースコードはモジュールとしてインポートすることも可能ですが、その場合はこの部分以降は実行されません。

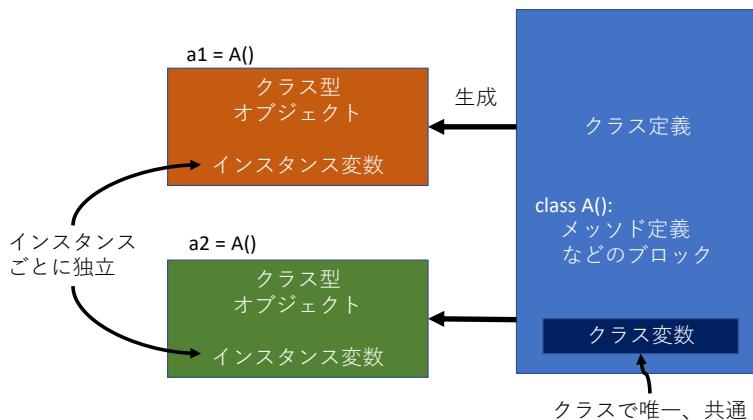


図 38 クラス変数とインスタンス変数

9.5 繙承

クラスを使ったプログラムの記述で重要なものに継承があります。本書では詳細は述べませんが、例えば `tkinter` の実装方法として紹介した例では `tkinter` の `Frame` クラスを継承したクラス `MyFrame` を定義して利用しました。これは `Frame` クラスとしての機能を継承したうえで、`Frame` 上のウィジェットの定義などを加える形で作られています。

9.6 インスタンスを起点にクラスを設計する

複雑なことを行うプログラムを作成する際にはクラスの利用は強力なツールになります。先に見たようにクラスはクラス型のオブジェクト（インスタンス）を生成する「型」という位置づけになります。しかしながら「型」からものごとを考えるのは難しいものです。

実際にクラスを設計する際には次のように具体的なインスタンスから考えるとよいでしょう。

- オブジェクトとしてまとめて取り扱いたいデータ（インスタンス変数の候補）と操作（メソッドの候補）を考える。
- このようなオブジェクトについてオブジェクトごとにクラスを考える。

- クラスをより広く使えるようにする
 - 複数のオブジェクト（クラス）で内容が同じなら、同じクラスでよい。
 - 値の設定などが異なるだけならば、値をインスタンス変数にして、オブジェクトの生成の際に引数で与えることを考える。
 - 共通のメソッドと個別のメソッドが混在する場合は、継承などを考える。

10. リスト

10.1 本章の学習の目標

ここまででは主にデータとしては、单一の数値や单一の文字列を扱ってきました。Python では複数のデータを一括して扱う方法がいくつかありますが「リスト」はその代表です¹。本章では Python におけるリストの扱いについて以下のことを学びます。

1. リストとはどのようなものかを知る
2. リストの生成法を知る
3. リストの要素へのアクセス方法を知る
4. リストを for 文で操作することを知る
5. リストを要素とするリスト（リストのリスト）で表を表し、二重の for 文で操作することを知る
6. リストの代入とコピーについて知る

10.2 Python Shell を用いた学習

本章の内容は短いコードが多く、エディタで編集して実行するよりは Python Shell で入力しながら動作を確認してゆくことで学習を効率的に進めることができます。ただし、以下に注意してください。

- Python Shell は 1 行ずつ処理しますので、複数行をコピー & ペーストで実行することはできません。1 行ずつ入力してください。
 - for 文などブロックを要求する行を入力すると、自動的にインデントしてブロックの入力を待ちます。ブロックを入力後、空行を 1 行入れると実行されます。
- 以下のプログラムで練習してみてください。

```
a = [1, 2, 3]
for d in a:
    print(d)
```

また、紙数の節約のため赤字の入力と青字の出力を続けて記載している箇所があります。Python Shell のプロンプトなどは省略していますが、赤字の部分を入力し、

¹ここまで複合的なデータの扱いとしてはクラスがあります。また Python にはリストと似た複合的なデータの扱い方としてタブルと辞書がありますが、本書では紹介しません。

青字の表記と実際の出力を確認するという形で学習を進めてください。

10.3 リストとは

日常生活では「買い物リスト」といえば、複数の買うべきものを書き出したメモを指します。これと同じように、複数のデータを一括して扱う Python の仕掛けがリスト（list）です。複数のデータに順番をつけた上で一つのものとして扱えるようになります。例えば

`a = [5, 1, 3, 4]`

と入力し

`print(a)`

とすればリスト全体を

`[5, 1, 3, 4]`

を

`print(a[0])`

とすれば 0 番目の要素

`5`

を

`print(a[2])`

とすれば 3 番目の要素

`3`

が表示されます。

10.4 リストの生成

10.4.1 要素を指定した生成

リストの生成は[] 内に「,」で区切って具体的に要素を書く形で

`a = [5, 1, 3, 4]`

とか

`b = ['三条', '四条', '五条', '七条']`

のように書きます。文字列を要素としてもかまいません。また

`c = 5`

`a = [c, 1, 3, 4]`

のように変数を含んでもかまいません。

10.4.2 range() との組みあわせ

空のリストは list クラスのオブジェクトとして

`e = list()`

でも生成できます。また range() 関数と組み合わせて

`n = list(range(5))`

とすると

`print(n)`

に対して

`[0, 1, 2, 3, 4]`

が得られ、n は 0 ~ 4 を要素とするリストであることが分かります。

10.4.3 文字列からの生成

range() の代わりに文字列からリストを生成することも可能です。

`s = list('abcde')`

と入力し

`print(s)`

とすると

`['a', 'b', 'c', 'd', 'e']`

が得られ、s は文字列 'abcde' を 1 文字ずつ分解したものであることが分かります。

文字列クラスには特定の文字で文字列を区切った単語リストを作る `split()` メソッドがあります。例えば

`t = "a textbook of Python"`

`tlist = t.split()`

とすると

`print(tlist)`

により

`['a', 'textbook', 'of', 'Python']`

が得られ、空白を区切りとした単語のリストが得られていることが分かります。

10.5 リストの要素へのアクセス

リストの要素へのアクセスは [] 内に要素の番号を入れることで行います。

`a = [5, 1, 3, 4]`

```
print(a[0])
```

により

5

が、また

```
a[1] = 2
```

```
print(a)
```

により

[5, 2, 3, 4]

が得られます。

リストの長さは len() 関数で得ます。

```
print(len(a))
```

により

4

が得られます。メソッド a.len() ではないことに注意してください。

10.6 リストを操作する for 文

10.6.1 リストの長さと range 関数を組み合わせる方法

リストの要素を for 文で順に操作するには range(len(a)) で要素の番号を生成することで、例えば

```
a = [5, 1, 3, 4]
```

```
for i in range(len(a)):
```

```
    print(i, a[i])
```

により

0 5

1 1

2 3

3 4

が得られます。

10.6.2 リストを for 文で直接使う方法

また、要素の値を参照するだけでよいなら以下のようないい書き方も可能です。

```
a = [5, 1, 3, 4]
```

```
for d in a:  
    print(a)
```

により

5
1
3
4

が得られます。この場合、d には要素の中身が与えられますので、d の値を変更してもリストの中身は変わりません。

演習 36. 平均値を求める

数値を要素とするリストの平均値は例えば

```
a = [5, 1, 3, 4]  
sum = 0  
for i in range((len(a)):  
    sum += a[i]  
average = sum/len(a)  
print(average)
```

で求められ、実行結果は以下のように得られます。

3.25

演習 37. リストを直接 for 文で利用する形に上のプログラムを書き換えなさい。

10.7 負の添え字とスライス

Python ではリストの添え字の多様な記述が許されています。

10.7.1 負の添え字

添え字が負の場合は後ろから添え字の絶対値だけ数えた要素を指します。

```
a = [5, 1, 3, 4]  
print(a[-1])
```

に対して、最後の

4

が得られます。

10.7.2 スライス

添え字として「先頭番号:終了番号」を与えると、リストの一部を取り出すことができます。これをリストのスライスと言います。終了番号より手前までが含まれることに留意してください。

```
a = [5, 1, 3 4]
```

```
b = a[1:3]
```

```
print(b)
```

により

```
[1, 3]
```

が得られます。

10.8 リストへの追加、結合

リストにはさまざまなメソッドが用意されています。ここではその中でリストに要素を追加する `append()` とリストを結合する `extend()` を紹介します。

10.8.1 `append`

リストの最後に引数で与えられた要素を追加します。

```
a = [5, 1, 3 4]
```

```
a.append(2)
```

```
print(a)
```

これによりリスト `a` の最後に `2` が追加され

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

が表示されます。

10.8.2 `extend`

2つのリストを統合するには `extend()` メソッドを使います。

```
a = [5, 1, 3, 4]
```

```
b = [2, 6]
```

```
a.extend(b)
```

```
print(a)
```

により、リスト `a` の後ろにリスト `b` の内容が追加され、以下が表示されます。

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

注意：以下ではリスト b そのものがリスト a の最後の要素として追加されてしまいます。

```
a = [5, 1, 3, 4]
b = [2, 6]
a.append(b)
print(a)
```

リスト a の後ろにリスト b そのものが追加され、以下が表示されます。

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

10.9 リストのリスト

Python のリストの要素は先の例に見たように数値や文字列だけでなく、リストでも構いません。これにより、表のようなデータを作成することができます。

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

リスト a 全体を表示

```
print(a)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

リスト a の最初の要素を表示

```
print(a[0])
[1, 2, 3]
```

リスト a の最初の要素の 1 番目の要素を表示

```
print(a[0][1])
2
```

表形式のデータを「リスト」の「リスト」として表した場合、すべての要素を参照するには for 文を入れ子にします。

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
sum = 0
for i in range(len(a)):
    for j in range(len(a[i])):
        sum += a[i][j]
```

```
print(a)
```

により以下を得ます

45

この例でリストの要素の値を参照するだけですので以下のように書くこともでき

ます。for 文の目標となる変数には番号でなくリストの要素そのものが設定されますので、意味が分かりやすいように row, element を使いました。

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
sum = 0  
for row in a:  
    for element in row:  
        sum += element  
print(a)
```

10.10 内包表記

値が添え字の 2 乗のリスト

[0, 1, 4, 9, 16]

を作ることを考えます。上のように直接、明示的に書いても構いませんし

```
a = []  
for i in range(5):  
    a.append(i*i)
```

と書いても構いません。このほか Python では for 文をリストの中に書く内包表記と呼ばれる使い方があります。

```
a = [i*i for i in range(5)]
```

10.11 リストの代入と複製

まず以下のプログラムを実行してみましょう。

```
a = [1, 2, 3]  
b = a  
print(a)  
print(b)
```

次のような結果が得られるはずです。

```
[1, 2, 3]  
[1, 2, 3]
```

それでは続けて以下のようなプログラムを実行したら結果はどのようになるか、予

想してください。

```
b[0] = 0
a[1] = 0
print(a)
print(b)
```

結果は

```
[1, 0, 3]
[0, 2, 3]
```

とはならず

```
[0, 0, 3]
[0, 0, 3]
```

となります。これは変数 **a** も **b** も全く同一のリストを指しているからで、実際

```
print(id(a), id(b))
```

とすると（動作状況により値は異なりますが）、**a** も **b** も同じ **id** を持っていることが分かります。

Python では変数はデータそのものを持つのではなく、「データの所在」を持っています。したがって

```
b = a
```

で **b** に代入されるものは **a** の表すデータそのものではなく、**a** の表すデータ（リスト）の所在になります。このため、**b** や **a** の要素への代入は同じリストを操作してしまうのです。

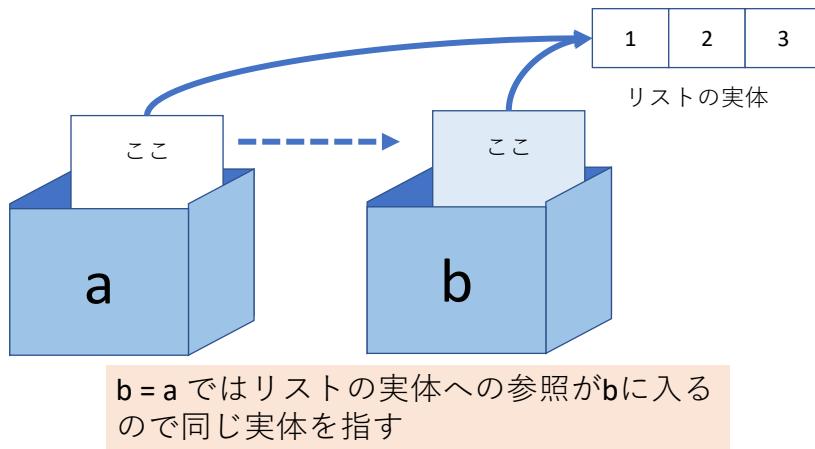


図 39 リストの代入

もし、**b** を **a** と独立に操作したいなら、リストの場合は明示的にコピーを作成して代入する必要があります。

b = a.copy()

同様の問題は関数の引数にリストを渡した場合にも生じます。

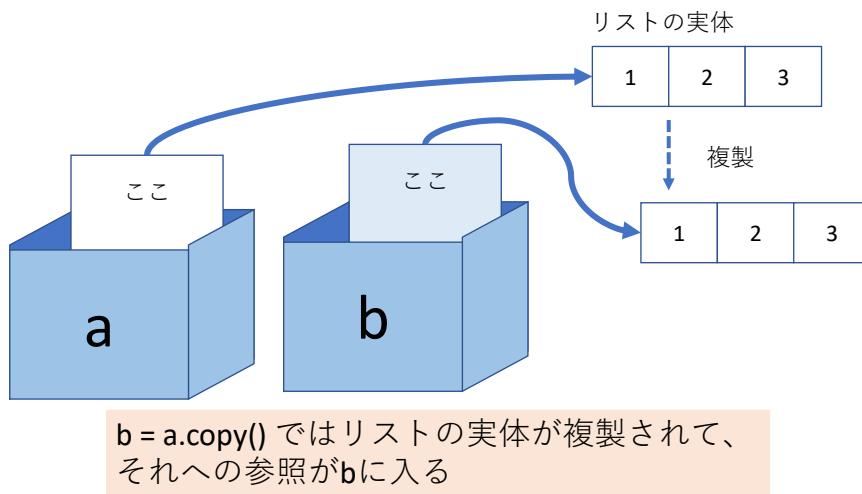


図 40 リストのコピー作成と代入

10.12 イミュータブルとミュータブル

次のようなコードではプログラムの動作は皆さんの予測と違和感がないはずです。

