Python プログラミング基礎

1章: プログラミングとPythonの基本知識

プログラミングに関する用語

プログラミング学習を始める前に、専門用語の意味を確認する。

「プログラミング」（programming）とは、コンピュータに行わせたい動作の手順を作成し、コンピュータに与えること。作成された手順をコンピュータプログラム（computer program）または単にプログラムという。プログラミングする人や職種をプログラマ（programmer）という。

上述の説明に登場した「コンピュータ」とは、複雑な計算を自動的に行う機械である。 コンピュータが行う計算の手順を記したものがプログラムであるので、コンピュータはプログラムに従って動作する機械とも捉えられる。

私達が普段使用するパソコンもコンピュータの一種である。 「パソコン」（personal computer）とは、個人向けのコンピュータ製品である。 小型、低価格であり、個人が手元に置いて直接操作して利用できる。 また汎用的であり、利用者がソフトウェアを追加することで用途を増やすことができる。

「ソフトウェア」（software）とは、何らかの機能や目的のために、プログラムを組み合わせまとめたものである。 プログラムの動作に必要なプログラム以外のデータ（例えば文章、画像、音声、動画など）が同梱されている場合もある。

ソフトウェアはハードウェアの対比として使われることが多い言葉である。 「ハードウェア」（hardware）とは、システムの構成要素のうち物理的な実体を持つもの（例えば回路、装置、機器、設備、施設など）である。 一方、ソフトウェアは物理的な実体を伴わない。

「システム」（system）とは、個々の要素が相互に作用し合いながら、全体として機能する仕組みである。 特に、ITにおけるシステムは、ハードウェアとソフトウェアを組み合わせて構成されたものである。 したがって、ソフトウェアの構成要素であるプログラムもまた、システムの中で相互に作用し合いながら機能している。

パソコンの構成部品

多くのプログラマは、パソコン上でプログラミングしている。 初心者がプログラミングを学ぶ際も、パソコン向けのプログラムを作成する場合が多い。 そこで、パソコンの構成部品について学ぼう。

典型的なパソコンは、マザーボード、中央演算装置（CPU）、メモリ、ハードディスクドライブ（HDD）、ソリッドステートドライブ（SSD）、マウス、キーボードなどから構成される。それぞれのぶひんについて説明する。

「マザーボード」は、コンピュータの主要な構成部品の一つである。 CPUやメモリなどの他の部品を装着することで、各部品に通電したり、部品間で通信できるようにする基盤である。 「CPU」も、主要な構成部品の一つで、他の装置や回路の制御やデータの演算などを行う装置である。 CPUの性能はコンピュータの処理速度に大きく影響する。 「メモリ」もまた、主要な構成部品の一つで、データの一時的な記憶装置である。

一般的に、パソコンで使われるメモリはパソコンの電源が入っている間のみデータを保持することができる。 そのため、パソコンの電源が切れると、メモリに記録されているデータは失われる。 そこで、電源の有無に関わらず残したいデータはHDDやSSDに記録する必要がある。

「HDD」は、コンピュータの主要なストレージ（外部記憶装置）の一つである。 HDDは薄くて硬い円盤（ディスク）の表面に塗布した磁性体の磁化状態を変化させてデータを記録している。 一台あたりの容量が大きく、容量あたりの単価が安いため、パソコンなどに内蔵されるストレージとして標準的な存在となっていた。 「SSD」は、近年HDDに代わって台頭しつつあるストレージの一種で、フラッシュメモリと呼ばれる装置を用いている。 HDDよりも高速で静かに動作する点で優れているが、価格や書き換え寿命など、HDDに劣る点が残っている。

「マウス」と「キーボード」は利用者がパソコンを操作するために直接触れる入力装置である。 マウスは、平らな面の上で卵大の装置を動かし、移動量や方向を指示する入力装置である。 外観がネズミに似ていることから、英語でネズミを表すマウスの名で呼ばれている。 一般的なものは表側に複数のボタンが搭載されており、それらを押下することで、画面上のボタンを押下したり、メニューを表示したりできる。 キーボートは、正方形や横長の長方形の小さなボタンが縦横に整然と並んだ、文字や記号などを入力するための装置である。

ソフトウェア

プログラミングの主な目的の一つにソフトウェア開発が挙げらる。 そこで、代表的なソフトウェアの種類について理解を深めよう。 私たちがパソコン上でよく使うソフトウェアとして、オペレーティングシステム（OS、オーエス）とアプリケーションを紹介する。

「OS」は、機器の管理や制御のための基本的な機能を有するソフトウェアである。 同じ機器で動作するソフトウェアが共通して利用する機能を有していたり、システム全体を管理できたりする場合もある。

パソコン用のOSの例として、Windows、macOS（旧Mac OS）、Linux などが挙げられる。 パソコンを利用するためには、これらのOSを導入してから、各種アプリケーションを導入する必要がある。

「アプリケーションソフトウェア」（application software）または「アプリケーション」とは、ある特定の機能や目的のために開発、使用されるソフトウェアである。 利用者がOS上にインストールできるソフトウェアでもある。 例えば、表計算アプリケーションや、画像編集アプリケーションなど、特定の用途のために作られたソフトウェアがこれに該当する。

さらに、動作環境に基づいてアプリケーションを分類することができる。 例えば、WebサーバやWebブラウザ上で動作するものをWebアプリケーション、スマートフォン上で動作するものをスマートフォンアプリケーションなどと呼ぶ。 パソコン上で動作するアプリケーションの中でも、OS上で直接動作するアプリケーションを、Webアプリケーションと対比して、スタンドアロンアプリケーションと呼ぶこともある。

ソフトウェアの開発に使う道具についても確認しよう。 ソフトウェアを構成するプログラムを作成する際には、一般的に「プログラミング言語」（programming language）を使用する。 プログラミング言語とは、プログラマがプログラムを記述、編集するために使用する人工言語である。

ソフトウェアは時代とともに複雑になる一方であり、プログラマがゼロから全てのプログラムを作ることは難しい。 そこで、一般的には、ライブラリやフレームワークという部品を活用することが多い。 「ライブラリ」（library）とは、ある機能を持ったプログラムを他のプログラムから利用できるように部品化し、複数まとめたものである。 そのような部品をさらに集めて、ある領域のソフトウェアに必要な汎用的な機能をまとめた半完成品のことを「ソフトウェアフレームワーク」や「アプリケーションフレームワーク」などと呼び、単に「フレームワーク」（framework）とも呼ぶ。 ライブラリとフレームワークはどちらもソフトウェアの構成要素であるが、ライブラリは部品で、フレームワークは土台であると考えると良い。

Webアプリケーション

先の節で、Webアプリケーションとは、WebサーバやWebブラウザ上動作するアプリケーションであると述べた。 Webアプリケーションは主にクライアントサイドのプログラムとサーバサイドのプログラムから構成される。 クライアントサイドのプログラムは、利用者のPCで実行されているWebブラウザ上で動作するプログラムである。 主にWebアプリケーションの画面を表示したり、利用者からWebアプリケーションに対する操作を受け付けたりする役割を担う。

サーバサイドのプログラムは、利用者のパソコンとは別にネットワーク上に接続されたコンピュータ上で実行される場合が一般的である。 サーバサイドには「データベース」（database）と呼ばれる、情報の蓄積や検索に特化したシステムが含まれる場合も多い。 クライアントサイドのプログラムに入力された情報を受け取って、入力された情報をデータベースに保存したり、外部サービスに情報を渡したり、 外部サービスから受け取った結果やデータベースの情報をクライアントサイドに渡したりする役割を担う。 クライアントサイドのプログラムだけでもWebアプリケーションを構成することはできるが、 ソーシャルネットワークサービスのように、複数の利用者間で情報をやり取りしたりする場合はサーバーサイドのプログラムが必要となる。

例えば、動画配信サービスにおけるクライアントサイドのプログラムとサーバサイドのプログラムの挙動を考えてみよう。 動画配信サービスにログインすると、オススメの動画や新着情報、視聴履歴がWebブラウザ上に表示される。このとき、サーバサイドが、ログイン情報をクライアントサイドから受け取り、ログインした利用者へのオススメの動画や視聴履歴、ログインした時点での新着情報をクライアントサイドへ渡す。クライアントサイドは、サーバサイドから受け取った、オススメの動画や新着情報、視聴履歴の情報を、利用者に見える形で、Webブラウザ上に描画する。

また、利用者がPythonに関する動画を調べようとして、検索入力欄に「Python 入門」と入力して検索を行う。すると、サーバサイドでは、クライアントサイドから「Python 入門」という文字列を受け取り、受け取った文字列に関連する検索結果をクライアントサイドに渡す。そして、クライアントサイドでは、サーバサイドから受け取った検索結果を利用者に見える形でWebブラウザ上に表示する。

スマートフォンアプリケーション

先の節で、スマートフォンアプリケーションとは、スマートフォン上で動作するアプリケーションであると述べた。 いわゆるスマホアプリである。

スマートフォンアプリケーションはスマートフォンに搭載された様々な装置を利用できる。 例えば、カメラアプリは、スマートフォンに搭載されたカメラを使用して写真を撮影する。 カメラに加えて、スマートフォンの動きや傾きを検知するセンサーも使用することで、手ぶれを補正できたり、パノラマ写真を撮影できたりするものもある。

スマートフォンアプリケーションが、Webアプリケーションと同様に、インターネットを介して接続されたサーバを使用する場合がある。 このときスマートフォンは、Webアプリケーションの節で述べたクライアントの役割を果たす。

プログラミング言語

プログラミング言語には様々な種類があり、本講義で扱うPythonは、プログラミング言語の一つである。 本節で、プログラミング言語の分類について説明する。

CPUが直接実行できるプログラムを記述するための言語を「機械語」もしくはマシン語と呼ぶ。 機械語はCPUに扱いやすいように設計されているため、人間が読み書きすることは困難である。 そこで、一般的にプログラマは機械語に変換する仕組みを備えたプログラミング言語を利用する。 機械にとって扱いやすい言語を「低水準言語」もしくは低級言語と呼び、人間にとって扱いやすい言語を「高水準言語」もしくは高級言語と呼ぶ。 プログラミングにおいて低水準や高水準といった言葉は優劣を表すものではない。

一般に、機械語以外のプログラミング言語で記述されたプログラムを「ソースコード」と呼ぶ。 低水準言語の中でも、機械語の命令コードと一対一対応する命令語から構成される言語を「アセンブリ言語」と呼ぶ。 アセンブリ言語で記述されたソースコードを機械語のプログラムに変換するソフトウェアを「アセンブラ」、変換することを「アセンブル」と呼ぶ。 また、高水準言語で記述されたソースコードを機械語のプログラムに変換するソフトウェアを「コンパイラ」、変換することを「コンパイル」と呼ぶ。

低水準言語、高水準言語それぞれで書かれたソースコードをコンピュータが読み取って実行するまでの流れを確認しよう。 低水準言語で書かれたソースコードを実行するには、アセンブラを使用してソースコードをアセンブルする、つまり機械語に変換する。 そして、変換された機械語のプログラムをCPUが実行する。 高水準言語で書かれたソースコードを実行するには、コンパイラを使用してソースコードをコンパイルする、つまり機械語に変換する。 そして、変換された機械語のプログラムをCPUが実行する。

ソースコードは「インタプリタ」を使って実行することもできる。 アセンブラやコンパイラを使い変換した機械語でプログラムを実行する場合は、ソースコード全文を一括して機械語に変換して実行することになる。 一方、インタプリタは、高水準言語で書かれたソースコードを、機械語への変換と同時に1文ずつ実行するソフトウェアである。 ソースコードを書いた後、コンパイルすることなく実行できるため、書いたソースコードの挙動をすぐに確認しながら開発や修正を進めることができる。 しかし、インタプリタによる実行は、コンパイラを使用した場合よりも、実行時の性能が低くなる欠点を有する。 なお、インタプリタで実行するプログラムを記述するための言語をスクリプト言語と呼ぶ。

プログラミング言語は、プログラムを整理する際の考え方によっても分類することができる。 旧来は処理とデータを分けて整理する手続き型プログラミング言語が主流であったが、 現在は処理とデータひとまとめにして整理するオブジェクト指向プログラミング言語が主流である。 その他にも、数学の関数に基づいて整理する関数型プログラミング言語や、 述語論理という数理論理学の概念に基づいて制する論理型プログラミング言語が存在する。 なお、近年では、上記の分類の中で良いところを取り出して組み合わせたハイブリッド言語が普及しつつあり、 プログラミング言語の分類分けの境界線は曖昧になりつつある。

用途によるプログラミング言語の違い

本節では、プログラミング言語の用途ごとに、Python言語を含むいくつかの代表的なプログラミング言語と各言語の代表的なフレームワークについて紹介する。

プログラミング言語の用途は、大きく分けると、Webアプリケーションのクライアントサイドのプログラムを作るための言語、スマートフォンアプリケーションのクライアントサイドのプログラムを作るための言語、Webアプリケーションやスマートフォンアプリケーションなどのサーバーサイドのプログラムを作るための言語に大別できる。その他にも、プログラミング言語の用途は考えられるが、本科目では上記の3つに焦点を当てて、プログラミング言語を紹介する。

すでに述べたとおり、Webアプリケーションのクライアントサイドのプログラムとは、Webブラウザ上で動作するプログラムを指す。スマートフォンアプリケーションのクライアントサイドのプログラムとは、スマートフォン上で動作するプログラムを指す。また、インターネットを介してクライアントプログラムから要求を受けて、命令を実行したり、命令の実行結果を返却するためのプログラムをサーバーサイドのプログラムと呼ぶ。Webアプリケーションであっても、スマートフォンアプリケーションであっても、サーバーサイドのプログラムが存在するケースが多い。

Webアプリケーションのクライアントサイド向けのプログラミング言語

Webアプリケーションにおいて、クライアントサイドのプログラムは主にJavaScriptというプログラミング言語で記述される。 また、Webブラウザ上でWebページを表示する上で、HTMLとCSSというコンピュータ言語が用いられる。 HTMLとCSSはプログラミング言語ではないが、Webア

プリケーションを開発するために必要不可欠な言語であるため、合わせて紹介する。

スマートフォンアプリケーションのクライアントサイド向けのプログラミング言語

スマートフォンアプリケーションのクライアントサイドに使用されるプログラミング言語はOSによって大きく異なる。 AndroidというOS向けには、JavaやKotlinという言語がよく使われる。 iOSやiPadOSというOS向けには、ObjectiveCやSwiftという言語がよく使われる。 また、複数のOS向けに使用できる言語として、Dartがある。

サーバーサイド向けの言語

サーバサイドのプログラムを記述する言語は数多くある。 特にシステム開発の現場では、Java、PHP、Ruby、JavaScript（Node.js）が採用される場合が多い。

プログラミング言語一覧

ここまでの説明で名前を挙げたプログラミング言語を簡単に紹介する。

JavaScript

JavaScriptは、Webページに組み込んだプログラムをWebブラウザ上で実行するために用いられるプログラミング言語で、スクリプト言語の一種である。 HTMLとCSSだけでも、Webページを作ることができるが、JavaScriptと組み合わせることで、 クライアントサイドだけで様々な処理を実行して、Webページの内容を動的に生成することができる。 例えば、JavaScriptを使うことで、サーバサイドのプログラムを用意しなくても、ブラウザ上で動作する電卓を作成することができる。 電卓はHTMLとCSSだけで実現することはできない。

ほとんど全てのブラウザでJavaScriptプログラムが動作するが、逆に、JavaScript以外のプログラムはほとんど動作しない。 そこで、JavaScriptプログラムに変換可能なプログラミング言語を開発することで、ブラウザ上で動作するプログラミング言語が生まれている。 あるプログラミング言語で記述されたプログラムを、同様に高水準言語である別のプログラミング言語のプログラムに変換することをトランスパイルと呼ぶ。

JavaScriptにトランスパイルできる言語が複数開発されているが、最もメジャーな言語は、Microsoft社が開発するTypeScriptである。 JavaScriptはプログラム中に型（データの種類）を明示的に記述しない動的型付け言語であるが、 TypeScriptはプログラム中に型（データの種類）を明示的に記述する静的型付け言語である。 静的型付け言語のほうがプログラムのバグを発見しやすいという利点があるため、 近年、JavaScript言語よりもTypeScript言語の方が好まれるケースが増えている。

JavaScritp向けに様々なフレームワークが存在するが、Webアプリケーション開発においてよく利用されるフレームワークとして、 React, Vue.js, Angularが挙げられる。これらのフレームワークを利用することで、複雑なWebアプリケーションを容易に開発できる。

なお、JavaScriptでサーバーサイドのプログラムを記述することもできるため、 クライアントサイドもサーバーサイドもJavaScriptで記述するという開発方針が取られることも多い。

HTML

HTMLはHypertext Markup Languageの略称である。HTMLはWebページを表現するためのマークアップ言語であり、Webアプリケーションで必ず利用される。 マークアップ言語とは、コンピュータによって処理されるコンピュータ言語の一種であり、データ中に特定の記法を用いて何らかの情報を埋め込むための言語である。 ブラウザはHTMLの内容に基づいてWebページの画面を表示している。そのため、Webページを作るためには、HTML文書を記述する必要がある。

HTMLでは `<` と `>` で囲まれたタグという特殊な記法を用いることで、文書の構造や装飾、ボタンなどを表現できる。 例えば、以下のように「HTMLについて学習しよう」という文字列を `h1` タグで囲むことで、該当文字列を見出しとして表現できる。

```
<h1>HTMLについて学習しよう</h1>
```

CSS

CSSはCascading Style Sheetsの略称である。CSSでは、Webページのレイアウトやスタイルなどを記述するためのスタイルシート言語である。スタイルシートとは、文書データの見た目に関する情報を表現するための記述のことであり、スタイルシートを記述するための言語がスタイルシート言語である。

CSSでは、HTMLで表現したWebページの構造に対して、文字のサイズや行間などの文字組みの設定や、 背景色や文字色などの色の設定、余白などのレイアウトの設定などをすることができる。

例えば、以下のように記述することで、HTMLの `h1` 要素の文字色を赤色に設定できる。

```
h1 {
  color: red;
}
```

Java

Javaは、主にサーバーサイドのプログラムやAndroid向けのスマートフォンアプリケーションを開発する際に利用されるプログラミング言語である。 Javaは、 `Write once, run anywhere` （「一度記述すれば、どこでも動く」という意味）という標語のもとで開発された言語であり、様々なプラットフォームやOSで動作するという特徴を備えている。

サーバサイドを開発するための著名なJavaフレームワークとしてSpring FrameworkやJakarta EEなどが挙げられる。 なお、JavaとJavaScriptは名前は似ているが、全く別のプログラミング言語である。

Kotlin

Kotlinは、Javaの代替言語として開発されたプログラミング言語であり、Javaプログラムが動作する環境で、Kotlinプログラムも動作するという特徴を備えている。 他のプログラミング言語と比較してJava言語の進化が遅かったたため、Javaの欠点を改善して、先進的な機能を備えたプログラミング言語となっている。 近年では、Android向けのスマートフォンアプリケーションを開発する際に用いられるケースが増えている。

Objective-C

Objective-Cは、プログラミング言語の1つであるC言語にオブジェクト指向プログラミングに関する機能を追加した言語である。 主にMacOS向けのアプリケーションやiOS向けのスマートフォンアプリケーションを開発する際に使用されるプログラミング言語である。

Swift

Swiftは、Objective-Cの代替言語として開発されたプログラミング言語である。 JavaとKotlinの関係性に似ているが、Javaの開発企業のKotlinの開発企業が異なるのに対して、 Objective-CとSwiftの開発企業はどちらもApple社であり、Objective-Cの後継者という側面が強い。 Objective-Cと同様に、主にMacOS向けのアプリケーションやiOS向けのスマートフォンアプリケーションを開発する際に使用される。

Dart

Dartは、Javaのように様々なプラットフォームおよびOSで動作するプログラミング言語である。 Dartが発表された当初はあまり注目が集まらなかったが、Flutterという様々なプラットフォームおよびOSで動作するアプリケーションを開発するためのフレームワークに採用されたことがきっかけとなり、近年、急速に注目を集めている言語である。 主にFlutterを用いたアプリケーションを開発する際に使用される。

PHP

Webアプリケーションのサーバーサイドのプログラムを作成することに力点が置かれたプログラミング言語であり、スクリプト言語の一種である。 当初は手続き型プログラミング言語として開発されたが、近年では、オブジェクト指向プログラミング言語の要素も取り込まれて進化している。 サーバサイドのプログラムを開発するための著名なPHPフレームワークとして、LaravelやCakePHPなどが挙げられる。

Ruby

プログラミング言語の多くは海外の開発者や海外の企業によって開発されているが、Rubyは日本の開発者が開発したプログラミング言語であり、スクリプト言語の一種である。 Pythonと同様に全てのデータをオブジェクトとして扱うオブジェクト指向プログラミング言語である。 RubyはPythonと比較されることが多いが、Rubyは多様な方法でプログラムを記述できることに重点が置かれているのに対し、 Pythonは限られた方法でプログラムを記述できることに重点が置かれている。 書き方のパターンが少ないという点では、Pythonの方が初学者向けであるとみなされるケースが多い。

Python言語

いくつかのプログラミング言語を紹介してきたが、以降ではPython言語について紹介をする。 Pythonはオブジェクト指向プログラミング言語であり、また、コンパイルを必要としないスクリプト言語でもある。 Rubyの説明でも取り上げたが、Pythonはあるプログラムを記述しようとした際に、 取り得る書き方のパターン数が少なくなるように開発されている。 そのため、開発者が覚えなければならない要素が比較的少ない、初学者に優しい言語であるとされる。

Pythonには古くから高速な数値計算を行うためのライブラリが存在しており、 そういった土壌がきっかけとなって、近年では様々なPython向けの機械学習および人工知能（AI）のライブラリが開発されている。 今では、AIといえばPythonといえるほどに、AIとの親和性が取り上げられるプログラミング言語となっている。

昨今のAIブームの前は、サーバーサイドのプログラムを開発するためによく利用されていた。 サーバサイドを開発するための著名なPythonフレームワークとして、DjangoやFlaskが挙げられる。

それでは、Pythonのサンプルプログラムを見てみよう。以下のプログラムを実行すると、 `Hello World` という文字が画面に表示される。

```
print("Hello World")
```

```
Hello World
```

`print("Hello World")` は1つのステートメント（statement; 日本語では文とも呼ぶ。）である。ステートメントとはPythonを含めた多くのプログラミング言語における命令を指す。Pythonでは最小単位の命令を単純ステートメント (simple statement; 単純文)と呼び、 `print("Hello World")` は単純ステートメントでもある。 `print` は組み込み関数と呼ばれるPythonがあらかじめ提供するプログラムの部品の一つであり、 `print()` 関数に `Hello World` という文字列を渡して実行することで、 `print()` 関数が `Hello World` という文字列を表示してくれる。

Pythonでは、基本的に1行に1ステートメントを記述する。そのため、2つの命令から構成されるプログラムを作成する際は、2行のプログラムを記述することとなる。次のプログラムを見てみよう。

```
print("Hello World")
print("Hello World")
```

```
Hello World
Hello World
```

上のプログラムを実行すると、 `Hello World` という文字列が2回表示される。多くのプログラミング言語と同様に、Pythonでは上から順番に1行ずつプログラムが実行される。

Pythonではセミコロン(`;`)を使って、1行に複数の単純ステートメントを記述することもできる。以下で、上のプログラムを1行で記述したサンプルプログラムを示す。

```
print("Hello World"); print("Hello World")
```

```
Hello World
Hello World
```

先ほどと同様に `Hello World` という文字列が2回表示される。なお、セミコロン(`;`)を使って、1行に複数の単純ステートメントを記述することは稀である。基本的には使わないようにしよう。

Online Judgeの使い方

Online Judgeでの問題の提出方法について紹介する。

まず、問題画面に移ろう。

現在表示しているページの一番上まで移動すると、右側に問題一覧が表示されている。問題一覧の中の `問題1` を選択する。

すると、問題画面に遷移して、問題が表示される。

画面の左側に問題が表示されている。

今回、説明を行う問題では、 `Hello World` を出力するプログラムをPythonで記述して提出することが求められている。

画面の右側の黒い部分にPythonのソースコードを記述できる。

今回の問題の正解となるソースコードを記述する。正解のソースコードは以下である。

```
print("Hello World")
```

画面右側の下部にある テスト実行 や 提出 や矢印をクリックすることで、プログラムのテスト実行や提出をする画面を表示できる。

画面の右側の テスト実行 で、提出せずに記述したプログラムを試しに実行できる。 実行 ボタンをクリックすることで、記述したプログラムを実行できる。 入力を求められる問題であれば、 入力 に記入するした内容をプログラムの入力として実行する。

今回の問題では、入力は求められていないため、プログラムを記述できたら、 実行 ボタンを押してみよう。

プログラム実行後に出力された値は、 出力 に表示される。エラーが発生した場合は、エラーメッセージが エラー に表示される。 変数 には、プログラム実行後に、ソースコード中に定義した変数に格納されている値が表示される。

今回の問題では、 `Hello World` が出力に表示されるはずである。

`テスト実行` を行って、正解となるプログラムが書けたと思ったら、 `提出` からソースコードを提出してみよう。 `提出` ボタンをクリックすることで、記述したプログラムを提出できる。正しくプログラムを書けていれば、 `判定` の部分に `AC` と表示される。 `AC` 以外が表示されている場合は、プログラムが間違えているので、修正して再度提出しよう。

以上を踏まえて、実際に `問題1` を解いてをプログラムを提出してみよう。

問題1

問題文

`Hello World` を出力するプログラムを作成せよ。

解答の雛形

```
# ここに解答を入力
```

2章: 数値計算と変数

演算子と数値計算

本節ではPythonで利用可能な演算子を紹介する。演算子（オペレータ）とは、数式などで計算処理を表す記号である。例えば、数学で `+` 演算子は演算子の左側と右側の数を足し合わせる計算を意味する。演算子の適用対象となる数などを被演算子（オペランド）と呼ぶ。例えば、`1 + 2` という式において、 `+` は演算子であり、 `1` と `2` は被演算子である。なお、式を構成する要素を項と呼ぶことから、被演算子を項と呼ぶこともある。

Pythonを含む多くのプログラミング言語では、数学で扱っている多数の演算子を利用できる。 早速、Pythonにおける四則演算の使い方を見ていこう。 以下のサンプルプログラムでは、加算・減算・乗算・除算を行っている。

```
print(3 + 2)
print(3 - 2)
print(3 * 2)
print(3 / 2)
```

```
5
1
6
1.5
```

`3 + 2` と `3 - 2` の表記は数学と同じである。 しかし、乗算の記号（×）はアスタリスク (`*`) で、除算の記号（÷）はスラッシュ (`/`) で示す点に注意しよう。

Pythonでは、加算・減算・乗算・除算以外にも様々な演算子を利用できる。 以下のサンプルプログラムでは、剰余演算子・切り捨て除算子・べき乗演算子を使用している。

```
print(3 % 2)
print(3 // 2)
print(3 ** 2)
```

```
1
1
9
```

`%` は剰余演算の演算子である。 `a % b` の場合、 `a` を `b` で割ったときの余りを計算する。 `3` を `2` で割ると `1` が余りとなるため、 `3 % 2` の計算結果は `1` となる。

`//` は切り捨て除算の演算子である。 `a // b` の場合、 `a` を `b` で割った結果の整数部分を算出する。 `3` を `2` で割った結果は `1.5` となり、 `1.5` の整数部分は `1` となるため、 `3 // 2` の結果は `1` となる。

`**` はべき乗演算の演算子である。 `a ** b` の場合、 `a` を `b` 乗した結果を算出する。 `3` を `2` 乗した結果は `9` となるため、 `3 ** 2` の結果は `9` となる。

他にもいくつか演算子は存在しており、次章以降で他の演算子も取り上げる。

数学と同様に演算子には優先順位があり、基本的に数学と同じ優先順位を持っている。演算子の優先順位を学ぶために、次のプログラムの実行結果を考えてみよう。

```
print(104 * 11 + 170 * 12)
```

```
3184
```

実行結果を当てられただろうか？演算子の優先順位が分かりやすくなるように括弧をつけて、もう一度、動かしてみよう。

```
print((104 * 11) + (170 * 12))
```

```
3184
```

先ほどと同じように `3184` と表示されたことを確認できた。両方のプログラムは同じ意味を持っている。

もう少し複雑な例を見てみよう。次のプログラムの実行結果を考えてみよう。

```
print(100 + 80 * 8 / 4 ** 2)
```

```
140.0
```

実行結果を当てられただろうか？演算子の優先順位が分かりやすくなるように括弧をつけて、もう一度、動かしてみよう。

```
print(100 + (80 * 8 / (4 ** 2)))
```

```
140.0
```

先ほどと同じように `140.0` と表示されたことを確認できた。

これまでに説明した演算子の優先順位は以下の表の通りだ。

優先順位	演算子
高	**
	*, /, //, %
低	+, -

同じ順位の演算子であれば、数学と同様に左側にある演算子から順番に計算する。

改めて、 `104 * 11 + 170 * 12` と `100 + 80 * 8 / 4 ** 2` の計算の順序について考えてみよう。

`104 * 11 + 170 * 12` の計算順序は、加算の演算子（ `+` ）と乗算の演算子（ `*` ）の優先順位を比較して、乗算の演算子の方が優先順位が高いため、 `104 * 11` と `170 * 12` を先に計算する。 そのあとに、 `104 * 11` と `170 * 12` の結果である `1144` と `2040` を加算して、 `3184` という計算結果が得られる。

`100 + 80 * 8 / 4 ** 2` の計算順序は、加算の演算子（ `+` ）と乗算の演算子（ `*` ）と除算の演算子（ `/` ）とべき乗の演算子（ `**` ）の優先順序を比較して、べき乗の演算子が最も優先順位が高いため、 `4 ** 2` を先に計算する。 `4 ** 2` の結果は `16` であるので、 `100 + 80 * 8 / 16` を計算することになる。 加算の演算子と乗算の演算子と除算の演算子の優先順位を比較して、乗算の演算子と除算の演算子の優先順位が加算の演算子よりも高いため、 `80 * 8 / 16` を計算することになる。また、乗算の演算子の方が、左側にあるので、まず `80 * 8` を計算する。 `80 * 8` の結果は `640` であるので、続いて、 `640 / 16` を計算する。 `640 / 16` の結果は `40` であるので、最後に、 `100 + 40` を計算して、 `140` という計算結果が得られる。

変数と代入

Pythonには多数のプログラミング言語と同様に変数という仕組みを持っている。

まず、変数に関連する言葉の意味を理解しよう。 Pythonではあらゆるデータがオブジェクトとして表現される。 Pythonのオブジェクトは値、型、同一性を持っている。 値とは、オブジェクトが有するデータであり、データの種類のことを型と呼ぶ。 同一性については、第4章で説明する。 変数とは、オブジェクトを一時的に記憶する領域に名前を付けたものである。 変数にオブジェクトを記憶させることを「変数にオブジェクトを代入する」と表現し、 また、変数からオブジェクトを読み取ることを「変数を参照する」と表現する。

ただし、オブジェクトのことを単に値と呼ぶこともあるため、 「変数にオブジェクトを代入する」の代わりに「変数に値を代入する」と表現することもある。 Pythonでは全てのデータはオブジェクトで表現されることから、上記の2文は同じ意味を持つ。 本講義においては、特段の理由がない限り、「変数に値を代入する」という表現を使うこととする。

さて、変数を使えば、ある計算の結果を記憶して、別の計算で利用することができる。 次のサンプルプログラムを通して、変数の概要を理解しよう。

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

```
1
```

`a = 1` は、 `a` が `1` と等しいという意味ではなく、 `a` という変数に `1` という値を代入するという意味である。 `=` は代入演算子と呼び、右辺の値を左辺の変数に代入する演算子である。 数学の等式で使う `=` とは意味が異なる点に注意しよう。 数学では `a = 1` も `1 = a` も同じ意味だが、Pythonでは意味が異なる。 Pythonでは、1に値を代入することはできないため、 `1 = a` はエラーになる

変数を参照する際は、単に変数名を式などの中で記述すれば良い。 `print(a)` では、変数 `a` に格納されている値を参照している。 `a` には `1` が代入されているため、 `print(a)` は `1` を表示する。

変数を参照する際は、定義された変数を参照しなければならない。 変数を定義する方法はいくつかあるが、初めて変数に値を代入することで、その変数を定義することができる。 上のプログラムでは `a = 1` というステートメントによって、変数 `a` を定義している。 1行目で変数 `a` を定義しているので、2行目の `print(a)` で変数 `a` を参照できる。

変数に何度でも代入することができるが、変数は最後に代入された値のみを記憶する。 次のプログラムの実行結果を考えてみよう。

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

```
2
```

第1章で説明したとおり、プログラムは上から順番に実行されるため、まず、 `a = 1` が実行される。 `a = 1` が実行されると、 `a` は `1` を記憶している状態となる。 続いて、 `a = 2` が実行される。 `a` は `1` を記憶しているが、 `a = 2` が実行された結果、 `a` は `2` を記憶する状態となる。 最後に、 `print(a)` が実行される。 `a` は `2` を記憶しているため、 `print(a)` は `2` を表示する。

変数は、その瞬間に記憶している値に基づいた値を計算して、その計算結果を代入することで、変数の値を更新できる、 次のプログラムの実行結果を考えてみよう。

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

```
2
```

まず、 `a = 1` が実行され、 `a` の中に `1` が入る。 続いて、 `a = a + 1` が実行される。

`=` の右辺に注目しよう。 `=` の右辺は、 `a + 1` である。直前のプログラムから、 `a` の中に `1` が入っているため、 `1 + 1` と直すことができる。 そのため、 `a = a + 1` は、 `a = 1 + 1` と直すことができる。 `1 + 1` の計算結果は、 `2` であるため、 `a = a + 1` の実行すると、 `a` には新たに `2` が代入される。

最後に、 `print(a)` が実行される。 `a` には、 `2` が入っているため、 `print(a)` は `2` を出力する。

変数を使うと、計算式を分解できる。次のプログラムの実行結果を考えてみよう。

```
a = 80
a = a * 8
a = a / 4
a = a + 100
print(a)
print(100 + (80 * 8 / 4))
```

```
260.0
260.0
```

`print(a)` の結果と `print(100 + (80 * 8 / (4 ** 2)))` の出力結果は同じになる。

`a = 80` の結果、 `a` の中に `80` が入る。 `a = a * 8` の結果、 `a` に `640` が入る。 `a = a / 4` の結果、 `a` に `160` が入る。 `a = a + 100` の結果、 `a` に `260` が入り、 `print(a)` の結果、 `260` が出力される。

代入演算子の左辺と右辺に同じ変数が現れて少し冗長に思うだろう。累算代入演算子を使うと、上記のサンプルプログラムのような代入処理を簡略化できる。次のサンプルプログラムを見てみよう。

```
a = 80
a *= 8
a /= 4
a += 100
print(a)
print(100 + (80 * 8 / 4))
```

```
260.0
260.0
```

上記のプログラムでも `print(a)` の結果と `print(100 + (80 * 8 / (4 ** 2)))` の出力結果は同じになる。 `*=` や `/=` 、 `+=` を累算代入演算子と呼ぶ。 累算代入演算子は、左辺と右辺の値を使い、加減乗除などの演算した結果を左辺に代入する。累算代入演算子1つで、演算の処理と代入の処理の両方を実行する。 サンプルプログラムにおいて、 `a *= 8` は `a = a * 8` と同一の処理を行い、 `a /= 4` は `a = a / 4` と同一の処理を行い、 `a += 100` は `a = a + 100` と同一の処理を行っている。

これまで説明した演算子は、演算子の2つの被演算子を持っていた。2つの被演算子を持つ演算子のことを二項演算子 (binary operator) と呼ぶ。Pythonは、二項演算子の他にも、単項演算子 (unary operator) と三項演算子 (ternary operator) を提供している。単項演算子と代入を組み合わせたサンプルプログラムを見てみよう。

```
a = 5
print(a)
print(-a)

a = -a
print(a)
print(-a)
```

```
5
-5
-5
5
```

上記プログラムでは、まず、変数 `a` に `5` を代入している。 代表的な単項演算子は `-` であり、 `-5` や `-a` などの例が挙げられる。 数学と同じ意味であり、 `-` の単項演算子の被演算子の値の符号を反転させる、 言い換えると、被演算子の値に `-1` をかける演算子である。

上2つの `print` では、それぞれ、 `5` と `-5` を表示している。 その後、変数 `a` の値を `-a` 、つまり、 `-5` で更新している。 そのため、下2つの `print` では、それぞれ、 `-5` と `5` を表示している。

`del` 文を使うと、定義した変数を削除することができる。削除した後に変数を参照するとエラーとなる。以下のプログラムでエラーとなる様子を見てみよう。

```
a = 1
print(a)
del a
print(a)
```

```
1
```

2つ目の `print()` 関数では変数 `a` が存在しないので、 `a` という名前が定義されていないというエラーが表示されている。

変数の名前には様々な単語や文字を採用できるので、誰が読んでも分かりやすいように名前をつけよう。次にプログラムを見てみよう。

```
a = 5
b = 9
print(a * b)

width = 5
height = 9
print(width * height)
```

```
45
45
```

上記のプログラムは二通りの方法で、横幅が5で高さが9の長方形の面積を計算しているが、上の書き方よりも下の書き方のほうが、プログラムの意図が伝わりやすいだろう。 その理由は、下の書き方では、変数の名前に横幅(`width`)と縦幅(`height`)しているため、変数名から面積を計算していることが理解しやすいためである。

プログラミングを学習している際は、自分のプログラムを自分で読むケースが大半ではあるが、実務等では複数人でソフトウェアを開発して、他者が自分のプログラムを読むケースが大半となる。 また、1人だけでソフトウェア開発をしていたとしても、長期に渡り開発をしていると、自分で書いた過去のプログラムを自分で読んで内容を理解し直す必要が増えてくる。 その際に、分かりにくいプログラムを書いていると、自分で書いたプログラムであっても理解できなくなる。できる限り分かりやすいプログラムを書くように心がけよう。

