Reverse Lookup in Python Standard Libraries

逆引き Python 標準ライブラリ

目的別の基本レシピ180+!

大津 真／田中 賢一郎 = 著

便利に使えるライブラリ!

スキルアップの近道

文字列／データ操作、テキスト処理、ファイル操作、数値演算、
ネットワークアクセス、GUIなど、使えるテクニック満載。

インプレス

■ソースコードについて
本書で使用しているソースコードは、以下のURLよりダウンロードできます。
https://book.impress.co.jp/books/1117101049

■正誤表について
正誤表を掲載した場合は、上記のURLのページに表示されます。

執筆時の環境はPyhon 3.6.3です。Pyhonでは、小数点以下のマイナーバージョンは頻繁に更新されており、それにともない標準ライブラリに含まれるモジュールのバージョンも更新されることがあります。バージョン3系の範囲では、下位互換性を無視した内容の大幅な変更はないと思われますが、場合によっては更新によって本書記載の内容と異なる可能性もあることをご了解ください。

まえがき

本書で解説するPythonは、オブジェクト指向のスクリプト型プログラミング言語です。わかりやすくシンプルなステートメントを特徴とし、小規模なユーティリティから、本格的なオブジェクト指向プログラムまで作成可能です。1991年の登場以降、瞬く間に世界的に広く普及し、今ではWindows、macOS、Linux、Raspberry Piなどさまざまなプラットフォームに移植されています。最近では機械学習や、IoTの制御言語としても注目を集めています。

プログラミング言語の人気度の目安となるPYPL（http://pypl.github.io/PYPL.html）の順位は、2017年12月の時点でJavaに続いて2位に位置しています。また、過去5年で最も人気度が向上したプログラミング言語であり、1位のJavaとの差も徐々に縮まってきています。

Pythonには「標準ライブラリ」として、多くのデータ型やモジュールなどが付属しています。それらをうまく活用することが、Pythonプログラミング上達の近道であることは間違いないでしょう。

本書ではPythonの基本的な文法を理解しているユーザーを対象に、標準ライブラリとして用意されている数多くのデータ型、関数、クラスの中から、初中級者が便利に使えるものを中心に基本的な使い方を解説していきます。

Pythonは現在、Python 2.x系からPython 3.x系への移行期になっています。Python 3.xでは、さまざま機能強化が行われ、さらに標準の文字コードがユニコードベースに変更されて日本語の取り扱いも容易になったことから、日本においてもさらなる普及が期待されます。そこで本書ではPython 3.6をベースに解説します。

まず、Chapter 1では、文字列とデータの基本操作について説明します。Chapter2ではリスト、タプル、辞書といったコレクションの取り扱いについて説明します。Chapter 3では正規表現やファイルの読み書きなどテキスト処理に特化したモジュールの操作について説明します。Chapter ４ではOS関連の関数やファイル操作、および日付・時刻の操作について、Chapter 5では数値演算関連、Chapter 6ではインターネット関連のモジュールを説明します。Chapter 7ではGUIプログラミングに欠かせないtkinterモジュールの使い方を中心に説明します。最後の Chapter 8ではそのほかの便利なモジュールをいくつか紹介します。

読者の皆様にとって本書がPythonプログラミングの幅をさらに広げる手助けとなることを願っています。

2018年1月　**執筆者を代表して　　大津 真**

目 次 contents

OSの機能を利用する
ファイル／ディレクトリ／時刻／日付

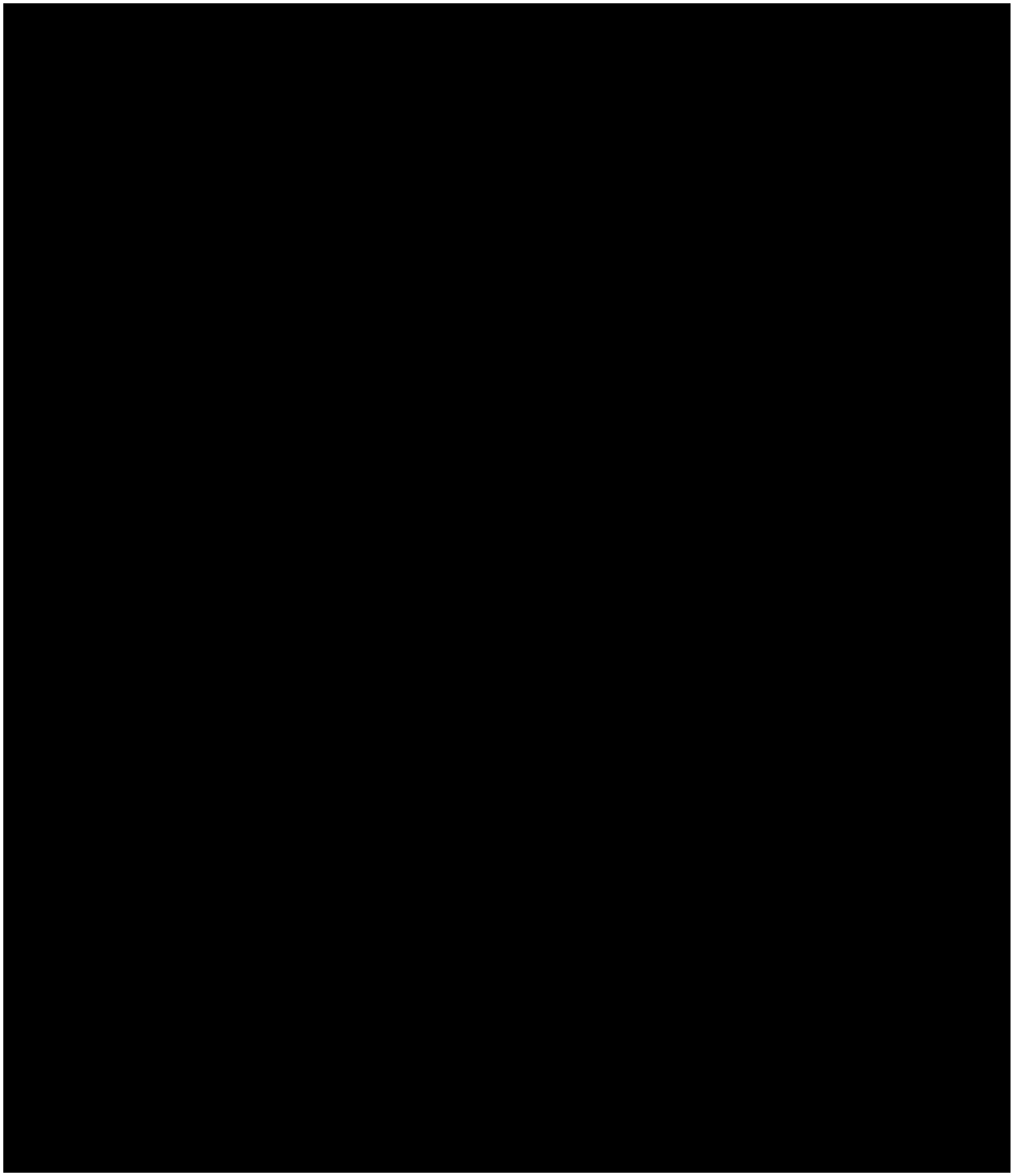
4-1　ファイルやディレクトリの取り扱い

4-2　多少高度なファイル操作

4-3　timeモジュールによる時刻の取り扱い

【執筆分担】　大津真　Introduction～Chapter 03（2-4を除く）
田中賢一郎　Chapter 2-4、Chapter 04～Chapter 08

introduction



Python
環境の構築

Pythonのさまざまなプログラミングテクニックをご紹介する前に、準備段階としてPythonの導入、プログラミング環境の構築などについて説明しておきましょう。すでにPythonを使われている方も復習にご利用ください。

1 | Pythonのインストール

現在Pythonは、バージョン2系（Python 2）とバージョン3系（Python 3）の２種類が広く使用されています。本書ではPython 3を対象に解説しています。そのインストール方法を説明しておきます。

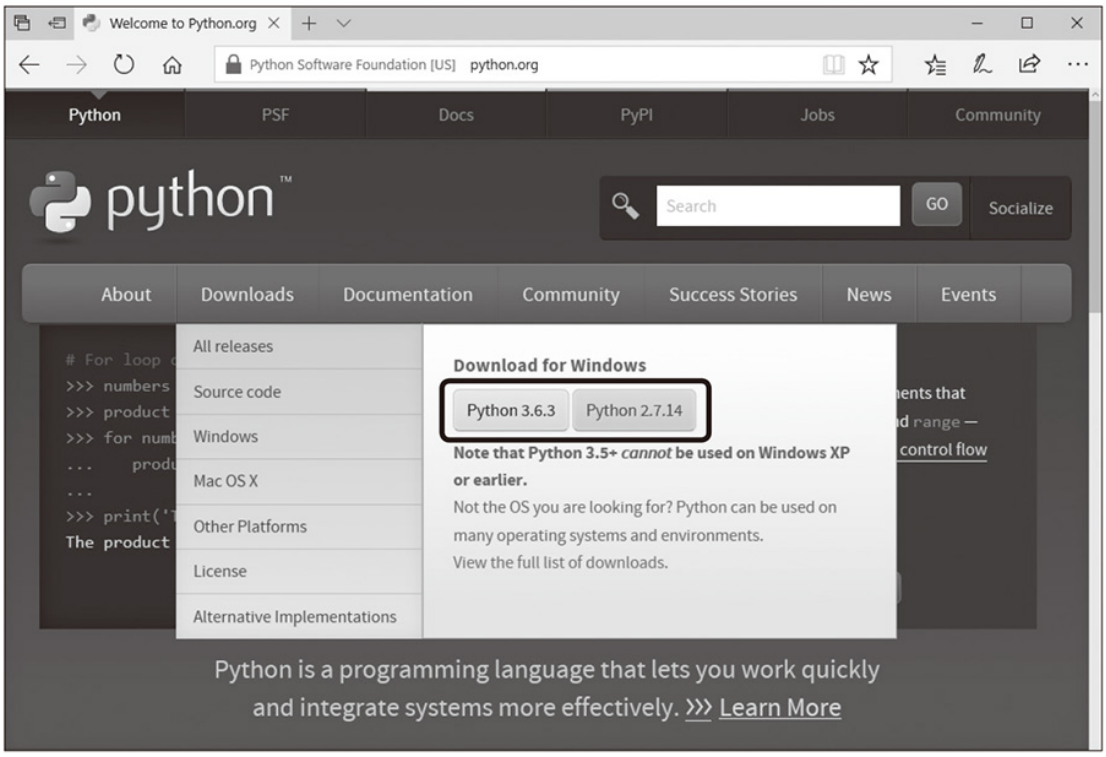
Python 3のインストール

まずは公式のPython 3(本稿執筆時点の最新版はPython 3.6.3）のインストール方法について説明します。

◎Windowsにインストール

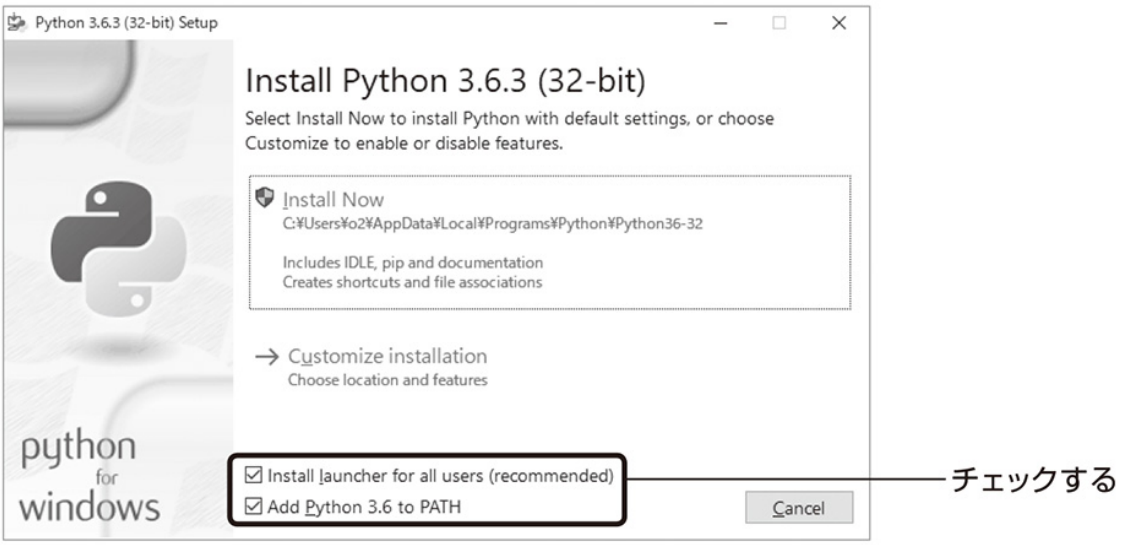
Windowsでのインストール手順は次のようになります。

❶ **Python のオフィシャルサイト（ https://www.python.org ） を開き、「Downloads」メニューから「Python 3.6.x」（執筆時点では3.6.3）を選択します。**



❷　インストーラを起動し、インストールを実行します。

最初の画面では「Install launcher for all users (recommended)」と「Add Python 3.6 to PATH」にチェックをつけ「Install Now」をクリックします。

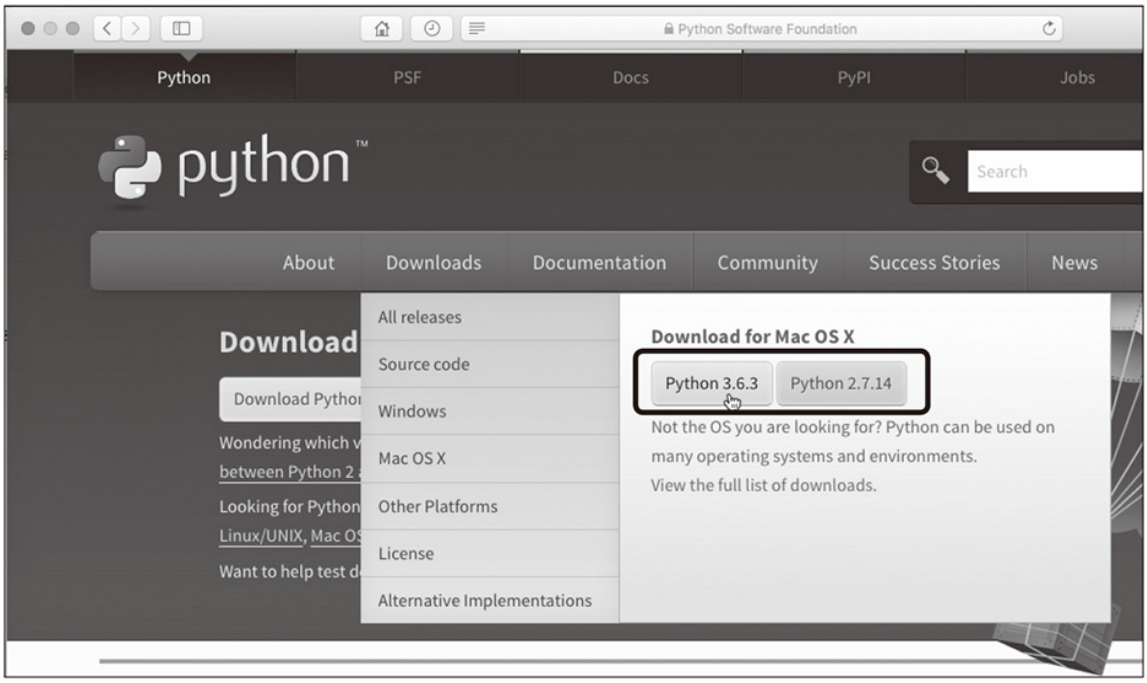


インストールが完了するとスタートメニューにPython関連のコマンドが追加されます。

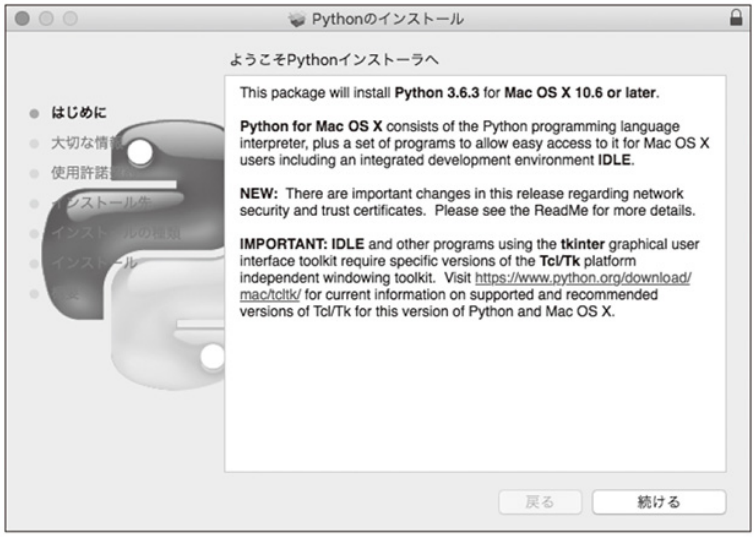
◎macOSにインストール

macOS（High Sierra）には標準でPython 2がインストールされています。本書で解説するPython 3は、別途次のようにしてインストールします。

❶ Pythonのオフィシャルサイト（https://www.python.org）の「Downloads」メニューから「Python 3.6.x」（執筆時点では「3.6.3」）を選択します。



❷ **ダウンロードしたpkg形式のインストールファイルをダブルクリックして起動し、その指示に従ってインストールを実行します。**



Python 2とPython 3は共存が可能です。ターミナルでPython 2の対話型インタプリタを起動するにはpythonコマン

ドを、Python 3の対話型インタプリタを起動するには**python3**コマンドを実行します。

◎ターミナルでpython3コマンドを実行

```
$ python3 Enter        ←Python3を起動
Python 3.6.3 (v3.6.3:2c5fed86e0, Oct  3 2017, 00:32:08)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> exit() Enter   ←Python 3を終了
```

◎Ubuntuにインストール

Ubuntu 17.10を標準インストールした場合、デフォルトでPython 3がインストールされています。手動でインストールするにはaptコマンドを使用して次のようにします。

◎ターミナルでpython3をインストール

```
$ sudo apt install python3   Enter
```

macOSと同様に、Python 3のコマンド名は「**python3**」になります。

ターミナルでpython3コマンドを実行

```
$ python3 Enter        ←Python3を起動
Python 3.6.3 (default, Oct  3 2017, 21:45:48)
[GCC 7.2.0] on linux
Type "help", "copyrigh
>>> exit() Enter        ←Python 3を終了
```

なお、パッケージ管理ツールpip（コマンド名はpip3）、およびグラフィックスライブラリtkinterを使用するのに必要な

python3-tkは標準ではインストールされません。次のようにaptコマンドでインストールする必要があります。

pipとpython3-tkをインストール

```
$ sudo apt install python3-pip Enter      ←pipをインストール
$ sudo apt install python3-tk Enter       ←python3-tkをインストール
```

Pythonのディストリビューション「Anaconda」

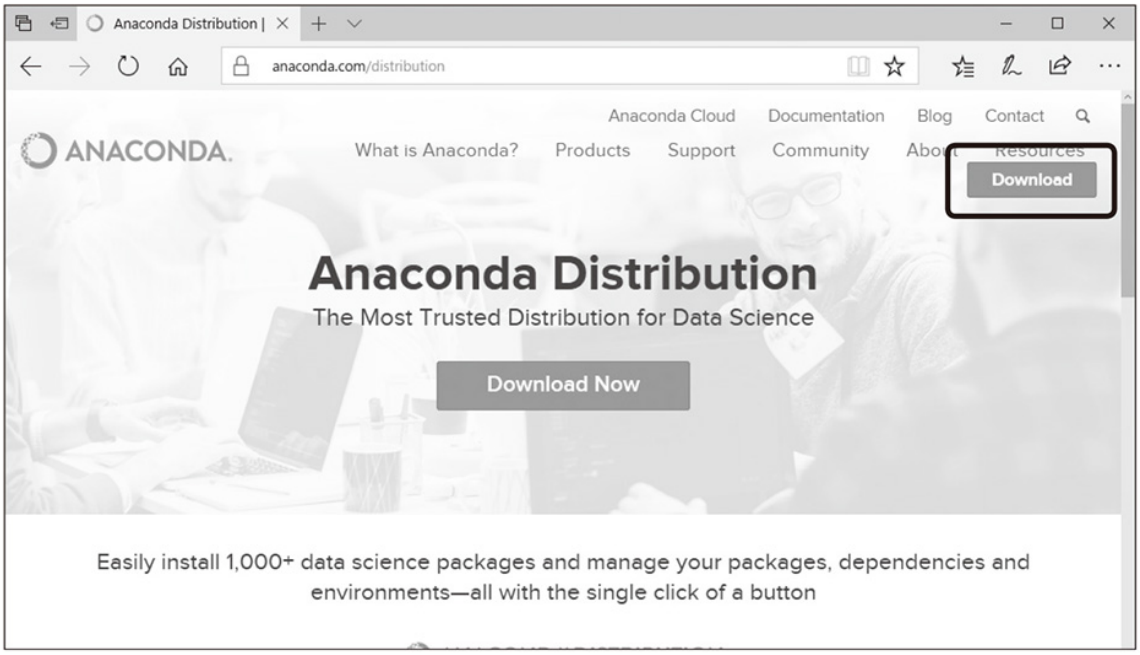
Pythonの配布形態にはPythonのオフィシャルサイトで配布されているバイナリディストリビューションのほかに、用途に応じたパッケージや統合開発環境などそのほかのソフトウエアを追加したPythonディストリビューションが存在します。

その代表といえるのが**Anaconda**です。Anacondaは、Python本体を含むほか、技術計算、データ分析、機械学習などに適したさまざまなパッケージを加え、さらに、統合開発環境「Spyder」や、高機能なパッケージマネージャ「Conda」を搭載しています。

◎Anacondaのインストール

Anacondaは、Windows、macOS、Linux版が無償でダウンロード可能です。

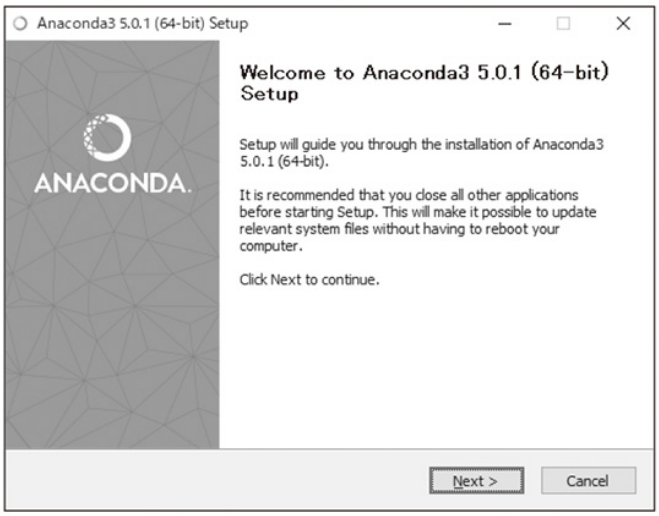
⦿Anacondaのオフィシャルサイト
https://www.anaconda.com/distribution/



「Download」ボタンをクリックし、「Anaconda 5.x.x For Windows Installer」から「Python 3.6 version」の「Download」を選択してダウンロードします。ダウンロードしたインストーラを起動し指示に従ってインストールを行います。

通常、設定はデフォルトのままでかまいません。

⦿Anaconda 5.x.x For Windows Installer



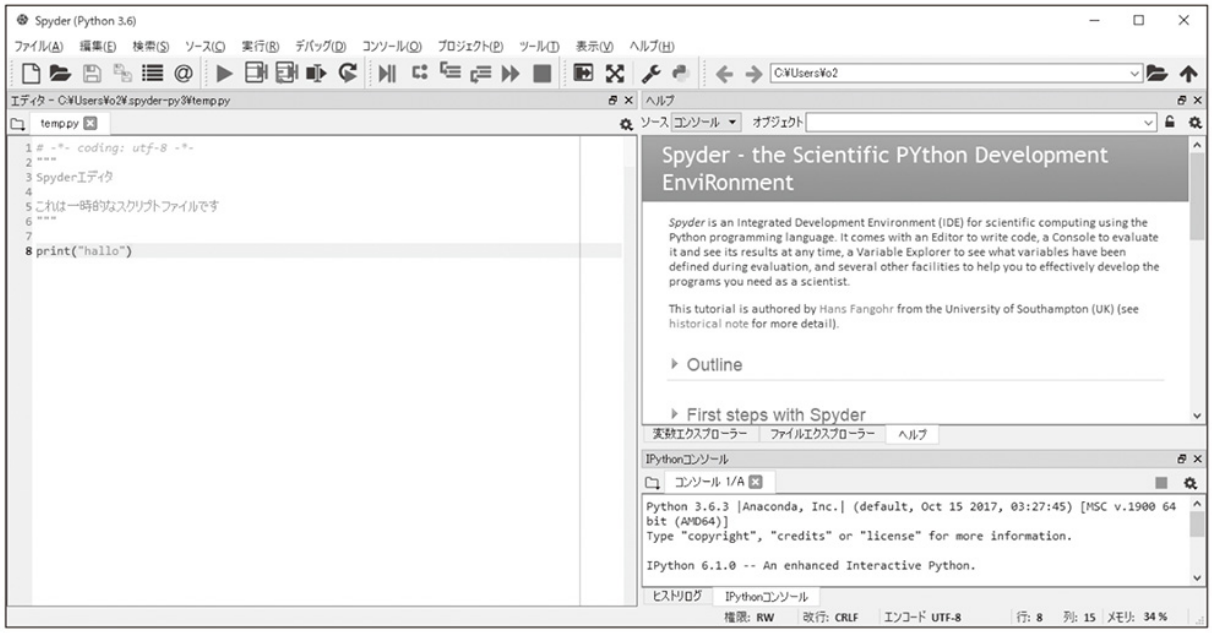
◎Anaconda Navigator

Windowsではスタートメニューからから「Anaconda Navigator」を選択することで、Anacondaに用意されたツールのランチャーである「Anaconda Navigator」が起動します。

⊙Anaconda Navigator

たとえば「Spyder」を選択すると統合開発環境であるSpyderが起動します。

⦿統合開発環境Spyder



2 | モジュールのインポートについて

Pythonには標準で多くのモジュールが付属しています。ここではモジュールをインポートしてプログラムで利用する方法についてまとめておきましょう。また外部のモジュールをインストールする方法についても説明します。

標準ライブラリについて

Pythonには、**標準ライブラリ**として、文字列（str）や整数（int）などの基本的なデータ型、リスト（list）やタプル（tuple）、辞書（dict）などのコレクションといった組み込み型、printやinputなどの組み込み関数、および、さまざまな機能を提供する**モジュール**が用意されています。

◉標準ライブラリ

モジュールのインポート

標準ライブラリに含まれているものであっても、組み込み型以外の関数などを使用するには、**import**コマンドで対応するモジュールをインポートする必要があります。

import モジュール名

たとえば多くの数値演算用の関数や定数が含まれるモジュールに**math**モジュールがあります。対話モードでmathモジュールをインポートするには、pythonコマンドなどでPython

インタプリタを起動した上で、importコマンドを使用して次のようにします。

◎モジュールのインポート

```
>>> import math    Enter
```

これでmathモジュールに含まれる関数に「**math.関数名(～)**」としてアクセスできるようになります。次に**sqrt**関数を使用して平方根を求める例を示します。

◎sqrt関数で平方根を求める

```
>>> math.sqrt(4) Enter        ←4の平方根を求める
2.0
```

mathモジュールに含まれる定数には「**math.定数名**」としてアクセスできます。次に円周率が格納された定数**pi**を使用して半径が4の外周を求める例を示します。

◎定数piを使う

```
>>> math.pi * 2 * 4    Enter
25.132741228718345
```

また、モジュールにはクラスが含まれているものもあります。たとえばカレンダーの機能を提供するクラスが用意された**calendar**モジュールをインポートするには次のようにします。

◎calendarモジュールをインポート

```
>>> import calendar    Enter
```

以上のコマンドを実行することで、calendarモジュールのクラスに「calendar.クラス名(～)」でアクセスできるようになります。

たとえば、calendarモジュールに用意されている**TextCalendar**クラスはシンプルなテキスト形式のカレンダーを管理するクラスです。このクラスを使用してインスタンスを生成し2018年4月のカレンダーを表示するには次のようにします。

◎TextCalendarクラスを利用してカレンダーを表示

```
>>> tc = calendar.TextCalendar() Enter
>>> tc.prmonth(2018, 4) Enter
      April 2018
Mo Tu We Th Fr Sa Su
                   1
2  3  4  5  6  7  8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

なお、次のようにしてインポートすると、モジュールに別名を設定できます。

import モジュール as 別名

たとえばcalendarモジュールをcalという別名でアクセスできるようにするには次のようにします。

◎モジュールをインポートし、別の名前でアクセスできるようにする

```
>>> import calendar as cal  Enter
>>> tc = cal.TextCalendar()  Enter
```

◎関数名やクラス名、定数名だけでアクセスするには

次のような形式でインポートすることにより、関数や定数などに名前だけでアクセスできます。

from モジュール import 名前

名前はカンマ「,」で区切って複数指定してもかまいません。

from モジュール import 名前1, 名前2

mathモジュールのsqrt関数、pi定数に名前だけでアクセスするには次のようにします。

◎sqrt関数、pi定数に名前だけでアクセスできるようにする

```
>>> from math import sqrt, pi  Enter
>>> sqrt(4) Enter   ←sqrt関数に名前だけでアクセス
2.0
>>> pi * 2 * 4 Enter   ←定数piに名前だけでアクセス
25.132741228718345
```

同様にクラスにも名前だけでアクセスできます。

◎TextCalendarクラスに名前だけでアクセスできるようにする

```
>>> from calendar import TextCalendar Enter
>>> tc = TextCalendar() Enter   ←TextCalendarクラスに名前だけでアクセス
>>> tc.prmonth(2018, 4) Enter
      April 2018
Mo Tu We Th Fr Sa Su
```

```
                        1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

◎すべての関数や定数に名前だけでアクセスするには

次のような形式でインポートすることにより、モジュール内のすべての関数や定数に名前だけでアクセスできるようになります。

from モジュール import *

たとえばmathモジュールのすべての機能をインポートするには次のようにします。

◎mathモジュールのすべての機能をインポートする

```
>>> from math import * Enter
```

ただし、この方法はオリジナルの関数や変数などの名前と衝突する可能性が高くなるため、使用にあたっては注意が必要です。

複数のモジュールをまとめたパッケージ

仲間のモジュールを1つのディレクトリ（あるいはそのサブディレクトリ）にまとめて「**パッケージ**」として管理でき

ます。パッケージのインポートはモジュールと同じくimport文を使用して次のようにします。

import パッケージ

あるいは

import パッケージ.(サブディレクトリ.)モジュール

たとえばHTTPに関連したモジュールを集めたパッケージには**http**パッケージがあります。httpパッケージのトップレベルには、HTTPステータスコードなどをまとめた**HTTPStatus**クラスがあります。

これをインポートして定数OKを表示するには次のようにします。

◎httpパッケージをインポートする

```
>>> import http Enter
>>> http.HTTPStatus.OK
<HTTPStatus.OK: 200>
```

NOTE 「import http」のようにパッケージ本体をインポートすると、そのディレクトリの初期化ファイル「__init__.py」が実行されます。HTTPStatusクラスは__init__.pyファイルで定義されています。

httpパッケージには、HTTPクライアントの機能を提供する**client**モジュールが用意されています。これをインポートして「www.google.com」にコネクションを張るには次のようにします。

◎clientモジュールを利用して「www.google.com」にコネクションを張る

```
>>> import http.client Enter
>>> c1 = http.client.HTTPConnection("www.google.com")  Enter
```

これは次のように記述してかまいません。

◎上記を書き換えたコード

```
>>> from http import client Enter
>>> c1 = client.HTTPConnection("www.google.com") Enter
```

外部パッケージを組み込むには

Pythonには標準のパッケージ管理システムとして**pip**（Pip Installs Packages）が用意されています。pipを使用するとPythonパッケージの公式リポジトリ「PyPI（Python Package Index）」より外部パッケージをダウンロードして組み込むことができます。

Python 3のpipのコマンド名はWindowsではpip、macOSおよびUbuntuはpip3です。次のような形式で実行します。

・Windows
pip サブコマンド
・macOS、Ubuntu
pip3 サブコマンド

◎インストールされているパッケージの確認　freeze

現在インストールされているパッケージの一覧を確認するには、**freeze**サブコマンドを使用します（以下、Windowsコ

マンドプロンプトでの実行例です。macOS、Ubuntuの方はpipをpip3に変更して実行してください）。

◎実行例

```
>pip freeze Enter
attrs==17.2.0
Automat==0.6.0
autopep8==1.3.3
constantly==15.1.0
hyperlink==17.3.1
incremental==17.5.0
～以下略～
```

◎パッケージをインストールする　install

　新たにパッケージをダウンロードしてインストールするには「**install パッケージ名**」サブコマンドを使用します。たとえば、JPGやPNGをはじめとするさまざまなフォーマットに対応したグラフィックスライブラリ「**Pillow**」をインストールするには次のようにします。

◎実行例

```
>pip install Pillow Enter
```

◎パッケージを削除する　uninstall

　パッケージを削除するには「**uninstall パッケージ名**」サブコマンドを使用します。

◎実行例

```
>pip uninstall ipython Enter
```

◎ヘルプを表示する　help

pipにはここで紹介したもの以外にもさまざまなサブコマンドが用意されています。利用可能なサブコマンドの一覧は**help**コマンドで確認できます。

◎実行例

```
>pip help Enter

Usage:
   pip <command> [options]

Commands:
   install                     Install packages.
   download                    Download packages.
   uninstall                   Uninstall packages.
   freeze                      Output installed packages in
requirements format.
   list                        List installed packages.
   show                        Show information about
installed packages.
   check                       Verify installed packages have
compatible dependencies.
   search                      Search PyPI for packages.
   wheel                       Build wheels from your
requirements.
   hash                        Compute hashes of package
archives.
   completion                  A helper command used for
command completion.
   help                        Show help for commands.
```

```
General Options:
  -h, --help                  Show help.
  --isolated                  Run pip in an isolated mode, ignoring environment variables and user configuration.
  -v, --verbose               Give more output. Option is additive, and can be used up to 3 times.
  -V, --version               Show version and exit.
  -q, --quiet                 Give less output. Option is additive, and can be used up to 3 times (corresponding to WARNING,
                              ERROR, and CRITICAL logging levels).
  --log <path>                Path to a verbose appending log.
  --proxy <proxy>             Specify a proxy in the form [user:passwd@]proxy.server:port.
  --retries <retries>         Maximum number of retries each connection should attempt (default 5 times).
～以下略～
```

◎標準ライブラリのモジュールのバージョンを確認するには

先に説明したように、pipでインストールした外部パッケージのバージョンは「pip freeze」（前出の「インストールされているパッケージの確認」）で確認できます。標準ライブラリのモジュールに関しては、pythonインタプリタを起動し、目的のモジュールをインポートして特殊変数「**__version__**」を表示することで確認できます。

◎実行例

```
>python    ←pythonインタプリタを起動（macOSの場合にはpython3）
>>> import json      ←jsonモジュールをインポート
```

```
>>> json.__version__       ←バージョンを確認
'2.0.9'
```

なお、「help("モジュール名")」を実行すると表示されるヘルプ画面の「VERSION」でバージョン番号を確認することもできます。

◎実行例

```
>>> help("json") Enter

～中略～

DATA
        __all__ = ['dump', 'dumps', 'load', 'loads', 'JSONDecoder',
'JSONDecod...

VERSION
      2.0.9   ←バージョン

AUTHOR
      Bob Ippolito <bob@redivi.com>

FILE
      /Library/Frameworks/Python.framework/Versions/3.6/lib/pytho
n3.6/json/__init__.py
```

N O T E モジュールによってはバージョン番号の定義されていないものもあります。

N O T E platformモジュールのpython_version関数を使用すると実行中のpythonのバージョンが確認できます。

◎実行例

```
>>> import platform Enter
>>> platform.python_version() Enter
'3.6.3'
```

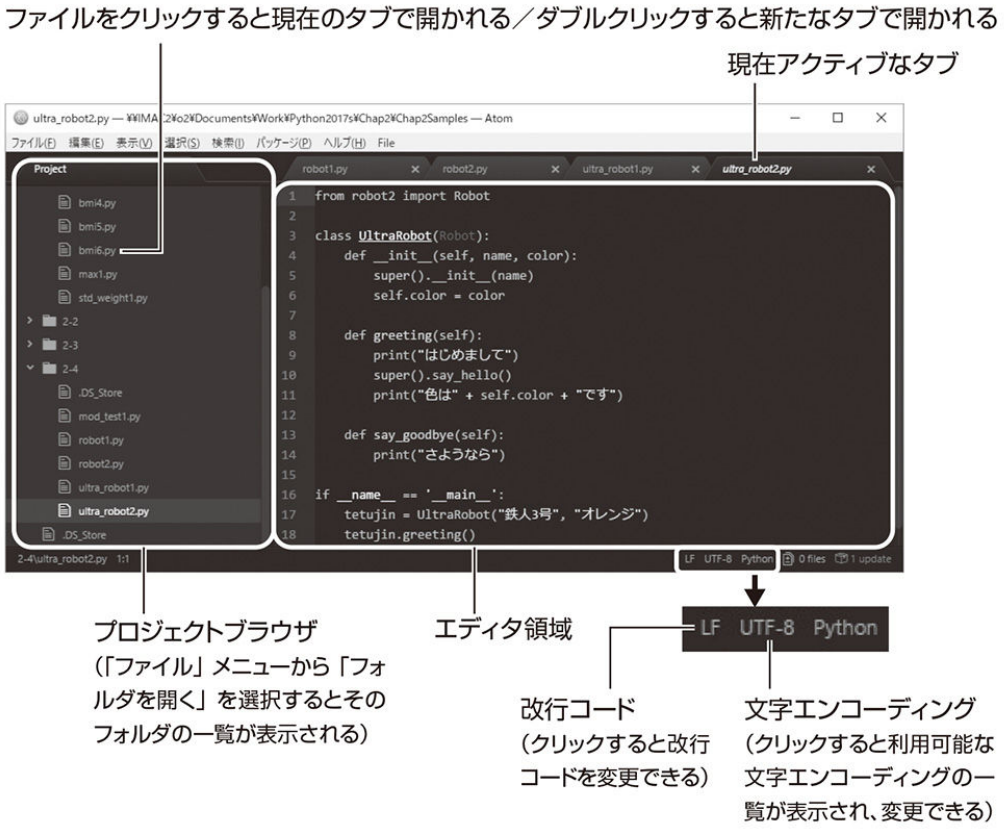
3 | Atomエディタについて

Pythonのプログラム作成に必須なのが、扱いやすいエディタです。ここでは高機能フリーのエディタとして人気の高いAtomエディタを紹介します。atom-runnerパッケージを追加することにより、Pythonプログラムをエディタ内で実行できます。