```
a = 1
b = a
b = 2
print(a, b)
```

Python のデータの扱いの重要な概念として「イミュータブル」と「ミュータブル」があります。

10.12.1 数値や文字列はイミュータブル（変更不能）なオブジェクト

Python では数値や文字列は値を変更することが不可能な「イミュータブル」なオブジェクトとして扱われます。このため、上のプログラムの 3 行目の `b = 2` は `b` が指しているデータ（値は 1）を書き換えるのではなく、2 というデータを別途作成して、その所在を `b` に代入するのです。実際、値の所在を以下のプログラムで見てみると

```
a = 1
b = a
```

```
print(id(a), id(b))  
b = 2  
print(id(a), id(b))
```

を実行すると

```
>>> a = 1  
>>> b = a  
>>> print(id(a), id(b))  
1434938848 1434938848  
>>> b = 2  
>>> print(id(a), id(b))  
1434938848 1434938880
```

となり、三行目では同じ場所を指していますが、5行目では異なる場所を指します。

10.12.2 リストはミュータブルなオブジェクト

これに対してリストはその要素などの変更を許す「ミュータブル」なオブジェクトとして扱われます。このため、同じリストへの参照をもつ2つの変数（a, b）があるとき、a や b の要素への変更は別の変数の指す内容にも反映されてしまうのです。

10.13 浅いコピー、深いコピー

リストに関して、さらに厄介な話で恐縮ですが以下のプログラムで変数 b は何を指すでしょうか。

```
a = [[1, 2], [3, 4]]  
b = a.copy()
```

試しに以下の操作を続けて行ってみましょう

```
b.append([5, 6])  
print(a)  
[[1, 2], [3, 4]]  
print(b)  
[[1, 2], [3, 4], [5, 6]]
```

たしかに b は a のコピーを代入したので b への append() は a とは独立に行えています。それでは

```
b[0][0] = 0
print(a)
[[0, 2], [3, 4]]
print(b)
[[0, 2], [3, 4], [5, 6]]
```

今度は `b` の`[0][0]` 要素への代入が `a` の要素に反映されています。`a[0]` と `b[0]` が何を指すか調べてみると

```
print(id(a[0]), id(b[0]))
```

に対して

3188920520008 3188920520008

となっており、同じオブジェクトを参照しています。

これは `copy()` メソッドが対象となるリスト（コピー元）とは別のリスト（コピー先）を用意して、コピー元の各要素について、その所在をコピー先に写し取っていったことによります。したがって、要素がリストの場合はそのコピーが作られていく訳ではありません。このようなコピーを「浅いコピー（shallow copy）」と呼び、要素までコピーを作成することを「深いコピー（deep copy）」と呼びます。

11. ファイル入出力

11.1 本章の学習の目標

1. Python で扱うファイルとしてテキストファイルについて知る
2. CSV 形式のファイルを通じて Python での計算結果を表計算ソフトで扱うこと を知る。
3. Python でのテキストファイルの読み書きについて知る。
4. tkinter での filedialog について知る。

11.2 データを永続的に利用するには

これまでのプログラムではプログラム内の変数に設定されたデータはプログラムが稼働している間だけ保持され、プログラムが終了すると消されてしまいます。また、プログラムへの入出力は GUI にせよ CUI にせよ、人が手で入力し、結果は人が読む形で利用していました。

プログラムでデータを永続的に利用するには、データをプログラムの外側で保存できる形で書き出したり、読み込んだりする必要があります。その候補としては

- コンピュータ上のファイル
- コンピュータ上のデータベース
- ネットワーク上のサービス

などがありますが、ここでは基礎となるコンピュータ上のファイルの操作について学びます。

11.3 ファイルについて

11.3.1 ファイルパス

コンピュータでのファイル扱いは Windows, macOS, linux などのオペレーティングシステムが管理しています。これらの OS では階層的なフォルダ(ディレクトリ)構造をとっており、ファイルの所在はフォルダ構成上で特定するようになっています。ファイルがどこにあるかを記述した文字列は「ファイルパス(file path)」と呼ばれ、「階層的なフォルダ構成の表記」と「ファイル名」が結合した形をとります。

具体的な例を挙げれば Windows では

M:¥documents¥python scripts¥ex1.py

といったものがファイルパスです。ここで

M: ドライブ名（ディスク装置やファイルサーバに対応します）

¥document¥python_scripts: フォルダのパス

ex1.py: ファイル名

.py: ファイル名の . 以降を特にファイルの種類を表す「拡張子」と言います。

なお Windows ではフォルダの区切りを表す文字として「¥」（日本語環境では¥で表示され、それ以外では逆スラッシュ(\)）が使われます。

ドライブ名からフォルダをすべて表記したものを「フルパス」と言います。フルパスならば、そのコンピュータ上のファイルを唯一に特定できます。

これ以外に、「現在、作業中のフォルダ」というものが設定されていて、そこからの差分だけを記述したものを「相対パス」と言います。例えば、作業中のフォルダが

M:¥documents¥python scripts

であれば相対パス表記

ex1.py

は

M:¥documents¥python scripts¥ex1.py

を意味します。

11.3.2 テキストファイル

テキストファイルとは「文字コード（と改行などの記号）で書かれたファイル」で、エディタなどを使えば人にも読み書きできるファイルの形式を指します。例えば Python のソースコードや電子メールのメッセージはテキストファイルです。

これに対してコンピュータの内部形式のデータで構成されたファイルを「バイナリファイル」と呼びます。「バイナリ」とは「2進数の」という意味です。バイナリファイルはコンピュータ内部の形式をそのままファイルに書き出したもので、内部形式であるため数値などは精度を失わない、データ量が少ないなどのメリットがありますが、ファイル内容の記述などがなければ何が書かれているのかは分かりません。

本章では Python でテキストファイルを読み書きすることを学びます。

11.3.3 CSV 形式

Python で作成したプログラムを他のツールと連携して利用できるとあまり手間をかけずに応用範囲が広がります。このためには簡単にデータを扱える形式として **CSV (Comma Separated Value)** 形式があります。これはテキストファイルの一形式で各行が

データ 1, データ 2, データ 3

などのようにデータとデータの間をカンマ「,」で区切った形式です。この形式のファイルのファイル名に `.csv` という拡張子を付けておけば Excel などの表計算ソフトで読み込むことができ、グラフ作成などが簡単に行えます。

CSV 形式のデータの出力は比較的簡単です。他方で読み込みは「カンマや改行を含む文字列の扱い」など面倒な問題もあり、データの内容によってはライブラリの活用などを考えるほうがよいでしょう。

11.3.4 文字コードの問題

日本語の文字コードは歴史的な経緯で、複数併存しており、しかも扱う OS によって使われる文字コードが異なっています。例えば日本語のファイル名に使われる文字コードは以下のようになっています。

Mac, Linux: Unicode

Windows: Shift-JIS

テキストファイルでは上記の文字コードの違いに加え改行を表すコードも違います。

Python 3 は内部では `Unicode` の表現方法の 1 つである `UTF-8` で扱います。IDLE で作成した Python のプログラム（スクリプト）も `UTF-8` でコード化されて保存されています。1 つの OS 内で Python を実行する場合は OS による差異を Python が適宜、調整してくれますのであまり気にしなくてよいのですが、異なる OS でプログラムを動作させる場合には注意が必要です。

11.3.5 エラー処理

ファイルの入出力ではエラー処理は極めて重要です。これはファイルやファイルシステム、読み込むデータの内容はプログラムでは統制できないためです。ファイルを開こうとしたら、ファイルやフォルダが存在しなかったり、書き込み権限がなかったり、書いている途中でディスク領域が不足したりなど、さまざまなことが生

じる可能性があることを意識しなければなりません。

11.4 まずは動かしてみよう

11.4.1 ソースコード

プログラム 26 ファイル入出力の例題

| 行 | ソースコード | 説明 |
|----|--|-----------------------------|
| 1 | # 今のワーキングディレクトリ（作業中のフォルダ） | |
| 2 | # を調べるために | |
| 3 | OS モジュールを import します | |
| 4 | | |
| 5 | import os | |
| 6 | | |
| 7 | # 今のワーキングディレクトリを得て画面に表示します | |
| 8 | print(os.getcwd()) | |
| 9 | | |
| 10 | # 日本語ファイル.txt という名称のファイルを作成し、内 容を書き出します | |
| 11 | | |
| 12 | f = open('日本語ファイル.txt','w') | |
| 13 | f.write('日本語¥n 日本語¥n 日本語¥n') | mac では「¥」で はなく「\」を入 力 |
| 14 | f.close() | |
| 15 | | |
| 16 | # 日本語ファイル.txt を読み込み用に open して、その内 容を表示します | |
| 17 | f = open('日本語ファイル.txt','r') | |
| 18 | s = f.read() | |
| 19 | f.close() | |
| | print(s) | |

11.4.2 プログラムのポイント

- ・ 今の作業フォルダ（カレントワーキングディレクトリ）を知る(7行目)
- ・ 「日本語ファイル.txt」という名前のファイルを書き込み用(w)に開き、以後 `f` という変数で扱う。相対パスとして表記されているので作業フォルダにこの名前で作成される。(12行)
- ・ ファイルへの文字列の書き出し (13行) 「¥n」はこの2文字で「改行」を意味します。
- ・ 書き出し用のファイルを閉じる(14行)
- ・ 同名のファイルを読み出し(r)用に開く。(16行)
- ・ ファイルの内容を変数 `s` にすべて読み出す。(17行)
- ・ ファイルを閉じる(18行)
- ・ 読み出したデータ（テキスト）の出力(19行)

11.5 Python でのファイルの読み書き

11.5.1 open 関数の利用

以下の手順でファイルを操作します。

1. `open` 関数でファイルを開き、返り値でファイルオブジェクトを得る。

```
file = open(ファイル名,モード)
```

モードは読む "r" 、書く "w" など。上の例では返り値を変数 `file` に代入しています。

なお、Python では特に指定しなければテキストファイルの文字コードは稼働している OS の標準の文字コードを想定します、明示的に文字コードを指定するには例えば以下のように `encoding` 引数を設定します。

```
file = open(ファイル名,モード, encoding="utf-8")
```

ファイルを開くのに失敗した場合は `IOError` という例外を発生させます。

2. ファイルオブジェクトへの読み書き

(ア) ファイルオブジェクトから `read()` メソッドで読みこむ

```
s = file.read()
```

上の例ではテキストファイルから全体を文字列として読んで変数 `s` に代入しています。

(イ) ファイルオブジェクトに `write()` メソッドで書き込む

```
file.write(s)
```

上の例では文字列型のデータ `s` をファイルに書き込みます。

ファイルを閉じるまで同様にして追記可能です。

3. ファイルを閉じる

```
file.close()
```

注意：`open` は組み込み関数、`read, write, close` はファイルオブジェクトのメソッド

上の例ではファイルの内容をすべて一括して読み込みました。1行だけ読み込むには `readline()` メソッドを使います。また、以下のように `for` 文でファイルの内容を1行ずつ処理することも可能です¹。

```
file = open("ファイル名 Q,"r")
for line in file:
    一行ずつ処理するブロック
```

11.5.2 `with` 文の利用—`close()` の自動化

`open()` 関数で開いたファイルは `close()` メソッドで閉じる必要がありますが、以下の理由で `close()` が行われない場合があります。

- 単純に `close()` メソッドを呼び忘れている。
- エラーなどで `close()` メソッドを書いている箇所が実行されない。

これらを避けるために Python では `with` 文が用意されており、`with` 文で開いたファイルはブロック終了後に自動的に閉じられます。