変数名には、以下の文字を使うことができる。

- 大文字のアルファベット
- 小文字のアルファベット
- `0` から `9` までの数字
- アンダースコア(`_`)

また、変数名の命名には以下のルールがある。

- 変数名の先頭に数字を使用できない。
- 予約語(ifやforなど)を使用できない。

予約語（キーワード; keyword）とは、プログラミング言語の開発者が、変数名などの識別名として使用できない単語やフレーズである。

以上のルールから、 `a1` は問題ないが、 `1a` はエラーになる。また、第3章で扱う、繰り返しや条件分岐を行う際に使用する単語である `for` や `if` も、変数名に使用することはできない。

型とリテラル

Pythonでは、整数や実数、真偽値（ブール値）、文字列、リストなど様々な種類のデータを扱える。既に述べたとおりデータの種類を型と呼ぶ。 また、プログラム中に直接記述した値および値の表記のことをリテラル (literal) と呼ぶ。 型は、Pythonに組み込まれている型と、プログラム作成者が独自に定義した型の二種類が存在しており、前者のことを標準型と呼ぶ。 本節では様々な標準型とリテラルを紹介する。

数値と真偽値

まず、単一の値から構成されるシンプルな型を見ていこう。 整数型、実数型、真偽値型（ブール型やブーリアン型とも呼ぶ）を扱っているプログラムを見てみよう。

```
print(123)
print(0.123)
print(True)
```

```
123
0.123
True
```

Pythonの整数には下限や上限はない。一方、実数は浮動小数点数であるため、精度や大きさに限界がある。真偽値には真と偽の二種類の値があり、それぞれ `True` と `False` と記述する。 なお、 `123` や `0.123` , `True` , `"Hello"` のようにプログラム中に記述されているデータがリテラルだ。

複数の値から構成される型

整数型、実数型、真偽値型以外にも様々な型およびリテラルがある。 以降で、複数の値から構成されるデータの型およびリテラルを紹介する。 複数の値から構成される標準型には、シーケンス型、集合型、辞書型が存在する。 本節では、これらの型の概要とリテラルの表記方法を説明する。

シーケンス型は、一度生成したら内容を変更できない変更不可能なシーケンス型（immutable sequence）と変更可能なシーケンス型（mutable sequence）の2種類に分けることができる。 本節では、変更不可能なシーケンス型である文字列型 (string) とタプル型 (tuple) を紹介した後に、変更可能なシーケンス型であるリスト型 (list) を紹介して、最後に、集合型 (set) と辞書型 (dictionary) を紹介する。

文字列

まず、文字列型を紹介する。文字列型は文字データが集まって構成される型である。 既に述べたとおり、文字列型は変更不可能なシーケンス型なので、一度生成した文字列の内容を実行中に変更することはできない。 基本的に、文字列リテラルはダブルクォーテーション (`"`) かシングルクォーテーション (`'`) で該当文字列を囲うことで表記できる。

```
print("Hello World!")
print('Hello World!')
```

```
Hello World!
Hello World!
```

プログラムの実行結果のように、ダブルクォーテーションを使ってもシングルクォーテーションを使っても、同じように文字列を表記できる。

ダブルクォーテーションとシングルクォーテーションの違いは、文字列内のダブルクォーテーションおよびシングルクォーテーションの表記方法にある。次のプログラムを見てみよう。

```
print("\"おはよう\"と言った")
print('"おはよう"と言った')

print("'おはよう'と言った")
print('\'おはよう\'と言った')
```

```
"おはよう"と言った
"おはよう"と言った
'おはよう'と言った
'おはよう'と言った
```

ダブルクォーテーションで記述している文字列リテラル中にダブルクォーテーションを加えたい場合は、文字列の終わりを示すダブルクォーテーションと区別するために、 `\"` と記述する必要がある。一方、ダブルクォーテーション中にシングルクォーテーションを加えたい場合は、単に `'` と記述すれば良い。

逆に、シングルクォーテーションで記述している文字列リテラル中でシングルクォーテーションを加えたい場合は、文字列の終わりを示すシングルクォーテーションと区別するために、 `\'` と記述する必要がある。一方、シングルクォーテーション中にダブルクォーテーションを加えたい場合は、単に `"` と記述すれば良い。

`\"` や `\'` のようなバックスラッシュから始まる特別な文字をエスケープ文字と呼ぶ。エスケープ文字は直接入力することが困難な文字を記述するための特別な表記だ。 `\"` や `\'` 以外にも様々なエスケープ文字がある。例えば、改行は `\n` 、タブは `\t` 、バックスラッシュは `\\` で表記できる。 次のプログラムで、これらのエスケープ文字による文字列の表記方法を確認しよう。

```
print("\"色々\"な\tエスケープ\\文字\nがあるよ")
print('\"色々\"な\tエスケープ\\文字\nがあるよ')
```

```
"色々"な エスケープ\文字
があるよ
"色々"な エスケープ\文字
があるよ
```

ダブルクォーテーションでもシングルクォーテーションでも、エスケープ文字が示す文字の内容は同じである。なお、シングルクォーテーションで囲まれた文字列リテラルの中で `\"` を使う必要性はないが、エスケープ文字を使っても `"` を表現できる。

バックスラッシュを多用するような文字列を表現する際は、エスケープ文字の記法を使って記述すると手間がかかることがある。そのようなケースでは、ダブルクォーテーションまたはシングルクォーテーションの直前に `r` を付けると、エスケープ文字を無効化して、バックスラッシュをそのまま表示することができる。次のプログラムで、 `r` による表現を確認してみよう。

```
print("\"色々\"な\tエスケープ\\文字\nがあるよ")
print(r"\"色々\"な\tエスケープ\\文字\nがあるよ")
```

```
"色々"な エスケープ\文字
があるよ
\"色々\"な\tエスケープ\\文字\nがあるよ
```

上のプログラムではダブルクォーテーションの例を示したが、シングルクォーテーションも同じように動作する。

続いて、改行を含む文字列の表記方法を見てみよう。エスケープ文字以外に改行などを含んだ文字列リテラルを表記する方法として、`"""` もしくは `'''` を使う方法がある。次のプログラムで `"""` の例を見てみよう。

```
print("---")
print("  あいうえお\n  かきくけこ\n  \"や\"\"も使える")
print("---")


print("---")
print("""  あいうえお
  かきくけこ
  "や""も使える""")
print("---")
```

```
---
  あいうえお
  かきくけこ
  "や""も使える
---
---
  あいうえお
  かきくけこ
  "や""も使える
---
```

上の `print()` 関数ではエスケープ文字を使っているが、下の `print()` 関数ではエスケープ文字を使わずに `"""` の中で改行やダブルクォーテーションを表記している。 なお、 `"""` の代わりに `'''` を使っても全く同じことを実現できる。

`"""` に囲まれている文字列の中で改行をしても、行の末尾に `\` を追加することで、文字列リテラルの中に改行を含まれないようにすることができる。次のプログラムを見てみよう。

```
print("---")
print("""
    あいうえお
    かきくけこ
    "や""も使える
""")
print("---")

print("---")
print("""\
    あいうえお
    かきくけこ\
    "や""も使える\
""")
print("---")
```

```
---

    あいうえお
    かきくけこ
    "や""も使える

---
---
    あいうえお
    かきくけこ    "や""も使える
---
```

上の `print` 関数では、 `"""` で囲まれた文字列の中に改行が5つあり、文字列リテラルの中にも全ての改行が含まれている。一方、下の `print` 関数では、 `あいうえお` 以外の行の末尾に `\` がついているため、 `あいうえお` と `かきくけこ` 以外には改行が含まれない文字列リテラルになっている。

ソースコード上では改行をしていても、文字列リテラルの中に改行を含めない方法がある。文字列リテラル同士を並べて書くと、1つの文字列リテラルに連結することができる。 例えば、 `"あいう" "えお"` と `"あいうえお"` はどちらも同じ `あいうえお` という文字列を表すリテラルである。次のプログラムを見てみよう。

```
print("とてもとてもとてもとてもとてもとてもとても長い文字列を2行に分けて書くことができる")
print("とてもとてもとてもとてもとてもとてもとても長い文字列を"
    "2行に分けて書くことができる")
```

```
とてもとてもとてもとてもとてもとてもとても長い文字列を2行に分けて書くことができる
とてもとてもとてもとてもとてもとてもとても長い文字列を2行に分けて書くことができる
```

2つの文字列リテラルの間に改行を入れても、文字列リテラルには含まれない。文字列リテラルに改行を含めずに、ソースコード上で文字列を2行で表現することができる。

文字列リテラル同士の連結であれば、文字列リテラルを並べるだけで良いが、変数に入っている文字列と文字列リテラルを結合する際などは、単に2つを並べるだけでは結合できない。そのようなケースでは、 `+` 演算子を使う必要がある。次のプログラムを見てみよう。

```
s = "文字列を"
print(s + "+で結合できる")
```

```
文字列を+で結合できる
```

`+` 演算子を使えば、左辺と右辺の文字列を結合して、新しい文字列を生成することができる。左辺や右辺は文字列型の値であれば、どんなものでも使える。

文字列では `+` 演算子に加えて、 `*` 演算子も使うことができる。 `*` 演算子を使えば、左辺の文字列を右辺の回数分だけ繰り返した文字列を生成できる。次のプログラムを見てみよう。

```
print("連結したり「" + "繰り返し" * 3 + "」たりできる")
```

```
連結したり「繰り返し繰り返し繰り返し」たりできる
```

演算子の優先順位は `*` の方が `+` よりも高いため、まず、 `"繰り返し" * 3` が実行されて、 `"繰り返し繰り返し繰り返し"` が生成される。その後、 `"連結したり「" + "繰り返し繰り返し繰り返し" + "」たりできる"` が実行されて、 `"連結したり「繰り返し繰り返し繰り返し」たりできる"` が生成される。

数値計算で紹介した累算代入演算子を使うこともできる。次のプログラムを見てみよう。

```
str = "繰り"
str += "返し"
str *= 3
print("連結したり「" + str + "」たりできる")
```

```
連結したり「繰り返し繰り返し繰り返し」たりできる
```

先ほどと同じ文字列を生成できていることを確認できた。 なお、 `str += "返し"` では文字列の内容を変更しているように見えるが、 `"繰り返し"` という新しい文字列を生成して、その値を `str` 変数に再代入している。 文字列の内容を変更することはできないが、新しい文字列を再代入することはできる点に注意しよう。

変更不可能なシーケンス型は、値を代入している変数に再代入できないという意味ではなく、 生成したシーケンス（上記例では文字列）の内容を書き換えられないということである。 「内容を書き換えられない」の意味するところ詳細は、第5章で説明する。

タプル

続いて、タプル型のリテラルを使ったプログラムを見てみよう。

```
print((1, 2, 3))

a = (1, 2, 3)
print(a)
```

```
(1, 2, 3)
(1, 2, 3)
```

タプルのリテラルはカンマ区切りで記述された複数の値を丸括弧（ `(` と `)` ）で囲うことで表記できる。 タプルは順序を保って複数の値を格納する。 文字列と同様に変更不可能なシーケンス型であるため、生成したタプルの値の内容を変えたり、値の個数を変えたりすることはできない。なお、タプルには格納する値の型は全て同じでもバラバラでもどちらでも良く、どちらのケースでも使われる。

リスト

続いて、リスト型のリテラルを使ったプログラムを見てみよう。

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

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

リストのリテラルはカンマ区切りで記述された複数の値を大括弧（ `[` と `]` ）で囲うことで表記できる。 リストは順序を保って複数の値を格納する。 リストは変更不可能なシーケンス型であるため、リスト生成後に、リストが格納している値を変更したり、値を追加したり削除したりすることもできる。 一般的には、リストに同じ型の値だけを格納するが、様々な型の値を格納することもできる。

集合

続いて、集合型のリテラルを使ったプログラムを見てみよう。

```
print({1, 2})
```

```
{1, 2}
```

集合はリストの大括弧（ `[` と `]` ）を波括弧(`{` と `}`)に置き換えることで表記できる。 集合もリストと同じように複数の値を格納することができ、後から格納する値の個数を増やしたり減らしたりすることもできる。 また、リスト同じように同じ型の値を格納するケースが多い。 ただし、リストとは異なり、値の順序は保持せず、また、重複して同じ値を格納することはできない。 集合を使うと、集合の中に特定の値が含まれているか否かを高速に判断することができる。

辞書

最後に、辞書型のリテラルを使ったプログラムを見てみよう。なお、辞書はマッピング (mapping) とも呼ばれる。

```
print({"気温": 23, "湿度": 60})
```

```
{'気温': 23, '湿度': 60}
```

辞書はこれまで紹介した複数の値を格納する型のリテラルと異なり、キーとバリューという2つの値のペアを1要素として、複数のペアから構成される型である。 キーに関しては集合と同じように、重複した値を格納することはできない。もしも重複したキーが存在する場合は、最後に保存したキーとバリューのみが記憶される。 なお、Python 3.7より前ではキーの順序は維持されなかったが、3.7以降では維持されるようになった。辞書で単語名から意味を調べるといった用途と同じように、キーからキーに紐づくバリューを取得するために使われる。

次のプログラムで同じキーが2回現れる辞書リテラルを使ったプログラムを見てみよう。

```
print({"気温": 23, "湿度": 60, "気温": 24})
```

```
{'気温': 24, '湿度': 60}
```

最後に記述した `"気温": 24` のみが記憶されていることを確認できる。

問題1

問題文

123456を7乗してから8倍して9で切り捨て除算した数を表示するプログラムを作成せよ。

問題2

問題文

123456を7乗した値Xと、Xを8倍した値Yと、Yを9で切り捨て除算した値Zについて、X,Y,Zの順に1行ずつ数値を表示するプログラムを作成せよ。

問題3

問題文

123と45の和を67の8乗の9による剰余で切り捨て除算した数を表示するプログラムを作成せよ。

問題4

問題文

123と45を加算した値Xと、67の8乗を9で割ったとき余りの値Yと、XをYで切り捨て除算した値Zについて、X,Y,Zの順に1行ずつ数値を表示するプログラムを作成せよ。

問題5

問題文

はじめに変数Xに7を代入し次の演算を順に行いなさい：Xに8を加算、XにXを乗算、Xを5で切り捨て除算。また、代入後および各演算後にxの値を出力せよ。

問題6

問題文

はじめに変数Xに26を代入し次の演算を順に行いなさい：XにXの2乗を加算、Xから2を減算、XにXの26による剰余を代入。また、代入後および各演算後にxの値を出力せよ。

問題7

問題文

`very` という文字列を1000回繰り返した文字列を表示するプログラムを作成せよ。

問題8

問題文

`very` という文字列を1000回繰り替えした文字列と、 `easy` という文字列を1000回繰り返した文字列を表示するプログラムを作成せよ。

3章: 制御フロー

リストの基礎

リストは複数のデータを集めたデータ型である。 まず、リストから各要素を参照する方法を説明する。

```
colors = ["red", "green", "blue"]
print(colors)
print(colors[0])
print(colors[1])
print(colors[2])
```

```
['red', 'green', 'blue']
red
green
blue
```

このサンプルプログラムの `colors` は `"red", "green", "blue"` の順番で、3つの文字列を格納しているリストである。 1つ目の要素を参照する際は `colors[0]` 、2つ目の要素を参照する際は `colors[1]` 、3つ目の要素を参照する際は `colors[2]` と記述する。 大括弧内の `0` や `1` 、 `2` のことをインデックス（添字）と呼ぶ。 Pythonを含めた多くのプログラミング言語では、 `0` 始まりのインデックスを採用しているため、1つ目の要素を参照するために `0` を使う必要がある。 我々は日常的に `1` 始まりで物事を考えることが多いため、プログラミングをする際は、 `0` 始まりのインデックスに注意をしてプログラムを作成するようにしよう。

リストでは、 `+` と `*` 演算子を利用できる。まず、 `+` 演算子による、リスト同士の連結から確認しよう。

```
print([0, 1, 2] + [3, 4])
print(["red", "green", "blue"] + [3, 4])

list = ["red", "green", "blue"]
print(list)
list += [3, 4]
print(list)
```

```
[0, 1, 2, 3, 4]
['red', 'green', 'blue', 3, 4]
['red', 'green', 'blue']
['red', 'green', 'blue', 3, 4]
```

`+` 演算子を用いると、左右のリストを連結できる。左右の項はどちらもリストである必要がある。 前章で説明したとおり、リストの要素は全てデータ型であるケースが一般的ではあるが、上記のサンプルプログラムの2つ目の `print` のように異なるデータ型が入り混じっていても構わない。 また、 累算代入演算子 `+=` を使うこともできる。

続いて、 `*` 演算子について見てみよう。 `*` 演算子を使うと、リストを繰り返し結合することができる。

```
print([1] * 5)
print(["red", "green"] * 3)

list = ["red", "green"]
print(list)
list *= 3
print(list)
```

```
[1, 1, 1, 1, 1]
['red', 'green', 'red', 'green', 'red', 'green']
['red', 'green']
['red', 'green', 'red', 'green', 'red', 'green']
```

上記のサンプルプログラムのように、`*` 演算子を用いると、左の項のリストを右の項の回数分繰り返し結合することができる。左の項はリストで、右の項は整数でなければならない。また、`+=` と同様に累算代入演算子 `*=` を使うこともできる。

繰り返し (1)

続いて、`for` 文について解説する。`for` 文は文字列やリストなどの反復可能なオブジェクト（イテラブルオブジェクト）に対して、その要素の数だけ繰り返し命令を実行することができる。 なお、反復可能内部オブジェクトの詳細については第6章で説明する。

```
colors = ["red", "green", "blue"]
for color in colors:
    print(color)
```

```
red
green
blue
```

`for` 文の構文は次のとおりである。

```
for <変数名> in <リストや文字列などのオブジェクト>:
    <繰り返し実行したい命令>
```

`in` の右側の値の集合から、1つずつ値を取り出して変数に代入して、記載された命令を繰り返し実行する。 命令は1行でも複数行でも構わないが、必ず文頭にインデントを追加しなければならない。 インデントには空白文字やタブ文字を何文字でも使うことができる。どちらを使っても構わないが、 一般的には空白2文字、空白4文字、タブ文字1文字のいずれかを使うケースが多い。 これらを混在させて使うこともできるが、コードが読みづらくなので、一貫性を持たせることが好ましい。 今回は、空白4文字で統一している。

例えば、上記のサンプルプログラムの `for` 文の例では、次のように処理が行われる。

1. `colors` から `["red", "green", "blue"]` を読み込む。
2. `color` に読み込んだ値の1つ目の要素 `"red"` を代入する。
3. `print(color)` を実行して `red` を表示する。
4. `color` に読み込んだ値の2つ目の要素 `"green"` を代入する。
5. `print(color)` を実行して `green` を表示する。
6. `color` に読み込んだ値の3つ目の要素 `"blue"` を代入する。
7. `print(color)` を実行して `blue` を表示する。

`for` 文で定義した変数は、`for` 文の中にある命令内でしか参照することができない。次のサンプルプログラムを見てみよう。

```
for color in "rgb":
    print(color)
    print(color * 2)

# 以降ではcolorを参照できない
print('for文の外側')
```

```
r
rr
g
gg
b
bb
for文の外側
```

`for` 文の次に続くステートメントは繰り返し実行する命令を表している。繰り返し実行する命令の範囲はインデントを使って表現できる。インデントレベルを、 `for` 文の開始位置以上まで上げることで、 `for` 文の繰り返し実行する命令の外側であることを表現できる。

上記のサンプルプログラムでは、上2つの `print` は `for` 文の中に記載されているため、3回繰り返し実行される。 一方、3つ目の `print` は `for` 文の外側に記載されているため、1回しか実行されない。

`for` 文では `range` 関数を組み合わせて使うことが多い。関数という概念の詳細は第4章で説明するが、ここでは `range` 関数のみ説明する。

```
for number in [0, 1, 2, 3, 4]:
    print(number)

for number in range(5):
    print(number)
```

```
0
1
2
3
4
0
1
2
3
4
```

0から4の数について `for` 文を実行したいときは、1つ目の `for` 文の例のように記述することもできるが、2つ目の `for` 文の例のようにも記述することができる。

`range` 関数を使うと、 `0` から、括弧内に記述した数から `1` を引いた数までの反復可能なオブジェクトを得ることができる。 その結果、 `for` 文と `range(5)` を組み合わせれば0から4の数まで繰り返し、 `for` 文と `range(10)` を組み合わせれば0から9の数まで繰り返すことができる。

`for` 文では必ずリスト等から各要素の値を受け取り、変数に代入する必要があるが、各要素の値を参照する必要がないケースもある。 例えば、次のような特定の文字列を5回表示するケースが該当する。

```
for number in range(5):
    print('------')
    print('|    |')

print()

for _ in range(5):
    print('------')
    print('|    |')
```

```
------
|    |
------
|    |
------
|    |
------
|    |
------
|    |

------
|    |
------
|    |
------
|    |
------
|    |
------
|    |
```

1つ目の `for` 文では `number` という変数を定義しているが、 `for` 文中で参照されていない。このような特定の動作を繰り返すケースなどでは、変数を参照しないことがある。

そのような場合、2つ目の `for` 文のように、変数名に `_` (アンダースコア)を使うことが一般的である。必ずこの変数にしなければならない、というわけではないが、この記号を使うことにより、「 `for` 文の中で各要素を参照する変数を使っていない」という意思表示になる。

条件分岐

続いて `if` 文について解説する。 `if` 文は、条件に応じてプログラムの処理を変えることができる。

```
a = 10
b = 23

if a == b:
    print("aとbは同じ値である")
else:
    print("aとbは異なる値である")
```

```
aとbは異なる値である
```

`if` 文の構文は次の通りである。

```
if <条件1>:
    <条件1に当てはまる場合の処理>
elif <条件2>:
    <条件1には当てはまらないが、条件2に当てはまる場合の処理>
else:
    <全ての条件に当てはまらない場合の処理>
```

`if` 文では上から条件を確認していき、条件に合っていたら、その下の記載された処理を行う。合っていなければ次の `elif` 節に移り、条件を判定する。そして同様に、条件に合ったら処理を行う。全ての条件に合わなかった場合、最後の `else` 節の処理を行う。 この `elif` 節は複数個書くことができる。また、 `elif` 節と `else` 節は省略することができる。 一度条件に合致したら、その節の処理のみを行い、それより下にある条件の判定は行わず処理も実行しない。

処理を書く際には、 `for` 文の命令と同様に、文頭のインデントのレベルを下げなければならない。

上記のサンプルプログラムの `if` 文の例について見てみよう。

条件のところに `a == b` と書かれている。これは、「変数 `a` と変数 `b` の値が等しいか」という条件になっている。今回は等しくないため、`if` 節の処理は実行されず、下の `else` 節の処理が実行され、「 `aとbは異なる値である` 」が実行される。

次のサンプルプログラムの `if` 文についての実行結果について考えてみよう。

```
a = 34
b = 23

if a < b:
    print("aはbより小さい値である")
elif a > b:
    print("aはbより大きな値である")
else:
    print("aとbは同じ値である")
```

```
aはbより大きな値である
```

このサンプルプログラムでは、 `if` 節の条件のところに `a < b` と書かれている。これは「変数 `a` の値が変数 `b` の値より小さいか」という条件になっている。今回は `a` の方が大きいので条件に合わず、直下の `elif` 節に移る。

`elif` 節の条件のところに `a > b` と書かれている。これは「変数 `a` の値が変数 `b` の値より大きいか」という条件になっている。この条件には合っているので、 `elif` 節の処理を行い、「 `aはbより大きな値である` 」が表示される。

`if` 文は、1行で書くこともできる。

```
a = 10
b = 10
if a == b: print("aとbは同じ値である")
```

```
aとbは同じ値である
```

`if` 節のみ、かつ、処理が1行のみの場合、上記のように1行で書くことができる。構文は次の通りである。

```
if <条件>: <条件に当てはまる場合の処理>
```

`else` 節がある場合でも1行で書くことができる。

```
a = 5
b = 10
print("aとbは異なる値である") if a != b else print("aとbは同じ値である")
```

```
aとbは異なる値である
```

`else` 節がある場合でも、 `if` 節、 `else` 節どちらの処理も1行のみならば、上記のように1行で書く記法がある。構文は次の通りである。

```
<条件に当てはまる場合の処理> if <条件> else <条件に当てはまらない場合の処理>
```

上記の例では、条件に `a != b` と書かれている。これは、「変数 `a` の値と変数 `b` の値が**等しくない**か」という条件になっている。つまり、`a` と `b` の値が異なっていれば、条件に当てはまる。今回は `a` と `b` の値が異なっているので、「`aとbは異なる値である`」が出力される。

条件には、以下のような種類がある。

条件	意味
a == b	aはbと等しい
a != b	aはbと等しくない
a > b	aはbより大きい
a >= b	aはb以上
a < b	aはbより小さい
a <= b	aはb以下

`if` 文では、複数の条件を指定することもできる。

```
a = 10
b = 5
c = 20

if a > b and a > c:
    print("aは、bとcより大きな値である")
elif a > b or a > c:
    print("aは、bかcより大きな値である")
```

```
aは、bかcより大きな値である
```

`if` 文では、 `and` や `or` を使って、複数の条件を指定することができる。 `and` では、両方の条件を満たしているか、 `or` では、どちらか片方の条件を満たしているかが判定される。

例えば、 `a > b and a > c` という条件は、「変数 `a` が変数 `b` より大きい **かつ** 変数 `a` が変数 `c` より大きいか」を表している。 `a > b or a > c` という条件は、「変数 `a` が変数 `b` より大きい **または** 変数 `a` が変数 `c` より大きいか」を表している。

上記のサンプルプログラムでは、 `a` は `b` より大きいが、 `c` より小さい。そのため、 `a > b` は満たすが `a > c` は満たさない。そのため、「`a > b and a > c`」の条件には合わないが、「`a > b or a > c`」の条件には合う。結果、「`aは、bかcより大きな値である`」が出力される。

`if` 文は、数字だけでなく、文字列を判定することもできる。

```
animals = ["dog", "cat", "rabbit"]
for animal in animals:
    if animal == "cat":
        print("猫を見つけた！")
    else:
        print(animal + "は、猫ではない")
```

```
dogは、猫ではない
猫を見つけた！
rabbitは、猫ではない
```

上記の例では、 `if` 文の条件に `animal == "cat"` が指定されている。これは、「変数 `animal` の値が `"cat"` かどうか」という判定をしている。 `animal` の中身が `"dog"` や `"rabbit"` の時は、条件に合わないためelse節の処理が実行されているが、 `animal` の中身が `"cat"` の時は、条件に合っているため「`"猫を見つけた！"`」が出力されている。

最後に、`elif`節が複数個ある場合を見てみよう。`if`文では、`elif`節を複数個追加して、条件分岐を増やすことができる。

```
for i in range(1, 20):
    if i % 15 == 0:
        print("3の倍数であり、5の倍数でもある")
    elif i % 5 == 0:
        print("5の倍数")
    elif i % 3 == 0:
        print("3の倍数")
    else:
        print(i)
```

```
1
2
3の倍数
4
5の倍数
3の倍数
7
8
3の倍数
5の倍数
11
3の倍数
13
14
3の倍数であり、5の倍数でもある
16
17
3の倍数
19
```

これは、1〜19までの数字が、3の倍数か5の倍数か、またはその両方かを判定するプログラムである。

上記のサンプルプログラムの`if`文の最初の条件には、`i % 15 == 0`と書かれている。`i % 15`は、「iを15で割ったときの余り」を表しているため、`i % 15 == 0`は「変数`i`を15で割ったときの余りが0か」を判定している。同様に、次の`elif`節の`i % 5 == 0`は「変数`i`を5で割ったときの余りが0か」、その次の`i % 3 == 0`は「変数`i`を3で割ったときの余りが0か」を判定している。

繰り返し (2)

続いて、`while`文について解説する。`while`文は、`for`文と同様に繰り返し命令を実行する場合に用いられる。`for`文とは構文が異なるため、実際にみてみよう。

```
i = 5
while i > 0:
    print(i)
    i -= 1
```

```
5
4
3
2
1
```

`while`文の構文は以下の通りである。

```
while <条件>:
  <繰り返し実行したい命令>
```

`for` 文とは異なり、 `while` の右に書くのは `<条件>` のみである。 `while` 文では、この `<条件>` を満たしている間は `<繰り返し実行したい命令>` を実行し続ける。 `<条件>` を満たさなくなった段階でループを抜ける。

`while` 文も `for` 文と同様、命令は文頭のインデントのレベルを下げなければならない。

上記のサンプルプログラムの `while` 文の例では、命令が1回実行されるごとに変数 `i` の値を1ずつ減らしていく。そして、 `i > 0` の条件を満たさなくなるまで `while` 文内の命令を実行し続けている。

繰り返しの制御

続いて、繰り返しの制御について解説する。

`while` 文は、プログラムの書き方によっては無限ループを引き起こし、プログラムの実行が終わらなくなってしまうことがある。以下の例を見てみよう。

```
print("コメントアウトを外すと無限ループが実行される")
# while True:
#     print("無限ループ")
```

```
コメントアウトを外すと無限ループが実行される
```

上記のサンプルプログラムの `while` 文の例では、「条件」の部分が `True` になっている。つまり、この `while` 文は常に条件を満たすため、 `while` 文内の命令を延々と実行し続け、プログラムの実行が終わらない状態になってしまう。

もし無限ループが起こってしまったら、プログラムを強制的に終了させる必要がある。

`while` 文は、条件を満たさなくなる他にも、 `break` 文を使ってループを抜け出すこともできる。

```
i = 0
while i < 10:
    if i == 3:
        break
    print(i)
    i += 1
```

```
0
1
2
```

`break` 文を実行すると、 `while` 文の条件にかかわらず `while` 文を抜け出すことができる。上記のサンプルプログラムでは、 `if` 文で「変数 `i` が3と等しいか」を判定し、その中で `break` 文が使われている。そのため、 `i` が3になった段階で `while` 文を抜け出しており、0〜2までしか出力されない。

この `break` 文は、 `for` 文に使うこともできる。

```
animals = ["dog", "fish", "cat"]

for animal in animals:
    if animal == "fish":
        print("魚を見つけた！")
        break
    print("魚ではない")
```

```
魚ではない
魚を見つけた！
```

上記の例では、変数 `animal` が「 `"fish"` 」であった場合、その時点で `for` 文を抜け出している。そのため、 `animal` の値が `"cat"` の場合の実行はされない。

`berak` 文は、 `for` 文で `range` 関数を扱った場合にも使用できる。

```
for i in range(10):
    if i > 5:
        break
    print(i)
```

```
0
1
2
3
4
5
```

`for` 文では `range(10)` が用いられており、本来は10回ループが回るはずである。しかし、 `for` 文内で `break` 文が用いられており、 `i > 5` の場合に実行されるようになっている。そのため、 `i` が6になった時点でループを抜け出している。ゆえに、ループは6回目で中断していることになる。このように、 `range` 関数でループ回数を指定していても、 `break` 文によってループを抜け出すと、指定より少ない回数しかループは実行されない。

特定の条件の場合に処理はスキップしたいが、それ以降のループも実行したい、という場合もあるだろう。そのような場合には、 `continue` 文を使うことができる。

```
i = 0
while i < 10:
    i += 1
    if i == 5:
        continue
    print(i)
```

```
1
2
3
4
6
7
8
9
10
```

`continue` 文が実行されると、 `break` 文と同様に、それ以降の `while` 文内の命令がスキップされる。一方、 `break` 文とは異なり、 `while` 文から抜けるわけではなく、再度 `while` 文の命令が実行される。

上記の例では、変数 `i` が5の時に `continue` 文が実行され、後ろの `print(i)` の処理がスキップされる。そのため、5を除く1〜10の数字が出力されている。

`while` 文では、 `if` 文のように `else` 節が使え、 `while` 文の条件を満たさなくなったときに実行できる。

```
i = 5
while i > 0:
    print(i)
    i -= 1
else:
    print("iは0になった")
```

```
5
4
3
2
1
iは0になった
```

上記の例では、 `i > 0` なら `while` 文の命令を実行し続ける。一方、 `i > 0` を満たさなくなったら、 `while` 文の処理を抜け、 `else` 節の処理が実行されている。

この `else` 節は、 `break` 文や `continue` 文で処理をスキップしたときには実行されない。

パターンマッチ

続いて、 `match` 文について解説する。 `match` 文は、式のパターンによって処理を変えることができる。

```
name = "太郎"
match name:
    case "花子":
        print("花子さん、こんにちは！")
    case "太郎":
        print("太郎くん、こんにちは！")
```

```
太郎くん、こんにちは！
```

`match` 文の構文は次の通りである。

```
match <式>:
  case <パターン1>:
    <パターン1に当てはまる場合の処理>
  case <パターン2>:
    <パターン2に当てはまる場合の処理>
  case _:
    <全てのパターンに当てはまらない場合の処理>
```

`<式>` には基本的に変数を記述し、 `<パターン>` には基本的に数字や文字列などの具体的な値を記述する。

まずは `match` の右に `<式>` を指定する。そして `<パターン1>` の値と `<式>` に記述した変数内の値を比較し、等しければ `<パターン1に当てはまる場合の処理>` を実行する。等しくなければ、次は `<パターン2>` の値と `<式>` に記述した変数内の値を比較する。 `<パターン2>` と `<式`

`>` を比較して等しければ `<パターン2に当てはまる場合の処理>` を実行する。全てのパターンに当てはまらなかった場合、 `case _:` と書かれた箇所の処理を実行する。 `case _:` は省略することができる。

このように、 `<式>` に書かれている変数の中の値と `<パターン>` に書かれている値を比較し、それらが合致したときに、その部分の処理を実行するのが `match` 文によるパターンマッチである。

`case <パターン>:` は、 `match <式>:` よりも文頭のインデントのレベルを下げなければならない。また、 `<パターンに当てはまる場合の処理>` も、 `case <パターン>:` よりさらにインデントのレベルを下げなければならない。

上記のサンプルプログラムの `match` 文の例では、 `<式>` として `name` が指定されている。最初のパターンは `"花子"` だが、 `name` は `"花子"` ではないため、次のパターンに移る。次のパターンは `"太郎"` だが、 `name` は `"太郎"` であるため、このパターンに当てはまる場合の処理 `print("太郎くん、こんにちは！")` が実行される。

`match` 文は基本的に `if` 文に書き換えることができる。上記のサンプルプログラムを `if` 文に書き換えると以下のようになる。

```
name = "太郎"
if name == "花子":
  print("花子さん、こんにちは！")
elif name == "太郎":
  print("太郎くん、こんにちは！")
```

```
太郎くん、こんにちは！
```

次に、全てのパターンに当てはまらない場合を見てみよう。

```
animal = "rabbit"
match animal:
    case "cat":
        print("ニャー")
    case "dog":
        print("ワンワン")
    case _:
        print("動物を見つけられなかった")
```

```
動物を見つけられなかった
```

変数 `animal` には `"rabbit"` が格納されている。しかし、 `match` 文のパターンには `"cat"` と `"dog"` しかなく、 `"rabbit"` はない。そのような全てのパターンに当てはまらない場合の処理を行いたいときがある。そのような場合は、 `case _:` のように `<パターン>` の箇所に `_` (アンダースコア)を用いたパターンを用いる。今回の例でもこの `case _:` が実行され、 `"動物を見つけられなかった"` が実行される。

上記のサンプルプログラムを `if` 文で表すと以下のように書き換えることができる。

```
animal = "rabbit"
if animal == "cat":
  print("ニャー")
elif animal == "dog":
  print("ワンワン")
else:
  print("動物を見つけられなかった")
```

```
動物を見つけられなかった
```

「`<式>`と`<パターン>`が等しい」時の判定は以上のようにできる。では、「`<式>`が`<パターン>`より大きい(小さい)」などの判定を行いたいときはどうすればよいだろうか。その例を以下に示す。

```
i = 22

match i:
    case  i if i < 0:
        print("iは、0より小さい")
    case i if i > 0:
        print("iは、0より大きい")
    case _:
        print("iの値は0である")
```

```
iは、0より大きい
```

1つ目のパターンには`i if i < 0`と書かれている。このように書くことによって、「`i < 0`なら」すなわち「`i`が0より小さいなら」というパターンを表すことができる。同様に、2つ目のパターン`i if i > 0`は「`i > 0`なら」すなわち「`i`が0より大きいなら」というパターンを表している。

上記のサンプルプログラムを`if`文で表すと以下のようになる。

```
i = 22

if i < 0:
  print("iは、0より小さい")
elif i > 0:
  print("iは、0より大きい")
else:
  print("iの値は0である")
```

```
iは、0より大きい
```

問題1

問題文

与えられたリストのうち6番目の要素を出力せよ。

解答の雛形

```
li = [1, 4, 1, 4, 2, 1, 3, 5, 6]
# ここに解答を入力
```

問題2

問題文

`li1`, `li2`の２つのリストがある。次の３つの操作を順番に実行せよ。

1. この２つのリストを結合する
2. 結合後のリストを100回繰り返す
3. 繰り返したリストの794番目の要素を出力する

解答の雛形

```
li1 = [3, 1, 4, 1, 5, 9, 2, 6, 5]
li2 = [2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
# ここに解答を入力
```

問題3

問題文

与えられたリストの各要素をfor文で足し合わせた合計値を表示せよ。

解答の雛形

```
li = [1, 4, 1, 4, 2, 1, 3, 5, 6]
total = 0
# ここに解答を入力
print(total)
```

問題4

問題文

与えられたリストのうち3番目から13番目までの要素の中で、 奇数の要素を全て足し合わせた合計値を表示せよ。 この問題でもfor文を用いること。

解答の雛形

```
li = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3]
total = 0
# ここに解答を入力
print(total)
```