Atomエディタの画面構成

Atomは、バージョン管理ツールで有名なGitHub社が開発元のオープンソースのテキストエディタです。Atomのインストール方法を説明する前に、Atomエディタの実行画面の特徴を示します。

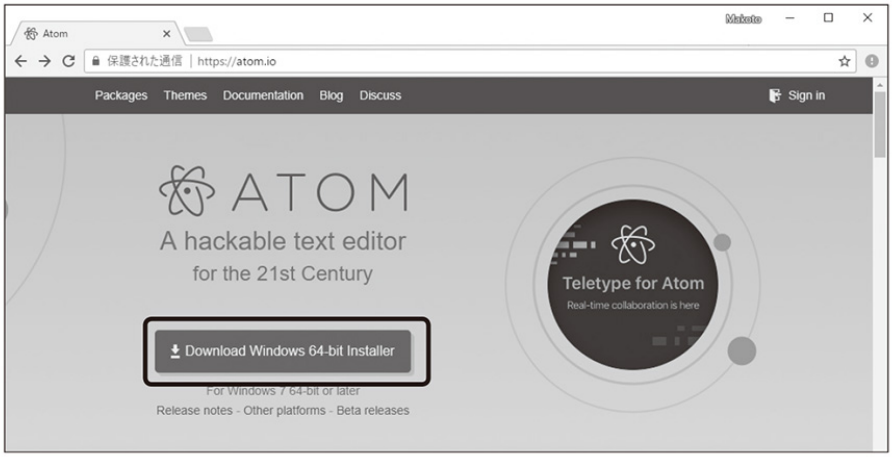
⊙Atomエディタの画面



Atomエディタのインストール

Atomエディタの対応OSは、macOS、Windows、Linuxになります。Atomをインストールするには、まずオフィシャルサイト「https://atom.io」より使用中のOSに応じたインストーラをダウンロードします。このインストーラを実行することで、Atomをインストールします。

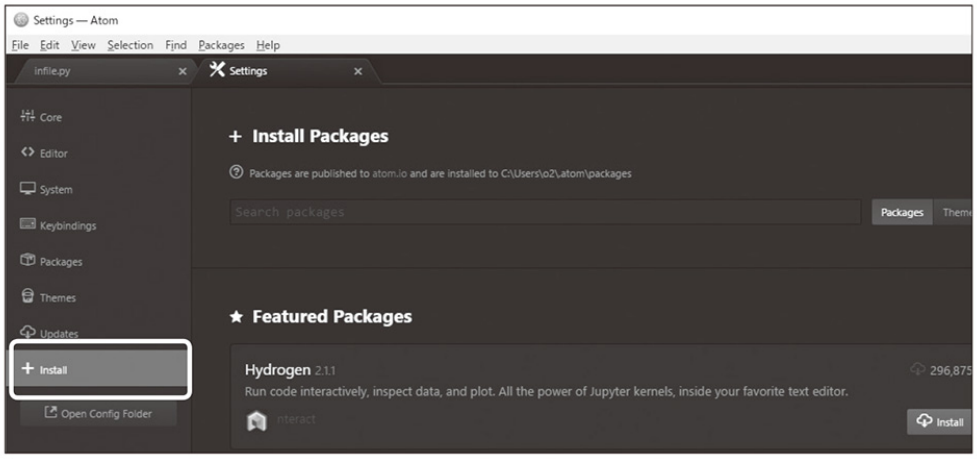
⊙Atomのオフィシャルサイト「https://atom.io」よりインストーラをダウンロード



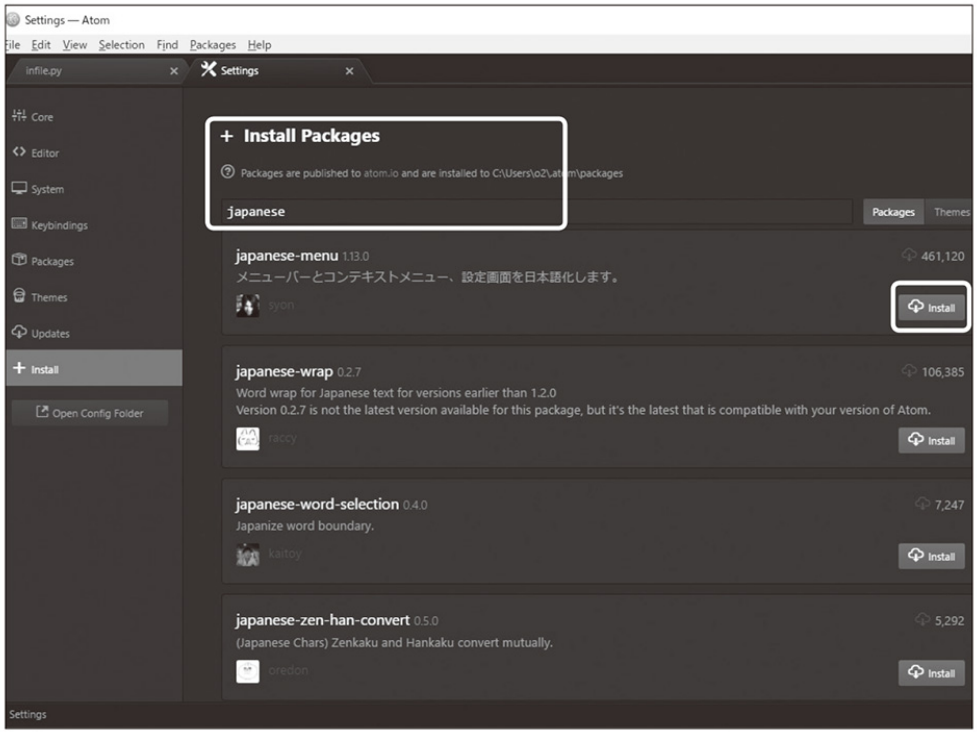
メニューの日本語化

Atomの大きな特徴は、必要に応じてパッケージを追加することによりさまざまな機能拡張が行えることです。まずはメニューや設定画面用の日本語化パッケージ「**japanese-menu**」をインストールしましょう。

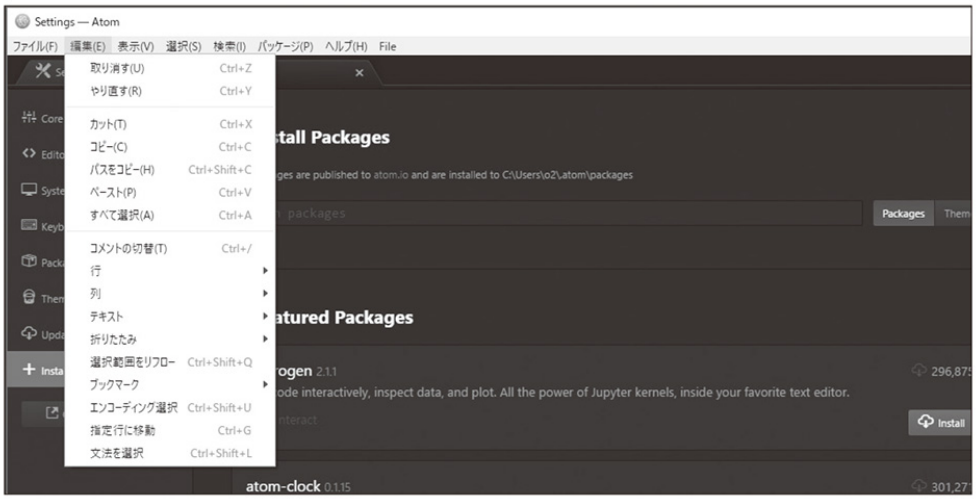
- **「File」メニューから「Settings」を選択します。左のリストから「Install」を選択します。**



❷ **「Install Packages」検索ボックスに「japanese」とタイプしEnterキーを押します。すると日本語関連のパッケージが検索されるので、「japanese-menu」の「Install」ボタンをクリックしてインストールします。**



以上で、メニューが日本語化されます。



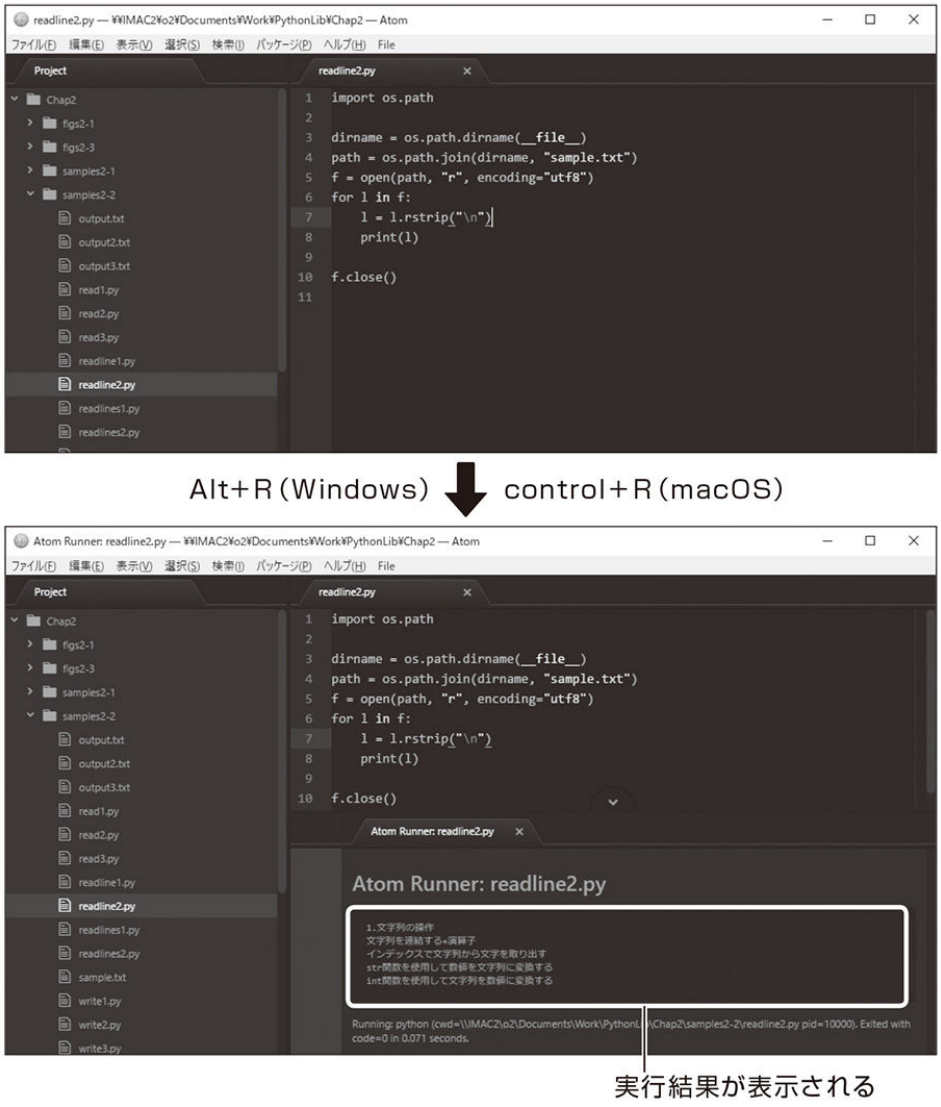
Pythonプログラムの開発に便利なパッケージ

Pythonプログラムの開発に便利なそのほかのパッケージをいくつか簡単に紹介しましょう。なお、ここで紹介するパッケージを使用するには、日本語化パッケージと同じように、インストールしておく必要があります。

◎atom-runner

atom-runnerは、編集中のPythonプログラムをエディタ内で実行できるようにするパッケージです。実行結果は下部のパネルに表示されます。Pythonプログラムを実行してこのパネル上に結果を表示するショートカットキーはAlt+R（Windows）、control+R（macOS）キーになります。

⦿atom-runner



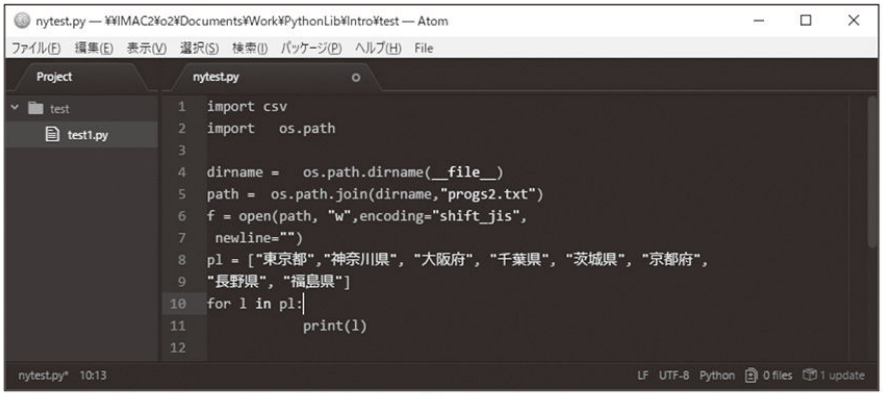
結果を消去するにはescキーを押します。

NOTE atom-runnerは標準入力（コマンドラインでのデータの入力）には対応していないため、input関数を使用することはできません。

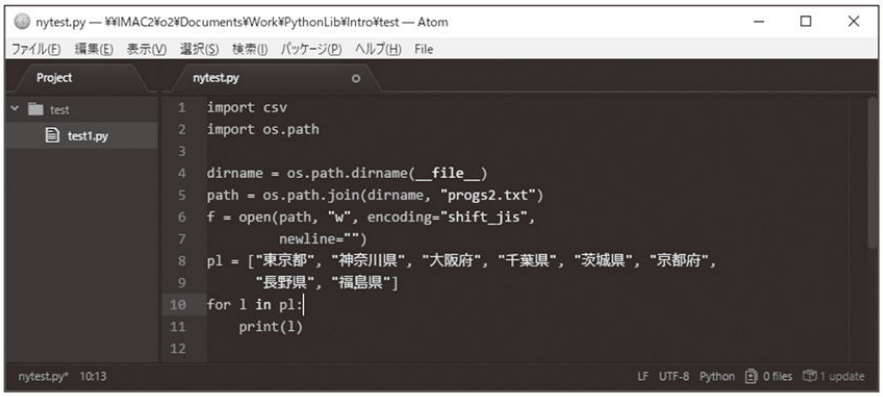
◎atom-beauty

atom-beautyは、PythonやJavaScriptなどさまざまなプログラミング言語のソースコードを美しく整形してくれるパッケージです。ショートカットキーはAlt+Ctrl+B（Windows）、option+control+B（macOS）です。

⊙atom-beauty



Alt+Ctrl+B (Windows) ⬇ option+control+B (macOS)



NOTE atom-beautyには、Pythonのautopep8パッケージが別途必要になります。次のようにコマンドライン上でpipコマンドを実行しインストールしておきます。

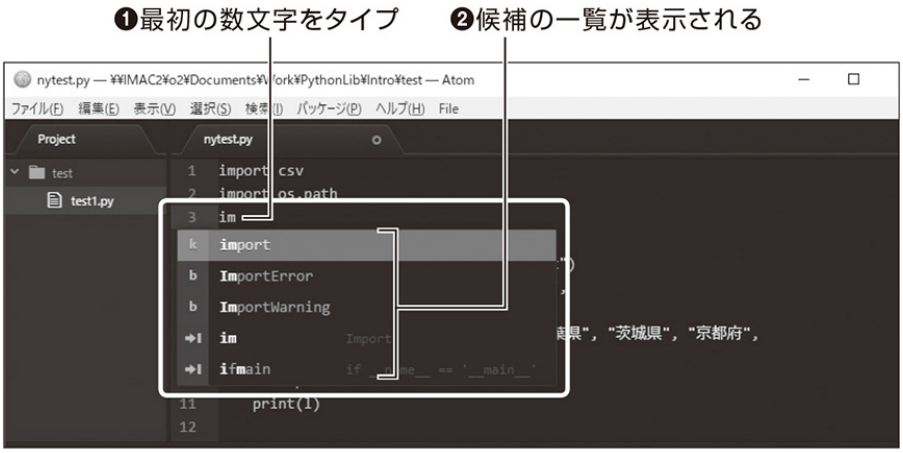
Windowsコマンドライン

```
>pip install autopep8 Enter
```

◎autocomplete-python

autocomplete-pythonは、Pythonプログラムの入力を自動で補完してくれるパッケージです。最初の数文字をタイプすると候補の一覧を表示します。

⊙autocomplete-python



◎wordcount

wordcountは、ステータスバーに編集中のファイルの単語数と文字数を表示してくれるパッケージです。範囲を選択した場合には、その範囲の単語数と文字数が表示されます。

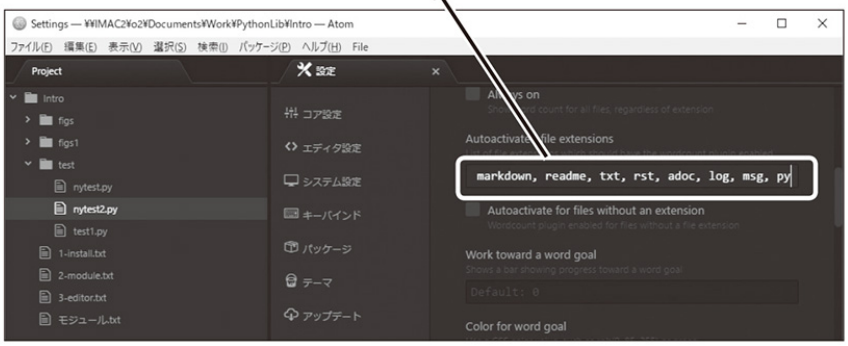
⊙wordcount

NOTE wordcountの機能は、エディタで開いているファイルの拡張子に応じて有効／無効を切り替えられます。Pythonのソースファイル（拡張子は「**.py**」）でwordcountを有効にするには、「環境設定」の「パッケージ」→「wordcount」→「設定」で「Autoactivated file extensions」に「py」を追加しておきます。

⦿wordcountを有効にする

```
markdown, readme, txt, rst, adoc, log, msg, py
```

拡張子として「py」を追加



1章以降のレシピの例を実行する方法

1章以降では、Pythonコードの対話モードでの実行例か、Pythonスクリプトファイル（.py）の例を掲載しています。対話モードは、次のように起動します。

◎Windowsコマンドプロンプトでpythonコマンドを実行

```
> python Enter      ←Python 3を起動
>>>                    ←対話モードでコマンドが入力できる
```

◎macOSやUbuntuのターミナルでpython3コマンドを実行

```
$ python3 Enter     ←Python 3を起動
>>>                    ←対話モードでコマンドが入力できる
```

Pythonスクリプトファイル（.py）は、以下のように実行します。以下は、kaibun1.py（007　文字列を反転するには）の実行例です。

◎Windowsコマンドプロンプトでスクリプトを実行

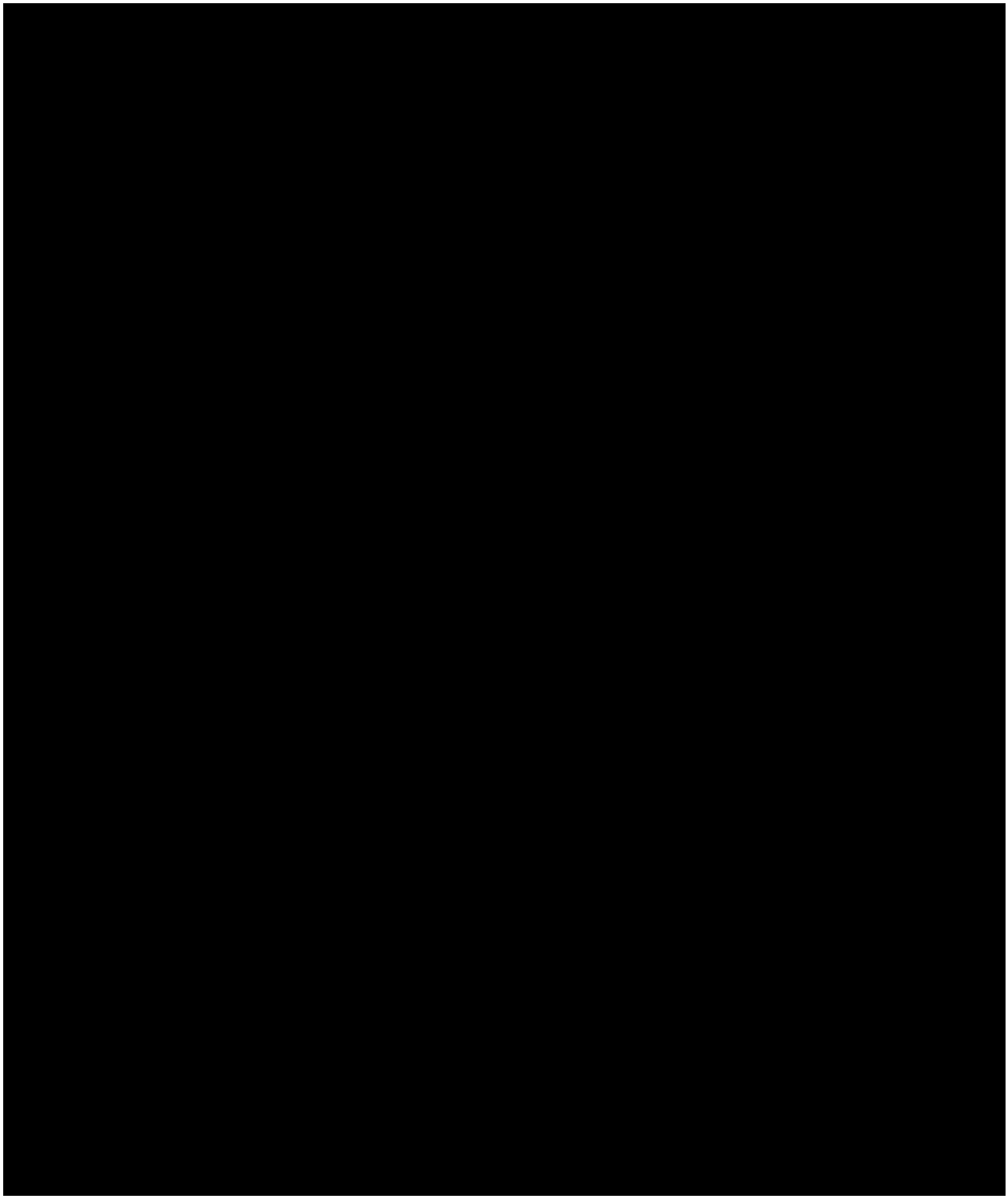
```
> python kaibun1.py Enter      ←スクリプトを実行
```

◎macOSやUbuntuのターミナルでスクリプトを実行

```
$ python3 kaibun1.py Enter      ←Python 3を起動
```

なお、1章以降では「> python」で統一して記載しています。macOSやUbuntuの場合は「$ python3」と読み替えてください。

Chapter 1



文字列とデータの基本操作

Pythonでは文字列はstrクラスのインスタンスです。このChapterではまず文字列の基本的な操作について説明します。そのあとで、関数を使用して値のデータ型やIdを調べる方法や、三項演算子を使用して条件判断をシンプルに行う方法などについて説明します。

1-1 文字列の基本操作

Pythonではすべてのデータはオブジェクトです。文字列はstrクラス、数値はintクラスやfloatクラスのインスタンスです。この節では文字列の基本操作について説明します。

➡001

文字列をリテラルによって生成するには

文字列クォーテーションで囲む

文字列リテラルにより文字列を生成するには、文字列をシングルクォーテーション「'」、もしくはダブルクォーテーション「"」で囲みます。

◎実行例

```
>>> s1 = 'こんにちは' Enter
>>> s2 = "さようなら" Enter
```

クォーテーションを含む文字列を生成したい場合には、もう一方のクォーテーションで囲みます。

◎実行例

```
>>> s2 = "What's going on?" Enter
```

●複数行の文字列を生成する

複数行の文字列を生成するには、3つのシングルクォーテーション、もしくは3つのダブルクォーテーションで囲みます。

◎**実行例**

```
>>> s3 = """ クォーテーション3つで囲むと Enter
... 複数行の文字列を生成できます。""" Enter
```

ＮＯＴＥ プログラム内に文字列リテラルだけの行を記述することもできます。そのためクォーテーション3つで囲んだ、複数行の文字列リテラルは、コメントとしても利用されます。

```
"""
   これはコメントです
   これもコメントです
"""
```

➡002

文字列の長さを求めるには

len関数を使用する

文字列の長さを求めるには、**len**関数を使用します。

◎**関数**　**len(s)**

カテゴリ	**組み込み関数**
引数	**文字列**
戻り値	**文字列の長さ**
説明	**引数sで指定した文字列の長さを戻す**
使用例	**l = len(str1)**

◎**実行例**

```
>>> len('hello') Enter
5
```

```
>>> len('こんにちは') Enter
5
```

●指定したエンコーディングのバイト数を求める

　strクラスの**encode**メソッド（024で解説）を使用すると、文字列を指定したエンコーディングのバイト列に変換できます。その結果をlen関数に渡せばバイト数がわかります。

◎実行例

```
>>> str = 'さようなら' Enter
>>> len(str.encode('shift-jis')) Enter
10
>>> len(str.encode('utf-8')) Enter
15
>>> len(str.encode('utf-16')) Enter
12
>>> len(str.encode('utf-32')) Enter
24
```

➡003

文字列を連結するには

「+」演算子、「*」演算子を使用する

　文字列同士を連結するには「**+**」演算子を使用します。

◎実行例

```
>>> 'こんにちは' + 'Python' Enter
'こんにちはPython'
>>> '今日は' + '月曜' + '日です' Enter
```

```
'今日は月曜日です'
```

　ただし、数値などほかのデータ型の値と文字列を直接連結することはできません。TypeError例外が送出されます。

◎実行例

```
>>> 2018 + '年' Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

　str関数（strコンストラクタ→016で解説）を使用して数値を文字列に変換してから連結する必要があります。

◎実行例

```
>>> str(2018) + '年' Enter
'2018年'
```

●指定した回数を連結する

　「*」演算子を使用すると、指定した回数だけ文字列を繰り返し連結できます。

文字列 * 整数

　次に'Hello'を3つ連結する例を示します。

◎実行例

```
>>> 'Hello' * 3 Enter
'HelloHelloHello'
```

➡004

アルファベットの大文字／小文字を変換するには

upperメソッド、lowerメソッド、swapcaseメソッドを使用する

小文字を大文字に変換するには、strクラスの**upper**メソッドを使用します。

◎メソッド　upper()

カテゴリ	**strクラス**
引数	**なし**
戻り値	**大文字に変換した文字列**
説明	**小文字を大文字にして戻す**
使用例	**str2 = str1.upper()**

◎実行例

```
>>> 'abc'.upper() Enter
'ABC'
```

逆に大文字を小文字に変換するには**lower**メソッドを使用します。

◎メソッド　lower()

カテゴリ	**strクラス**
引数	**なし**
戻り値	**小文字に変換した文字列**
説明	**大文字を小文字にして戻す**
使用例	**str2 = str1.lower()**

◎実行例

```
>>> 'ABC'.lower() Enter
'abc'
```

これらのメソッドは、半角アルファベットだけでなく全角アルファベットに対しても機能します。

◎実行例

```
>>> 'Ａ Ｂ Ｃ Ｄ'.lower Enter　←全角の'Ａ Ｂ Ｃ Ｄ'
'ａ ｂ ｃ ｄ'
```

●小文字を大文字に、大文字を小文字に

文字列内のアルファベット小文字を大文字に、大文字を小文字に変換するには**swapcase**メソッドを使用します。

◎メソッド　swapcase()

カテゴリ	**strクラス**
引数	**なし**
戻り値	**小文字を大文字に、大文字を小文字にした文字列**
説明	**アルファベットの小文字を大文字に、大文字を小文字にして戻す**
使用例	**str2 = str1.swapcase()**

◎実行例

```
>>> 'Hello World'.swapcase() Enter
'hELLO wORLD'
```

➡005
文字列から指定した位置の文字を取り出すには
インデックスを使用する

文字列はリストなどと同じようにシーケンス型のデータなので、次の書式で**インデックス**を指定することで、対応する

位置の文字を取り出すことができます。

文字列[インデックス]

インデックスは先頭の文字を0とする整数値です。マイナスの数値を使用すると、文字列の最後から位置を指定して文字を取り出せます。最後の文字を「-1」とします。

◎実行例

```
>>> str1 = 'こんにちは' Enter
>>> str1[0] Enter   ←先頭から1文字目
'こ'
>>> str1[1] Enter   ←先頭から２文字目
'ん'
>>> str1[len(str1) - 1] Enter    ←最後の文字
'は'
>>> str1[-1] Enter   ←最後の文字
'は'
>>> str1[-2] Enter   ←最後から２文字目
'ち'
```

➡006

文字列の一部を取り出すには

スライスを使用する

文字列から**スライス**（部分文字列）を取り出すには、次のように指定します。

文字列[最初の文字のインデックス:最後の文字のインデックス+1]

◎実行例

```
>>> str2 = '0123456789' Enter
>>> str2[0:4] Enter    ←最初の4文字を取り出す
'0123'
>>> str2[1:3] Enter    ←2文字目から3文字目までを取り出す
'12'
```

　最初のインデックスを省略した場合には「0」が、終わりのインデックスを省略した場合には文字列の長さ（len(文字列)）が指定されているものと見なされます。

◎実行例

```
>>> str2[:2] Enter    ←str2[0:2]と同じ
'01'
>>> str2[4:] Enter    ←str2[4:len(str2)]と同じ
'456789'
```

　この場合も、インデックスに負の値を指定して最後の文字からの位置を指定することもできます。次のようにすると拡張子「.txt」を取り除けます。

◎実行例

```
>>> filename = 'sample.txt' Enter
>>> filename[:-4] Enter    ←最後の4文字を取り除く
'sample'
```

●途中の文字をスキップして取り出す

　次のように、スライスを指定する部分の最後にステップ数（何番目ごとに取り出すか）を指定できます。

文字列[最初の文字のインデックス:最後の文字のインデックス+1:ステップ数]

◎実行例

```
>>> str2 = '0123456789' Enter
>>> str2[::2] Enter    ←2文字ごとに取り出す
'02468'
>>> str2[::3] Enter    ←3文字ごとに取り出す
'0369'
```

➡007

文字列を反転するには

「文字列[::-1]」を使用する

スライスのステップ数を設定する個所に「-1」指定し、最初と最後のインデックスを省略すると文字列を反転できます。

文字列[::-1]

◎実行例

```
>>> 'Hello'[::-1] Enter
'olleH'
>>> 'パイソン'[::-1] Enter
'ンソイパ'
```

次に、コマンドラインから文をひらがなやカタカナで入力し、それが回文かどうかを判断する例を示します。

◎リスト kaibun1.py

```
str1 = input('文字列を入力: ')
if str1 == str1[::-1]: 1
   print('回文です')
else:
   print('回文ではありません')
```

1のif文で、入力した文字列と反転した文字列を比較しています。

■ 実行結果

```
文字列を入力: このこどもどこのこ Enter
回文です
文字列を入力: パイソンパイソン Enter
回文ではありません
文字列を入力: たけやぶやけた Enter
回文です
```

➡008

文字列の一部を置換するには

replaceメソッドを使う

replaceメソッドを使用すると、文字列内の指定した文字列を別の文字列に置き換えることができます。

◎メソッド　replace(old, new[, count])

カテゴリ	**strクラス**
引数	**old：置換する文字列、new：置換後の文字列、count：回数**
戻り値	**置換された文字列**

説明	oldをnewに置換した文字列を戻す。置換対象の文字列が複数ある場合、引数countを指定するとその数だけ置換する
使用例	s = s.replace("You", "He")

引数**count**を指定しなかった場合は、見つかったすべての文字列が置換されます。

◎実行例

```
>>> "good new good new good".replace("good", "bad") Enter
'bad new bad new bad'
```

引数countを指定した場合には、最大でその数だけ置換されます。

◎実行例

```
>>> "good new good new good".replace("good", "bad", 2) Enter
'bad new bad new good'
```

NOTE 正規表現と呼ばれる表記を使用するとより柔軟な置換が行えます（101「パターンにマッチする部分を置換するには」参照）。

➡009
文字列内の文字を変換表に従って置換するには

translateメソッド、maketransメソッドを組み合わせて使う

translateメソッドでは「a」を「A」に、「b」を「B」に、といったように、指定した変換表に基づいて文字単位で置換を行うことができます。

◎メソッド　translate(table)

カテゴリ	**strクラス**
引数	**変換表**
戻り値	**置換された文字列**
説明	**文字を変換表に従って変換する**
使用例	**s = "AabcD".translate(str.maketrans("abc", "ABC"))**

変換表を作成するには**maketrans**メソッドを使用します。

◎メソッド　maketrans(x, y)

カテゴリ	**strクラス**
引数	**x:変換元の文字の並び、y：変換後の文字の並び**
戻り値	**変換表**
説明	**translateメソッドで使用する変換表を作成する**
使用例	**tbl = str.maketrans("abc", "ABC")**

引数xと引数yは同じ文字数である必要があります。次に、1、2、3をそれぞれ①、②、③に置換する例を示します。

◎実行例

```
>>> "1apple 2orange 3banana".translate(str.maketrans("123", "①②③")) Enter
'①apple ②orange ③banana'
```

ＮＯＴＥ maketransはインスタンスに依存しないクラスメソッド（スタティックメソッド）です。「str.maketrans(～)」の形式で使用します。

➡010
文字列内を検索するには


in演算子、not in演算子、findメソッド、rfindメソッド、indexメソッドを使用する

in演算子を使用すると、文字列内に部分文字列が含まれているかを調べられます。

検索する文字列 in 文字列

含まれている場合にはTrueを、含まれていない場合にはFalseを戻します。

◎実行例

```
>>> "fine" in "I'm fine" Enter
True
>>> "good" in "I'm fine" Enter
False
```

in演算子の代わりに「**not in**」演算子を使用すると、文字列が含まれていない場合にTrueを、含まれていた場合にはFalseを戻します。

◎実行例

```
>>> "fine" not in "I'm fine" Enter
False
>>> "good" not in "I'm fine" Enter
True
```

●文字列のインデックスを調べる

in演算子で調べられるのは部分文字列が含まれているどうかだけで、その位置まではわかりません。

findメソッドを使用すると、部分文字列が見つかった場合にそのインデックスを調べることができます。

◎メソッド　find(sub,[start[, end]])

カテゴリ	**strクラス**
引数	**sub：部分文字列、start：開始位置のインデックス、end：終了位置の次のインデックス**
戻り値	**検索された文字列のインデックス**
説明	**文字列に部分文字列が含まれていた場合にそのインデックスを戻す。見つからなかった場合には「-1」を戻す。引数startで開始位置、引数endで終了位置を指定することも可能。**
使用例	**ix = s.find("Python")**

インデックスは先頭を0とする整数値です。見つからなかった場合には「-1」を戻します。

◎実行例

```
>>> "012345678901234567890".find("34") Enter
3
>>> "012345678901234567890".find("abc") Enter
-1
```

引数**start**で検索を開始する位置のインデックスを指定できます。

◎実行例

```
>>> "012345678901234567890".find("34", 9) Enter
13
```

引数**end**で検索を終了する位置の次のインデックスを指定できます。

◎実行例

```
>>> "012345678901234567890".find("34", 9, 14) Enter
-1
```

●文字列の最後から検索する

findメソッドの仲間に**rfind**メソッドがあります。findメソッドは文字列の先頭から検索が行われますが、rfindメソッドでは終わりから検索が行われます。

◎実行例

```
>>> "012345678901234567890".rfind("34") Enter
13
```

●文字列が見つからなかった場合には例外を送出するようにする

findメソッドの場合、文字列が見つからなかった場合「-1」が戻されます。それに対して**index**メソッドではValueError例外が送出されます。

◎メソッド　index(sub,[start[, end]])

カテゴリ	**strクラス**
引数	**sub：部分文字列、start：開始位置のインデックス、end：終了位置の次のインデックス**
戻り値	**検索された文字列のインデックス**
説明	**文字列に部分文字列が含まれていた場合にそのインデックスを戻す。見つからなかった場合にはValueError例外が送出される。startで開始位置、endで終了位置を指定することも可能**
使用例	**ix = s.index("Python")**

使用方法はfindメソッドと同じです。

◎実行例

```
>>> "012345678901234567890".index("34") Enter
3
>>> "012345678901234567890".index("134") Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

次の例は、ユーザーに文字列を繰り返し入力させ、それが変数str1に含まれていればそのインデックスを表示します。含まれていなければ「ValueError例外です」と表示してループを抜けます。

◎リスト index1.py

```
str1 = "春夏秋冬"
while True:
    search = input("季節名を入力: ")
    try: 1
        index = str1.index(search) 2
        print("インデックス:", index)
    except ValueError: 3
        print("ValueError例外です")
        break
```

1のtry文のブロックで2のindexメソッドの結果を処理し、ValueErrorが発生すれば3のexcept文で捕まえています。

◎実行結果

```
>python index1.py
```

```
検索文字列を入力: 春 Enter
インデックス: 0
検索文字列を入力: 夏 Enter
インデックス: 1
検索文字列を入力: 秋 Enter
インデックス: 2
検索文字列を入力: 赤 Enter
ValueError例外です
```

➡011

文字列内の文字列の数をカウントするには

countメソッドを使用する

countメソッドを使用すると、文字列内の指定した文字列の出現回数を調べることができます。

◎メソッド　count(sub[, start[, end]])

カテゴリ	strクラス
引数	sub：文字列、start：開始位置のインデックス、end：終了位置の次のインデックス
戻り値	文字列の数
説明	startとendの間に文字列subが出現する回数を戻す
使用例	c = s.count("Python")

◎実行例

```
>>> "good bad good".count("good") Enter
2
>>> "010101".count("0", 1) Enter
2
```

●大文字／小文字を無視してカウントするには

strクラスに用意されている文字列を戻すメソッドは、ピリオド「.」で接続することにより連続実行できます。

文字列.メソッド1(～).メソッド2(～).メソッド3(～)

たとえば、大文字／小文字の相違を無視して文字列をカウントするには、upperメソッド（004で解説）により大文字に変換してからcountメソッドを実行します。

◎実行例

```
>>> "Python basic python PYTHON swift".upper().count("PYTHON") Enter
3
```

➡012

文字列内に値を埋め込むには（その１）

formatメソッドを使用する

formatメソッドを使用すると、文字列内の指定した位置に、別の文字列や数値を埋め込むことができます

◎メソッド　format(*args)

カテゴリ	**strクラス**
引数	**値の並び**
戻り値	**埋め込まれた文字列**
説明	**文字列内の「{～}」で指定した箇所に引数を埋め込んで戻す**
使用例	**s = '{}年{:02d}月 {:.2f}'.format(year, 9, n)**

簡単な例から説明しましょう。formatメソッドは元の文字列の「**{}**」で指定した箇所に、引数で指定した値を順に埋め込みます。値は文字列でも数値でもかまいません。

◎実行例

```
>>> '{}年{}月{}日'.format(2018, 4, 1) Enter
'2018年4月1日'
```

●位置引数を指定する

「**{番号}**」とすると、指定した位置の引数を埋め込めます。これを「位置引数」と言います。番号は最初の引数を「0」とする整数値です。次の例は、2番目、3番目、最初の引数を順に埋め込みます。