```
with open(ファイル名など open 関数の引数) as ファイルオブジェクト用変数:
    ファイルを操作するブロック
```

¹ Python の `for` 文は `range()` 関数、文字列（1文字ずつ）、リスト（要素ごと）など様々な対象に適用できますが、これらは「繰り返し処理が可能な対象」としてイテレータという性格を与えられているからです。ファイルオブジェクトも「一行ずつ」という形でイテレータとして使えます。

11.6 例題1 波の近似

11.6.1 例題のポイント

- ・ ファイルパスを正確に端末から入力するのは面倒なので tkinter (の filedialog だけ) を使います。
- ・ 計算結果を csv 形式で出力して、表計算ソフトと連動します。
- ・ 例題として周期関数を三角関数の和で表現する例を用います。

11.6.2 周期関数の三角関数の和での近似

周期関数（ある周期で値が繰り返す関数）はその周期の整数倍の正弦関数(sin)と余弦関数(cos)の和で近似できることが知られています。のこぎり波（鋸歯状波、鋸の歯のような波）は以下のように近似できることが知られています（コラム「フーリエ級数」も参照してください）。

$$f(x) = \frac{\sin(x)}{1} + \frac{\sin(2x)}{2} + \frac{\sin(3x)}{3} + \frac{\sin(4x)}{4} \dots$$

下の図は第1～第5項までの和をプロットしたもの。

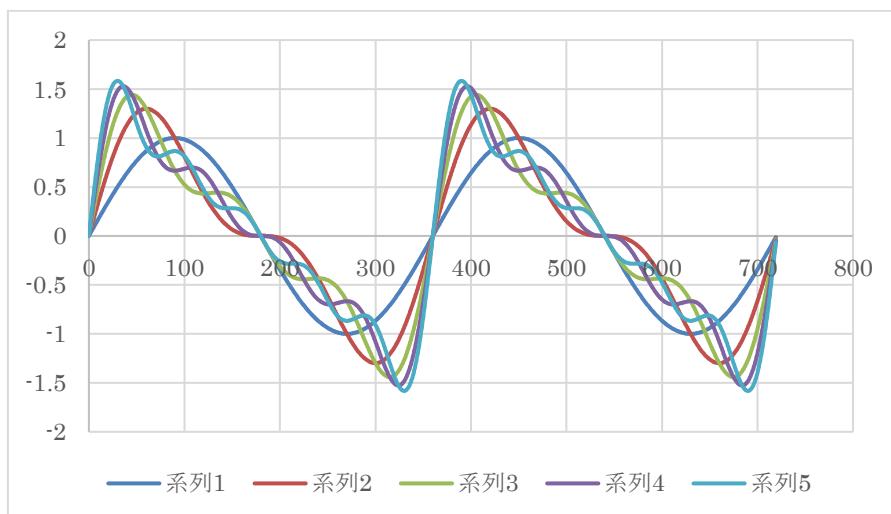


図 41 のこぎり波の三角関数の和での近似

上記は簡単のため、各項の符号を揃えていますが、原点を傾き正で通るのこぎり波については以下のように交互に符号が変わります。

$$f(x) = \frac{\sin(x)}{1} - \frac{\sin(2x)}{2} + \frac{\sin(3x)}{3} - \frac{\sin(4x)}{4} \dots$$

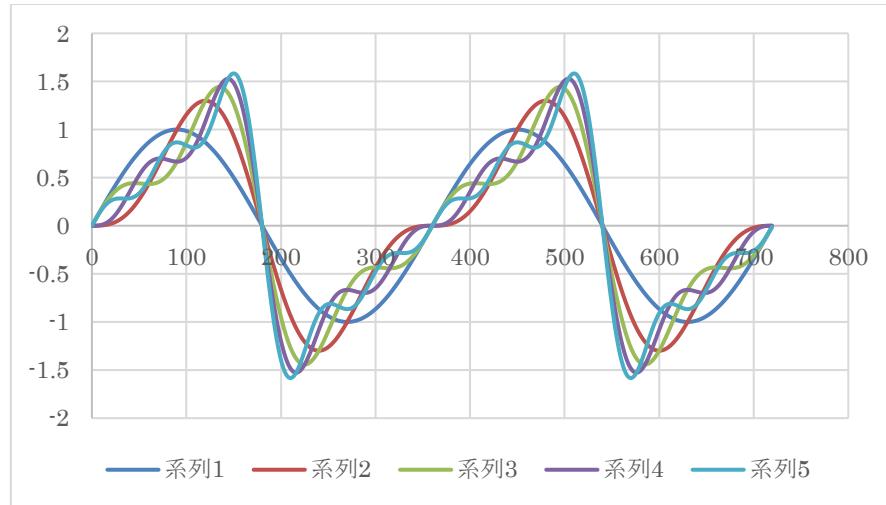


図 42 のこぎり波の三角関数の和での近似（原点で傾きが正の場合）

また、振幅が 1（最大、最小値が ± 1 ）ののこぎり波は全体に係数 $(2/\pi)$ がかかります。

11.6.3 ソースコード

プログラム 27 のこぎり波の三角関数の和での近似

| 行 | ソースコード | 説明 |
|----|---|---------------------|
| 1 | import tkinter as tk | |
| 2 | import tkinter.filedialog | filedialog もインポート |
| 3 | import math | |
| 4 | # | |
| 5 | # tkinter の filedialog だけを利用する例 | |
| 6 | # | |
| 7 | # root ウィンドウは withdraw() メソッドを読んで隠す | |
| 8 | root = tk.Tk() | |
| 9 | root.withdraw() | |
| 10 | # | |
| 11 | # 書き出し用の filedialog を読んでファイル名を得る | ファイル名をダイアログで得て戻ります。 |
| 12 | # | |
| 13 | filename = tkinter.filedialog.asksaveasfilename() | |
| 14 | # | |
| 15 | # ファイル名がもらえなければ終了 | |
| 16 | # | pass は特に処理 |

```

17 if filename:
18     pass
19 else:
20     print("No file specified")
21     exit()
22 #
23 # 正弦波の重ね合わせで鋸波を近似する
24 #
25 # w = sin(t) + sin(2t)/2 + sin(3t)/3 + sin(4t)/4 ...
26 #
27 # 2 周期分、全体は 1000 ステップで、高調波は 5 番目まで
28 #
29 cycles = 2
30 steps = 1000
31 harmonics = 5
32 # ファイルが開けないときのエラー対応
33 try:
34     # ファイルを開く
35     with open(filename,'w') as file:
36         for i in range(steps):
37             angle_in_degree = 360*cycles*i/steps
38             angle = math.radians(angle_in_degree)
39             s = str(angle_in_degree)
40             w = 0
41             for i in range(1,harmonics+1):
42                 w += math.sin(angle*(i))/i
43                 s = s+", "+ str(w)
44             #     print(s)
45             file.write(s+"\n")
46             print("Writing to file "+ filename + " is finished")
47 except IOError:
48     print("Unable to open file")

```

理をしない命令

角度を文字列に
和 w を「,」を前
において s に追
加
改行「\n」を加
えて書き出し

11.6.4 プログラムのポイント

1) tkinter の filedialog の利用（1～21 行）

- Windows のアプリケーションなどでファイルを開いたり、ファイルに保存したりする際に別ウインドウでファイルを探したり、指定したりできます。tkinter ではこのための仕組みとして filedialog が用意されています。このプログラムでは filedialog の機能だけを使うので tkinter のメインウインドウは 8 行目で作成しますが、使わないので 9 行目で見えなくしています。また mainloop() メソッドは呼んでいないことに注意してください。
- filedialog では利用目的によりいくつかの形式があるのですが、ここでは「名前をつけて保存」用の `asksaveasfilename()` メソッドを 13 行目で呼び出し、返り値で得られるファイル名（パス名）を filename で得ます。キャンセルなどの操作の場合、filename には何も入りませんので、if 文の false の場合にプログラムを終了しています。

2) 計算とファイル出力（29～48 行）

- この部分が三角関数の重ね合わせを計算して CSV 形式で出力している部分です。
- ファイルの取り扱いではファイルを開くことができないなどエラーに対応する必要があり 33 行目で try 文でファイル操作するブロックを扱っています。対応するエラー処理は 47, 48 行です。
- 35 行目で with 文でファイルを開いています。filedialog で得たファイル名 filename のファイルを開き、開いたファイルは変数 file で扱っています。
- 計算し、出力する 1 行の内容は

角度、第 1 項、第 2 項までの和、第 3 項までの和、第 4 項までの和、第 5 項までの和です。和は変数 w に計算してゆき、1 行の内容は変数 s に文字列として書き加えています。CSV 形式にするため、w の値を文字列に変換して s に加える際に

`s = s+", "+ str(w)`

と "," を間に挟んでいます。カンマのあとに¹スペースを入れているのは出来たファイルを読みやすくするためです。

¹ 実行できるプログラムの一部をコメントにして、実行を抑止する方法は「コメントアウト」と呼ばれ、プログラムの動作確認などでしばしば用いられます。

- `for` 文で第 5 項までの計算が終わったあと、44, 45 行目でファイルに出力しています。

```
#     print(s)
    file.write(s+"¥n")
```

44 行名はコメントになっていますが、Python Shell で結果を確認したい場合は `#` を削除してください。45 行目では `file.write()` でファイルに書き出していますが、1 行の文字列 `s` に「改行」を加えるために「¥n」が付加されています。mac ユーザは「¥」ではなくバックスラッシュ「\」を入力してください。

演習 38. 矩形波（方形波）の近似

矩形波（方形波）（ ± 1 の値を交互にとる周期関数）は以下のように三角関数で近似できます。¹

$$f(x) = \frac{\sin(x)}{1} + \frac{\sin(3x)}{3} + \frac{\sin(5x)}{5} + \frac{\sin(7x)}{7} \dots$$

例題と同様の方法で方形波を三角関数での近似を計算し、`csv` ファイルに出力した後、表計算ソフトでグラフを作成してください。

演習 39. 例題 1 のリストを使った実装

例題 1 のプログラムでは計算結果は逐次、文字列として結合し、1 行ごとにファイルに書き出しています。これを以下のように計算と出力を分離した形で再実装してください。

- 計算結果はリストを使ってリスト上に書き込む。
- 計算の終了後、そのリストを参照する形で例題 1 と同じ形式の CSV ファイルを書き出す。

なお、リストの構成法として、以下の 2 通りの考え方があります。どちらの実装法でも構いません。

¹ 振幅 1 の矩形波については全体に係数 $4/\pi$ がかかります。

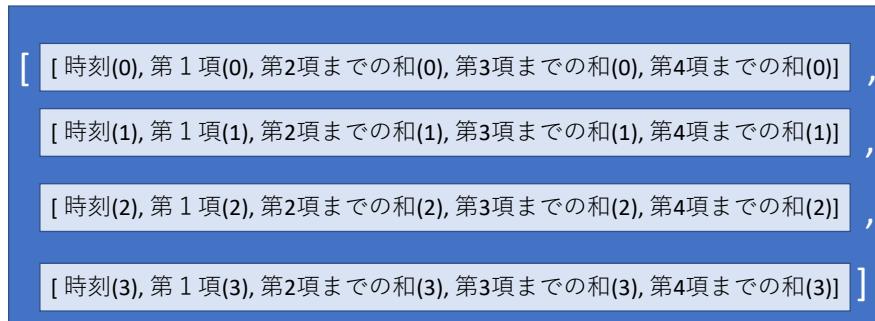


図 43 「各時刻のデータのリスト」のリストとして扱う

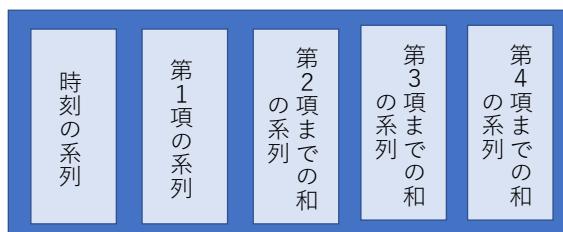


図 44 「各系列のリスト」のリストとして扱う

11.7 例題 2

tkinter を用いれば簡単なテキストエディタを作ることができます。tkinter の機能としてはこれまでに学んだもののほか、メッセージ表示用のダイアログである messagebox, filedialog のファイル読み込み用のメソッド、tkinter の Menu ウィジェット、Text ウィジェットなどを使っています。ジオメトリマネージャは構成が単純なため grid ではなく pack を使っています。

また、ファイルの漢字コードは特に指定していないので Python では OS ごとの標準的コードを想定します。Windows では Shift-JIS コード(cp932)が使われているとして扱われます。

プログラム 28 tkinter を用いた簡単なテキストエディタ

| 行 | ソースコード |
|---|---------------------------|
| 1 | import tkinter as tk |
| 2 | import tkinter.messagebox |

```
3 import tkinter.filedialog
4 # messagebox, filedialog は明示的なインポートが必要
5 #
6 # tk.Frame を継承した MyFrame というクラスを作り
7 # その中でウィジェットやコールバック関数（メソッド）を
8 # 設定する。tkinter をつかう定番
9 #
10 class MyFrame(tk.Frame):
11     # __init__ はクラスオブジェクトを作る際の初期化メソッド
12     def __init__(self, master = None):
13         super().__init__(master)
14         self.master.title('Simple Editor')
15
16     # メニューを作る menubar -> filemenu -> Open, Save as, Exit
17     menubar = tk.Menu(self)
18     filemenu = tk.Menu(menubar, tearoff = 0)
19     filemenu.add_command(label = "Open", command = self.openfile)
20     filemenu.add_command(label = "Save as...", command = self.saveas)
21     filemenu.add_command(label = "Exit", command = self.master.destroy)
22     menubar.add_cascade(label = "File", menu = filemenu)
23     self.master.config(menu = menubar)
24
25     # 編集用 Text ウィジェットをクラスの変数 editbox としてつくる
26     self.editbox = tk.Text(self)
27     self.editbox.pack()
28
29     # ファイルを開くメソッド、関数とちがい self という引数が必要
30     def openfile(self):
31         # filedialog でファイル名を得る
32         filename = tkinter.filedialog.askopenfilename()
33         # filename が空でなければ処理
34         if filename:
35             tkinter.messagebox.showinfo("Filename","Open: "+filename)
36         # with 文で file という変数でファイルを開く
```

```
35         with open(filename,'r') as file:  
36             text = file.read()  
37             # Text ウィジェット editbox にファイル内容を設定  
38             self.editbox.delete('1.0',tk.END)  
39             self.editbox.insert('1.0',text)  
40         else:  
41             tkinter.messagebox.showinfo("Filename","Canceled")  
42  
43             # ファイルに保存するメソッド  
44             def saveas(self):  
45                 # with 文で file という変数でファイルを開く  
46                 filename = tkinter.filedialog.asksaveasfilename()  
47                 if filename:  
48                     with open(filename,'w') as file:  
49                         text = file.write(self.editbox.get('1.0',tk.END))  
50                         tkinter.messagebox.showinfo("Filename","Saved AS:"+filename)  
51                 else:  
52                     tkinter.messagebox.showinfo("Filename","Canceled")  
53  
54             # ここからメインプログラム  
55             root = tk.Tk()  
56             f = MyFrame(root)  
57             f.pack()  
58             f.mainloop()
```

12. 三目並べで学ぶプログラム開発

12.1 本章の学習の目標

この章では三目並べを例に課題を与えられてプログラムを開発することを学びます。

1. 三目並べをプレイすることを分析し、プログラムで表現する必要のある事項を洗い出します。
2. プログラムのテストに備えて棋譜を準備します。
3. プログラムを構成するデータや関数を小さなものから順に作成します。
4. 全体を組み上げて三目並べのプログラムを完成させます。

12.2 プログラムを開発するということ

これまでのいくつかのプログラムを構成してきましたが、何か具体的な課題を与えられてプログラムを作ることは、それ自体が初学者には難しいことのようです。プログラミング言語さまざまな要素を使えるようになると、まとめたプログラムを1から開発することは異なる能力であるからです。例えると、金槌や鋸を使えるからと言って家を建てることができる訳ではありません、家を建てるには、家とはどのような構成になっていて、どのような順番で設計し、施工する必要があるかを知っていなければならぬからです。プログラムも同様です。

12.3 設計手順—コンピュータを使う前にすることがある

プログラムの設計・作成手順は以下のようになります。最初からコンピュータが必要な訳ではありません。

- コンピュータを使う前にすること
 - 実現したいことを言葉で表現する
 - プログラムとして作るべきものを特定する

- ✧ 変数として表現すべきこと
- ✧ 変数のとる値として表現すべきこと
- ✧ 手順（関数）として表現すべきこと
- ✧ 人とのやりとりとして表現すべきこと
- プログラムとして作成する順序を決める
- テストの方法を決める
- ここからコンピュータで作業
 - 他に依存しない部分からプログラム（関数）を作る
 - 作った関数をテストする（単体テスト）
 - 全体をテストする（結合テスト）

12.4 三目並べを例にしたプログラムの設計

12.4.1 三目並べ (tic-tac-toe)

三目並べのルールとゲームの進行を言葉で表現してみてください。

| | | |
|---|---|---|
| ○ | × | |
| | ○ | |
| ○ | | × |

12.4.2 文章の分析

三目並べのルールとゲームの進行を表現した文章から品詞（名詞、動詞）に着目して文章を分析します。以下のような事項が得られます。

- 特定の状態をとる事項（名詞）：変数の候補です
 - 3×3 マスの盤面、手番
- 事項の状態：変数の取り得る値の候補です
 - 各マスの状態（空、○（先手）、×（後手））
 - どちらの手番か
- 状態を調べる動作（関数の候補です）
 - どちらの手番か
 - マスの状態

- 先手勝ち、後手勝ち、引き分け
- 状態を変える動作（関数の候補です）
 - マスに○や×を置く
 - 手番を入れ替える

12.4.3 棋譜の作成—テストの準備として

プログラムを作る前にテスト用の棋譜をいくつか作成しておきましょう。すべての場合をつくすことは難しいですが、以下のようなことを考えて準備します。

- 先手勝ち、後手勝ち、引き分けを含むこと
 - 勝ち方のパターン（縦（3通り）、横（3通り）、斜め（通り））を含むこと
- テストケースを先に用意することは「**テストファースト**」と呼ばれ、プログラムを作り始めること「**コーディングファースト**」より以下のような点で効果があるとされています。
- 後からだとテストケースの作成に手を抜く
 - プログラムの作成中、いつでもテストできる
 - テストすることをコーディングで意識できる

| 手番 | row | column | | row | column | | row | column | |
|-----|-------|--------|-----|-------|--------|-----|-------|--------|-----|
| 1 ○ | 0 | 0 | | 0 | 0 | | 0 | 1 | |
| 2 × | 1 | 1 | | 1 | 0 | | 0 | 0 | |
| 3 ○ | 1 | 0 | | 1 | 1 | | 2 | 1 | |
| 4 × | 2 | 0 | | 2 | 2 | | 1 | 1 | |
| 5 ○ | 0 | 2 | | 0 | 2 | | 2 | 2 | |
| 6 × | 0 | 1 | | 0 | 1 | | 2 | 0 | |
| 7 ○ | 2 | 1 | | 2 | 0 | | 1 | 0 | |
| 8 × | 2 | 2 | | | | | 0 | 2 | |
| 9 ○ | 1 | 2 | | | | | | | |
| 結果 | 引き分け | | | 先手勝ち | | | 後手勝ち | | |
| | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| | 0 1 ○ | 6 × | 5 ○ | 0 1 ○ | 6 × | 5 ○ | 0 2 × | 1 ○ | 8 × |
| | 1 3 ○ | 2 × | 9 ○ | 1 2 × | 3 ○ | | 1 7 ○ | 4 × | |
| | 2 4 × | 7 ○ | 8 × | 2 7 ○ | | 4 × | 2 6 × | 3 ○ | 5 ○ |

図 45 三目並べ、棋譜の例

12.4.4 変数の設計

1) 盤面

- 3×3 の盤面を2重のリスト（要素は整数）で表すことにします。