問題5

問題文

与えられたリストの要素を順番に見て、要素毎に改行区切りで次の文字列を出力すること。この問題ではif文やelif文を用いること。

- 5で割った余りが2のとき - `red`
- 5で割った余りが3のとき - `green`

- いずれにも該当しないとき - `blue`

解答の雛形

```
li = [2, 7, 1, 8, 2, 8, 1, 8, 2, 8, 4]
for element in li:
  # ここに解答を入力
```

問題6

問題文

3のX乗(Xは整数)のうち、初めてYを超える数を出力せよ。

制約

- $1 \leq Y \leq 10000$
- 入力は全て整数である。

入力

入力は次の形式で与えられる。

```
Y
```

出力

問題文に即した値を出力せよ。

入力例1

```
7
```

出力例1

```
9
```

解答の雛形

```
Y = int(input())
value = 3
while value < 2**63:
  # ここに解答を入力
  value *= 3
print(value)
```

問題7

問題文

2のX乗(Xは整数)のうち、Y以上かつZ未満の数字を出力せよ。

制約

- $2 \leq Y \leq Z \leq 10000$
- 入力は全て整数である。

入力

入力は次の形式で与えられる。

```
Y Z
```

出力

問題文に即した値を出力せよ。

入力例1

```
5 20
```

出力例1

```
8
16
```

解答の雛形

```
Y, Z = map(int, input().split())
value = 1
while value < 2**63:
  value *= 2
  # ここに解答を入力
  print(value)
```

問題8

問題文

インターネットでは、ページにアクセスしたときに、正常にアクセスできたかどうかなどを表す数字が返ってくる。 これはHTTPステータスコードと呼ばれたりする。 そして例えば以下のように、各数字には名前がついている。

- 200 - `OK`
- 202 - `Accepted`

- 400 - `Bad Request`
- 403 - `Forbidden`
- 500 - `Internal Server Error`

入力として数字(`status`)が与えられるので、対応する名前を出力せよ。もし上記５つのいずれにも該当しない場合には、 `Invalid Input` と出力せよ。

制約

- $200 \leq status \leq 500$
- 入力は全て整数である。

入力

入力は次の形式で与えられる。

```
status
```

出力

対応する名前を出力せよ。

入力例1

```
200
```

出力例1

```
OK
```

解答の雛形

```
status = int(input())
match status:
  # ここに解答を入力
```

4章: 関数とメソッド

関数

関数定義と呼び出し

本節では、Pythonにおける関数という仕組みについて紹介する。関数とは、何らかの関連性のある処理や計算などを一つにまとめ、外部から呼び出せるようにしたものである。 関数を使うには、関数の「定義」（関数を作成する）を行い、定義した関数を「呼び出す」（関数にまとまっている処理を実行する）必要である。

`[XXX]` を電車1両分に見立てて、電車を表示するプログラムを考えてみよう。

```
print('[XXX]-[XXX]')
print('[XXX]-[XXX]-[XXX]-[XXX]')
```

```
[XXX]-[XXX]
[XXX]-[XXX]-[XXX]-[XXX]
```

2両編成の電車と4両編成を表示できた。

ここで、電車の見た目を `[###]` に変更してほしいとお願いされた。電車の見た目を修正するには、以下のようにコードを書き直すだろう。

```
print('[###]-[###]')
print('[###]-[###]-[###]-[###]')
```

```
[###]-[###]
[###]-[###]-[###]-[###]
```

無事に修正が完了し、電車の見た目が `[###]` になった。 しかし、元のコードから `X` を18文字も `#` に書き換える必要があった。 もしこれが100両編成の電車を表示するプログラムだった場合、もっと多くの文字を修正しなければならず、修正漏れが発生するかもしれないし、時間もかかってしまう。

こんなときには関数を使おう。関数を使えば、定義した命令を繰り返し実行することができる。関数の中身を変更すれば、関数を使っている全て箇所の動きを一括して変えられるようになる。まず、最初の電車を表示してみよう。

```
def train():
    return "[XXX]"

train2 = train() + "-" + train()
print(train2)

train4 = train() + "-" + train() + "-" + train() + "-" + train()
print(train4)
```

```
[XXX]-[XXX]
[XXX]-[XXX]-[XXX]-[XXX]
```

冒頭2行で関数の定義を行っている。 文頭に `def` と書き、その後ろに関数名と丸括弧 `()` 続けることで関数の定義を行うことができる。 サンプルプログラムでは、 `train` という名前の関数を作成している。

関数の実体を構成する処理は次の行から始め、インデントのレベルを下げて書き始める。 `train` 関数では、電車の見た目である `[XXX]` という文字列を戻り値として定義している。

戻り値とは、関数内の処理を行った結果として呼び出し元に渡される値のことである。 `return` の後ろに配置された式が、その関数の戻り値となる。例えば、 `return 1` 、 `return 1+2` , `return train2` 、 `return train()` など、値や数式、変数、関数などを配置できる。 `return` は一つの関数内で1回だけ使用することができ、 `return` が記述された行より下に書かれた関数内の処理は実行されない。

関数名と丸括弧 `()` を表記することで、関数を呼び出すことができる。 `train()` と記述された箇所で関数の呼び出しを行い、 `print(train2)` や `print(train4)` で変数に入った値を出力している。

上記のサンプルプログラムでの電車の見た目は、 `[XXX]` であるが、電車の見た目を `[###]` に変更したいとき、どのようにすべきだろうか。

```
def train():
    return "[###]"

train2 = train() + "-" + train()
print(train2)

train4 = train() + "-" + train() + "-" + train() + "-" + train()
print(train4)
```

```
[###]-[###]
[###]-[###]-[###]-[###]
```

`train` 関数の戻り値を、わずか3文字直すだけで、電車の見た目を変えることができた。 ソフトウェアの変化は激しいので、仕様変更や修正が入ったときに、「漏れがなく」「誰でも」「簡単に」変更できるようにプログラムを作ることが重要である。

今度は、見た目の異なる車両をつなげた電車をかけるようにしてみよう。

```
def train(mark):
    result = ""
    result += "["
    result += mark * 3
    result += "]"
    return result

train2 = train("X") + "-" + train("X")
print(train2)

train4 = train("#") + "-" + train("X") + "-" + train("X") + "-" + train("#")
print(train4)
```

```
[XXX]-[XXX]
[###]-[XXX]-[XXX]-[###]
```

上記サンプルプログラムは今までと違い、関数名の後ろの丸括弧 `()` 内に記述がある。関数名の後ろの丸括弧 `()` 内に書いてあるものを `仮引数（パラメータ; parameter）` という。仮引数とは、呼び出し元から渡された値を受け取るために宣言された変数のことで、この変数を関数内で使用して処理を記述することができる。 サンプルプログラムでは、受け取った引数を `mark` という名前で利用し、 `mark * 3` とすることで電車の車両の見た目を作成している。

関数を呼び出している箇所にも注目してみよう。先程までは `train()` と呼び出すことができたが、今回は関数側が仮引数を定義し、関数内の処理で使用しているので、呼び出す側でも対応する必要がある。 `train("X")` とすることで、 `train` 関数に `X` という文字を渡している。このときの丸括弧 `()` 内の `"X"` を `実引数（argument）` という。

このように、関数定義に仮引数を使うことで、関数の動きを変化させることができ、より複雑なプログラムを作ることができるようになる。

引数は複数あっても良い。複数の引数を使った関数のサンプルプログラムを見てみよう。

```
def train(mark, is_tail):
    result = ""
    result += "["
    result += mark * 3
    result += "]"
    if not is_tail:
        result += "-"
    return result

train2 = train("X", False) + train("X", True)
print(train2)

train4 = train("#", False) + train("X", False) + train("X", False) + train("#", True)
print(train4)
```

```
[XXX]-[XXX]
[###]-[XXX]-[XXX]-[###]
```

プログラム中で共通する処理を関数に入れれば入れるほど、関数を呼び出す部分がシンプルになって、プログラムを理解しやすくなる。

先程までは、車両のつなぎ目の `-` は、電車を作るとき（例えば `train2` を作るときなど）に必要な数だけ書き出していた。この部分も関数の処理に入れたのが、上記のサンプルプログラムである。 `is_tail` という仮引数を受け取って、車両のつなぎ目である `-` を戻り値に追加するかを判定している。

実引数は、関数を定義する際に記述した仮引数と同じ順番で指定しないといけないので注意しよう。

電車の電車のつなぎ目である「 `-` 」は、電車の最後列以外では出力するので、 `train` 関数を呼び出すときの `is_tail` の実引数には、 `False` が多く指定することになる。何度も同じ引数を記述するのは面倒なので、そういうときはデフォルト引数を使うと便利である。

```
def train(mark, is_tail=False):
    result = ""
    result += "["
    result += mark * 3
    result += "]"
    if not is_tail:
        result += "-"
    return result

train2 = train("X") + train("X", True)
print(train2)

train4 = train("#") + train("X") + train("X") + train("#", True)
print(train4)
```

```
[XXX]-[XXX]
[###]-[XXX]-[XXX]-[###]
```

関数の呼び出し部分に注目してみよう。最後尾車両以外には、 `is_tall` の実引数がセットされていない。それにも関わらず、出力は期待通りになっている。

関数の定義をしている部分にも注目してみよう。先程までのサンプルプログラムとは違い、 `is_tail` 仮引数が、 `is_tail=False` となっている。 このように、仮引数に値を代入しておくことで、呼び出し元からその実引数が渡ってこなかった場合、仮引数に代入した値を使って関数の処理が実行される。

これにより、関数の呼び出し部分がさらにシンプルになった。

デフォルト引数は複数使えるので、車両の見た目である仮引数 `mark` にも、デフォルト値を設定してみよう。

```
def train(mark="#", is_tail=False):
    result = ""
    result += "["
    result += mark * 3
    result += "]"
    if not is_tail:
        result += "-"
    return result

train2 = train("X") + train("X", True)
print(train2)

train4 = train() + train("X") + train("X") + train("#", True)
print(train4)
```

```
[XXX]-[XXX]
[###]-[XXX]-[XXX]-[###]
```

仮引数でも実引数でもデフォルト引数は末尾にないといけない。

例えば、関数を定義するときに、 `def train(mark='#', is_tail):` は、デフォルト引数が戦闘になってしまっているのでエラーになってしまう。 `def train(mark, is_tail=False)` と定義した関数を呼び出す際に、 `train(False)` としてしまうと、 `mark` に `False` が代入されてしまう。

train関数を、一度呼び出すだけで複数の車両を表示できるように修正してみよう。

```
def train(mark="#", repeat=1, is_tail=False):
    result = ""
    for i in range(repeat):
        result += "["
        result += mark * 3
        result += "]"
        if i < repeat - 1 or not is_tail:
            result += "-"
    return result

train2 = train("X", 2, True)
print(train2)

train4 = train() + train("X", 2) + train("#", 1, True)
print(train4)

train10 = train("X", 5) + train("#", 5, True)
print(train10)
```

```
[XXX]-[XXX]
[###]-[XXX]-[XXX]-[###]
[XXX]-[XXX]-[XXX]-[XXX]-[XXX]-[###]-[###]-[###]-[###]-[###]
```

`repeat` という仮引数を関数定義に追加し、いくつ車両を作るかを実引数で指定できるようにしたことで、長い電車を簡単につくることができるようになった。

このように、仮引数の数が増えてくると、何番目の引数がどの意味を持つのかわかりにくくなってしまう。そんなときは、キーワード引数を使おう。

```
def train(mark="#", repeat=1, is_tail=False):
    result = ""
    for i in range(repeat):
        result += "["
        result += mark * 3
        result += "]"
        if i < repeat - 1 or not is_tail:
            result += "-"
    return result

train2 = train("X", 2, True)
print(train2)

train4 = train() + train("X", repeat=2) + train("#", is_tail=True, repeat=1)
print(train4)

train10 = train("X", 5) + train(repeat=5, mark="#", is_tail=True)
print(train10)
```

```
[XXX]-[XXX]
[###]-[XXX]-[XXX]-[###]
[XXX]-[XXX]-[XXX]-[XXX]-[XXX]-[###]-[###]-[###]-[###]-[###]
```

先程、「実引数は、関数を定義する際に記述した仮引数と同じ順番で指定しないといけないので注意しよう。」と説明したが、キーワード引数を使えば、関数定義の仮引数の順番を無視して、自由に実引数を並び替えることができる。しかし、`train(repeat=5, mark="#", True)`など、キーワード引数の後ろに、キーワードのない実引数を書くことはできないので注意しよう。

ここまで、`train`関数を呼び出して、その結果(戻り値)を`print()`関数で出力していたが、`train`関数で出力まで行うように書き換えてみよう。

```
def train(mark='#', repeat=1, is_tail=False):
    for i in range(repeat):
        print('[', end='')
        for _ in range(3):
            print(mark, end='')
        print(']', end='')
        if i < repeat - 1 or not is_tail:
            print('-', end='')
        elif is_tail:
            print()
    return

train("X", 2, True)

train()
train("X", repeat=2)
train("#", is_tail=True, repeat=1)

train("X", 5)
train(repeat=5, mark="#", is_tail=True)
```

```
[XXX]-[XXX]
[###]-[XXX]-[XXX]-[###]
[XXX]-[XXX]-[XXX]-[XXX]-[XXX]-[###]-[###]-[###]-[###]-[###]
```

`print()`関数はPythonの組み込み関数だが、この`print()`関数にも仮引数が設定されている。その中に、`end`というキーワード引数が定義されている。この仮引数`end`は名前の通り、`print()`関数の出力の最後尾の文字列を指定することができる。この仮引数`end`には、デフォルトで改行コード`\n`が設定されている。なので、`print('-', end='')`は、「`-`という文字列を出力するときに、最後尾で改行せずに出力する」というプログラムになる。

また、`print()`関数は、出力する文字列が指定されなかった場合、仮引数`end`のみを出力するようにプログラムされている。そのため、`print()`は「出力を改行する」というプログラムになる。

`return`の直後に変数や値を記述した場合、`return`の直後に記述した値や変数の中の値を関数の戻り値として呼び出し元に渡されることは先程説明したが、上記のサンプルプログラムを見てみると、`return`という単語しか記述されていない。この`return`の後ろには、戻り値`None`が省略されている。

`None`とは、値がないことを表すために頻繁に使用されるオブジェクトのことである。

サンプルプログラムの`return`は、値を渡すためではなく、関数の呼び出しを終了するための使用されている。

そして、`return`は、関数の途中でも使用することができ、また（戻り値を定義しない場合）省略することもできる。サンプルプログラムの`return`も省略可能であるので、実際に`return`を削除して実行してみよう。

再帰関数

突然だが、フィボナッチ数列を知っているだろうか。フィボナッチ数列とは、イタリアの数学者フィボナッチが紹介した、「２つ前にある数字と1つ前にある数字を数字を足した数字」が並んでいる数列のことである。以下がフィボナッチ数列である。

`1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377 ...`

数列の要素に注目してみよう。 数列の4番目の数字は、3番目の`2`と2番目の`1`の和である`3`になっている。 数列の5番目の数字は、4番目の`3`と3番目の`2`の和である`5`になっている。 数列の6番目の数字は、5番目の`5`と4番目の`3`の和である`8`になっている...

といった具合に無限に数列が続いていくのである。

フィボナッチ数列のn番目の数字を出力するプロブラムを見てみよう。

```
def fibonatti(n):
    a = 0
    b = 1
    for _ in range(n):
        c = a + b
        a = b
        b = c
    return a

print(fibonatti(10))
```

```
55
```

上記サンプルプログラムでは、フィボナッチ数列の10番目の数字を求めている。出力は`55`になっているはずだ。フィボナッチ数列をもう一度見てみよう。先頭から10番目は`55`なので、サンプルプログラムの出力は正しいことがわかる。

`1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377 ...`

サンプルプログラムの`fibonatti`関数はどのように処理を実行しているのでしょうか。`fibonatti(5)`を呼び出した際の処理の流れを解説する。

1. a と b にそれぞれ0と1を代入
2. for文内で、n=1のとき
    1. cに1を代入(a=0, b=1なので、c=0+1)
    2. aに1を代入(b=1なので)
    3. bに1を代入(c=1なので)
3. for文内で、n=2のとき
    1. cに2を代入(a=1, b=1なので、c=1+1)
    2. aに1を代入(b=1なので)

    3. bに2を代入(c=2なので)
4. for文内で、n=3のとき
    1. cに3を代入(a=1, b=2なので、c=1+2)
    2. aに2を代入(b=2なので)
    3. bに3を代入(c=3なので)
5. for文内で、n=4のとき
    1. cに5を代入(a=2, b=3なので、c=2+3)
    2. aに2を代入(b=3なので)
    3. bに5を代入(c=5なので)
6. for文内で、n=5のとき
    1. cに7を代入(a=2, b=5なので、c=2+5)
    2. aに5を代入(b=5なので)
    3. bに7を代入(c=7なので)
7. `fibonatti` 関数の呼び出し元にa(a=5)を返す

for文内に入ったときの各変数の状態は、変数 `a` は `n-1` 番目の数値、変数 `b` は `n` 番目の数値であり、変数 `c` に変数 `a` と変数 `b` から算出した `n+1` 番目の数値を計算している。その後、次のfor文の処理に備えて、変数 `a` に `n` 番目の数値である変数 `b` を代入し、変数 `b` に `n+1` 番目の数値である変数 `c` を代入している。なので、for文内の処理が終わるときには、フィボナッチ数列の `n` 番目の値は変数 `a` に代入されている。これをn回文繰り返し、最後に変数 `a` を `fibonatti` 関数の結果(戻り値)として呼び出し元に渡されている。

先程のサンプルプログラムだと、変数の値がfor文の繰り返しごとに更新されていくので、変数がどのように変化しているのかを読み取るのが大変である。そういうときは再帰関数を使おう。

再帰関数とは、「自分自身を呼び出す関数」のことである。再帰関数を使うとフィボナッチ数列を簡単に実装できる。フィボナッチ数列のn番目の値を返す関数を、再帰関数でプログラミングしてみよう。

```
def fibonatti(n):
    if n == 0: return 0
    if n == 1: return 1
    return fibonatti(n - 1) + fibonatti(n - 2)

print(fibonatti(10))
```

```
55
```

上記サンプルプログラムでも、フィボナッチ数列の10番目の値を求めている。出力は `55` になる。

先程と同様に、 `fibonatti(5)` を呼び出した際の処理の流れを解説する。

1. `n=5` なので、 `fibonatti(5 - 1)` となり、 `fibonatti(4)` を呼び出す
    1. `n=4` なので、 `fibonatti(4 - 1)` となり、 `fibonatti(3)` を呼び出す
        1. `n=3` なので、 `fibonatti(3 - 1)` となり、 `fibonatti(2)` を呼び出す
            1. `n=2` なので、 `fibonatti(2 - 1)` となり、 `fibonatti(1)` を呼び出す
                1. `if n == 1: return 1` なので、 `1` を返す
            2. `n=2` なので、 `fibonatti(2 - 2)` となり、 `fibonatti(0)` を呼び出す
                1. `if n == 0: return 0` なので、 `0` を返す
            3. `fibonatti(1) + fibonatti(0)` なので、 `1+0` となり、 `return 1` となる
        2. `n=3` なので、 `fibonatti(3 - 2)` となり、 `fibonatti(1)` を呼び出す
            1. `if n == 1: return 1` なので、 `1` を返す
        3. `fibonatti(2) + fibonatti(1)` なので、 `1+1` となり、 `return 2` となる
    2. `n=4` なので、 `fibonatti(4 - 2)` となり、 `fibonatti(2)` を呼び出す
        1. `n=2` なので、 `fibonatti(2 - 1)` となり、 `fibonatti(1)` を呼び出す
            1. `if n == 1: return 1` なので、 `1` を返す
        2. `n=2` なので、 `fibonatti(2 - 2)` となり、 `fibonatti(0)` を呼び出す
            1. `if n == 0: return 0` なので、 `0` を返す

        3. `fibonatti(1) + fibonatti(0)` なので、 `1+0` となり、 `return 1` となる
    3. `fibonatti(3) + fibonatti(2)` なので、 `2+1` となり、 `return 3` となる
2. `n=5` なので、 `fibonatti(5 - 2)` となり、 `fibonatti(3)` を呼び出す
    1. `n=3` なので、 `fibonatti(3 - 1)` となり、 `fibonatti(2)` を呼び出す
        1. `n=2` なので、 `fibonatti(2 - 1)` となり、 `fibonatti(1)` を呼び出す
            1. `if n == 1: return 1` なので、 `1` を返す
        2. `n=2` なので、 `fibonatti(2 - 2)` となり、 `fibonatti(0)` を呼び出す
            1. `if n == 0: return 0` なので、 `0` を返す
        3. `fibonatti(1) + fibonatti(0)` なので、 `1+0` となり、 `return 1` となる
    2. `n=3` なので、 `fibonatti(3 - 2)` となり、 `fibonatti(1)` を呼び出す
        1. `if n == 1: return 1` なので、 `1` を返す
    3. `fibonatti(2) + fibonatti(1)` なので、 `1+1` となり、 `return 2` となる
3. `fibonatti(4) + fibonatti(3)` なので、 `3+2` となり、 `retun 5` となる

`n` が `0` または `1` のときは、フィボナッチ数列の0番目と1番目に当たるので、初期値の `0` または `1` を返却している。それ以外のときは、「どの数字も前２つの数字を足した数字」であるので、フィボナッチ数列の `n-1` 番目の数値と `n-2` 番目の数値を足した結果を返却している。

オブジェクトとメソッド

文字列

オブジェクトは、同一性 (identity)、型、値を持っている。同一性とは、各オブジェクトに与えられる固有の番号のようなものである。 日本語において同一性という単語は一般的ではなく、言葉の意味が分かりにくいため、本科目では同一性のことをIDと呼ぶこととする。

多くのPythonの実行環境に置いて、IDにオブジェクトが格納されているメモリのアドレスが利用されており、生成されたあとは変更されることはない。オブジェクトのIDは、 `id` 関数を使って取得することができる。

オブジェクトの比較方法はに、以下の2種類がある。

1. `is` 演算子を使用して比較
2. `==` を使用して比較

1の `is` 演算子を使って比較すると、両側の被演算子のIDが等価であることを確認できる。

2の `==` を使って比較すると、オブジェクトの持つ「値」が等しいかどうかを確認できる。 「値」とはデータの内容であり、たとえ、両者のオブジェクトが別のアドレスに保存されていても、内容が等しければ `==` は `True` を返す。

一方、たとえ値が等しかったとしても、両者のオブジェクトが別のアドレスに保存されていれば、 `is` 演算子 は `False` を返す。

以下のプログラムで、 `==` と `is` の違いを見てみよう。

```
str1 = "abc"
str2 = "ab"
print(str1 is str2, str1 == str2)
print(id(str1), id(str2))

str2 += "c"
print(str1 is str2, str1 == str2)
print(id(str1), id(str2))
```

```
False False
4560938544 4580263088
False True
4560938544 4607035760
```

`str1 is str2` では、オブジェクトのIDの比較を行っている。変数 `str1` と変数 `str2` は、当然別のオブジェクトであるので、 `False` が返却されている。

`str1 == str2` では、値の比較を行っており、変数 `str1` の値は `abc` 、変数 `str2` の値は `ab` なので、結果は `False` になる。

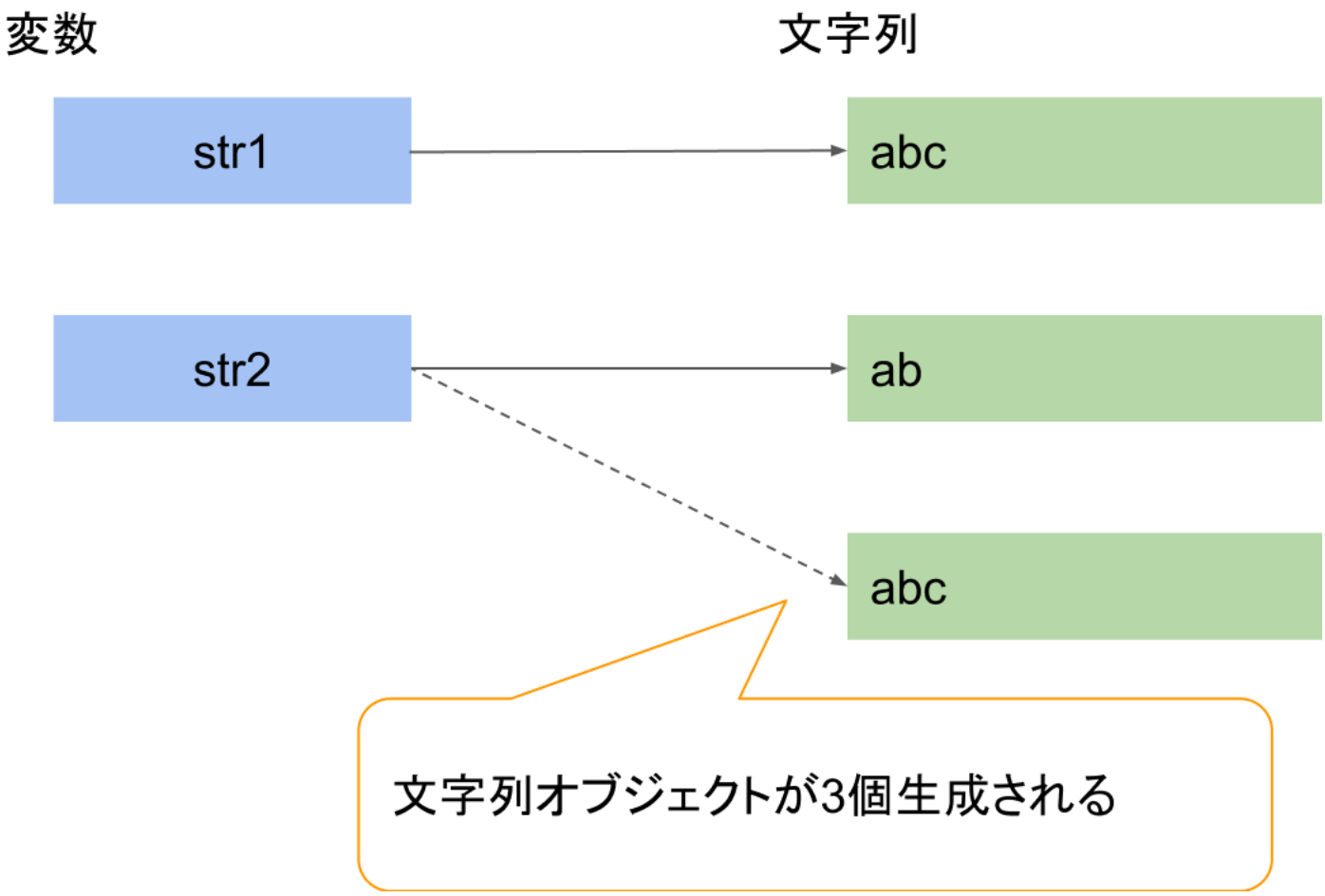
`id(str1)` と `id(str2)` では、それぞれのオブジェクトのIDを出力している。

では、 `str2 += "c"` が行われたあとはどうなるだろうか。

まず、 `str1 is str2` だが、変数 `str1` と変数 `str2` は、同一オブジェクトではないので、 `False` が返却されている。

`str1 == str2` では、変数 `str2` のは、元の値である `ab` に `c` が追加されて `abc` になったので、変数 `str1` の値である `abc` との比較の結果は `True` になる。

ここで、第2章で文字列は変更不可能なシーケンスであり、文字列に対する `+` 演算子は新しい文字列を生成すると説明したことを思い出してみよう。 `id(str2)` の1回目の出力と2回目の出力の結果はどうなるだろうか。出力結果を見てみると、異なる出力がされていることだろう。これは、 `str2 += "c"` が行われたあとの変数 `str2` は、行われる前の `str2` とIDが異なり、新しい文字列に置き換わっていることを示している。



オブジェクトはメソッドを持っている。メソッドはオブジェクトに属する関数のようなもので、典型的には、オブジェクトが持つ値に基づいて振る舞いが変わる。 まずは、文字列のメソッドをいくつか紹介しよう。次のプログラムを見てみよう。

```
hello = "Hello World"

print(hello.upper())
print(hello.startswith("H"))
print(hello.replace("World", "Python"))
```

```
HELLO WORLD
True
Hello Python
```

オブジェクト `hello` でいくつかのメソッドを実行してみた。

`upper()` は、変数 `hello` 内の全ての大小文字の区別のある文字が大文字に変換されるメソッドである。その逆（小文字に変換する）`lower()` といメソッドも存在する。

`startswith()` は、呼び出し時に指定された実引数でその文字列が始まっているかを判定し、 `True/False` を返却するメソッドである。末尾の判定を行う `endswith()` というメソッドもある。

他にも様々なメソッドがあるので、Pythonの公式ドキュメントを読んでみよう。

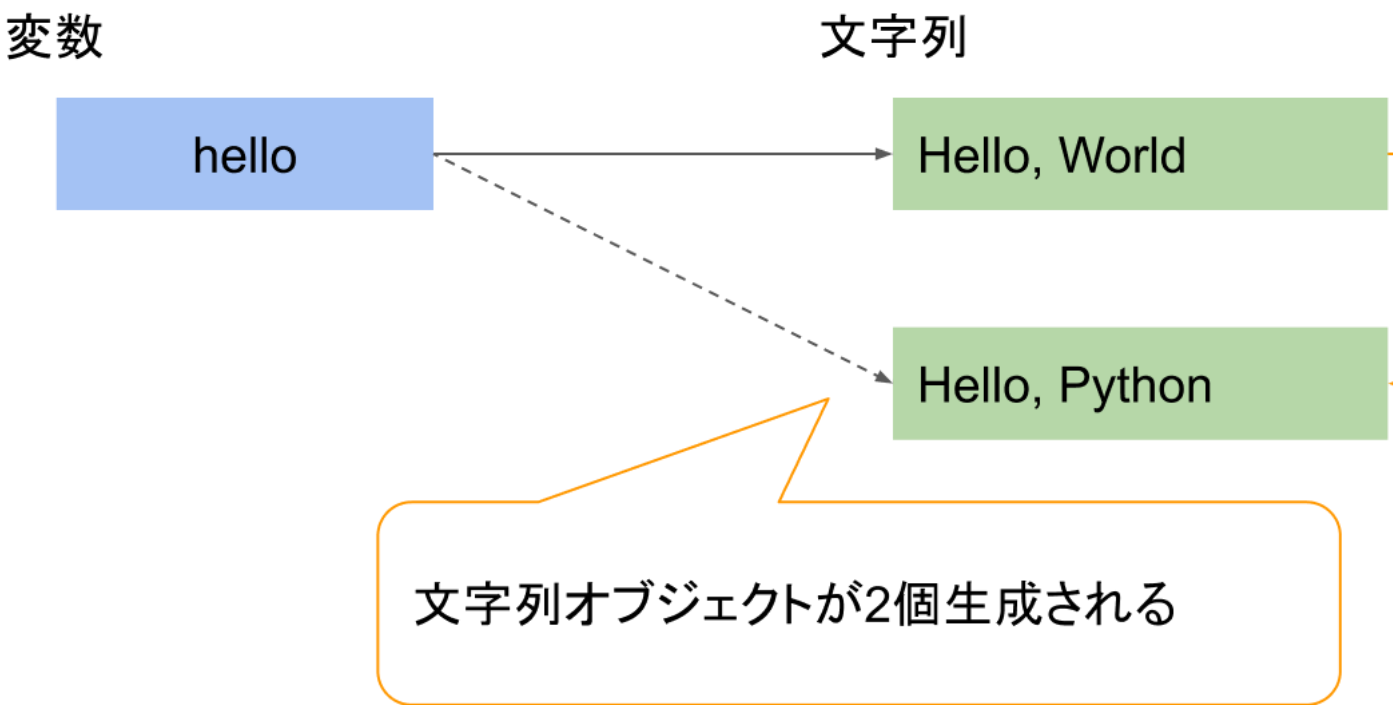
ここで、文字列は変更不可能なシーケンスであることを思い出して、以下のサンプルプログラムを見てみよう。

```
hello = "Hello, World"
print(id(hello))

hello = hello.replace("World", "Python")
print(id(hello))
```

```
4603490800
4607038832
```

`replace()` を使用し、変数 `hello` の `World` という文字列を、 `Python` という文字列に置き換える前と後の変数 `hello` のIDを比較すると、IDが変わっていることが確認できる。このように、文字列に対し `replace()` や `upper()` を実行すると、その度に新しい文字列が生成されるのである。



リスト

リストとは、コンマ区切りの値 (要素) の並びを角括弧で囲んだデータ型である。リストは異なる型の要素を含むこともあるが、通常は同じ型の要素のみを持つ。以下が、リストの例である。

```
list1 = [1, 2, 3, 4, 5]
list2 = ['one', 'two', 3, 4, 'five']
```

文字列と同様に、リストもメソッドを持っている。いくつかのリストのメソッドをプログラムした、サンプルプログラムを見てみよう。

```
list = [1, 2, 3]

list.append(4)
print(list)

list.extend([5, 6])
print(list)

list.reverse()
print(list)
```

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

`append()` はリストの末尾に実引数で渡された要素を追加する。

`extend()` は、 `append()` に似ているが、実引数にリストを取り、実引数のリストの要素をリストの末尾に追加（結合）する。

`reverse()` は、リストの要素の順番を逆順にする。

先程の文字列とは異なり、リストは変更可能なシーケンスである。なので、リストのメソッドを使うと、リストの値を書き換えることができる。サンプルプログラムを見ると、出力しているのは常に変数 `list` である。

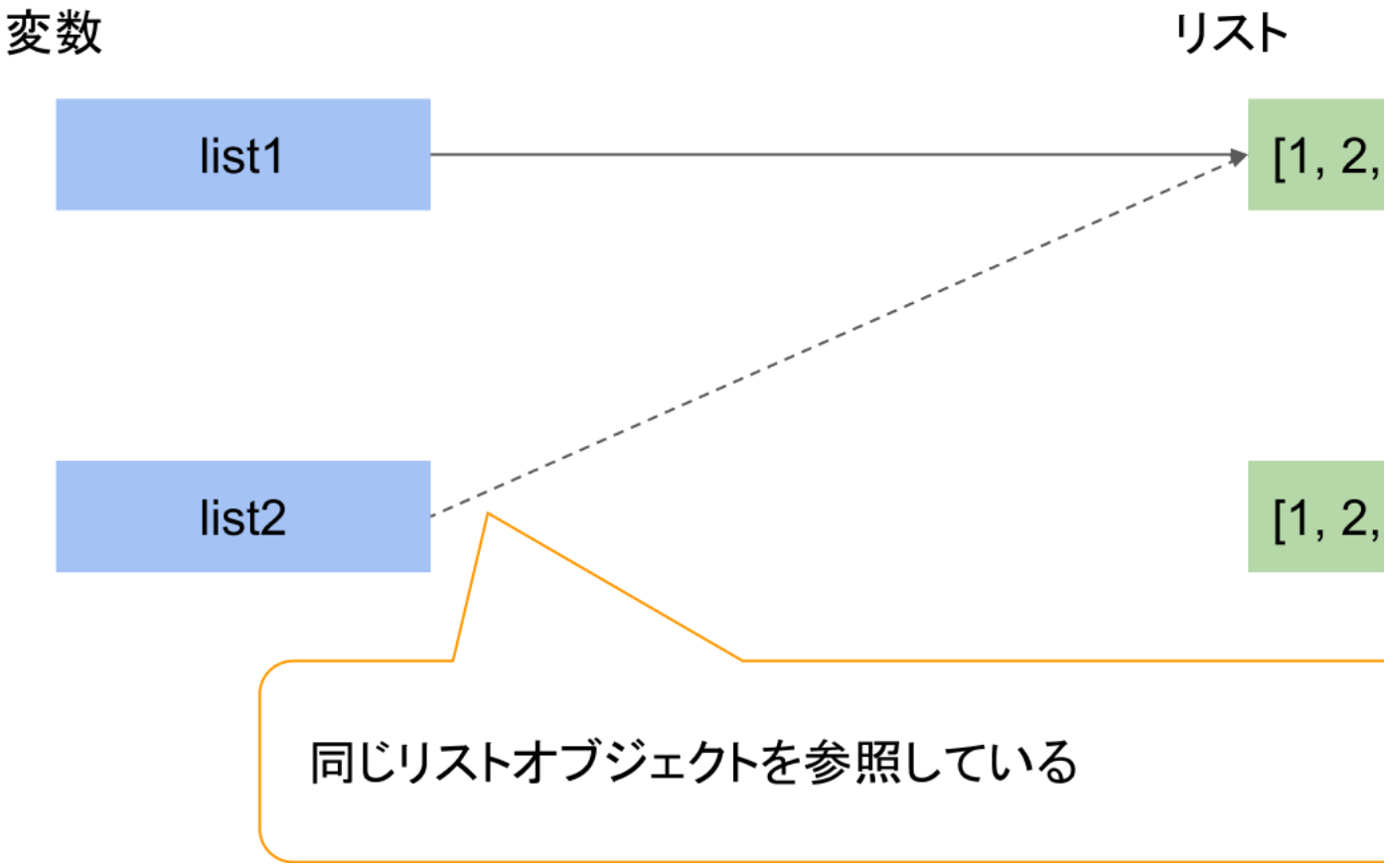
変数にオブジェクトを代入すると、オブジェクトのデータそのものを変数に格納するわけではなく、オブジェクトが存在する位置のみを変数に格納する。 このことを「変数はオブジェクトの参照を格納する」と呼ぶ。 したがって、ある変数が格納しているオブジェクトの参照を、別の変数に代入すると、複数の変数が同じオブジェクトを参照する状況を作り出せる。 早速、リストを使った次のプログラムの実行結果を見てみよう。

```
list1 = [1, 2, 3]
list2 = list1
print(list1, list2)

list1.append(4)
print(list1, list2)
```