◎実行例

```
>>> '{1}, {2}, {0}'.format('one', 'two', 'three') Enter
'two, three, one'
```

同じ番号を指定することで、同じ引数を複数回埋め込むこともできます。

◎実行例

```
>>> '{0}, {0}, {0}'.format('one', 'two', 'three') Enter
'one, one, one'
```

●桁数と配置位置を指定する

表示する桁数（文字数）を指定するには「**{番号:桁数}**」を指定します（番号は省略可能）。文字列の場合にはデフォルトで左寄せになります。

◎実行例

```
>>> '|{:10}|'.format('hello') Enter
'|hello     |'
```

　右寄せにするには「**>**」を、中央寄せにするには「**^**」を桁数の前に指定します。

◎実行例

```
>>> '|{:>10}|'.format('hello') Enter
'|     hello|'
>>> '|{:^10}|'.format('hello') Enter
'|  hello   |'
```

●整数のフォーマット指定

　「**{番号:フォーマット}**」の形式で指定すると、表示形式を指定できます。整数の場合デフォルトのフォーマットは「**d**」（10進数)です。「**b**」を指定すると2進数、「**o**」を指定すると8進数、「**x**」を指定すると16進数で表示されます。
　次に10進数の255を、10進数、16進数、2進数の形式で埋め込む例を示します。

◎実行例

```
>>> '{0} -> {0:x} -> {0:b}'.format(255) Enter
'255 -> ff -> 11111111'
```

　デフォルトでは負の値のみ数値の前に「-」記号が表示されます。フォーマットで「**+**」を指定すると、正、負のどちらの値にも符号を付けます。

◎実行例

```
>>> '{}'.format(-10) Enter
'-10' ←デフォルトでは負の値のみ符号が付く
```

```
>>> '{:+}'.format(10) Enter
'+10' ←「+」を指定すると正の値にも符号が付く
```

文字列を埋め込む場合と同じく、桁を指定するには「**{番号:桁d}**」を指定します。

◎実行例

```
>>> '{:10d}'.format(4) Enter
'         4'
```

桁の前に「0」を記述すると前方を「0」で埋めます。

◎実行例

```
>>> '{:02d}時{:02d}分'.format(3, 4) Enter
'03時04分'
```

フォーマットに「**,**」を指定すると、数値の3桁ごとに「,」をセパレータとして表示します。

◎実行例

```
>>> '合計{:,}円'.format(1034044) Enter
'合計1,034,044円'
```

●浮動小数点数のフォーマット指定

浮動小数点数形式で埋め込みたい場合にはフォーマットに「**f**」を指定します。

◎実行例

```
>>> '{:f}'.format(4) Enter
'4.000000'
```

デフォルトでは小数点以下第6位まで表示されますが、「**{番号:.桁数f}**」で桁数を指定できます。

◎実行例

```
>>> '{:f}'.format(1/3) Enter
'0.333333' ←1/3の結果を表示。デフォルトでは（6桁）
>>> '{:.3f}'.format(1/3) Enter
'0.333 ←3桁まで表示
```

「f」の代わりに「**e**」を指定すると指数表記、「%」を指定するとパーセント表記になります。

◎実行例

```
>>> '{:.3e}'.format(1/3) Enter
'3.333e-01'
>>> "{:.2%}".format(1/3) Enter
'33.33%'
```

●位置引数に名前をつける

formatメソッドの引数に「名前=値」を指定することにより位置引数に名前をつけることができます。そうすることにより番号の代わりに名前でアクセスすることができます。

◎実行例

```
>>> name = "山田太郎" Enter
>>>  '{name}:  {age:d} 才   {height:.2f}cm'.format(name=name, age=43, height=165.454) Enter
'山田太郎: 43才 165.45cm'
```

●日付時刻をフォーマットする

datetimeモジュール（138で解説）の**date**クラスや**datetime**クラスのインスタンスをformatメソッドの引数にすることにより、日付時刻のデータを個別に取り出して表示することができます。その場合、次のように「**%文字**」で取り出すフィールドを指定できます。

⦿日付時刻のフィールド指定の例

文字	説明
%Y	西暦4桁の年
%m	月
%d	日
%A	曜日名
%H	時（24時間表記）
%I	時（12時間表記）
%M	分
%S	秒
%p	AMもしくはPM

次に今日の日付を「XXXX年X月X日」の形式で、時刻を12時間表記で表示する例を示します。

◎リスト format1.py

```
from datetime import datetime

now = datetime.now()
datestr = '{0:%Y}年{0:%m}月{0:%d}日'.format(now)  1
print(datestr)
timestr = '{0:%p} {0:%I}時{0:%M}分'.format(now)  2
print(timestr)
```

1で日付を、**2**で時刻を文字列に埋め込んでprint関数で表示しています。

■実行結果

2017年12月13日
PM 09時59分

➡013

文字列内に値を埋め込むには（その2）

f文字列を使用する

Python 3.6以降では、文字列内に別の値を埋め込むのに「**f文字列**（formatted string）」というよりシンプルな書式が用意されました。

f'文字列'

文字列内の値を埋め込みたい位置に「**{値}**」を記述します。変数や計算式を埋め込んだり、formatメソッドと同じく**{値:フォーマット}**でフォーマット指定することもできます。

◎実行例

```
>>> f'今年は{2018}年です' Enter
'今年は2018年です'
>>> doll = 5 Enter
>>> rate = 108 Enter
>>> f'{doll}ドルは{doll * rate}円です' Enter　←変数や計算式も埋め込める
'5ドルは540円です'
```

◎実行例（{値:フォーマット}でフォーマット指定）

```
>>> f'{1434555:,}' Enter　←3桁区切り
'1,434,555'
```

```
>>> points = 43 Enter
>>> total = 97 Enter
>>> f'{points/total:.2%}パーセント' Enter　←パーセント表記
'44.33%パーセント'
```

次に、前出のformat1.pyのformatメソッドをf文字列に変更した例を示します。

◎リスト format2.py

```
from datetime import datetime

now = datetime.now()
datestr = f'{now:%Y}年{now:%m}月{now:%d}日'  1
print(datestr)
timestr = f'{now:%p} {now:%I}時{now:%M}分'  2
print(timestr)
```

1、**2**でformatメソッドの代わりにf文字列を使用しています。

■実行結果

```
2017年12月13日
PM 10時37分
```

➡014

文字列の先頭／最後から指定した文字を取り除くには

strip、lstrip、rstripメソッドを使用する

文字列の先頭および末尾から指定した文字を取り除くには、**strip**メソッドを使用します。

◎メソッド　strip([chars])

カテゴリ	**strクラス**
引数	**取り除く文字の並び**
戻り値	**削除後の文字列**
説明	**先頭および末尾から引数で指定した文字を取り除いて新たな文字列を戻す**
使用例	**s = s.strip(',.')**

stripメソッドを、引数を省略して実行した場合には、文字列の先頭、および末尾から空白文字（スペース、タブ、改行など）が削除されます。

◎実行例

```
>>> '　\n　こんにちは さようなら\t'.strip() Enter
'こんにちは さようなら'
```

NOTE 空白文字の一覧はstringモジュールのwhitespaceプロパティで確認できます。

実行例

```
>>> import string Enter
>>> string.whitespace Enter
' \t\n\r\x0b\x0c'
```

引数charsでは削除する文字の並びを指定できます（文字列ではない点に注意してください）。'<'および'>'を取り除くには「'<>'」を指定します。

◎実行例

```
>>> '<Python>'.strip('<>') Enter
```

```
'Python'
```

'.'、'|'、'!'を取り除くには'.|!'を指定して次のようにします。

◎実行例

```
>>> '!..green..red|.|.|'.strip('.|!') Enter
'green..red'
```

●文字列の先頭あるいは最後から文字を取り除く

lstrip、**rstrip**メソッドはstripメソッドの簡易版です。lstripメソッドは文字列の先頭から引数で指定した文字を取り除きます。

◎実行例

```
>>> '!..green..red|.|.|'.lstrip('.|!') Enter
'green..red|.|.|'
```

rstirpメソッドは文字列の最後から引数で指定した文字を取り除きます。

◎実行例

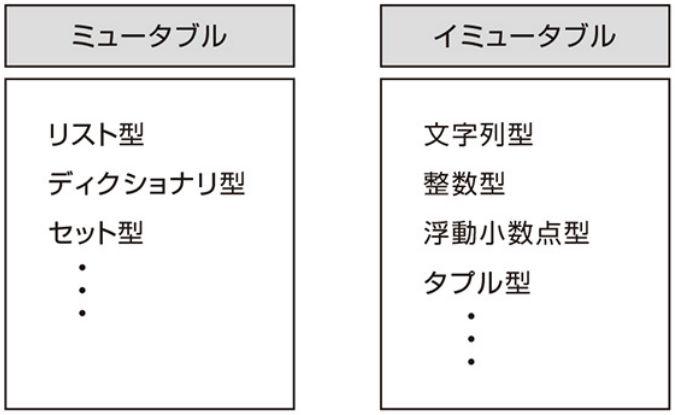
```
>>> '!..green..red|.|.|'.rstrip('.|!') Enter
'!..green..red'
```

ミュータブルな型とイミュータブルな型

Pythonのデータ型は、あとから値を変更可能な「**ミュータブル**」（変更可）な型と、変更できない「**イミュータブ**

ル」（変更不可）な型に大別されます。

⊙ミュータブルとイミュータブル



➡015

リスト、タプル、辞書の要素を接続した文字列を生成するには

joinメソッドを使用する

リスト、タプル、セット、辞書などイテレート（繰り返し処理）可能なオブジェクトの要素を順に接続した文字列を生成するには**join**メソッドを使用します。

◎メソッド　join(iterable)

カテゴリ	**strクラス**
引数	**イテレート（繰り返し処理）可能なオブジェクト**
戻り値	**連結後の文字列**
説明	**引数で指定したオブジェクトの要素をセパレータで連結して戻す**
使用例	**s = ','.join(['赤', '青', '黄'])**

joinメソッドは次の形式で実行します。

'セパレータ'.join(イテレート可能なオブジェクト)

セパレータ（区切り文字）は空文字列''、あるいは複数の文字でもかまいません。次にリストを接続する例を示します。

◎リスト join1.py

```
lst = ['春', '夏', '秋', '冬']

# 要素をそのまま接続
print(''.join(lst))
# セパレータを「,」にして接続
print(','.join(lst))
# セパレータを「<->」にして接続
print('<->'.join(lst))
```

■実行結果

```
春夏秋冬
春,夏,秋,冬
春<->夏<->秋<->冬
```

●数値の要素を接続する

joinメソッドで接続するためには、要素がすべて文字列である必要があります。要素が1つでも数値である場合、それらを接続しようとするとエラーになります。

◎実行例

```
>>> ''.join([1, 'hello', 'python']) Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

数値の要素を接続したい場合には、**map**関数（064で解説）などを使用してあらかじめ文字列に変換しておきます。

◎リスト join2.py

```
lst = ['オレンジ', 34, 4, 'いちご']
lststr = map(str, lst)    1
print(','.join(lststr))    2
```

1のmap関数で、リストlstの各要素にstr関数を実行して文字列に変換し、2のjoinメソッドで接続しています。

■実行結果

```
オレンジ,34,4,いちご
```

●辞書のキー、もしくは値を接続する

辞書をjoinメソッドの引数にして連結することもできます。この場合、キーだけが連結されます。

◎実行例

```
>>> dic = {'赤':'red', '青':'blue', '黄色':'yellow'} Enter
>>> ','.join(dic) Enter
'赤,青,黄色'
```

値だけを連結したい場合には、**values**メソッド（073で解説）で値のリストを取得します。

◎実行例

```
>>> ','.join(dic.values()) Enter
'red,blue,yellow'
```

➡016

数値を文字列に変換するには

str関数、bin関数、oct関数、hex関数を使用する

数値を文字列に変換するには、**str**関数（strコンストラクタ）を使用します。

◎関数　**str(num)**

カテゴリ	**strクラス**
引数	**数値**
戻り値	**変換後の文字列**
説明	**引数numを文字列に変換して戻す**
使用例	**s = str(59)**

引数には整数だけでなく、浮動小数点数も指定できます。

◎実行例

```
>>> str(255) Enter
'255'
>>> str(1 / 3) Enter
'0.3333333333333333'
>>> str(2e5) Enter
'200000.0'
```

引数を16進数や2進数のリテラルで記述した場合でも、10進数の文字列が戻されます。

◎実行例

```
>>> str(0xff) Enter
'255'
```

```
>>> str(0b1111) Enter
'15'
```

●２進数、8進数、16進数の文字列に変換する

整数を2進数の文字列にするには**bin**関数を使用します。

◎実行例

```
>>> bin(255) Enter
'0b11111111'
```

8進数の文字列にするには**oct**関数を使用します。

◎実行例

```
>>> oct(255) Enter
'0o377'
```

16進数の文字列にするには**hex**関数を使用します。

◎実行例

```
>>> hex(255) Enter
'0xff'
```

いずれの関数も浮動小数点数を引数にするとエラーになります。

◎実行例

```
>>> bin(4.0) Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
>>> hex(14.54) Enter
```

```
Traceback (most recent call last):
File '<stdin>', line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

int関数（017で解説）などを使用して整数化したのちに引数にする必要があります。

◎実行例

```
>>> bin(int(4.0)) Enter
'0b100
>>> hex(int(14.54)) Enter
'0xe
```

➡017

文字列を数値に変換するには

int関数、float関数を使用する

'495'のような数値を表す文字列を整数値に変換するには、**int**関数（intコンストラクタ）を使用します。

◎関数　**int(x, [radix])**

カテゴリ	**組み込み関数**
引数	**x：整数を表す文字列、radix：基数**
戻り値	**変換後の整数**
説明	**引数xを引数radixで指定した基数の整数に変換して戻す**
使用例	**s = int('255', 16)**

デフォルトでは基数は10（10進数）です。

◎実行例

```
>>> int('123') Enter　←基数を10に
123
>>> int('ff', 16) Enter　←基数を16に
255
>>> int('1010', 2) Enter　←基数を2に
10
>>> int("777", 8) Enter　　←基数を8に
511
```

ＮＯＴＥ 整数に変換できない値を指定した場合にはValueError例外が送出されます。

◎実行例

```
>>> int('hello') Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hello'
```

int関数を使用すると浮動小数点数を整数に変換することもできます。

◎実行例

```
>>> int(14.0) Enter
14
```

小数点以下の桁は切り捨てられます。

◎実行例

```
>>> int(3.14) Enter
3
```

●浮動小数点数に変換する

数値を表す文字列を浮動小数点数に変換するには**float**関数（floatコンストラクタ）を使用します。

◎関数　float(x)

カテゴリ	**組み込み関数**
引数	**数値を表す文字列**
戻り値	**変換後の浮動小数点数**
説明	**引数xを浮動小数点数に変換して戻す。変換できない場合にはValueError例外が送出される**
使用例	**f = float('3.14')**

引数に整数を表す文字列を指定しても浮動小数点数に変換されます。

◎実行例

```
>>> float('33') Enter
33.0
>>> float('10.5') Enter
10.5
```

➡018
文字列が数値／アルファベット／空白文字であることを調べるには

is～メソッドを使用する

文字列が数値やアルファベットであるかを調べるメソッドをまとめておきます。これらのメソッドは、Trueもしくは Falseを戻します。

⦿is～メソッド

メソッド	説明
isalnum	すべての文字がアルファベットもしくは数字であればTrue
isalpha	すべての文字がアルファベットであればTrue
isdecimal	すべての文字が十進数字であればTrue
isdigit	すべての文字が数字であればTrue(上付き文字の数字もTrue)
isnumeric	すべての文字が数を表す文字であればTrue(漢数字、ローマ数字もTrue)
isidentifier	文字列が有効な識別子であればTrue
islower	文字列がアルファベット小文字であればTrue
isprintable	すべての文字が印字可能文字であればTrue
isspace	すべての文字が空白文字であればTrue
isupper	文字列がアルファベット大文字であればTrue

◎実行例

```
>>> 'hello1234'.isalnum() Enter
True
>>> '2017'.isdecimal() Enter
True
>>> '\u00B2'.isdecimal() Enter   ←上付き文字の数字（2）
False
>>> '\u00B2'.isdigit() Enter
True
>>> '四'.isdecimal() Enter
False
>>> '四'.isnumeric() Enter
True
>>> 'IX'.isdigit() Enter
False
>>> 'IX'.isnumeric() Enter
True
>>> ' \n\t'.isspace() Enter
True
```

➡019

文字列を指定した文字位置で分解しリストに変換するには

splitメソッド、rsplitメソッド、splitlinesメソッドを使用する

splitメソッドを使用すると、文字列を、指定した文字で分割してリストの要素とすることができます。

◎メソッド　split(sep=None, maxsplit=-1)

カテゴリ	**strクラス**
引数	**sep：セパレータ、maxsplit：最大分割数**
戻り値	**分割された要素のリスト**
説明	**引数sepの位置で文字列を分割し、それを要素とするリストを生成して戻す**
使用例	**s = s.split("-")**

文字列をカンマ「,」の位置で分割し、それを要素とするリストを戻すには次のようにします**1**。また、キーワード引数**maxsplit**が与えられた場合には、リストの要素数は最大で「maxsplit + 1」となります**2**。

◎実行例1

```
>>> '春,夏,秋,冬'.split(',') Enter
['春', '夏', '秋', '冬']
```

◎実行例2

```
>>> '春,夏,秋,冬'.split(',', maxsplit=2) Enter
['春', '夏', '秋,冬']
```

引数**sep**は文字列でもかまいません。

◎実行例

```
>>> '赤->青->紫'.split('->') Enter
['赤', '青', '紫']
```

●空白文字で分割する

splitメソッドに引数sepを与えない場合には、スペースやタブ、改行などの空白文字（もしくはその連続）で分割されます。このとき、先頭と終わりの空白文字は無視されます。

◎実行例

```
>>> '　空\n海　\t山'.split() Enter
['空', '海', '山']
```

●右から数えて分割する

splitメソッドの代わりに**rsplit**メソッドを使用すると、引数maxsplitが与えられた場合に分割数を右から数えます。

◎メソッド　rsplit(sep=None, maxsplit=-1)

カテゴリ	**strクラス**
引数	**sep：セパレータ、maxsplit：最大分割数**
戻り値	**リスト**
説明	**引数maxsplitで分割数を指定した場合に、文字列を右から数えて分割する**
使用例	**l = s.rsplit(',', maxsplit=1)**

次に、引数maxsplitを「2」に設定し、rsplitメソッドを実行する例を示します。

◎実行例

```
>>> '春,夏,秋,冬'.rsplit(',', maxsplit=2) Enter
['春,夏', '秋', '冬']
```

●改行で分割する

　文字列を改行の位置で分解してリストにするには**splitlines**メソッドを使用します。

◎メソッド　splitlines([keepends])

カテゴリ	**strクラス**
引数	**改行を残すかどうか**
戻り値	**リスト**
説明	**文字列を改行で分割したリストを戻す**
使用例	**s = s.splitlines()**

　引数を省略した場合には分割後に改行が削除されます **1**。引数**keepends**がTrueの場合には行末の改行を削除しません **2**。

◎実行例 1

```
>>> 'ギター\nピアノ\nベース\n'.splitlines() Enter
['ギター', 'ピアノ', 'ベース']
```

◎実行例 2

```
>>> 'ギター\nピアノ\nベース\n'.splitlines(keepends=True) Enter
['ギター\n', 'ピアノ\n', 'ベース\n']
```

➡020
文字列をセパレータの位置で３分割したタプルを戻すには

partitionメソッド、rpartitionメソッドを使う

splitと似たメソッドに**partition**があります。partitionメソッドは文字列をセパレータの位置を基準に、前方部分、セパレータ、後方部分に3分割します。

◎メソッド　partition(sep)

カテゴリ	**strクラス**
引数	**セパレータ**
戻り値	**タプル**
説明	**セパレータの位置で3分割した文字列を要素とするタプルを戻す**
使用例	**s = s.partition('-')**

次に、'の'の位置で文字列を3分割する例を示します。

◎実行例

```
>>> '私の家'.partition('の') Enter
('私', 'の', '家')
```

セパレータが見つからない場合は、元の文字列と、空文字2つのタプルが返されます。

◎実行例

```
>>> '私の家'.partition(',') Enter
('私の家', '', '')
```

セパレータが複数ある場合には最初の位置で分割されます。

◎実行例

```
>>> '私の家の庭'.partition('の') Enter
('私', 'の', '家の庭')
```

●右から分割する

　セパレータが複数ある場合に最後のセパレータの位置で分割するには**rpartition**メソッドを使用します。

◎実行例

```
>>> '私の家の庭'.rpartition('の') Enter
('私の家', 'の', '庭')
```

➡021

文字列の先頭／最後が指定したものかを調べるには

startswithメソッド、endswithメソッドを使用する

　文字列の先頭が指定された文字列で始まるかを調べるには、**startswith**メソッドを使用します。

◎メソッド　startswith(prefix[, start[, end]])

カテゴリ	**strクラス**
引数	**prefix：検索する文字列、start：開始位置、end：終了位置**
戻り値	**一致すればTrue、しなければFalse**
説明	**文字列の先頭と引数prefixを比較し一致すればTrueを、そうでなければfalseを戻す。引数startで比較の開始位置、引数endで比較の停止位置を指定することもできる**
使用例	**ok = s.startswith('python')**

◎実行例

```
>>> 'hello_python.py'.startswith('hello') Enter
True
>>> 'hello_python.py'.startswith('ello', 1, 4) Enter
False
>>> 'hello_python.py'.startswith('ello', 1, 5) Enter
True
```

文字列の最後が指定した文字列で終わるかを調べるには、**endswith**メソッドを使用します。

◎メソッド　endswith(suffix[, start[, end]])

カテゴリ	**strクラス**
引数	**suffix：検索する文字列、start：開始位置、end：終了位置**
戻り値	**一致すればTrue、しなければFalse**
説明	**文字列の最後と引数suffixを比較し一致すればTrueを、そうでなければfalseを戻す。引数startで比較の開始位置、引数endで比較の停止位置を指定することもできる**
使用例	**end = s.endswith('.html')**

次に、文字列の拡張子が「.py」であるかを調べる例を示します。

◎実行例

```
>>> 'hello_python.py'.endswith('.py') Enter
True
```

●先頭／終わりの文字列をタプルで指定する

startswithメソッド、endswithメソッドの引数は、タプルの要素として複数指定してもかまいません。その場合、いずれかの要素に一致した場合にTrueを戻します。

◎実行例

```
>>> 'Goodboy'.startswith(('Good', 'Bad')) Enter
True
>>> 'Badboy'.startswith(('Good', 'Bad')) Enter
True
>>> 'Lonelyboy'.startswith(('Good', 'Bad')) Enter
False
```

次に、タプルfilenamesに格納した複数のファイル名から、拡張子が「.txt」と「.html」のどちらかであるファイル名を表示する例を示します。

◎リスト endswith1.py

```
filenames = ('test.py', 'sample.txt', 'index.html', 'info.html', "new.js")

for file in filenames:
    if file.endswith(('.txt', '.html')): 1
        print(file)
```

1で、endswithメソッドの引数にタプル('.txt', '.html')を指定しています。

■実行結果

```
sample.txt
index.html
info.html
```

➡022
文字列を左寄せ／中央寄せ／右寄せにするには

ljustメソッド、centerメソッド、rjustメソッドを使用する

文字列を格納する幅を広げて文字列を左寄せにするには**ljust**メソッドを使用します。

◎メソッド　ljust(width[, fillchar])

カテゴリ	**strクラス**
引数	**width：幅、fillchar：余ったスペースを埋める文字**
戻り値	**文字列が左寄せになった文字列**
説明	**文字列を引数widthで指定した幅に左寄せにして戻す。引数widthが元の文字列の長さ以下であれば元の文字列が返される**
使用例	**s = s.ljust(20)**

引数fillcharを指定しない場合には、右側がスペースで埋められます。引数fillcharを指定した場合にはその文字で埋められます。

◎実行例

```
>>> 'Python'.ljust(10) Enter      ←引数fillcharを指定しない
'Python    '   ←文字の右側がスペースで埋められる
>>> 'Python'.ljust(10, '-') Enter      ←引数fillcharにハイフン「-」を指定
'Python----'   ←文字の右側がハイフン「-」で埋められる
```

中央寄せにするには**center**メソッドを使用します。

◎メソッド　center(width[, fillchar])

カテゴリ	**strクラス**
引数	**width：幅、fillchar：余ったスペースを埋める文字**
戻り値	**文字列が中央寄せになった文字列**

説明	**文字列を引数widthで指定した幅に中央寄せにして戻す**
使用例	**s = s.center(15)**

次に、10文字分の幅に中央寄せにする例を示します。

◎実行例

```
>>> 'Python'.center(10, '-') Enter
'--Python--'
```

右寄せにするには**rjust**メソッドを使用します。

◎メソッド　rjust(width[, fillchar])

カテゴリ	**strクラス**
引数	**width：幅、fillchar：余ったスペースを埋める文字**
戻り値	**文字列が右寄せになった文字列**
説明	**文字列を引数widthで指定した幅に右寄せにして戻す**
使用例	**s = s.rjust(15)**

◎実行例

```
>>> 'Python'.rjust(10, '-') Enter      ←10文字分の幅に右寄せにする
'----Python'
```

➡023

文字列の先頭を「0」で埋めるには

zfillメソッドを使用する

rjustに似たメソッドに**zfill**があります。zfillの場合、文字列を右寄せにして左側を「0」で埋めます。zfillメソッドは数値を桁合わせして表示したい場合に便利です。

◎メソッド　**zfill(width)**

カテゴリ	strクラス
引数	幅
戻り値	先頭を「0」で埋め文字列が右寄せになった文字列
説明	文字列を指定した幅で右寄せし先頭を'０'で埋める
使用例	s = "44".zfill(4)

◎実行例

```
>>> '33'.zfill(6) Enter
'000033'
>>> '1045'.zfill(6) Enter
'001045'
```

　rjustメソッドと異なり、先頭が「+」「-」などの符号の場合は「0」の前に表示されます。

◎実行例

```
>>> '-33'.zfill(6) Enter
'-00033'
>>> '+33'.zfill(6) Enter
'+00033'
```

➡024
文字列を指定したエンコーディングでエンコード／デコードするには

encodeメソッド、decodeメソッドを使用する

●エンコードする

encodeメソッドを使用すると、文字列を指定したエンコーディングでエンコードしたバイト列オブジェクトを生成できます。

◎メソッド encode(encoding="utf-8"[, errors="strict"])

カテゴリ	**strクラス**
引数	**encoding：エンコーディング、errors：エラー処理の方法**
戻り値	**エンコーディング後のバイト列オブジェクト**
説明	**文字列をencodingで指定したエンコーディングでエンコードする**
使用例	**en = s.encode()**

次にutf-8と、shift-jisでエンコーディングする例を示します。

◎実行例

```
>>> "こんにちは".encode() Enter　←デフォルトはutf-8
b'\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf'
>>> "こんにちは".encode(encoding="shift-jis") Enter ←shift-jisでエンコード
b'\x82\xb1\x82\xf1\x82\xc9\x82\xbf\x82\xcd'
```

⦿利用可能なエンコーディングの例

エンコーディング	説明
ascii	アスキー文字
euc_jp	日本語EUC
iso2022_jp	JIS
shift_jis	シフトJIS

エンコーディング	説明
utf_32	UTF32
utf_16	UTF16
utf_8	UTF8

●デコードする

バイト列オブジェクトをデコードするには**decode**メソッドを使用します。

◎ メソッド decode(encoding='utf-8'[, errors='strict'])

カテゴリ	**bytesクラス**
引数	**encoding：エンコーディング、errors：エラー処理の方法**
戻り値	**デコード後の文字列**
説明	**バイト列オブジェクトを引数encodingで指定したエンコーディングでデコードして戻す**
使用例	**s = estr.decode()**

次に、shift-jisでエンコーディングされたバイト列をデコードする例を示します。

◎実行例

```
>>> s = 'こんにちは'.encode(encoding='shift-jis') Enter
>>> s.decode('shift-jis') Enter
'こんにちは'
```

ＮＯＴＥ decodeメソッドの代わりにstr関数を、引数encodingによりエンコーディングを指定して実行してもデコードできます。

```
>>> str(s, encoding='shift_jis') Enter
'こんにちは'
```

●エラー処理の方法を指定する

encodeメソッドおよびdecodeメソッドの引数**errors**では、エラーが発生した場合の処理の方法を指定できます。

⦿引数errorsの値

値	説明
strict	UnicodeError例外を送出
ignore	無視する
replace	エラーが発生した文字を「?」に置き換える

　たとえば、絵文字'😄'をshift_jisでエンコードしようとするとエラーになります。

◎実行例

```
>>> 'はい😄'.encode(encoding='shift_jis') Enter
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
UnicodeEncodeError: 'shift_jis' codec can't encode character '\U0001f604' in position 2: illegal multibyte sequence
```

　エラーの起こった文字を「?」に変更するには引数errorsに'replace'を指定します。

◎実行例

```
>>> 'はい😄'.encode(encoding='shift_jis', errors='replace') Enter
b'\x82\xcd\x82\xa2?'
```

➡025

文字とユニコードのコードポイントを相互変換するには

ord関数、chr関数を使用する

　指定した文字のユニコードのコードポイントを表示するには、**ord**関数を使用します。

◎関数　**ord(c)**

カテゴリ	**組み込み関数**
引数	**文字**
戻り値	**コードポイントを表す整数**
説明	**引数cのコードポイントを表す値を戻す**
使用例	**cp = ord('😄')**

◎実行例

```
>>> ord("a") Enter
97
>>> ord('😄') Enter
128516
```

結果は10進数です。16進数表記にしたければ結果をhex関数に渡します。

◎実行例

```
>>> hex(ord('😄')) Enter
'0x1f604'
```

逆に、コードポイントの値を対応する文字に変換するには**chr**関数を使用します。

◎関数　**chr(i)**

カテゴリ	**組み込み関数**
引数	**文字**
戻り値	**コードポイントを表す整数**
説明	**コードポイントが引数iである文字を戻す**
使用例	**c = chr(97)**

◎実行例

```
>>> chr(0x1f604) Enter
```

'😄'

➡026

タブを展開するには

expandtabsメソッドを使用する

　文字列内のタブ（\t）を指定したタブ幅のスペースに展開するには、**expandtabs**メソッドを使用します。

◎メソッド　epandtab(tabsize=8)

カテゴリ	**strクラス**
引数	**タブ幅**
戻り値	**展開後の文字列**
説明	**文字列内のタブを引数tabsizeのタブ幅のスペースで展開した文字列を戻す**
使用例	**st = s.expandtabs()**

　デフォルトのタブ幅は8になります。

◎実行例

```
>>> "Hi\tLow\tGood\tOK".expandtabs() Enter
'Hi          Low         Good        OK'
```

　引数tabsizeでタブ幅を指定できます。

◎実行例

```
>>> "Hi\tLow\tGood\tOK".expandtabs(tabsize=6) Enter
'Hi      Low    Good   OK'
>>> "Hi\tLow\tGood\tOK".expandtabs(tabsize=12) Enter
```

'Hi Low Good OK'

1-2 データの基本操作

この節では、Pythonにおける基本的なデータを取り扱う上で便利な関数やメソッドなどについてまとめておきます。

➡027

値のデータ型を調べるには

type関数、isinstance関数を使用する

オブジェクトのデータ型は、**type**関数でわかります。

◎関数　type(object)

カテゴリ	**組み込み関数**
引数	**調べるオブジェクト**
戻り値	**オブジェクトの型**
説明	**引数で指定したオブジェクトの型を返す**
使用例	**t = type(s)**

◎実行例

```
>>> num = 15 Enter
>>> type(num) Enter
<class 'int'>
>>> t = (1, 2, 3) Enter
>>> type(t) Enter
<class 'tuple'>
```

リテラルを引数にして実行してもかまいません。

◎実行例

```
>>> type("hello") Enter
<class 'str'>
>>> type([3, 4, 5]) Enter
<class 'list'>
```

ＮＯＴＥ 特殊変数「__class__」を使用してクラスを調べることもできます。

```
>>> "Hello".__class__ Enter
<class 'str'>
>>> "Hello".__class__.__name__ Enter
'str'
```

●オブジェクトがクラスのインスタンスであることを調べる

　オブジェクトがあるクラスのインスタンスであることを調べるには、**isinstance**関数を使用します。

◎関数　isinstance(object, classinfo)

カテゴリ	**組み込み関数**
引数	**object：調べる値、classinfo：クラス名**
戻り値	**TrueもしくはFalse**
説明	**オブジェクトが引数で指定したクラス（およびそこから派生するクラス）のインスタンスであればTrueを返す**
使用例	**r = isinstance(19, int)**

　たとえば、文字列"hello"がstrクラスのインスタンスであることを調べるには次のようにします。

◎実行例

```
>>> isinstance("hello", str) Enter
True
```

前出のtype関数と異なり、isinstance関数は継承元のクラスも調べられます。たとえば、calendarモジュールのTextCalendarクラスは、Calendarクラスのサブクラスです。次のようにTextCalendarクラスのインスタンスtcalについてTextCalendarおよびCalendarを引数にして実行すると、どちらもTrueとなります。

◎実行例

```
>>> from calendar import TextCalendar, Calendar Enter
>>> tcal = TextCalendar() Enter
>>> tcal.prmonth(2019, 2) Enter
February 2019
Mo Tu We Th Fr Sa Su
             1  2  3
4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28
>>> isinstance(tcal, TextCalendar) Enter
True
>>> isinstance(tcal, Calendar) Enter
True
```

➡028

オブジェクトのidを調べるには

id関数を使用する

Pythonではすべてのオブジェクトに重複しないid番号が振られます。id番号は**id**関数で調べることができます。

◎関数　id(object)

カテゴリ	組み込み関数
引数	調べるオブジェクト
戻り値	id番号（整数値）
説明	オブジェクトのid番号を戻す
使用例	print(id(s))

id番号はオブジェクトの有効期間中は一意になります。

◎実行例

```
>>> str1 = "hello" Enter
>>> id(str1) Enter
4368043736
```

既存の変数を別の変数に代入した場合、オブジェクトの参照先は同じになるためid番号も同じです。

◎実行例

```
>>> num1 = 3 Enter
>>> num2 = num1 Enter
>>> id(num1) Enter
4305280096
>>> id(num2) Enter
4305280096
```

➡029
値を画面に表示するには

print関数を使用する

引数で指定した値を画面に表示するには、**print**関数を使用します。

◎ 関 数　print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

カテゴリ	**組み込み関数**
引数	**object：出力する値、sep：セパレータ（デフォルトはスペース）、end：最後に出力する文字（デフォルトは改行）、file：出力先（デフォルトは標準出力）、flush：出力を強制的にフラッシュするかどうか（デフォルトは出力先に応じて自動で決定される）**
戻り値	**なし**
説明	**引数で指定した値を、引数sepで区切りながら引数fileに出力する**
使用例	**print("java", "python", sep="&")**

複数の引数を指定した場合、デフォルトでは半角スペースで区切られて表示されます。

◎実行例

```
>>> print("hello", "python") Enter
hello python
```

区切り文字は引数sepで設定可能です。たとえば、引数を"&"で区切って表示するには「sep="&"」を追加します。

◎実行例

```
>>> print("hello", "python", sep="&") Enter
hello&python
```

●改行を抑制する

デフォルトでは最後に改行"\n"が出力されますが、引数endを空文字列""にすることで改行を抑制できます。

◎リスト print1.py

```
print("こんにちは。", end="")
print("Pythonの世界へ", end="")
print("ようこそ！")
```

◎実行結果

```
こんにちは。Pythonの世界へようこそ！
```

次に、リストcolorsの要素を1行に3つずつ表示する例を示します。

◎リスト print2.py

```
colors = ["赤", "青", "黄", "緑", "黒", "白", "紫", "灰", "ピンク"]

for i, c in enumerate(colors): 1
    if (i + 1) % 3 == 0: 2
        print(c)
    else:
        print(c, end=" ") 3
```

1のfor文でenumerate関数（058で解説）を使用し、リストcolorsの要素のインデックスと値を1つずつ取り出し、それぞれ変数iと変数cに代入しています。2のif文で「i + 1」が3の倍数、つまり要素3つごとに改行付きで出力しています。そうでなければ3で引数に「end=" "」を指定して改行なしで出力しています。

■実行結果

```
赤 青 黄
緑 黒 白
紫 灰 ピンク
```

●ファイルに出力する

　引数**file**に、ファイルオブジェクトを指定すると、ファイルに出力できます。

◎リスト print3.py

```
f = open("test.txt", "w") 1
print("Pythonの世界へようこそ", file=f) 2
f.close()
```

1でopen関数を使用してtest.txtを書き込みモードで開き、**2**のprint関数で"Pythonの世界へようこそ"を出力しています。

■実行結果　test.txt

```
Pythonの世界へようこそ
```

➡030
文字列をPythonの式や文として実行するには
eval関数、exec関数を使用する

◎関数　eval(expression[, globals[, locals]])

カテゴリ	組み込み関数
引数	expression：式（文字列）、globals：グローバルスコープの名前を管理する辞書、locals：ローカルスコープ

	の名前を管理する辞書
戻り値	式の実行結果
説明	引数expressionで与えられた式を評価して実行結果を戻す
使用例	s = eval("str1 + str2")

たとえば、文字列"3 + 10"を式として実行するには次のようにします。

◎実行例

```
>>> eval("3 + 10") Enter
13
```

変数を使用することもできます。引数globals、引数localsが省略された場合、現在のスコープで実行されます。

◎実行例

```
>>> num = 15 Enter
>>> eval("num * 4") Enter
60
```

●変数を辞書として与える

引数**globals**ではグローバルスコープ、引数**locals**ではローカルスコープが管理する名前と値を辞書として与えることができます。これを使用すると、たとえば、引数expressionの式に、直接変数を渡すことができます。

次に変数nameを「"Hello"」に設定し、"name * 4"という式を実行する例を示します。

◎実行例

```
>>> eval("name * 4", {"name": "Hello"}) Enter
```

```
'HelloHelloHelloHello'
```

●文を実行する

　eval関数に与えられるのは式だけです。たとえば、次のように文を引数にすると、エラーになります。

◎実行例

```
>>> eval("x = 3 * 4") Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    x = 3 * 4
      ^
SyntaxError: invalid syntax
```

　文を実行したい場合には**exec**関数を使用します。

◎関数　exec(object[, globals[, locals]])

カテゴリ	**組み込み関数**
引数	**object：Pythonのプログラムコード（文字列）、globals：グローバルスコープの名前を管理する辞書、locals：ローカルスコープの名前を管理する辞書**
戻り値	**なし**
説明	**引数objectで与えられたPythonプログラムを実行する**
使用例	**exec("x = 3 * 4")**