```
board = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

- 初期状態はすべて 0（空き）
- 値の意味は 0 空き、1 先手（○）、2 後手（×）とします。
- このために定数（Python では値の変更を禁じる方法はありません）を定義します。定数であることを意識するため大文字を使います。手番や結果への利用も考慮して以下のようにします。

```
OPEN = 0
```

```
FIRST = 1
```

```
SECOND = 2
```

```
EVEN = 3
```

2) 手番

- どちらの手番(turn)かを整数変数で表します。初期値は FIRST(先手)
- turn = FIRST
- 値は先の定数 (FIRST, SECOND) を利用し FIRST: 先手と SECOND: 後手

3) 棋譜

- 手番は「先手→後手→先手…」と決まっているので、打った場所の行、列のリストで表現します。
- リストの最後に結果（未定、先手勝ち、後手勝ち、引き分け）を入れることにします¹。
- 棋譜 = [[初手の行, 列], [二手目の行, 列], ..., [最終手の行, 列], [結果]]
- 行, 列はそれぞれ 0, 1, 2 の整数、結果は 0, 1, 2, 3 の整数

12.4.5 盤面と手番に関する関数

状態を「操作」、状態の「検査」、画面に「表示」する、初期化（操作のひとつ）するなどの関数を作ります。

盤面や手番などをグローバル変数として共有するので、関数内で関数外の変数の値を変えるためには `global` 宣言が必要です。以下のようないくつかの関数が必要でしょう。

1) 手番について

- 操作：手番を初期化する

¹ 棋譜を1つのリストで表現したため、リストの要素は「打った場所」と「結果」という異なる意味を持ってします。あまり分かりやすい実装とは言えません。

- 操作：手番を交代する
- 表示：手番を表示する（ための文字列を生成する）

2) 盤面について

- 操作：盤面を初期化する
- 操作：盤面の指定されたマスを指定された手番にマークする
- 検査：盤面の個々のマスの状態を知る
- 検査：盤面がどちらの手番の勝ちであるかを知る
- 検査：盤面がすべて埋まっているかを知る
- 表示：盤面全体を出力する

3) 棋譜について

- 操作：棋譜を用いて対戦を再生する

4) 勝敗判定のアルゴリズム

日頃、皆さんが実際にプレーしているゲームですが、明示的に勝敗判定の方法を書き出すと以下のようになります。

- ある行・列方向にどちらかの手番（以下、 t とします）の勝ちを判定する。
 - もし注目する方向上の 3 つの位置のコマがすべて t ならば勝ち、
 - そうでなければ勝ちではない
- ある方向に勝ちかどうかを判定する
 - 横、あるいは縦方向なら
 - ❖ もし第 0 行目（第 0 列目）が勝ちならば勝ち
 - ❖ そうでなければもし第 1 行目（第 1 列目）が勝ちならば勝ち
 - ❖ そうでなければもし第 2 行目（第 2 列目）が勝ちならば勝ち
 - ❖ そうでなければ勝ちではない
 - 対角あるいは逆対角方向なら、その方向に勝ちならば勝ち
- 上記を使った勝ち判定
 1. もし横方向縦に勝ちならば勝ち。
 2. そうでなければもし縦方向に勝ちならば勝ち。
 3. そうでなければもし対角方向に勝ちならば勝ち。
 4. そうでなければもし逆対角方向に勝ちならば勝ち。
 5. そうでなければ t の勝ちではない

なお、逐次勝敗を判定している盤面では生じませんがランダムに生成した盤面では

先手、後手ともに勝っているという状況があり得ます。

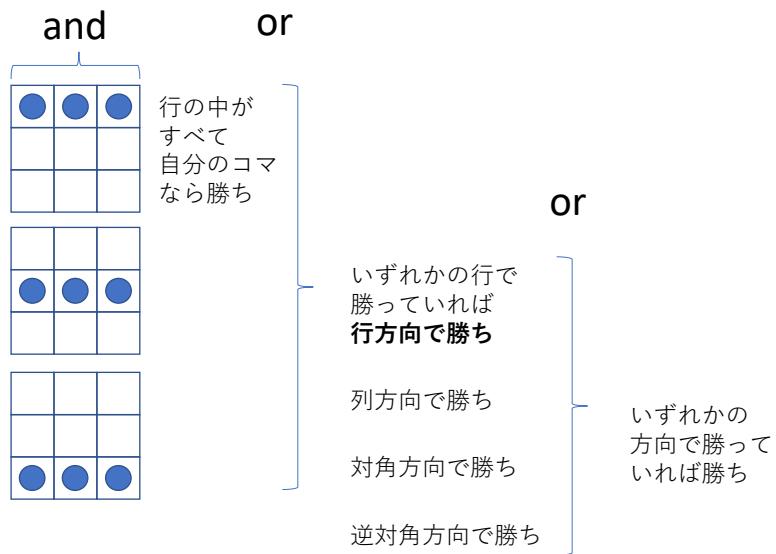


図 46 着目する手番の勝ち判定

● 勝敗を判定する

上の手続きを（関数）を用いて三目並べでは以下のように勝敗を判定できます。

1. 先手が勝っているかどうかを調べ、もし先手が勝っていれば「先手勝ち」
2. そうでなければ後手が勝っているかどうかをしらべ、もし後手が勝っていれば「後手勝ち」
3. そうでなければ、もし盤面に空きがあればまだ「未決着」
4. そうでなければ（盤面に空きがない）ならば「引き分け」

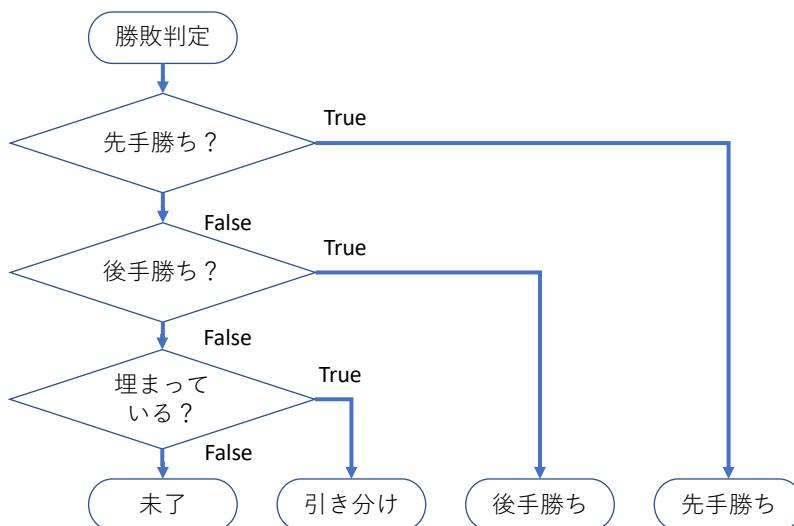


図 47 勝敗判定

12.4.6 ゲームの進行

メインプログラムの流れを考えます。入出力は Python Shell 上での文字での入出力を考えます。先に定めた関数を使って容易に実装できるかを確認します。

- ゲームを初期化する
- 盤面を表示する
- 以下、勝ち負け、引き分けまで繰り返す
 - 正しい入力が得られるまで手番側の入力を促進する
 - ◆ 手番側の入力を得る
 - 盤面を更新する
 - 盤面を表示する
 - 勝ち負け、引き分けを判定する
 - ◆ ゲーム終了なら結果を示して脱出
 - 手番を交代する

12.5 プログラムの実装

1) ソースコードの構成

設計が完了したらプログラムの実装を始めます、Python のソースコードは以下のよ
うな構成にすることを確認しておきます。



図 48 ソースコードの全体構成**2) 実装例 (`tic_tac_toe.py`)**

プログラムの実装例を以下に示します。実装の方針や記法についてポイントを説明します。

- プログラムは図 29 に示した順に実装されています。
- プログラムは全体で 500 行近くありますが、背景を黄色にした部分はテスト用の関数です。
- 関数等に `docstring` を付しています。複数行にわたる `docstring` は ""(シングルクオート 3 つ)で開始し、"" で終わるという記法を用いています。
- 特にインポートしなければならないモジュールはありません。
- 表示するための関数は、「画面に表示する」のではなく、`print()` 関数に渡せる文字列を生成する形をとっています。
- メインプログラムは単に「三目並べ」と表示するだけです。すべての関数が読み込まれた状態になっていますので、Python Shell でテスト用の関数や実際のプレイ用の関数を呼び出すことができます。

プログラム 29 三目並べのプログラム、実装例（その1 グローバル変数）

| 行 | ソースコード |
|----|---|
| 1 | # |
| 2 | # 三目並べ |
| 3 | # |
| 4 | # 特にインポートするモジュールはありません |
| 5 | # |
| 6 | # 定数の定義 |
| 7 | # |
| 8 | # |
| 9 | # play() の中で棋譜を作成する（要完成） |
| 10 | # |
| 11 | '三目並べのプログラムです' |
| 12 | OPEN = 0 |
| 13 | FIRST = 1 |
| 14 | SECOND = 2 |
| 15 | EVEN = 3 |
| 16 | # |
| 17 | # 恒常的な変数 |
| 18 | # |
| 19 | turn = 1 |
| 20 | board = [[0, 0, 0], [0, 0, 0], [0, 0, 0]] |
| 21 | # |
| 22 | # テスト用の棋譜 |
| 23 | # |
| 24 | log1 = [[0, 0], [1, 1], [1, 0], [2, 0], [0, 2], [0, 1], [2, 1], [2, 2], [1, 2], [EVEN]] |
| 25 | log2 = [[0, 0], [1, 0], [1, 1], [2, 2], [0, 1], [2, 0], [FIRST]] |
| 26 | log3 = [[0, 1], [0, 0], [2, 1], [1, 1], [2, 2], [2, 0], [1, 0], [0, 2], [SECOND]] |
| 27 | |

プログラム 30 三目並べのプログラム、実装例（その2 手番関連の関数）

```
28  #
29  # 手番関連の関数
30  #
31  # 手番を文字列に
32  #
33  def show_turn():
34      '手番を示す文字列を返す'
35      if turn == FIRST:
36          return('先手')
37      elif turn == SECOND:
38          return('後手')
39      else:
40          return('手番の値が不適切です')
41  #
42  # 手番の初期化
43  #
44  def init_turn():
45      '手番を初期化する'
46      global turn
47      turn = 1
48  #
49  # 手番の交代
50  #
51  def change_turn():
52      '手番を交代する'
53      global turn
54      if turn == FIRST:
55          turn = SECOND
56      elif turn == SECOND:
57          turn = FIRST
58  #
59  # 手番関連の関数のテスト
60  #
61  def test_turn():
62      '手番をテストする'
63      init_turn()
64      print(show_turn(), "の番です")
65      change_turn()
66      print(show_turn(), "の番です")
67      change_turn()
68      print(show_turn(), "の番です")
69  
```

プログラム 31 三目並べのプログラム、実装例（その 3 盤面関連の関数その 1）

```

70  #
71  # 盤面関連の関数
72  #
73  # 盤面を表示する文字列
74  #
75  def show_board():
76      '盤面を表す文字列を返す'
77      s = ' :0 1 2¥n-----¥n'
78      for i in range(3):
79          s = s + str(i) + ': '
80          for j in range(3):
81              cell = ''
82              if board[i][j] == OPEN:
83                  cell = ' '
84              elif board[i][j] == FIRST:
85                  cell = '0'
86              elif board[i][j] == SECOND:
87                  cell = 'X'
88              else:
89                  cell = '?'
90              s = s + cell + ' '
91      s = s + '¥n'
92      return s
93  #
94  # 盤面の初期化
95  #
96  def init_board():
97      '盤面をすべて空 (OPEN)に初期化する'
98      for i in range(3):
99          for j in range(3):
100             board[i][j] = OPEN
101     #
102    # 盤面の i, j の位置の値を返す
103    #
104    def examine_board(i, j):
105        '盤面の i 行 j 列の値を返す'
106        return board[i][j]
107    #
108    # 盤面の i, j に手番 t を登録、状態を文字列で返す
109    #
110    def set_board(i, j, t):
111        '',
112        盤面の i, j に手番 t を登録、状態を文字列で返す
113        返す値は
114        'ok' 成功
115        'Not empty' 空いている場所ではない
116        'illegal turn' 手番が正しくない
117        'illegal slot' 指定された場所が正しくない
118        '',
119        if (i>=0) and (i<3) and (j>=0) and (j<3):