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

list1とlist2は同じリストオブジェクトの参照を格納しているため、片方でリストの値を書き換えると、もう片方の変数からも書き換わった値を読み込める。出力を見ると、 `print(list1, list2)` で、同じ出力になっていることが確認できる。



文字列とは違い、リスト変更可能なシーケンスである。以下のサンプルプログラムを見てみよう。

```
list = [1, 2, 3]
print(id(list))

list.append(4)
print(id(list))
```

```
4603492928
4603492928
```

`append()` メソッドを実行する前と後の、オブジェクトのIDを比較してみると、IDが変わっていないことが確認できる。

文字列と同じように、値が同じであっても、参照しているオブジェクトが異なれば、オブジェクトのIDが等しくならないケースがある。 次のプログラムの実行結果を見てみよう。

```
list1 = [1, 2, 3]
list2 = [1, 2]
print(list1 is list2, list1 == list2)
print(id(list1), id(list2))

list2.append(3)
print(list1 is list2, list1 == list2)
print(id(list1), id(list2))
```

```
False False
4607090944 4607089728
False True
4607090944 4607089728
```

文字列でも解説したように、`is` 演算子は、オブジェクトのIDを比較している。`print(list1 is list2)` の出力は `False` になる。

一方で、`==` はオブジェクトの値を比較しているので、`list2.append(3)` を行った後の `print(list1 == list2)` の出力は、`True` になる。

リストは変更可能なシーケンスであるので、`list2.append(3)` を行う前でも後でも、`print(id(list2))` の結果が変わらない。

関数

実は関数もオブジェクトの一種であり、変数に代入することができる。次のサンプルプログラムを見てみよう。

```
def hello(name):
    print('Hello ' + name)

hello('Taro')
greet = hello
greet('Taro')
```

```
Hello Taro
Hello Taro
```

`hello` 関数を定義した後に、`greet` 変数に `hello` 関数のオブジェクトを代入している。その後、`hello` 関数と同じように、`greet` の関数呼び出しを記述すると、`hello` 関数と全く同じ動きを取る。関数の引数に関数オブジェクトを渡すこともでき、関数の呼び出し側が関数の挙動をカスタマイズすることもできる。次にサンプルプログラムを見てみよう。

```
def greet_all(names, greet):
    for name in names:
        greet(name)

def hello(name):
    print("Hello " + name)

def bye(name):
    print("Bye " + name)

greet_all(["Taro", "Jiro"], hello)
print("-----")
greet_all(["Taro", "Jiro"], bye)
```

```
Hello Taro
Hello Jiro
-----
Bye Taro
Bye Jiro
```

`greet_all` 関数は、引数で受け取った `names` リストの各要素を、引数で受け取った `greet` 関数オブジェクトに渡している。1回目の呼び出しでは `hello` 関数のオブジェクトを渡しているため、`Hello Taro` と `Hello Jiro` が表示される。一方、2回目の呼び出しでは `bye` 関数のオブジェクトを渡しているため、`Bye Taro` と `Bye Jiro` が表示される。なお、一般に、関数を引数で受け取る関数のことを高階関数と呼ぶ。

問題1

問題文

文字列 `function1` を出力する関数 `func1` の関数呼び出しを記述せよ。

解答の雛形

```
def func1():
    print("function1")

# ここに解答を入力
```

問題2

問題文

文字列 `function2` を出力する関数 `func2` を作成せよ。

解答の雛形

```
# ここに解答を入力
func2()
```

問題3

問題文

トリボナッチ数列の10番目の値を求める関数を**for文を用いて**作成せよ。

なおフィボナッチ数列は直前2項の和として各項が定まるのに対し、トリボナッチ数列は直前3項の和として各項が定まる。

トリボナッチ数列の第0,1,2項の値はそれぞれ0,0,1である。

解答の雛形

```
def tribonacci(n):
    a = 0
    b = 0
    c = 1
    for _ in range(n):
    # ここに解答を入力

print(tribonacci(10))
```

問題4

問題文

トリボナッチ数列の10番目の値を求める関数を**再帰関数を用いて**作成せよ。

解答の雛形

```
def tribonacci(n):
    if n == 0: return 0
    if n == 1: return 0
    if n == 2: return 1
    # ここに解答を入力

print(tribonacci(10))
```

問題5

問題文

`.` を横方向に5個出力する、つまり `.....` と表示するように関数呼び出しを記述せよ。

解答の雛形

```
def character_print(character='.', length=10, is_vertical=True):
    if is_vertical: #縦方向にlength回出力する
        for _ in range(length):
            print(character)
    else: #横方向にlength回出力する
        for _ in range(length):
            print(character,end='')
        print()

# ここに解答を入力
```

問題6

問題文

整数が格納されたリスト `nums` および整数 `value` に対して、 `nums` の要素中の最大値が `value` 以上かを判定する関数を作成せよ。

またbool値 `is_max` を追加で与えることにより、 `is_max` が `False` であるときには最小値が `value` 以下かを判定するようにせよ。ただし引数として `is_max` の値が与えられない場合には `True` であるものとする。

条件を満たすとき `Yes` 、満たさないならば `No` を出力するようにせよ。

なお、リスト `nums` の中の最大値、最小値はそれぞれ `max(nums)` 、 `min(nums)` で取得できる。

解答の雛形

```
# ここに解答を入力

judge_maxmin(value=2, nums=[0,2,1,3])
```

問題7

問題文

以下に示すプログラムを記述せよ。

- `str1` に文字列 `Hello` を代入する
- `str2` に、 `str1` の `Hello` を `World` に置換した文字列を代入する(`replace` メソッドを用いて)

そして以下に示すものを `print()` 関数を用いて出力・確認せよ。

- `str1` と `str2` を等号比較し、 `str1` に変化があるかを確認する
- `id(str1)` と `id(str2)` を等号比較し、 `id(str1)` と `id(str2)` が同じかどうかを確認する。

解答の雛形

```
# ここに解答を入力
```

問題8

問題文

`str1` に文字列 `Hello World Python` を代入し、 `str1` の文字列全てを大文字に変えたものを文字列 `str2` とする。

`str2` を出力し、その下に `str2` を空白区切りでリスト形式にしたものを出力するプログラムを作成せよ。

解答の雛形

```
# ここに解答を入力
```

5章: コレクション

コレクションの概要

コレクションには様々な種類があり、コレクションを組み合わせることで、さらに複雑なコレクションが作られている。 まずは、基盤となるコレクションについて、代表的なものを取り上げて説明する。 なお、これらのコレクションはユーザーが直接利用するケースは少なく、 これまで説明してきたリストや辞書などを構成する部品となっている。

- `Container` : `__contains__()` メソッドを持つ。つまり、 `in` 演算子で利用できる。
- `Sized` : `__len__()` メソッドを持つ。つまり、 `len()` 関数の引数に渡せる。
- `Iterable` : `__iter__()` メソッドを持つ。つまり、 `for` 文に渡せる。
- `Reversible` : `__reversed__()` メソッドを持つ。つまり、 `reversed()` 関数の引数に渡せる。
- `Collection` : `Sized` , `Iterable` , `Container` の組み合わせである。
- `Sequence` : `Reversible` , `Collection` の組み合わせであり、 `__getitem__()` メソッドを持つ。つまり、逆順に走査できるコレクションで、インデクサを持っている。

ユーザーが直接利用するリスト・タプル・集合・辞書について見ていこう。それぞれの特徴は次のとおりだ。

- リスト: 最も一般的なコレクション。順序付けられた任意の長さの要素の集まり。
- タプル: 変更不可能なリストであるコレクション。少量のデータをひとまとめにして引数に渡したり、戻り値から返したり、辞書型のキーに使うといった用途が一般的。
- 集合: 重複する要素をもたない、順序づけられていない要素の集まりを表現するためのコレクション。集合に特定の要素が含まれているかどうか確認することに適したコレクション。
- 辞書: キーと値のペアであり、キーは重複する要素を持ってはいけない。キーから値を探す用途に適したコレクション。

以下で、上記のコレクション間の関係性を調べるサンプルプログラムを示す。 `import` 文や `isinstance` 関数など、まだ扱ったことのないものを使っているが、 サンプルプログラムの詳細は深く考えずに、サンプルプログラムの実行結果を見ていこう。

```
from collections.abc import Container, Sequence, Sized, Iterable, Reversible, Collection


l = [1, 2, 3]
print("List は Container  の一種:", isinstance(l, Container))
print("List は Sized      の一種:", isinstance(l, Sized))
print("List は Iterable   の一種:", isinstance(l, Iterable))
print("List は Reversible の一種:", isinstance(l, Reversible))
print("List は Collection の一種:", isinstance(l, Collection))
print("List は Sequence   の一種:", isinstance(l, Sequence))
print("-----------------------------------")

t = (1, 2, 3)
print("Tuple は Container  の一種:", isinstance(t, Container))
print("Tuple は Sized      の一種:", isinstance(t, Sized))
print("Tuple は Iterable   の一種:", isinstance(t, Iterable))
print("Tuple は Reversible の一種:", isinstance(t, Reversible))
print("Tuple は Collection の一種:", isinstance(t, Collection))
print("Tuple は Sequence   の一種:", isinstance(t, Sequence))
print("-----------------------------------")


s = {1, 2, 3}
print("Set は Container  の一種:", isinstance(s, Container))
print("Set は Sized      の一種:", isinstance(s, Sized))
print("Set は Iterable   の一種:", isinstance(s, Iterable))
print("Set は Reversible の一種:", isinstance(s, Reversible))
print("Set は Collection の一種:", isinstance(s, Collection))
print("Set は Sequence   の一種:", isinstance(s, Sequence))
print("-----------------------------------")


d = {"a":1, "b":2, "c":3}
print("Dictionary は Container  の一種:", isinstance(d, Container))
print("Dictionary は Sized      の一種:", isinstance(d, Sized))
print("Dictionary は Iterable   の一種:", isinstance(d, Iterable))
print("Dictionary は Reversible の一種:", isinstance(d, Reversible))
print("Dictionary は Collection の一種:", isinstance(d, Collection))
print("Dictionary は Sequence   の一種:", isinstance(d, Sequence))
```

```
List は Container  の一種: True
List は Sized      の一種: True
List は Iterable   の一種: True
List は Reversible の一種: True
List は Collection の一種: True
List は Sequence   の一種: True
----------------------------------
Tuple は Container  の一種: True
Tuple は Sized      の一種: True
Tuple は Iterable   の一種: True
Tuple は Reversible の一種: True
Tuple は Collection の一種: True
Tuple は Sequence   の一種: True
----------------------------------
Set は Container  の一種: True
Set は Sized      の一種: True
Set は Iterable   の一種: True
Set は Reversible の一種: False
Set は Collection の一種: True
Set は Sequence   の一種: False
----------------------------------
Dictionary は Container  の一種: True
Dictionary は Sized      の一種: True
Dictionary は Iterable   の一種: True
Dictionary は Reversible の一種: True
Dictionary は Collection の一種: True
Dictionary は Sequence   の一種: False
```

いずれも `Container`, `Sized`, `Iterable` の一種である。つまり、後述の `in` 演算子が使え、`len()` 関数で要素の数を数えることができ、`for` 文などで各要素を取り出すことができる。また、いずれも上記3つの特徴を兼ね備えた `Collection` の一種でもある。 リストとタプルは、`Reversible` と `Sequence` の一種でもある。つまり、逆順に要素をとりだしたり、インデクサ(`[]`)で任意の位置の要素を取り出したりできる。 Setは順序付けられた要素の集合ではないため、`Reversible` と `Sequence` の一種ではない。 Dictionaryは逆順に要素をとりだしたり、インデクサ(`[]`)で任意の位置の要素を取り出したりできるが、インデクサに渡す値が数値ではなく、キーで指定した値であり、シーケンスとは挙動が異なるから、`Reversible` ではあるが、`Sequence` ではない。

リストが変更可能なシーケンスであることを確認しよう。

```
l = [1,2,3]
l.append(4)
print(l)
```

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

`append` 関数の引数に `4` を渡し、リスト `l` に `4` を追加するプロログラムである。`print(l)` で実行結果を確認すると、`4` が追加されていることが確認できる。

リストと違い、タプルは変更不可能なシーケンスである。サンプルプログラムで確認してみよう。

```
t = (1,2,3)
t.append(4)
print(t)
```

タプル `t` は変更不可能なシーケンスであるので、`append()` メソッドが用意されていない。出力を確認するとエラーになっていることが確認できる。

シーケンス (Sequence)

List と Tuple は Sequence の一種であるため、どちらもSequenceのメソッドや演算子を使える。そのうちの一部を紹介する。

なお、各クラスのメソッドの一覧はこちらのページから見ることができる。例えば、Sequenceであれば、Pythonの公式ドキュメントのトップページからライブラリーリファレンスのリンクを押して、目次からシーケンス型 --- list, tuple, rangeのページに遷移することでメソッドの一覧を見れる。

Sequenceのメソッドから、インデクサとスライスを紹介する。次のサンプルプログラムを見てみよう。

```
# インデクサとスライス
list1 = ['a', 'b', 'c', 'd', 'e']
tuple1 = ('a', 'b', 'c', 'd', 'e')

def check_index_and_slice_of_sequence(sequence):
    print(sequence[2])
    print(sequence[1:4])
    print(sequence[2:])
    print(sequence[:2])
    print(sequence[:])

check_index_and_slice_of_sequence(list1)
print("-----")
check_index_and_slice_of_sequence(tuple1)
```

```
c
['b', 'c', 'd']
['c', 'd', 'e']
['a', 'b']
['a', 'b', 'c', 'd', 'e']
-----
c
('b', 'c', 'd')
('c', 'd', 'e')
('a', 'b')
('a', 'b', 'c', 'd', 'e')
```

リスト `list1` が実引数に指定されている場合を例に取り、 `check_index_and_slice_of_sequence` 関数内で出力をしている箇所をひとつずつ見ていこう。

`sequence[2]` は、「引数 `sequence` の0から数えて `2` 番目の要素を取り出す」プログラムなので、出力は `c` になる。

`sequence[1:4]` は、「引数 `sequence` の要素の中から、0から数えて `1` 番目から `4` 番目までを取り出す」プログラムである。「`1` から `4` まで」を数式で表すと、 `1 <= 要素 < 4` となり、0から数えて `4` 番目の要素は含まれないので注意しよう。なので、出力は `['b', 'c', 'd']` となる。

`sequence[2:]` を見てみると、 `sequence[1:4]` と似ていることが分かる。この `sequence[2:]` は、 `sequence[1:4]` の、 `:` の後の数字が省略されている形である。 `:` の後の数字が省略された場合、 `:` の後ろには `len(sequence)` が補完される。なので、 `sequence[2:]` は、 `sequence[2:5]` という値で実行されているのである。 そのため、 `sequence[2:]` は、「引数 `sequence` の要素の中から、0から数えて `2` 番目から末尾(`len(sequence)`)までを取り出す」プログラムであるので、出力は `['c', 'd', 'e']` となる。

`sequence[:2]` も同様に、 `sequence[1:4]` の、 `:` の前の数字が省略されている形である。 `:` の前の数字が省略された場合、 `:` の前には `0` が補完される。なので、 `sequence[:2]` は、 `sequence[0:2]` という値で実行されているのである。 は、「引数 `sequence` の要素の中から、 `0` 番目から数えて `2` 番目までを取り出す」プログラムとなるので、出力は `['a', 'b']` となる。

`sequence[:]` はどうだろうか。 `:` の前の数字が省略された場合は、 `:` の前には `0` が補完され、 `:` の後ろには `len(sequence)` が補完されるので、 `sequence[:]` は `sequence[0:5]` として実行される。ということは、「引数 `sequence` の要素の中から、0から数えて末尾(`len(sequence)`)までを取り出す」プログラムになるので、出力は `['a', 'b', 'c', 'd', 'e']` となる。

コンテナのメソッドを紹介する。リスト・タプル・集合・辞書はどれもコンテナであるため、リスト・タプル・集合・辞書すべてで使用できる。

```
# コンテナのメソッド紹介
list1 = ['a', 'a', 'b', 'c', 'd', 'e']
tuple1 = ('a', 'a', 'b', 'c', 'd')

def check_methods_of_container(sequence):
    print('e' in sequence, 'e' not in sequence)

check_methods_of_container(list1)
print("-----")
check_methods_of_container(tuple1)
```

```
True False
-----
False True
```

`check_methods_of_container` 関数内を見てみよう。 `'e' in sequence` は、仮引数 `sequence` 内に、'e'と等しい要素があれば `True` 、なければ `False` が返却される。

`'e' not in sequence` はその逆で、'e'と等しい要素が無ければ `True` 、あれば `False` が返却される。

イテラブル（Iterable）のメソッドを紹介する。リスト・タプル・集合・辞書はどれもイテラブルなので、すべてで使える。イテラブルはfor文を使うことができる。

```
# シーケンスのメソッド紹介
list1 = ['blue', 'gray', 'orange', 'red', 'yellow']
tuple1 = ('blue', 'gray', 'orange', 'red', 'yellow')

def check_methods_of_iterable(sequence):
    for e in sequence:
      print(e)

check_methods_of_iterable(list1)
print("-----")
check_methods_of_iterable(tuple1)
```

```
blue
gray
orange
red
yellow
-----
blue
gray
orange
red
yellow
```

`for` 文については第3章で解説したので省略する。イテラブルの一種であるリスト・タプル・集合・辞書などは `for` 文を使うことができる。

アンパックについて解説する。アンパックとは、すでにリストやタプルなど、コレクションの要素を、個別の変数に代入することである。サンプルプログラムを見てみよう。

```
list1 = [0, 1, 2, 3]
tuple1 = (0, 1, 2, 3)
set1 = {0, 1, 2, 3}
dic1 = {"first":1, "second":2, "third":3, "fourth":4}

def check_unpack_of_sequence(first="", second="", third="", fourth=""):
  print("１つ目の要素: ", first)
  print("2つ目の要素: ", second)
  print("3つ目の要素: ", third)
  print("4つ目の要素: ", fourth)


check_unpack_of_sequence(*list1)
print("-----")
check_unpack_of_sequence(*tuple1)
print("-----")
check_unpack_of_sequence(*set1)
print("-----")
check_unpack_of_sequence(*dic1)
print("-----")
check_unpack_of_sequence(**dic1)
```

```
１つ目の要素:  0
2つ目の要素:  1
3つ目の要素:  2
4つ目の要素:  3
-----
１つ目の要素:  0
2つ目の要素:  1
3つ目の要素:  2
4つ目の要素:  3
-----
１つ目の要素:  0
2つ目の要素:  1
3つ目の要素:  2
4つ目の要素:  3
-----
１つ目の要素:  first
2つ目の要素:  second
3つ目の要素:  third
4つ目の要素:  fourth
-----
１つ目の要素:  1
2つ目の要素:  2
3つ目の要素:  3
4つ目の要素:  4
```

`check_unpack_of_sequence(*list1)` などの関数の呼び出し箇所に注目してみよう。 `check_unpack_of_sequence` 関数は、文字列の仮引数 `a` 、 `b` 、 `c` 、 `d` を定義しているのに、呼び出し元では `*list1` や `*set1` のように、リストやタプルなどのコレクションが指定されている。これが許容されるのはなぜだろう。

呼び出し元をよく見てみると、実引数に `*` がついている。これが引数のアンパックである。 `check_unpack_of_sequence` で要求されている仮引数 `a` 、 `b` 、 `c` 、 `d` に、リストやタプルなどのコレクションの要素を当てはめ、関数を呼び出しているのである。

`check_unpack_of_sequence(*list1)` を例にとってみよう。実引数として、アンパックされた `list1` が指定されているので、実際は、 `check_unpack_of_sequence((first="0", second="1", third="2", fourth="3")` と呼び出されているのである。

では、キーとバリューを持つ辞書をアンパックした場合はどうなるだろう。 まず、 `check_unpack_of_sequence(*dic1)` では、仮引数 `a` 、 `b` 、 `c` 、 `d` に、 `dic1` のキーが先頭から当てはめられている。 `print("１つ目の要素: ", first)` では、 `１つ目の要素: first` と出力されているだろう。

辞書をアンパックして、バリューを当てはめたい場合は、`**` を使おう。`check_unpack_of_sequence(**dic1)` のようにすると、仮引数 `a`、`b`、`c`、`d` に、`dic1` のバリューが当てはめられる。`print("１つ目の要素: ", first)` を見ると、`１つ目の要素: 0` と出力されているだろう。

リスト （List）

リストとは、複数のデータに番号を付与して特定の順番で並べたデータ構造である。サンプルプログラムを見てみよう。

```
colors = ["red", "green", "blue"]
print(colors)
```

```
['red', 'green', 'blue']
```

`colors` は、複数の色について順番に並べたリストである。

次に、リストで使えるメソッドを見ていこう。

まずは、4章でも紹介した `append` メソッドである。`appned` メソッドは、受け取った引数をリストの末尾に追加するメソッドである。`append` メソッドは、`colors += ["magenta"]` と書き換えることもできる。

```
colors = ["red", "green", "blue"]
colors.append("yellow")
print(colors)

colors += ["magenta"]
print(colors)
```

```
['red', 'green', 'blue', 'yellow']
['red', 'green', 'blue', 'yellow', 'magenta']
```

出力を確認すると、`yellow` と `magenta` がリストの末尾に追加されていることが確認できる。

`extend` メソッドは、`append` メソッドに似ているが、違う点は引数にリストが取れることができ、そのリストの要素を対象のリストの末尾に追加するということである。

```
colors = ["red", "green", "blue"]
colors.extend(["orange", "purple"])
print(colors)
```

```
['red', 'green', 'blue', 'orange', 'purple']
```

`["orange", "purple"]` を引数に取り実行すると、`orange` と `purple` の2つがリストの末尾に追加された。

`remove` は、リストの要素のうち、受け取った引数が等価となる**最初の**要素を取り除くメソッドである。

```
colors = ["red", "green", "blue", "green"]
colors.remove("green")
print(colors)
```

```
['red', 'blue', 'green']
```

出力を確認すると、リスト `colors` の中にある先頭から数えて１つ目の `green` 要素だけ削除されていることが確認できる。

`pop` メソッドは、リストから指定のインデックスの要素を取り出し、または、取り除くことができる。

```
colors = ["red", "green", "blue"]
color = colors.pop(1)
print(color)
print(colors)
```

```
green
['red', 'blue']
```

`colors.pop(1)` では、リスト `colors` の0から数えて `1` 番目の要素を取り出している。インデックスの指定がないときは、デフォルトで `-1` がセットされるため、リストの最後の要素が取り出される。

`pop` メソッドは、値を取り出しているだけでなく、リストの中身も書き換えられているので注意しよう。

`apend` メソッドでは、要素は無条件に最後尾に追加されたが、 `insert` メソッドは、指定したインデックスに指定した要素を追加することができる。

```
colors = ["red", "green", "blue"]
colors.insert(1, "black")
print(colors)
```

```
['red', 'black', 'green', 'blue']
```

`colors.insert(1, "black")` では、リスト `colors` の1番目に `black` という要素を追加している。

`del` は、指定されたリストの要素を削除する。

```
colors = ["red", "green", "blue"]
del colors[0]
print(colors)
```

```
['green', 'blue']
```

`remove` では、リストの要素の値を指定して、リスト内を先頭から検索し、見つけた最初の要素の最初を取り除いていたが、 `del` ではインデックスを指定して要素を取り除く。

`enumerate` は、受け取った引数のリストの各要素に、インデックスをつけた新しいオブジェクトを作成する。

```
colors = ["red", "green", "blue"]
print(*enumerate(colors))

for (index, value) in enumerate(colors):
    print(f"{index}: {value}")
colors.clear()
print(colors)
```

```
(0, 'red') (1, 'green') (2, 'blue')
0: red
1: green
2: blue
[]
```

`for (index, value) in enumerate(colors):` では、リスト `colors` の各要素にインデックスをつけ、`print(f"{index}: {value}")` でインデックスと要素を出力している。

`sort` メソッドは、その名の通り、リスト内の要素をソートするメソッドである。数字だけでけでなく、文字列もソートできる。

```
list1 = [1,3,5,0,6,7,8,9,2,4]
list1.sort()
print(list1)

list2 = ["A", "d", "C", "b", "c", "B", "D", "a"]
list2.sort()
print(list2)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
['A', 'B', 'C', 'D', 'a', 'b', 'c', 'd']
```

ソートの結果を確認すると、リスト `list1` は0から順番に並び替えられており、文字列も `A` から順に並び替えられている。

`reverse` メソッドは、要素の順番を逆転させる。

```
num = [1,3,5,0,6,7,8,9,2,4]
num.reverse()
print(num)
```

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

出力を確認すると、リストの要素が逆順になっていることを確認できる。

リストを複製したいときはどうすれば良いだろう。次のサンプルプログラムを見てみよう。

```
animals = ["dog", "cat", "rabbit"]
pets = animals
pets.append("fish")
print(f"animals: {animals}")
print(f"pets: {pets}")

print(id(animals))
print(id(pets))
```

```
animals: ['dog', 'cat', 'rabbit', 'fish']
pets: ['dog', 'cat', 'rabbit', 'fish']
4357304256
4357304256
```

`pets = animals` で変数 `pet` にリスト `animals` をコピーして、 `pets.append("fish")` でリスト `pets` のみに `fish` という要素を追加したつもりが、出力を確認すると、 `animals` にも `fish` が追加されてしまっている。 これは、 `animals` の要素のみを `pets` に代入したのではなく、 `animals` の同一性を共有してしまっているからである。 `id` 関数を使って同一性を確認してみると、 `animals` でも `pets` でも同一性が同じであることが確認できる。

こういうときは `copy` メソッドを使おう。

```
animals = ["dog", "cat", "rabbit"]
pets = animals.copy()
animals.remove("dog")
pets.append("snake")
print(f"animals: {animals}")
print(f"pets: {pets}")

print(id(animals))
print(id(pets))
```

```
animals: ['cat', 'rabbit']
pets: ['dog', 'cat', 'rabbit', 'snake']
4375792000
4375958080
```

`pets = animals.copy()` と、 `copy` メソッドを使って変数 `pets` にリスト `animal` をコピーした。すると、 `animals.remove("dog")` や `pets.append("snake")` と、それぞれのリストに操作しても、その操作はお互いに干渉していないことが確認できる。

`id` 関数を使って同一性を確認すると、違う値であることが確認できる。

タプル (Tuple)

タプルとは、複数のデータの組み合わせをひとかたまりのデータを組みとして保持するデータ構造である。タプルはリストとよく似ているが、リストは複数のデータの繰り返しに用いることが通常であり、タプルは全体が一つのデータであって、各要素をデータの属性を表すために用いる。例えば、とあるシステムのユーザー名一覧を一つのデータ組みで保持するにはリストを用いるが、とあるユーザーについて、名前、年齢、性別、メールアドレスなどをひとつのデータ組みで持つときはタプルを用いる。

サンプルプログラムを見てみよう。

```
user1 = ("太郎", "田中", 36, "M", "taro_tanaka@example.com")
print(user1)

firstName, lastName, age, gender, email = user1
print(firstName, lastName, age, gender, email)
```

```
('太郎', '田中', 36, 'M', 'taro_tanaka@example.com')
太郎 田中 36 M taro_tanaka@example.com
```

`user1` は、とあるユーザーの情報を一つにまとめたタプルである。 `firstName, lastName, age, gender, email = user1` のように、タプルをアンパックすることもできる。

イテラブルの一種であるタプルは `for` 文を使うことができる。サンプルプログラムを見てみよう。

```
user1 = ("John", "Smith", 36, "M", "john_smith@example.com")

for value in user1:
    print(value)
```

```
John
Smith
36
M
john_smith@example.com
```

集合 (Set)

集合とは、重複を許さず、また、順序を保持せずに複数の値の集合を記録するデータ構造である。サンプルプログラムを見てみよう。

```
colors = {"red", "green", "blue", "red", "green", "green", "green", "green", "green", "green", "green"}
print(colors)
```

```
{'red', 'blue', 'green'}
```

上記が集合のサンプルプログラムである。集合は重複を許さないので、2つ目以降の `red` や `green` は削除されていることが確認できる。

集合に要素を追加したいときは、 `add` を使おう。

```
colors = {"red", "green", "blue"}
colors.add("yellow")
colors.add("yellow")
colors.add("yellow")
print(colors)
```

```
{'yellow', 'red', 'blue', 'green'}
```

ここでも、集合は重複は許容されていないので、 `colors.add("yellow")` を3回実行しても、追加される要素 `yellow` はひとつである。

要素を削除したいときは、2つの方法がある。

```
colors = {"red", "green", "blue", "yellow"}
colors.remove("yellow")
print(colors)
```

```
{'blue', 'red', 'green'}
```

まず1つ目が `remove` である。要素 `yellow` が削除されていることが確認できる。 ただしこの `remove` は、削除したい要素が必ず集合内に存在していないといけない。削除したい要素が集合になかったときに `remove` を実行すると、エラーになってしまう。

削除した要素が集合に存在するかどうかわからないときは、 `discard` を使おう。

```
colors = {"red", "green", "blue"}
# colors.remove("yellow") # yellowがないとエラーが出る
colors.discard("yellow") # yellow がなくてもエラーが出ない (Listにはないメソッド)
colors.discard("green")
print(colors)
```

```
{'red', 'blue'}
```

集合 `colors` に要素 `yellow` は存在しないが、 `colors.discard("yellow")` を実行してもエラーにはならない。
`colors.discard("green")` で、存在する要素 `green` を削除できていることが出力から確認することができる。

集合同士を比較したいときはどうすればよいだろう。まずは、2つの集合で共通する要素を抜き出すサンプルプログラムを見てみよう。

```
colors = {"red", "green", "blue", "yellow"}
other_colors = {"cyan", "magenta", "yellow"}
print(colors.intersection(other_colors))
```

```
{'yellow'}
```

`intersection` を使うと、2つの集合で共通する要素を抜き出すことができる。 `colors` と `other_colors` で共通する要素 `yellow` が抜き出せているのが出力で確認することができる。

2つの集合を比較して、異なる要素を見つけたいときはどのようにするのか、サンプルプログラムを見てみよう。

```
colors = {"red", "green", "blue", "yellow"}
other_colors = {"cyan", "magenta", "yellow"}
print(colors.difference(other_colors))
print(other_colors.difference(colors))

print(colors)
colors.difference_update(other_colors)
print(colors)
```

```
{'red', 'blue', 'green'}
{'cyan', 'magenta'}
{'blue', 'red', 'yellow', 'green'}
{'blue', 'red', 'green'}
```

まず、 `difference` を紹介しよう。 `colors.difference(other_colors)` を例に取ると、「 `colors` に含まれており、 `other_colors` に含まれない要素」を抜き出している。同様に、 `other_colors.difference(colors)` は、「 `other_colors` に含まれており、 `colors` に含まれない要素」を抜き出している。出力を確認してみよう。

`difference` では、元の集合を更新しないが、 `difference_update` は元の集合を更新する。
`colors.difference_update(other_colors)` では、「 `colors` に含まれており、 `other_colors` に含まれない要素を抜き出し、 `colors` を更新」している。 `colors.difference_update(other_colors)` を実行する前と後では、 `print(colors)` の出力が変わっているのが確認できる。

次に、2つの集合を比較してどちらかの集合にしか含まれない要素を抜き出す方法を紹介しよう。

```
colors = {"red", "green", "blue", "yellow"}
other_colors = {"cyan", "magenta", "yellow"}
print(other_colors.symmetric_difference(colors))

print(colors)
colors.symmetric_difference_update(other_colors)
print(colors)
```

```
{'blue', 'magenta', 'green', 'cyan', 'red'}
{'blue', 'red', 'yellow', 'green'}
{'blue', 'magenta', 'green', 'cyan', 'red'}
```

`other_colors.symmetric_difference(colors)` では、「`colors` と `other_colors` のいずれか一方だけに含まれる要素」を抜き出している。 `symmetric_difference` では、元の集合は更新されない。

しかし、 `symmetric_difference_update` は、元の集合が更新される。 `colors.symmetric_difference_update(other_colors)` では、「`colors` と `other_colors` のいずれか一方だけに含まれる要素を抜き出し、 `colors` を更新」しているので、実行する前と後では `print(colors)` の結果が違うことが確認できる。

集合もイテラブルの一種なので、 `for` 文を使うことができる。サンプルプログラムを実行してみよう。

```
colors = {"red", "green", "blue", "yellow"}
for color in colors:
    print(color)
```

```
blue
red
yellow
green
```

辞書 (Dictionary)

辞書とは、複数のデータをキー（データ検索のID）とバリュー（実際のデータ）の対応関係で保持するデータ構造である。サンプルプログラムを見ていこう。

```
user = {"firstName": "太郎", "lastName": "田中"}
print(user)
print(user["firstName"])

print(user.get("firstName"))
print(user.get("age"))
print(user.get("address", "日本"))
```

```
{'firstName': '太郎', 'lastName': '田中'}
太郎
太郎
None
日本
```

変数 `user` は、とあるユーザーの情報を格納した辞書である。 `user["firstName"]` のように、辞書名の横に角括弧 `[]` を書き、その中に辞書のキーを指定することで、そのバリューを取得することができる。

また、`get`メソッドでもバリューを取得することができる。`user.get("firstName")`のように、引数に取得したいバリューのキーを指定することで、バリューを取得することができる。`user.get("age")`のように、存在しないキーを指定した場合、`None`が出力される。`user.get("address", "日本")`のように第２引数を指定することで、値が存在しなかった場合の出力を変更することができる。`user.get("address", "日本")`では、`日本`と出力される。

辞書に要素を追加するサンプルプログラムを見てみよう。

```
user = {"firstName": "太郎", "lastName": "田中"}
user.update({"address": "東京都", "age": "36"})
print(user)
```

```
{'firstName': '太郎', 'lastName': '田中', 'address': '東京都', 'age': '36'}
```

辞書`user`に、`address`という情報と`age`という情報を追加した。追加するときには、`update`メソッドを使用し、`user.update({"address": "東京都", "age": "36"})`のように、追加したいキーとバリューをセットにして引数にセットする。

要素のバリューを書き換えることもできる。

```
user = {"firstName": "太郎", "lastName": "田中"}
user["lastName"] = "佐藤"
print(user)
```

```
{'firstName': '太郎', 'lastName': '佐藤'}
```

上記のサンプルプログラムでは、`user["lastName"] = "佐藤"`で、`lastName`の値を`田中`から`佐藤`に書き換えている。

辞書の要素について、取得できる値はいくつかある。次のサンプルプログラムを見てみよう。

```
user = {"firstName": "太郎", "lastName": "田中"}

keys = user.keys()
print(keys)
for key in keys:
    print(key, "のバリューは ", user[key])
print("-----------------")

values = user.values()
print(values)
for value in values:
    print(value)
print("-----------------")

items = user.items()
print(items)
for item in items:
    print(item)
print("-----------------")

user.update({"address": "東京都", "age": "36"})
print(keys)
print(values)
print(items)
```

```
dict_keys(['firstName', 'lastName'])
firstName のバリューは  太郎
lastName のバリューは  田中
-----------------
dict_values(['太郎', '田中'])
太郎
田中
-----------------
dict_items([('firstName', '太郎'), ('lastName', '田中')])
('firstName', '太郎')
('lastName', '田中')
-----------------
dict_keys(['firstName', 'lastName', 'address', 'age'])
dict_values(['太郎', '田中', '東京都', '36'])
dict_items([('firstName', '太郎'), ('lastName', '田中'), ('address', '東京都'), ('age', '36')])
```

`keys` メソッドは、辞書のキーの一覧を取得することができる。 `values` メソッドは、辞書のバリューの一覧を取得することができる。 `items` メソッドは、辞書のキーとバリューのセットを取得することができる。

これらによって返却されるオブジェクトは、辞書ビューオブジェクトと呼ばれるもので、元になる辞書が更新された際に、辞書ビューオブジェクトも自動で更新される。

`user.update({"address": "東京都", "age": "36"})` が行われた後の `print(keys)` 、 `print(values)` 、 `print(items)` の出力を確認して、更新されているか確認してみよう。

末尾の要素を削除するときは、 `popitem` を使おう。

```
user = {"firstName": "太郎", "lastName": "田中", "address": "東京都", "age": "36"}
print(user.popitem())
print(user)
```

```
('age', '36')
{'firstName': '太郎', 'lastName': '田中', 'address': '東京都'}
```

末尾の `"age": "36"` が削除されている。

辞書に特定のキーがあるかどうかを調べるときは `in` を使おう。

```
user = {"firstName": "太郎", "lastName": "田中"}
print(print("lastName" in user))
print("age" in user)
```

```
True
None
False
```

`print("lastName" in user)` を例にとると、 `lastName` というキーが辞書 `user` に存在しているか調べていて、存在する場合 `True` 、存在しない場合 `False` が返却される。

辞書でも `for` 文を使うことができる。

```
user = {"firstName": "太郎", "lastName": "田中"}
for value in user:
    print(value)

for(key, value) in user.items():
    print(f"{key} : {value}")
```

```
firstName
lastName
firstName : 太郎
lastName : 田中
```

問題1

問題文

文字列 `S` が与えられるので、この文字列の `A` 番目から `B` 番目と `C` 番目から `D` 番目までを結合した文字列を表示しなさい。

制約

- $1 \leq len(S) \leq 100$
- $1 \leq A \leq B \leq len(S)$
- $1 \leq C \leq D \leq len(S)$
- S は文字列(アルファベット大文字のみ)である
- A,B,C,D は整数である。

入力

入力は次の形式で与えられる。

```
S
A B C D
```

出力

問題文に即した文字列を出力せよ。

入力例1

```
ABCDE
1 2 4 5
```

出力例1

```
ABDE
```

入力例2

```
TOKYO
3 5 1 2
```

出力例2

```
KYOTO
```

解答の雛形

```
s = str(input())
A,B,C,D=(int(x) for x in input().split())
# ここに解答を入力
```