◎実行例

```
>>> exec("x = 3 * 4") Enter
>>> x Enter
12
```

　複数の文を実行するには途中に改行"\n"を入れます。

◎実行例

```
>>> exec("x = 3\ny =4\nprint(x +y)") Enter
7
```

●プログラムファイルを読み込んで実行する

　exec関数を使用すると、Pythonプログラムのソースファイルの内容を読み込んで実行することもできます。次のようなPythonのソースファイルがあるとします。

◎リスト infile.py

```
print("こんにちは")
print("これはinfile.pyです")
```

　これを読み込んで実行するには次のようにします。

◎リスト exec1.py

```
f = open("infile.py", encoding="utf8")
cmd = f.read()
exec(cmd)
```

■実行結果

```
こんにちは
これはinfile.pyです
```

➡031
実行中のプログラムファイルのパスを調べるには

__file__変数を使用する

組み込み変数「**__file__**」には、実行中のプログラムファイルのパスが格納されています。「__file__」のパスはカレントディレクトリからの相対パスになります。

◎リスト file1.py

```
print(__file__)
```

上記のプログラムがカレントディレクトリにある場合、実行結果は次のようになります。

■実行結果

```
file1.py
```

➡032

実行中のプログラムファイルの絶対パスや名前を調べるには

__file__をabspath関数、basename関数、dirname関数に渡す

現在実行中のプログラムファイルの絶対パスを求めるには、前述の__file__を**os.path**モジュールの**abspath**関数に渡します。ファイル名を求めるには**basename**関数（116を参照）に_ _file_ _を渡します。ディレクトリまでの絶対パスを

求めるにはabspathの結果を**dirname**関数（116を参照）に渡します。次に使用例を示します。

◎リスト file2.py

```
import os.path
# 絶対パス
print(os.path.abspath(__file__))
# ファイル名
print(os.path.basename(__file__))
# ディレクトリまでの絶対パス
print(os.path.dirname(os.path.abspath(__file__)))
```

次に、ユーザー「o2」のDocumentsディレクトリの下の「Python Scripts」ディレクトリに「file2.py」が保存されている場合の実行結果を示します。

■実行結果

```
・macOS
/Users/o2/Documents/Python Scripts/file2.py
file2.py
/Users/o2/Documents/Python Scripts
・Windows
C:\Users\o2\Documents\Python Scripts\file2.py
file2.py
C:\Users\o2\Documents\Python Scripts
```

NOTE パスの区切り文字は、OSによって異なります。Windowsの場合には「\」、macOS/Linuxの場合には「/」になります。

条件判断を簡潔に行うには

三項演算子を使用する

三項演算子を使用することで、条件判断を簡潔に記述できます。書式は次の通りです。

変数 = 条件式がTrueの場合の値 if 条件式 else 条件式がFalseのときの値

たとえば、次のようなif~else文があったとします。

◎リスト if1.py

```
age = 10
if age >= 20:
    adult = True
else:
    adult = False
print(adult)
```

これを三項演算子で記述すると次のようになります。

◎リスト if2.py

```
age = 10
adult = True if age >= 20 else False
print(adult)
```

ステートメントの途中で改行するには

ステートメントが長くなる場合でも、三項演算子の途中にそのまま改行を入れることはできません。改行を入れたい場合には行末に「\」を記述します。

◎NGな例

```
adult = True
if age >= 20 else False
```

◎OKな例

```
adult = True \
if age >= 20 else False
```

➡034

サブクラスかどうかを調べるには

issubclass関数を使用する

あるクラスが別のクラスのサブクラスかどうかを調べるには、**issubclass**関数を使用します。

◎関数　issubclass(class, classinfo)

カテゴリ	**組み込み関数**
引数	**class：調べるクラス、classinfo：元になるクラス**
戻り値	**TrueもしくはFalse**
説明	**引数classが引数classinfoのサブクラスであればTrueを戻す**
使用例	**r = issubclass(MyNewClass, MyClass)**

たとえば、calendarモジュールのTextCalendarクラスはCaledarクラスのサブクラスです。次のようにして調べられま

す。

◎実行例

```
>>> import calendar Enter
>>> issubclass(calendar.TextCalendar, calendar.Calendar) Enter
True
```

➡035

グローバル、ローカルのシンボルテーブルを表示するには

globals、locals関数を使用する

　グローバルスコープのシンボルテーブル（変数名や関数名）を表示するには、**globals**関数を使用します。

◎関数　globals()

カテゴリ	**組み込み関数**
引数	**なし**
戻り値	**グローバルのシンボルテーブルの辞書**
説明	**グローバルスコープのシンボルテーブルを辞書オブジェクトとして戻す**
使用例	**g = globals()**

　ローカルスコープのシンボルテーブルを表示するには、**locals**関数を使用します。

◎関数　locals()

カテゴリ	**組み込み関数**
引数	**なし**
戻り値	**ローカルのシンボルテーブルの辞書**

説明	ローカルスコープのシンボルテーブルを辞書オブジェクトとして戻す
使用例	l = locals()

どちらも戻り値は辞書データになります。次に関数testを定義し、関数test内でlocals関数の結果を、関数testの外部でglobals関数の結果を表示する例を示します。

◎リスト global1.py

```
def test(num):
    msg = "hello " * num
    print(msg)
    print("----locals-----")
    print(locals())

globalnum1 = 3
test(globalnum1)
print("----globals-----")
print(globals())
```

■実行結果

```
hello hello hello
----locals-----
{'msg': 'hello hello hello ', 'num': 3}
----globals-----
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x101dbb2e8>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': '/Users/o2/Documents/Work/PythonLib/Chap1/samples1-5/global1.py', '__cached__': None, 'test': <function test at 0x101d62e18>, 'globalnum1': 3}
```

➡036

コマンドラインから入力を受け取るには

input関数を使用する

プログラムの実行中に、ユーザーがコマンドラインから入力した文字列を取得するには**input**関数を使用します。

◎関数　input([prompt])

カテゴリ	**組み込み関数**
引数	**プロンプトとして表示する文字列**
戻り値	**入力した文字列**
説明	**コマンドラインにプロンプトを表示し、ユーザーが入力した行を文字列として戻す**
使用例	**name = input("名前は?: ")**

ユーザーが入力した行の末尾の改行は取り除かれます。

◎実行例

```
>>> answer = input("答えは？: ") Enter
答えは？: Hello Enter
>>> answer Enter
'Hello'
```

➡037

2つの値を比較するには

比較演算子を使用する

比較演算子を使用すると2つの値の大小を比較できます。

⦿比較演算子

演算子	例	説明
<	a < b	aがbより小さい
<=	a <= b	aがb以下
>	a > b	aがbより大きい
>=	a >= b	aがb以上
==	a == b	aとbが等しい
!=	a != b	aとbが等しくない

比較演算子は左辺と右辺の値を比較し、TrueもしくはFalseを返します。

◎実行例

```
>>> 3 < 4 Enter
True
>>> "abc" == "bcd" Enter
False
>>> (3, 2) < (3, 2, 3) Enter
True
>>> l1 = ["東京都", "千葉県", "神奈川県"] Enter
>>> l2 = ["東京都", "千葉県", "神奈川県"] Enter
>>> l1 == l2 Enter
True
```

●異なる型を比較した場合

整数と浮動小数点数を比較した場合には、整数が浮動小数点数に変換されて比較されます。

◎実行例

```
>>> 3 == 3.0 Enter
```

```
True
```

数値以外の異なる型同士を「==」もしくは「!=」で比較した場合には、必ず異なる値と判断されます。たとえば、「==」演算子でリストとタプルを比較すると要素が同じ場合でもFalseとなります。

◎実行例

```
>>> (3, 2, 1) == (3, 2, 1) Enter　←タプル同士を比較
True
>>> (3, 2, 1) == [3, 2, 1] Enter ←リストとタプルを比較
False
```

なお、「<」「<=」「>」「>=」演算子を使用して異なる型の値同士を比較した場合には**TypeError**例外が送出されます。

◎実行例

```
>>> {"春", "夏"} < ("春", "夏", "秋") Enter　←セットとタプルを比較
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'set' and 'tuple'
```

➡038

オブジェクトが同じかどうかを判断するには

is演算子、is not演算子を使用する

「==」や「!=」といった比較演算子は値が同じであるかをどうかを調べる演算子で、参照先のオブジェクトが異なっているかは判断しません。次の例は、変数l1と変数l2に同じ要素のリストを代入し「==」演算子で比較しています。結果はTrueとなります。

◎実行例

```
>>> l1 = ["東京都", "千葉県", "神奈川県"] Enter
>>> l2 = ["東京都", "千葉県", "神奈川県"] Enter
>>> l1 == l2 Enter
True
```

それに対して**is**演算子、**is not**演算子はオブジェクトそのものが同じかどうかを調べ、TrueもしくはFalseを戻す演算子です。

⦿オブジェクトを比較する演算子

演算子	説明	例
is	a is b	aとbのオブジェクトが同じ
is not	a is not b	aとbのオブジェクトが異なる

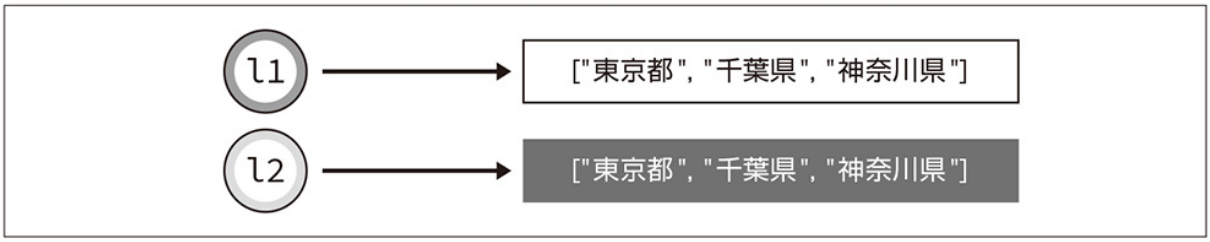
前述の変数l1と変数l2にis演算子を使用すると結果はFalseに、is not演算子を使用するとTrueとなります。

◎実行例

```
>>> l1 = ["東京都", "千葉県", "神奈川県"] Enter
>>> l2 = ["東京都", "千葉県", "神奈川県"] Enter
>>> l1 is l2 Enter
False
>>> l1 is not l2 Enter
True
```

変数l1と変数l2は、どちらも要素は同じリストですが、実体は異なるオブジェクトを参照するからです。

⊙変数l1と変数l2はリストの要素は同じでも異なるオブジェクトを参照している

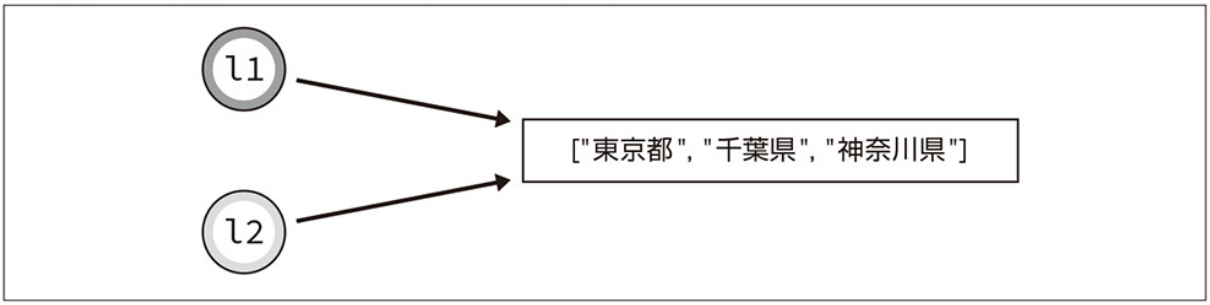


次のように、変数l1を変数l2に代入した場合には同じオブジェクトを参照することになり、is演算子の結果はTrueとなります。

◎実行例

```
>>> l1 = ["東京都", "千葉県", "神奈川県"] Enter
>>> l2 = l1 Enter
>>> l1 is l2 Enter
True
>>> l2 is not l1 Enter
False
```

⊙変数l1と変数l2は同じオブジェクトを参照する



➡039

ヘルプを表示するには

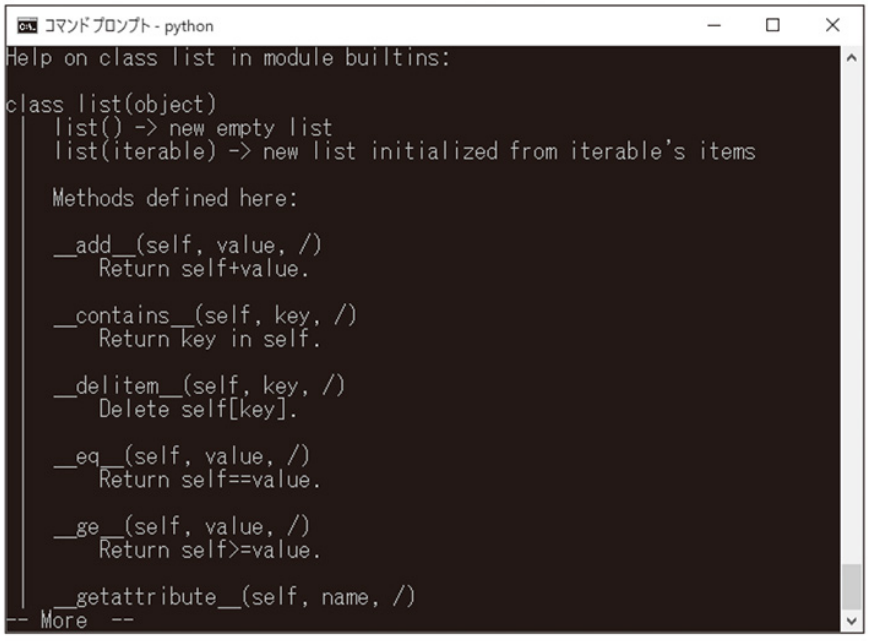
help関数を使う

◎関数　help(obj)

カテゴリ	**組み込み関数**
引数	**obj：オブジェクトもしくは文字列**
戻り値	**なし**
説明	**引数に関するヘルプを表示する**
使用例	**help(list)、help("http")など**

　引数がクラスやモジュール、関数などのオブジェクトの場合、そのヘルプメッセージが表示されます。たとえば、インタラクティブシェルで「help(list)Enter」とするとlistクラスに関するヘルプが表示されます。

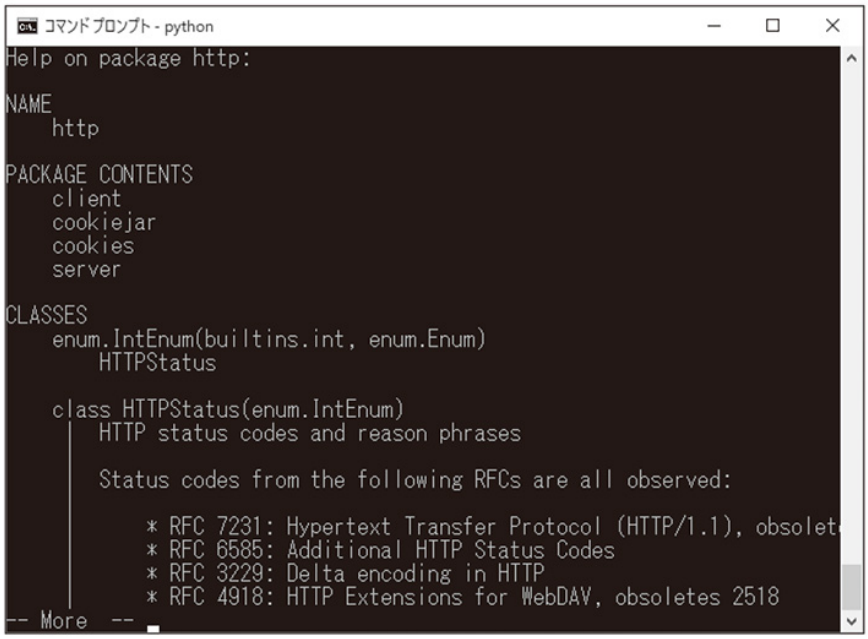
⊙listクラスのヘルプを表示



表示はOSに応じたページャで表示されます。Windowsのコマンドプロンプトの場合にはmoreページャで、macOSの場合にはlessページャで表示され、どちらもスペースキーで次のページに、Qキーで終了します。

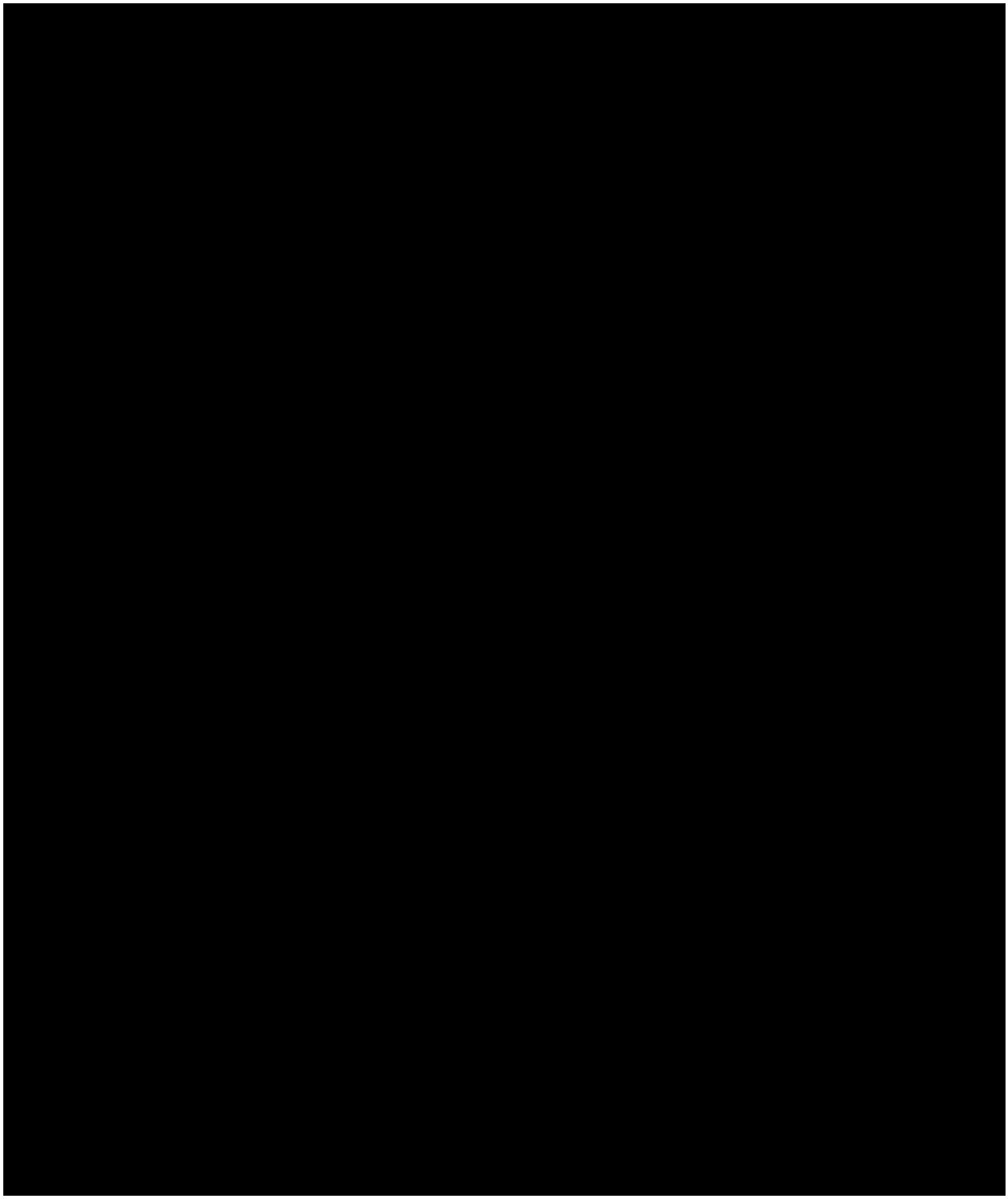
引数には文字列も指定できます。たとえば、httpモジュールに関するヘルプを表示したければ「help("http") 」を実行します。

⊙httpモジュールのヘルプを表示



NOTE 同じhttpモジュールのヘルプを表示するのに「help(http) Enter」とオブジェクトを引数にすることもできますが、その場合にはあらかじめhttpモジュールをインポートしておく必要があります。

Chapter 2



コレクションの
取り扱い

Pythonの標準ライブラリにはリスト（list）、タプル（tuple）、辞書（dict）、セット（set）といった一連のデータをまとめて扱う「コレクション」と呼ばれるタイプのデータ型が用意されています。このChapterではそれらの基本的な取り扱いを中心に説明します。

2-1 リスト、タプルの操作

この節では、シーケンス型の代表であるリストおよびタプルの基本操作について説明します。どちらも要素に変数名とインデックスでアクセスします。

➡040

リスト、タプルをリテラルとして生成するには

リストは [値1, 値2, 値3, ...] 、タプルは (値1, 値2, 値3, ...) を使用する

リスト(list)、**タプル**（tuple)とも複数の要素をまとめて管理するデータ型です。リストは要素をあとから変更できますが、タプルはできません。リストをリテラルで記述するには次の書式を使用します。

[値1, 値2, 値3, ...]

◎実行例

```
>>> l = [3, 4, 5, 6] Enter
>>> l Enter
[3, 4, 5, 6]
```

タプルをリテラルで記述するには次の書式を使用します。

(値1, 値2, 値3, ...)

◎**実行例**

```
>>> t = ("春", "夏", "秋", "冬") Enter
>>> t Enter
('春', '夏', '秋', '冬')
```

ＮＯＴＥ 実際には、タプルのリテラルはカンマ「,」で区切られた要素の並びです。次のように「( )」は省略可能な場合がほとんどです。

◎**実行例**

```
>>> seasons = "春", "夏", "秋", "冬" Enter
```

●要素が1つのタプルを生成する

要素が1つのタプルを生成するには、要素の後ろにカンマ「,」が必要になります。

◎**実行例**

```
>>> one = ("One", ) Enter
>>> one Enter
('One',)
```

●空のリスト、タプルを生成する

空のリストを生成するには、単に「[]」を記述します。

◎**実行例**

```
>>> emptylst = [] Enter
```

空のタプルを生成するには単に「()」を記述します。

◎**実行例**

```
>>> emptytuple = () Enter
```

➡041

リスト、タプルの要素数を求めるには

len関数を使用する

　リスト、タプルは文字列と同じくシーケンス型のデータなので、**len**関数で要素数が求まります。

◎関数　**len(s)**

カテゴリ	**組み込み関数**
引数	**リストまたはタプル**
戻り値	**要素数（整数値）**
説明	**引数sの要素数を戻す**
使用例	**l = len(["one", "two", "three"])**

◎実行例

```
>>> seasons = ("春", "夏", "秋", "冬") Enter
>>> len(seasons) Enter
4
```

➡042

リスト、タプルを連結するには

「+」演算子、「*」演算子を使用する

リスト、タプルを連結して新たにリスト、タプルを生成して戻すには「**+**」演算子を使用します。

◎実行例

```
>>> (1, 2, 3) + (4, 5) Enter
(1, 2, 3, 4, 5)
```

NOTE 連結できるのはリスト同士、もしくはタプル同士です。リストとタプルを連結することはできません。

●指定した回数を連結する

「*****」演算子を使用すると、指定した回数で、リスト、タプルを繰り返し連結できます。

◎実行例

```
>>> ["赤", "青", "黄"] * 3 Enter
['赤', '青', '黄', '赤', '青', '黄', '赤', '青', '黄']
```

➡043

リスト、タプルの指定した位置の要素を取り出すには

インデックスを使用する

インデックスを使用することで、指定した位置の要素を取り出すことができます。

リストもしくはタプル[インデックス]

インデックスは先頭の要素を0とする整数値です。

◎実行例

```
>>> weekdays = ['月', '火', '水', '木', '金', '土', '日'] Enter
>>> weekdays[0] Enter
'月'
>>> weekdays[3] Enter
'木'
```

最後の要素のインデックスは「**len関数の値 - 1**」になります。

◎実行例

```
>>> weekdays[len(weekdays) - 1] Enter
'日'
```

インデックスに、マイナスの数値を指定すると、リスト、タプルの最後から位置を指定して要素を取り出せます。最後の要素を「-1」とします。

◎実行例

```
>>> weekdays[-1] Enter          ← 最後の要素（weekdays[len(weekdays) - 1]と同じ）
'日'
>>> weekdays[-2] Enter      ←最後から2番目の要素
'土'
```

➡044

範囲を指定してリスト、タプルの要素を取り出すには

スライスを使用する

文字列と同じように、**スライス**（一部の要素）の範囲を指定して要素を取り出すことができます。

リストもしくはタプル[最初の要素のインデックス:最後の要素のインデックス＋1]

次に、リストweekdaysからインデックスが2から3までの要素を取り出す例を示します。

◎実行例

```
>>> weekdays = ['月', '火', '水', '木', '金', '土', '日'] Enter
>>> weekdays[2:4] Enter
['水', '木']
```

最初のインデックスを省略した場合には「0」が指定されているものとし、最後のインデックスを省略した場合には要素数が、指定されているものと見なされます。

◎実行例

```
>>> weekdays[:4] Enter       ←最初のインデックス0から3まで
['月', '火', '水', '木']
>>> weekdays[4:] Enter     ←インデックスが4以降
['金', '土', '日']
```

●途中の要素をスキップして取り出す

スライスの最後に、**ステップ数**（何番目ごとに取り出すか）を指定できます。

リストもしくはタプル[最初の要素のインデックス:最後の要素のインデックス＋1:ステップ数]

「最初の要素のインデックス」、「最後の要素のインデックス＋1」は省略可能です。次に例を示します。

◎実行例

```
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
>>> nums[::2] Enter
[0, 2, 4, 6, 8, 10]
>>> nums[1::2] Enter
[1, 3, 5, 7, 9]
>>> nums[0::3] Enter
[0, 3, 6, 9]
```

➡045

リスト、タプルの要素の順番を反転するには

スライス、reverseメソッドを使用する

スライスの最初と最後のインデックスを省略し、ステップ数に「-1」を指定して**[::-1]**とすると、リスト、タプルの要素の順番を反転したものを取得できます。

◎実行例

```
>>> nums = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) Enter
>>> nums[::-1] Enter
```

```
(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

●リストの要素の順番を直接反転する

　上記のようにスライスのステップ数に「-1」を指定した場合、元のリスト、タプルはそのままで、要素の順番を反転させた新たなリスト、タプルを生成して戻します。

　それに対して、リストの**reverse**メソッドを使用すると、リストの要素の順番を直接反転させます（新たなリストを生成しません）。

◎メソッド　reverse()

カテゴリ	**listクラス**
引数・戻り値	**なし**
説明	**リストの要素の順番を反転させる**
使用例	**l.reverse()**

◎実行例

```
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
>>> nums.reverse() Enter
>>> nums Enter
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

➡046
タプルとリストを相互変換するには

tupleコンストラクタ、listコンストラクタを使用する

　tupleコンストラクタの引数にリストを指定して実行すると、リストをタプルに変換できます。

◎**実行例**

```
>>> mylist = ["東京都", "神奈川県", "千葉県"] Enter
>>> mytuple = tuple(mylist) Enter
>>> type(mytuple) Enter   ←type関数で確認
<class 'tuple'>
```

listコンストラクタの引数にタプルを指定して実行すると、タプルをリストに変換できます。

◎**実行例**

```
>>> mytuple = ("アメリカ", "日本", "中国", "イギリス") Enter
>>> mylist = list(mytuple) Enter
>>> type(mylist) Enter
<class 'list'>
```

➡047

リストの要素の値を変更するには

インデックス、スライスを使用する

リストはミュータブル（変更可）なデータ型です。インデックスで指定した要素に、別の値を代入することで値を変更できます。

リスト[インデックス] = 値

◎**実行例**

```
>>> colors = ["blue", "green", "red", "yellow"] Enter
>>> colors[1] = "pink" Enter
>>> colors Enter
```

```
['blue', 'pink', 'red', 'yellow']
```

●スライスを使用して複数の要素を変更する

　スライスで範囲を指定して、複数の要素をまとめて入れ替えることができます。

リスト[最初の要素のインデックス:最後の要素のインデックス＋1:ステップ数] = 値

　値はリストやタプルで指定します。次に最初の2つの要素を0にする例を示します。

◎実行例

```
>>> nums = [1, 2, 3, 4, 5, 6, 7] Enter
>>> nums[0:2] = [0,0] Enter
>>> nums Enter
[0, 0, 3, 4, 5, 6, 7]
```

　代入する要素がスライスより少なくても、多くてもかまいません。

◎実行例

```
>>> nums = [1, 2, 3, 4, 5, 6, 7] Enter
>>> nums[0:4] = [0] Enter   ←代入する要素が少ない場合
>>> nums Enter
[0, 5, 6, 7]
>>> nums = [1, 2, 3, 4, 5, 6, 7] Enter
>>> nums[0:1] = [0, 0, 0, 0] Enter   ←代入する要素が多い場合
>>> nums Enter
[0, 0, 0, 0, 2, 3, 4, 5, 6, 7]
```

ステップ数を指定することもできます。次にリストnumsの奇数番目（インデックスが0、2、4...）の要素の値を「0」にする例を示します。

◎実行例

```
>>> nums = [1, 2, 3, 4, 5, 6, 7] Enter
>>> nums[::2] = [0, 0, 0, 0] Enter
>>> nums Enter
[0, 2, 0, 4, 0, 6, 0]
```

なお、ステップ数を指定した場合には、スライスの数と代入する要素の数が同じである必要があります。数が同じではない場合は、次のようにValueError例外が送出されます。

◎実行例

```
>>> nums[1::2] = [0] Enter   ←代入する要素が少ない
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 1 to extended slice of size 3
```

➡048

リストの最後に要素を追加するには

appendメソッド、extendメソッド、スライスを使用する

リストの最後に要素を1つ追加するには**append**メソッドを使用します。

◎メソッド　**append(x)**

カテゴリ	listクラス
引数	追加する値
戻り値	なし
説明	リストの最後に引数xで指定した要素を追加する
使用例	l.append(3)

◎実行例

```
>>> mylist = ["東京都", "神奈川県", "千葉県"] Enter
>>> mylist.append("北海道") Enter
>>> mylist Enter
['東京都', '神奈川県', '千葉県', '北海道']
```

●extendメソッドでリストを拡張する

extendメソッドを使用してリストを拡張することもできます。appendメソッドと異なり、extendメソッドの引数にはリストなどイテレート（繰り返し処理）可能なオブジェクトを指定し、その要素をまとめて追加します。

◎メソッド　**extend(t)**

カテゴリ	listクラス
引数	リストやタプルなどイテレート可能なオブジェクト
戻り値	なし
説明	リストの最後に別のリストやタプルの要素を追加する
使用例	l.extend([3,4])

次に、リストyearsに別のリストの要素を追加する例を示します。

◎実行例

```
>>> years = [2001, 2002] Enter
>>> years.extend([2003, 2004, 2005]) Enter
```

```
>>> years Enter
[2001, 2002, 2003, 2004, 2005]
```

　辞書を引数に指定してextendメソッドを実行した場合は、キーのみが追加されます。

◎実行例

```
>>> prefs = ["東京", "大阪"] Enter
>>> prefs.extend({"北海道":3, "千葉":4}) Enter
>>> prefs Enter
['東京', '大阪', '北海道', '千葉']
```

●スライスを指定してリストを拡張する

　次の形式でスライスを指定することでも、リストの最後に複数の要素を追加できます。

リスト[len(リスト):] = リストやタプル

◎実行例

```
>>> ages = [44, 55] Enter
>>> ages[len(ages):] = [31] Enter　←31を追加
>>> ages[len(ages):] = [44, 45] Enter　←44と45を追加
>>> ages Enter
[44, 55, 31, 44, 45]
```

➡049
リストの指定した位置に要素を挿入するには

insertメソッド、スライスを使用する

insertメソッドを使用すると、指定したインデックスの位置に要素を挿入できます。

◎メソッド　**insert(i, x)**

カテゴリ	**listクラス、tupleクラス**
引数	**i：インデックス、x：挿入する値**
戻り値	**なし**
説明	**引数iで指定したインデックス位置に、引数xに指定した要素を挿入する**
使用例	**prefs.insert(1, "山梨")**

要素を挿入すると、その位置以降の要素が後ろにずれます。次に実行例を示します。

◎実行例

```
>>> langs = ["c", "basic", "swift", "JavaScript"] Enter
>>> langs.insert(1, "python") Enter
>>> langs Enter
['c', 'python', 'basic', 'swift', 'JavaScript']
```

●スライスを指定して要素を挿入する

次のような形式でスライスを使用することで、リストの指定したインデックス位置に要素を追加することもできます。

リスト[インデックス:インデックス] = リストやタプル

次にリストagesのインデックスが1の位置に「13」と「14」を挿入する例を示します。

◎実行例

```
>>> ages = [44, 55] Enter
```

```
>>> ages[1:1] = (13, 14) Enter
>>> ages Enter
[44, 13, 14, 55]
```

右辺は、リストやタプルなどイテレート可能なオブジェクトである必要があります。たとえば、リストagesに1つの要素を挿入しようとして、整数を指定することはできません。

◎実行例

```
>>> ages[1:1] = 3          ←これを実行するとTypeErrorが発生
```

➡050

リストから指定した位置の要素を削除するには

del文、clearメソッド、popメソッドを使用する

リストの要素を削除するには**del文**を使用します。

del リストの要素

リストlangsからインデックス1の要素を削除するには、次のようにします。

◎実行例

```
>>> langs = ["c", "basic", "swift", "JavaScript"] Enter
>>> del langs[1] Enter
>>> langs Enter
['c', 'swift', 'JavaScript']
```

del文ではスライスを指定することもできます。たとえば、リストnumsからインデックス2から4までの要素を削除するには、次のようにします。

◎実行例

```
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
>>> del nums[2: 5] Enter
>>> nums Enter
[0, 1, 5, 6, 7, 8, 9, 10]
```

del文に「**リスト[:]**」を指定すると、すべての要素を削除できます。

◎実行例

```
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
>>> del nums[:] Enter
>>> nums Enter
[] Enter←リストが空になった
```

ステップ数を指定することもできます。たとえば、奇数番目（インデックスが0、2、4...）の要素を削除するには次のようにします。

◎実行例

```
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
>>> del nums[::2] Enter
>>> nums Enter
[1, 3, 5, 7, 9]
```

●clearメソッドでリストを空にする

clearメソッドを使用しても、要素をすべて削除できます。

◎メソッド　**clear()**

カテゴリ	**listクラス**
引数・戻り値	**なし**
説明	**リストの要素を空にする**
使用例	**l.clear()**

◎実行例

```
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
>>> nums.clear() Enter
>>> nums Enter
[]
```

●リスト自体を削除する

「del リスト[:]」の場合には、空のリストが残りますが、del文にリスト名を指定するとリスト自体を削除できます。

◎実行例

```
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
>>> del nums Enter　←リストnumsを削除
>>> nums Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'nums' is not defined ←リストnumsが未定義
```

●要素を削除してそれを取得する

popメソッドでは、指定した位置の要素を削除してそれを戻り値として返せます。

◎メソッド　**pop([i])**

カテゴリ	listクラス
引数	インデックス
戻り値	削除した要素
説明	引数iで指定したインデックス位置の要素を削除しそれを戻り値とする。引数iを指定しなければ最後の要素を削除して戻す
使用例	el = l.pop(3)

popメソッドを、引数を指定しないで実行した場合、最後の要素を削除してその値を戻します。

◎実行例

```
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
>>> last = nums.pop() Enter    ←最後の要素を削除
>>> last Enter
10
>>> first = nums.pop(0) Enter    ←最初の要素を削除
>>> first Enter
0
>>> nums Enter
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

➡051

指定した値の要素がリスト、タプルに存在するかを調べるには

in演算子、not in演算子、indexメソッドを使用する

ある値がリスト、タプルに含まれるかどうかを調べるには**in**演算子を使用します。

値 in リストまたはタプル

in演算子は、リストに値が含まれていればTrueを、そうでなければFalseを戻します。

◎実行例

```
>>> colors = ["blue", "green", "red", "yellow"] Enter
>>> "green" in colors Enter
True
>>> "black" in colors Enter
False
```

in演算子の代わりに**not in**演算子を使用すると、値が含まれていない場合にTrueを、含まれていた場合にはFalseを戻します。

◎実行例

```
>>> "green" not in colors Enter
False
```

●要素のインデックスを調べる

in演算子では要素が存在しているかだけを確認します。存在していた場合に、そのインデックスを取得したい場合には**index**メソッドを使用します。

◎メソッド　index(x[, i[, j]])

カテゴリ	**listクラス**
引数	**x：調べる値、i：開始位置のインデックス、j：終了位置のインデックス + 1**
戻り値	**検索された要素のインデックス**

説明	引数xで指定した値のインデックスを戻す。一致する値が見つからない場合にはValueError例外が送出される
使用例	idx = l.index("Hello")

一致する要素が複数ある場合、最初に見つかった要素のインデックスが戻されます。

◎実行例

```
>>> years = [1965, 1959, 2001, 1987, 1959, 2011] Enter
>>> years.index(1959) Enter
1
```

値が見つからなかった場合には、**ValueError**例外が送出されます。

◎実行例

```
>>> years.index(1985) Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 1985 is not in list
```

引数iで検索を開始する位置、引数jで終了位置を指定できます。次の例は3番目以降の要素から値が「1959」の要素を検索し、インデックスを戻します。

◎実行例

```
>>> years.index(1959, 2) Enter
4
```

リストから指定した値の要素を削除するには

removeメソッドを使用する

リストから引数で指定した値の要素を削除するには**remove**メソッドを使用します。

◎メソッド　remove(x)

カテゴリ	**listクラス**
引数	**削除する値**
戻り値	**なし**
説明	**リストから引数xで指定した値の要素を削除する。一致する値が見つからない場合にはValueError例外が送出される**
使用例	**colors.remove("red")**

一致する値が複数ある場合には最初の要素のみが削除されます。

◎実行例

```
>>> prefs = ["東京", "神奈川", "千葉", "東京", "東京", "大阪"] Enter
>>> prefs.remove("東京") Enter
>>> prefs Enter
['神奈川', '千葉', '東京', '東京', '大阪']
```

removeメソッドにより一致する要素をすべて削除するにはループを使用します。次に、リストyearsから値が「1959」の要素をすべて削除する例を示します。

◎リスト remove1.py

```
years = [1965, 1959, 2001, 1987, 1959, 2011, 1959]
```

```
while True: 1
    try:
        years.remove(1959) 2
    except ValueError:
        break 3

print(years)
```

1のwhileループで2を繰り返し、removeメソッドを使用して値が「1959」の要素を削除しています。ValueError例外が発生した場合には3でループを抜けます。

■実行結果

```
[1965, 2001, 1987, 2011]
```

NOTE filter関数（063で解説）やリストの内包表記（062で解説）を使用して、指定した値の要素をすべて削除したリストを生成することもできます。

●リストから重複した要素をすべて削除する

リストから重複した要素をすべて削除するには、リストを**set**コンストラクタの引数にしていったんセット（077で解説）を生成し、さらにlistコンストラクタの引数に渡します。

◎実行例