```

```
120     if (t>0) and (t<3):
121         if examine_board(i, j) == 0:
122             board[i][j] = t
123             return 'OK'
124         else:
125             return 'Not empty'
126         else:
127             return 'illegal turn'
128     else:
129         return 'illegal slot'
130 #
131 # 盤面のテスト関数
132 #
133 def test_board1():
134     '盤面についてのテストプログラムの1つめです'
135     init_board()
136     print(show_board())
137     print(set_board(0, 0, 1))
138     print(show_board())
139     print(set_board(1, 1, 2))
140     print(show_board())
141     print(set_board(1, 1, 1))
142     print(show_board())
```

プログラム 32 三目並べのプログラム、実装例（その 4 盤面関連の関数その 2）

```

143 # 
144 # 水平方向での手番 t の勝ちの判定
145 #
146 def check_board_horizontal(t):
147     '水平方向に手番 t が勝ちであることを判定します'
148     for i in range (3):
149         if (board[i][0] == t) and (board[i][1] == t) and (board[i][2] == t):
150             return True
151     return False
152 #
153 # 垂直方向での手番 t の勝ちの判定
154 #
155 def check_board_vertical(t):
156     '垂直方向に手番 t が勝ちであることを判定します'
157     for j in range (3):
158         if (board[0][j] == t) and (board[1][j] == t) and (board[2][j] == t):
159             return True
160     return False
161 #
162 # 対角方向での手番 t の勝ちの判定
163 #
164 def check_board_diagonal(t):
165     '対角方向に手番 t が勝ちであることを判定します'
166     if (board[0][0] == t) and (board[1][1] == t) and (board[2][2] == t):
167         return True
168     return False
169 #
170 # 逆対角方向での手番 t の勝ちの判定
171 #
172 def check_board_inverse_diagonal(t):
173     '逆対角方向に手番 t が勝ちであることを判定します'
174     if (board[0][2] == t) and (board[1][1] == t) and (board[2][0] == t):
175         return True
176     return False
177 #
178 # 手番 t の勝ちの単純な判定
179 #
180 def is_win_simple(t):
181     '手番 t が勝ちであることを判定します。相手が勝っていることはチェックしません'
182     if check_board_horizontal(t):
183         return True
184     if check_board_vertical(t):
185         return True
186     if check_board_diagonal(t):
187         return True
188     if check_board_inverse_diagonal(t):
189         return True
190     return False
191 #
192 # 相手が勝っていないことを確認しての勝ちの判定

```

```
193 #  
194 def is_win_actual(t):  
195     '手番 t が勝ちであることを判定します。相手が勝っていないことも確認します'  
196     if not is_win_simple(t):  
197         return False  
198     if t==FIRST:  
199         if is_win_simple(SECOND):  
200             return False  
201     else:  
202         if is_win_simple(FIRST):  
203             return False  
204     return True  
205 #  
206 # 盤面が埋まっていることの判定  
207 #  
208 def is_full():  
209     '盤面に空きがないことを確認します'  
210     for i in range(3):  
211         for j in range(3):  
212             if board[i][j] == OPEN:  
213                 return False  
214     return True  
215 #  
216 # 引き分けの判定  
217 #  
218 def is_even():  
219     '盤面が引き分けであることを判定します'  
220     if is_win_simple(FIRST):  
221         return False  
222     if is_win_simple(SECOND):  
223         return False  
224     if not is_full():  
225         return False  
226     return True
```

プログラム 33 三目並べのプログラム、実装例（その 5 盤面のテスト関数その 2）

```
227 #  
228 # 盤面のテスト関数 2 つめ、勝ち判定のテスト  
229 #  
230 def test_board2():  
    '盤面をテストする関数の 2 番目'  
    init_board()  
    board[0][0] = FIRST  
    board[1][0] = FIRST  
    board[2][0] = FIRST  
    print(show_board())  
    print("HORIZONTAL FIRST: ", check_board_horizontal(FIRST))  
    print("HORIZONTAL SECOND: ", check_board_horizontal(SECOND))  
    print("VERTICAL FIRST: ", check_board_vertical(FIRST))  
    print("VERTICAL SECOND: ", check_board_vertical(SECOND))  
    init_board()  
    board[0][0] = SECOND  
    board[1][0] = SECOND  
    board[2][0] = SECOND  
    print(show_board())  
    print("HORIZONTAL FIRST: ", check_board_horizontal(FIRST))  
    print("HORIZONTAL SECOND: ", check_board_horizontal(SECOND))  
    print("VERTICAL FIRST: ", check_board_vertical(FIRST))  
    print("VERTICAL SECOND: ", check_board_vertical(SECOND))  
    init_board()  
    board[0][0] = FIRST  
    board[0][1] = FIRST  
    board[0][2] = FIRST  
    print(show_board())  
    print("HORIZONTAL FIRST: ", check_board_horizontal(FIRST))  
    print("HORIZONTAL SECOND: ", check_board_horizontal(SECOND))  
    print("VERTICAL FIRST: ", check_board_vertical(FIRST))  
    print("VERTICAL SECOND: ", check_board_vertical(SECOND))  
    init_board()  
    board[0][0] = SECOND  
    board[0][1] = SECOND  
    board[0][2] = SECOND  
    print(show_board())  
    print("HORIZONTAL FIRST: ", check_board_horizontal(FIRST))  
    print("HORIZONTAL SECOND: ", check_board_horizontal(SECOND))  
    print("VERTICAL FIRST: ", check_board_vertical(FIRST))  
    print("VERTICAL SECOND: ", check_board_vertical(SECOND))  
    init_board()  
    board[0][0] = FIRST  
    board[1][1] = FIRST  
    board[2][2] = FIRST  
    print(show_board())  
    print("DIAGONAL FIRST: ", check_board_diagonal(FIRST))  
    print("DIAGONAL SECOND: ", check_board_diagonal(SECOND))
```

```
277     print("INV DIAGONAL FIRST: " , check_board_inverse_diagonal(FIRST))
278     print("INV DIAGONAL SECOND: " , check_board_inverse_diagonal(SECOND))
279     init_board()
280     board[0][0] = SECOND
281     board[1][1] = SECOND
282     board[2][2] = SECOND
283     print(show_board())
284     print("DIAGONAL FIRST: " , check_board_diagonal(FIRST))
285     print("DIAGONAL SECOND: " , check_board_diagonal(SECOND))
286     print("INV DIAGONAL FIRST: " , check_board_inverse_diagonal(FIRST))
287     print("INV DIAGONAL SECOND: " , check_board_inverse_diagonal(SECOND))
288
289     init_board()
290     board[0][2] = FIRST
291     board[1][1] = FIRST
292     board[2][0] = FIRST
293     print(show_board())
294     print("DIAGONAL FIRST: " , check_board_diagonal(FIRST))
295     print("DIAGONAL SECOND: " , check_board_diagonal(SECOND))
296     print("INV DIAGONAL FIRST: " , check_board_inverse_diagonal(FIRST))
297     print("INV DIAGONAL SECOND: " , check_board_inverse_diagonal(SECOND))
298     init_board()
299     board[0][2] = SECOND
300     board[1][1] = SECOND
301     board[2][0] = SECOND
302     print(show_board())
303     print("DIAGONAL FIRST: " , check_board_diagonal(FIRST))
304     print("DIAGONAL SECOND: " , check_board_diagonal(SECOND))
305     print("INV DIAGONAL FIRST: " , check_board_inverse_diagonal(FIRST))
306     print("INV DIAGONAL SECOND: " , check_board_inverse_diagonal(SECOND))
```

プログラム 34 三目並べのプログラム、実装例（その 6 盤面のテスト関数その 3）

```

307  #
308  # 盤面のテスト関数 3、勝ち、引き分けの判定
309  #
310 def test_board3():
311     '盤面をテストする関数の 3 番目'
312     init_board()
313     board[0][0] = FIRST
314     board[1][0] = FIRST
315     board[2][0] = SECOND
316     board[0][1] = SECOND
317     board[1][1] = SECOND
318     board[2][1] = FIRST
319     board[0][2] = FIRST
320     board[1][2] = FIRST
321     board[2][2] = SECOND
322     print(show_board())
323     print("HORIZONTAL FIRST: ", check_board_horizontal(FIRST))
324     print("HORIZONTAL SECOND: ", check_board_horizontal(SECOND))
325     print("VERTICAL FIRST: ", check_board_vertical(FIRST))
326     print("VERTICAL SECOND: ", check_board_vertical(SECOND))
327     print("DIAGONAL FIRST: ", check_board_diagonal(FIRST))
328     print("DIAGONAL SECOND: ", check_board_diagonal(SECOND))
329     print("INV DIAGONAL FIRST: ", check_board_inverse_diagonal(FIRST))
330     print("INV DIAGONAL SECOND: ", check_board_inverse_diagonal(SECOND))
331     print("IS WIN SIMPLE FIRST", is_win_simple(FIRST))
332     print("IS WIN SIMPLE SECOND", is_win_simple(SECOND))
333     print("IS WIN ACTUAL FIRST", is_win_actual(FIRST))
334     print("IS WIN ACTUAL SECOND", is_win_actual(SECOND))
335     print("IS FULL", is_full())
336     print("IS EVEN", is_even())
337
338     init_board()
339     board[0][0] = FIRST
340     board[1][0] = SECOND
341     board[2][0] = FIRST
342     board[0][1] = SECOND
343     board[1][1] = FIRST
344     board[2][1] = OPEN
345     board[0][2] = FIRST
346     board[1][2] = OPEN
347     board[2][2] = SECOND
348     print(show_board())
349     print("HORIZONTAL FIRST: ", check_board_horizontal(FIRST))
350     print("HORIZONTAL SECOND: ", check_board_horizontal(SECOND))
351     print("VERTICAL FIRST: ", check_board_vertical(FIRST))
352     print("VERTICAL SECOND: ", check_board_vertical(SECOND))
353     print("DIAGONAL FIRST: ", check_board_diagonal(FIRST))
354     print("DIAGONAL SECOND: ", check_board_diagonal(SECOND))
355     print("INV DIAGONAL FIRST: ", check_board_inverse_diagonal(FIRST))
356     print("INV DIAGONAL SECOND: ", check_board_inverse_diagonal(SECOND))

```

```
357     print("IS WIN SIMPLE FIRST", is_win_simple(FIRST))
358     print("IS WIN SIMPLE SECOND", is_win_simple(SECOND))
359     print("IS WIN ACTUAL FIRST", is_win_actual(FIRST))
360     print("IS WIN ACTUAL SECOND", is_win_actual(SECOND))
361     print("IS FULL", is_full())
362     print("IS EVEN", is_even())
363
364     init_board()
365     board[0][0] = SECOND
366     board[1][0] = FIRST
367     board[2][0] = SECOND
368     board[0][1] = FIRST
369     board[1][1] = SECOND
370     board[2][1] = FIRST
371     board[0][2] = SECOND
372     board[1][2] = OPEN
373     board[2][2] = FIRST
374     print(show_board())
375     print("HORIZONTAL FIRST: ", check_board_horizontal(FIRST))
376     print("HORIZONTAL SECOND: ", check_board_horizontal(SECOND))
377     print("VERTICAL FIRST: ", check_board_vertical(FIRST))
378     print("VERTICAL SECOND: ", check_board_vertical(SECOND))
379     print("DIAGONAL FIRST: ", check_board_diagonal(FIRST))
380     print("DIAGONAL SECOND: ", check_board_diagonal(SECOND))
381     print("INV DIAGONAL FIRST: ", check_board_inverse_diagonal(FIRST))
382     print("INV DIAGONAL SECOND: ", check_board_inverse_diagonal(SECOND))
383     print("IS WIN SIMPLE FIRST", is_win_simple(FIRST))
384     print("IS WIN SIMPLE SECOND", is_win_simple(SECOND))
385     print("IS WIN ACTUAL FIRST", is_win_actual(FIRST))
386     print("IS WIN ACTUAL SECOND", is_win_actual(SECOND))
387     print("IS FULL", is_full())
388     print("IS EVEN", is_even())
```

プログラム 35 三目並べのプログラム、実装例（その 7 棋譜関連の関数）

```
389 #  
390 # ログのリプレイ  
391 #  
392 def replay_log(log):  
    '棋譜 log をたどります。print 文で画面に出力します'  
    init_board()  
    init_turn()  
    print(show_board())  
    for m in log:  
        if len(m) == 2:  
            print(show_turn(), "の番です")  
            print(set_board(m[0], m[1], turn))  
            print(show_board())  
            print("IS WIN", turn, ": ", is_win_actual(turn))  
            change_turn()  
        else:  
            print("RESULT IN LOG: ", m[0])  
    print("IS WIN FIRST: ", is_win_actual(FIRST))  
    print("IS WIN SECOND: ", is_win_actual(SECOND))  
    print("IS EVEN: ", is_even())  
409 #  
410 # ログのテスト  
411 #  
412 def test_log():  
    '棋譜をテストします'  
    print("LOG1")  
    replay_log(log1)  
    print("LOG2")  
    replay_log(log2)  
    print("LOG3")  
    replay_log(log3)  
420 #  
421 # すべてのテスト  
422 #  
423 def test_all():  
    'すべてのテストを行います'  
    test_turn()  
    test_board1()  
    test_board2()  
    test_board3()  
    test_log()
```