問題2

問題文

空のリストに対していくつかのクエリが以下の形式で与えられます。

- `-1` 処理を終了する。
- `0 x` リストの末尾に `x` を追加する。
- `1 x` リストから要素 `x` をすべて消去する。
- `2 x y` リストの要素 `x` をすべて `y` に置換する。

これらのクエリを処理するプログラムを書いて下さい。 また、 `-1` 以外の処理の後には、そのときのリストの形式を

制約

- $1 \leq x, y \leq 100$
- 入力は全て整数である。

入力

入力は次の形式の繰り返しで送られる。

```
QUERY
```

QUERYの形式は以下の4つのいずれかである。

- `-1`
- `0 x`
- `1 x`
- `2 x y`

そして入力の最後のQUERYは `-1` である。

出力

問題文に即してリストを操作したときの途中状態を出力せよ。

入力例1

```
0 1
0 1
0 2
0 2
0 3
0 4
1 1
2 2 3
1 3
-1
```

出力例1

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

解答の雛形

```
li = []
while True:
    query = [int(x) for x in input().split()]
    q = query[0]
    # ここに解答を入力
    print(li)
```

問題3

問題文

整数列のリスト `A` が与えられます。これをソートして大きい方から `X` 番目と小さい方から `Y` 番目の要素を表示してください。

制約

- $1 \leq len(A) \leq 100$
- $1 \leq Ai \leq 100$
- $1 \leq X, Y \leq len(A)$
- 入力は全て整数である。

入力

入力は次の形式で与えられる。

```
A_{0} A_{1} ...
X Y
```

出力

問題文に即した値を出力せよ。

入力例1

```
3 1 4 1 5 9 2 6 5
1 2
```

出力例1

```
9 1
```

解答の雛形

```
li = [int(x) for x in input().split()]
X,Y = (int(x) for x in input().split())
# ここに解答を入力
```

問題4

問題文

長さ3のタプルのリストが与えられます。このリストの各要素に対して(インデクスを3で割ったあまり)番目の要素を取り出して足し合わせてください。 例えばインデクスが1の要素ならば1番目の要素を、インデクスが5の要素ならば2番目の要素を、インデクスが9の要素ならば0番目の要素を取り出します。

制約

- $1 \leq X, Y, Z \leq 100$
- 入力は全て整数である。

入力

入力は次の形式で与えられる。

```
[TUPLE, TUPLE, ...]
```

TUPLEは次の形式で与えられる。

```
(X, Y, Z)
```

出力

問題文に即した値を出力せよ。

入力例1

```
[(1,10,100),(2,20,200),(3,30,300),(4,40,400)]
```

出力例1

```
325
```

解答の雛形

```
li = [x for x in input()[2:-2].split('),(')]
for i in range(len(li)):
    li[i] = tuple(int(x) for x in li[i].split(','))
# ここに解答を入力
```

問題5

問題文

空の集合に対していくつかのクエリが以下の形式で与えられます。

- `-1` 処理を終了する。
- `0 x` 集合に `x` を追加する。
- `1 x` 集合から `x` を消去する。
- `2` 集合を空にする。

これらのクエリを処理するプログラムを書いて下さい。

制約

- $1 \leq x \leq 1000$
- 入力は全て整数である。

入力

入力は次の形式で与えられる。

```
QUERY
QUERY
...
```

QUERYは次のうちいずれかの形式で与えられる。

- `-1`
- `0 x`
- `1 x`
- `2`

入力の最後の行は `-1` が与えられる。

出力

`-1` 以外の各処理の後の集合をそれぞれ出力せよ。

入力例1

```
0 1
0 2
0 3
1 2
2
0 111
-1
```

出力例1

```
{1}
{1, 2}
{1, 2, 3}
{1, 3}
set()
{111}
```

解答の雛形

```
s = set()
while True:
    query = [int(x) for x in input().split()]
    # ここに解答を入力
    print(s)
```

問題5

問題文

空の辞書に対していくつかのクエリが以下の形式で与えられます。

- `-1` 処理を終了する。
- `0 x y` 辞書に `x` をキーとする要素 `y` を追加する。もしすでに `x` をキーとする要素がある場合は要素を `y` に上書きする。
- `1 x` 辞書内の `x` をキーとする要素を消去する。

これらのクエリを処理するプログラムを書いて下さい。

制約

- $1 \leq x, y \leq 1000$
- 入力は全て整数である。

入力

入力は次の形式で与えられる。

```
QUERY
QUERY
...
```

QUERYは次のうちいずれかの形式で与えられる。

- `-1`
- `0 x y`
- `1 x`

入力の最後の行は `-1` が与えられる。

出力

`-1` 以外の各処理の後の辞書をそれぞれ出力せよ。

入力例1

```
0 1 2
0 1 55
0 2 1
1 1
1 2
-1
```

出力例1

```
{1: 2}
{1: 55}
{1: 55, 2: 1}
{2: 1}
{}
```

解答の雛形

```python
d = {}
while True:
    query = [int(x) for x in input().split()]
    q = query[0]
    # ここに解答を入力
    print(d)
```

6章: クラス

名前空間とスコープ

Pythonは名前空間の仕組みを活用してクラスという概念を実現している。そこで、まず、名前空間 (namespace) について解説する。 名前空間とは、名前からオブジェクトへの対応付けであり、変数の名前と格納されている値のペアの集合と捉えられる。 例えば、次のプログラムを実行した際に生成される名前空間には、変数 `a` と変数 `b` が含まれることとなる。

```
a = 1
b = 2
```

Pythonでは、ほとんどの名前空間が、Pythonの辞書(`Dictionary`)で実装されている。例えば、 `a = 1` というステートメントを実行すると、名前空間を表現している辞書（ `namespace` と呼ぶこととする。）に対して、 `"a"` という文字列のキーで `1` という数値を記憶させるという処理が実行される。したがって、上のプログラムはPythonの内部的には以下のような処理となる。

```
namespace = {}
namespace["a"] = 1
namespace["b"] = 2
```

名前空間にはグローバルなものとローカルなものがあり、グローバルな名前空間はモジュール内（1つのPythonファイル内）で共通の名前空間であり、ローカルな名前空間は特定の状況でのみ存在する名前空間である。ローカルな名前空間は様々なタイミングで作成されるが、その代表例の一つが関数が呼び出されたときである。関数が呼び出されると、関数の内部でローカルな名前空間が作成される。また、その関数から抜けると、作成したローカルな名前空間は削除される。どのようにして名前空間がプログラムの挙動に影響するかを理解するために、次のプログラムを見てみよう。

```
def func1():
    a = 1
    print(a)

    def func2():
        a = 2
        print(a)

    func2()
    print(a)

a = 0
func1()
print(a)
```

```
1
2
1
0
```

10行目（上から3番目）の `print()` 関数では、 `2` ではなく `1` が表示され、14行目（上から4番目）の `print()` 関数では、 `1` ではなく `0` が表示されていることに注意しよう。このような挙動になる理由は、 `func1` 関数の外側の名前空間、 `func1` 関数の内側の名前空間、 `func2` 関数の内側の名前空間がそれぞれ異なるからである。以下で、辞書を使って、上のプログラムの名前空間を表現したプログラムを示す。

```
global_namespace = {}

def func1():
    local_namespace1 = {}
    local_namespace1["a"] = 1
    print(local_namespace1["a"])

    def func2():
        local_namespace2 = {}
        local_namespace2["a"] = 2
        print(local_namespace2["a"])
        del local_namespace2

    func2()
    print(local_namespace1["a"])
    del local_namespace1

global_namespace["a"] = 0
func1()
print(global_namespace["a"])
```

```
1
2
1
0
```

上から1番目と3番目の `print()` 関数では `local_namespace1` の辞書（名前空間）を、2番目の `print()` 関数では `local_namespace2` の辞書（名前空間）、4番目の `print()` 関数では `global_namespace` の辞書（名前空間）を参照している。だから、上から3番目の `print()` 関数では、 `2` ではなく `1` が表示され、上から4番目の `print()` 関数では、 `1` ではなく `0` が表示されるのだ。

名前空間に関連してスコープという用語を解説する。スコープ(scope)とは、ある名前空間に直接アクセスできるような、Pythonプログラムのテキスト上の領域を指す。例えば、 `func2` の名前空間に対するスコープ（これを `func2` 関数のローカルスコープと呼ぶ）は、5行目から7行目が該当する。また、 `func1` 関数のローカルスコープは `func2` 関数のローカルスコープを含まないため、1,2,3,9,10行目が該当する。

変数を参照する際の名前空間を変えるために `nonlocal` と `global` というキーワードを使うことができる。 `nonlocal <変数名>` を実行すると、その外側の関数などの名前空間の変数を参照することができる。また、 `global <変数名>` を実行すると、グローバルな名前空間を参照することができる。次のプログラムを見てみてよう。

```
def scope_test():
    def do_local():
        text = "local"

    def do_nonlocal():
        nonlocal text
        text = "nonlocal"

    def do_global():
        global text
        text = "global"

    text = "test"
    do_local()
    print("内側の関数のローカルスコープでの代入後:", text)
    do_nonlocal()
    print("内側の関数のローカルスコープでnonlocalを使用した代入後:", text)
    do_global()
    print("内側の関数のローカルスコープでglobalを使用した代入後:", text)

scope_test()
print("グローバルスコープ:", text)
```

```
内側の関数のローカルスコープでの代入後: test
内側の関数のローカルスコープでnonlocalを使用した代入後: nonlocal
内側の関数のローカルスコープでglobalを使用した代入後: nonlocal
グローバルスコープ: global
```

`nonlocal` と `global` の挙動を深く理解するために、以下で、辞書を使って、上のプログラムの名前空間を表現したプログラムを示す。

```
global_namespace = {}

def scope_test():
    local_namespace1 = {}
    def do_local():
        local_namespace2 = {}
        local_namespace2["text"] = "local"
        del local_namespace2

    def do_nonlocal():
        local_namespace3 = {}
        # nonlocal text
        local_namespace1["text"] = "nonlocal"
        del local_namespace3

    def do_global():
        local_namespace4 = {}
        # global text
        global_namespace["text"] = "global"
        del local_namespace4

    local_namespace1["text"] = "test"
    do_local()
    print("内側の関数のローカルスコープでの代入後:", local_namespace1["text"])
    do_nonlocal()
    print("内側の関数のローカルスコープでnonlocalを使用した代入後:", local_namespace1["text"])
    do_global()
    print("内側の関数のローカルスコープでglobalを使用した代入後:", local_namespace1["text"])
    del local_namespace1

scope_test()
print("グローバルスコープ:", global_namespace["text"])
```

```
内側の関数のローカルスコープでの代入後: test
内側の関数のローカルスコープでnonlocalを使用した代入後: nonlocal
内側の関数のローカルスコープでglobalを使用した代入後: nonlocal
グローバルスコープ: global
```

1つ目の `print()` 関数を実行する前に、 `do_local` 関数内で `text` 変数を書き換えている。 ここでは、 `do_local` 関数のローカルな名前空間の `text` を書き換えているため、 `scope_test` 関数のローカルな名前空間の `text` は変化しない。 そのため、1つ目の `print()` 関数では、 `test` が表示される。

2つ目の `print()` 関数を実行する前に、 `do_nonlocal` 関数内で `text` 変数を書き換えている。 ここでは、 `nonlocal` を使うことで、 `scope_test` 関数のローカルな名前空間の `text` を書き換えているため、 2つ目の `print()` 関数では `nonlocal` が表示される。

3つ目の `print()` 関数を実行する前に、 `do_global` 関数内で `text` 変数を書き換えている。 ここでは、 `global` を使うことで、グローバルな名前空間の `text` を書き換えているため、 `scope_test` 関数のローカルな名前空間の `text` は変化しない。 そのため、3つ目の `print()` 関数では `nonlocal` が表示される。

4つ目の `print()` 関数では、グローバルな名前空間の `text` を参照しているため、 `do_global` 関数内で `text` 変数に代入した値である `global` が表示される。

クラス

```
class ClassName:
    <ステートメント1>
    .
    .
    .
    <ステートメントN>
```

クラスを定義する際の最も単純な形は、上記の通りである。各ステートメントには、任意のステートメントを記述できますが、一般的には関数や変数を定義する。 クラス内で定義した関数や変数は、クラスに紐付いた属性となります。属性の詳細は後ほど説明する。

クラスを定義するとクラスオブジェクトが生成される。クラスはインスタンス化することができ、インスタンス化するとインスタンスオブジェクトを生成できる。 1つのクラスに対して、1つのクラスオブジェクトしか生成されないが、インスタンスオブジェクトは何個でも生成することができる。

まず、クラスオブジェクトについて解説した後、インスタンスオブジェクトについて解説する。

クラスオブジェクト

クラスオブジェクトは、クラスを定義することで生成されるオブジェクトであり、クラス名を介して参照できる。次のサンプルプログラムで、クラスオブジェクトの使い方を見てみよう。

```
class GreetingClass:
    message = "Hello"

    def greet(self, name):
        print(f"{self.message} {name}!")

print(GreetingClass.message)
GreetingClass.message = "Goodbye"
print(GreetingClass.message)
GreetingClass.greet(GreetingClass, "Taro")
```

```
Hello
Goodbye
Goodbye Taro!
```

上記の例では、 `GreetingClass` クラスを定義しており、 `message` と `greet` という属性を持つ `GreetingClass` のクラスオブジェクトが生成されている。 `message` 属性は `"Hello"` という文字列オブジェクトを、 `greet` 属性は関数オブジェクトを格納している。 これらの属性を参照するためには、 `<クラスオブジェクトが格納された変数名>.<属性名>` という記法を使う。 つまり、 `GreetingClass.message` や `GreetingClass.greet` などが該当する。 このようにして、オブジェクトに紐付いた属性を読み取ることを属性参照と呼ぶ。

属性は通常の変数と同様に読み書きができる。読み込み専用（書き換え不可能な）属性も存在するが、上の例では書き換え可能である。 そのため、1つ目の `print()` 関数では `Hello` を表示するが、2つ目の `print()` 関数では `Goodbye` を表示する。

`GreetingClass.greet(GreetingClass, "Taro")` を実行すると、 `greet` 関数の第一引数である `self` に `GreetingClass` のクラスオブジェクトが、 第二引数である `name` に `"Taro"` の文字列オブジェクトが代入される。 `self.message` では `GreetingClass` のクラスオブジェクトの `message` 属性を参照するため、 `"Goodbye"` が得られる。 `f"{self.message} {name}!"` は、 `str(self.message) + " " + str(name) + "!"` と等価であり、 最終的には、 `"Goodbye Taro!"` という文字列が生成されて、画面に表示される。

なお、 `f"<文字列>"` は、文字列の中で、 `{<式>}` を記載すると、 式の実行結果を文字列に変換して文字列中に埋め込まれるため、 容易に文字列や数値などを結合して1つの文字列に変換することができる。

クラスオブジェクトは `()` をつけて関数呼び出しのように記述すると、クラスのインスタンス化ができる。インスタンス化すると、該当クラスのインスタンスオブジェクトを生成することができる。クラスオブジェクトはクラスを定義した際に1つだけ生成され、基本的に2つ以上生成する

ことができないが、クラスをインスタンス化は何度でも行うことができ、インスタンス化するたびに、新しいインスタンスオブジェクトを生成することができる。以下で、複数のインスタンスオブジェクトを生成するサンプルプログラムを見てみよう。

```
class GreetingClass:
    # 中身がからのクラスを使う際は、何もしない pass ステートメントを使う
    pass

g1 = GreetingClass()
g2 = GreetingClass()

print("GreetingClass is GreetingClass:", GreetingClass is GreetingClass)
print("g1 is g2:", g1 is g2)
```

```
GreetingClass is GreetingClass: True
g1 is g2: False
```

上の例では、`GreetingClass` クラスのインスタンス化を2回行い、`g1` と `g2` にインスタンスオブジェクトを代入して、両者のIDが異なることを確認している。

なお、`pass` とは何もしないステートメントである。クラスや関数などを定義する際に、少なくとも1つはステートメントを記載する必要がある。 そのため、何もしないラスや関数を定義する際に、`pass` を使う必要がある。

クラスに `__init__` という名前の特別な関数を定義すると、インスタンス化する際に、追加の処理を実行することができる。また、`__init__` 関数に仮引数を追加すれば、インスタンス化する際に実引数を渡すことができるようになる。以下で、`__init__` 関数のサンプルプログラムを見てみよう。

```
class GreetingClass:
    def __init__(self, name):
        print(f"Hello {name}!")

g1 = GreetingClass("Taro")
g2 = GreetingClass("Jiro")
print("-----")
print("g1 is g2:", g1 is g2)
```

```
Hello Taro!
Hello Jiro!
-----
g1 is g2: False
```

上の例では、`__init__` 関数が2つのパラメータを受け取れるようになっている。一方、インスタンス化する際は、1つの引数のみを渡している。 詳細は次節で説明するが、`__init__` 関数の1つ目の引数には自動的にインスタンスオブジェクトが渡され、`"Taro"` や `"Jiro"` は2つ目の引数の `name` に渡される。 そのため、1回目のインスタンス化では `Hello Taro!` が表示され、2回目のインスタンス化では `Hello Jiro!` が表示される。 インスタンス化のパラメータによらず、インスタンス化するたびに新しいインスタンスオブジェクトが生成されるため、両者のIDは異なる。

インスタンスオブジェクト

本節ではインスタンスオブジェクトの詳細について説明する。以下で、インスタンスオブジェクトごとに異なるデータを持たせることで、二人の人それぞれに向けた挨拶文を生成するサンプルプログラムを示す。

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

    def greet(self, message):
        print(f"{message} {self.name}")

taro = GreetingClass("Taro")
jiro = GreetingClass("Jiro")

print(taro.name)
print(jiro.name)

print("-----")
taro.greet("Hi")
jiro.greet("Hello")
```

```
Taro
Jiro
-----
Hi Taro
Hello Jiro
```

前節で説明したとおり、`__init__` の第一引数には生成されるインスタンスオブジェクトが渡される。`__init__` 関数の中では `self.name` に `name` を代入しているが、`self` はインスタンスオブジェクトであるため、インスタンスオブジェクトに `name` という属性を作り、`name` 引数の値、つまり、`"Taro"` や `"Jiro"` を代入している。

作成したインスタンスオブジェクトは、`taro` 変数と `jiro` 変数に代入している。`taro` 変数のインスタンスオブジェクトの `name` 属性には `"Taro"` が格納されていて、`jiro` 変数のインスタンスオブジェクトの `name` 属性には `"Jiro"` が格納されている。したがって、1つ目の `print()` 関数では `Taro` が、2つ目の `print()` 関数では `Jiro` が表示される。

また、最後に2行ではインスタンスオブジェクトに紐づく `greet` 関数を呼び出しているが、定義時には仮引数が2つ記載されているものの、呼び出し時はパラメータが1つしか渡されていない。`greet` 関数も `__init__` 関数同様に、自動的に第一引数にインスタンスオブジェクトが渡されるためである。そのため、`taro.greet("Hi")` では、`taro` 変数に格納されているインスタンスオブジェクトの `name` が参照され、`jiro.greet("Hello")` では、`jiro` 変数に格納されているインスタンスオブジェクトの `name` が参照される。その結果、`Hi Taro` および `Hello Jiro` が画面に表示される。

インスタンスオブジェクトに紐づく属性のうち、`name` などのデータを持つ属性をデータ属性、`greet` などの関数の属性をメソッドと呼ぶ。なお、メソッドの第一引数はインスタンスオブジェクトが自動的に渡され、他の引数と区別するために慣習として `self` という名前を付ける。ただし、`self` という名前自体に特別な意味はなく、`self` 以外にも任意の変数名を採用することはできる。

上の例では、インスタンスオブジェクトにしか `name` 属性はないが、`greet` 属性については、クラスオブジェクトにもインスタンスオブジェクトにも存在してる。メソッドの詳細は後ほど説明するとして、まずは、データ属性の詳細を理解するために、次のサンプルプログラムを見てみよう。

```
class GreetingClass:
    message = "Hello"

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

    def greet(self):
        print(f"{self.message} {self.name}")

taro = GreetingClass("Taro")
jiro = GreetingClass("Jiro")

taro.greet()
GreetingClass.message = "Hi"
jiro.greet()

print("-----")
print("taro is jiro:", taro is jiro)
print("taro.name is jiro.name:", taro.name is jiro.name)

print("-----")
print("GreetingClass.message is taro.message:", GreetingClass.message is taro.message)
print("GreetingClass.message is jiro.message:", GreetingClass.message is jiro.message)
print("taro.message is jiro.message:", taro.message is jiro.message)
```

```
Hello Taro
Hi Jiro
-----
taro is jiro: False
taro.name is jiro.name: False
-----
GreetingClass.message is taro.message: True
GreetingClass.message is jiro.message: True
taro.message is jiro.message: True
```

既に説明したとおり、`taro` と `jiro` 変数には異なるインスタンスオブジェクトが代入されている。さらに、インスタンスオブジェクトを生成する際に、前者には `"Taro"` という文字列オブジェクトが、後者には `"Jiro"` という文字列オブジェクトが渡されており、それらが、`name` 属性に代入されている。そのため、`taro.name` と `jiro.name` に代入されている文字列オブジェクトも異なる。

一方、`GreetingClass.message` と `taro.message` と `jiro.message` は、全て同じ文字列オブジェクトのが代入されている。`message` 属性に代入している箇所は、2行目と14行目のみである。クラス定義の中、かつ、関数の外で属性に値を代入すると、その属性はクラスオブジェクトに紐づく。一方、`__init__` 関数の中では、`slef.message` に対して値が代入されておらず、インスタンスオブジェクトに紐付いた `message` 属性は存在しない。そのようなケースにおいて、インスタンスオブジェクトから `message` 属性を参照すると、インスタンスオブジェクトの代わりに、インスタンスオブジェクトの生成に使われたクラスオブジェクトの属性を参照することとなる。

つまり、`taro.message` という属性参照では、まず、`taro` 変数に格納されているインスタンスオブジェクトの `message` 属性を探し、それがない場合は、インスタンスオブジェクトに生成に使われた `GreetingClass` クラスオブジェクトの `message` 属性が探されるということになる。そして、既に説明したとおり、クラスオブジェクトは1つのクラスに対して1つしか存在しないオブジェクトである。つまり、`GreetingClass` インスタンスオブジェクトに共通する属性であるとも捉えられる。

なお、クラスオブジェクトに紐づく属性のことをクラス変数、インスタンスオブジェクトに紐づく属性のことをインスタンス変数と呼ぶ。`message` はクラス変数であり、`name` はインスタンス変数である。`message` は `GreetingClass` クラスのクラスオブジェクトおよびインスタンスオブジェクトに共通するデータ属性であるが、`name` は `GreetingClass` クラスの各インスタンスオブジェクトが個別に保有するデータ属性となる。

そのため、14行目で `GreetingClass.message` の中身を書き換えると、`jiro.message` の内容も変化する。なぜなら、クラス変数はクラスオブジェクトおよびインスタンスオブジェクト間で共通するデータであるからである。

続いて、メソッドの詳細を理解するために、次のサンプルプログラムを見てみよう。

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

    def greet(self):
        print(f"Hello {self.name}")

taro = GreetingClass("Taro")
jiro = GreetingClass("Jiro")

GreetingClass.greet(taro)
GreetingClass.greet(jiro)

print("-----")
taro.greet()
jiro.greet()

print("-----")
greetTaro = taro.greet
greetJiro = jiro.greet
greetTaro()
greetJiro()
```

```
Hello Taro
Hello Jiro
-----
Hello Taro
Hello Jiro
-----
Hello Taro
Hello Jiro
```

クラス内で関数を定義すると、クラスオブジェクトに関数オブジェクトが代入された属性が紐づく。例えば、`greet`が該当する属性である。クラス変数と同じ要領で、`GreetingClass.greet`のようにクラスオブジェクトに紐付いた関数オブジェクトを参照することができ、関数呼び出しも可能である。`self`という仮引数を1つ持っているので、`GreetingClass.greet(taro)`のように、パラメータを1つ渡さなければならない。

一方、`taro.greet`のように、インスタンスオブジェクトから関数オブジェクトの属性を参照すると、自動的にメソッドオブジェクトが生成される。そのため、`greet`のような関数オブジェクトの属性をメソッドと呼ぶ。クラス変数と異なり、メソッドはインスタンスオブジェクトに紐づく。

メソッドオブジェクトは、`.`の前に記述されたインスタンスオブジェクトを自動的に第一引数に渡す仕組みを備えた関数オブジェクトのようなオブジェクトであり、第二引数以降のパラメータを渡すことで、元となる関数オブジェクトを呼び出すことができる。例えば、`greet`関数は仮引数が1つだけであり、1つ目のパラメータとしてインスタンスオブジェクトが自動的に渡されるので、`taro.greet()`のように末尾に`()`をつけるだけで、メソッドを呼び出すことができる。以上から、`GreetingClass.greet(taro)`と`taro.greet()`は全く同じ挙動となる。

メソッドオブジェクトは関数オブジェクトと同様に、別の変数などに代入することができる。別の変数に代入したとしても、第一引数にインスタンスオブジェクトを自動的に受け渡す性質は維持される。そのため、`greetTaro = taro.greet`を実行した後に、`greetTaro()`を実行すると、`taro.greet()`と同じ挙動になる。

メソッドはインスタンスオブジェクトに紐づくことから、インスタンス変数に近い概念である。それでは、メソッドのクラス変数版は存在するのだろうか？Pythonではアノテーションという仕組みを使うことで、クラス変数のようなメソッドを定義できる。これをクラスメソッドと呼ぶ。次のサンプルプログラムで、クラスメソッドの定義の仕方と使い方を見てみよう。

```
class MyClass:
    @classmethod
    def class_name(cls):
        print(f"This class is {cls.__name__}")

MyClass.class_name()

mycls = MyClass()
mycls.class_name()
```

```
This class is MyClass
This class is MyClass
```

`class_name` というクラスメソッドは、1つ目の仮引数 `cls` の `__name__` 属性を表示する処理を行う。 クラスオブジェクトが生成されると、自動的に `__name__` 属性が定義され、クラスの名前の文字列が代入される。 メソッドでは第一引数にインスタンスオブジェクトが自動的に渡されたが、クラスメソッドでは第一引数にクラスオブジェクトが自動的に渡される。 `MyClass.class_name()` を実行すると、 `class_name` 関数オブジェクトの第一引数に `MyClass` クラスオブジェクトが渡され、 その結果、 `This class is MyClass` という文字列が表示される。

なお、クラスメソッドはクラス変数に似た性質を持っており、インスタンスオブジェクトからも呼び出すことができる。 その場合でも、インスタンスオブジェクトが第一引数に渡されるのではなく、インスタンスオブジェクトの生成時に使われたクラスオブジェクトが第一引数に渡される。 そのため、上のプログラムにおいて、 `MyClass.class_name()` と `mycls.class_name()` の挙動は同じである。

先ほどの例における `self.message = new_message` では変数を定義する文であったことから、新たにインスタンス変数が作成されていた。 一方で上の例における `self.messages.append(new_message)` では変数を定義する文ではなく、書き換える文である。 そのためこの場合にはクラス変数が書き換えられ、全てのインスタンスに変更の影響が出る。

```
# プライベート変数(命名の慣習)
class GreetingClass5:
    __message = "Hello"

    def check_message(self):
        print(self.__message)

my_class = GreetingClass5()
my_class.check_message()
# print(GreetingClass5.__message) # 直接アクセスするとエラーになる
print(GreetingClass5._GreetingClass5__message) # 迂回することはできてしまう
```

```
Hello
Hello
```

多くのプログラミング言語では、属性の公開範囲を指定することができ、特定の方法でしか属性にアクセスできないように制限を加えることができる。 このような仕組みは、オブジェクト指向プログラミングにおけるカプセル化を実現する上で必要となる。 カプセル化とは、データと処理をクラスでまとめた際に、外部からデータを隠すことである。 カプセル化を行うことで、クラスの設計者が予期しない内容にデータを書き換えられることを防いだり、 クラスの利用者がクラスの実装の詳細を意識せずに利用できるようにしたりすることができる。

Pythonには、属性の公開範囲を指定する仕組みはないが、外部に公開したくないデータ属性やメソッドの名前の先頭に `_` を付ける慣習がある。 ただし、あくまで `_` は目印でしかなく、そのようなデータ属性やメソッドにアクセスするプログラムを記述することはできてしまう。 なお、 `__` を付けると、アクセスしにくい属性の名前に自動変換することができるが、本章では詳細を割愛する。

オブジェクト指向プログラミング

クラスを活用するとオブジェクト指向プログラミングという考え方に基づいてプログラムを記述できる。オブジェクト指向プログラミングとは、プログラムをオブジェクトという概念に基づいて整理して部品化していく考え方である。オブジェクトとは、データと処理の組み合わせを指して

おり、Pythonでにおけるデータ属性とメソッドの組み合わせだと考えれば良い。

オブジェクト指向プログラミングが普及する以前は、手続型プログラミングという考え方が一般的であり、データと処理は切り離されて整理されていた。しかし、現実世界では、何らかのものに対して、ものが持つ特徴や性質などのデータに基づいて、処理をすると考えることが一般的である。例えば、ある人Aさんは、体重と身長という数値を持っていて、それらの数値データに基づいてBMI値を計算することができる、と整理することは自然である。なお、BMI値とは `体重(kg) / (身長(m) * 身長(m))` で計算される値であり、肥満度を表す指標である。以下で、オブジェクト指向プログラミングの考え方を使わずに、BMI値を計算するプログラムを示す。

```
def calculate_bmi(person):
    return person['weight'] / person['height'] ** 2

taro = {
    'weight': 54.6,
    'height': 1.7
}
bmi = calculate_bmi(taro)
print(bmi)
```

```
18.892733564013845
```

`taro` という辞書オブジェクトを `calculate_bmi` 関数に渡すことで、BMI値を計算することができる。 ただし、 `taro` 辞書オブジェクトと、 `calculate_bmi` 関数に繋がりはなく、データと処理が完全に分離している。

オブジェクト指向プログラミングの考え方に基づいて、上のプログラムを書き直す。以下の書き直したサンプルプログラムを見てみよう。

```
class Person:
    def __init__(self, weight, height):
        self.weight = weight
        self.height = height

    def calculate_bmi(self):
        return self.weight / self.height ** 2

taro = Person(54.6, 1.7)
bmi = taro.calculate_bmi()
print(bmi)
```

```
18.892733564013845
```

上のコードでは、 `Person` クラスにおいて、 `weight` と `height` というインスタンス変数と、 `calculate_bmi` というメソッドが定義されている。 `Person` クラスのインスタンスオブジェクトを生成する際に、 `weight` と `height` のデータを引数で渡す必要がある。 一度渡してしまえば、 `Person` クラスのインスタンスオブジェクトにデータが記録され、引数を渡さずに `calculate_bmi()` を呼び出すだけで、BMI値を計算できる。 このように、データと処理を組み合わせてプログラムを整理したほうが、人間が現実世界を理解するときの考え方に近く、プログラムを理解しやすくなる。

ひとつ前のプログラムでは、「BMI計算機に太郎君のデータを渡してBMIを計算する」という考え方でプログラムが構成されていたが、 今回の例では、「太郎君のBMI値を計算する」というという考え方でプログラムが構成されていて、より分かりやすくなっている。

継承

継承とは、あるクラスの機能を引き継いだ新しいクラスを定義することである。引き継がれるクラスのことを親クラス、引き継いだ新しいクラスのことを子クラスと呼ぶ。 クラスを定義する際に継承元の親クラスを指定することができ、 `class <クラス名>(<継承する親クラス名>):` と記述すれば良い。 以下で、 `Animal` クラスと、 `Animal` クラスを継承した `Dog` と `Lion` クラスのサンプルプログラムを見てみよう。

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

    def run(self):
        print(f"{self.name}は走りだした")

class Dog(Animal):
    def bark(self):
        print(f"{self.name}はワンワンと吠えた")

class Lion(Animal):
    def __init__(self):
        super().__init__("ライオン")

    def bark(self):
        print(f"{self.name}はガオーと吠えた")

animal1 = Animal("羊")
animal1.run()

dog1 = Dog("犬")
dog1.run()
dog1.bark()

lion1 = Lion()
lion1.run()
lion1.bark()
```

```
羊は走りだした
犬は走りだした
犬はワンワンと吠えた
ライオンは走りだした
ライオンはガオーと吠えた
```

継承すると子クラスは親クラスの属性を引き継ぐことができる。そのため、`Dog` および `Lion` のインスタンスオブジェクトは、`name` 属性と `run` メソッドを持つ。また、`Dog` クラスと `Lion` クラスには、`bark` メソッドが定義されており、`Dog` および `Lion` のインスタンスオブジェクトから `bark` メソッドを呼び出すことができる。なお、`Animal` クラスには `bark` メソッドが定義されていないため、`Animal` のインスタンスオブジェクトから `bark` メソッドを呼び出すことはできない。

`Dog` クラスは `Animal` の `__init__` メソッドも引き継いでいるため、`Dog` のインスタンス化をする際に `name` 引数に対するパラメータを記載しないといけない。一方、`Lion` は独自の `__init__` メソッドを定義している。子クラスが親クラスと同じ名前の属性を定義した場合は、子クラスの定義が優先的に利用される。そのため、`Lion` をインスタンス化する際に呼び出される `__init__` メソッドは、`Lion` クラスの中で定義された `__init__` メソッドである。`Lion` クラスの `__init__` メソッドは `self` 以外に引数を受け取らないため、インスタンス化する際はパラメータを渡してはいけない。

`Lion` クラスの `__init__` メソッドの内容は、`super().__init__("ライオン")` になっている。`super()` とは親クラスのメソッドにアクセスするための特別なオブジェクトを返す関数である。そのため、`super().__init__("ライオン")` は、`"ライオン"` という文字列オブジェクトを渡して、`Lion` クラスではなく `Animal` クラスの `__init__` メソッドを呼び出している。

Pythonには、2つのクラス間に継承関係があるかどうかを判定する `issubclass()` 関数と、指定したインスタンスオブジェクトが指定したクラスのインスタンスであるかどうかを判定する `isinstance()` 関数が用意されている。なお、インスタンスオブジェクトのクラスが継承をしていて、親クラスやさらにその親（先祖）クラスがある場合、該当インスタンスオブジェクトは、親クラスや先祖クラスのインスタンスでもある。次のプログラムで `issubclass()` 関数と `isinstance()` 関数のサンプルプログラムを示す。

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

    def run(self):
        print(f"{self.name}は走りだした")

class Dog(Animal):
    def bark(self):
        print(f"{self.name}はワンワンと吠えた")

class Lion(Animal):
    def __init__(self):
        super().__init__("ライオン")

    def bark(self):
        print(f"{self.name}はガオーと吠えた")

class ShibaInu(Dog):
    pass

animal1 = Animal("羊")
dog1 = Dog("犬")
lion1 = Lion()
shiba1 = ShibaInu("柴犬")

print(f"DogはAnimalの子クラスもしくは子孫クラス？", issubclass(Dog, Animal))
print(f"ShibaInuはAnimalの子クラスもしくは子孫クラス？", issubclass(ShibaInu, Animal))
print(f"ShibaInuはDogの子クラスもしくは子孫クラス？", issubclass(ShibaInu, Dog))

print("-----")
print(f"AnimalはShibaInuの子クラスもしくは子孫クラス？", issubclass(Animal, ShibaInu))
print(f"AnimalはDogの子クラスもしくは子孫クラス？", issubclass(Animal, Dog))
print(f"DogはShibaInuの子クラスもしくは子孫クラス？", issubclass(Dog, ShibaInu))

print("-----")
print(f"LionはAnimalの子クラスもしくは子孫クラス？", issubclass(Lion, Animal))
print(f"LionはDogの子クラスもしくは子孫クラス？", issubclass(Lion, Dog))
print(f"LionはShibaInuの子クラスもしくは子孫クラス？", issubclass(Lion, ShibaInu))
```

```
DogはAnimalの子クラス？ True
ShibaInuはAnimalの子クラス？ True
ShibaInuはDogの子クラス？ True
-----
AnimalはShibaInuの子クラス？ False
AnimalはDogの子クラス？ False
DogはShibaInuの子クラス？ False
-----
LionはAnimalの子クラス？ True
LionはDogの子クラス？ False
LionはShibaInuの子クラス？ False
```

`ShibaInu` は `Dog` の子クラスであり、 `Dog` は `Animal` の子クラスである。したがって、 `ShibaInu` は `Animal` の子孫クラスである。 `Lion` は `Animal` の子クラスであるが、 `Lion` は `Dog` や `ShibaInu` と継承関係はないということが読み取れる。

継承を使うと、親クラスと共通するメソッドなどを定義する手間を省くことができる。 プログラムの分量はできるだけ少ないほうが、開発する際も修正する際も手間を減らすことができる。 継承によって、重複部分を減らすことで、プログラムの分量を減らし、開発や修正の手間を減らせる。 また、プログラムを読む人にとっても、継承は現実世界にも類似した考え方があるため、プログラムを理解しやすくなる。

問題1

問題文

プライベートなインスタンス変数として数字のリスト `numbers` を持っているクラス `number_list` を作成し、

- リスト `numbers` に数字を追加するインスタンスメソッド `add_number`
- リスト `numbers` の要素を昇順にソートするインスタンスメソッド `sort_numbers`
- リスト `numbers` の要素を表示するインスタンスメソッド `print_numbers`

を定義せよ。

解答の雛形

```
class number_list:
    def __init__(self):
        self.__numbers = []
    # ここに解答を入力

list1 = number_list()
list2 = number_list()

list1.add_number(1)
list1.add_number(4)
list1.add_number(3)

list2.add_number(1)
list2.add_number(5)
list2.add_number(2)
list2.sort_numbers()

list1.print_numbers()
list2.print_numbers()
```

問題2

問題文

`__str__` の関数を実装し、インスタンスを文字列として出力する際にはクラス名が出力されるようにせよ。

解答の雛形

```
class PrintClass:
  # ここに解答を入力

obj = PrintClass()
print(obj)
```

問題3

問題文

動作確認部分が動くようなクラスを定義せよ。 なお `measure` メソッドは、1度目の出力では `170 cm` と、2度目の出力では `54.6 cm` と返すとする。