```
>>> l = [1, 9, 9, 1, 3, 5] Enter
>>> l = list(set(l)) Enter
>>> l Enter
[1, 3, 5, 9]
```

➡053

リスト、タプルの指定した値の要素の数を調べるには

countメソッドを使用する

countメソッドを使用すると、引数で指定した値の要素がリスト、タプルにいくつ含まれるかがわかります。

◎メソッド　**count(x)**

カテゴリ	**listクラス、tupleクラス**
引数	**調べる値**
戻り値	**要素の個数（整数値）**
説明	**引数xで指定した値がリストやタプルに含まれている個数を戻す**
使用例	**c = l.count("pink")**

次に「1959」がリストyearsにいくつ含まれているかを調べる例を示します。

◎実行例

```
>>> years = [1965, 1959, 2001, 1987, 1959, 2011] Enter
>>> years.count(1959) Enter
2
```

➡054

リスト、タプルの要素をソートするには

sorted関数、sortメソッドを使用する

sorted関数を使用するとリスト、タプルの要素をソートして新たなリストとして戻すことができます。

◎関数　**sorted(s, key=None, reverse=False)**

カテゴリ	**組み込み関数**
引数	**s：イテレート可能なオブジェクト、key：比較するキーとして使用する関数、reverse：True 昇順に並べる False 降順に並べる**
戻り値	**ソートされたリスト**
説明	**引数sで指定したリストやタプルを、引数keyで指定した関数をキーにソートして結果をリストとして戻す**
使用例	**sl = sorted(colors)**

引数**reverse**を指定しなければ昇順にソートされます。

◎実行例

```
>>> ages = [9, 15, 13, 1, 9, 77, 55, 44, 12] Enter
>>> sorted(ages) Enter
[1, 9, 9, 12, 13, 15, 44, 55, 77]
```

引数reverseをTrueにして実行すると降順にソートされます。

◎実行例

```
>>> sorted(ages, reverse=True) Enter
[77, 55, 44, 15, 13, 12, 9, 9, 1]
```

●ソートの基準となるキーを指定する

デフォルトでは、数値の場合には数値の順、文字列の場合には文字コードの順にソートされますが、引数**key**でソートの基準となる値を戻す関数／メソッドを指定できます。たと

えば、引数keyで文字列（strクラス）の**lower**メソッドを指定して、小文字に変換することで、大文字／小文字を区別しないソートを行えます。

◎実行例

```
>>> str1 = ["good", "bye", "Good", "Bye"] Enter
>>> sorted(str1) Enter
['Bye', 'Good', 'bye', 'good']
>>> sorted(str1, key=str.lower) Enter ←大文字/小文字を区別しないでソート
['bye', 'Bye', 'good', 'Good']
```

　ソートの順番は引数keyで指定した関数で設定できます。関数は処理対象の一つの要素を引数に取り、ソートに利用されるキーを返す必要があります。次の例は、get_age関数をキーとして使用し"**名前 年齢**"形式の要素のリストnamesを年齢順にソートします。

◎リスト sort1.py

```
names = ["山田太郎 15", "井上毅 5", "大津肇 11", "森勇 55", "林香 34"]

def get_age(element): 1
    return int(element.split()[1]) 2

sorted_names = sorted(names, key=get_age) 3
print(sorted_names)
```

1がget_age関数の定義です。**2**で**split**メソッド（100で解説）により要素をスペースで名前と年齢に分割し、年齢を整数に変換して戻しています。

3でget_age関数を引数keyに指定し、sorted関数を実行しています。

■実行結果

```
['井上毅 5', '大津肇 11', '山田太郎 15', '林香 34', '森勇 55']
```

●lambda式でキーを指定する

sorted関数の引数keyで指定する関数は**lambda**式（061のあとのコラム「lambda式」を参照）にすることもできます。以下はsort1.pyのget_age関数をlambda式に変更した例です。

◎リスト sort2.py

```
names = ["山田太郎 15", "井上毅 5", "大津肇 11", "森勇 55", "林香 34"]

sorted_names = sorted(names, key=lambda e1:int(e1.split()[1]) )
print(sorted_names)  1
```

1で引数keyにlambda式を設定してsorted関数を実行しています。

■実行結果

```
['井上毅 5', '大津肇 11', '山田太郎 15', '林香 34', '森勇 55']
```

●sortメソッドでリストの要素を直接ソートする

sorted関数ではソートした結果を新たなリストとして戻します。それに対し、listクラスの**sort**メソッドではリストの要素を直接並べ替えることができます。

◎メソッド　sort(key=None, reverse=False)

カテゴリ	**listクラス**
引数	**key：比較するキーとして使用する関数、reverse：True 昇順に並べる　False 降順に並べる**
戻り値	**なし**
説明	**引数keyで指定した関数をキーに、リストの要素をソートする**
使用例	**sorted(colors)**

引数keyと引数reverseはsorted関数と同じです。引数keyを設定することで並べ替えの基準となるキーを変更することができます。次に、前出のsort2.pyのsorted関数をsortメソッドに変更した例を示します。

◎リスト sort3.py

```
names = ["山田太郎 15", "井上毅 5", "大津肇 11", "森勇 55", "林香 34"]

names.sort(key=lambda e1:int(e1.split()[1]) ) 1
print(names)
```

1でsortメソッドを実行しています。新たなリストを生成することなく、リストnamesの要素が直接並べ替えられた点に注目してください。

■実行結果

```
['井上毅 5', '大津肇 11', '山田太郎 15', '林香 34', '森勇 55']
```

➡055
リストやタプルの要素の最小値／最大値を求めるには

max関数、min関数を使用する

リストやタプルの要素の中で最大値を求めるには、**max**関数を使用します。

◎関数　max(s[, key, default])

カテゴリ	**組み込み関数**
引数	**s：イテレート可能なオブジェクト、key：比較するキーとして使用する関数、default：引数sが空の場合に戻すデータ**
戻り値	**最大値**
説明	**引数sの要素の最大値を戻す。引数sが空の場合に引数defaultが設定されていないとValueError例外が送出される**
使用例	**mv = max(nums)**

◎実行例

```
>>> nums = [9, 5, 11, 3] Enter
>>> max(nums) Enter
11
```

リストやタプルの要素の中で最小値を求めるには**min**関数を使用します。

◎関数　min(s[, key, default])

カテゴリ	**組み込み関数**
引数	**s：イテレート可能なオブジェクト、key：比較するキーとして使用する関数、default：引数sが空の場合に戻すデータ**
戻り値	**最小値**
説明	**引数sの要素の最小値を戻す。引数sが空の場合に引数defaultが設定されていないとValueError例外が送出**

	される
使用例	**mv = min(nums)**

◎実行例

```
>>> nums = [9, 5, 11, 3] Enter
>>> min(nums) Enter
3
```

N O T E max関数、min関数の引数には数値や文字列など同じ型の値を複数指定することもできます。

実行例

```
>>> max(9, 5, 11, 3) Enter   ←数値の並びを引数にした
11
```

●比較の基準となる関数を指定する

　max/min関数ではsorted関数／sortメソッド（054で解説）と同じように、引数**key**で比較の基準となる値を戻す関数／メソッドを指定できます。

　次に、"**名前 年齢**"の形式の要素のリストnamesから、最も年齢が高い人の名前と年齢、および最も年齢が低い人の名前と年齢を表示する例を示します。

◎リスト maxmin.py

```
names = ["山田太郎 15", "井上毅 5", "大津肇 11", "森勇 55", "林香 34"]

max_age = max(names, key=lambda e1:int(e1.split()[1]) ) 1
print("max関数: " + max_age + "才")
min_age = min(names, key=lambda e1:int(e1.split()[1]) ) 2
print("min関数: " + min_age + "才")
```

1 **2**でそれぞれmax関数、min関数を実行しています。どちらも引数keyに対して、要素から年齢を取り出すlambda式を設定しています。

■実行結果

```
max関数: 森勇 55才
min関数: 井上毅 5才
```

➡056

リストをコピーするには

copyメソッドを使用する

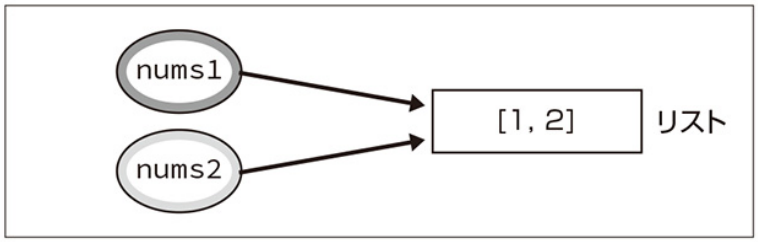
リストはミュータブル（変更可）なデータ型ですが、リストを別の変数に代入した場合に注意が必要です。一方の変数からリストを変更すると、もう一方の変数からアクセスした場合も変更されてしまいます。

◎実行例

```
>>> nums1 = [1, 2] Enter
>>> nums2 = nums1 Enter   ←nums1をnums2に代入
>>> nums1[0] = 10 Enter   ←nums1の要素を変更
>>> nums2 Enter
[10, 2] ←nums2の要素も変更される
```

このように変更されるのは、変数nums1と変数nums2が同じリストを参照しているからです。

⊙変数nums1と変数nums2は同じリストを参照している



　そうではなくて、リストをコピーして別の変数に代入するには**copy**メソッドを使用します。

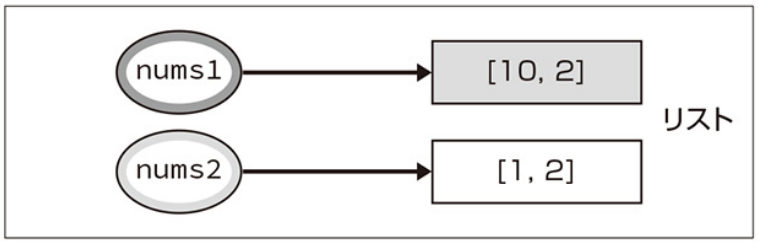
◎メソッド　**copy()**

カテゴリ	**listクラス**
引数	**なし**
戻り値	**リスト**
説明	**リストをコピーして戻す**
使用例	**nums2 = nums1.copy()**

◎実行例

```
>>> nums1 = [1, 2] Enter
>>> nums2 = nums1.copy() Enter　←nums1をnums2にコピー
>>> nums1[0] = 10 Enter　←nums1の要素を変更
>>> nums2 Enter
[1, 2] ←nums2の要素は変更されない
```

⊙変数nums1と変数nums2は別のリストを参照している



NOTE 次のようにスライスを使用してもリストをコピーできます。

実行例

```
>>> nums2 = nums1[:] Enter
```

➡057

リストやタプルの要素を順に取り出して処理するには

for文を使用する

リストやタプルなどイテレート可能なオブジェクトの要素を順に取り出して処理するには、**for**文を使用する方法が一般的です。for文の書式は次の通りです。

```
for 変数 in イテレート可能なオブジェクト:
    処理1
    処理2
    ....
```

次に、平成年が格納されているリストheiseisから要素を順に取り出して、西暦の年を表示する例を示します。

◎リスト heisei1.py

```
heiseis = [20, 15, 4, 29]
for h in heiseis: 1
    print("平成{}年は西暦{}年".format(h, h + 1988)) 2
```

1のfor文でリストheiseisから要素を順に取り出し変数hに格納しています。2のprint関数ではformatメソッドを使用し

て平成年と、それから計算した西暦を埋め込んで表示しています。

■実行結果

```
平成20年は西暦2008年
平成15年は西暦2003年
平成4年は西暦1992年
平成29年は西暦2017年
```

NOTE この例ではリストを使用していますが、次のようにタプルにしても同じです。

実行例

```
heiseis = (20, 15, 4, 29)
```

➡058

リストやタプルのインデックスと要素を順に取り出すには

enumerate関数を使用する

for文を使用して、リストやタプルのインデックスと要素のペアを順に取り出すには、**enumerate**関数を使用すると便利です。

◎関数　enumerate(I)

カテゴリ	**組み込み関数**
引数	**イテレート可能なオブジェクト**
戻り値	**enumerateオブジェクト**
説明	**リストやタプルなどのイテレート可能なオブジェクトを引数に、イテレート可能なenumerateオブジェクトを**

	戻す
使用例	**for i, c in enumerate(colors):　　print(i, c)**

　リストやタプルを引数にenumerate関数を実行すると、インデックスと要素のペアをタプルとして順に戻す、イテレート可能なenumerateオブジェクトを生成します。次のようにfor文と組み合わせてしばしば使用されます。

◎リスト enumerate1.py

```
countries = ("アメリカ", "日本", "中国", "イギリス")
for i, c in enumerate(countries): 1
    print(i, c)
```

1でfor文に、タプルcountriesを引数にしたenumerate関数を渡しています。変数iとcにはタプルのインデックスと値が順に代入されます。

■実行結果

```
0 アメリカ
1 日本
2 中国
3 イギリス
```

➡059
2つのリスト、タプルの要素を組み合わせてループで処理するには

zip関数を使用する

zip関数を使用すると、2つのリスト、タプルの要素をまとめてループで処理できます。

◎関数　zip(l1, l2)

カテゴリ	**組み込み関数**
引数	**l1：イテレート可能なオブジェクト、l2：イテレート可能なオブジェクト**
戻り値	**zipオブジェクト**
説明	**引数で指定した複数のイテレート可能なオブジェクトの要素を組み合わせた、zipオブジェクトを生成する**
使用例	**z = zip(l1, l2)**

zip関数は、2つのリスト、タプルから順に取得したタプルを要素とするイテレート可能なオブジェクトを作成して戻します。次に、リストl1とl2をzip関数で組み合わせてリストに変換する例を示します。

◎実行例

```
>>> l1 = [1, 3, 5, 7] Enter
>>> l2 = [10, 11, 22] Enter
>>> z = zip(l1, l2) Enter
>>> list(z) Enter
[(1, 10), (3, 11), (5, 22)]
```

上記のように要素数は、少ないほうのリストに合わせられます。したがってループで処理する場合、どちらかのリストの要素がすべて取り出されるとループが終了します。

◎リスト zip1.py

```
l1 = [1, 3, 5, 7] 1
l2 = [10, 11, 22] 2
```

```
for x, y in zip(l1, l2): 3
    print(x, y)
```

1のl1は要素数が4つ、2のl2は要素数が3つのリストです。3のようにfor文とzip関数を組み合わせて使用すると、要素数が少ないl2の要素に対する処理が完了した時点でループが終了します。

■実行結果

```
1 10
3 11
5 22
```

●3つ以上のリストを組み合わせる

zip関数では3つ以上のリスト、タプルの要素を組み合わせて処理することもできます。

◎リスト zip2.py

```
l1 = [1, 3, 5, 7]
l2 = [10, 11, 22]
l3 = [5, 4, 3, 2]
l4 = [100, 101, 202, 103]
for n1, n2, n3, n4 in zip(l1, l2, l3, l4):
    print(n1, n2, n3, n4)
```

この例では、l1、l2、l3、l4の4つのリストの要素をzip関数で組み合わせています。ループの回数は、最も要素数の少ないl2に合わせて3回になります。

■実行結果

```
1 10 5 100
```

```
3 11 4 101
5 22 3 202
```

NOTE zip関数で組み合わせるオブジェクトは、イテレート可能であれば異なるタイプのオブジェクトであってもかまいません。次のようにリストとタプルを引数にしてもかまいません。

実行例

```
>>> z = zip((1,2), [24, 23]) Enter
```

➡060

整数の並びのリストやタプルを生成するには

rangeオブジェクトをリスト／タプルに変換する

rangeオブジェクトは、整数の並びで構成されるイテレート可能なオブジェクトです。

◎コンストラクタ　**range(stop)**

◎コンストラクタ　**range(start, stop[, step]))**

カテゴリ	**rangeオブジェクト**
引数	**start：開始する値、stop：終了位置の次の値、step：ステップ数**
戻り値	**rangeオブジェクト**
説明	**引数startから引数stopの前の値までの整数の並びのrangeオブジェクトを生成する。引数stepではステップ数を指定できる**
使用例	**r = range(10)**

引数**stop**のみを指定した場合には0から始まります。たとえば、引数に「10」を指定した場合には0から9までの整数の並びのrangeオブジェクトが生成されます。

◎実行例

```
>>> for i in range(10): Enter
...         print(i) Enter
... Enter
0
1
2
～中略～
8
9
```

rangeオブジェクトを、listコンストラクタに渡すと整数の並びのリストを生成できます。

◎実行例

```
>>> list(range(1, 10)) Enter
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

引数**step**ではステップ数を指定できます。また、引数にはマイナスの値も指定できます。

◎実行例

```
>>> list(range(1, 21, 5)) Enter    ←引数stepにステップ数を指定
[1, 6, 11, 16]
>>> list(range(1, -10, -2)) Enter    ←引数にマイナスの値を指定
[1, -1, -3, -5, -7, -9]
```

➡061

リスト、タプルの数値の要素の合計を求めるには

sum関数を使用する

リスト、タプルの要素が数値である場合、**sum**関数を使用するとその合計を取得できます。

◎関数　sum(i[, start])

カテゴリ	**組み込み関数**
引数	**i：イテレート可能なオブジェクト、start：合計に加える値（デフォルトは0）**
戻り値	**要素の合計**
説明	**引数iで指定したリストやタプルの要素の合計を求める**
使用例	**s = sum(points)**

次にリストnumsの要素の合計を求める例を示します。

◎実行例

```
>>> nums = [1, 2, 9, 4, 5] Enter
>>> sum(nums) Enter
21
```

引数**start**では合計を計算する際の初期値とする数値を指定できます。次の例では10にリストnumsの合計が加えられます。

◎実行例

```
>>> sum(nums, 10) Enter
31
```

NOTE sum関数の引数に、rangeオブジェクトをすることで、一連の整数の合計を求めることができます。

```
>>> sum(range(10)) Enter　←1～9の和
45
```

lambda式

Pythonでは、**lambda**式（ラムダ式）を使用することでシンプルな関数を作成可能です。lambda式は「無名関数」と呼ばれる種類の関数で、文字通り名前のない関数です。sortメソッドやfilter関数など、リスト、タプル、辞書といったイテレート可能なオブジェクトの要素の処理に多用されます。lambda式の基本的な書式は次の通りです。

lambda 引数1, 引数2, 引数3, ...: 処理

たとえば、引数として与えられた2つの文字列の長いほうを戻す関数をdef文で定義する例は次のようになります。

◎実行例

```
>>> def longer(s1, s2): Enter
...     return s1 if len(s1) > len(s2) else s2 Enter
... Enter
```

これを、lambda式として定義し、変数longerに代入する例を示します。

◎実行例

```
>>> longer = lambda s1, s2: s1 if len(s1) > len(s2) else s2 Enter
```

以上で、無名関数をlongerという名前の関数のように呼び出すことができます。

◎実行例

```
>>> longer("abc", "hello python") Enter
'hello python'
>>> longer("good morning", "おはよう") Enter
'good morning'
```

➡062

シンプルな記述が可能な内包表記でリストを生成するには

リストの内包表記を使用する

内包表記と呼ばれる機能を使用するとリストを効率的に生成できます。**リストの内包表記**の書式は次の通りです。

[式 for 変数 in イテレート可能なオブジェクト if 条件式]

最後の「if 条件式」は必要がなければ省略可能です。
リストの内包表記はforループでリストの要素を追加していく処理の簡略形と考えると理解しやすいでしょう。たとえば、平成の年が要素として格納されているタプルから、対応する西暦の年を格納するリストを生成するforループの例は次のようになります。

◎リスト heisei2.py

```
heiseis = [20, 15, 4, 29, 1, 11, 10]
seirekis = []
for h in heiseis: 1
    seirekis.append(h + 1988) 2

print(seirekis)
```

1のforループでリストheiseisから要素を1つずつ取り出し、**2**でその値に1988を足し、appendメソッドでリストseirekisの要素に追加しています。

■実行結果

```
[2008, 2003, 1992, 2017, 1989, 1999, 1998]
```

これを内包表記で記述すると次のようになります。

◎リスト heisei3.py

```
heiseis = [20, 15, 4, 29, 1, 11, 10]
seirekis = [h + 1988 for h in heiseis] 1

print(seirekis)
```

1でheisei2.pyのfor文を内包表記に置き換えています。

■実行結果

```
[2008, 2003, 1992, 2017, 1989, 1999, 1998]
```

●条件に一致する要素を取り出す

リストの内包表記の最後に「**if 条件式**」を記述すると、条件に一致する要素のみを取り出すことができます。heisei3.py

を変更して、平成10年以降の年から、西暦の年のリストを生成するには次のようにします。

◎リスト heisei4.py

```
heiseis = [20, 15, 4, 29, 1, 11, 10]
seirekis = [h + 1988 for h in heiseis if h >= 10] 1
print(seirekis)
```

1で内包表記の最後に「if h >= 10」を加え、平成年が10以上の要素に絞り込んでいます。

■実行結果

```
[2008, 2003, 2017, 1999, 1998]
```

リストの内包表記で要素を取り出すオブジェクトは、イテレート可能なオブジェクトならばリスト以外でもかまいません。次に日本語の曜日をキーにして英語の曜日が格納された辞書weekdaysからキーを取り出し、「～曜日」の形式の要素を持つリストを生成する例を示します。

◎実行例

```
>>> weekdays = {"月":"Mo", "火":"Tu", "水":"We", "木":"Th",
"金":"Fr", "土":"Sat", "日":"Su"} Enter
>>> jpweekdays = [w + "曜日" for w in weekdays] Enter
>>> jpweekdays Enter
['月曜日', '火曜日', '水曜日', '木曜日', '金曜日', '土曜日', '日曜日']
```

➡063

関数を使用してリストやタプルの要素を抽出するには

filter関数を使用する

filter関数を使用すると、リストやタプルから条件に一致する要素を抽出できます。

◎関数　filter(f, l)

カテゴリ	**組み込み関数**
引数	**f：関数、l：イテレート可能なオブジェクト**
戻り値	**引数lの要素を抽出したfilterオブジェクト**
説明	**引数lで指定したリストやタプルの要素に対して、引数fで指定した関数を実行し結果がTrueとなる要素から構成されるfilterオブジェクトを戻す**
使用例	**years = list(filter(lambda h: h >= 10, heiseis))**

引数fで指定する関数はlambda式でもかまいません。前出のheisei4.pyでは、内包表記を使用して、平成10年以降の年から、西暦の年のリストを生成しましたが、これをfilter関数で記述すると次のようになります。

◎リスト heisei5.py

```
heiseis = [20, 15, 4, 29, 1, 11, 10]
seirekis = []

for s in filter(lambda h: h >= 10, heiseis): 1
    seirekis.append(s + 1988)

print(seirekis)
```

1でfilter関数を実行してリストheiseisの要素を絞り込んでいます。

■実行結果

```
[2008, 2003, 2017, 1999, 1998]
```

NOTE filter関数の戻り値は、イテレート可能なfilterオブジェクトです。リストに変換するにはlistコンストラクタに渡します。

```
>>> heiseis = [20, 15, 4, 29, 1, 11, 10] Enter
>>> years = list(filter(lambda h: h >= 10, heiseis))
Enter
>>> years Enter
[20, 15, 29, 11, 10]
```

➡064

関数を使用してリストやタプルの要素を順に処理するには

map関数を使用する

map関数を使用すると、リストやタプルの要素を順に処理して新たなオブジェクトとして取得できます。

◎関数　map(f, l)

カテゴリ	**組み込み関数**
引数	**f：関数、l：イテレート可能なオブジェクト**
戻り値	**引数lの要素を処理したmapオブジェクト**
説明	**引数lで指定したリストやタプルの要素に対して、引数fで指定した関数を実行し結果を要素とするmapオブジ**

エクトを戻す

使用例 **areas = list(map(lambda x: x*x, sides))**

map関数は前出のfilter関数と同じく、最初の引数に処理を行う関数を指定します。戻り値はイテレート可能なmapオブジェクトなります。リストに変換するにはlistコンストラクタに渡します。

次に、正方形の辺の長さが格納されているリストsidesから、面積のリストareasを生成する例を示します。

◎実行例

```
>>> sides = [4, 5, 9, 8] Enter
>>> areas = list(map(lambda x: x*x, sides)) Enter
>>> areas Enter
[16, 25, 81, 64]
```

NOTE これはリストの内包表記を使用すると次のように記述できます。

```
>>> areas = [x * x for x in sides] Enter
```

イテレート可能なオブジェクト

max関数や、strクラスのjoinメソッドのように、Pythonの関数やメソッドにはイテレート可能（iterable）なオブジェクトを引数に取るものが数多くあります。また、for文での繰り返しにも使用されます。

イテレート可能なオブジェクトには、文字列（str）、リスト（list）、タプル（tuple）、辞書（dict）、セット（set）などがあります。

「**イテレート**」（iterate）とは日本語では繰り返し処理するといった意味です。オブジェクトから順に要素を取り出せることを表します。

イテレート可能なオブジェクトは、たとえば、for文の実行時に「**イテレータ**」と呼ばれるタイプのオブジェクトに変換され、ループのたびに要素が順に取り出され、変数に格納されます。

イテレータとは**iter**クラスのインスタンスです。イテレート可能なオブジェクトは、iterコンストラクタを使用して明示的にイテレータに変換できます。

イテレータにnext関数を使用することで、次の要素が順に取り出せます。

◎関数　next(イテレータ)

説明	引数で指定したイテレータの次の要素を取得する

◎実行例

```
>>> l = ["東京都", "神奈川県", "千葉県"] Enter
>>> itr = iter(l) Enter　←イテレータに変換
>>> next(itr) Enter
'東京都'
>>> next(itr) Enter
'神奈川県'
>>> next(itr) Enter
'千葉県'
```

要素がなくなるとStopIteration例外が送出されます。

◎実行例

```
>>> next(itr) Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

2-2 辞書の操作

辞書(dict)は、キーと値のペアで要素を管理するイテレート可能なオブジェクトです。この節では辞書の基本操作について説明します。

➡065

辞書をリテラルとして生成するには

 {キー1:値1, キー2:値2, キー3:値3,....} を使用する

辞書オブジェクトをリテラルとして記述するには次の書式を使用します。

{キー1:値1, キー2:値2, キー3:値3,....}

「キー:値」をカンマで区切って並べ全体を「{ }」で囲みます。

次に、プログラム言語名をキーに、その得票数を値とする辞書の生成例を示します。好きなプログラム言語についてのアンケートの結果を格納したものと考えてください。

◎実行例

```
>>> langs = {"Python":105, "JavaScript":20, "Swift":3, "Basic":5, "C":33} Enter
```

●空の辞書を生成する

空のリストを生成するには単に「{}」を記述します。

◎実行例

```
>>> mydic = {} Enter
```

NOTE 辞書のキーはイミュータブルなオブジェクトでなければなりません。

```
s = {("東京", "世田谷"):100, ("千葉", "流山"):45}
←タプルはOK
s = {["東京", "世田谷"]:100, ["千葉", "流山"]:45}
←リストはNG
```

➡066

指定したキーに対応する要素を取得するには

「辞書[キー]」、getメソッド、setdefaultメソッドを使用する

辞書の要素にアクセスするには「[ ]」内に、キーを指定します。

辞書[キー]

◎実行例

```
>>> langs = {"Python":105, "JavaScript":20, "Swift":3, "Basic":5,
"C":33} Enter
>>> langs["Swift"] Enter
3
```

●キーが存在しない場合にエラーにならないようにする

「辞書[キー]」の形式でアクセスした場合、存在しないキーの値を取得しようとするとKeyError例外が送出されます。

◎実行例

```
>>> langs["Objective-C"] Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Objective-C'
```

エラーを避けるには**get**メソッドを使用します。

◎メソッド　get(key[, default])

カテゴリ	**dictクラス**
引数	**key：キー** **default：キーが存在しない場合のデフォルトの値（デフォルトはNone）**
戻り値	**キーの値**
説明	**引数keyで指定したキーの値を取得して戻す**
使用例	**l = langs.get("Objective-C")**

キーが存在しない場合にはエラーにならずデフォルトではNoneが戻されます。次に、辞書langsからgetメソッドを使用して存在しないキー「"Objective-C"」の値を取得する例を示します。

◎実行例

```
>>> langs = {"Python":105, "JavaScript":20, "Swift":3, "Basic":5, "C":33} Enter
>>> l = langs.get("Objective-C")      Enter   ← 存在しないキーを指定してもエラーにならない
>>> type(l) Enter
```

```
<class 'NoneType'> ←戻り値のタイプはNone
```

●キーが存在しない場合に新たな値を挿入する

getメソッドの代わりに、**setdefault**メソッドを使用するとキーが存在していた場合にはその値を返し、存在しなければキーにdefaultの値をセットし、その値を返します。

◎メソッド　setdefault(key[, default])

カテゴリ	**dictクラス**
引数	**key：キー** **default：キーが存在しない場合のデフォルトの値（デフォルトはNone）**
戻り値	**指定したキーに対する値**
説明	**キーが存在しない場合には値が引数defaultの要素を作成する**
使用例	**age = customer.setdefault("age")**

次に、辞書customerにsetdefaultメソッドを使用して、存在しないキー「"age"」の値を取得しようとした例を示します。

◎実行例

```
>>> customer = {"name":"井上太郎", "pref":"東京都"} Enter
>>> age = customer.setdefault("age") Enter
>>> customer Enter
{'name': '井上太郎', 'pref': '東京都', 'age': None}
```

同様に、存在しないキー「"height"」にアクセスして、その値に「180.1」を設定する例を示します。

◎実行例

```
>>> height = customer.setdefault("height", 180.1) Enter
>>> customer Enter
{'name': '井上太郎', 'pref': '東京都', 'age': None, 'height': 180.1}
←heightが180.1に設定された
```

➡067

辞書に要素を追加する／値を変更するには

辞書 [キー] に値を代入する

辞書はミュータブルなデータ型です。指定したキーに対する値を代入することで値を変更できます。

◎実行例

```
>>> langs = {"Python":105, "JavaScript":20, "Swift":3} Enter
>>> langs["Swift"] = 9 Enter
>>> langs Enter
{'Python': 105, 'JavaScript': 20, 'Swift': 9}
```

存在しないキーを指定して辞書に値を代入すると、新たなキーと値のペアが作成されます。

◎実行例

```
>>> mydic = {} Enter　←空の辞書を作成
>>> mydic["name"] = "田中一郎" Enter　←キー"name"、値"田中一郎"を挿入
>>> mydic["age"] = 44 Enter　←キー"age"、値「44」を挿入
>>> mydic Enter
```

```
{'name': '田中一郎', 'age': 44}
```

➡068

辞書の要素数を取得するには

len関数を使用する

リストやタプルと同じく、辞書の要素数を求めるには**len**関数を使用します。

◎関数　**len(s)**

カテゴリ	**組み込み関数**
引数	**辞書**
戻り値	**辞書の要素数**
説明	**引数sで指定した辞書の要素数を戻す**
使用例	**l = len(mydic)**

◎実行例

```
>>> langs = {"Python":105, "JavaScript":20, "Swift":3, "Basic":5, "C":33} Enter
>>> len(langs) Enter
5
```

➡069

辞書の内容を別の辞書で更新／連結するには

updateメソッドを使用する

リストやタプルと異なり、辞書は「+」演算子を使用して連結できません。辞書に別の辞書を連結するには**update**メソッドを使用します

◎メソッド　update(other)

カテゴリ	**dictクラス**
引数	**辞書**
戻り値	**なし**
説明	**辞書の要素に引数otherで指定した辞書の要素を加える**
使用例	**d1.update(d2)**

◎実行例

```
>>> d1 = {"yellow":"黄", "red":"赤"} Enter
>>> d2 = {"blue":"青", "green":"緑"} Enter
>>> d1.update(d2) Enter
>>> d1 Enter
{'yellow': '黄', 'red': '赤', 'blue': '青', 'green': '緑'}
```

updateメソッド実行時に既存のキーがある場合には、値が上書きされます。

◎実行例

```
>>> n1 = {"name":"山田花子", "age":42} Enter
>>> n2 = {"number":15, "age":44} Enter
>>> n1.update(n2) Enter
>>> n1 Enter
{'name': '山田花子', 'age': 44, 'number': 15}
```

●「キー=値」の形式で要素を追加／更新する

updateメソッドの引数に「**キー=値**」の形式で引数を指定することによっても、要素を追加／更新できます。一度に複

数の要素を追加／更新することも可能です。

◎実行例

```
>>> d3 = {"man":"男性"} Enter
>>> d3.update(tree="木", sun="太陽") Enter　←要素を2つ追加
>>> d3 Enter
{'man': '男性', 'tree': '木', 'sun': '太陽'}
```

➡070

辞書の要素を削除するには

del文、popメソッド、 popitemメソッド、clearメソッドを使用する

辞書から、指定したキーの要素を削除するには**del文**を使用します。

del 辞書[キー]

◎実行例

```
>>> langs = {"Python":105, "JavaScript":20, "Swift":3, "Basic":5, "C":33} Enter
>>> del langs["Basic"] Enter
>>> langs Enter
{'Python': 105, 'JavaScript': 20, 'Swift': 3, 'C': 33}
```

●存在しないキーの要素を削除する場合にエラーにならないようにする

del文による削除では、指定したキーの要素が存在していないと**KeyError**例外が送出されます。

◎実行例

```
>>> del langs["C++"] Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'C++'
```

存在しないキーの要素を削除しようとした場合にエラーにならないようにするには、**pop**メソッドを使用します。

◎メソッド　pop(key[, default])

カテゴリ	**dictクラス**
引数	**key：キー、default：キーが存在しない場合に返す値**
戻り値	**削除した要素の値**
説明	**引数キーで指定した要素を削除しその値を戻す。引数defaultを設定しない場合、キーが存在しないとKeyError例外が送出される**
使用例	**val = colors.pop("blue", None)**

◎実行例

```
>>> colors = {"red":"赤", "blue":"青", "green":"緑"} Enter
>>> colors.pop("blue") Enter
'青'
>>> colors Enter
{'red': '赤', 'green': '緑'}
>>> colors.pop("black", None) Enter    ←defaultを指定すると存在しないキーを指定してもエラーにならない
>>> colors.pop("black") Enter   ←defaultを設定しないとエラー
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
KeyError: 'black
```

●任意の要素を削除する

　辞書からランダムに抽出した要素を削除するには**popitem**メソッドを使用します。

◎メソッド　popitem()

カテゴリ	**dictクラス**
引数	**なし**
戻り値	**キーと値のペアのタプル**
説明	**辞書から任意の要素を削除してそのキーと値のペアをタプルとして戻す**
使用例	**v = langs.popitem()**

　popitemメソッドは、辞書の要素が空になるまで、ランダムに要素を取得して削除していくのに便利です。
　次に、辞書langsの要素を1つずつ表示しながら削除していく例を示します。

◎リスト popitem1.py

```
langs = {"Python":105, "JavaScript":20, "Swift":3, "Basic":5, "C":33}

while langs:
    print(langs.popitem())
```

■実行結果

```
('C', 33)
('Basic', 5)
('Swift', 3)
('JavaScript', 20)
```

```
('Python', 105)
```

●すべての要素を削除する

辞書からすべての要素を削除するには**clear**メソッドを使用します。

◎メソッド　clear()

カテゴリ	**dictクラス**
引数・戻り値	**なし**
説明	**辞書からすべての要素を削除する**
使用例	**mydic.clear()**

clearメソッドを実行すると辞書は空の状態「{}」になります。

◎実行例

```
>>> langs.clear() Enter
>>> langs Enter
{}
```

➡071
辞書に指定したキーの要素が存在するか調べるには
in演算子、not in演算子を使用する

in演算子を使用すると、指定したキーの要素が辞書に存在しているかがわかります。キーが存在していればTrueを、そうでなければFalseを返します。

キー in 辞書

◎実行例

```
>>> colors = {"red":"赤", "blue":"青", "green":"緑", "white":"白"}
Enter
>>> "red" in colors Enter
True
>>> "gray" in colors Enter
False
```

逆に、**not in**演算子はキーが存在していればFalseを、存在していなければTrueを戻します。

キー not in 辞書

◎実行例

```
>>> "gray" not in colors Enter
True
```

NOTE Python 3では、Python 2に用意されていたin演算子と同じ動作をするhas_keyメソッドは廃止されました。

➡072
辞書のすべてのキーを取得するには

keysメソッドを使用する

keysメソッドを使用すると、辞書のすべてのキーを取得できます。

◎メソッド **keys()**

カテゴリ	**dictクラス**
引数	**なし**
戻り値	**すべてのキーをまとめたdict_keysオブジェクト**
説明	**辞書のすべてのキーを要素とするdict_keysオブジェクトを戻す**
使用例	**ks = langs.keys()**

keysメソッドの戻り値はリストの仲間である**dict_keys**オブジェクトです。

◎実行例

```
>>> colors = {"red":"赤", "blue":"青", "green":"緑", "white":"白"}
Enter
>>> colors.keys() Enter
dict_keys(['red', 'blue', 'green', 'white'])
```

dict_keysオブジェクトはイテレート可能です。keysメソッドの戻り値としてdict_keysオブジェクトをそのままfor文に渡せば値を1つずつ取り出せます。

◎実行例

```
>>> for k in colors.keys(): Enter
...         print(k) Enter
... Enter
red
blue
green
white
```

listコンストラクタに渡せばリストに変換できます。

◎実行例

```
>>> l = list(colors.keys()) Enter
>>> l Enter
['red', 'blue', 'green', 'white']
```

➡073

辞書のすべての値を取得するには

valuesメソッドを使用する

valuesメソッドを使用すると辞書のすべての値を取得できます。

◎メソッド　**values()**

カテゴリ	**dictクラス**
引数	**なし**
戻り値	**すべての値をまとめたdict_valuesオブジェクト**
説明	**辞書のすべての値を要素とするdict_valuesオブジェクトを戻す**
使用例	**vs = langs.values()**

valuesメソッドの戻り値はイテレート可能な**dict_values**オブジェクトです。

◎実行例

```
>>> colors = {"red":"赤", "blue":"青", "green":"緑", "white":"白"}
Enter
>>> colors.values() Enter
dict_values(['赤', '青', '緑', '白'])
```

valuesメソッドの戻り値としてdict_valuesオブジェクトをそのままfor文に渡せば値を1つずつ取り出せます。

◎実行例

```
>>> for v in colors.values(): Enter
...     print(v) Enter
... Enter
赤
青
緑
白
```

●辞書に値が存在するかを調べる

辞書に指定したキーが存在しているかを調べるにはin演算子（071で解説）を使用しました。それに対して、辞書に指定した値が存在しているかを調べるには、valuesメソッドの戻り値に対してin演算子を使用します。

◎実行例

```
>>> "緑" in colors.values() Enter
True
```

➡074
辞書からすべてのキーと値のペアを取得するには

 itemsメソッドを使用する

itemsメソッドを使用すると辞書からキーと値のペアをすべて取得できます。

◎メソッド　items()

カテゴリ	**dictクラス**
引数	**なし**
戻り値	**辞書のすべてのキーと値のペアをまとめたdict_itemsオブジェクト**
説明	**辞書のすべてのキーと値のペアを要素とするdict_itemsオブジェクトを戻す**
使用例	**its = langs.items()**

itemsメソッドの戻り値はキー、値のペアをタプルとする**dict_items**オブジェクトです。

◎実行例