プログラム 36 三目並べのプログラム、実装例（その 8 プレイ関数とメインプログラム）

```

430 # 
431 # 実際のプレイ
432 #
433 def play():
434     '端末への入出力を用いて実際に三目並べをプレイする関数です'
435     init_turn()
436     init_board()
437     print(show_board())
438 # 棋譜用の空リストを作る。play() の外側でアクセスするなら global 嘱咐
439 # global log
440     log = []
441     while True:
442         print(show_turn(), "の番です")
443         while(True):
444             row = int(input("行を入力してください: "))
445             column = int(input("列を入力してください: "))
446             result = set_board(row, column, turn)
447             print(result)
448             if (result == "OK"):
449                 break
450             print("不適切な入力です。再度、入力して下さい")
451 # ここ(内側の while の外)で log に手を追加
452 #
453 # 要追加
454 #
455     print(show_board())
456     if (is_even()):
457         print("引き分けです")
458         # ここで棋譜に勝敗(引き分け)を追加
459         log.append([EVEN])
460         break
461     if (is_win_actual(turn)):
462         print(show_turn(), "の勝ちです")
463         # ここで棋譜に勝敗(turnの勝ち)を追加
464         break
465         change_turn()
466     # ここで棋譜のリプレイ
467     # 現在は log は空なので判定して処理
468     if len(log)>0:
469         replay_log(log)
470     else:
471         print("棋譜は作成されていません")
472 if __name__ == '__main__':
473     print('三目並べ')

```

3) プログラムのテスト

上記のプログラムを読み、Python Shell で実行した後、シェルからテスト用の関数を実行してプログラムの動作を確認してください。用意されているテスト関数は上から順に以下のものです。

```
test_turn()  
test_board1()  
test_board2()  
test_board3()  
test_log()  
test_all()
```

4) プログラムの実行

上記のプログラムを実行後、Python Shell から play() 関数を呼び出して実際に三目並べを行ってみてください。

```
play()
```

演習 40. 棋譜の採取

上記のプログラムを拡張し、play() 関数の中で、そのプレイの棋譜を採取するようにしてください。また勝敗が確定した後、棋譜を再生するようにしてください。

12.6 力試し

これまでに学んだ方法を使い、力試しとして、以下のようなことに挑戦してください。

- CUI ではなく、tkinter を用いて GUI で動く三目並べを作る
- 手番や盤面、棋譜などをそれを操作する関数をメソッドとしてクラスで実装する。
- 人間同士ではなく、片側をコンピュータがプレイするように拡張する。
- 三目並べではなく、オセロゲームについて同様のプログラムを開発する。

12.7 プログラムの開発に関連するいくつかの話題

12.7.1 テストで分かること

コンピュータのプログラムは通常、無限ともいえる場合に適正に稼働しなければなりません。例えば三目並べでも可能な棋譜の数は相当に多いものです。このため、テストについては

- テストにパスしなかったプログラムは誤りがあることは容易に分かりますが
 - テストにパスしたことが期待されるすべての場合にプログラムが適切に稼働することは保証しない
- ということにも留意しましょう。

部品となる関数のテストは、それらを組み合わせた関数のテストよりも行いやすく、正しく動作する部品で作ることで複雑な関数への信頼度が高まります。

12.7.2 リファクタリング

プログラムの改善には以下の2つの方向があります。

- プログラムの機能を強化する
 - プログラムの機能は維持し、実装を整理して維持管理や拡張を容易にすること
- 後者のような改善を「リファクタリング」と呼びます。例えば、三目並べのプログラムについてクラスを用いて実装しなおす、などはこれにあたります。リファクタリングが必要とされる理由はいくつか挙げられます。
- プログラムは長期にわたって使われるため、保守が容易でなければならない。
 - プログラムの開発者は入れ替わる可能性がある。
 - プログラムへの機能追加などの要求が継続的に発生する。

12.7.3 ソフトウェア開発のVモデル

上記の三目並べのような小さな例でも、プログラムの開発の前半は、何を作るべきか（要求仕様）を明確にし、それを段階的に詳細化して実装する関数などを明確にしています。後半は、相互依存のない関数から順に実装とテストを繰り返し、全体を積み上げて完成に至ります。

このようなプロセスは下の図のようなV型で表せ、ソフトウェア開発のVモデル

ルと呼ばれます。

V型の構造のため、全体に近い（V型の上方）では、設計から実装・テストまではその下に多くの作業が含まれ、距離が遠くなります。このため上方での設計の不適切さは実装とテストで発見されるまで時間がかかり、手戻りによる手直しも多くなり慎重に行う必要があります。

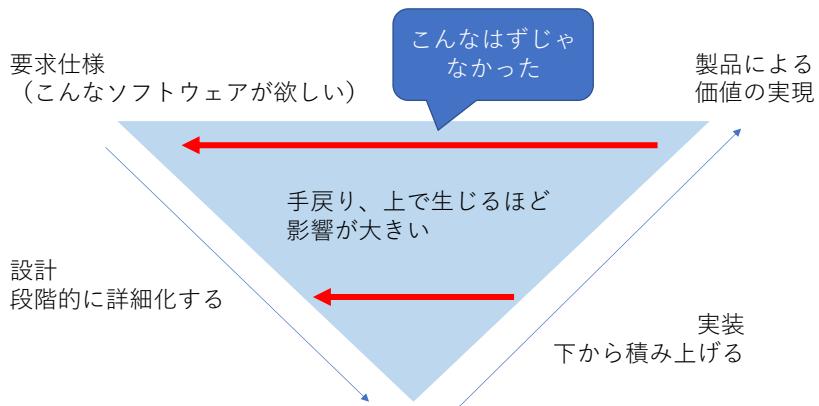


図 49 ソフトウェア開発の V モデル

13. Python の学術利用

13.1 本章の学習の目標

Python が注目されている理由の一つは数値計算など学術利用に適したライブラリが豊富に提供されていることです。本章では以下の 3 つのライブラリについて、NumPy や pandas でのデータの扱い方と、Matplotlib でのグラフ描画の基礎を学びます。それぞれ、かなり高機能なパッケージですし、適用領域の知識も必要になりますので、ここでは基礎的な事項だけ学びます。

1. NumPy: 科学技術領域での数値計算のための基礎的なパッケージです
SciPy: より高度な数値計算のためのライブラリです。
2. Matplotlib: データをグラフにプロットするためのパッケージです
3. pandas: データ分析のためのパッケージです

これらは次の図のように相互に関連しています。

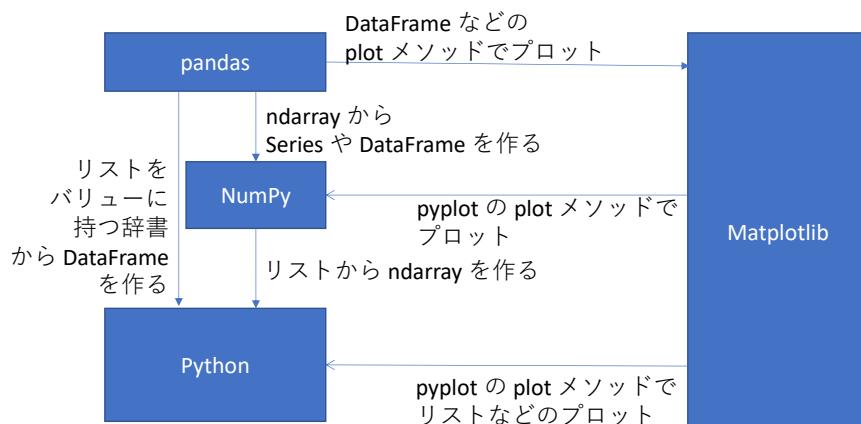


図 50 NumPy, Matplotlib, pandas の関連

13.2 import 時の別名

NumPy, Pandas, Matplotlib などの `import` には以下のような別名がよく使われます。本書でもこれを用います。

```

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
  
```

13.3 NumPy

Python はそれ自体は実行の遅いプログラミング言語ですが、NumPy の内部では C 言語で実装されており、ベクトルや行列などの演算が高速で実行できます。

13.3.1 多次元配列の生成

Python でデータを一括して扱うにはリストを使いますが、NumPy では固有のデータ形式である ndarray（別名 array）を使います。

1) リストから生成する

```
import numpy as np
data1 = [1, 2, 3]
arr1 = np.array(data1) # 1 次元のデータ
data2 = [[1,2,3],[4,5,6]]
arr2 = np.array(data2)
```

2) すべての要素が 0 の配列を作る

```
np.zeros(5) # 大きさ 5 の 1 次元配列
array([0., 0., 0., 0., 0.])
np.zeros((2,2)) # 大きさ (2,2) の 2 次元配列、() が二重に注意
array([[0., 0.],
       [0., 0.]])
a = np.array([[1,2,3],[4,5,6]])
np.zeros_like(a) # 配列 a と同じ大きさ
array([[0, 0, 0],
       [0, 0, 0]])
```

すべての要素が 1 の配列は同様に ones, ones_like で作れます。

13.3.2 ndarray の属性

ndarray では以下のような属性をしらべることができます。

- dim : ndarray の次元
- shape: ndarrray の大きさ
- dtype: データ型（整数は高速の演算が可能な型が採用されています）

```

import numpy as np
arr2 = np.array([[1,2,3],[4,5,6]]
arr2.dim
2
arr2.shape
(2,3)
arr2.dtype
dtype('Int32')

```

13.3.3 ndarray の要素へのアクセス

ndarray の要素はリストと同じように [] でアクセスできます。添え字の 0 始まりも同じです。

```

arr1 = np.array([1,2,3])
arr1[0]
1
arr1[1] = 1
arr1
array([1,1,3])

```

多次元配列では [][] の代わりに [,] という記法も使えます。

```

arr2[0][0]
arr2[0,0]

```

13.3.4 スライス

ndarray でもリスト同様、スライスが使えます。

```

arr1[2:]
array([3])

```

多次元配列では[:,:] という記法を用います。

```

arr2[0:2,0:2]
array([1,2][4,5])

```

注意 ndarray のスライスの結果は「コピー」ではなく、もとの ndarray の一部への参照です。

スライスにスカラー値を代入するとすべての要素に代入されます。

13.3.5 ndarray の演算

ndarray 型のデータに対しては四則演算、べき乗、比較などができます。これらは要素ごとの演算になります。

行列の積は @ という演算を使います。

スカラー値との演算では、すべての要素にその値が適用されます。

```
arr1 = np.array([1,2,3])
arr1*2
array([2,4,6])
arr1 + 1
array([2,3,4])
```

13.3.6 条件を満たす要素の抽出

以下の方法で条件を満たす要素を抽出できます。

```
arr1 = np.array([1,2,3,4,5])
cond = arr1 > 2 # 条件を満たすかどうかの配列の生成
cond
array([False, False,  True,  True,  True])
arr1[cond]          # 条件を指定してスライス
array([3, 4, 5])
```

13.3.7 行列計算

Numpy では ndarray を対象に数学の行列計算を簡単に実行できます。

- 行列の転置（行と列の入れ替え）
ndarray の T 属性を使います。
- 行列積
@ 演算を使います。
- linalg モジュールの利用：numpy.linalg (numpy を np でインポートしていれば np.linalg) には以下のような行列を扱う関数が定義されています。
 - diag (対角要素),
 - trace (対角要素の和) ,
 - det (行列式),
 - eig (固有値),

- inv (逆行列),
- solve (一次方程式を解く)

13.3.8 亂数

Numpy では一括して乱数を生成することができます。

- seed 亂数生成の初期値を設定します
- rand 一様連續乱数を生成します
- randn 標準正規分布に従う乱数を生成します
- randint 与えられた範囲の乱数を生成します

以下が使用例です

- `np.random.rand(10)`
0 から 1 までの浮動小数点乱数を 10 個発生
- `np.random.randn(5,5)`
大きさ $(5,5)$ の 2 次元配列として標準正規分布に従う乱数を発生
- `np.random.permutation([1,2,3,4,5])`
リスト $[1,2,3,4,5]$ のランダムな並べ替えを生成。引数には `range()` や `ndarray` も指定可。多次元配列では最初の添え字のみ入れ替え
- `np.random.randint(2,size=10)`
(0 以上) 2 未満の整数乱数を大きさ 10 で生成。下限、上限を与えることが可能なため、大きさは `size =` として指定

13.4 Matplotlib

13.4.1 backend: グラフ出力環境

Matplotlib ではグラフの出力方法を選ぶことが可能ですが。ここでは IDLE で動かすため tkagg という tkinter を利用した環境の指定方法を説明します。

- Matplotlib ではグラフを実際に出力する環境のことを `backend` と呼びます。さまざまな `backend` が用意されています。
- tkagg は Tkinter を使ってグラフを出力する `backend` です。
- IDLE 環境では、matplotlib のインポートのあとに `use()` 関数を使って指定します。

```
import matplotlib
matplotlib.use('tkagg')
```

- `use()` の指定はプロット用のモジュール `matplotlib.pyplot` のインポートより前にしなければなりません。
- Ipython やそれを使う Jupyter Notebook では `matplotlib` の利用に先だって使用したい `backend` に応じて以下を指定します。

```
%matplotlib notebook
%matplotlib tk
```

13.4.2 日本語での文字出力

- `matplotlib` の標準のフォントは日本語文字を持たないので表示は□になってしまいます。
- `matplotlib` version 3.1 以降では ttc フォントが使えるようになりました。これ以降のバージョンをご利用の方はフォントの追加インストールは不要です¹。
- 指定方法はいくつかありますが、ここではプログラム内で一括指定する方法を示します。
- フォントを追加でインストールした場合はユーザのフォルダにある `.matplotlib` フォルダの `fontList.json` が古ければいったん消去します。
- 利用法の概要

```
import matplotlib
# 出力先として tkinter を設定、pyplot のインポートより前に
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
# matplotlib で日本語表示を可能にする
# matplotlib version 3.1 以降なら Yu Gothic が使用可能
matplotlib.rc('font', **{'family':'Yu Gothic'})
# 以下はプロット例
data = [1,2,3]
plt.plot(data)
plt.title('タイトル')
plt.show()
```

13.4.3 タイトル、軸ラベル、線種の設定

- グラフのタイトルを設定する関数は `title` です。

¹ 以下の例では `Yu Gothic` を設定していますが `Yu Mincho`、`MS Gothic`、`MS Mincho` なども使えるはずです。

- X-軸ラベルを設定する関数は xlabel です。
- Y-軸ラベルを設定する関数は ylabel です。
- 線種の指定は plot 関数の引数で与えます。
 - 色、線種、マーカの文字列で指定する方法
 - color, linestyle, linewidthなどの引数で指定する方法。
- 使用例（該当部分）

```
plt.plot([1,2,3],‘k-’) #黒の実線  
plt.plot([2,3,4],‘r--’) #赤の破線  
plt.plot([3,4,5],‘b--o’) #青の破線、○のマーカ  
plt.title(‘タイトル’)  
plt.xlabel(‘横軸’)  
plt.ylabel(‘縦軸’)  
plt.show()
```