解答の雛形

```
# ここに解答を入力

height = MeasureClass("cm")
print(height.measure(170))

weight = MeasureClass("kg")
print(weight.measure(54.6))
```

問題4

問題文

`GreetingBaseClass` というクラスを継承する `HelloClass` という名前のクラスを作成し、動作確認コードが正常に動くようにせよ。 また `HelloClass` では `message` メソッドをオーバーライドし、 `Hello` と出力するようにせよ。

解答の雛形

```
class GreetingBaseClass:
  def greet(self):
    print("-" * 10)
    print(self.message())
    print("-" * 10)

  def message(self):
    pass

# ここに解答を入力

hello = HelloClass()
hello.greet()
```

問題5

問題文

東京都や新宿区など、以下の6つの場所をクラス化せよ。 このとき、下記の箇条書きに即した継承関係にせよ。 今回はトップダウン型の継承、つまりは東京都は日本を継承するといった順序にせよ。

- 日本
  - 東京都
    - 新宿区
    - 渋谷区
  - 大阪府
    - 堺市

中身の存在しないクラスの作成方法は、 `Japan` クラスを参考してほしい。 正しく実装されている場合は、 `issubclass` を用いた動作確認コードが全て `True` を返す。

解答の雛形

```
class Japan:
  pass

# ここに解答を入力

print(issubclass(Tokyo, Japan) == True)
print(issubclass(Shinjuku, Japan) == True)
print(issubclass(Shibuya, Japan) == True)
print(issubclass(Osaka, Japan) == True)
print(issubclass(Sakai, Japan) == True)

print(issubclass(Japan, Tokyo) == False)
print(issubclass(Shinjuku, Tokyo) == True)
print(issubclass(Shibuya, Tokyo) == True)
print(issubclass(Osaka, Tokyo) == False)
print(issubclass(Sakai, Tokyo) == False)

print(issubclass(Japan, Osaka) == False)
print(issubclass(Tokyo, Osaka) == False)
print(issubclass(Shinjuku, Osaka) == False)
print(issubclass(Shibuya, Osaka) == False)
print(issubclass(Sakai, Osaka) == True)
```

問題6

問題文

先ほど同様に、東京都や新宿区など、以下の6つの場所を継承関係付きでクラス化せよ。 このとき、下記の箇条書きに即した継承関係にせよ。 今回はボトムアップ型の継承、つまりは日本は東京都を継承するといった順序にせよ。

- 日本
  - 東京都
    - 新宿区
    - 渋谷区
  - 大阪府
    - 堺市

解答の雛形

```
# ここに解答を入力

print(issubclass(Shibuya, Shinjuku) == False)
print(issubclass(Tokyo, Shinjuku) == True)
print(issubclass(Sakai, Shinjuku) == False)
print(issubclass(Osaka, Shinjuku) == False)
print(issubclass(Japan, Shinjuku) == True)

print(issubclass(Shinjuku, Tokyo) == False)
print(issubclass(Shibuya, Tokyo) == False)
print(issubclass(Sakai, Tokyo) == False)
print(issubclass(Osaka, Tokyo) == False)
print(issubclass(Japan, Tokyo) == True)

print(issubclass(Shinjuku, Japan) == False)
print(issubclass(Shibuya, Japan) == False)
print(issubclass(Tokyo, Japan) == False)
print(issubclass(Sakai, Japan) == False)
print(issubclass(Osaka, Japan) == False)
```

7章: モジュールとパッケージ

モジュール

ソフトウェアは大規模化および複雑化の一途を辿っており、ソフトウェアに必要なプログラムをゼロから全て作成することは極めて困難である。したがって、オープンソースソフトウェアのライブラリなど、第三者が再利用可能な形で公開されている部品を活用して、ソフトウェアを開発することが一般的である。 Pythonでは、プログラムを部品化するための仕組みとして、モジュールとパッケージという仕組みが提供されている。まず、モジュールについて詳細を説明する。

1つのPythonファイルは、1つのモジュールになる。Pythonプログラムは `import` 文を使うことで、別のモジュールを読み込み、モジュールの中で定義されている関数や変数などを利用することができる。この仕組を使うことで、1つのプログラムを複数のPythonファイルに分割することができる。基本的に、長大なファイルは内容を理解したり変更したりすることが難しくなるので、適切な粒度で複数のモジュール（つまり、複数のファイル）に分割することが必要である。以下で、与えられた数値よりも小さい数までのフィボナッチ数列を表示する `fib()` 関数と、与えられた数値より小さい数までのフィボナッチ数列のリストを返す `fib2()` 関数、10番目のフィボナッチ数が代入された `tenth_fib` 変数を定義しているサンプルプログラムを示す。

```
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result

tenth_fib = 34
```

上記のサンプルプログラムを `fibo.py` として保存したとしよう。その上で、上記の `fib()` 関数と `fib2()` 関数を呼び出したり、`tenth_fib` 変数を参照するサンプルプログラムを以下で示す。

```
import fibo

fibo.fib(1000)
print(fibo.fib2(100))
print(fibo.tenth_fib)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
34
```

`import fibo` というステートメントを実行すると、 `fibo.py` を読み込むことができる。すると、 `fibo` 変数には `fibo.py` のグローバルな名前空間で定義された関数や変数が属性となっているオブジェクトが代入される。 `fibo.py` のグローバルな名前空間には、 `fib()` 関数と `fib2()` 関数と `tenth_fib` 変数が定義されている。そのため、 `fibo.fib` や `fibo.fib2` 、 `fibo.tenth_fib` と記述すれば関数や変数を参照することができる。

モジュールのオブジェクトは自動的に `__name__` 属性が与えられる。 `__name__` 属性には、モジュールの名前が入っています。以下で、`__name__` 属性の中身を表示するサンプルプログラムを示す。

```
import fibo

print(fibo.__name__)
```

```
fibo
```

`fibo.__name__` には `"fibo"` という文字列が入っていることを確認できた。モジュールのオブジェクトを別の変数に代入したり、モジュールのオブジェクトが持っている関数オブジェクトを別の変数に代入したりすることもできる。以下でモジュールのオブジェクトとモジュールのオブジェクトが持っている関数を変数に代入するサンプルプログラムを示す。

```
import fibo

f = fibo
print(f.__name__)

fib = f.fib
fib(10)
```

```
fibo
0 1 1 2 3 5 8
```

他のオブジェクトと同様に変数に代入できることを確認できた。なお、 `import` ステートメントはプログラムの一番上に記述することが慣習となっているが、途中に置いても動作はする。

`import` 文において `as` を使うと、モジュールの名前の変数にモジュールのオブジェクトを代入する代わりに、別の名前の変数に代入することができる。これを利用すると、自由な名前でモジュールを参照できるようになる。以下で、 `as` を使ったサンプルプログラムを示す。

```
import fibo as fibonacci
fibonacci.fib(500)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Pythonでは、慣習として特定のモジュールを特定の別名で `import` することがある。例えば、 `numpy` というモジュールを `import` する際は、 `np` という別名で取り込むことが一般的である。

`import` 文の書き方を変えて、 `from <モジュール名> import <モジュール内の関数や変数>` と記述すると、モジュール名の変数の中にモジュールのオブジェクトを代入する代わりに、モジュール内の関数や変数を直接取り込んで、それぞれの変数を定義することができる。これを活用すると、上の例では `fibo.fib()` の代わりに、 `fib()` と記述できるようになる。以下で、 `fibo` モジュールから `fib()` 関数と `tenth_fib` 変数を直接取り込むサンプルプログラムを示す。

```
from fibo import fib, tenth_fib

fib(500)
print(tenth_fib)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
34
```

`fibo` モジュールから `fib()` 関数と `tenth_fib` 変数を取り込み、 `fib.` という記述をせずに、 `fib()` 関数と `tenth_fib` 変数を使うことができた。

アスタリスク(`*`)を使うと、モジュール内のすべての関数や定義を直接取り込むことができる。以下で、アスタリスクを使ったサンプルプログラムを示す。

```
from fibo import *

fib(500)
print(tenth_fib)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
34
```

同様に、 `fib.` という記述をせずに、 `fib()` 関数と `tenth_fib` 変数を使うことができた。しかし、一般に、アスタリスクの利用は好まれない。なぜなら、どのような関数や変数が取り込まれるのかが分かりづらく、場合によっては、定義済みの関数や変数を意図せず上書きしてしまう可能性があるためである。

モジュール内には関数や変数の定義だけでなく、これまで紹介してきたような一般的なPythonプログラムと同様に自由にステートメントを記述することができる。その場合、 `import` した際に、記述されている全てのステートメントが実行される。

モジュールが実行されるタイミングは、モジュールが他のモジュールから読み込まれるタイミングに加えて、モジュールが `python` コマンドによって直接実行されるタイミングも該当する。どちらのタイミングであっても、実行したいステートメントもあれば、直接実行されたときだけ実行したいステートメントもある。モジュールが直接実行されたときは、 `__name__` 変数に `__main__` という文字列が代入されるため、 `__name__` の値が `__main__` と等しい場合に、直接実行されたと判断できる。以下で、直接実行されたときだけ実行されるステートメントを含むサンプルプログラムを示す。

```
print("モジュールが読み込まれました。")

if __name__ == "__main__":
    print("モジュールが直接実行されました。")
```

```
モジュールが読み込まれました。
モジュールが直接実行されました。
```

上記のサンプルプログラムを直接実行すると、 `モジュールが読み込まれました。` と `モジュールが直接実行されました。` の両方のメッセージが表示される。 一方、別のモジュールから読み込まれる場合は、 `モジュールが読み込まれました。` というメッセージのみが表示される。なお、モジュールは初めてアクセスするときに一度だけ実行されるため、 `import` 文を複数記述したり、複数のモジュールから同じモジュールを参照しても、プログラム全体では1度しか実行されないことに注意しよう。

Pythonは標準ライブラリという名前で様々な組み込みのモジュールを提供しており、公式サイトの「Python 標準ライブラリ」 https://docs.python.org/ja/3/library/index.html から各モジュールの詳細を閲覧できる。以下で、標準モジュールの一つである `sys` モジュールを使ったサンプルプログラムを示す。なお、 `sys` モジュールの詳細は https://docs.python.org/ja/3/library/sys.html で閲覧できる。

```
import sys

sys.stdout.write("Hello World")
print("Hello World")
```

```
Hello WorldHello World
```

`sys` モジュールは `stdout` という標準出力のオブジェクトを提供しており、 `stdout` のオブジェクトの `write()` メソッドを呼び出すことで、標準出力に文字列を出力できる。第9章で詳細を説明するが、標準出力に出力した文字列は画面に表示される。 `print()` 関数も標準出力に文字列を出力する命令であり、 `sys.stdout.write()` と `print()` は同じような用途で利用できる。

パッケージ

パッケージは階層構造を用いてモジュールの集合を表現するための仕組みである。パッケージを利用すると、 `my_math.fibonacci` などのように、 `.` 区切りで階層構造を作ることができる。これは、 `my_math` パッケージの `fibonacci` モジュールを指す。例えば、パッケージによる階層構造を作ることで、数学パッケージでは、数列サブパッケージのフィボナッチ数列モジュールと、数列サブパッケージのトリボナッチ数列モジュールと、計算サブパッケージの最大公約数モジュールをひとまとめにして提供できる。実際に、今例に出した数学パッケージのフォルダ構成を見てみよう。なお、トリボナッチ数列はフィボナッチ数列に類似した数列であるが、トリボナッチ数列とは何かは重要ではないため、ここでは説明を割愛する。

- `my_math` ディレクトリ（数学パッケージ）
  - `calculation` ディレクトリ（計算サブパッケージ）
    - `gcd.py` ファイル（最大公約数モジュール）
    - `__init__.py` ファイル（ `calculation` ディレクトリがパッケージであることを示すためのファイル）
  - `sequence` ディレクトリ（数列サブパッケージ）
    - `fibo.py` ファイル（フィボナッチ数列モジュール）
    - `tribo.py` ファイル（トリボナッチ数列モジュール）
    - `__init__.py` ファイル（ `sequence` ディレクトリがパッケージであることを示すためのファイル）
  - `__init__.py` ファイル（ `my_math` ディレクトリがパッケージであることを示すためのファイル）

`my_math` と `calculation` 、 `sequence` はディレクトリであり、それ以外は全てファイルである。ディレクトリの階層構造が、そのままパッケージの階層構造となる。例えば、 `gcd` モジュールにアクセスするためには、 `my_math.calculation.gcd` と記述しなければならない。

`__init__.py` はディレクトリがパッケージであることを示すための特別なファイルである。このファイルもPythonプログラムであるため、任意のステートメントを記述することができ、パッケージに初めてアクセスするときに一度だけ実行される。なお、 `__init__.py` ファイルの中に何も記述しなくても問題ないが、 `__init__.py` ファイルを作成しないと、Pythonがディレクトリをパッケージとして認識してくれなくなるので、作成し忘れないように注意しよう。

それでは、以下で、数学パッケージ内の最大公約数モジュールとフィボナッチ数列モジュール、トリボナッチ数列モジュールを使用するサンプルプログラムを見てみよう。

```
import my_math.calculation.gcd
import my_math.sequence.fibo
import my_math.sequence.tribo

print(my_math.calculation.gcd.gcd(10, 5))
print(my_math.sequence.fibo.fib(10))
print(my_math.sequence.tribo.tri(10))
```

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

1つ目の `print()` 関数では10と5の最大公約数を、2つ目の `print()` 関数では10以下のフィボナッチ数列を、3つ目の `print()` 関数では10以下のトリボナッチ数列を表示している。最大公約数モジュールは `my_math.calculation.gcd` でアクセスでき、モジュール内に `gcd` 関数があるので、 `my_math.calculation.gcd.gcd()` で関数を呼び出すことができる。

パッケージにすることで様々なモジュールをひとまとめにでき、また、類似するモジュールをサブパッケージなどでグルーピングができて便利である。しかし、上のサンプルプログラムでは、モジュールを参照するの記述量が多くなり、プログラムを書くときも読むときも時間がかかってしまう。そこで、モジュールで説明した `from` が役に立つ。以下で、 `from` を使ったサンプルプログラムを見てみよう。

```
from my_math.calculation.gcd import gcd
from my_math.sequence import fibo
# from my_math import sequence

print(gcd(10, 5))
print(fibo.fib(10))
# print(sequence.tribo.tri(10))
```

```
5
[0, 1, 1, 2, 3, 5, 8]
```

`from` を使うことで、モジュールのオブジェクトやモジュール内の関数オブジェクトなどを変数に代入することができる。 上の例では、 `my_math.calculation.gcd` モジュールの `gcd` 関数オブジェクトを `gcd` 変数に代入して、 また、 `my_math.sequence` パッケージの `fibo` モジュールのオブジェクトを `fibo` 変数に代入している。 なお、パッケージはオブジェクトではないため、 `my_math` パッケージの `sequence` サブパッケージを変数に代入することはできないことに注意しよう。

なお、パッケージ内のモジュール間で `import` をする場合は、特別な記法を使うことができる。例えば、 `fibo.py` から `tribo` モジュールを読み込みたい場合は、 `import my_math.sequence.tribo` と書いても `import .tribo` と書いても良い。 `.` の意味は同じディレクトリ内のモジュールであることを指す。もしも、 `my_math` パッケージ内にある `basic` モジュールが存在していて、 `fibo.py` から `basic` モジュールを読み込みたい場合は、 `import my_math.basic` と書いても `import ..basic` と書いても良い。 `..` の意味は親ディレクトリ内のモジュールであることを指す。親の親の場合は `...` のように、ドット(`.`)の数を増やすことで先祖をたどることができる。

標準ライブラリ

既に説明したとおり、Pythonは標準ライブラリを介して様々なモジュールを提供している。本節では、いくつかのモジュールを取り上げて説明する。

まず、 `math` モジュールを紹介する。 `math` モジュールは代表的な数学の関数を提供している。以下で、141と252の最大公約数と、4の階乗を表示するサンプルプログラムを紹介する。

```
import math

print(math.gcd(141, 252))
print(math.factorial(4))
```

```
3
24
```

その他の関数については、 https://docs.python.org/ja/3/library/math.html を参照すること。

続いて、 `datetime` モジュールを紹介する。 `datetime` モジュールは時間や日付に関する処理を行う関数を提供している。以下で、現在時刻と現在の日付を表示するサンプルプログラムを紹介する。

```
from datetime import datetime
from datetime import date

print(datetime.now())
print(date.today())
```

```
2022-02-17 00:10:54.671939
2022-02-17
```

その他の関数については、 https://docs.python.org/ja/3/library/datetime.html を参照すること。

`math` や `datetime` 以外にも様々なモジュールが提供されている。 https://docs.python.org/ja/3/library/index.html に一覧が記載されているので、一度は目を通しておくと良いだろう。各モジュールの細かい使い方を覚える必要はなく、必要になるたびに公式サイトのドキュメントを閲覧して、随時、使い方を学んでいくことが重要だ。

問題1

問題文

mathモジュールを使って、入力された `x` に対して、 `sin(x)` 、 `cos(x)` 、 `tan(x)` をそれぞれそれぞれ求めるプログラムを作成せよ。 出力する際は、小数点以下第3位までを出力せよ。 小数点以下第3位までを出力する方法は、例えば以下の方法がある。

```
a = 3.141592
print(f'{a:.3f}')
```

制約

- $0 \leq x \leq 6.28$
- tan(x)が定義されない `x` は与えられない

入力

入力は次の形式で与えられる。

```
x
```

出力

`sin(x)` 、 `cos(x)` 、 `tan(x)` を小数点以下第3位まで出力せよ。

入力例1

```
0
```

出力例1

```
0.00
1.00
0.00
```

入力例2

```
0.785
```

出力例2

```
0.707
0.707
0.999
```

解答の雛形

```python
import math

x = float(input())
# ここに解答を入力
```

問題2

問題文

mathモジュールを使って、与えられた水素イオン濃度h_plusをもとにpHを求めるプログラムを作成せよ。ここで、pHは$-log_{10}h_plus$と計算することができる。 出力する際は、小数点以下第3位までを出力せよ。 小数点以下第3位までを出力する方法は、例えば以下の方法がある。

```python
a = 3.141592
print(f'{a:.3f}')
```

制約

- $0 < h\_plus \leq 1$

入力

入力は次の形式で与えられる。

```
h_plus
```

出力

pHを小数点以下第3位まで出力せよ。

入力例1

```
0.1
```

出力例1

```
1.000
```

入力例2

```
0.0005
```

出力例2

```
3.301
```

解答の雛形

```python
import math

h_plus = float(input())
# ここに解答を入力
```

問題3

問題文

datetimeモジュールを使って、入力された日付Xから日付Yまで、何日が経過したか表示するプログラムを作成せよ。

制約

- 日付X(year_x年 month_x月 day_x日)は実在する日付
- 日付Xは、2000年1月1日から2020年12月31日までのいずれか
- 日付Yも、日付Xと同様の制約
- 日付Yは日付Xより後の日付

入力

入力は次の形式で与えられる。

```
year_x month_x day_x
year_y month_y day_y
```

出力

pHを小数点以下第3位まで出力せよ。

入力例1

```
2000 1 1
2000 1 3
```

出力例1

```
2
```

入力例2

```
2000 1 1
2020 12 31
```

出力例2

```
7670
```

解答の雛形

```
import datetime

year_x, month_x, day_x = map(int, input().split())
year_y, month_y, day_y = map(int, input().split())
# ここに解答を入力
```

問題4

問題文

datetimeモジュールを使って、入力された日付が何曜日かを出力するプログラムを作成せよ。

制約

- 入力される日付(year年 month月 day日)は実在する日付
- 入力される日付は、2000年1月1日から2020年12月31日までのいずれか

入力

入力は次の形式で与えられる。

```
year month day
```

出力

入力された日付の曜日を出力。 曜日は以下のいずれかを出力すること。

```
'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'
```

入力例1

```
2000 1 1
```

出力例1

```
Sat
```

入力例2

```
2020 12 31
```

出力例2

```
Thu
```

解答の雛形

```
import datetime
weekday_en = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun')

year, month, day = map(int, input().split())
# ここに解答を入力
```

問題5

問題文

datetimeモジュールを使って、入力された日付の75日後の日付を出力するプログラムを作成せよ。

制約

- 入力される日付(year年 month月 day日)は実在する日付
- 入力される日付は、2000年1月1日から2020年12月31日までのいずれか

入力

入力は次の形式で与えられる。

```
year month day
```

出力

問題文に即した日付を、年月日の順に空白区切りで出力せよ。

入力例1

```
2000 1 1
```

出力例1

```
2000 3 16
```

入力例2

```
2020 9 1
```

出力例2

```
2020 11 15
```

解答の雛形

```
import datetime

year, month, day = map(int, input().split())
# ここに解答を入力
```

問題6

問題文

statisticsモジュールを使って、与えられた配列(`A`)の中央値(median)と最頻値(mode)を出力するプログラムを作成せよ。

制約

- $1 \leq len(A) \leq 100$
- $0 \leq A_i \leq 100$
- 最頻値は1つに定まる

入力

入力は次の形式で与えられる。

$A_0 A_1 ...$

出力

配列の中央値と最頻値を改行区切りで出力せよ。

入力例1

```
1 2 3 3 4 5
```

出力例1

```
3.0
3
```

入力例2

```
3 1 4 1 5 9 2 6
```

出力例2

```
3.5
1
```

解答の雛形

```python
import statistics

l = list(map(int, input().split()))
# ここに解答を入力
```

8章: 入出力

ファイルの入出力

本節ではPythonでのファイルの入出力の方法を紹介する。

ファイルとは、コンピュータ上でデータを管理する際に使用するデータのまとまりを表す概念である。例えば、パソコン上で作成したレポートや資料は文字列などのデータがまとまったファイルである。ファイルに対して文字などを書き込むことは、プログラムに値を与えるのと同じことであり、ファイルの入力と呼ぶ。ファイルから文字を読み込むことは、プログラムから出力を得るのと同じことを意味するため、ファイルの出力と呼ぶ。ファイルの入出力を行う際に必要になるのが、ファイルを開くこととファイルを閉じることである。ファイルを開くことで、入出力を行う事ができ、閉じることでファイルに対しての操作を正常に終えることができる。

Pythonでは、ファイルの入出力操作はopen()関数を用いてファイルオブジェクトを取得し、ファイルオブジェクトを介して行う。open()関数は、引数に指定されたファイルに対応するファイルオブジェクトを返す関数である。ファイルオブジェクトは、下位のリソースへのファイル指向API (read() や write() メソッドを持つもの) を公開しているオブジェクトで、作成された手段によって、実際のディスク上のファイルや、その他のタイプのストレージや通信デバイス (例えば、標準入出力、インメモリバッファ、ソケット、パイプ、等) へのアクセスを媒介できるオブジェクトを表す。

それでは、早速、open()関数を使ってみよう。以下のサンプルプログラムでは、書き込み用にファイルを開いて閉じる操作を行っている。

```
file = open("テスト.txt", "w")
file.close()
```

`テスト.txt` は今回開きたいファイルの名前を表す。この部分は自分が付けたいファイルの名前を指定でき、相対パスまたは絶対パスを指定する。今回は例として `テスト.txt` というカレントディレクトリにあるファイルの名前を相対パスで指定したものを使用する。
`w` はファイルのモードと呼ばれるopen()関数の引数で、ファイルに対してどのような操作を行うかを指定する。今回は例として `"テスト.txt"` というファイルに対して書き込むためのモードで開いた。書き込む為のモードの詳しい説明は、読み込むためのモードの説明の後に説明する。
`file` にはopen()関数の戻り値でファイルオブジェクトが代入される。 `file` は変数なので変数名は `f` でも `h` でも何にしても文法上問題ないが、慣習的に `f` や `file` という変数名を付けることが多いため、ここでは `file` とした。
最後にファイルの入出力操作で注意するべきこととして、ファイルはファイルを開いた後は閉じないとリソースを解放せず、思わぬバグの原因になりやすいので、 `file.close()` で開いたファイルを閉じる。実際にどのようなバグが起こるのかについての例や、close()関数についてはこの章のクリーンアップ処理の節でもう少し詳しく説明する。

open()関数にはモードと呼ばれる引数が存在する。この引数は、ファイルに対してどのような操作を行うのかを指定する引数であり、Pythonでは大きく分けて書き込むためのモードと、読み込むためのモードの2つが存在する。これら二つのモードについて説明していく。

<!-- モードに関しての参照：https://docs.python.org/ja/3/library/functions.html#open -->

まずは、ファイルを読み込むためのモードについて説明する。
読み込むためのモードで開くと、read()関数などを用いてファイルの出力、つまり読み込みを行うことができるが、後述するwrite()関数を用いてファイルを書き込むことはできないことに注意しよう。

読み込みのモードで開いたファイルはいくつかの方法で読み込むことができる。その中で、本資料ではread()関数、readline()関数、readlines()関数の3つの関数を使う方法と、for文での読み込み、listを使った読み込みの合計5つの方法での読み込み方について説明する。

まず読み込むためのファイルを作るために、まず書き込み方法を説明する。
書き込むためのモードは読み込みの説明の後に詳しく説明する。

```
file = open("テスト.txt", "w")
print(file.write("読み込みテスト1行目\n読み込みテスト2行目"))
file.close()
```

```
21
```

このサンプルプログラムで `テスト.txt` ファイルに21文字の文字を書き込むことができる。実際に書き込めたか確かめるために、次は読み込んでみよう。

まずは、読み込みためのモードについて説明する。このモードは、open()関数の二つ目の引数に `r` を指定することで実行できるモードである。指定された名前のファイルが存在すれば、正常に開くことができ、データを読み込むことができる。
まずは、次のサンプルプログラムを見てみよう。

```
file = open("読めないファイル.txt", "r")
```

このサンプルプログラムを実行すると、エラーとなるはずである。このモードでは読み込む対象のファイルがなければ作成するのではなく、エラーとなって開くことができないことに注意する。

次に、読み込むためのモードで正常に開いた後に読み込む方法について説明していく。まずは、read()関数を用いる方法である。この関数はファイル内の全ての文字を一括して読み込む関数である。
次のサンプルプログラムを見てみよう。

```
file = open("テスト.txt", "r")
print("1回目の読み込み")
print(file.read())
file.close()
print("読み込み終了")
```

```
1回目の読み込み
読み込みテスト1行目
読み込みテスト2行目
読み込み終了
```

このサンプルプログラムでは、先ほど作成したファイル `テスト.txt` ファイルを開き、読み込んでいる。read()関数は、ファイルの中身を全て一括して読み込む関数であるため、先程書き込んだ内容を全て読み込むことができている。

次に、readline()関数を用いる方法を説明する。この関数は、ファイルの内容を行単位で読み込むことができる関数である。
次のサンプルプログラムを見てみよう。

```
file = open("テスト.txt", "r")
print("1回目の読み込み")
print(file.readline()) # readline関数例
print("1回目の読み込み終了")
print("2回目の読み込み")
print(file.readline())
print("2回目の読み込み終了")
file.close()
print("読み込み終了")
```

```
1回目の読み込み
読み込みテスト1行目

1回目の読み込み終了
2回目の読み込み
読み込みテスト2行目
2回目の読み込み終了
読み込み終了
```

read()関数とは違い、1行ずつreadline()関数を呼ぶたびに内容が読み込まれる。一つ目の出力結果から分かる通り、readline()関数で読み込まれる1行には改行コードも含まれてしまう事に注意しよう。読み込んだ内容と文字列を比較するときなどによく起こりやすいミスなので、readline()関数を使って文字列の比較を行うときなどには気を付けよう。

次に、readlines()関数を用いる方法について説明する。この関数は、ファイルの内容を行単位で読み込んだうえで、その全ての行を一括して読み込む関数である。
次のサンプルプログラムを見てみよう。

```
file = open("テスト.txt", "r")
print("1回目の読み込み")
print(file.readlines())
print("1回目の読み込み終了")
file.close()
print("読み込み終了")
```

```
1回目の読み込み
['読み込みテスト1行目\n', '読み込みテスト2行目']
1回目の読み込み終了
読み込み終了
```

全て読み込む点ではread()関数と同じだが、読み込み方はreadline()と同様に、行単位で読み込んだ内容を1行ずつリストの要素にして読み込むという動作をする。行単位で読み込みたいが、readline()関数で1行ずつ読むのではなく、全部読み込みたいときに使う関数である。

次に、関数ではなく、直接for文を使った読み込む方法について説明する。ここまでは関数を用いて読み込んだが、open()関数が返すファイルオブジェクトは、リストなどと同じ反復可能なオブジェクトなので、for文を使う事ができる。
次のサンプルプログラムを見てみよう。

```
file = open("テスト.txt", "r")
print("読み込み開始")
i = 0
for line in file: # for文での読み込みの例
    i += 1
    print(i, "回目の読み込み")
    print(line)
    print(i, "回目の読み込み終了")
file.close()
print("読み込み終了")
```

```
読み込み開始
1 回目の読み込み
読み込みテスト1行目

1 回目の読み込み終了
2 回目の読み込み
読み込みテスト2行目
2 回目の読み込み終了
読み込み終了
```

readline()関数と同じように行単位で読み込むことができる。readline()関数をfor文の中で使う事もできるが、readline()関数を使わなくてもfor文だけで読み込むことも可能である。

次に、list()を使った方法について説明する。これもfor文の時と同じ理由で、ファイルオブジェクトは反復可能なオブジェクトなので、list()関数も使える。
次のサンプルプログラムを見てみよう。

```
file = open("テスト.txt", "r")
print("1回目の読み込み")
print(list(file)) # list(file)の例
print("1回目の読み込み終了")
file.close()
print("読み込み終了")
```

```
1回目の読み込み
['読み込みテスト1行目\n', '読み込みテスト2行目']
1回目の読み込み終了
読み込み終了
```

list()関数を用いた方法ではreadlines()関数を用いた方法と同じ出力結果になる。プログラムの可読性を上げるために視覚的にリスト型であることを明示したい場合は、readlines()関数を使うよりも、このlist()関数を使う方法の方が有効である。

次に書き込むためのモードについて説明していく。

書き込むためのモードで開くと、write()関数を用いてファイルの入力、つまり書き込みを行うことができるが、read()関数などを用いてファイルを読み込むことはできないことに注意しよう。
この書き込むためのモードの中でも、排他的生成、書き込み、追記の3つのモードが存在する。それぞれの違いについてサンプルプログラムを用いながら、説明する。

まず、最もよく使われる書き込みを説明する。 open()関数の二つ目の引数に、 `w` を指定したモードである。指定された名前のファイルに対して、指定したファイルが存在するなら元々記載されていた内容は全て削除し、指定したファイルが存在しないなら新しくファイルを作成し、書き込むためのモードとして開くことを指定するモードである。
文字だけの説明ではわかりにくいので、実際に次のサンプルプログラムを実行して、その特性を説明する。

```
file = open("テスト.txt", "w")
print("1回目の書き込み中..")
file.write("テスト1")
file.close()

file = open("テスト.txt", "r")
print("1回目の読み込み中..")
print(file.read())
file.close()

file = open("テスト.txt", "w")
print("2回目の書き込み中..")
file.write("テスト2")
file.close()

file = open("テスト.txt", "r")
print("2回目の読み込み中..")
print(file.read())
file.close()
```

```
1回目の書き込み中..
1回目の読み込み中..
テスト1
2回目の書き込み中..
2回目の読み込み中..
テスト2
```

このサンプルプログラムではモード `w` の特徴を説明するために2回に分けて書き込む処理を行っており、1回目に `テスト1` 、2回目に `テスト2` という文字を書き込んでいる。 この時、1回目の書き込みで記載した文字は2回目の書き込みの後はなくなり、2回目に書き込んだ文字で上書きされることがわかる。

<!-- この通り、モード`w`は書き込んだ内容で前に記載されていた文字を全て上書きするモードである。 -->

次に、追記を説明する。 このモードはopen()関数の二つ目の引数に `a` を指定するモードで、指定された名前のファイルに対して、存在しないなら新しくファイルを作成し、存在するなら内容を消さずにその内容の末尾から、書き込むためのモードとして開くことを指定するモードである。
先ほどの書き込みの説明と同様に、次のサンプルプログラムを見てみよう。

```
file = open("テスト.txt", "a")
print("3回目の書き込み中..")
file.write("追記1")
file.close()

file = open("テスト.txt", "r")
print("3回目の読み込み中..")
print(file.read())
file.close()
```

```
3回目の書き込み中..
3回目の読み込み中..
テスト2追記1
```

このサンプルプログラムでは、先ほど使用した `テスト.txt` ファイルに対して追記のモードで開き、 `追記1` という文字を書き込んでいる。この時、元々あった内容は消さずに末尾から追記していくため、 `テスト2` のすぐ末尾から `追記1` が書き込まれる。今回は改行を入れなかったため、 `テスト2` の次の行から追記されるのではなく、すぐ後ろから追記されることに注意しよう。

最後に、排他的生成を説明する。
このモードはopen()関数の二つ目の引数に `x` を指定することで、実行できるモードである。存在しないファイルに対してファイルを生成し、書き込むためのモードとして開くことを指定するモードである。
次のサンプルプログラムを見てみよう。

```
file = open("テスト.txt", "x")
```

このサンプルプログラムを実行すると、エラーとなる。エラーとなるのは、排他的生成でファイルを開くと、一度生成されているファイルは開くことができないためである。よって、9章で説明するエラーと例外処理を用いることによって、存在していないファイルに対してのみ処理を行うようにできる。したがって、書き込みや追記のように既存のファイルの内容を削除したり、追記したりせずに、あくまでも新しいファイルに書き込みたい場合に使う事が多い。

次のサンプルプログラムを見てみよう。今まで作成しなかったファイルを開くことによって正常にファイルを開き、書き込むプログラムとなっている。

```
file = open("排他的生成.txt", "x")
print("書き込み中..")
file.write("排他的生成\r\n排他的生成\r\n")
file.close()

file = open("排他的生成.txt", "r")
print("読み込み中..")
print(file.read())
file.close()
```

```
書き込み中..
読み込み中..
排他的生成
排他的生成
```

今度は `排他的生成.txt` という存在しないファイルを指定することによって、エラーなく処理ができる。このプログラムによって、 `排他的生成.txt` という名前のファイルに出力された通りの文字が記載される。
`\r\n` はWindows環境での改行を表す文字記号で、このプログラムでは `排他的生成` という文字の後に改行を入れるようにして書き込んでいる。この `\r\n` を入れないと続けて文字が並べられることに注意しよう。

ここまでファイルの入出力の方法を説明してきたが、ここまでで扱ってきたものは全てテキストモードと呼ばれるモードで、ファイルに対して文字列、つまりstr型の変数を書き込んだり読み込んだりするモードである。しかし、ファイルの入出力にはバイナリモードと呼ばれるモードも存在する。このモードは実行ファイル `EXE` や画像ファイル `JPEG` ファイルなどバイナリつまり、01が並んだような形式のファイルを読み込むためのモードである。
ここで、実際にファイルを読み込む場合、文字コードで変換して読み込むか、01のままで読み込むかの違いのほかに、テキストモードとバイナリモードには違いがある。それは、改行コードの扱いである。改行コードは、改行を表す文字を意味するが、動作する環境によって改行を表す文字記号に違いがある。主なOSで言うと、Windows環境では `\r\n` 、Mac環境やLinux環境では `\n` といった違いが存在する。この改行コードの違いを加味するために、テキストモードでは、改行コードを変換してから読み書きを行う。例えば、Windows環境で作成されたファイルをテキストモードで読み込むと、改行コードを `\r\n` から `\n` に変換してから読み込むという動作をする。一方でバイナリモードではデータを壊さないようにするためにテキストモードのような改行コードを変換せずに読み書きを行うため、もし改行コードがある場合は作成された環境の改行コードのまま読み込むことができる。

ここからはバイナリモードでのファイルの扱い方を説明する。
バイナリモードでファイルを扱う場合は、読み込む場合も書き込む場合もモードの文字に `b` を追加すればよい。例えば、読み込みモードであれば `rb` とモードを指定すればいい。よって、どの開き方でも変わらないので、読み込みでの開き方だけを例として説明する。
次のプログラムを見てみよう。

```
file = open("排他的生成.txt", "r")
print("テキストモードでの読み込み")
print(file.readlines())
file.close()
print("テキストモードでの読み込み終了")
print("バイナリモードでの読み込み")
file = open("排他的生成.txt", "rb")
print(file.readlines())
file.close()
print("バイナリモードでの読み込み終了")
```

```
テキストモードでの読み込み
['排他的生成\n', '排他的生成\n']
テキストモードでの読み込み終了
バイナリモードでの読み込み
[b'\xe6\x8e\x92\xe4\xbb\x96\xe7\x9a\x84\xe7\x94\x9f\xe6\x88\x90\r\n', b'\xe6\x8e\x92\xe4\xbb\x96\xe7\x9a\x84\xe7\x94\x9f\
バイナリモードでの読み込み終了
```

バイナリモードで読み込むと、文字列は人間にわかる文字に変換される前のASCII文字コードであるバイナリコードが出力される。1行ごとに読み込んだ出力結果から分かる通り、確かにテキストモードでは改行コードが `\n` に変換されているが、バイナリモードでは `\r\n` のままになっていることが確認できるはずである。よって、テキストモードで開くと、少なくとも改行コードに関しては元のファイルとは違う文字に変換してしまうため、今回扱ったテキストファイルではなく、 `JPEG` ファイルなどのバイナリファイルを開く場合は、バイナリモードを使うのが良い。

クリーンアップ処理

読み込みモードで開いたファイルは、閉じなくても削除することができる。書き込みモードで開いたファイルは、閉じなくても、同じファイルを書き込みモードでも読み込みモードでも開ける。読み込みモードで開いたファイルを閉じずに削除した後、読み込みをもう一度行うと中身がなくなっていることになるだけだが、書き込みモードで開いたファイルを閉じずにもう一度書き込みモードで開くと、ここまでで説明してきた挙動とは違う動作をするので、初めのうちは1回1回閉じることをお勧めする。

例えば、次のサンプルプログラムを見てみよう。このサンプルプログラムではわざと閉じずに書き込んだり、読み込んだり削除したりを行っている。ここで、英字を書き込んでいるのはそうしないと読み込む際にエラーとなってしまい、おかしな動作をしていることを示せないためである。