```
>>> colors = {"red":"赤", "blue":"青", "green":"緑", "white":"白"} Enter
>>> colors.items() Enter
dict_items([('red', '赤'), ('blue', '青'), ('green', '緑'), ('white', '白')])
```

listコンストラクタに渡すと、キーと値のペアのタプルを要素とするリストに変換できます。

◎実行例

```
>>> list(colors.items()) Enter
[('red', '赤'), ('blue', '青'), ('green', '緑'), ('white', '白')]
```

dict_itemsオブジェクトはイテレート可能なので、次のようにfor文を使用することでキー、値の一覧を表示できます。

◎実行例

```
>>> for k, v in colors.items(): Enter
...         print(k, v) Enter
... Enter
red 赤
blue 青
green 緑
white 白
```

●値の最大値を求める

keys、values、itemsメソッドの戻り値はイテレート可能なオブジェクトなのでmax関数やmin関数（055で解説）に渡すことができます。

次のリストでは、プログラム言語をキーに、その得票数を値とする辞書langsがあるとしています。この辞書から、最も得票数の多い言語を見つけるには、max関数にkey引数を設定して次のようにします。

◎リスト items1.py（値の最大値を求める）

```
langs = {"Python":105, "JavaScript":20, "Swift":3, "Basic":5, "C":33}
maxlang = max(langs.items(), key=lambda l:l[1]) 1
print(maxlang)
```

1でmax関数を実行しています。itemsメソッドによりキーと値のペアを取り出し、key引数ではlambda式で辞書の値の最大値を求めるための基準の要素を設定しています。

■実行結果

```
('Python', 105)
```

●値の大きい順に表示する

同様にkeys、values、itemsメソッドの戻り値をsorted関数に渡してソートすることもできます。次に、辞書langsを値の大きい順に表示する例を示します。

◎リスト items2.py（値の大きい順に表示する）

```
langs = {"Python":105, "JavaScript":20, "Swift":3, "Basic":5, "C":33}
sorted_langs = sorted(langs.items(), key=lambda l:l[1], reverse=True) 1
for l, v in sorted_langs:
    print(l, v)
```

1でsorted関数を実行しています。item1.pyと同様にkey引数により辞書の値を基準の要素となるように設定しています。

■実行結果

```
Python 105
C 33
JavaScript 20
Basic 5
Swift 3
```

辞書をコピーするには

copyメソッドを使用する

辞書をコピーするには**copy**メソッドを使用します。

◎メソッド　copy()

カテゴリ	**dictクラス**
引数	**なし**
戻り値	**コピーされた辞書**
説明	**辞書をコピーして戻す**
使用例	**dict2 = dict1.copy()**

◎実行例

```
>>> colors = {"red":"赤", "blue":"青", "green":"緑", "white":"白"}
>>> colors Enter
{'red': '赤', 'blue': '青', 'green': '緑', 'white': '白'}
>>> new_colors = colors.copy() Enter　←辞書colorsをコピー
>>> colors["gray"] = "灰色" Enter　←辞書colorsに要素を追加
>>> colors Enter
{'red': '赤', 'blue': '青', 'green': '緑', 'white': '白', 'gray': '灰色'}
>>> new_colors Enter　↓new_colorsは変わらない
{'red': '赤', 'blue': '青', 'green': '緑', 'white': '白'}
```

●浅いコピー

copyメソッドはいわゆる浅いコピーを行います。内部のオブジェクトはコピーされません。次の辞書d1は、colorsキーに対してリストを値に設定しています。

◎実行例

```
>>> d1 = {"name":"mac", "colors":["gray", "black"]} Enter
```

辞書d1をd2にコピーした場合、d1のcolorsキー要素の変更はd2にも反映されます。

◎実行例

```
>>> d2 = d1.copy() Enter  ←d1をd2にコピー
>>>  d1["colors"].append("red")  Enter      ←d1 の colors の 値に"red"を追加
>>> d1["colors"] Enter
['gray', 'black', 'red']
>>> d2["colors"] Enter
['gray', 'black', 'red'] ←d2のcolorsキーの値も変更される
```

➡076

シンプルな記述が可能な内包表記で辞書を生成するには

「{キー:値 for 変数 in イテレート可能なオブジェクト}」を使用する

リストの内包表記（062で解説）と同じく、**辞書の内包表記**を使用するとシンプルな記述で、多少複雑な辞書が生成できます。辞書の内包表記の書式は次のようになります。

{キー:値 for 変数(キー), 変数(値) in イテレート可能なオブジェクト}

次にリストcolorsの要素をキーに、リストnumbersの要素を値とする辞書colors_dicを生成する例を示します。

◎リスト dict_comp1.py

```
colors = ["blue", "green", "red", "yellow"]
numbers = [9, 3, 4, 2]

colors_dic = {c:v for c,v in zip(colors, numbers)} 1
print(colors_dic)
```

1が内包表記による辞書の生成部分です。zip関数（059で解説）を使用してリストを組み合わせている点に注目してください。

■実行結果

```
{'blue': 9, 'green': 3, 'red': 4, 'yellow': 2}
```

NOTE 実際には「dict_comp1.py」の例は、内包表記を使用しないで、zip関数の結果をdictコンストラクタに渡しても同じです。

```
colors_dic= dict(zip(colors, numbers))
```

内包表記が活躍するのは何らかの処理が必要なケースです。たとえば、dict_comp1.pyを、辞書の値にリストnumbersの要素を2倍にして代入するように変更したい場合には次のようにします。

◎リスト dict_comp2.py（一部）

```
colors_dic = {c:v*2 for c,v in zip(colors, numbers)}
```

■実行結果

```
{'blue': 18, 'green': 6, 'red': 8, 'yellow': 4}
```

●条件式も使える

　リストの内包表記ではif文による条件式も使用できます。次に、dic_comp1.pyを変更して、リストnumbersの要素が偶数の場合にのみ、辞書の要素とする例を示します。

◎リスト dict_comp3.py（一部）

```
colors_dic = {c:v for c,v in zip(colors, numbers) if v%2==0}
```

■実行結果

```
{'red': 4, 'yellow': 2}
```

●辞書のキーと値を入れ替える

　次に、辞書の内包表記を使用して、辞書のキーと値を入れ替えた新たな辞書を生成する例を示します。

◎リスト dic_comp4.py

```
seasons = {" 春 ":"Spring", " 夏 ":"Summer", " 秋 ":"Autumn", "冬":"Winter"}

new_seasons = {v:k for k, v in seasons.items()} 1
print(new_seasons)
```

　1で変数kに入れたキーを値に、変数vに入れた値をキーにすることでキーと値を入れ替えています。

■実行結果

```
{'Spring': '春', 'Summer': '夏', 'Autumn': '秋', 'Winter': '冬'}
```

2-3 セットの操作

セット（set）は要素の重複を許さないイテレート可能なデータ型です。また、ミュータブルなデータ型です。あとから要素を追加/変更可能です。この節ではセットの取り扱いについて説明します。

➡077

セットをリテラルとして生成するには

「{値1, 値2, 値3, ...}」を使用する

セット（set）オブジェクトをリテラルで記述するには次の書式を使用します。

{値1, 値2, 値3, ...}

◎実行例

```
>>> s1 = {"dog", "cat", "pig", "cow"} Enter
>>> s1 Enter
{'cat', 'cow', 'pig', 'dog'}
```

●重複が取り除かれる

なお、リテラルで記述した要素に重複がある場合、自動的に重複が取り除かれます。

◎実行例

```
>>> s2 = {"red", "green", "blue", "red", "green"} Enter
>>> s2 Enter
```

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

ＮＯＴＥ 空のセットを作成するにはsetコンストラクタを引数なしで実行します。

```
>>> es = set() Enter     ←空のセット
```

単に「{}」を記述すると空の辞書が作成されてしまうので注意してください。

```
>>> es = {} Enter    ←空の辞書
```

➡078

セットをリストやタプルなどから生成するには

setコンストラクタを使用する

　リストやタプルなどイテレート可能なオブジェクトをsetコンストラクタの引数にすることで、セットが生成できます。この場合も要素の重複が削除されます。

◎実行例

```
>>> t1 = ("アメリカ", "日本", "中国", "イギリス", "日本", "中国") Enter
>>> s1 = set(t1) Enter
>>> s1 Enter
{'日本', 'アメリカ', 'イギリス', '中国'}
```

　辞書を引数にした場合には、キーからセットが生成されます。

◎**実行例**

```
>>> d1 = {"春":"Spring", "夏":"Summer", "秋":"Autumn", "冬":
"Winter"} Enter
>>> s2 = set(d1) Enter
>>> s2 Enter
{'夏', '秋', '冬', '春'}
```

　辞書の要素の値からセットを生成するには辞書の**values**メソッドを使用します。

◎**実行例**

```
>>> s3 = set(d1.values()) Enter
>>> s3 Enter
{'Autumn', 'Summer', 'Winter', 'Spring'}
```

　rangeオブジェクトからセットを生成することもできます。

◎**実行例**

```
>>> s4 = set(range(0,20,4)) Enter
>>> s4 Enter
{0, 4, 8, 12, 16}
```

➡079

セットの要素数を求めるには

len関数を使用する

セットの要素数は**len**関数で調べられます。

◎関数　**len(s)**

カテゴリ	**組み込み関数**
引数	**セット**
戻り値	**要素数**
説明	**セットの要素数を戻す**
使用例	**sl = len(s1)**

◎実行例

```
>>> s1 = {"red", "green", "blue", "white", "yellow"} Enter
>>> len(s1) Enter
5
```

➡080

セットに要素を1つ追加するには

addメソッドを使用する

既存のセットに要素を1つ追加するには**add**メソッドを使用します。

◎メソッド　**add(el)**

カテゴリ	**setクラス**
引数	**追加する値**
戻り値	**なし**
説明	**セットに引数elで指定した値の要素を追加する**
使用例	**s1.add("東京都")**

◎実行例

```
>>> s1 = {"red", "green", "blue"} Enter
```

```
>>> s1.add("black") Enter
>>> s1 Enter
{'green', 'black', 'blue', 'red'}
```

ＮＯＴＥ すでに存在する要素を追加しようとしても何も起こりません（エラーにはなりません）。

```
>>> s1.add("red")  Enter
```

➡081

セットを別のコレクションで更新するには

updateメソッドとその仲間のメソッドを使う

updateメソッドを使用すると、セットに、引数で指定したリストやタプル、セットの要素をまとめて追加できます。

◎メソッド　update(others)

カテゴリ	**setクラス**
引数	**イテレート可能なオブジェクト**
戻り値	**なし**
説明	**セットに引数othersで指定したオブジェクトの要素を加える**
使用例	**s.update(["アメリカ", "イギリス"])**

◎実行例

```
>>> s1 = {"red", "green", "blue"} Enter
>>> s1.update(["black", "yellow"]) Enter
>>> s1 Enter
{'blue', 'black', 'red', 'yellow', 'green'}
```

NOTE 文字列を引数にしてupdateメソッドを実行すると、文字列が文字に分解されて追加されてしまうので注意してください。

◎実行例

```
>>> s1 = {"red", "green", "blue"} Enter
>>> s1.update("brown") Enter
>>> s1 Enter
{'green', 'r', 'n', 'w', 'blue', 'red', 'b', 'o'}
```

●共通の要素を残すには

intersection_updateメソッドを使用すると現在のセットの要素と、引数で指定したリストやタプル、セットの要素と共通する要素を残して、それ以外は削除します。

◎メソッド **intersection_update(others)**

カテゴリ	**setクラス**
引数	**イテレート可能なオブジェクト**
戻り値	**なし**
説明	**セットの要素を、引数othersで指定したオブジェクトの要素と共通の要素にする**
使用例	**s.intersection_update(["アメリカ", "イギリス"])**

◎実行例

```
>>> s2 = {1, 2, 3, 4, 5, 6} Enter
>>> s2.intersection_update({3, 4, 9}) Enter
>>> s2 Enter
{3, 4}
```

●引数で指定したオブジェクトの要素を取り除く

difference_updateメソッドを使用すると、現在のセットの要素から、引数で指定したリストやタプル、セットなどと共通な要素を削除します。

◎メソッド　difference_update(others)

カテゴリ	**setクラス**
引数	**イテレート可能なオブジェクト**
戻り値	**なし**
説明	**セットの要素から、引数othersで指定したオブジェクトとの共通の要素を取り除く**
使用例	**s.difference_update({"red", "blue"})**

◎実行例

```
>>> s2 = {1, 2, 3, 4, 5, 6} Enter
>>> s2.difference_update((4, 5, 200)) Enter
>>> s2 Enter
{1, 2, 3, 6}
```

rangeオブジェクトを引数にすることもできます。次の例は偶数を削除します。

◎実行例

```
>>> s2 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} Enter
>>> s2.difference_update(range(0, 11, 2)) Enter
{1, 3, 5, 7, 9}
```

● 現在のセットの要素と別のオブジェクトの要素のうちどちらかに含まれる要素を残す

symmetric_difference_updateメソッドを使用すると、現在のセットと引数で指定したオブジェクトの共通の要素を削除し、どちらかのみに含まれている要素を残します。

◎メソッド **symmetric_difference_update(others)**

カテゴリ	setクラス
引数	イテレート可能なオブジェクト
戻り値	なし
説明	セットの要素と、引数othersで指定したオブジェクトのどちらかに含まれている要素を残してほかを削除する
使用例	s.symmetric_difference_update(["ロシア"])

◎実行例

```
>>> s1 = {"blue", "black", "red", "green"} Enter
>>>  s1.symmetric_difference_update({"blue",  "white",  "green"})
Enter
>>> s1 Enter
{'white', 'red', 'black'}
```

ＮＯＴＥ update、intersection_update、difference_updateメソッドは複数の引数を取ることができます。

```
>>> s1 = {1, 2, 3, 4, 5, 6} Enter
>>> s1.difference_update({3,4}, {5}) Enter
```

●演算子によるセットの更新

updateメソッド、intersection_updateメソッド、difference_updateメソッド、symmetric_difference_updateメソッドの代わりに次のような演算子を使用してセットを更新することもできます。

⊙セットを更新するための演算子

演算子	説明
s1 |= s2	s1にs2の要素を加える
s1 &= s2	s1の要素をs2と共通の要素にする
s1 -= s2	s1の要素からs2の要素を削除する
s1 ^= s2	s1の要素を、s1とs2のどちらかに含まれている要素にする

◎実行例

```
>>> s1 = {1, 4, 5} Enter
>>> s1 |= {4, 3,20} Enter
>>> s1 Enter
{1, 3, 4, 5, 20}
>>> s1 &= {4, 5, 20, 33} Enter
>>> s1 Enter
{5, 20, 4}
>>> s1 ^= {5, 6, 9, 10} Enter
>>> s1 Enter
{4, 6, 9, 10, 20}
```

NOTE 「|=」「&=」「-=」は、それぞれ「|」「&」「-」を続けることにより複数の値でセットを更新することができます。

```
>>> s1 = {1, 2, 3} Enter
>>> s1 |= {4, 8} | {9, 10} Enter
```

➡082

2つのセットを合わせて新たなセットを生成するには

union メソッド、intersection メソッド、difference メソッドなどを使用する

update 、 intersection_update 、 difference_update 、 symmetric_difference_updateと似たメソッドに、**union**、**intersection**、**difference**、**symmetric_difference**があります。前者と異なり、これらのメソッドは新たにセットを生成して戻します。また、引数はセットである必要があります。

⦿新たにセットを生成するメソッド

メソッド	説明
union(*others)	セットと引数で指定したセットの要素からなる新たなセットを生成して戻す
intersection(*others)	セットと引数で指定したセットの共通の要素からなる新たなセットを生成して戻す
difference(*others)	セットに含まれているが、引数で指定したセットに含まれない要素を持つ新たなセットを生成して戻す
symmetric_difference(other)	セットと引数で指定したセットのどちらかに含まれるセットを生成して戻す

次にこれらのメソッドの使用例を示します。

◎リスト makeset1.py

```
s1 = {1, 4, 9, 5, 3}

print(s1.union({4, 5, 100}))
print(s1.intersection({4, 5, 100}))
print(s1.difference({4, 5, 100}))
print(s1.symmetric_difference({4, 5, 100}))
```

■実行結果

```
{1, 3, 4, 5, 100, 9}
{4, 5}
{1, 3, 9}
```

```
{1, 3, 100, 9}
```

●演算子を使用する

unionメソッド、intersectionメソッド、differenceメソッド、symmetric_differenceメソッドの代わりに、次のような演算子を使用することもできます。

⦿新たなセットを生成する演算子

演算子	説明
s1 | s2	s1とs2の要素からなる新たなセットを生成して戻す
s1 & s2	s1とs2のどちらにも含まれる要素からなる新たなセットを生成して戻す
s1 - s2	s1に含まれs2に含まれない要素からなる新たなセットを生成して戻す
s1 ^ s2	s1とs2のどちらかだけに含まれる要素からなる新たなセットを生成して戻る

◎実行例

```
>>> {1, 4, 5} | { 4, 3, 20} Enter
{1, 3, 4, 5, 20}
>>> {1, 4, 5} & { 4, 3, 20} Enter
{4}
>>> {1, 4, 5} - { 4, 3, 20} Enter
{1, 5}
>>> {1, 4, 5} ^ { 4, 3, 20} Enter
{1, 3, 5, 20}
```

3つ以上のセットにも使用可能です。

◎実行例

```
>>> {1, 4, 5} - { 4, 3, 20} - {5} Enter
{1}
```

➡083

セットの要素を削除するには

removeメソッド、discardメソッド、popメソッド、clearメソッド、del文を使用する

セットから、指定した要素を削除するには、**remove**メソッドを使用します。

◎メソッド　**remove(elem)**

カテゴリ	**setクラス**
引数	**削除する要素**
戻り値	**なし**
説明	**セットから引数elemで指定した要素を削除する**
使用例	**s.remove("アメリカ")**

◎実行例

```
>>> s1 = {"blue", "black", "red", "green"} Enter
>>> s1.remove("blue") Enter
>>> s1 Enter
{'black', 'green', 'red'}
```

●削除時に要素がない場合にエラーにならないようにする

removeメソッドは引数で指定した要素が存在しない場合には、**KeyError**例外が送出されます。

◎実行例

```
>>> s1.remove("gold") Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'gold'
```

削除時に要素が存在しない場合に、エラーにならないようにしたいときは、**discard**メソッドを使用します。

◎メソッド　discard(elem)

カテゴリ	**setクラス**
引数	**削除する要素**
戻り値	**なし**
説明	**セットから引数elemで指定した要素を削除する**
使用例	**s.discard("アメリカ")**

◎実行例

```
>>> s1.discard("gold") Enter
```

●任意の要素を削除する

popメソッドで、セットからランダムに選んだ要素を削除し、その値を戻します。

◎メソッド　pop()

カテゴリ	**setクラス**
引数	**なし**
戻り値	**削除した要素**
説明	**セットから任意の要素を削除し、その値を戻す。セットが空の場合にはKeyError例外が送出される**
使用例	**n = age.pop()**

次に、セットmysetから、要素を順に取り除いて表示する例を示します。

◎リスト pop1.py

```
myset = {1965, 1959, 2001, 1987, 1959, 2011}
```

```
while True: 1
    try:
        print(myset.pop()) 2
    except:
        break 3
```

1のwhileループで処理を繰り返し、2でpopメソッドにより任意の要素を削除しその値を表示しています。セットの要素が空になると例外が発生し、3のbreak文でループを抜けます。

■実行結果

```
1987
1959
1965
2001
2011
```

●すべての要素を削除する

すべての要素を削除するには**clear**メソッドを使用します。

◎メソッド　clear()

カテゴリ	**setクラス**
引数・戻り値	**なし**
説明	**セットを空にする**
使用例	**s.clear()**

◎実行例

```
>>> s1 = {"blue", "black", "red", "green"} Enter
>>> s1.clear() Enter
>>> s1 Enter
```

set()

●セットそのものを削除する

セット自体を削除するには**del文**を使用します。

del セット

◎実行例

```
>>> s1 = {"blue", "black", "red", "green"} Enter
>>> del s1 Enter
```

➡084

セットをリストに変換するには

listコンストラクタを使用する

セットを**list**コンストラクタに渡すとリストに変換できます。

◎実行例

```
>>> s1 = {"blue", "black", "red", "green"} Enter
>>> l1 = list(s1) Enter
>>> l1 Enter
['red', 'blue', 'green', 'black']
```

同様にtupleコンストラクタにセットを渡せばタプルに変換できます。

➡085
セットに指定した要素が含まれているかを調べるには

in、not in演算子を使用する

ある値がセットの要素に含まれているかを調べるには**in**演算子を使用します。

値 in セット

値が、含まれていればTrueを、いなければFalseを戻します。

◎実行例

```
>>> s1 = {"blue", "black", "red", "green"} Enter
>>> "black" in s1 Enter
True
>>> "pink" in s1 Enter
False
```

逆に含まれていないかを調べるには**not in**演算子を使用します。値が含まれていなければTrueを、含まれていればFalseを戻します。

値 not in セット

◎実行例

```
>>> "black" not in s1 Enter
False
```

➡086

セットの要素を順に取得するには

forループを使用する

セットはイテレート可能なオブジェクトなので**for**ループで順に値を取得し処理できます。次に、セットyearsから順に要素を取り出し表示する例を示します。

リスト setfor1.py

```
years = {1965, 1959, 2001, 1987, 1959, 2011} 1
for y in years:
    print(str(y) + "年") 2
```

1のfor文で年が格納されたセットyearsから年を1つずつ取り出し、**2**で"年"と連結して表示しています。

■実行結果

```
1987年
1959年
1965年
2001年
2011年
```

➡087

2つのセットの関係を調べるには

isdisjointメソッド、issubsetメソッド、issupersetメソッドを使用する

セットと引数で指定したセットの関係を調べ、Trueもしくはfalseを返すメソッドを示します。

⊙セットの関係を調べるメソッド

メソッド	説明
isdisjoint(other)	セットが引数otherと共通の要素を持たなければTrue、そうでなければFalseを返す
issubset(other)	セットのすべての要素が引数で指定したセットotherに含まれればTrue、そうでなければFalseを返す
issuperset(other)	引数で指定したセットotherのすべての要素が、セットに含まれればTrue、そうでなければFalseを返す

次に実行例を示します。

◎リスト checkset1.py

```
s1 = {1, 4, 9, 5, 3}

print(s1.isdisjoint({-1, 2}))
print(s1.isdisjoint({1,2}))

print(s1.issubset({9, 1, 4, 3, 5, 10}))
print(s1.issubset({9, 1, 4, 10}))

print(s1.issuperset({1, 4}))
print(s1.issuperset({3, 101}))
```

■実行結果

```
True
False
```

```
True
False
True
False
```

●演算子を使用する

次のような演算子を使用しても2つのセットの関係を調べられます。

⊙セットの関係を調べる演算子

演算子	説明
s1 <= s2	s1の要素がs2の要素にすべて含まれればTrue、そうでなければFalseを返す
s1 < s2	s1の要素がs2の要素にすべて含まれればTrue、そうでなければFalse。ただしs1とs2の要素が同じであればFalseを返す
s1 >= s2	s2の要素がs1の要素にすべて含まれればTrue、そうでなければFalseを返す
s1 > s2	s2の要素がs1の要素にすべて含まれればTrue、そうでなければFalse。ただしs1とs2の要素が同じであればFalseを返す

◎実行例

```
>>> {1, 2} <= {1, 2} Enter
True
>>> {1, 2} <= {1, 2, 3} Enter
True
>>> {4, 5, 6} > {4, 5, 6} Enter
False
>>> {4, 5, 6} > {4, 5} Enter
True
```

➡088
セットの要素を辞書のキーとして使用するには

frozensetオブジェクトを使用する

セット（set）はミュータブルなオブジェクトなのでそのままでは辞書のキーとして使用できません。次のように、セットをそのまま辞書のキーにしようとするとエラーになります。

◎実行例

```
>>> s1 = {"green", "blue"} Enter
>>> s2 = {"black", "white"} Enter
>>> d1 = {s1:3 , s2:4} Enter　←セットをキーにしようとするとエラー
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

セットを辞書のキーにするには、イミュータブルなセットである**frozenset**クラスのインスタンスに変換する必要があります。

◎コンストラクタ　frozenset(s)

カテゴリ	**frozensetクラス**
引数	**イテレート可能なオブジェクト**
戻り値	**frozensetオブジェクト**
説明	**セットから、イミュータブルなセットであるfrozensetオブジェクトを生成して戻す**

◎実行例

```
>>> s1 = {"green", "blue"} Enter
>>> s2 = {"black", "white"} Enter
>>> d1 = {frozenset(s1):3 , frozenset(s2):4} Enter
>>> d1 Enter
```

```
{frozenset({'green', 'blue'}): 3, frozenset({'black', 'white'}): 4}
```

➡089

セットをコピーするには

copyメソッドを使用する

セットをコピーするには**copy**メソッドを使用します。

◎メソッド　**copy()**

カテゴリ	**setクラス**
引数	**なし**
戻り値	**コピーされたセット**
説明	**セットをコピーして新たなセットを生成して戻す**
使用例	**s2 = s1.copy()**

◎実行例

```
>>> s1 = {"東京都", "神奈川県", "千葉県"} Enter
>>> s2 = s1.copy() Enter　←s1をs2にコピー
>>> s1.add("大阪府") Enter　←s1に要素を加える
>>> s2 Enter
{'東京都', '神奈川県', '千葉県'} ←s2は変更されない
```

➡090

シンプルな記述が可能な内包表記でセットを生成するには

{式 for 変数 in イテレート可能なオブジェクト if 条件式} を使用する

リスト内包表記（062で解説）や辞書の内包表記（076で解説）と同じく、**セットの内包表記**を使用するとシンプルな記述で、多少複雑なセットが生成できます。
以下に、セットの内包表記の書式を示します。

{式 for 変数 in イテレート可能なオブジェクト if 条件式}

たとえば、次のような("名前", "性別")の形式のタプルを要素とするリストnamesがあるとします。

リストnames

```
names =  [("田中一郎", "男"), ("山田太郎", "男"), ("佐藤花子", "女"), ....]
```

リストnamesから性別が「"男"」の要素を取り出して、名前を要素とするセットmensetを作成するには次のようにします。

◎リスト setcmp1.py

```
names = [("田中一郎", "男"), ("山田太郎", "男"), ("佐藤花子", "女"),
("猫山五朗", "男"), ("小林直子", "女"), ("大木虎夫", "男")]

menset = {n[0] for n in names if n[1] == "男"} 1
print(menset)
```

1で内包表記を使用しています。「if n[1] == "男"」で男性のみを取り出し、「n[0]」で名前をセットの要素としています。

■実行結果

```
{'田中一郎', '大木虎夫', '山田太郎', '猫山五朗'}
```

2-4 collectionsモジュールの便利なクラス

Pythonのコレクション型というと、タプル、リスト、辞書、集合といったところが有名ですが、collectionsモジュールにも便利なクラスがいくつか用意されています。ここではそれらのクラスを紹介します。

➡091 タプルやリストの要素の出現回数を数えるには

collectionsモジュールのCounterクラスを使用する

タプルやリストの要素の出現回数を数えるときには辞書型のオブジェクトが適しています。たとえば、タプルfruitsに格納された単語の出現頻度を調べて結果を辞書countに格納するコード例は以下のようになるでしょう。

◎実行例

```
>>> fruits = ("apple", "melon", "orange", "apple", "orange", "apple") Enter
>>> count = {} Enter
>>> for f in fruits: Enter
...     if f in count: Enter
...         count[f] += 1 Enter
...     else: Enter
...         count[f] = 1 Enter
... Enter
>>> count Enter
{'apple': 3, 'melon': 1, 'orange': 2}
```

これでも問題なく単語の出現回数をカウントできますが、**Counter**クラスを使うとよりシンプルに実装することができます。

◎コンストラクタ　Counter()

カテゴリ	**collectionsモジュール**
引数	**なし（キーワード引数や辞書型オブジェクトで初期化することも可能）**
戻り値	**Counterオブジェクト**
説明	**出現頻度を数えるカウンターオブジェクトを作成する**
使用例	**c = collections.Counter(["A","B","A","C","B","A"])**

以下は、Counterオブジェクトを作成し、"hello world"の文字の出現回数を数える例です。

◎実行例

```
>>> import collections Enter
>>> c = collections.Counter() Enter
>>> for v in "hello world": Enter
...     c[v] += 1 Enter
... Enter
>>> c Enter
Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})   1
```

1の結果を見ると、lが3回、oが2回、h～dが1回ずつというように、出現回数順に並んでいます。

Counterオブジェクトは、辞書と同様に変数名[キー]でアクセスすることで値を取得できます。通常の辞書のようにキーの存在を事前に調べる必要はありません。また、コンストラクタの引数として初期値を与えることも可能です。

以下は、前出の例をCounterクラスで処理した例です。

◎実行例

```
>>> import collections Enter
>>> collections.Counter(("apple", "melon", "orange", "apple",
"orange", "apple")) Enter
Counter({'apple': 3, 'orange': 2, 'melon': 1})
```

先の例に比べて記述が格段にシンプルになり、出力結果も出現回数順に並んでいます。

●出現回数の多い順にソートする

出現回数を処理する場合、出現回数が最も多い項目を調べるケースも多いでしょう。そのような用途のために**most_common**メソッドが用意されています。

◎メソッド　most_common([n])

カテゴリ	**collectionsモジュール**
引数	**取得する要素の数**
戻り値	**出現回数の多い順からn個分のキーと値のリスト**
説明	**引数を省略した場合はすべての要素を多い順に返す**
使用例	**counter_obj.most_common(2)**

◎実行例

```
>>> import collections Enter
>>> count = collections.Counter(("apple", "melon", "orange",
"apple", "orange", "apple")) Enter
>>> print(count.most_common(2))      Enter  1
[('apple', 3), ('orange', 2)]
>>> print(count.most_common(1))      Enter  2
[('apple', 3)]
```

Counterのコンストラクタにタプルを渡しています。**1**のmost_common(2)では、出現回数が多い順に2個表示してい

ます。**2**のmost_common(1)は、最も出現回数が多いものを表示しています。

➡092
辞書にデフォルト値を設定するには

collectionsクラスのdefaultdictクラスを使用する

　辞書へのアクセスにはキーの存在が必要です。存在しないキーにアクセスするとエラーが返されます。

◎実行例

```
>>> d = {} Enter
>>> d["apple"] Enter
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'apple'
```

　キーが存在するか否かチェックするにはin演算子を使用します。

◎実行例

```
>>> d = {"one":"apple", "two":"melon"} Enter
>>> "one" in d Enter
True
>>> "three" in d Enter
False
```

　辞書にキーが存在することを確認してなんらかの処理を行うには、次のようにif文を使用する必要があります。

if キー in 辞書:
　　処理

　存在しないキーにアクセスする可能性がある場合に、その都度in演算子でチェックするのは面倒です。ここでは、存在しないキーにアクセスした場合に、引数で指定した関数を実行してその結果を要素とすることが可能な**defaultdict**クラスを紹介しましょう。

◎コンストラクタ　defaultdict(default_factory)

カテゴリ	**collectionsモジュール**
引数	**デフォルト値を生成する関数やコンストラクタなど**
戻り値	**default値をもった辞書（defaultdictオブジェクト）**
説明	**デフォルト値をもった辞書を作成する**
使用例	**d = collections.defaultdict(lambda: 0)**

　defaultdictオブジェクトではキーが存在しなくてもエラーになりません。コンストラクタの引数で指定された関数やコンストラクタが呼び出され、それらが返す値がデフォルト値として使用されます。
　実際の使用例を見てみましょう。次に、年齢と名前のタプルを要素するタプルpeopleから、年齢をキーに名前のリストを要素とするdefaultdictオブジェクトを生成する例を示します。

```
people = ((20,"Taro"),(23,"Ken"),(21,"Rin"),(20,"Joe"),(22,"Risa"))
```

↓　**default_dict_sample.py**

```
{20: ['Taro', 'Joe'], 23: ['Ken'], 21: ['Rin'], 22: ['Risa']})
```

◎リスト default_dict_sample.py

```
import collections 1
people = ((20,"Taro"),(23,"Ken"),(21,"Rin"),(20,"Joe"),(22,"Risa"))
d = collections.defaultdict(list) 2
for age, name in people: 3
    d[age].append(name)
print(d)
```

1でcollectionsモジュールを読み込みます。peopleは年齢と名前からなるタプルのタプルです。

2でdefaultdictオブジェクトを作成しています。デフォルトにlistのコンストラクタを指定しているので、デフォルト値は空のリストとなります。

3のfor文でpeopleから順に要素を取り出し、年齢ごとのリストを作成し、そこに名前をappendメソッドで追加しています。

■実行結果

```
defaultdict(<class 'list'>, {20: ['Taro', 'Joe'], 23: ['Ken'], 21: ['Rin'], 22: ['Risa']})
```

NOTE 辞書のsetdefaultメソッド（066で解説）やgetメソッド(066で解説)を使用しても同じようなことが行えます。例えば2 3はsetdefaultメソッドを使用すると次のようになります。

```
d = {}
for age, name in people:
    d.setdefault(age, []).append(name)
```

両者を比べてみるとわかるように、defaultdicのほうが、通常の辞書と同じく「辞書[キー]」の形式で要素にアクセスできるのでスマートです。

➡093

タプルの要素に名前を付けるには

collectionsモジュールのnamedtupleクラスを使用する

namedtupleクラスを使用するとタプルを構成する要素に名前を付けることができます。既存のコードに影響せず使えるので、単なるタプルから名前付きタプルに移行することができます。

◎ コンストラクタ　namedtuple(typename, field_names)

カテゴリ	**collectionsモジュール**
引数	**typename：型名（tupleのサブクラスの型名） field_names：フィールド名（['x', 'y']のような文字列のシーケンス）**
戻り値	**typenameで指定した型のコンストラクタ**
説明	**各フィールドに名前でアクセス可能となるtupleのサブクラスを作成する**
使用例	**Person = collections.namedtuple("Person", "name age")**

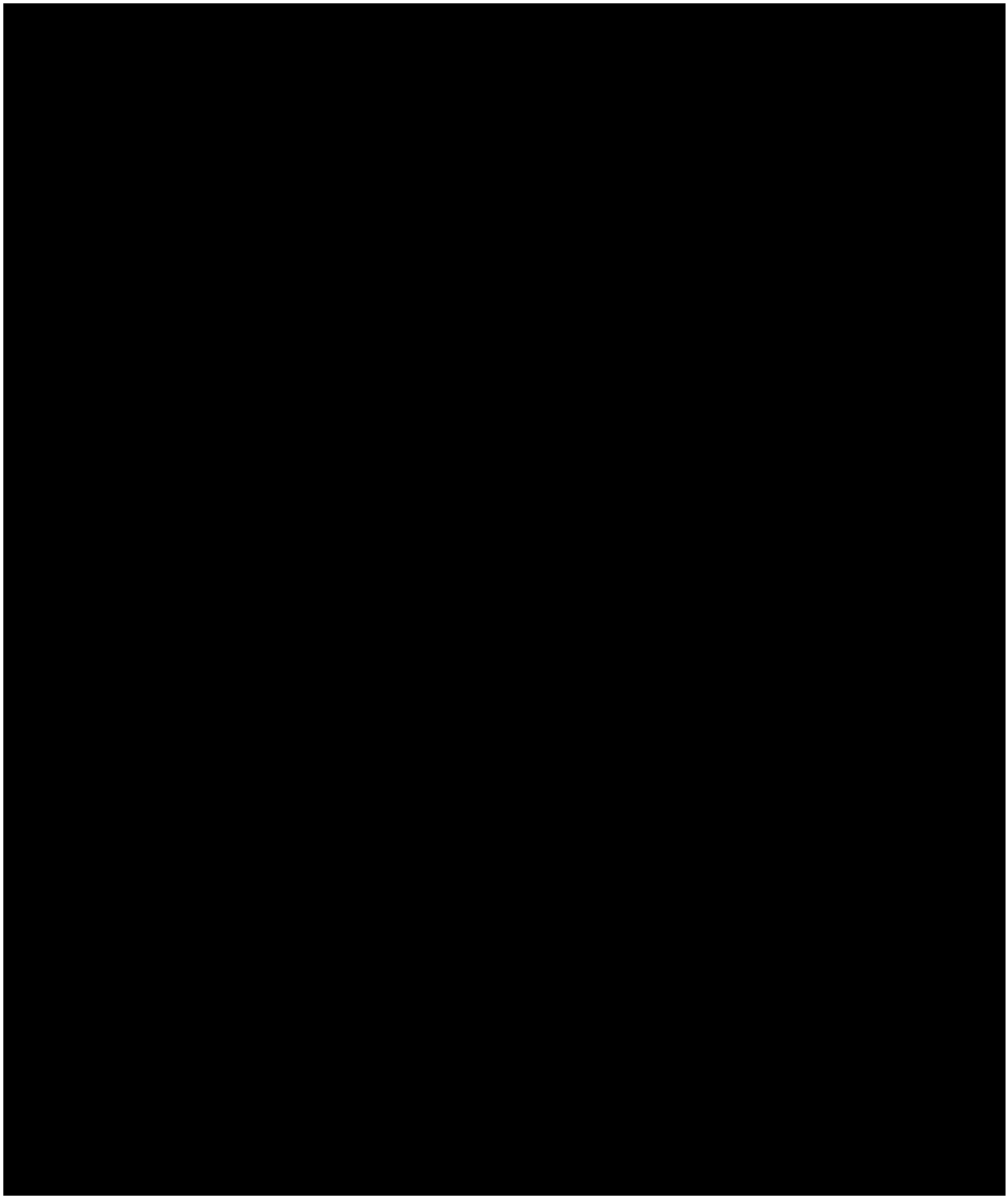
◎実行例

```
>>> from collections import namedtuple Enter
>>> POS = namedtuple("POS", "x y") Enter
>>> p0 = POS(x=12, y=50) Enter
>>> p0.x Enter      ←名前でアクセス
```

```
12
>>> p0[0] Enter       ←インデックスでアクセス
12
>>> p0 Enter
POS(x=12, y=50)
```

　タプルは便利ですが、番号でアクセスする必要があるため、可読性は高くありません。タプルを名前付きタプルで置き換えていくと、ソースコードの可読性は高くなります。タプルとの互換性は担保されているので、ソースコードのメンテナンスをする際などに、置き換えていくとよいかもしれません。

Chapter 3



いろいろな
テキスト処理

このChapterでは、Pythonの標準ライブラリに含まれるモジュールを使用したテキスト処理について解説します。まず正規表現の取り扱いについて説明し、そのあとで、テキストファイル、CSVファイル、JSONファイルの読み書きについて説明します。

3-1 正規表現

この節では、「正規表現」と呼ばれる柔軟な表記法を使用したテキストの検索や置換について説明します。Pythonの標準ライブラリには、正規表現を効率的に扱うモジュールとして「re」が用意されています。