13.4.4 利用例

1) use_matplotlib.py

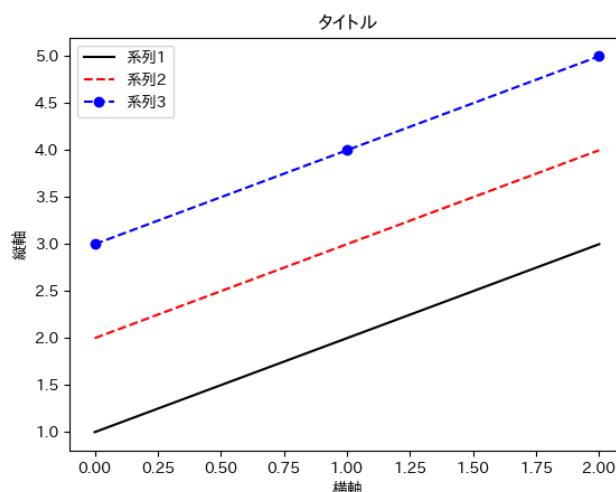


図 51 Matplotlib の使用例

プログラム 37 `use_matplotlib.py`

| 行 | ソースコード |
|----|---|
| 1 | # |
| 2 | # matplotlib 基本の使い方 |
| 3 | # |
| 4 | import matplotlib |
| 5 | # |
| 6 | # 出力先として tkinter を use で設定、pyplot のインポートより前に |
| 7 | # |
| 8 | matplotlib.use('tkagg') |
| 9 | import matplotlib.pyplot as plt |
| 10 | # |
| 11 | # matplotlib で日本語表示を可能にする |
| 12 | # |
| 13 | #matplotlib.rcParams['font'].update({'family':'Yu Gothic'}) |
| 14 | # |
| 15 | # 教育用端末なら下の設定で |
| 16 | matplotlib.rcParams['font'].update({'family':'IPAPGothic'}) |
| 17 | # |
| 18 | # 3 本の線グラフを書く |
| 19 | # |
| 20 | plt.plot([1,2,3],'k-',label='系列 1') |
| 21 | plt.plot([2,3,4],'r--',label='系列 2') |
| 22 | plt.plot([3,4,5],'b--o',label='系列 3') |
| 23 | # |
| 24 | plt.title('タイトル') |
| 25 | plt.xlabel('横軸') |
| 26 | plt.ylabel('縦軸') |
| 27 | plt.legend() # 凡例 |
| 28 | plt.show() |

2) 散布図の描画

散布図は pyplot の scatter 関数に x 軸データ、y 軸データを与えて描画します。

- use_matplotlib_scatter.py

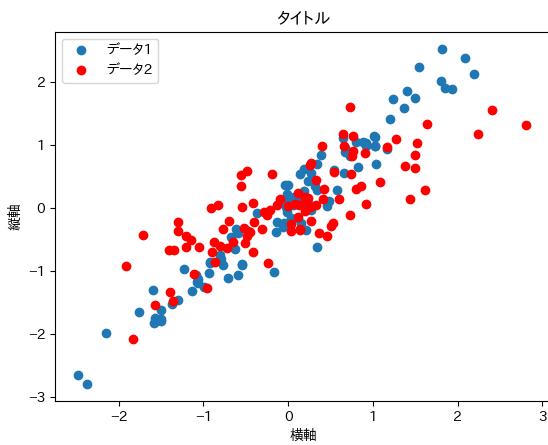


図 52 散布図の描画

プログラム 38 use_matplotlib_scatter.py

| 行 | ソースコード |
|----|---|
| 1 | # |
| 2 | # matplotlib で散布図を描く |
| 3 | # |
| 4 | import matplotlib |
| 5 | |
| 6 | matplotlib.use('tkagg') |
| 7 | import matplotlib.pyplot as plt |
| 8 | import numpy as np |
| 9 | # |
| 10 | # matplotlib で日本語表示を可能にする |
| 11 | matplotlib.rc('font', **{'family':'Yu Gothic'}) |
| 12 | # |
| 13 | # ランダムなデータの作成 |
| 14 | # |
| 15 | datax = np.random.randn(100) |
| 16 | datay = datax + np.random.randn(100)*0.3 |
| 17 | # |
| 18 | # 散布図の描画 |
| 19 | # |
| 20 | plt.scatter(datax,datay,label='データ 1') |

```
21      #
22      # 別のデータの作成
23      #
24      datax = np.random.randn(100)
25      datay = 0.6*datax + np.random.randn(100)*0.4
26      #
27      # 色を指定して散布図を作成
28      #
29      plt.scatter(datax,datay,color='red',label='データ 2')
30      #
31      # タイトル、軸ラベル、凡例の記入
32      #
33      plt.title('タイトル')
34      plt.xlabel('横軸')
35      plt.ylabel('縦軸')
36      plt.legend()
37      #
38      # 表示
39      #
40      plt.show()
```

•

3) ヒストグラムの描画

ヒストグラムは pyplot の hist 関数にデータを与えて描画します。階級数などは自動調整されますが、指定することも可能です。

use_matplotlib_hist.py

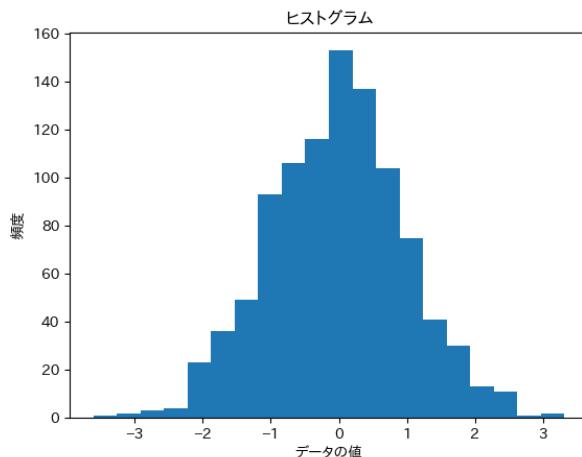


図 53 ヒストグラムの描画

プログラム 39 use_matplotlib_hist.py

| 行 | ソースコード |
|----|---|
| 1 | # |
| 2 | # matplotlib でヒストグラムを描く |
| 3 | # |
| 4 | import matplotlib |
| 5 | # |
| 6 | # 出力先として tkinter を設定、pyplot のインポートより前に |
| 7 | # |
| 8 | matplotlib.use('tkagg') |
| 9 | import matplotlib.pyplot as plt |
| 10 | import numpy as np |
| 11 | |
| 12 | # |
| 13 | # matplotlib で日本語表示を可能にする |
| 14 | matplotlib.rcParams['font'] = {'family': 'Yu Gothic'} |
| 15 | # |
| 16 | # ヒストグラムの作成 |
| 17 | # |
| 18 | data = np.random.randn(1000) |
| 19 | plt.hist(data,bins=20) |
| 20 | # |

```

21 # タイトル、軸ラベルを設定
22 #
23 plt.title('ヒストグラム')
24 plt.xlabel('データの値')
25 plt.ylabel('頻度')
26 #
27 # 表示
28 #
29 plt.show()

```

4) 複数グラフの描画

Matplotlib では以下の方法で複数のグラフを並べて描画できます。

- pyplot の figure 関数で Figure オブジェクトを得ます。
- Figure オブジェクトに pyplot の add_subplot 関数で subplot を追加します。結果を変数に保存します。
- subplot の間隔を pyplot の subplots_adjust 関数で調整します。
- 各 subplot に plot, scatter, hist 関数で描画します。
- タイトル、軸ラベルは set_title, set_xlabel, set_ylabel で追加。関数名に注意。

use_matplotlib_subplot.py

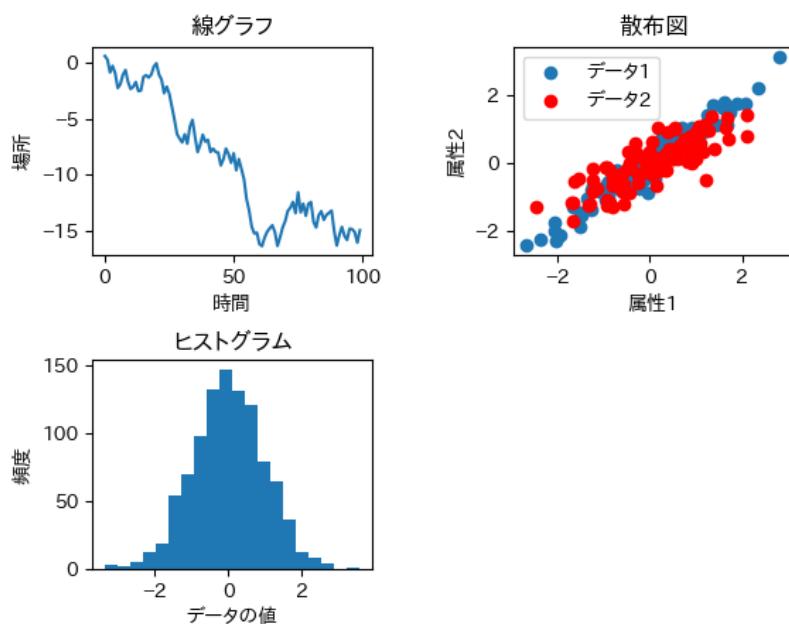


図 54 複数のグラフの描画

プログラム 40 use_matplotlib_subplot.py

| 行 | ソースコード |
|----|---|
| 1 | # |
| 2 | # subplot を使う例 |
| 3 | # |
| 4 | import matplotlib |
| 5 | matplotlib.use('tkagg') |
| 6 | import matplotlib.pyplot as plt |
| 7 | import numpy as np |
| 8 | # |
| 9 | # テキストで日本語を表示できるようにする |
| 10 | # |
| 11 | matplotlib.rc('font', **{'family':'Yu Gothic'}) |
| 12 | # |
| 13 | # 3 つの subplot を作成、間隔を調整 |
| 14 | # |
| 15 | fig = plt.figure() |
| 16 | ax1 = fig.add_subplot(2,2,1) |
| 17 | ax2 = fig.add_subplot(2,2,2) |
| 18 | ax3 = fig.add_subplot(2,2,3) |
| 19 | plt.subplots_adjust(hspace=0.5, wspace= 0.5) |
| 20 | # |
| 21 | # 1 つめに線グラフを出力 |
| 22 | # |
| 23 | data = np.random.randn(100).cumsum() |
| 24 | ax1.plot(data) |
| 25 | ax1.set_title('線グラフ') |
| 26 | ax1.set_xlabel('時間') |
| 27 | ax1.set_ylabel('場所') |
| 28 | # |
| 29 | # 2 つめに散布図を出力 |
| 30 | # |

```
31 datax = np.random.randn(100)
32 datay = datax + np.random.randn(100)*0.3
33 ax2.scatter(datax,datay,label='データ 1')
34
35 datax = np.random.randn(100)
36 datay = 0.6*datax + np.random.randn(100)*0.4
37 ax2.scatter(datax,datay,color='red',label='データ 2')
38
39 ax2.set_title('散布図')
40 ax2.set_xlabel('属性 1')
41 ax2.set_ylabel('属性 2')
42 ax2.legend()
43
44 #
45 # 3 つめにヒストグラムを出力
46 #
47 data = np.random.randn(1000)
48 ax3.hist(data,bins=20)
49
50 ax3.set_title('ヒストグラム')
51 ax3.set_xlabel('データの値')
52 ax3.set_ylabel('頻度')
53
54 #
55 # グラフを表示
56 #
57 plt.show()
```

13.5 pandas

13.5.1 Dataframe

Pandas 固有のデータ形式には以下のものがあります。

- 1 次元の Series

● 2次元の DataFrame

DataFrame には行名(index)と列名(column)がつけられます。



13.5.2 DataFrame を作る

1) numpy の array から作る

```
import numpy as np
import pandas as pd
d = np.array([[1,2,3],[4,5,6],[7,8,9]])
df = pd.DataFrame(d,columns=['a','b','c'])
df
   a   b   c
0  1   2   3
1  4   5   6
2  7   8   9
```

列名と行名はそれぞれ

```
df.columns
df.index
```

で調べられます。

2) リスト型を値にもった辞書から作る

```
df = pd.DataFrame({'a': [1,4,7], 'b':[2,5,8], 'c':[3,6,9]})
df
```

| | a | b | c |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 1 | 4 | 5 | 6 |
| 2 | 7 | 8 | 9 |

辞書は「キー」と「値（バリュー）」のペアの集合で作られる Python のデータ型です。

```
dic = {'a':1, 'b':2, 'c':3}
```

キーから値を検索できます。

```
dic['a']
```

```
1
```

13.5.3 csv ファイルを読み込む

- csv ファイル形式で保存された表計算ソフトのデータを読み込んで DataFrame を作れます

```
df = pd.read_csv(ファイル名)
```

- 先頭行が列名として扱われます。

➤ すべてをデータとして読むときには、オプションとして以下を指定します。

```
header = None か names = [列名のリスト]
```

- Windows で日本語を含むファイルを読み込む場合は漢字コードを指定します。

```
encoding = 'SHIFT-JIS'
```

- 列名に漢字コードを使うと、列名でのデータの指定などでエラーが出ます。

- sample2.csv は以下のようなファイルです。

| ID | Japanese | English | Mathematics | Science | Social Studies |
|----|----------|---------|-------------|---------|----------------|
| A | 91 | 69 | 100 | 82 | 94 |
| B | 80 | 60 | 45 | 52 | 46 |
| C | 92 | 92 | 76 | 73 | 97 |
| D | 58 | 50 | 60 | 71 | 77 |
| E | 58 | 75 | 96 | 96 | 94 |
| F | 92 | 89 | 86 | 82 | 74 |
| G | 97 | 87 | 59 | 55 | 56 |

以下の手順で読み込みます (use_read_csv.py)

- numpy のインポート

2. pandas のインポート
3. os のインポート
4. os.chdir で csv ファイルのあるフォルダに移動

```
os.chdir(r"フォルダのフルパス")  
# r を付けるのは特殊な文字の解釈  
# をしないため
```

5. CSV ファイルの読み込み

```
df = pd.read_csv("ファイル名")
```

注意：pd.read_csv は日本語のファイル名を正しく処理できないようです。

プログラム 41 use_read_csv.py

| 行 | ソースコード |
|----|--|
| 1 | import numpy as np |
| 2 | import pandas as pd |
| 3 | import os |
| 4 | # |
| 5 | # データあるフォルダに移動、文字列に r を付けて特殊な文字の解釈を止める |
| 6 | # |
| 7 | # pandas は日本語のファイル名をうまく処理できない |
| 8 | # |
| 9 | os.chdir(r"C:\Users\一\Documents\My Documents\授業\プログラミング演習 python") |
| 10 | df = pd.read_csv("sample2.csv") |
| 11 | # |
| 12 | # 水平方向(axis = 1) に総和をとり、Total という列を作る |
| 13 | df['Total'] = df.sum(axis=1) |
| 14 | # データフレーム df を表示 |
| 15 | print(df) |
| 16 | # データフレーム df の要約統計量を表示 |
| 17 | print(df.describe()) |