```
file = open("おかしな動作.txt", "w")
print("書き込み")
file.write("okasikunaru 1 lines\ndoing 2 lines")
file = open("おかしな動作.txt", "w")
print("閉じずに開き直して書き込み")
file.write("should overwrite")
file.close()

file = open("おかしな動作.txt", "r")
print("読み込み")
print(file.read())
print("読み込み終了")
print("閉じずにファイルを削除")
! rm おかしな動作.txt
print("再度読み込み")
print(file.read())
print("読み込み終了")
file.close()
```

```
書き込み
閉じずに開き直して書き込み
読み込み
should overwritenes
doing 2 lines
読み込み終了
閉じずにファイルを削除
再度読み込み

読み込み終了
```

まず、このサンプルプログラムで本来期待される動作について説明する。書き込みに関しての動作では、2回とも、 `w` モードでファイルを開いて書き込んでいるため、ここまでの説明で紹介した通り、1回目に開いて書き込んだ文字 `"okasikunaru 1 lines\ndoing 2 lines` が、2回目に開いたときには、全て削除され、2回目に書き込んだ文字 `should overwrite` で全て上書きされることが期待される。次に、読み込みに関しては、開いたままファイルを削除でき、ファイルは存在しないがリソースは解放されていないので、ファイルが存在し、中身が削除されたとみなされて、再度読み込むことができることを示すサンプルプログラムとなっている。
しかし、このサンプルプログラムの書き込みにおける動作は、書き込みに関しては期待された動作とは少し違った動作をしている。具体的には、1回目に開いて書き込んだ文字 `"okasikunaru 1 lines\ndoing 2 lines` が、2回目に開いたときには、全て削除されずに残ってしまっている。つまり、2回目に書き込んだ後の出力は `should overwrite` が表示されることが期待されるが、 `should overwritenes\ndoing 2 lines` という出力になっており、1行目の `should overwrite` の文字数分だけ上書きしたような動作となり、期待された動作をしていないことがわかる。この動作は、今までのサンプルプログラムのように日本語で行うと、中途半端な文字までが2回目に書き込んだ文字で上書きされることにより、文字コードが解読できないコードとなってしまい、読み込むことすらできなくなることを引き起こす。よって、本来は上書きして新しいファイルを作り直したうえで読み込むことを期待していたが、意図しない文字が読み込まれており、バグの原因となっている。
このことから、ファイルなどのリソースは他の変数と違い、新しく開き直したからと言って、自動的にリソースが解放されるわけではなく残り続

けてしまい、意図した動作とは違う動作を引き起こす原因となる。よって、ファイルは一度開いたらすぐ閉じる癖をつける必要があるので、ここまでの説明でもファイルを開いたら閉じるといったことを行ってきた。

ここからは、意図した動作をするようにファイルを開いた後に閉じる方法として、close()関数を使う方法と、`with`文を使う方法を紹介していく。

まずは、ここまでの説明でも何度か使用してきたclose()関数の説明をしていく

```
file = open("テスト.txt", "r")
print(file.read())
file.close()
```

```
テスト2追記1
```

ここまでの説明で行ってきたように読み込みが終わった後で、close()関数を呼ぶ。close()関数を読んだ後は定義したファイルオブジェクトにアクセスすることはできなくなるので、注意しよう。

次に、`with`文を使う方法を説明する。`with`を使うとファイルを開いているスコープが分かり易く、`with`文のスコープを抜ければ、自動的にファイルが閉じられるため、閉じ忘れになる心配がない。しかし、`with`文の外ではファイルオブジェクトにアクセスできないという点は注意する必要がある。
次のサンプルプログラムを見てみよう。

```
print("書き込み")
with open("テスト.txt", "w") as file: # withの例
    file.write("with文で書き込み\n")

print("読み込み")
with open("テスト.txt", "r") as file:
    print(file.read())
print("読み込み終了")
```

```
書き込み
読み込み
with文で書き込み

読み込み終了
```

`with`文は9章で説明される例外処理を1文で行えるような文となっている。まず、open()関数を実行する関数とそれを終了する関数（close()関数）をロードする。その後、open()関数を実行した時に、エラーが出なければそのままファイルが開かれ、9章で説明されるfilenally節で、必ずファイルを閉じる動作を行う。もしエラーが出たり、エラーが出なくても途中で`with`文ののスコープから抜けたときにも9章で説明するtry-except節のような構造で、エラーをキャッチし、ファイルを閉じる動作を行い、閉じる動作が失敗すればエラーを投げるという構造になっている。よって、これらの動作を`with`文1文で実現しているのでとても便利で、ファイルの閉じ忘れもなくなるので、この`with`文をファイル操作の場合には使う事が多い。

<!-- 参照：https://docs.python.org/ja/3/reference/compound_stmts.html?highlight=with -->

標準入出力

標準入出力に直接関わる関数

標準入力とは実行中のプログラムが利用する標準的なデータの入力元を示す。Pythonでは、input()関数という関数を呼び出すことで、プログラムの実行中に、キーボードを使ったユーザからの入力を標準入力として受け付けることができる。
次のサンプルプログラムを見てみよう。

```
sentence = input()
# ここでキーボードで「標準入力テスト」と入力する。
print(sentence)
```

```

```

input()関数は、プログラムの実行中にユーザからの入力を待ち、その入力された値を戻り値として返す関数である。この関数の戻り値を用いることによって、キーボードの入力を受け取って、表示することができている。

標準出力とは実行中のプログラムが利用する標準的なデータの出力先を示す。ここまでの説明でも使ってきたprint()関数などによって、実行中のプログラムが標準出力に出力し、ターミナルなどに表示している。
次のサンプルプログラムを見てみよう。

```
print("標準出力テスト")
```

```
標準出力テスト
```

print()関数は引数に指定された文字、数値を標準出力に表示する関数である。よって、今まで使っていたprint()関数で標準出力に表示できる。

Pythonには、標準で `sys.stdout` というファイルオブジェクトが用意されている。このファイルオブジェクトは今までのファイルとは違い特別なファイルで標準出力を表すファイルオブジェクトである為、常に開いた状態になっている。この特別なファイルオブジェクトを使うと、標準出力というファイルに書き込むという動作を行う事により、直接標準出力に表示することができる。
次のプログラムを見てみよう。

```
import sys
sys.stdout.write("標準出力テスト")
```

```
標準出力テスト
```

`sys` というのは7章で説明されたモジュールで、そのモジュールの中でもPythonの標準ライブラリとして用意されているモジュールである。その為、今回の `sys.stdout` というファイルオブジェクトを使うためには、 `sys` モジュールをimportする必要がある為、まずimportしている。この `sys.stdout` を使うと、このプログラムの出力結果から分かる通り、print()関数を使わなくても直接標準出力を出すことができる。

標準エラー出力とは実行中のプログラムが標準的に利用できる出力先で、エラーや警告などプログラムの実行を継続することが不可能、または実行を継続すると問題が発生する可能性があることを示す文字を表示することを目的とした出力先のことである。
次のプログラムを見てみよう。

```
import sys
print("標準エラー出力テスト", file=sys.stderr)
```

このプログラムのように端末上では標準出力とは区別ができない出力となる。print()関数はfile引数に出力先を指定でき、このプログラムでは `sys.stderr` という標準エラー出力を表す特別なファイルオブジェクトを指定することにより、標準エラー出力を出している。

標準エラー出力も標準出力と同様にPythonに標準で `sys.stderr` というファイルオブジェクトが用意されている。このファイルオブジェクトも標準出力と同様にプログラムの実行中は常に開いた状態になっているため、通常のファイル操作のようにwrite()関数を用いて書き込むことによって、標準エラー出力とすることができる。
次のプログラムを見てみよう。

```
import sys
sys.stderr.write("標準エラー出力テスト")
```

標準出力と区別はできないが、print()関数で出力した時と同様に表示することができる。

主に入力の成形などに使われる関数

入力するときは余分なスペースが入ることがある。入力時に入ってしまったスペースによって、文字列比較などで同じ文字列のはずなのに違う文字列と判定されてしまうといった意図しない動作を引き起こすことがある。その為、strip()関数を利用すれば、この余分なスペースを削除することができる。
早速次のプログラムを見てみよう。

```
sentence = "    ようこそ Pythonの世界へ!    "
print(sentence)
print(sentence.strip())
```

```
    ようこそ Pythonの世界へ!
ようこそ Pythonの世界へ!
```

入力として与えた文字列の前と後にあったスペースが、strip()関数を使ったときは、削除できている。よって、空白も含めたそのままの文字列を扱いたい場合以外は、入力の文字列にはstrip()関数を適用してスペースを削除することが多い。

入力された文字を空白毎に区切って分解したいときがある。そんなときはsplit()を使おう。
次のサンプルプログラムを見てみよう。

```
sentence = "ようこそ Pythonの世界へ!"
print(sentence.split())
```

```
['ようこそ', 'Pythonの世界へ!']
```

split()関数は、指定されたデリミタ文字列によって区切った単語のリストを返す関数である。デリミタ文字列はsplit()関数の引数として受け取る区切り文字のことで、何も指定しない場合は空白の文字列が指定される。引数に何も指定せずにsplit()関数を使う事で、`ようこそ`と`Pythonの世界へ!`の二つに分解されたリストが得られる。また、デリミタ文字列にはどんな文字列も指定できるが、空白や`,`が使われることが多い。

int()関数を用いることによって、文字列を数値に変換することもできる。
次のサンプルプログラムを見てみよう。このサンプルプログラムは、int型の数値に変換するプログラムである。

```
value_of_str = "123"
print(type(value_of_str))
print(value_of_str)

value_of_int = int(value_of_str)
print(type(value_of_int))
print(value_of_int)
```

```
<class 'str'>
123
<class 'int'>
123
```

文字列を表すstr型から、数値を表すint型にint()関数を使う事で変換することができた。int()関数で変換できる文字列は、数値になっている文字列のみとなっているため、先ほどまでの例で使用した `ようこそ Pythonの世界へ!` などの通常の文字は変換することができないことに注意しよう。

mapとlistを組み合わせると、文字列のリストを整数のリストに変換できる。map()関数は、一つ目の引数に関数、2つ目の引数にリストやタプルのような反復可能なオブジェクトを指定し、2つ目の引数に指定したオブジェクトの全ての要素に対して、1つ目の引数で指定した関数を適用するためのイテレータを返す関数である。イテレータとは、データの流れを表現するオブジェクトで、next()関数を使う事でしか次の要素にアクセスできないオブジェクトを表す。その為、リストなどとは違いprint()関数でその内容を一度に全て表示することはできないオブジェクトある為、通常、表示したい場合はlist()関数を使ってリストに変換してから表示する。
次のサンプルプログラムを見てみよう。このサンプルプログラムは、 `1,2,3` という文字列を `,` という区切り文字で分解したリストを作成した後に、リストの要素それぞれをint型に変換したリストに変えるプログラムである。

```
values_of_str = "1,2,3".split(",")
values_of_int = list(map(int, values_of_str))

print(values_of_str)
for value in values_of_str:
    print(type(value))

print("-----")

print(values_of_int)
for value in values_of_int:
    print(type(value))
```

```
['1', '2', '3']
<class 'str'>
<class 'str'>
<class 'str'>
-----
[1, 2, 3]
<class 'int'>
<class 'int'>
<class 'int'>
```

文字列のstr型の変数を要素を持つリストがmap()関数とlist()関数を用いることによって、数値を表すint型の変数を要素に持つリストに変換できた。このように通常、for文を使ってするような変換をmap()関数を使えば、1行で実現することができるため、リストの要素の型変換などでよく使われる。

主に出力の整形などに使われる関数

文字列の結合は文字列同士でないと結合できない。文字列とそれ以外のオブジェクトを結合したいときはstr()関数を使い、文字列に変換してから行おう。str()関数はオブジェクトを文字列に変換し、改行コードなどの文字列記号もその記号の意味する文字列に変換する関数である。
次のサンプルプログラムを見てみよう。このサンプルプログラムでは、数値を表すint型の変数と数値を要素に持つリストを文字列に変換したプログラムである。

```
num1 = 123
list1 = [1, 2, 3]

print("数値は" + str(num1) + "で、リストは" + str(list1) + "です。")
```

```
数値は123で、リストは[1, 2, 3]です。
```

数値もリストも文字列に変換することで文字列として結合できた。

<!-- 結合しなくてもprint()関数で表示するだけであれば、print()関数内部で文字列に変換してくれるため、str()関数をユーザが指定して文字列に変換する必要はない。 -->

デバッグなどで、文字列に空白があるか、改行があるか明示的に確認したいことがある。しかし、str()関数ではこれらの文字は明示的には表現されない。この時、repr()関数を使う事でこの空白文字や改行文字に関しても文字として表現することができる。
次のサンプルプログラムを見てみよう。このサンプルプログラムでは、空白や改行のある文字をrepr()関数を用いて文字列に変換し、表示するプログラムである。

```
sentence = "    ようこそ Pythonの世界へ!    \n"
sentences = repr(sentence)
print(sentence)
print(sentences)
```

```
    ようこそ Pythonの世界へ!

'    ようこそ Pythonの世界へ!    \n'
```

一つ目の出力では `世界へ！` の後にある空白は推測するしかなく、自分で入力した文字列ではなかった場合、空白があることは分からない。しかし、二つ目の出力では入力文字列をそのまま表示しているため、空白があることも改行があることも明示的に示すことができる。

str()関数やrepr()関数の他にも、数値などのオブジェクトを文字列に変換する方法として、フォーマット済み文字列リテラルやformat()関数、 `%` 演算子を使う方法も存在する。これら3つの方法について順番に説明していく。ただし、この中で `%` 演算子を使う方法は古い書式であり、Pythonの公式ではあまり推奨されていない方法である。よって、実際に使う際にはformat()関数やフォーマット済み文字列リテラルを使う事が推奨される。

まずは、フォーマット済み文字列リテラルを説明する。フォーマット済み文字列リテラルは文字列の頭に `f` か `F` を付け、文字列に変換したいオブジェクトを表す変数を `{}` で囲んでいる文字列のことを表す。
次のサンプルプログラムを見てみよう。

```
num1 = 123
list1 = [1, 2, 3]

print(f"数値は{num1}で、リストは{list1}です。")
```

```
数値は123で、リストは[1, 2, 3]です。
```

文字列の一部として、数値などのオブジェクトを文字列に変換することができている。フォーマット済み文字列リテラルを使う事で、文字列の一部として記載することができるため、便利である。

次にformat()関数を説明する。format()関数は、str型のオブジェクトの関数であるので、通常の関数と同様に `"".format()` といった使い方をする関数である。format()関数の引数は、文字列に変換したいオブジェクトを指定し、 `""` または `''` で囲まれた文字列の中の `{}` で囲まれた部分を指定されたオブジェクトを文字列に変換したもので置換する。
次のサンプルプログラムを見てみよう。

```
num1 = 123
list1 = [1, 2, 3]

print("数値は{}で、リストは{}です。".format(num1, list1))
print("数値は{0}で、リストは{1}です。".format(num1, list1))
print("リストは{1}で、数値は{0}です。".format(num1, list1))
print("数値は{0}で、リストは{0}です。".format(num1, list1))
```

```
数値は123で、リストは[1, 2, 3]です。
数値は123で、リストは[1, 2, 3]です。
リストは[1, 2, 3]で、数値は123です。
数値は123で、リストは123です。
```

基本的な使い方としては、一つ目の出力のように `{}` で囲まれた部分をformat()関数の引数に指定した順番に置換する。また、二つ目以降の出力のように、format()関数の引数と `{}` で囲まれた部分の対応は `{}` に数値の指定がある場合は、指定された位置の引数と対応付けされる。

文字列への変換の方法の最後として、 `%` 演算子を使う方法について説明する。 `%` 演算子は、 `""` または `''` で囲まれた文字列を一つ目のオペランド、 `()` で囲まれたオブジェクトのタプルまたはオブジェクトを二つ目のオペランドとして使う。動作としては、一つ目のオペランドである `""` または `''` で囲まれた文字列の中の `%s` などの `%` で始まる部分を、二つ目のオペランドであるオブジェクトのタプルの中身または、オブジェクトで順番に置換する動作をする。
次のサンプルプログラムを見てみよう。

```
num1 = 123
list1 = [1, 2, 3]

print("数値は%sです。" % num1)
print("数値は%sで、リストは%sです。" % (num1, list1))
print("リストは%sで、数値は%sです。" % (list1, num1))
```

```
数値は123です。
数値は123で、リストは[1, 2, 3]です。
リストは[1, 2, 3]で、数値は123です。
```

一つ目の出力は数値だけのオブジェクトを二つ目のオペランドとしたときの出力で、二つ目以降の出力は、二つ以上のオブジェクトのタプルを二つ目のオペランドとしたときの出力である。二つ目のオペランドで指定したオブジェクトを指定した順番に、文字列の中の `%s` の部分に置換している。

ここまでの説明で紹介したフォーマット済み文字列リテラル、format()関数、 `%` 演算子を使う方法で文字列に変換する方法は、基本的な使い方である。よって、フォーマット済み文字列リテラル、format()関数、 `%` 演算子を実数などの数値を文字列に変換して表示するときは、応用的な使い方として、桁数や小数点以下の桁数を指定するオプションが存在する。
次のサンプルプログラムを見てみよう。

```
import math
print(f"通常のπの値の表示 {math.pi}")
print(f"フォーマット済み文字列リテラルで小数点以下3桁までを表示 {math.pi:.3f}")
print("format()関数で小数点以下3桁までを表示 {0:.3f}".format(math.pi))
print("パーセント演算子で小数点以下3桁までを表示 %.3f" % math.pi)

print(f"フォーマット済み文字列リテラルで全体の文字数（桁数）を10になるように表示 {math.pi:10.3f}")
print("format()関数で全体の文字数（桁数）を10になるように表示 {0:10.3f}".format(math.pi))
print("パーセント演算子で全体の文字数（桁数）を10になるように表示 %10.3f" % math.pi)

print(f"フォーマット済み文字列リテラルで全体の文字数（桁数）を10になるように表示し、足りない桁は0埋め {math.pi:010.3f}")
print("format()関数で全体の文字数（桁数）を10になるように表示し、足りない桁は0埋め {0:010.3f}".format(math.pi))
print("パーセント演算子で全体の文字数（桁数）を10になるように表示し、足りない桁は0埋め %010.3f" % math.pi)
```

```
通常のπの値の表示 3.141592653589793
フォーマット済み文字列リテラルで小数点以下3桁までを表示 3.142
format()関数で小数点以下3桁までを表示 3.142
パーセント演算子で小数点以下3桁までを表示 3.142
フォーマット済み文字列リテラルで全体の文字数（桁数）を10になるように表示       3.142
format()関数で全体の文字数（桁数）を10になるように表示       3.142
パーセント演算子で全体の文字数（桁数）を10になるように表示       3.142
フォーマット済み文字列リテラルで全体の文字数（桁数）を10になるように表示し、足りない桁は0埋め 000003.142
format()関数で全体の文字数（桁数）を10になるように表示し、足りない桁は0埋め 000003.142
パーセント演算子で全体の文字数（桁数）を10になるように表示し、足りない桁は0埋め 000003.142
```

小数点や整数の桁数を指定したい場合は、例えば小数点以下3桁であれば、フォーマット済み文字列リテラル、format()関数を使う場合は `{}` の中で `{math.pi:.3f}` のように `:` の後に桁数を指定すればいい。 `%` 演算子の場合は `%.3f` のように `%` の後にそのまま記載すればいい。 `.` は小数点を表し、 `.` の後の数字の値で小数点以下の桁数を指定している。 `.3f` の `f` は文字列のフォーマットで浮動小数点を表し、 `math.pi` のような `float` 型などの実数を表現する際に指定する。
また、小数点以下ではなく全体の桁数を指定でき、 `.` の前または `.` なしの場合で指定した数値で指定できる。この場合、足りない桁の分は空白で埋められる。このサンプルプログラムでは、全体の文字数が10になるように表示した上で桁が足りない分は空白になって表示されることが確認できる。空白だと桁数が分かりづらい場合は、その数値の前に `0` を追加することで、桁が足りない分を空白の代わりに `0` で埋めることができる。このサンプルプログラムでは、全体の文字数が10になるように表示した上で足りない分は0で埋めるように指定していることが確認できる。

問題6

問題文

空白を含むN個の単語列$A[1], A[2], A[3], ...A[N]$が与えられます。$A[1], A[2], A[3], ...A[N]$のそれぞれの単語を改行区切りで出力してください。

制約

- $1 \leq N \leq 100$
- $1 \leq len(A[x]) \leq 10$
- 入力は全て文字列である。
- 文字列に改行は含まれない。

入力

入力は次の形式で与えられます。

$A[1]A[2]...A[N]$

出力

$A[1], A[2], A[3], ...A[N]$を改行区切りで出力してください。

入力例 1

```
input
```

出力例 1

```
input
```

入力例 2

```
input words
```

出力例 2

```
input
words
```

解答の雛形

```
# ここに解答を入力
```

問題2

問題文

n行に渡って整数A, Bが与えられます。$A * B + n$の計算結果をそれぞれ改行区切りで出力してください。

制約

- $0 \leq A, B \leq 10^8$
- $1 \leq n \leq 100$
- 入力は全て整数である。

入力

入力は次の形式で与えられます。

```
$n$
$A[1]$ $B[1]$
$A[2]$ $B[2]$
...
$A[n]$ $B[n]$
```

出力

$A * B + n$をそれぞれ改行区切りで出力してください。

入力例 1

```
1
2 5
```

出力例 1

```
11
```

入力例 2

```
19
72641946 40584646
84881071 64412869
51198586 79134948
2267581 51397492
95484189 98031437
96719676 76847990
95614353 4917238
27541280 93624564
54013676 81594543
92995887 91134521
35009911 42448544
87965377 7086108
35513922 94599387
12036471 57995518
70564553 62427226
35741031 27150266
8359719 46428555
37104992 37246810
58593626 64513326
```

出力例 2

```
2948147663161135
5467433306902718
4051597440783547
116547976306871
9360452258449612
7432712694051259
470158529917033
2578540332001939
4407221208970087
8475135616715146
1486119747519603
623332161682735
3359595251165833
698061370536997
4405149297719997
970378498764265
388129673376064
1382042587075539
3780069695660095
```

解答の雛形

```
# ここに解答を入力
```

問題3

問題文

文字列Aが与えられます。 文字列Aの長さが5文字より長い場合は

```
[文字列A] is long
```

5文字以下の場合は

```
[文字列A] is short
```

と出力してください。

制約

- $1 \leq len(A) \leq 10$
- 入力は全て文字列である。
- 文字列に改行は含まれない。

入力

入力は次の形式で与えられます。

```
A
```

出力

入力で与えられたAを埋め込み、

```
[文字列A] is long
```

または

```
[文字列A] is short
```

と出力してください。

入力例1

```
longer
```

出力例1

```
longer is long
```

入力例2

```
input
```

出力例2

```
input is short
```

解答の雛形

```
# ここに解答を入力
```

問題4

問題文

整数A, B, Cが与えられます。 A / Bをの結果を小数点以下C桁まで表示し(C+1桁目を四捨五入し)、

```
[A] / [B] = [結果]
```

と出力してください。

制約

- $1 \leq A, B \leq 10^9$
- $0 \leq C \leq 10$

入力

入力は次の形式で与えられます。

```
A B C
```

出力

```
[A] / [B] = [結果]
```

と出力してください。

全てアラビア数字で出力してください。漢数字に変換する必要はありません。

入力例 1

```
1 3 3
```

出力例1

```
1 / 3 = 0.333
```

入力例2

```
800 400 5
```

出力例2

```
800 / 400 = 2.00000
```

参考

`.format` を用いて指定する桁数に変数を使用する場合、以下のように書くことができます。

```
import math

digits = 3
print('{0:.{1}f}'.format(math.pi, digits))

>>> 3.142
```

解答の雛形

```
# ここに解答を入力
```

問題5

文章Aが与えられます。

文章の中に、同じ文字の組み合わせが3文字以上連続している場合、その文章を `ダジャレである` とします。

ただし、同じ文字の組み合わせ部分が重複している場合は、ダジャレとみなされません。

与えられた文章が `ダジャレである` かどうか判定してください。

制約

- $6 \leq len(A) \leq 100$

入力

入力は次の形式で与えられます。

```
A
```

出力

ダジャレである場合には、

```
[A]はダジャレです。
```

ダジャレでない場合には

```
[A]はダジャレではありません。
```

と出力してください。

入力例1

```
たばたでばたばた
```

出力例1

```
たばたでばたばたはダジャレです。
```

入力例2

```
ふとんがふっとんだ
```

出力例2

```
ふとんがふっとんだはダジャレではありません。
```

解答の雛形

```
# ここに解答を入力
```

問題6

整数A, Bが与えられます。 AをBで割り切れなくなるまで割り続け、割り切れなくなった時点でその値を出力してください。 ただし、AをBの累乗の形で表せる場合は、 `[A] = [B] ** [n]` と出力してください。

制約

$1 \leq A, B \leq 10^9$

入力

入力は次の形式で与えられます。

```
A B
```

出力

AがBの累乗の形で表せる場合は

```
[A] = [B] ** [n]
```

と、それ以外の場合は計算結果を出力してください。

入力例1

```
12 3
```

出力例1

```
4
```

12 ÷ 3 = 4, 4は3で割り切れないことより、出力すべき値は `4` となります。

入力例2

```
8 2
```

出力例2

```
8 = 2 ** 3
```

解答の雛形

```
# ここに解答を入力
```

9章: エラーと例外

エラーの種類

本節ではPythonのエラーの種類について紹介する。エラーの種類は、「構文エラー」と「例外」の2つに分けることができる。

構文エラー

構文エラーとは、Pythonの構文として間違っているコードを書いた場合に発生するエラーである。 構文エラーの例について2つ紹介する。

まず、こちらのサンプルプログラムを実行してみよう。

```
for i in range(5):
    print("Hello World!)
print("end of code")
```

上記のサンプルプログラムを実行すると、エラーが発生し、エラーの内容が出力される。 `SyntaxError` とエラー文に記述されている。このエラー文は、構文エラーであることを示している。 `SyntaxError:` の後の文を見ると、 `unterminated string literal (detected at line 2)` と出力されている。2行目において文字列リテラルが終わっていない、ということを意味しているエラー文である。 実際にソースコードをよく見ると、2行目の `print("Hello World!)` には、 `Hello World!` の前にはダブルクォーテーション(")が存在する一方で、後ろには存在していないことが分かる。 よってエラーを取り除くには、2行目で `print("Hello World!")` のように、ダブルクォーテーションを追加すれば良い。

```
value1 = 20
value2 = 39
multiplied = value1 x value2
```

このサンプルプログラムでも `SyntaxError` と出力されていることから、構文エラーであることが分かる。 `SyntaxError:` の後の文を見ると、 `invalid syntax` 、と出力されており、有効ではない構文が記述されていることが分かる。 実際にソースコードを見てみると、3行目の `multiplied = value1 x value2` に有効でない構文が記述されている。 3行目では、恐らくは掛け算をしたかったのであろう場所でアルファベットの `x` を記述している。 Pythonでは掛け算は `*` の演算子によって行うため、 `x` を使用するのは間違えである。 上記のサンプルプログラムでは、 `value1` , `x` , `value2` の3つの変数をスペース区切りで書いてあるだけであり、そのような記法はPythonには存在しない。

エラーを取り除くには、3行目を `multiplied = value1 * value2` のように、正しい演算子を用いれば良い。

```
num = 10
divide = 0
num / divide
```

このサンプルプログラムでは `ZeroDivisionError` とエラー文が出力されている。このエラー文は、ゼロで割り算をしてしまったという例外であることを示す。 3行目の割られる値は `divide` であることから、 `divide` の中の値が何故 `0` になっているのか分かれば、例外を直せると考えることができる。 今回の例では2行目で `divide = 0` と記述されており、 `divide` に `0` を代入していることが原因と分かる。そのため、2行目で `divide = 2` と記述するなど、 `divide` に `0` 以外の数字を代入することで、例外を発生させないようにできる。 コードの文脈によっては `divide` の値が関数の戻り値などから与えられている場合があり、その場合には `divide` の値を計算している場所に遡って、例外の原因を探る必要がある。

```
value_str = 'a'
value_int = int(value_str)
```

このサンプルプログラムでは `ValueError` とエラー文が出力されている。このエラー文は、値が間違っているという例外であることを示す。 `ValueError` の後の文では ` invalid literal for int() with base 10: 'a'` と出力されていることから、 `int()` 関数の引数に `'a'` という間違ったリテラルを与えてしまったことで例外になっていることが分かる。 引数は `value_str` であり、 `value_str` は1行目

で定義している。 1行目では `value_str = 'a'` と記述されて、 `value_str` に対して `'a'` を代入していることが分かる。 `int()` 関数は文字列になっている数字をint型に変換する関数であることから、 `value_str` も文字列になっている数字である必要がある。 よって例外が発生しないようにするには、1行目を `value_str = '1'` など `value_str` に文字列になっている数字を代入すれば良い。 もちろんこの例外も、コードの文脈によっては直す方法が違う場合があるため注意が必要である。

```
a = input()
print(a)
```

このサンプルプログラムを実行すると `input()` 関数により入力を受け付ける。入力を受け付けしている途中で、処理を中断する。 すると、 `KeyboardIntrrupt` というエラー文が出力される。 このエラー文はプログラムの中断を命令するコマンドが入力されたことを示す例外である。 この例外は直さなくてもよい場合もある。もし処理を中断するコマンドを打った時に何か別の処理を行いたいときなどには、次の節で紹介する例外の処理を活用する場合もある。

例外の処理

概要

例外が発生したときに何もしなければ、発生した時点でプログラムが終了してしまう。 例外の発生により、プログラムが終了してもよい場合もあるが、例えば、車を動かすシステムなどでは常にプログラムが動いていることが想定されるプログラムが、実行途中に終了してしまっては事故の原因にもなりうる。 例外が発生してもプログラムは動き続けて欲しいし、例外の種類によっては運転手に警告を出して運転を止めさせたい。

車を例にすると複雑すぎるため、ここでは「値を受け取って、10をその値で割る」という簡単なプログラムを考えてみる。

```
div = int(input()) # 入力例: 0
10 / div
```

入力を受け取るとき、常に想像通りの入力をしてくれるとは限らない。 上記のサンプルプログラムでは、入力によって様々な例外が発生する場合がある。 例えば以下のような例外が発生する。 各例外が発生する理由は、前節の通りである。

- `0` を入力した場合: `ZeroDivisionError`
- `'a'` など数字以外の文字列を入力した場合: `ValueError`
- 処理を中断した場合: `KeyboardInterrupt`

ここでは少なくとも3種類の例外が発生するが、発生した例外の種類によって処理を分けたいという場面が多々ある。 発生した例外の種類によって処理を分ける方法を紹介する。

上記のサンプルプログラムを実行して、 `0` , `'a'` を入力したり、処理を中断したりすることで、上記3つの例外それぞれが起きた場合の挙動を確認できる。 `1` や `2` などを入力すると、例外が起きなかった場合の挙動を確認できる。 出力例は、全てのケースを網羅すると分量が膨大になってしまうため、いずれか1つを抜粋して用意している。

try文のtry節とexcept節

例外によって、処理を分けたい場合、 `try` 文を使用する。 サンプルプログラムを使って、 `try` 文について説明する。

```
try:
    div = int(input()) # 入力例: 0
    10 / div
    print("No error occurred") # エラーがこの行より前で発生した場合は、実行されない
except:
    print("Error occurred")
print("Continue processing")
```

```
0
Error occurred
Continue processing
```

上記のサンプルプログラムの `try` 文は、 `try` 節と `except` 節で構成されている。 例外処理を行いたいコードは、上記のサンプルプログラムのように `try` 節の中に含める。 すると、 `try` 節の中の処理を実行中に例外が発生した場合、以降の `try` 節中のコードをスキップして、後ろに記述されている `except` 節の中の処理が始まる。 このようにすることで、例外が発生したら即座にプログラムが終了という前節のようなことは起きず、一度 `except` 節の中の処理を行なった後、 `try` 文の後ろの処理を引き続き実行していく。 このサンプルプログラムでは `try` 節で発生した例外がどの種類であっても、 `except` 節の中の処理を実行する。

特定の例外が発生したときのみ、 `except` 節の中の処理を実行することもできる。 次のサンプルプログラムをみてみよう。

```
try:
    div = int(input()) # 入力例: ctrl-C
    10 / div
except Exception as e:
    print(e)
print("Continue processing")
```

```
division by zero
Continue processing
```

このサンプルプログラムでは、 `except` 節で `except Exception as e` と記述している。 `except Exception as e` と記述することで、発生した例外が `Exception` というクラスかその子孫の場合にのみ、 `except` 節の中の処理を実行する。 また `except` 節の中の処理を実行する前に、発生した例外のオブジェクトを `e` という変数に格納する。 例外の継承関係については公式ドキュメントのページから確認できる。

試しに `Exception` の子孫である `ZeroDivisionError` や `ValueError` といった例外を発生させる入力をすると、 `print(e)` や `try` 文の後ろの `print()` 関数を実行している。 よって `except` 節が実行されて処理が継続していることが確認できる。

一方で `KeyboardInterrupt` という例外を発生させる入力をすると、 `print(e)` を実行した結果とは異なり、tracebackを表示して、 `try` 文の後ろの `print()` 関数は実行されない。 よって、 `Exception` の子孫でない例外が発生すると、 `except` 節が実行されず、例外の発生時にプログラムが強制終了していることが確認できる。

```
try:
    div = int(input()) # 入力例: 0
    10 / div
except ZeroDivisionError:
    print("Error, can't divide by zero")
except Exception:
    print("Error!")
```

```
0
Error, can't divide by zero
```

`except` 節は1つの `try` 文の中で複数繋げることができる。 `except` 節を複数使用する場合は上から順に `except` 節の実行判定を行い、発生した例外が子孫関係にあると最初に判定された `except` 節のみを実行する。 上記のサンプルプログラムでは、 `ZeroDivisionError` が発生した場合は `Error, can't divide by zero` のみが出力され、 `ValueError` が発生した場合は `Error!` のみが出力される。 `KeyboardInterrupt` が発生した場合は、どの `except` 節も実行されずにプログラムが終了する。

```
try:
    div = int(input()) # 入力例: a
    10 / div
except (ZeroDivisionError, ValueError):
    print("Error!")
```

```
a
Error!
```

また、1つの `except` 節に複数の例外を指定したりすることもできる。1つの `except` 節に複数の例外を指定する場合は、 `except` の後に処理を行いたい例外をタプルで記述する。 この場合は、発生した例外がタプルの中のいずれかの子孫関係にあると判定されると、 `except` 節の中の処理を実行する。 上記のサンプルプログラムでは、 `ZeroDivisionError` や `ValueError` が発生した場合は `Error!` が出力される。 `KeyboardInterrupt` が発生した場合はどの `except` 節も実行されずにプログラムが終了する。

try文のelse節

`if` 文などでよく用いられる `else` は、 `try` 文においても使用することができる。

```
try:
    div = int(input()) # 入力例: 2
    10 / div
except Exception:
    print("Error!")
else:
    print("No error")
```

```
2
No error
```

`try` 文での `else` 節には、例外が発生しなかったときに実行する処理を記述できる。 上記のサンプルプログラムでは、 `1` や `2` など例外の発生しない入力を行うと、 `No error` と出力される。 例外の発生する入力を行なったときには、 `else` 節の中の処理は実行されない。

try文のfinally節

```
try:
    div = int(input()) # 入力例: a
    10 / div
except ZeroDivisionError:
    print("Error, can't divide by zero")
finally:
    print("Always executed")
```

```
Error, can't divide by zero
Always executed
```

`finally` 節の中に処理は、例外の有無によらず、 `try` 文の処理が終わる前に必ず実行される。 上記のサンプルプログラムでは `Always executed` はtry文の処理が終わる前に必ず出力される。 例え、 `ValueError` などによりプログラムが途中で終了するとしても `finally` 節の中の処理は実行される。

`finally` 節で実行するような、「例外の有無によらず、 `try` 文の処理が終わる前に必ず実行したい処理」はクリーンアップ処理と呼ばれることがある。

色々な例外

```
try:
    div = int(input()) # 入力例: a
    10 / div
except ZeroDivisionError:
    print("Error, can't divide by zero")
except Exception as e:
    print(e)
except:
    print("Error!")
else:
    print("No error")
finally:
    print("Always executed")
```

```
invalid literal for int() with base 10: 'a'
Always executed
```

ここまでの紹介した例外処理を使うと、上記のようなプログラムを記述できる。 上記のプログラムでは、 `ZeroDivisionError` の例外が発生したら、 `Error, can't divide by zero` を出力する。 `ZeroDivisionError` 以外の `Exception` クラスないしは `Exception` クラスを継承する例外が発生したら、発生した例外のエラーメッセージを出力する。 `KeyboardInterrupt` など、 `Exception` クラスないしは `Exception` クラスを継承する例外以外の例外が発生したら、 `Error!` と出力する。例外が発生しなかったら、 `No error` と出力する。例外が発生してもしなくても、最後の `finally` 節によって、 `Always executed` を出力する。

実際には必要に応じて例外処理を組み合わせながら、求める処理を実行する。