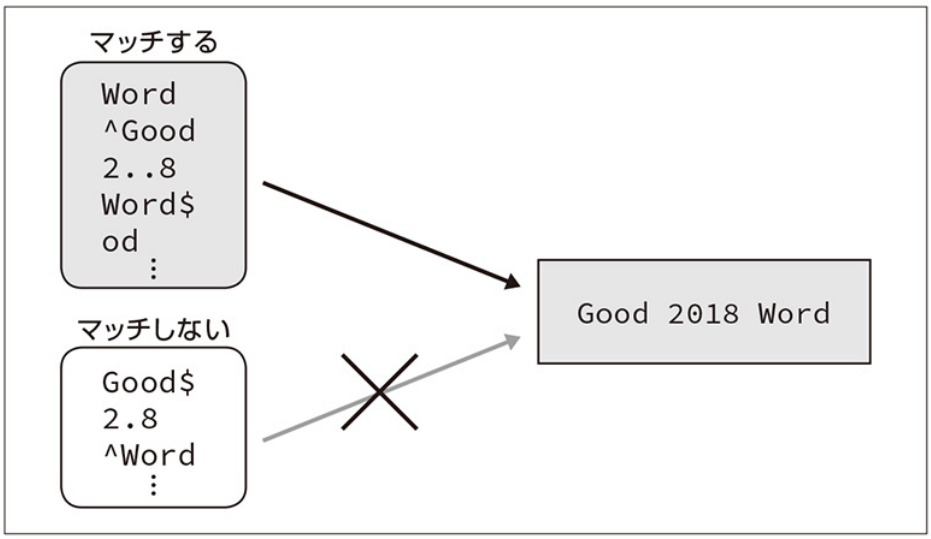
➡094
正規表現を使用して検索を行うには


search関数、match関数を使用する

正規表現では、**特殊文字**（**メタキャラクタ**）を使用したパターンにより、柔軟な検索を行うことが可能です。これを「**パターンマッチ**」と言います。たとえば、「**^**」は文頭、「**$**」は文末、「.」は任意の文字を表す特殊文字です（100の直前のコラム「正規表現の主な特殊文字」参照）。

「Good 2018 Word」という文字列には、「Word」「^Good」や「2..8」などのパターンがマッチします。

⊙「Good 2018 Word」にマッチする／しないパターン



reモジュールには、パターンマッチを行う基本的な関数として**search**関数が用意されています。

◎関数　**search(pattern, string, flags=0)**

カテゴリ	**reモジュール**
引数	**pattern：パターン、string：検索対象の文字列、flags：検索条件を設定するフラグ（095の直前のコラム「search/matchメソッドのフラグ」参照）**
戻り値	**matchオブジェクトもしくはNone**
説明	**引数stringから引数patternを検索しマッチした場合に結果をmatchオブジェクトとして戻す。マッチしなかった場合にはNoneを戻す**
使用例	**r = re.search("^OK", s)**

検索パターンを引数にしてsearch関数を実行すると、パターンにマッチしなかった場合にはNoneが、マッチした場合にはその情報が格納された**match**オブジェクトが戻されます。

したがって、単にマッチしたかどうかの判断はif文で調べられます（マッチした部分の文字列や位置などの情報を取り出す方法については、「095　マッチした部分の文字列や位置を取得するには」参照）。

◎リスト search1.py

```
import re
s = "Good 2018 Word"          ←検索対象の文字列
p = "2..8" ←パターン「2..8」
if re.search(p, s): 1    ←検索を実行
      print("パターン「" + p + "」が見つかりました")
```

1でsearch関数を実行して、if文でパターンが見つかったかどうかを調べています。

■実行結果

```
パターン「2..8」が見つかりました
```

●文字列の先頭でマッチするかを調べる

search関数と似た関数に、**match**関数があります。

◎関数　match(pattern, string, flags=0)

カテゴリ	**reモジュール**
引数	**pattern：パターン、string：検索対象の文字列、flags：検索条件を設定するフラグ**
戻り値	**matchオブジェクトもしくはNone**
説明	**引数stringの先頭から引数patternを検索し、マッチした場合に結果をmatchオブジェクトとして戻す。マッチしなかった場合にはNoneを戻す**
使用例	**m = re.match("<p>", s)**

search関数は文字列の任意の位置にマッチさせることができますが、match関数は文字列の先頭のみにマッチします。

◎実行例

```
>>> import re Enter
```

```
>>> s = "Bad News" Enter
>>> re.search("News", s) Enter　←マッチする
<_sre.SRE_Match object; span=(4, 8), match='News'>
>>> re.match("News", s) Enter　←マッチしない
>>> re.match("Bad", s) Enter　←マッチする
<_sre.SRE_Match object; span=(0, 3), match='Bad'>
```

search/matchメソッドのフラグ

reモジュールに用意されている、match、findall、subメソッドといった検索／置換メソッドの引数**flags**では、検索条件を設定するフラグを指定できます。

⊙search/matchメソッドの主なフラグ

フラグ	説明
re.ASCII	特殊文字「\w、\W、\b、\B、\d、\D、\s、\S 」がASCII文字のみマッチするようにする
re.IGNORECASE	大文字/小文字の相違を無視する
re.MULTILINE	複数行の文字列を指定した場合に「^」が各行の先頭にマッチし、「$」が各行の行末にマッチするようにする
re.DOTALL	特殊文字「.」を改行にもマッチさせるようにする
re.VERBOSE	パターンにコメントを付けたり、途中で改行を許可するなど、より読みやすい書式でパターンを記述する

フラグ（RegexFlagオブジェクト）は「|」で連結して複数指定することが可能です。次の例はre.MULTILINEとre.IGNORECASEの2つのフラグを指定しています。

◎**実行例**

```
>>> m = re.search("^Start", s, flags=re.MULTILINE | re.IGNORECASE) Enter
```

➡095

マッチした部分の文字列や位置を取得するには

matchオブジェクトのメソッドを使用する

search/match関数を実行し引数で指定したパターンにマッチした場合には、マッチした文字列や位置を格納する**match**オブジェクトが戻されます。matchオブジェクトの**group**メソッドを使用するとマッチした文字列が取得できます。

◎メソッド　**group([n])**

カテゴリ	**reモジュールの_sre.SRE_Matchクラス**
引数	**グループ番号（デフォルトは0）**
戻り値	**文字列**
説明	**マッチした文字列を戻す。引数を指定しないか0を指定した場合にはマッチした全体を戻す**
使用例	**s = m.group()**

以下に、パターン「P....n」にマッチした文字列を表示する例を示します。

◎実行例

```
>>> import re Enter
>>> m = re.search("P....n", "Hello Python") Enter
>>> m.group() Enter
'Python'
```

●マッチした位置を表示する

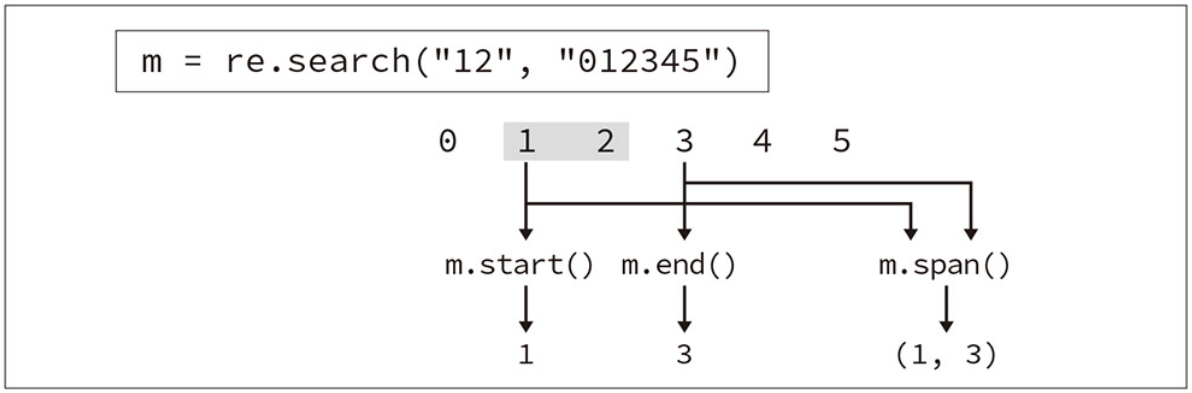
パターンにマッチした位置は、matchオブジェクトの次のメソッドで取得できます。

⊙matchオブジェクトのメソッド

メソッド	説明
start()	開始位置
end()	終了位置
span()	開始位置と終了位置をタプルとして戻す

　位置は最初の文字を0とし、終了位置は最後の文字の次の位置になります。

⊙開始位置、終了位置に注意



◎実行例

```
>>> m = re.search("P....n", "Hello Python") Enter
>>> m.start() Enter   ←開始位置
6
>>> m.end() Enter   ←終了位置
12
>>> m.span() Enter   ←範囲
(6, 12)
```

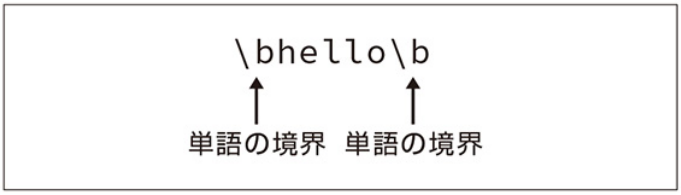
➡096

パターン内のPythonのエスケープシーケンスを無効にするには

raw文字列記法「r"パターン"」を使用する

正規表現の特殊文字と、Pythonの特殊文字が衝突するケースがあります。たとえば、正規表現では「**\b**」は単語の境界を表す特殊文字です。

⦿正規表現では「\b」は単語の境界を表す特殊文字



この場合、search関数のパターンに「\bHello\b」を指定しても文字列「Hello World"」とマッチしません。

◎実行例

```
>>> re.search("\bHello\b", "Hello World") Enter　←マッチしない
```

「\b」はPythonの文字列においてもバックスペースを表す特殊文字となっているため、バックスペースとしてパターンに渡されてしまうからです。したがって、「\\b」として「\」をエスケープしてからパターンに渡す必要があります。

◎実行例

```
>>> re.search("\\bHello\\b", "Hello World") Enter
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
```

別の例として、文字列内の「****」を文字そのものとして検索するには、「\\\\」のように「\」を4つ繋げる必要があります。

◎実行例

```
>>> re.search("\\\\", "Good\Bad") Enter
<_sre.SRE_Match object; span=(4, 5), match='\\'>
```

「\」はPythonにとっても、正規表現にとっても特殊文字となっているため、どちらにも前に「\」を記述してエスケープする必要があるからです。

●便利なraw文字列記法

「\」をそのまま正規表現のパターンに渡すには、次のような「**raw文字列記法**」を使用します。

r"パターン"

これで「\」がPythonで解釈されずにパターンに直接引き渡されます。以下、前述の例に、raw文字列記法を使用した例を示します。

◎実行例

```
>>> re.search(r"\bHello\b", "Hello World") Enter
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> re.search(r"\\",  "Good\Bad") Enter
<_sre.SRE_Match object; span=(4, 5), match='\\'>
```

NOTE Windowsのコマンドプロンプトではフォントの設定がデフォルトの「MSゴシック」の場合、「バックスラッシュ」が「円記号」で表示されます。

➡097

最短マッチを行うには

「*」「+」「?」の代わりに「*?」「+?」「??」を使用する

「*****」「**+**」「**?**」といった繰り返しを表す特殊文字は、範囲が一番長くなるようにマッチします。これを「**欲張りマッチ**」などと言います。たとえば、「+」は直前の文字の1回以上の繰り返しにマッチします。したがって、「**.+**」というパターンは1文字以上の任意の長さの文字列を表します（「.」は任意の文字）。

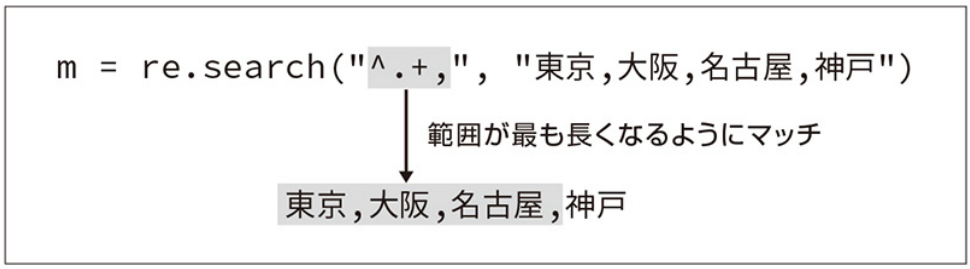
そのため、「"東京,大阪,名古屋,神戸"」という文字列から、最初の「"東京,"」部分を取り出そうとして、パターンに「**^.+,**」を指定してもうまくいきません。

◎実行例

```
>>> m = re.search("^.+,", "東京,大阪,名古屋,神戸") Enter
>>> m.group() Enter
'東京,大阪,名古屋,'
```

文字列の先頭からカンマ「,」までの一番長くなる範囲がマッチするため、このように「"東京,大阪,名古屋,"」が取り出されてしまうからです。

⊙「^.+,"」は先頭からカンマ「,」までの一番長くなる範囲がマッチする



それに対して「+」の後ろに「**?**」を記述して、「**+?**」を指定すると最短マッチになります。

◎実行例

```
>>> m = re.search("^.+?,", "東京,大阪,名古屋,神戸") Enter
>>> m.group()  ←「"東京,」だけが取り出される Enter
'東京,'
```

同様に「*」「?」の代わりに、それぞれ「***?**」「**??**」を指定しても最短マッチになります。

➡098

パターンをグループ化してマッチした部分を複数取り出すには

正規表現をグループ化する

matchオブジェクトの**group**メソッド、**start**メソッド、**end**メソッド、**span**メソッドを引数なしで実行すると、パターンにマッチした全体が取り出されます。

文字列から、正規表現のパターンにマッチした部分を1つもしくは複数取り出したい場合には、パターン内の正規表現を「**( )**」で囲って**グループ化**します。

(正規表現)

こうすると、groupメソッド、startメソッド、endメソッド、spanメソッドの引数に**グループ番号**を指定することで、そのグループの文字列や位置が取り出せます。グループ番号は最初のグループを1とする整数値です。たとえば、文字列"東京,大阪,名古屋,神戸"から、最初のカンマ「,」の前までの「"東京"」のみを取り出すには次のようにします。

◎実行例

```
>>> m = re.search("^(.+?),", "東京,大阪,名古屋,神戸") Enter
>>> m.group() Enter  ←引数を指定しないか0を指定すると全体が取り出される
'東京,'
>>> m.group(1) Enter ←番号を指定するとグループ化した範囲が取り出される
'東京'
>>> m.span(1) Enter  ←"東京"の範囲を表示
(0, 2)
```

●複数のグループを指定する

パターン内に複数のグループを指定してもかまいません。たとえば、名前、年齢、性別がカンマ「,」で区切られている以下の文字列があるとします。名前、年齢、性別を個別に取り出すには3つのグループを作成し、groupメソッドにグループ番号を指定します。

"山田一郎,15,男性"

◎実行例

```
>>> s = "山田一郎,15,男性" Enter
```

```
>>> m = re.search("(.+),(.+),(.+)", s) Enter　←3つのグループを作成
>>> m.group(1) Enter
'山田一郎'
>>> m.group(2) Enter
'15'
>>> m.group(3) Enter
'男性'
```

➡099

マッチしたすべての部分をリストとして取り出すには

findall関数、finditer関数を使用する

search関数では文字列内にパターンにマッチする部分が複数ある場合に、最初にマッチした範囲の情報のみがmatchオブジェクトとして戻されます。それに対して**findall**関数ではマッチしたすべての範囲の文字列を要素とするリストとして戻します。

◎関数　findall(pattern, string, flags=0)

カテゴリ	**reモジュール**
引数	**pattern：パターン、string：検索対象の文字列 flags：検索条件を設定するフラグ(095の直前のコラム「search/matchメソッドのフラグ」参照)**
戻り値	**マッチした範囲を要素とするリスト**
説明	**引数stringから引数patternを検索し、マッチした範囲の文字列を要素とするリストを戻す**
使用例	**l = re.findall(" (.+?) ", s)**

4桁の西暦がカンマ「,」で区切って格納された文字列から、1900年代を取り出すには次のようにします。

◎実行例

```
>>> re.findall(r"\b19\d\d\b", "2003, 1959, 1930, 1943, 2001") Enter
['1959', '1930', '1943']
```

NOTE 「\b」は単語の区切り、「\d」は任意の数字を表す特殊文字です。「\b」がPythonによって解釈されないように、raw文字列記法を使用しています。

パターンが見つからない場合には空のリストが戻されます。

◎実行例

```
>>> re.findall("white", "blue, green blue blue") Enter
[]
```

●グループを指定する

findall関数の、引数のパターン内でグループを指定した場合には、グループにマッチする部分を要素とするリストが戻されます。

次に、文字列"月曜日, 火曜日, 木曜日, 金曜日"から、"曜日"を取り除いたリストを生成する例を示します。この例ではパターン内にグループを1つ指定しています。

◎実行例

```
>>> re.findall(r"\b(.+?)曜日,?","月曜日,  火曜日, 木曜日, 金曜日") Enter
['月', '火', '木', '金']
```

グループが複数ある場合には、個々のグループにマッチする部分を要素とするタプルとして、さらにそのタプルを要素とするリストが生成されます。

たとえば、生年月日が「年/月/日」の形式でカンマ「,」で区切られている文字列から、年、月、日を要素とするタプルのリストを生成するには次のようにします。

◎実行例

```
>>> s = "2003/12/3, 1959/1/14, 1930/9/4, 1943/11/6" Enter
>>> re.findall(r"\b(\d{4})/(\d\d?)/(\d\d?)\b", s) Enter
[('2003', '12', '3'), ('1959', '1', '14'), ('1930', '9', '4'), ('1943', '11', '6')]
```

●マッチした範囲をイテレート可能オブジェクトとして戻すには

findall関数の代わりに、**finditer**関数を使用すると、マッチした部分をリストではなく、イテレート可能なオブジェクトとして戻すことができます。

◎関数　finditer(pattern, string, flags=0)

カテゴリ	reモジュール
引数	pattern：パターン、string：検索対象の文字列、flags：検索条件を設定するフラグ
戻り値	matchオブジェクトを要素とするイテレート可能オブジェクト
説明	引数stringから引数patternを検索し、マッチした場合、その範囲をmatchオブジェクトを要素とするイテレート可能オブジェクトとして戻す
使用例	itr = re.finditer(r"\b(.+?)曜日,?", s)

findall関数を使用して一部の文字列（曜日）を取り除いた上でリストを生成する例を取り上げました（「●グループを指定する」の最初の実行例）。ここでは、この例をもとにfinditer

関数を実行する例を示します。findall関数と異なり、戻り値はmatchオブジェクトを要素とするイテレート可能オブジェクトなので、spanメソッドで範囲を取得することが可能です。

◎リスト finditer1.py

```
import re

itr = re.finditer(r"\b(.+?)曜日,?", "月曜日, 火曜日, 木曜日, 金曜日")
for m in itr:
    print(m.group(1)) 1
    print("範囲:", m.span(1)) 2
```

1で曜日を、**2**でその範囲を表示しています。

■実行結果

```
月
範囲: (0, 1)
火
範囲: (5, 6)
木
範囲: (10, 11)
金
範囲: (15, 16)
```

正規表現の主な特殊文字

Pythonの正規表現で利用可能なよく使う特殊文字をまとめておきます。

⊙正規表現の主な特殊文字

特殊文字	説明
.	改行以外の任意の1文字
^	文字列の先頭
$	文字列の末尾
*	直前の文字(グループ)の0回以上の繰り返し
+	直前の文字(グループ)の1回以上の繰り返し
?	直前の文字(グループ)の0回もしくは1回以上の繰り返し
{m}	直前の正規表現のm回の繰り返し
{m,n}	直前の正規表現のm回からn回までの繰り返し
[文字の並び]	文字の並びのいずれかの文字。文字をハイフン「-」で接続することにより範囲を指定することもできる 例: [abc] a、b、cのどれか [0-9] 0～9のどれか [a-zA-Z] アルファベット
[^文字の並び]	文字の並び以外の文字
(正規表現)	正規表現をグループ化する
\b	単語の境界
\d	数字
\D	数字以外
\s	空白文字
\S	空白文字以外
\w	単語を構成することが可能な文字。ASCIIフラグを指定すると[a-zA-Z0-9_]と同じ
\W	単語構成する文字以外
\\	「\」文字自身
正規表現A|正規表現B	正規表現Aと正規表現Bのどちらか

➡100

文字列をパターンにマッチする部分で分割するには

split関数を使用する

split関数を使用すると、パターンにマッチする部分で文字列を分割し、リストの要素とすることができます。

◎関数　split(pattern, string, maxsplit=0, flags=0)

カテゴリ	reモジュール
引数	pattern：パターン、string：検索対象の文字列、maxsplit：最大分割数、flags：検索条件を設定するフラグ(095の直前のコラム「search/matchメソッドのフラグ」参照)
戻り値	文字列をパターンで分割したリスト
説明	引数patternにマッチした文字列の位置で、引数stringを分割しそれを要素とするリストを戻す
使用例	l = re.split("[\s,]+", s)

以下に、空白文字「\s」もしくはカンマ「,」の並びで文字列を分割して、リストの要素とする例を示します。

◎リスト split1.py

```
import re

s = "田無駅,　花小金井駅\t　,新宿駅　　小平駅　"
l = re.split("[\s,]+", s) 1
for st in l:
　　print(st)
```

1でパターンに「[\s,]+」を指定し、空白文字「\s」もしくはカンマ「,」の1つ以上の繰り返し「+」にマッチする部分に対して、引数sの文字列「s」を分割するようにsplit関数を実行しています。

■実行結果

```
田無駅
花小金井駅
```

新宿駅
小平駅

➡101
パターンにマッチする部分を置換するには


sub関数を使用する

　正規表現のパターンとマッチする範囲を、別の文字列で置換するには、**sub**関数を使用します。

◎関数　sub(pattern, repl, string, count=0, flags=0)

カテゴリ	reモジュール
引数	pattern：パターン、repl：置換後の文字列、string：検索対象の文字列、count：置換が実行される最大回数。省略もしくは0を指定するとすべて、flags：検索条件を設定するフラグ(095の直前のコラム「search/matchメソッドのフラグ」参照)
戻り値	置換されたあとの文字列
説明	引数stringに対して引数patternで指定したパターンにマッチする部分を引数replで指定した文字列で置換する
使用例	r = re.sub("\d+", "", s)

　たとえば、文字列「"hello hello hello"」の行頭の「hello」を「Hello」に置換するには次のようにします。

◎実行例

```
>>> re.sub("^hello", "Hello", "hello hello hello") Enter
'Hello hello hello'
```

●グループをあとから参照する後方参照

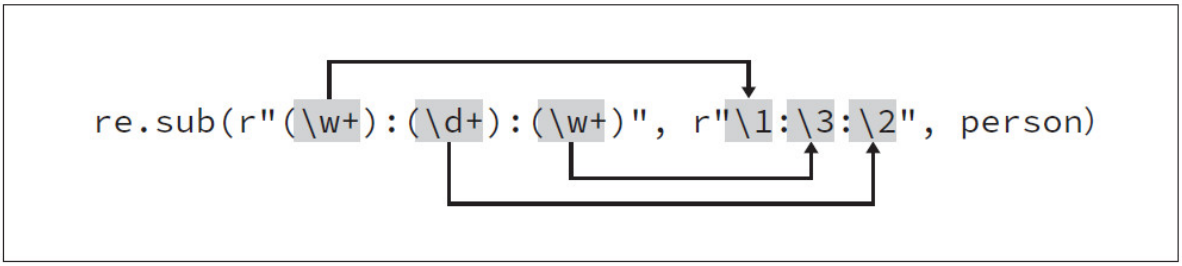
グループ化した正規表現にマッチした範囲は、「**\番号**」で参照できます。これを「**後方参照**」と呼びます。この場合、「\」がPythonによってエスケープされないようにraw文字列記法を使用する必要があります。sub関数の引数replで後方参照を使用して、「名前:年齢:性別」のリストを「名前:性別,年齢,」に変更する例を示します。

◎リスト sub1.py

```
import re

people1 = ["山田太郎:45:男性", "中田功:34:男性", "山本洋子:25:女性"]
people2 = []
for person in people1:
        people2.append(re.sub(r"(\w+):(\d+):(\w+)",  r"\1:\3:\2", person)) 1

print(people2)
```

1で後方参照を使用してsub関数を実行しています。

⊙1のsub関数の仕組み



また、置換後の文字列をリストのappendメソッドを使用してリストpeople2に加えています。

■実行結果

['山田太郎:男性:45', '中田功:男性:34', '山本洋子:女性:25']

ＮＯＴＥ 後方参照はsearchメソッドなどほかのメソッドでも使用できます。以下にコロン「:」で区切られた、3桁の同じ数値の並びを検索する例を示します。

◎実行例

```
>>> re.search(r"^(\d{3}):\1:", "123:123:156") Enter
←マッチする
<_sre.SRE_Match object; span=(0, 8), match='123:123:'>
>>> re.search(r"^(\d{3}):\1:", "123:456:156") Enter
←マッチしない
```

➡102

パターンをコンパイルして実行速度を上げるには

compile関数を使用する

正規表現のパターンをなんども使い回す場合、あらかじめ**compile**関数でコンパイルしておくとより高速な検索が可能になります。

◎関数　compile(pattern, flags=0)

カテゴリ	**reモジュール**
引数	**pattern：パターン、flags：検索条件を設定するフラグ(095の直前のコラム「search/matchメソッドのフラグ」参照)**
戻り値	**regexオブジェクト**

説明	正規表現をコンパイルしてregexオブジェクトとして戻す
使用例	p = re.compile(":男性$")

コンパイルしないパターンの場合、search、match、split、sub、findall、finditerなどが使用できました。

compile関数で生成されるregexオブジェクトによるパターンマッチでは、同じ名前の関数に対応したregexクラスのメソッドを使用します。たとえば、search関数の代わりにsearchメソッドを使用します。

例を示しましょう。コンパイルしないパターンで、「名前:年齢:性別」の文字列を要素とするリストから、男性のみの要素からなるリストを生成するには、次のようにします。

◎リスト comp1.py

```
import re
people = ["山田太郎:45:男性", "中田功:34:男性",
            "山本洋子:25:女性",  "井上花子:23:女性"]
men = []
for person in people:
    m = re.search(":男性$", person) 1
    if m:
        men.append(person) 2

print(men)
```

1でsearch関数を使用してリストから最後が「":男性"」で終わる要素を取り出し、2でappendメソッドによりリストmenに加えています。

■実行結果

```
['山田太郎:45:男性', '中田功:34:男性']
```

コンパイルしたパターンを使用するように、上記のcomp1.pyを変更した例を示します。

◎リスト comp2.py

```
import re
people = ["山田太郎:45:男性", "中田功:34:男性",
          "山本洋子:25:女性",  "井上花子:23:女性"]
men = []
p = re.compile(":男性$") 1
for person in people:
    m = p.search(person) 2
    if m:
        men.append(person)

print(men)
```

1でcompile関数を使用してパターンからregexオブジェクトを生成し、**2**でsearchメソッドにより検索を行っています。

■実行結果

```
['山田太郎:45:男性', '中田功:34:男性']
```

NOTE 実際に筆者のマシン（iMac： 3.1GHz Intel Core i5）でcomp1.pyとcomp2.pyの処理を10万回実行し、実行時間を測定したところ、次のような結果になりました。
comp1.py　約0.64秒
comp2.py　約0.38秒

Windowsのコマンドプロンプトの文字コードをUTF-8にする

Python 3ではプログラムの文字エンコーディングはUTF-8が基本です。macOSやLinuxのターミナルはデフォルトの文字エンコーディングはUTF-8ですが、Windowsのコマンドプロンプトの文字エンコーディングはShift-JISがデフォルトです。他のOSと合わせるには、次のようにして、コマンドプロンプトの文字エンコーディングをUTF-8に設定するとよいでしょう。

◎実行例

```
>chcp 65001 Enter
```

元に戻すには次のようにします。

◎実行例

```
>chcp 932 Enter
```

3-2 テキストファイルの読み書き

この節から、Pythonプログラムを使用したテキストファイルの読み書きについて説明します。まずは、最も基本的なプレーンテキストファイルの取り扱いについて説明します。

➡103

テキストファイルの内容をまとめて読み込むには

readメソッドを使用する

テキストファイルの内容をまとめて読み込むには、あらかじめ**open**関数でファイルを開き、**read**メソッドで読み込みます。ファイルを使い終わったら**close**メソッドで閉じます。

◎ 関 数 open(file, mode='r', encoding=None, newline=None)

カテゴリ	**組み込み関数**
引数	**file：ファイルのパス、mode：'r' 読み込み用　'w' 書き込み用　'a' 追記　'x' 排他的書き込み　't' テキストモード（デフォルト）　'b' バイナリーモード　encoding：文字エンコーディング、newline：None（デフォルト：ユニバーサル改行モード）　'\n'　'\r'　'\r\n'**
戻り値	**ファイルオブジェクト**
説明	**引数fileで指定したファイルを開きファイルオブジェクト（テキストモードの場合にはTextIOBaseクラス）を返す**
使用例	**f = open("sample.txt", "r", encoding="utf8")**

◎メソッド read(size)

カテゴリ	**io.TextIOBaseクラス**

引数	最大文字数（Noneの場合にはファイルの最後まで）
戻り値	文字列
説明	ファイルオブジェクトから最大引数sizeで指定した文字数を読み込む
使用例	s= f.read()

◎メソッド　close()

カテゴリ	io.IOBaseクラス
引数	なし
戻り値	なし
説明	ファイルを閉じる
使用例	f.close()

以下に、カレントディレクトリに保存されている、文字エンコーディングがutf-8のテキストファイル「sample.txt」を読み込んで、表示する例を示します。

◎リスト read1.py

```
f = open("sample.txt", "r", encoding="utf8")
contents = f.read() 1
print(contents, end="") 2
f.close()
```

1でreadメソッドを使用してテキストファイルの中身を変数contentsに読み込んで、2で表示しています。読み込んだ文字列の末尾には改行が含まれているので、2のprint文では引数で「end=""」を指定し、改行を抑制しています。

■実行結果

```
1.文字列の操作
文字列を連結する+演算子
インデックスで文字列から文字を取り出す
str関数を使用して数値を文字列に変換する
```

int関数を使用して文字列を数値に変換する

●任意のディレクトリからファイルを読み込めるようにする

　open関数の引数に、プログラムファイルからの相対パスでファイルのパスを指定した場合、プログラムファイルのディレクトリから実行しないとファイルが見つからないといったエラーになります。

　たとえば、ホームディレクトリ→Documentsディレクトリ→PythonProgディレクトリに「read1.py」と「sample.txt」を保存してある場合、Documentsディレクトリから「read1.py」を実行すると次のようなエラーになります。

◎実行例：Windowsの場合

```
C:\Users\o2\Documents>python PythonProg/read1.py Enter
Traceback (most recent call last):
  File "PythonProg/read1.py", line 1, in <module>
    f = open("sample.txt", "r", encoding="utf8")
FileNotFoundError: [Errno 2] No such file or directory: 'sample.txt'
```

◎実行例：macOSの場合

```
$ python3 PythonProg/read1.py Enter
Traceback (most recent call last):
  File "PythonProg/read1.py", line 1, in <module>
    f = open("sample.txt", "r", encoding="utf8")
FileNotFoundError: [Errno 2] No such file or directory: 'sample.txt'
```

　OSによるパスの違いを吸収するには、**os.path**モジュール（116で解説）に用意されている、パスからファイル名なしのディレクトリのみを取り出す**dirname**関数（116で解説）、およびパスを接続する**join**関数（118で解説）を使用すると便利です。

プログラムファイルのパスは特殊変数「**__file__**」に格納されているため、次のようにすると任意のディレクトリから実行できるようになります。

◎リスト read2.py

```
import os.path

dirname = os.path.dirname(__file__) 1
path = os.path.join(dirname, "sample.txt") 2
f = open(path, "r", encoding="utf8")
contents = f.read()
print(contents, end="")
f.close()
```

1で「__file__」からdirname関数でディレクトリのパスを取り出し、**2**のjoin関数を「sample.txt」でパスと接続しています。

■実行結果

```
1.文字列の操作
文字列を連結する+演算子
インデックスで文字列から文字を取り出す
str関数を使用して数値を文字列に変換する
int関数を使用して文字列を数値に変換する
```

with文を使用するとよりシンプルな記述が可能

Pythonに用意されている**with文**を使用すると、よりシンプルにファイルの読み書きといった処理が行えます。

```
with open(～) as 変数:
    処理
```

上記のようにwithのあとにopen関数を、asのあとにファイルオブジェクトを格納する変数を記述します。with文のブロックでは開いたファイルの処理を記述します。ブロックを抜けると自動的にcloseメソッドが実行されファイルが閉じられます。

以下に、前出のread2.pyを、with文を使用するように変更した例を示します。

◎リスト read3.py

```
import os.path

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "sample.txt")

with open(path, "r", encoding="utf8") as f:
    contents = f.read()
    print(contents, end="")
```

■実行結果

```
1.文字列の操作
文字列を連結する+演算子
インデックスで文字列から文字を取り出す
str関数を使用して数値を文字列に変換する
int関数を使用して文字列を数値に変換する
```

➡104

テキストファイルの各行をリストの要素として取り出すには

readlinesメソッドを使用する

テキストファイルを読み込んで、ファイルの各行を要素とするリストを生成するには、**readlines**メソッドを使用します。

◎メソッド　readlines(size=-1)

カテゴリ	**io.TextIOBaseクラス**
引数	**最大文字数（「-1」もしくは指定しない場合にはファイルの最後まで）**
戻り値	**行を要素とするリスト**
説明	**テキストファイルを読み込んで、行単位の文字列を要素とするリストを戻す**
使用例	**s = f.readlines()**

以下にreadlinesメソッドにより「sample.txt」をリストとして読み込んで表示する例を示します。

◎リスト readlines1.py

```
import os.path

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "sample.txt")
f = open(path, "r", encoding="utf8")
contents = f.readlines() 1
print(contents, end="") 2
f.close()
```

1でreadlinesメソッドを使用してファイルの内容をリストとして読み込み、変数contentsに代入し、**2**で表示しています。

■実行結果

```
['1.文字列の操作\n', '文字列を連結する+演算子\n', 'インデックスで文字列から文字を取り出す\n', 'str関数を使用して数値を文字列に変換する\n', 'int関数を使用して文字列を数値に変換する\n']
```

●改行を取り除くには

　readlines1.pyの実行結果を見るとわかるように、リストの各要素は最後に改行コード"\n"が付加されています。これを削除するには、たとえば、map関数（064で解説）を使用して次のようにします。

◎リスト readlines2.py

```
import os.path

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "sample.txt")
f = open(path, "r", encoding="utf8")
contents = f.readlines()

newlines = list(map(lambda s:s.rstrip("\n"), contents)) 1
print(newlines)
f.close()
```

　追加したのは1のステートメントです。map関数の引数では、lambda式（「Chapter 2　コレクションの取り扱い」の「lambda式」参照）でrstripメソッド（014で解説）を"\n"を引数に実行して、リストcontentsの各要素から行末の改行「\n」を取り除いています。map関数の結果をlistコンストラクタに渡してリストに変換し、変数newlinesに代入しています。
　これで、リストnewlinesにはreadlines関数で読み込んだファイルの各行が改行なしで要素として格納されます。

■実行結果

```
['1.文字列の操作', '文字列を連結する+演算子', 'インデックスで文字列から文字を取り出す', 'str関数を使用して数値を文字列に変換する', 'int関数を使用して文字列を数値に変換する']
```

➡105

テキストファイルの各行を1行ずつ読み込むには

readlineメソッドを使用する、ファイルオブジェクトをイテレートする

readメソッド、readlinesメソッドではテキストファイルの中身を一度にまとめて読み込むため、ファイルのサイズが大きいとその分メモリを消費してしまいます。

ファイルのサイズが大きい場合や不明な場合には、可能なら**readline**メソッドで1行ずつ読み込んで処理するほうがよいでしょう。

◎メソッド　readline(size=-1)

カテゴリ	**io.TextIOBaseクラス**
引数	**最大文字数（指定しない場合には行の最後まで）**
戻り値	**文字列**
説明	**ファイルオブジェクトから1行ずつ読み込んで文字列として戻す**
使用例	**line = f.readline()**

readlineメソッドは、ファイルの最後まで読み込まれると空文字列「""」を戻します。また、読み込まれた文字列の最後には改行コードが付いているので、必要に応じてrstripメソッドなどで取り除きます。

◎リスト readline1.py

```
import os.path

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "sample.txt")
f = open(path, "r", encoding="utf8")
while True: 1
    l = f.readline() 2
    if l == "":
        break 3
    l = l.rstrip("\n") 4
    print(l)
f.close()
```

1のwhile文では、**2**でreadlineメソッドを使用して、ファイルから1行ずつ読み込んでいきます。ファイルの最後まで読み込まれると、**3**のbreak文でループを抜けます。**4**でrstripメソッドを使用して行末の「\n」を取り除いて表示しています。

■実行結果

```
1.文字列の操作
文字列を連結する+演算子
インデックスで文字列から文字を取り出す
str関数を使用して数値を文字列に変換する
int関数を使用して文字列を数値に変換する
```

●ファイルオブジェクトをイテレートする

実際にはreadlineメソッドを使用しなくても、ファイルオブジェクトをfor文などで直接イテレートすることで1行ずつ読み込むことができます。

◎リスト readline2.py

```
import os.path

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "sample.txt")
f = open(path, "r", encoding="utf8")
for l in f:
    l = l.rstrip("\n") 1
    print(l)

f.close()
```

1のfor文でファイルから1行ずつ取り出して表示しています。inのあとにopen関数で開いたファイルオブジェクトfを直接指定している点に注目してください。

■実行結果

```
1.文字列の操作
文字列を連結する+演算子
インデックスで文字列から文字を取り出す
str関数を使用して数値を文字列に変換する
int関数を使用して文字列を数値に変換する
```

➡106

文字列をテキストファイルに書き出すには

writeメソッドを使用する

open関数を使用して書き込みモードで開いたファイルに、文字列を書き出すには、**write**メソッドを使用します。

◎メソッド　write(s)

カテゴリ	**io.TextIOBaseクラス**
引数	**書き出す文字列**
戻り値	**書き出した文字数**
説明	**引数sで指定した文字列をファイルオブジェクトに書き出す**
使用例	**f.write("Python" + "\n")**

writeメソッドは改行コードを自動で書き出しません。改行を書き出したい場合には"\n"を追加します。デフォルトではユニバーサル改行モード（107のあとのコラム「ユニバーサル改行モード」）に設定されているため、OSに応じた改行コードに自動的に変換されます。

◎リスト write1.py

```
import os.path

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "output.txt")
f = open(path, "w", encoding="utf8") 1
f.write("Hello") 2
f.write("Python" + "\n") 3
f.write("Python入門" + "\n")
f.close()
```

1でopen関数を使用して、**"w"**（書き込みモード）でファイル「output.txt」を開いています。**2**で改行なし、**3**で改行付きで文字列をファイルに書き出しています。

■ 実行結果 output.txt

```
HelloPython
Python入門
```

●ファイルに追記する

既存のファイルに追加するには、open関数のモードで**"a"**を指定してファイルを開きます。

◎リスト write2.py（一部）

```
f = open(path, "a", encoding="utf8")
```

●ファイルが存在する場合にエラーにする

書き込み対象のファイルが存在する場合にエラー（**FileExistsError**例外）になるようにするには、open関数のモードで**"x"**を指定します。