13.5.4 要約統計量の表示

describe メソッドで要約統計量を表示できます。

13.5.5 Pandas のデータのプロット

Pandas でのプロットは DataFrame 側の plot メソッドを呼び出します。

(use_DadaFrame_plot.py)

```
df.plot()      #折れ線グラフ
df.plot.bar(stacked=True) # 積み上げ棒グラフ
df.plot.scatter('Japanese','English') # 列を指定して散布図
df['Japanese'].plot.hist() # 列を指定してヒストグラム
```

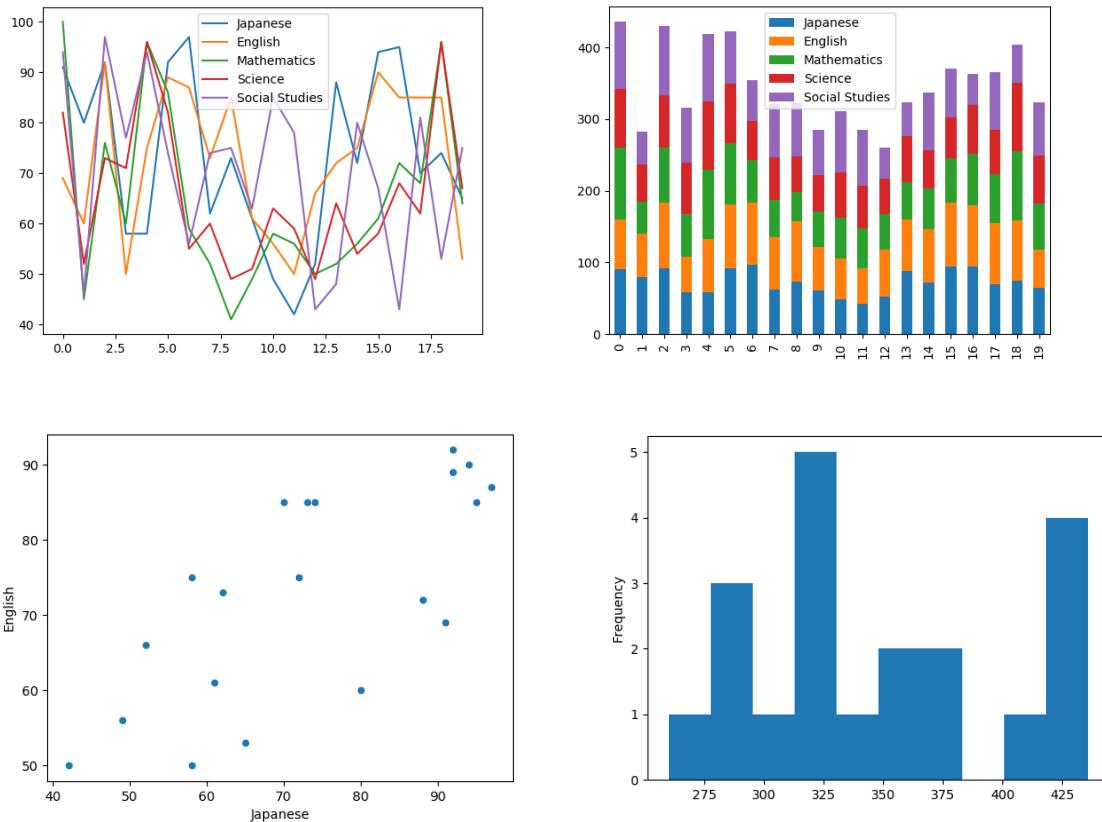


図 55 pandas でのグラフ作成

プログラム 42 use_DataFrame_plot.py

| 行 | ソースコード |
|----|--|
| 1 | import numpy as np |
| 2 | import pandas as pd |
| 3 | import os |
| 4 | import matplotlib |
| 5 | matplotlib.use('tkagg') |
| 6 | import matplotlib.pyplot as plt |
| 7 | # |
| 8 | # データあるフォルダに移動、文字列に r を付けて特殊な文字の解釈を止める |
| 9 | # |
| 10 | # pandas は日本語のファイル名をうまく処理できない |
| 11 | # 個人環境に合わせたディレクトリ（フォルダ）の移動 |
| 12 | # os.chdir(r"C:\Users\Documents\My Documents\授業\プログラミング演習 python") |

```
13    #
14    # 教育用端末なら以下で
15    os.chdir(r"M:\$Documents")
16    #
17    df = pd.read_csv("sample2.csv")
18    #
19    #
20    # 折れ線グラフ
21    #
22
23    df.plot()
24    print("次に進むにはグラフィンドウを閉じてください")
25    plt.show()
26    #
27    # 積み上げ棒グラフ
28    #
29    df.plot.bar(stacked=True)
30    print("次に進むにはグラフィンドウを閉じてください")
31    plt.show()
32    #
33    # 散布図
34    #
35    df.plot.scatter('Japanese','English')
36    print("次に進むにはグラフィンドウを閉じてください")
37    plt.show()
38    #
39    # 水平方向(axis = 1) に総和をとり、Total という列を作る
40    #
41    df['Total'] = df.sum(axis=1)
42
43    #
44    # ヒストグラム
45    #
46    df['Total'].plot.hist()
```

```

47     print("次に進むにはグラフィンドウを閉じてください")
48     plt.show()

```

13.6 課題

`np_matplotlib.py` は Numpy と matplotlib を使って 1 乗~4 乗のグラフを描くプログラムです。

演習 41. これを改造して鋸波のフーリエ近似を描くプログラムを作成しなさい。

Numpy (np) では円周率は `np.pi`, 正弦関数は `np.sin()` で利用できます。

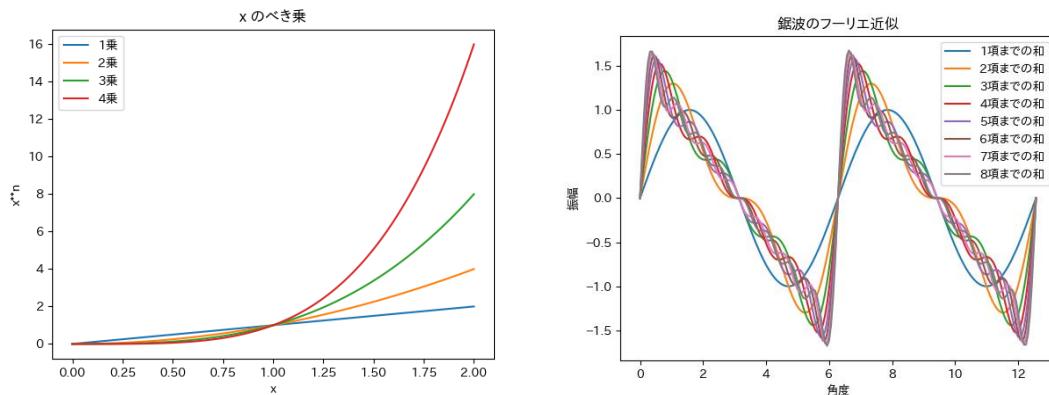


図 56 べき乗のグラフとのこぎり波の三角関数の和での近似

プログラム 43 Numpy と Matplotlib で 1~4 乗のグラフを描く

| 行 | ソースコード | 説明 |
|---|---|--------------------|
| 1 | # | |
| 2 | # | |
| 3 | # Numpy のデータを plot する例題 | |
| 4 | import matplotlib | matplotlib の準備 |
| 5 | matplotlib.use('tkagg') | |
| 6 | import matplotlib.pyplot as plt | |
| 7 | import numpy as np | numpy の準備 |
| 8 | matplotlib.rc('font', **{'family':'Yu Gothic'}) | matplotlib のフォント指定 |
| 9 | # | |

```
10 # x の 1乗 ~ 4乗をプロットする
11 #
12 steps = 100
13 order = 4
14 maxx = 2
15 #
16 # 要素の値 0 で steps 行、 order 列の行列を作成
17 #
18 datalist = np.zeros((steps, order))
19 #
20 # 凡例用のリスト
21 #
22 legend_label=[]
23 #
24 # x の値を linspace で作成
25 #
26 x = np.linspace(0,maxx,steps)
27 #
28 # 各列について、一気に計算する
29 #
30 for j in range(1,order+1):
31     datalist[:,j-1] = x**j
32     legend_label.append(str(j) +'乗')
33 #
34 # プロット
35 #
36 plt.plot(x, datalist)
37 plt.title('x のべき乗')
38 plt.xlabel('x')
39 plt.ylabel('x**n')
40 plt.legend(legend_label)
41 plt.show()
```

参考文献

- [17] Wes McKinney 著、瀬戸山ほか訳：Python によるデータ分析入門、第2版、オライリージャパン（2018）

以下のサイトのチュートリアルのページなどが参考になります。

- [18] NumPy の Web サイト
<http://www.numpy.org/index.html>
- [19] Pandas の Web サイト
<https://pandas.pydata.org/>
- [20] Matplotlib の Web サイト
<https://matplotlib.org/>
- [21] TkAgg のボタン操作など（見つけにくい）
https://matplotlib.org/users/navigation_toolbar.html

14. 振り返りとこれから

14.1 本章の学習の目標

1. この授業はここで最後です。この章ではこれまでの学習で何を学べたのか振り返ります。
2. 本授業では Python の演習を統合環境 IDLE で学びました。IDLE を採用した理由は機能が限定されていて学びやすいからですが、他にどのようなものが利用できるのかを知ります。
3. 学んだことをどう活かすのかを考える。

14.2 振り返り

この授業の受講前と今とを比較して学習を振り返ってください。

- 何ができるようになりましたか
- 受講前の想定・期待とどのように違っていましたか
- これから、どのような学習目標を設定しますか

14.3 Python の利用環境

この授業では初学者にとっての利用しやすさから構成の単純な IDLE を Python の統合利用環境として用いました。ただ IDLE は単純なので「捨てられる」環境ともいわれています。Python については、さまざまな利用環境があります。

- Anaconda では Jupyter Notebook や Spider が含まれています。
- これらは ipython という Python シェルよりもさらに対話的な実行環境で稼働します。
- このほかにも Python に適したエディタとコマンドラインでの実行 (python, ipython) というスタイルも使われます。

14.4 モジュール等の追加

Python の特徴の一つが多くの方々がさまざまなライブラリを開発してくれているということです。具体的な応用例について web 上などのさまざまな情報が掲載されて

いますが、これらを利用する際にはライブライであるモジュールの追加が必要になります。Anaconda では conda コマンドで、anaconda で対象としているものや、配布パッケージとして Python を使われた方は pip コマンドを使ってモジュールの追加を行います。

14.5 本書で紹介しなかった話題

Python は応用の広いプログラミング言語ですが、応用に際しては応用に関連した領域の知識が必要になります。本書で紹介した NumPy や pandas も数値計算や統計処理の知識がないと使えません。このため、関連領域の知識が必要な話題は取り上げませんでした。具体的には以下のようないふしが挙げられます。ご自身の興味に沿って勉強して頂ければと思います。

- ネットワークや web 関連の話題
- 画像処理などマルチメディアを扱う話題
- データベースを扱う話題
- 機械学習など人工知能関連の話題

14.6 感謝と恩返し—学んだことをどう活かすのか

家庭で木工をする人は少なくありません。木工ができると Do It Yourself で簡単な家具などは自分で作ることで住まいの問題解決ができるようになります。一方で、ボール盤、旋盤、フライス盤などの機械が必要な金属を加工することは少し敷居が高いでしょう。ですので、DIY 的に金属製品を作ろうとする人はあまりいません¹。

Python によらずコンピュータのプログラミングができるようになると、「こういうことはコンピュータでやれるはず」という「ものの見方」ができるようなっていると思います。それなら、皆さん自身がコンピュータやプログラミングを通じて社会にどう貢献するかをぜひ考えてください。

Python を含め、コンピュータやプログラミング言語などのソフトウェアは「誰かが作ってくれた」もの、多くの技術者やプログラマーからのプレゼントです。プログラミングを楽しんだら、このことに感謝し、ぜひ恩返しをしましょう。

¹ 筆者が中学生のとき、自宅が鉄工所だった友人は自分たちで自転車をつくることを考えていました。何ができる環境に居るかで自分ごととして考えられることが変わるのであります。

15. IDLE Python 便利帳

15.1 Python 便利メモ

- `help()` 関数：モジュールや関数を引数として説明を読みます。
- `globals()`：グローバルに定義されている変数などが表示されます。
- `id(x)`：オブジェクト `x` の番号が分かります。異なる変数が同じオブジェクトなのか確認できます。
- `type(x)`：オブジェクト `x` の型が分かります。変数にどのようなオブジェクトが代入されているかを確認できます。

15.2 ファイル名に注意

インポートするモジュールを作成するファイル名には使わない。

Python はモジュールを定められたフォルダで探索します。実行するファイルと同じフォルダはモジュール探索の対象です。例えば、`turtle` モジュールを使うときに、`turtle.py` という名称でファイルがあると、Python は間違って、このファイルをモジュールと理解します。

15.3 IDLE メモ—Python シェルのキー操作

- Ctrl-C: 実行中のプログラムの中断
- Ctrl-D: 端末入力でのファイルの終わり
➤ 注意：Shell での対話モードでは `shell` が終了します。
- TAB キー：スマートインデント
➤ 文字に続けると補完候補を表示
- ALT-P: 履歴戻る（すでに入力した行などを再利用できます。P: previous）
- ALT-N: 履歴次に（すでに入力した行などを再利用できます。N: next）
- スクリプトの実行
➤ エディタで作成したスクリプトの実行後はその環境で対話モードとなります。スクリプト内の関数呼び出しやグローバル変数の確認が可能です。

15.4 IDLE メモ—エディタ

- Ctrl-]: 選択した範囲をインデント
- Ctrl-[: 選択した範囲のインデントを減らす
- ALT-3: 選択した範囲をコメント化
- ALT-4: 選択した範囲のコメント化をもどす
- Ctrl-BS: 左側の 1 語を消す
- Ctrl-Del: 右側の 1 語を消す