```
# 読み込み権限でファイルを開いて、書き込みをする
def write_to_file(message):
    try:
        file = open("./tmp.txt")
        print("File opened")
        try:
            print("File writing")
            file.write(f"{message}\n")
        except:
            print("Error writing to file")
        finally:
            file.close()
    except:
        print("Error opening file")

try:
    write_to_file("Hello")
except:
    pass

# 存在しないファイルを開いて、読み込みをする
try:
    file = open("notfound.txt")
except FileNotFoundError:
    print("File Not found")

# ファイルではなくディレクトリを開こうとする
try:
    file = open(".")
except IsADirectoryError:
    print("Is a directory")
```

```
Error opening file
File Not found
Is a directory
```

例えば、上記のサンプルプログラムのように例外処理を組み合わせることで、ファイルの読み書きにおいて発生する例外の処理を記述できる。サンプルプログラム全体の構造を説明すると、まず、`write_to_file`という関数を定義している。続いて、3つの`try`文を記述している。`write_to_file()`関数内では、`try`文の中に`try`文を記述されており、`try`文が入れ子構造になっている。

サンプルプログラムの処理の流れを説明する。

まず、1つ目の`try`文の`try`節で、実引数に`Hello`と文字列を入れて`write_to_file()`関数を呼び出す。`write_to_file()`関数では、まず外側の`try`文の`try`節を実行する。`file = open("./tmp.txt")`の処理で、`tmp.txt`ファイルを開こうとする。`tmp.txt`ファイルが存在しないため、例外が発生し、`try`節の中の`file = open("./tmp.txt")`より後に記述された処理が実行されず、`except`節の中の処理である`print("Error opening file")`を実行して、`Error opening file`を実行する。

続いて、2つ目の`try`文を実行する。`try`節で、`file = open("notfound.txt")`の処理で、`notfound.txt`ファイルを開こうとする。`notfound.txt`ファイルが存在しないため、`FileNotFoundError`クラスの例外が発生する。`except`節では、`except FileNotFoundError:`と記述されており、`FileNotFoundError`クラスの例外を受け取り、処理を実行する。`try`節で、`FileNotFoundError`クラスの例外が発生するため、`except`節の中の処理である`print("File Not found")`を実行して、`File Not found`を出力する。

続いて、3つ目の`try`文を実行する。`try`節で、`file = open(".")`の処理で、ファイルではなくディレクトリを開こうとする。`open()`関数はファイルを開く関数であるにもかかわらずディレクトリを開こうとしたため、`IsADirectoryError`クラスの例外が発生する。`except`節では、`except IsADirectoryError:`と記述されており、`IsADirectoryError`クラスの例外を受け取り、処理を実行する。`try`節で、`IsADirectoryError`クラスの例外が発生するため、`except`節の中の処理である`print("Is a directory")`を実行して、`Is a directory`を出力する。

例外の送出

```python
# raise文
def divide(num1, num2):
    if type(num1) is not int or type(num2) is not int:
        raise TypeError(f"{num1} and {num2} must be integers")
    elif num2 == 0:
        raise ZeroDivisionError
    num1 / num2

divide(10, 2)
divide(10, "2")
```

ここまでの説明に使用した例外はプログラムの実行時に勝手に発生するものだったが、自分で例外を発生させることもできる。自分で例外を発生させるには、`raise`を上記のサンプルプログラムのように用いることで実現できる。`TypeError`の場合は例外インスタンスを`raise`の引数に与えている。`ZeroDivisionError`の場合は例外クラスを`raise`の引数に与えているが、暗黙的に`ZeroDivisionError`の引数無しのコンストラクタが呼ばれ、例外インスタンスを与えたときと同様の処理が行われている。

```python
try:
    10 / 0
except ZeroDivisionError:
    print("Error occured")
    raise
print("Continue processing")
```

```
Error occured
```

`raise` の処理を実行すると、例外が発生するため、発生した例外を `try` 文で受け取らない限り、プログラムは終了する。 上記のサンプルプログラムでは、 `try` 文の後続の処理である `print("Continue processing")` を実行するということがなくなる。

`raise` を使うことで、 `try` 文の中に `try` 文を入れ子にした際の、内側の `try` で発生した例外を外側の `try` 文に送出できる。

```
try:
    try:
        10 / 0
    except ZeroDivisionError:
        print("Error, can't divide by zero")
        raise
    print("Continue processing")
except ZeroDivisionError as exc:
    raise RuntimeError("Error occured") from exc
```

```
Error, can't divide by zero
```

上記のサンプルプログラムでは、内側の `try` 文で `ZeroDivisionError` という例外が発生した場合、外側の `try` 文では `RuntimeError` という例外として送出している。

サンプルプログラムの処理について説明する。

内側の `try` 文の `try` 節で、 `10 / 0` を実行することで `ZeroDivisionError` クラスの例外が発生する。内側の `try` 文の `except` 節では、 `except ZeroDivisionError:` と記述されているため、 `try` 節で発生した `ZeroDivisionError` クラスの例外を受け取り、 `except` 節の中の処理を実行する。 `print("Error, can't divide by zero")` を実行し、 `Error, can't divide by zero` を出力する。そして、 `raise` を実行する。 `except` 節の中で引数なしの `raise` を実行すると、 `except` 節が受け取った例外を発生させる。サンプルプログラムでは、内側の `try` 文の `except` 節は、 `ZeroDivisionError` クラスの例外を受け取っているため、 `raise` でも `ZeroDivisionError` クラスの例外を発生させる。

内側の `try` 文の `except` 節で発生した例外は、外側の `try` 文の `try` 節にある `try` 文で発生した `ZeroDivisionError` クラスの例外として処理を進めるため、 `print("Continue processing")` は実行されない。

外側の `try` 文の `except` 節では、 `except ZeroDivisionError as exc:` と記述されているため、外側の `try` 文の `try` 節で発生した `ZeroDivisionError` クラスの例外を受け取り、受け取った例外オブジェクトを `exc` に格納して、 `except` 節の中の処理である `raise RuntimeError("Error occured") from exc` を実行する。 `from` を使うことで、 `RuntimeError("Error occured")` と `exc` 内の例外オブジェクトを連鎖させることができる。連鎖させることで、外側の `try` 文の `except` 節で発生した `RuntimeError` が内側の `try` 文で発生した `ZeroDivisionError` が紐づいていることをTracebackから読み取ることができる。

内側の `try` 文で発生した例外を送出して、外側の `try` 文の例外と連鎖させることで、 `try` 文を入れ子構造にしたときに内側の `try` 文のどの例外が原因で、外側の `try` 文で例外が発生したか知ることができる。

```
try:
    try:
        10 / 0
    except ZeroDivisionError:
        print("Error, can't divide by zero")
        raise
    print("Continue processing")
except ZeroDivisionError as exc:
    raise RuntimeError("Error occured") from None
```

```
Error, can't divide by zero
```

`form` に続く式には例外だけでなく、 `None` を指定することもできる。 `None` を指定することで、内側の `try` 文で発生した例外は無視できる。

独自例外の定義

例外のクラスは自分で作ることができる。 以下のサンプルプログラムをみてみよう。

```
# カスタム例外の作成
class NegativeValueError(ValueError):
    pass

class TooBigException(ValueError):
    pass

class NotIntValueError(TypeError):
    pass

def fibonatti(n):
    if type(n) is not int:
        raise NotIntValueError(f"{n} is not integer")
    elif n < 0:
        raise NegativeValueError(f"{n} is negative")
    elif n > 35:
        raise TooBigException(f"{n} is too big")

    if n == 0: return 0
    if n == 1: return 1
    return fibonatti(n - 1) + fibonatti(n - 2)

try:
    fibonatti(36)
except Exception as e:
    print(e)

try:
    fibonatti(-10)
except Exception as e:
    print(e)

try:
    fibonatti(0.5)
except Exception as e:
    print(e)

try:
    fibonatti("10")
except Exception as e:
    print(e)

try:
    try:
        fibonatti(-10)
    except NegativeValueError:
        print("Do something for handling error")
        raise
except Exception as e:
    print("Error")
    print(e)
```

```
36 is too big
-10 is negative
0.5 is not integer
10 is not integer
Do something for handling error
Error
-10 is negative
```

例外のクラスは自分で作ることで、例外の理由を型で表現することができるため、例外処理をしやすくなる。

上記のサンプルプログラムでは、まず、3つの例外クラスを定義している。

1つ目の例外クラスは、`ValueError` クラスを継承した `NegativeValueError` クラスを定義している。定義したクラスの内部には何も実装を行わないため、何も処理を行わない `pass` を記述する。何も実装しないからといって `class NegativeValueError(ValueError):` 以降に何も記述しないと、構文エラーになるため、何も実装しない場合は、クラス内部の実装として何も処理を行わない `pass` を記述する。

2つ目の例外クラスは、`ValueError` クラスを継承した `TooBigException` クラスを定義している。定義したクラスの内部には何も実装を行わないため、1つ目と同様に、何も処理を行わない `pass` を記述する。

3つ目の例外クラスは、`TypeError` クラスを継承した `NotIntValueError` クラスを定義している。定義したクラスの内部には何も実装を行わないため、1つ目と同様に、何も処理を行わない `pass` を記述する。

続いて、`fibonatti()` 関数を定義している。この関数では、フィボナッチ数を再帰的に呼び出す関数である。さらに、`fibonatti()` 関数内に記述されている初めの `if` 文で、`fibonatti()` 関数に渡ってきた値によって、引数にとったエラーメッセージとともに例外を返す処理を記述している。`n` がint型でない場合は、`raise NotIntValueError(f"{n} is not integer")` で例外を返している。`n` が `0` より小さい値（負の値）であれば、`raise NegativeValueError(f"{n} is negative")` で例外を返している。`n` が `35` より大きい値であれば、`raise TooBigException(f"{n} is too big")` を返している。

続いて、`fibonatti()` 関数の定義のあとに記述された4つの `try` 文の `try` 節にて、`fibonatti()` 関数を実行している。`fibonatti()` 関数の実引数には、それぞれ、`36` 、`-10` 、`0.5` 、`"10"` を使用しており、いずれも `fibonatti()` 関数内部で今回独自に定義した例外が発生するため、呼び出し元には例外が返される。独自に定義した例外、いずれも `Exception` と継承関係にあるため、4つの `try` 文の `except` 節の中の処理を実行し、`fibonatti()` 関数で定義したエラーメッセージを出力する。

最後に、サンプルプログラムには、入れ子構造の `try` 文が記述されている。内側の `try` 文の `try` 節で `fibonatti(-10)` を実行するため、独自定義した `NegativeValueError` クラスの例外が発生する。内側の `try` 文の `except` 節では、`except NegativeValueError:` と記述されており、独自定義した `NegativeValueError` クラスの例外を受け取ることができる。このように、`except` 節で受け取る例外のクラスに独自定義した例外クラスを使用することができる。内側の `try` 文の `try` 節で `NegativeValueError` クラスの例外が発生するため、内側の `try` 文の `except` 節の中の処理を実行する。まず、`print("Do something for handling error")` を実行し、`Do something for handling error` を出力する。続いて、`raise` を実行し、`NegativeValueError` クラスの例外オブジェクトを送出する。内側の `try` 文で、`NegativeValueError` クラスの例外オブジェクトを送出したことで、外側の `try` 文の `except` 節が例外オブジェクトを受け取り、`except` 節の中の処理を実行する。まず、`print("Error")` を実行し、`Error` を出力する。続いて、`print(e)` を実行する。`e` には、内側の `try` 文で送出した `NegativeValueError` クラスの例外オブジェクトが格納されている。そのため、送出された `NegativeValueError` クラスの例外オブジェクトの中のエラーメッセージを出力する。

独自例外は基本的には `Exception` クラスを継承するか、`Exception` クラスを継承しているクラスを継承する。つまりは `Exception` クラスの子孫クラスにしておくのが一般的である。

問題1

問題文

数値のリスト `nums` を引数に取り、for文で先頭3つの要素の和を計算して出力する関数 `func` の中の処理を書け。

解答の雛形

```
def func(nums):
  # ここに解答を入力

def func_wrapper(nums):
  try:
    func(nums)
  except:
    print("Invalid input")

func_wrapper([1, 10, 100])
func_wrapper([1, 10, "a"])
func_wrapper([1, 10])
```

問題2

問題文

問題1で作成した関数 `func` に対して、正しく例外処理を加えよ。

例外は次のように処理せよ：

- リストの要素数が3未満の場合
  - `IndexError` 例外が発生するので、 `The length of list is too short.` と出力せよ。
- リストに数値以外の要素が含まれていた場合
  - `TypeError` 例外が発生するので、 `All elements of the list must be of int only.` と出力せよ。
- それ以外のエラーが発生した場合
  - 例外をそのまま出力せよ。
- エラーが発生しなかった場合
  - `No error occured.` と出力せよ。

最後に、例外が発生してもしていなくても、 `Execution finished.` と出力せよ。

解答の雛形

```
def func(nums):
  # ここに解答を入力

func([1, 10, 100])
func([1, 10, "a"])
func([1, 10])
```

問題3

問題文

文字列のリスト `strs` を引数に取り、リスト内の文字列を先頭から連結して出力する関数 `func2` がある。

リスト `strs` の中に `str` 型以外の要素が入っていた場合、 `Error occured.` とだけ出力されて実行が終了するように、 `func2` の中に例外処理を書き加えよ。

解答の雛形

```
def run(strs):
  try:
    func2(strs)
  except Exception:
    print("Error occured.")

def func2(strs):
  res = ""
  # ここに解答を入力
  print(res)

run(["apple", "banana", "orange"])
run(["apple", "banana", 3])
```

問題4

問題文

問題3と同じ `func2` 関数があるが、今回は `func2` 関数は `decolator` 関数を経由して実行される。

リスト `strs` の中に `str` 型以外の要素が入っていた場合、 `Error occured.` とだけ出力されて実行が終了するように、 `func2` の中に例外処理を書き加えよ。

解答の雛形

```
def run(strs):
  try:
    decolator(strs)
  except Exception as e:
    print("Error occured.")

def decolator(strs):
  print("--- start ---")
  func2(strs)
  print("--- end ---")

def func2(strs):
  res = ""
  # ここに解答を入力
  print(res)

run(["apple", "banana", "orange"])
run(["apple", "banana", 3])
```

問題5

問題文

数値のリストを入力に受け取り、リストの中の要素の最大値を出力するプログラムがある。

このプログラムにエラーが発生した場合、例外をキャッチして何行目でエラーが発生したかを出力して終了するように書き加えよ。

ただし、例えば2行目でエラーが発生した場合 `Error occured in 2 line.` のように出力せよ。

解答の雛形

```
import traceback

try:
  nums = list(map(int, input().split()))
  max_value = nums[0]
  for i in range(1, len(nums)):
    max_value = max(max_value, nums[i])
  print(max_value)
except Exception as e:
  # ここに解答を入力
```

10章: ソフトウェア品質とコーディング規約

ソフトウェア品質モデルの規格

日本工業規格であるJIS X 25010:2013では、ソフトウェア品質（software quality）を「明示された状況下で使用されたとき，明示的ニーズ及び暗黙のニーズをソフトウェア製品が満足させる度合い。」と定義している。ソフトウェア品質は、利用者がソフトウェア使用する際に操作するユーザインタフェースに関する品質から、開発者がソフトウェアを保守する際の品質まで、幅広い概念を扱っている。本節では、JIS X 25010:2013、および、JIS X 25010:2013がベースとした国際規格であるISO/IEC 25010:2011の概要を説明する。

ISO/IEC 25010:2011の前進は、1991年に刊行された国際標準化機構(International Organization for Standardization; ISO）および国際電気標準会議（International Electrotechnical Commission; IEC）のISO/IEC 9126である。ISO/IEC 9126では、6つの品質特性と27の品質副特性から品質モデルを定義した。品質モデル（quality model）は、「品質要求事項の仕様化及び品質評価に対する枠組みを提供する特性の定義された集合及び特性間の関係の集合」と定義されており、何をもってソフトウェア品質とするかを整理したものである。また、ソフトウェア品質特性（software quality characteristic）は「ソフトウェア品質に影響を及ぼすソフトウェア品質属性の分類」と定義されている。ソフトウェア品質特性は、複数の階層の副特性に詳細化され、最終的にはソフトウェア品質属性にまで詳細化することができる。

1991年以降の情報通信技術の発展などを踏まえて、2011年にISO/IEC 25010:2011が刊行されている。ISO/IEC 25000 から 25099 までの番号は、前述のISO/IEC 9126シリーズとISO/IEC 14598シリーズを統合した Software Product Quality Requirements and Evaluation (SQuaRE) シリーズを制定するために用意されており、ISO/IEC 25010:2011はそのうちの一つとして品質モデルに関する内容を記述している。また、JIS X 25010:2013は、ISO/IEC 25010:2011の日本語版になっている。

ソフトウェア品質モデル

本節では、JIS X 25010:2013、および、ISO/IEC 25010:2011において定義されている品質モデルを解説することで、ソフトウェア品質とは何か、また、ソフトウェア品質を高めるために何をする必要があるかを説明する。なお、JIS X 25010:2013、および、ISO/IEC 25010:2011を総称して、本資料ではSQuaREシリーズと呼ぶこととする。

SQuaREシリーズでは、ソフトウェア品質モデルを製品品質モデル、利用時の品質モデル、データ品質モデルの3種類に分類している。製品品質モデルは、ソフトウェアの静的特徴及びコンピュータシステムの動的特徴に関係する八つの特性から構成されるモデルである。利用時の品質モデルは、特定の使用状況において製品が使用されるときに、利用者とソフトウェアの対話の成果に関係する五つの特性から構成されるモデルである。データ品質モデルは、ISO/IEC 25012:2011で定義されているモデルであり、プログラムに関する品質から少し離れた話題となるので、本授業では割愛する。

製品品質モデルは機能適合性・性能効率性・互換性・使用性・信頼性・セキュリティ・保守性・移植性の8つの品質特性、利用時の品質モデルは有効性・効率性・満足性・リスク回避性・利用状況網羅性の5つの品質特性から構成される。各品質特性の定義について説明する。

機能適合性（functional suitability）は、明示された状況下で使用するとき、明示的ニーズ及び暗黙のニーズを満足させる機能を、製品又はシステムが提供する度合いを指す。さらに、機能適合性は、機能完全性、機能正確性、機能適切性の3つの品質副特性から構成される。機能完全性（functional completeness）は、機能の集合が明示された作業及び利用者の目的の全てを網羅する度合いを指す。機能正確性（functional correctness）は、正確さの必要な程度での正しい結果を、製品又はシステムが提供する度合いを指す。機能適切性（functional appropriateness）は、明示された作業及び目的の達成を、機能が促進する度合いを指す。

性能効率性（performance efficiency）は、明記された状態（条件）で使用する資源の量に関係する性能の度合いを指す。さらに、性能効率性は、時間効率性、資源効率性、容量満足性の3つの品質副特性から構成される。時間効率性（time behaviour）は、製品又はシステムの機能を実行するとき、製品又はシステムの応答時間及び処理時間、並びにスループット速度が要求事項を満足する度合いを指す。資源効率性（resource utilization）は、製品又はシステムの機能を実行するとき、製品又はシステムで使用される資源の量及び種類が要求事項を満足する度合いを指す。容量満足性（capacity）は、製品又はシステムのパラメータの最大限度が要求事項を満足させる度合いを指す。

互換性（compatibility）は、同じハードウェア環境又はソフトウェア環境を共有する間、製品、システム又は構成要素が他の製品、システム又は構成要素の情報を交換することができる度合い、及び／又はその要求された機能を実行することができる度合いを指す。さらに、互換性は、共存性、相互運用性の2つの品質副特性から構成される。共存性（co-existence）は、その他の製品に有害な影響を与えずに、他の製品と共通の環境及び資源を共有する間、製品が要求された機能を効率的に実行することができる度合いを指す。相互運用性（interoperability）は、二つ以上のシステム、製品又は構成要素が情報を交換し、既に交換された情報を使用することができる度合いを指す。

使用性（usability）は、明示された利用状況において、有効性、効率性及び満足性をもって明示された目標を達成するために、明示された利用者が製品又はシステムを利用することができる度合いを指す。さらに、使用性は、適切度認識性、習得性、運用操作性、ユーザエラー防止性、ユー

ザインタフェース快美性、アクセシビリティの6つの品質副特性から構成される。適切度認識性（appropriateness recognizability）は、製品又はシステムが利用者のニーズに適切であるかどうかを利用者が認識できる度合いを指す。習得性（learnability）は、明示された利用状況において、有効性、効率性、リスク回避性及び満足性をもって製品又はシステムを使用するために明示された学習目標を達成するために、明示された利用者が製品又はシステムを利用できる度合いを指す。運用操作性（operability）は、製品又はシステムが、それらを運用操作しやすく、制御しやすくする属性をもっている度合いを指す。ユーザエラー防止性（user error protection）は、利用者が間違いを起こすことをシステムが防止する度合いを指す。ユーザインタフェース快美性（user interface aesthetics）は、ユーザインタフェースが、利用者にとって楽しく、満足のいく対話を可能にする度合いを指す。アクセシビリティ（accessibility）は、製品又はシステムが、明示された利用状況において、明示された目標を達成するために、幅広い範囲の心身特性及び能力の人々によって使用できる度合いを指す。

信頼性（reliability）は、明示された時間帯で、明示された条件下に、システム、製品又は構成要素が明示された機能を実行する度合いを指す。さらに、信頼性は、成熟性、可用性、障害許容性（耐故障性）、回復性の4つの品質副特性から構成される。成熟性（maturity）は、通常の運用操作の下で、システム、製品又は構成要素が信頼性に対するニーズに合致している度合いを指す。可用性（availability）は、使用することを要求されたとき、システム、製品又は構成要素が運用操作可能及びアクセス可能な度合いを指す。障害許容性（耐故障性）（fault tolerance）は、ハードウェア又はソフトウェア障害にもかかわらず、システム、製品又は構成要素が意図したように運用操作できる度合いを指す。回復性（recoverability）は、中断時又は故障時に、製品又はシステムが直接的に影響を受けたデータを回復し、システムを希望する状態に復元することができる度合いを指す。

セキュリティ（security）は、人間又は他の製品若しくはシステムが，認められた権限の種類及び水準に応じたデータアクセスの度合いをもてるように，製品又はシステムが情報及びデータを保護する度合いを指す。さらに、セキュリティは、機密性、インテグリティ、否認防止性、責任追跡性、真正性の5つの品質副特性から構成される。機密性（confidentiality）は、製品又はシステムが、アクセスすることを認められたデータだけにアクセスすることができることを確実にする度合いを指す。インテグリティ（integrity）は、コンピュータプログラム又はデータに権限をもたないでアクセスすること又は修正することを、システム、製品又は構成要素が防止する度合いを指す。否認防止性（non-repudiation）は、事象又は行為が後になって否認されることがないように、行為又は事象が引き起こされたことを証明することができる度合いを指す。責任追跡性（accountability）は、実体の行為がその実体に一意的に追跡可能である度合いを指す。真正性（authenticity）は、ある主体又は資源の同一性が主張したとおりであることを証明できる度合いを指す。

保守性（maintainability）は、意図した保守者によって、製品又はシステムが修正することができる有効性及び効率性の度合いを指す。さらに、保守性は、モジュール性、再利用性、解析性、修正性、試験性の5つの品質副特性から構成される。モジュール性（modularity）は、一つの構成要素に対する変更が他の構成要素に与える影響が最小になるように、システム又はコンピュータプログラムが別々の構成要素から構成されている度合いを指す。再利用性（reusability）は、一つ以上のシステムに、又は他の資産作りに、資産を使用することができる度合いを指す。解析性（analysability）は、製品若しくはシステムの一つ以上の部分への意図した変更が製品若しくはシステムに与える影響を総合評価すること、欠陥若しくは故障の原因を診断すること、又は修正しなければならない部分を識別することが可能であることについての有効性及び効率性の度合いを指す。修正性（modifiability）は、欠陥の取込みも既存の製品品質の低下もなく、有効的に、かつ、効率的に製品又はシステムを修正することができる度合いを指す。試験性（testability）は、システム、製品又は構成要素について試験基準を確立することができ、その基準が満たされているかどうかを決定するために試験を実行することができる有効性及び効率性の度合いを指す。

移植性（portability）は、一つのハードウェア、ソフトウェア又は他の運用環境若しくは利用環境からその他の環境に、システム、製品又は構成要素を移すことができる有効性及び効率性の度合いを指す。さらに、移植性は、適応性、設置性、置換性の3つの品質副特性から構成される。適応性（adaptability）は、異なる又は進化していくハードウェア、ソフトウェア又は他の運用環境若しくは利用環境に、製品又はシステムが適応できる有効性及び効率性の度合いを指す。設置性（installability）は、明示された環境において、製品又はシステムをうまく設置及び／又は削除できる有効性及び効率性の度合いを指す。置換性（replaceability）は、同じ環境において、製品が同じ目的の別の明示された製品と置き換えることができる度合いを指す。

有効性（effectiveness）は、明示された目標を利用者が達成する上での正確さ及び完全さの度合いを指す。

効率性（efficiency）は、利用者が特定の目標を達成するための正確さ及び完全さに関連して、使用した資源の度合いを指す。

満足性（satisfaction）は、製品又はシステムが明示された利用状況において使用されるとき、利用者ニーズが満足される度合いを指す。さらに、満足性は、実用性、信用性、快感性、快適性の4つの品質副特性から構成される。実用性（usefulness）は、利用の結果及び利用の影響を含め、利用者が把握した目標の達成状況によって得られる利用者の満足の度合いを指す。信用性（trust）は、利用者又は他の利害関係者がもつ、製品又はシステムが意図したとおりに動作するという確信の度合いを指す。快感性（pleasure）は、個人的なニーズを満たすことから利用者が感じる喜びの度合いを指す。快適性（comfort）は、利用者が（システム又はソフトウェアを利用する時の）快適さに満足する度合いを指す。

リスク回避性（freedom from risk）は、製品又はシステムが、経済状況、人間の生活又は環境に対する潜在的なリスクを緩和する度合いを指す。さらに、リスク回避性は、経済リスク緩和性、健康・安全リスク緩和性、環境リスク緩和性の3つの品質副特性から構成される。経済リスク緩和性（economic risk mitigation）は、意図した利用状況において、財政状況、効率的運用操作、商業資産、評判又は他の資源に対する潜在的なリスクを、製品又はシステムが緩和する度合いを指す。健康・安全リスク緩和性（health and safety risk mitigation）は、意図した利用状況に

おいて、製品又はシステムが人々に対する潜在的なリスクを緩和する度合いを指す。環境リスク緩和性（environmental risk mitigation）は、意図した利用状況において、環境に対する潜在的なリスクを製品又はシステムが軽減する度合いを指す。

利用状況網羅性（context coverage）は、明示された利用状況及び当初明確に識別されていた状況を超越した状況の両方の状況において、有効性、効率性、リスク回避性及び満足性を伴って製品又はシステムが使用できる度合いを指す。さらに、利用状況網羅性は、利用状況完全性、柔軟性の2つの品質副特性から構成される。利用状況完全性（context completeness）は、明示された全ての利用状況において、有効性、効率性、リスク回避性及び満足性を伴って製品又はシステムが使用できる度合いを指す。柔軟性（flexibility）は、要求事項の中で初めに明示された状況を逸脱した状況において、有効性、効率性、リスク回避性及び満足性を伴って製品又はシステムが使用できる度合いを指す。

...

本授業では、主に機能適合性、性能効率性、保守性について学ぶ。本科目の演習問題において、与えられた入力から期待される出力を生成できるようにプログラムを作ることは、機能適合性の高いプログラムの作成について学ぶこととなる。また、本科目の演習問題の一部では、与えられた時間内に計算を終えることが求められるケースがあり、そのようなケースでは、性能効率性の高いプログラムの作成について学ぶこととなる。本章では、良いプログラムの記述方法を学ぶことで、保守性の高いプログラムの作成に関して学ぶ。

コーディング規約

コーディング規約とは、プログラムを記述する際に守るべきルールである。例えば、Pythonのコーディング規約のルールの1つの例として、インデントのレベルを1つ下げるときは、空白4文字を使うというルールが挙げられる。コーディング規約を制定して、ソフトウェア開発者がコーディング規約を遵守することで、プログラムの書き方に一貫性が生まれ、また、多くの開発者の共通認識として読みやすいと感じるプログラムを記述することができる。その結果、第三者がプログラムを読む際に、より速く正確にプログラムの内容を理解することができ、また、開発者がプログラムを記述する際も、複数存在する記述方法から一つ選ぶ手間を省くことができる。これによって、ソフトウェア品質における、保守性、特に、解析性・修正性・試験性を高めることができる。

本節ではPythonにおいて最も著名なコーディング規約であるPEP 8を解説する。PEP 8はPythonの公式サイト内の https://www.python.org/dev/peps/pep-0008/ において公開されている。公式サイトのドキュメントは英語で記述されており、有志が https://pep8-ja.readthedocs.io/ja/latest/ にて日本語版を公開している。

以降で、PEP 8のコーディング規約の中から、いくつかのルールを抜粋して解説する。

インデント

インデントのレベルを1つ下げるときは、空白4文字を使う。関数を定義する際や呼び出す際に、仮引数もしくは実引数の数が多く、途中で改行を入れる場合は、次の例のように開き括弧の位置に合わせてインデントするか、もしくは、後続のプログラムと区別できるようにインデントする。

```
# コーディング規約を遵守している例

# 開き括弧に揃える
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# 開き括弧の後ろに何も書かない場合は、インデントのレベルを1つ下げる
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)

# 引数と後続のプログラムを区別するため、さらにインデントのレベルを1つ追加で下げる
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

```
# コーディング規約を遵守していない例

# 改行後の引数の位置を開き括弧に合わせない場合は、開き括弧の後ろに引数を書いてはいけない
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# 引数と引数以降のプログラムの区別ができなくなるため、引数部分のインデントのレベルを1つ下げなければならない
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

二項演算子の改行位置

長い数式を記述する際に、過去には二項演算子の後ろで改行するスタイルが推奨されてきた。しかし、下の例のように演算子の位置が散らばってしまうため、数式が読みにくくなるという問題がある。それに対して、数学者や数学の本の出版社は、二項演算子の前で改行するスタイルを推奨している。後者のほうが演算子の位置が揃うため、数式が読みやすくなる。Pythonにおいても、同様に二項演算子の前で改行するスタイルが推奨されている。

```
# コーディング規約を遵守している例

# 演算子が見やすくなっている
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

```
# コーディング規約を遵守していない例

# 演算子が見にくくなっている
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

式や文中の空白文字

基本的にカンマの後に空白を入れる必要があるが、それ以外の場所には、空白を入れないことが推奨されている。いくつかの具体例を通して、空白を入れるべき場所および入れないべき場所を理解しよう。

次の例のように、丸括弧(`(`)、大括弧(`[`)、波括弧(`{`)のはじめの直後と、終わりの直前に空白を入れてはいけない。

```
# コーディング規約を遵守している例
spam(ham[1], {eggs: 2})

# コーディング規約を遵守していない例
spam( ham[ 1 ], { eggs: 2 } )
```

次の例のように、末尾のカンマ(`,`)と、その後に続く閉じ括弧の間に空白を入れてはいけない。

```
# コーディング規約を遵守している例
tuple1 = (0,)

# コーディング規約を遵守していない例
tuple1 = (0, )
```

次の例のように、カンマ(`,`)やセミコロン(`;`)、コロン(`:`)の直前に空白を入れてはいけない。なお、セミコロンを使うと、複数行のステートメントを1行で記述することができる。

```
# コーディング規約を遵守している例
if x == 4: print(x, y); x, y = y, x

# コーディング規約を遵守していない例
if x == 4 : print(x , y) ; x , y = y , x
```

次の例のように、関数呼び出しの引数リストをはじめる開き括弧の直前に空白を入れてはいけない。

```
# コーディング規約を遵守している例
spam(1)

# コーディング規約を遵守していない例
spam (1)
```

次の例のように、インデックスやスライスの開き括弧の直前に空白を入れてはいけない。

```
# コーディング規約を遵守している例
dct['key'] = lst[index]

# コーディング規約を遵守していない例
dct ['key'] = lst [index]
```

次の例のように、演算子を揃えるために、演算子の周囲に2つ以上のスペースを入れてはいけない。

```
# コーディング規約を遵守している例
x = 1
y = 2
long_variable = 3

# コーディング規約を遵守していない例
x             = 1
y             = 2
long_variable = 3
```

コメント

次の例のように、プログラムの内容と矛盾するコメントを記述してはいけない。あるタイミングでプログラムの内容を説明するコメントを記載した後に、プログラムだけを変更すると、プログラムとコメントの矛盾が起こってしまう。プログラムと同様にコメントもメンテナンスする必要があるため、不必要なコメントを書かないように注意をすべきである。

なお、コメントは英語で記述することが推奨されているが、プログラムを読む者全てが理解している言語で記述する分には問題がないため、本科目では日本語を採用している。

```
# コーディング規約を遵守している例

# a と b を足した結果を返す関数
def func(a, b):
    return a + b


# コーディング規約を遵守していない例

# a と b を足した結果を返す関数
def func(a, b):
    return a * b
```

命名規約

クラスの名前はCapWords方式で命名することが推奨されている。CapWords方式とは、名前の先頭と単語の切れ目の先頭を大文字にする方式である。パスカルケース (pascal case) とも呼ばれる。例えば、`TextFileReader`はCapWords方式になっているが、`textFileReader`や`text_file_reader`はCapWords方式になっていない。

関数や変数の名前は全て小文字にして、必要に応じてアンダースコア(`_`)で単語間を区切ることが推奨されている。例えば、`read_text_file`が該当する例である。

プログラム中で変化しない値を変数に代入したもの（つまり、再代入しない変数）を定数と呼ぶが、定数は全て大文字にして、単語間をアンダースコア(`_`)で区切る。例えば、`MAX_VALUE`が該当する例である。

コーディング規約の違反例

コーディング規約に違反すると、プログラムの内容を理解することが難しくなるということを体感するため、以下のプログラムを読んで、実行せずに実行結果がどうなるかを考えてみよう。

```
def trAin(
            abc = '#' ,
O = 1 ,
            変数A = False ) :
    for i \
        in range \
            ( O ) : # なんかする
        print ('['
    ,
    end = ''         )
        print(
abc
                *
3,end = ''
)
        print(
    ']' ,
    end=''
    )
        if (i<
            O -1 or
        not 変数A
        ):
            print(
                    '-'
    ,end=''
                )
        elif \
変数A:




            print()

trAin(O = 5)
```

```
[###]-[###]-[###]-[###]-[###]-
```

プログラムの内容を理解することが非常に難しいと実感できたはずだ。下のプログラムは、PEP 8に遵守するように上のプログラムを書き直したものだ。改めて実行結果がどうなる考えてみよう。

```
def train(mark='#', repeat=1, is_tail=False):
    for i in range(repeat):
        print('[', end='')
        print(mark * 3, end='')
        print(']', end='')

        # is_tailがFalseで、かつ、最後の繰り返しの場合は「-」を表示する
        if i < repeat - 1 or not is_tail:
            print('-', end='')
        elif is_tail:
            print()

train(repeat=5)
```

```
[###]-[###]-[###]-[###]-[###]-
```

問題1

問題文

空白1文字で区切った整数列が与えられ、その平均値を計算するプログラムを作る必要がある。 すでに、ある開発者が下のような正常に動作するプログラムを作成しているが、残念ながらPEP 8に違反したプログラムになっている。 そこで、以下のプログラムがPEP 8を遵守できるように、以下のプログラムに空白( )だけを補ったプログラムを作成せよ。

```
nums=list(map(int,input().split(' ')))

sum=0
for num in nums:
    sum+=num
print(sum/len(nums))
```

解答の雛形

```
nums=list(map(int,input().split(' ')))

sum=0
for num in nums:
    sum+=num
print(sum/len(nums))
```

問題2

問題文

文字列(S)と整数(A)が1つずつ与えられる。 文字列を０文字目から A 文字飛ばしで１文字ずつ読み取った上で、結合して逆順に出力するプログラムがある。 しかし空白文字がPEP 8が示すよりも非常に多くなっている。 そこで、以下のプログラムがPEP 8を遵守できるように、以下のプログラムから適度に空白( )を取り除いたプログラムを作成せよ。

```
S           =  input  ()
A          =  int(  input ()  )
result   = ""
for  i  in  range  ( 0,  len  ( S ) , A + 1 ):
    result  +=  S  [  i  ]
print  ( result [  :  : -1 ] )
```

解答の雛形

```
S           =  input  ()
A          =  int(  input ()  )
result   = ""
for  i  in  range  ( 0,  len  ( S ) , A + 1 ):
    result  +=  S  [  i  ]
print  ( result [  :  : -1 ] )
```

問題3

問題文

二次方程式の解の1つを計算する関数があるのだが、形が崩れている。 そのため、PEP 8を遵守できるように以下のプログラムを修正せよ。

```
def solve_quadratic_equation(
    a, b, c) :
    top = - b + (
        b **
        2 -
        4  *
        a *
        c
    ) ** 0.5
    bottom =  2 * a
    return top  / bottom

a,  b, c = map(int, input().split() )
print(
solve_quadratic_equation( a, b , c) )
```

解答の雛形

```
def solve_quadratic_equation(
    a, b, c) :
    top = - b + (
        b **
        2 -
        4  *
        a *
        c
    ) ** 0.5
    bottom =  2 * a
    return top  / bottom

a,  b, c = map(int, input().split() )
print(
solve_quadratic_equation( a, b , c) )
```

問題4

問題文

次のコードの変数名などは、大文字小文字などの命名規則がバラバラである。 そのため、PEP 8を遵守できるように以下のプログラムを修正せよ。

```
MorningGreeting = "Good Morning!!!"

class greeting_class:
    def GREET_AT_MORNING(self):
        print(MorningGreeting)

Greeting = greeting_class()
Greeting.GREET_AT_MORNING()
```

解答の雛形

```
MorningGreeting = "Good Morning!!!"

class greeting_class:
    def GREET_AT_MORNING(self):
        print(MorningGreeting)

Greeting = greeting_class()
Greeting.GREET_AT_MORNING()
```