以下に、try～except文で、ファイル「output.txt」が存在していた場合に、FileExistsError例外を捕まえて「ファイルが存在します」と表示する例を示します。

◎リスト write3.py

```
import os.path

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "output.txt")
try:
    f = open(path, "x", encoding="utf8")   1
    f.write("Hello")
    f.write("Python" + "\n")
    f.write("Python入門" + "\n")
    f.close()
except FileExistsError:   2
    print("ファイルが存在します")
```

1で、open関数を、モードを"x"にして実行しています。**2**のexcept文でFileExistsError例外を捕まえています。

■実行結果

```
ファイルが存在します
```

➡107

リストの要素をまとめてテキストファイルに書き出すには

writelinesメソッドを使用する

ファイルにリストやタプルなどシーケンス型のオブジェクトの要素をまとめて書き出すには、**writelines**メソッドを使用します。

◎メソッド　writelines(lines)

カテゴリ	**io.TextIOBaseクラス**
引数	**書き出す文字列（シーケンス型のオブジェクト)**
戻り値	**なし**
説明	**引数linesの要素をファイルオブジェクトに書き出す**
使用例	**f.writelines(mylist)**

writelinesメソッドでは、要素の区切り文字は書き出されません。そのため、すべての要素が文字列として連結されます。

◎リスト writelines1.py

```
import os.path
names = ("櫻井五郎", "白石孝", "猪熊太郎", "大川花子")

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "output2.txt")
f = open(path, "w", encoding="utf8")
```

```
f.writelines(names) 1
f.close()
```

1でリストnamesをwritelinesメソッドで書き出しています。結果は次のように1行になります。

■実行結果 output2.txt

```
櫻井五郎白石孝猪熊太郎大川花子
```

たとえば、改行で区切りたければ、要素にあらかじめ改行コード「\n」を追加しておきます。

◎リスト writelines2.py

```
import os.path
names = ("櫻井五郎", "白石孝", "猪熊太郎", "大川花子")

names2 = map(lambda n:n + "\n", names) 1
dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "output3.txt")
f = open(path, "w", encoding="utf8")
f.writelines(names2) 2
f.close()
```

1でmap関数（064で解説）を実行し、タプルnamesの各要素に改行コード「\n」を追加して変数names2に代入しています。**2**でwritelinesメソッドを実行し、リストnames2をファイルに書き出しています。

■実行結果 output3.txt

```
櫻井五郎
白石孝
猪熊太郎
```

大川花子

次のようにwith文を使用して書き出してもかまいません。

◎リスト writelines3.py（一部）

```
with open(path, "w", encoding="utf8") as f:
    f.writelines(names2)
```

ユニバーサル改行モード

テキストファイルの改行コードは、OSに応じて標準とするものが異なります。

⊙主なモジュール

改行コード	標準とするOS
CRLF("\r\n")	Windows
LF("\n")	macOSやLinuxなど
CR("\r")	旧Mac OS(OS9以前)

Pythonのopen関数では、デフォルトでOSに応じた改行コードの相違を吸収するための**ユニバーサル改行モード**が有効になっています。これは、Pythonプログラム内部ではLF（"\n"）が標準改行コードとして扱われ、ファイルの読み込み時には、CRLF（"\r\n"）もしくはCR（"\r"）が、LF("\n")に自動的に変換されるというものです。

また、書き出し時には文字列内のLF("\n")がOSに応じた改行コードに変更されます。たとえばWindowsならばCRLF（"\r\n"）に変換され、macOSやLinuxならばLF（"\n"）のまま出力されます。

ユニバーサル改行モードをオフにし、指定した改行コードを使用するにはopen関数（103で解説）の引数**newline**を設定します。たとえば改行コードを、CRLF（"\r\n")にするには次のようにします。

```
f2 = open("out.txt", "w", encoding="utf8", newline="\r\n")
```

なお、テキストファイルの読み書き時に改行コードを変換しないようにするには、「newline=""」とします。

```
f = open("out.txt", "w", encoding="utf_8", newline="")
```

3-3 CSVファイルの取り扱い

複数の項目をカンマで区切るCSV（Comma Separated Value）は、古くから使用されるテキストベースのフォーマットです。この節では、csvモジュールを使用したCSVファイルの読み書きについて解説します。

➡108

CSVファイルを読み込むには

reader関数を使用する

Pythonの標準ライブラリに含まれる**csv**モジュールには、CSVファイルを読み込む**reader**関数が用意されています。

◎ 関数 reader(csvfile, dialect='excel', **fmtparams)

カテゴリ	**csvモジュール**
引数	**csvfile：CSVファイル、dialect：特定の書式のための設定（デフォルトはExcel形式）、fmtparams：区切り文字などの書式パラメータ**
戻り値	**readerオブジェクト**
説明	**CSVファイルから各行を読み込み、イテレート可能なreaderオブジェクトとして戻す。各行は文字列のリストとなる**

reader関数で返されるイテレート可能なオブジェクトは、CSVファイルの1行分のデータを要素とするリストで構成されます。したがって、for文と組み合わせることでファイル全体を読み込めます。以下に、CSVファイル「meibo.csv」を読み

込んで表示する例を示します。このCSVファイルは、「shift_jis」の文字エンコーディングで保存されています。

◎リスト reader1.py

```
import csv 1
import os.path

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "meibo.csv")
f = open(path, "r", encoding="shift_jis") 2
reader = csv.reader(f) 3
for l in reader:
    print(l)    4
f.close()
```

1でcsvモジュールをインポートしています。
2でencoding引数を「shift_jis」に設定してopen関数を実行し、CSVファイル「meibo.csv」を開いています。
3でreader関数を実行してreaderオブジェクトを生成し**4**のfor文で1行ずつ取り出して表示しています。

■実行結果

```
['会員番号', '名前', '出身地', '年齢', '性別']
['1', '井上五郎', '東京都', '44', '男性']
['2', '山田太郎', '千葉県', '33', '男性']
['3', '江藤花子', '東京都', '42', '女性']
['4', '小泉純一', '神奈川県', '19', '男性']
['5', '加藤道夫', '東京都', '59', '男性']
['6', '大木薫', '埼玉県', '23', '女性']
```

CSVファイルとは

CSV（Comma Separated Value）とは、その名が示す通り、カンマ（Comma）「,」で値（Value）を区切る（Separated）という、シンプルなテキストファイルのフォーマットです。Excelなどの表計算ソフトとのデータの受け渡しにしばしば使用されます。

◎リスト meibo.csv

```
会員番号,名前,出身地,年齢,性別
1,井上五郎,東京都,44,男性
2,山田太郎,千葉県,33,男性
3,江藤花子,東京都,42,女性
4,小泉純一,神奈川県,19,男性
5,加藤道夫,東京都,59,男性
6,大木薫,埼玉県,23,女性
```

➡109

文字列をCSVファイルに書き出すには

writer関数、writerowメソッドを使用する

文字列をCSVファイルに書き出すには、あらかじめCSVファイルのファイルオブジェクトを引数にして**writer**関数を実行して、writerオブジェクトを生成しておきます。

◎　関　数　writer(csvfile, dialect='excel', **fmtparams)

カテゴリ　csvモジュール

引数	csvfile：CSVファイル、dialect：特定の書式のための設定（デフォルトはExcel形式）、fmtparams：区切り文字などの書式パラメータ
戻り値	writerオブジェクト
説明	CSVファイルに書き込みを行うwriterオブジェクトを生成して戻す
使用例	writer = csv.writer(f)

生成されたwriterオブジェクトを使用してリストとして用意した1行分のデータを書き込むには、**writerow**メソッドを使用します。

◎メソッド　writerow(row)

カテゴリ	csvモジュール
引数	1行に書き出すリストもしくはタプル
戻り値	なし
説明	引数rowで指定したリストをwriterオブジェクトに書き出す
使用例	writer.writerow(first)

余分な改行が入ることを抑制するため、書き込むCSVファイルはopen関数の引数に「**newline=''**」を指定し、改行コードを変換しないように設定して開きます（「Chapter 3　いろいろなテキスト処理」の「ユニバーサル改行モード」）。

以下に、このレシピの例を示します。この例では、リストで指定したデータをCSVファイル「progs1.csv」に書き出します。

◎リスト writer1.py

```
import csv
import os.path

dirname = os.path.dirname(__file__)
```

```
path = os.path.join(dirname, "progs1.csv")
f = open(path, "w", encoding="shift_jis", newline="")
writer = csv.writer(f)  1
writer.writerow(["番号", "名称", "読み"])
writer.writerow([1, "Python", "パイソン"])
writer.writerow([2, "Java", "ジャバ"])  2
writer.writerow([3, "JavaScript", "ジャバスクリプト"])
writer.writerow([4, "Ruby", "ルビー"])
f.close()
```

1でwriter関数によりwriterオブジェクトを生成しています。2で5つのwriterowメソッドの引数にリストを指定し、それぞれ1行分のデータを書き出しています。

■実行結果 progs1.csv

```
番号,名称,読み
1,Python,パイソン
2,Java,ジャバ
3,JavaScript,ジャバスクリプト
4,Ruby,ルビー
```

●区切り文字を変更するには

CSVファイルではデータの区切り文字はカンマ「,」ですが、readerメソッド、writerメソッドで引数**delimiter**を指定することにより、別の区切り文字で読み書きすることもできます。たとえば、「writer1.py」を変更し、区切り文字をコロン「:」にして書き出すには次のようにします。

◎リスト writer2.py（writer関数部分）

```
writer = csv.writer(f, delimiter=':')
```

writer2.pyを実行すると、区切り文字がカンマ「,」の代わりにコロン「:」になります。

■実行結果 progs2.csv

```
番号:名称:読み
1:Python:パイソン
2:Java:ジャバ
3:JavaScript:ジャバスクリプト
4:Ruby:ルビー
```

➡110

CSVファイルを辞書形式で読み込むには

DictReaderクラスを使用する

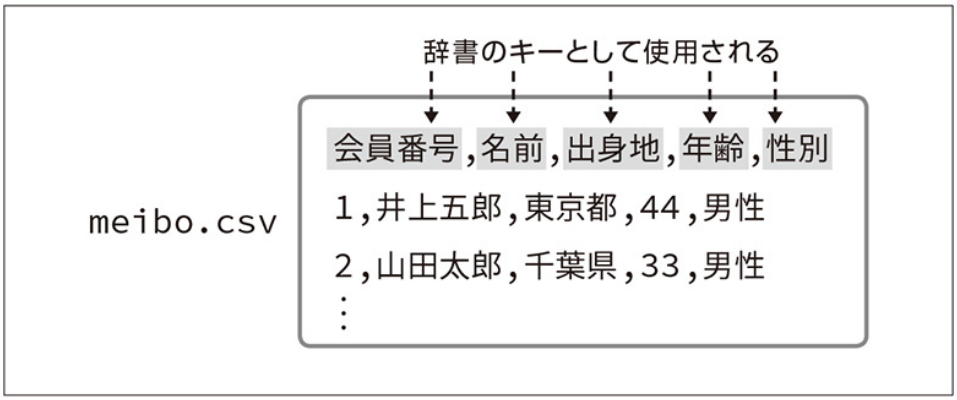
DictReaderクラスを使用すると、CSVファイルを、キーと値の対応の辞書形式で読み込むことができます。DictReaderクラスのコンストラクタを以下に示します。

◎ コンストラクタ DictReader(csvfile, fieldnames=None,dialect='excel', **fmtparams)

カテゴリ	**csvモジュール**
引数	**csvfile：CSVファイル、fieldnames：キーとして使用するリスト（デフォルトは最初の行の値）、dialect：特定の書式のための設定（デフォルトはExcel形式）、fmtparams：区切り文字などの書式パラメータ**
戻り値	**DictReaderオブジェクト**
説明	**CSVファイルから各行を読み込み、イテレート可能なDictReaderオブジェクトとして戻す。各行は辞書（OrderedDictクラス）となる**
使用例	**reader = csv.DictReader(f)**

引数**fieldnames**を指定しない場合には、最初の行のデータが辞書のキーとして使用されます。

⊙引数fieldnamesを指定しない場合は最初の行が辞書のキーとなる



次にカレントディレクトリのmeibo.csvを読み込んで表示する例を示します。

◎リスト dictreader1.py

```
import csv
import os.path

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "meibo.csv")
f = open(path, "r", encoding="shift_jis")
reader = csv.DictReader(f) 1
for line in reader:
      print(line["会員番号"], line["名前"], line["出身地"], 2
            line["年齢"], line["性別"])   3

f.close()
```

1でDictRreaderオブジェクトを生成し、**2**のfor文で1行ずつ読み込んで表示しています。**3**のprint関数では、「line["会

員番号"]」のように辞書として値にアクセスしている点、およびキーには最初の行のデータが使用されている点に注目してください。

■実行結果

```
1 井上五郎 東京都 44 男性
2 山田太郎 千葉県 33 男性
3 江藤花子 東京都 42 女性
4 小泉純一 神奈川県 19 男性
5 加藤道夫 東京都 59 男性
6 大木薫 埼玉県 23 女性
```

➡111

CSVファイルに辞書形式で書き込むには

DictWriterクラスを使用する

DictWriterクラスを使用すると、辞書形式のデータをCSVファイルに書き込むことができます。DictWriterクラスのコンストラクタを以下に示します。

◎コンストラクタ　DictWriter(csvfile, fieldnames, dialect='excel', **fmtparams)

カテゴリ	**csvモジュール**
引数	**csvfile：CSVファイル、fieldnames：キーとして使用するリスト　dialect：特定の書式のための設定（デフォルトはExcel形式）　fmtparams：区切り文字などの書式パラメータ**
戻り値	**DictWriterオブジェクト**
説明	**CSVファイルに辞書形式のデータを書き出すためのDictWriterオブジェクトを生成する**

使用例	writer = csv.DictWriter(f, fieldnames=fieldnames)

writer1.py（109で解説）と同様に、DictWriterオブジェクトでは、writerowメソッドを使用して辞書形式で要素を書き込みます。また、**writeheader**メソッドを使用して、DictWriterコンストラクタの引数fieldnamesで指定したヘッダー部分のデータを書き出します。

◎リスト dictwriter1.py

```
import csv
import os.path

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "progs3.csv")
f = open(path, "w", encoding="shift_jis", newline="")

fieldnames = ["番号", "名称", "読み"] 1
writer = csv.DictWriter(f, fieldnames=fieldnames) 2
writer.writeheader() 3
writer.writerow({"番号":1, "名称":"Python", "読み":"パイソン"})
writer.writerow({"番号":2, "名称":"Java", "読み":"ジャバ"})
4
writer.writerow({"番号":3, "名称":"JavaScript", "読み":"ジャバスクリプト"})
writer.writerow({"番号":4, "名称":"Ruby", "読み":"ルビー"})
f.close()
```

1でヘッダーとして使用する文字列をリストにして、変数fieldnamesに代入しています。2でそのリストを引数にしてDictWriterオブジェクトを生成し、変数writerに代入しています。

3でwriteheaderメソッドを使用してヘッダーを書き出し、4でwriterowメソッドにより各行を書き出しています。引数

には辞書形式で値を指定している点に注目してください。

■実行結果 progs3.csv

```
番号,名称,読み
1,Python,パイソン
2,Java,ジャバ
3,JavaScript,ジャバスクリプト
4,Ruby,ルビー
```

3-4 JSONデータの取り扱い

JSON（JavaScript Object Notation）は、テキストベースの軽量データ交換フォーマットです。この節ではJSONデータを読み込んでPythonのオブジェクトに変換する方法、および、逆にPythonのオブジェクトをJSONデータに変換する方法について説明します。

➡112

JSONファイルを読み込んでPythonのオブジェクトに変換するには

load関数を使用する

JSON形式のテキストファイルを読み込んで、データをPythonのオブジェクトに変換するには、**load**関数を使用します。

◎関数　load(fp)

カテゴリ	**jsonモジュール**
引数	**ファイルオブジェクト**
戻り値	**Pythonのオブジェクト**
説明	**テキストファイルからJSONデータを読み込んでPythonのオブジェクトに変換する**
使用例	**json = json.load(file)**

load関数で読み込んだデータは、データ型に応じてリストや辞書といったPythonのオブジェクトに変換されます　（114の直前のコラム「JSONのオブジェクトとPythonのオブジェクトの対応」参照）。

ここでは、次のような、顧客情報を管理するJSONファイル「customer1.json」を例に説明しましょう。

◎リスト customer1.json

```
{
  "customers":[
    {
      "name": "井上三郎",
      "age": 45,
      "mail": "save@example.com"
    },
    {
      "name": "田中花子",
      "age": 51,
      "mail": "hanar@example.com"
    },
    {
      "name": "江藤真",
      "age": 23,
      "mail": "makoto2@example.com"
    }
  ]
}
```

トップの階層は要素数が1の連想配列です。"customers"をキーにして、3件の顧客データを配列として保存しています。個々の顧客のデータは"name"（名前）、"age"（年齢）、"mail"（メールアドレス）の3種類です。

●JSONファイルを読み込む

以下に、load関数を使用してカレントディレクトリの「customer1.json」を読み込んで、Pythonのオブジェクトに変換するプログラム例を示します。

◎リスト load1.py

```
import os
import json 1

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "customer1.json")
in_file = open(path, "r",   encoding="utf-8") 2
json_obj = json.load(in_file) 3
print(json_obj) 4
in_file.close()
```

1でjsonモジュールをインポートしています。2で、open関数によりcustomer1.jsonをオープンしてファイルオブジェクトを変数in_fileに格納しています。3でload関数を使用してin_fileからJSONデータを読み込み、Pythonのオブジェクトに変換して変数json_objに格納しています。4でそれをprint関数で表示しています。

■実行結果

```
{'customers': [{'name': '井上三郎', 'age': 45, 'mail': 'save@example.com'}, {'name': '田中花子', 'age': 51, 'mail': 'hanar@example.com'}, {'name': '江藤真', 'age': 23, 'mail': 'makoto2@example.com'}]}
```

●load関数で読み込んだデータを処理する

load関数で読み込んだデータは、オブジェクトは辞書（dict型）、配列はリスト（list型）として扱えます（「Chapter 3 いろいろなテキスト処理」の「JSONのオブジェクトとPythonのオブジェクトの対応」参照）。たとえば、前述の「customer1.json」を読み込んで、Pythonのオブジェクト

「json_obj」に変換した場合、最初の顧客の名前（name）には次のようにしてアクセスできます。

```
json_obj["customers"][0]["name"]
```

以下に、読み込んだデータをfor文で処理して、顧客の「名前」「年齢」「メールアドレス」を順に表示する例を示します。

◎リスト load2.py

```
import os
import json

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "customer1.json")
in_file = open(path, "r", encoding="utf-8")
json_obj = json.load(in_file)

customers = json_obj["customers"] 1

for customer in customers:
    print(customer["name"], str(customer["age"]) + "才", 2
        customer["mail"])  3
in_file.close()
```

変換後のPythonのオブジェクト「json_obj」のトップレベルの階層は辞書（dict型）です。1で"customers"をキーに顧客データを取り出し、変数customersに代入しています。
2のfor文で、顧客を一人ずつ取り出し、3のprint関数で「名前 ～才 メールアドレス」の形式で表示しています。

■実行結果

```
井上三郎 45才 save@example.com
```

```
田中花子 51才 hanar@example.com
江藤真 23才 makoto2@example.com
```

NOTE age（年齢)はint型となるため、文字列「"才"」と連結する際にstr関数で文字列に変換している点に注目してください。

JSONとは

JSONは、「JavaScript Object Notation」の略称で、その名前が示す通り、もともとはJavaScriptにおけるオブジェクトの記述用のテキストフォーマットとして開発されました。現在ではその利便性から多くの言語でサポートされています。

JSONのデータは、全体を「{ }」で囲み、コロン「:」で接続したキー（プロパティ）となる文字列と値のペアを作り、各ペアをカンマ「,」で区切って指定する、いわゆる連想配列形式です。

◎リスト JSONのデータ

```
{
  "name": "sample.py",
  "year": 2018,
  "version": "1.2.0",
  "homepage": "http://example.com/test"
}
```

なお、JavaScriptのオブジェクトの場合にはキーは文字列でなくてもかまいませんが、JSONの場合には必ずクォーテーション「"」で囲って文字列にします。値には、数値、文

字列、真偽値（trueまたはfalse）、配列、連想配列、nullが指定可能です。
　配列は、Pythonのリストと同じく要素をカンマ「,」で区切り、全体を「[ ]」で囲みます。次の例は「colors」というキーの値として、4つの要素をもつ配列を記述しています。

◎リスト JSONの配列

```
{
  "colors": ["白", "黒", "紫", "緑"]
}
```

➡113

JSONデータの文字列をPythonのオブジェクトに変換するには

loads関数を使用する

　JSONデータが格納された文字列を、Pythonのオブジェクトに変換するには、**loads**関数を使用します。

◎関数　loads(str)

カテゴリ	**jsonモジュール**
引数	**JSON形式の文字列**
戻り値	**Pythonのオブジェクト**
説明	**JSONのテキストデータをPythonのオブジェクトに変換する**
使用例	**json_obj = json.loads(json_str)**

　以下に、JSON形式の文字列「json_str」をPythonのオブジェクトに変換し、for文を使用してキーと値を順に表示する例を示します。

◎リスト loads1.py

```
import json

json_str = """
{
      "名前": "田中一郎",   1
      "年齢": 33,
      "出身地": "山梨"
}
"""

json_obj = json.loads(json_str) 2
for key, value in json_obj.items(): 3
      print(key, value)
```

1でJSONデータを複数行の文字列として記述して、変数json_strに代入しています。ここでは"名前"、"年齢"、"出身地"をキーにする辞書データを用意しています。

2でloads関数によりJSONデータをPythonの文字列に変換し、3のfor文でitemsメソッド（074で解説）によりキーと値を順に取り出して表示しています。

■実行結果

```
名前 田中一郎
年齢 33
出身地 山梨
```

NOTE loads関数でPythonのオブジェクトに変換できない場合は、JSONDecodeError例外が送出されます。

JSONのオブジェクトとPythonのオブジェクトの対応

次の表に、本節で紹介したjsonモジュールの関数における、JSONのオブジェクトと、Pythonのオブジェクトのデフォルトの対応表を示します。

⊙JSONのオブジェクトとPythonのオブジェクトの対応表

JSON	Python
オブジェクト	dict
配列	list
文字列	str
数値(整数)	int
数値(浮動小数点数)	float
true	True
false	False
null	None

JSONのオブジェクトとPythonのオブジェクトの対応は、JSONのデコーダであるjson.JSONDecoderクラス、およびエンコーダであるjson.JSONEncoderクラスのサブクラスを作成することでカスタマイズすることが可能です。

➡114

PythonのオブジェクトをJSONデータに変換するには

dumps関数を使用する

Pythonのオブジェクトを、JSONのテキストデータに変換するには、**dumps**関数を使用します。

◎関数 dumps(obj, sort_keys=False,ensure_ascii=True, indent=None)

カテゴリ	jsonモジュール
引数	obj：Pythonのオブジェクト、sort_keys：dictデータをソートするかどうか ensure_ascii：非ASCII文字をエスケープするか indent：インデントして見やすく表示するかどうか（正の値を指定した場合にはその数のスペースでインデントされる。負もしくは0、""の場合には改行のみ）
戻り値	JSONデータ（文字列）
説明	PythonのオブジェクトをJSON形式の文字列に変換する
使用例	json_str = json.dumps(obj)

以下に、Pythonのオブジェクト「obj」をJSON形式の文字列に変換して表示する例を示します。

◎リスト dumps1.py

```
import json

obj = {"students": [{"name": "田中一郎", "age": 14}, 1
                    {"name": "山田五郎", "age": 15},
                    {"name": "相川花子", "age": 13}
                    ]}

json_str = json.dumps(obj)   2
print(json_str)   3
```

1で"students"をキーに、3人の生徒（名前、年齢）を要素とする配列を定義して変数objに代入しています。2でdumps関数によりJSON文字列に変換し、3でprint関数により表示しています。

■実行結果

```
{"students": [{"name": "\u7530\u4e2d\u4e00\u90ce", "age": 14},
{"name": "\u5c71\u7530\u4e94\u90ce", "age": 15}, {"name":
"\u76f8\u5ddd\u82b1\u5b50", "age": 13}]}
```

●JSONデータをわかりやすく表示する

dumps1.pyの実行結果を見るとわかるように、デフォルトでは、日本語のような非ASCII文字がUnicodeにエンコードされます。これを日本語のまま表示するには、dumps関数の引数に「**ensure_ascii=False**」を指定します。また、引数**indent**を指定してインデントすると、結果をより見やすくできます。

◎リスト dumps2.py

```
import json

obj = {"students": [{"name": "田中一郎", "age": 14},
                    {"name": "山田五郎", "age": 15},
                    {"name": "相川花子", "age": 13}
                    ]}

json_str = json.dumps(obj, ensure_ascii=False, indent=4) 1
print(json_str)
```

1で引数ensure_asciiをFalseに、引数indentを「4」に設定しています。

■実行結果

```
{
    "students": [
        {
```

```
            "name": "田中一郎",
            "age": 14
        },
        {
            "name": "山田五郎",
            "age": 15
        },
        {
            "name": "相川花子",
            "age": 13
        }
    ]
}
```

●辞書データをソートする

dumps関数で、引数**sort_keys**にTrueを指定すると辞書データ（dict）のキーを基準にソートしてくれます。

◎リスト dumps3.py

```
import json

obj = {"ua112":123, "ua395":425, "ua535":512, "ua102":432}

json_str = json.dumps(obj, ensure_ascii=False, indent=4, sort_keys=True) 1
print(json_str)
```

1で「sort_keys=True」を指定してdumps関数を実行しています。結果を見るとわかるように、キーに文字列を指定した場合、文字列として昇順にソートされます。

■実行結果

```
{
```

```
    "ua102": 432,
    "ua112": 123,
    "ua395": 425,
    "ua535": 512
}
```

➡115

JSONデータをファイルに書き出すには

dump関数を使用する

dumps関数の代わりに、**dump**関数を使用すると、Pythonのオブジェクトから変換したJSONデータをテキストファイルに書き出すことができます。

◎ 関 数 dump(obj, fp, sort_keys=False, ensure_ascii=True, indent=None)

カテゴリ	**jsonモジュール**
引数	**obj：Pythonのオブジェクト、fp：ファイルオブジェクト、sort_keys：dictデータをソートするかどうか、ensure_ascii：非ASCII文字をエスケープするか indent：インデントして見やすく表示するかどうか（正の値を指定した場合にはその数のスペースでインデントされる。負もしくは0、""の場合には改行のみ）**
戻り値	**なし**
説明	**PythonのオブジェクトをJSONデータに変換してテキストファイルに書き出す**
使用例	**json.dump(obj, out_file, ensure_ascii=False, indent=4)**

dump関数のファイルオブジェクト以外の引数は、前述のdumps関数と同じです。以下に、「dumps2.py」を変更し、結

果をテキストファイル「students.json」に書き出す例を示します。

◎リスト dump3.py

```
import os
import json

obj = {"students": [{"name": "田中一郎", "age": 14},
                    {"name": "山田五郎", "age": 15},
                    {"name": "相川花子", "age": 13}
                    ]}

dirname = os.path.dirname(__file__)
path = os.path.join(dirname, "students.json")
out_file = open(path, "w", encoding="utf-8")
json.dump(obj, out_file, ensure_ascii=False, indent=4) 1
out_file.close()
```

1でdump関数を実行し、カレントディレクトリのJSONファイル「students.json」にデータを書き出しています。

■実行結果 students.json

```
{
    "students": [
        {
            "name": "田中一郎",
            "age": 14
        },
        {
            "name": "山田五郎",
            "age": 15
        },
        {
            "name": "相川花子",
```

```
            "age": 13
        }
    ]
}
```

コマンドラインでのJSONファイルの処理

json.toolモジュールを使用すると、コマンドラインでJSONファイルを整形したり、検証を行うことが可能です。pythonコマンド（macOSの場合にはpython3コマンド）を次の形式で実行します。

python -m json.tool ファイル

◎リスト sample.json

```
{
"langs": [{"name":"Python","likes": 100},
{"name":"Swift","likes": 80},
  {"name": "JavaScript","likes": 50}]
}
```

たとえば、上記のような整形されていないJSONファイルを引数にして実行すると以下のように整形して表示されます。

◎実行例

```
>python -m json.tool sample.json Enter
{
    "langs": [
        {
            "name": "Python",
            "likes": 100
```

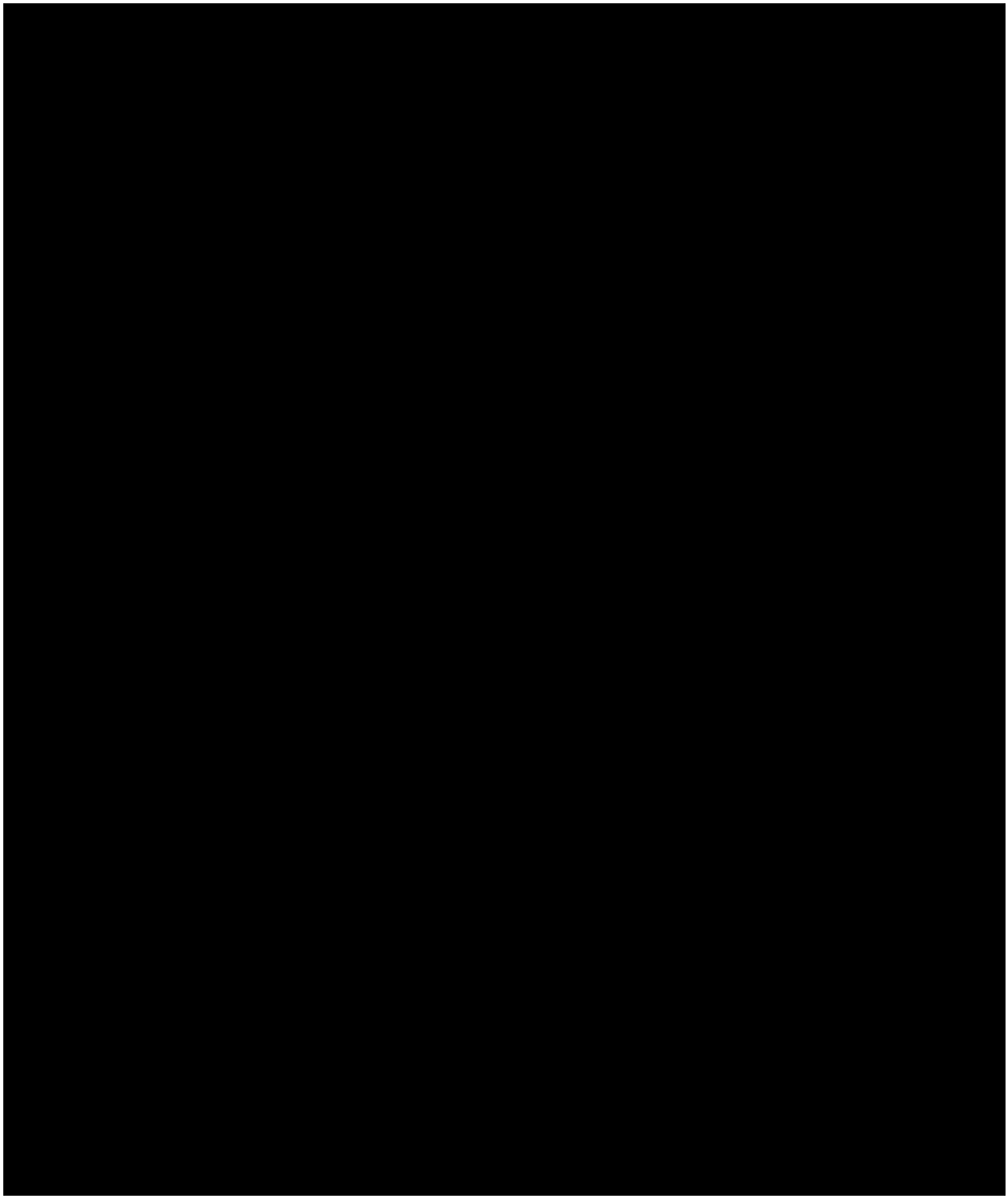
```
        },
～以下略～
    ]
}
```

最後の引数に出力ファイルを指定すると、結果をファイル（下記の例ではnew.json）に書き出すことができます。また、JSONファイルにエラーがある場合はその内容と位置が表示されます。ただし、残念ながら日本語はUnicodeにエンコードされて表示されてしまいます。

◎実行例（ファイルに書き出す）

```
>python -m json.tool sample.json new.json Enter   ←ファイルに書き出す
>python -m json.tool sample.json    Enter
Expecting ',' delimiter: line 2 column 42 (char 43)   ←エラーがある場合
```

Chapter 4



OSの機能を利用する
ーファイル／ディレクトリ／時刻／日付

OSはファイル管理、メモリ管理、入出力管理、プロセス管理といったさまざまな機能を提供します。代表的なOSにはmacOS、Windows、Linuxなどがありますが、本章ではPythonからOSの機能を利用する方法について説明します。特に、OSの機能によって提供される、ディレクトリやファイル、日付／時刻情報に対して操作・アクセスする方法を取り上げます。

4-1 ファイルやディレクトリの取り扱い

Pythonにはパスを記述したり、ディレクトリやファイルを作成、削除したり、名前を変えたりする関数やメソッドが揃っています。ディレクトリやファイルを扱うにはカレントディレクトリ、絶対パス／相対パスなどの知識も必要となります。

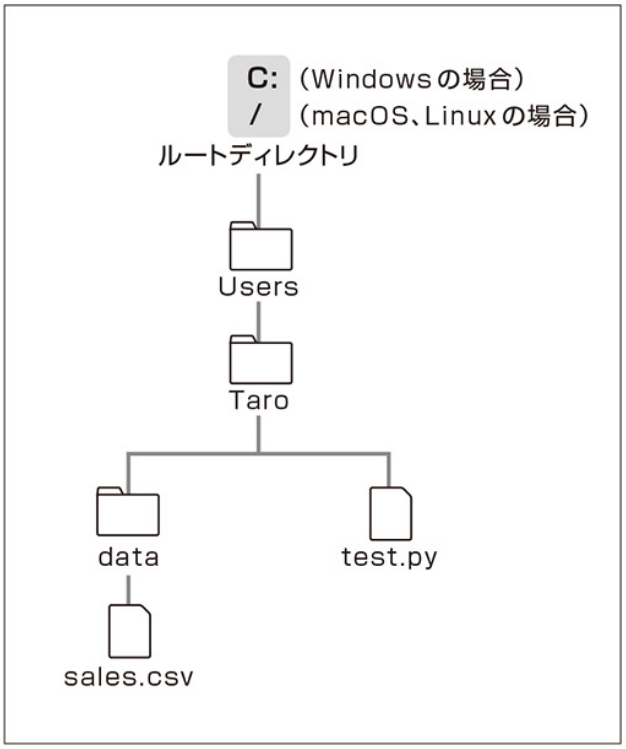
各OSのファイルシステムについて

　OSの提供する機能の1つにファイルシステムがあります。さまざまな「データ」（ファイル）を「**ディレクトリ**」（フォルダ）といった単位でまとめて管理してくれます。Pythonでもファイルの読み書きを行うことができますが、どのファイルかを一意に特定するためには「カレントディレクトリ」や「パス」などの概念を正しく理解しておく必要があります。

　ファイルやディレクトリを特定するために使われるのがパスです。最上位のディレクトリを「**ルートディレクトリ**」といいますが、そのディレクトリからすべてのディレクトリをつないで目的のファイルやディレクトリを指定する方法を「**絶対パス**」と呼びます。

　Windowsでは最初に「C:」や「D:」といったドライブレターに続き、\記号でディレクトリを繋ぎ、ファイルを特定します。macOSやLinuxでは最初の/記号からディレクトリを/で繋ぎファイルを特定します。

⦿ディレクトリ、ファイルの構成例



　たとえば、図のようなディレクトリ構成の場合、test.pyへの絶対パスは以下のようになります。

◎Windowsの絶対パス表記

```
C:\Users\Taro\test.py
```

◎macOSの絶対パス表記

```
/Users/Taro/test.py
```

　一方、特定のディレクトリからの相対位置でファイルやディレクトリを特定する方法もあります。これを「**相対パス**」と呼びます。実行中のプログラムに関連付けられているディレクトリを「**カレントディレクトリ**」といいます。つまり、

**　カレントディレクトリ + 相対パス = 絶対パス**

のような関係となります。

前図の例で、カレントディレクトリがTaroの状態でtest.pyを実行していたとします。このときsales.csvにアクセスするには以下のような相対パスを使用します。

◎Windowsの相対パス表記

```
data\sales.csv
```

◎macOSの相対パス表記

```
data/sales.csv
```

ちなみにログインした直後のカレントディレクトリを「**ホームディレクトリ**」と呼ぶこともあります。

◎OS非依存のディレクトリ区切り文字を使用するには

ディレクトリの区切り文字はWindowsが「￥」、LinuxやmacOSでは「**/**」となります。このような違いを吸収するため、以下のようなプロパティが用意されています。

⊙ディレクトリ区切り文字の違いを解決するためのプロパティ

プロパティ	Windows	Linux/macOS
os.sep	￥￥	/
os.path.sep	￥￥	/

os.sepとos.path.sepがパス中のディレクトリ区切り文字を表します。os.pathsepという似たプロパティがありますが、これは別ものなので注意してください。次項以降で各プロパティを使用した例を示していきます。

➡116

パス名（ディレクトリやファイル）を分割するには

os.pathモジュールのsplit、basename、dirnameを利用する

何らかのファイルへのパスが与えられた場合、ディレクトリ部分とファイル部分に分割したいときがあります。そのようなときには**os.path**モジュールの**split**関数を使用します。

◎関数　split（path）

カテゴリ	**os.pathモジュール**
引数	**パス名**
戻り値	**パスを最後の要素とそれ以外の部分に分割し、それらからなるタプルを返す**
説明	**パスがスラッシュ「/」で終わっているときは最後の部分は空文字列になる**
使用例	**p = os.path.split("/hello/world/python.txt")**

split関数はディレクトリとファイルに分割したタプルを返しますが、ファイル部分だけ必要な場合には**basename**関数を使用します。

◎関数　basename（path）

カテゴリ	**os.pathモジュール**
引数	**パス名**
戻り値	**パスの末尾のファイル名部分を返す**
説明	**split関数の戻り値の2番目の要素に等しい**
使用例	**p = os.path.basename("/hello/world/python.txt")**

basename関数の逆で、ディレクトリ部分だけを取り出すには**dirname**関数を使います。

◎関数　dirname（path）

カテゴリ	**os.pathモジュール**
引数	**パス名**
戻り値	**パスのディレクトリ部分を返す**
説明	**split関数の戻り値の1番目の要素に等しい**
使用例	**p = os.path.dirname("/hello/world/python.txt")**

◎リスト os_path.py

```
import os.path 1

path = "/home/user1/data/sales.csv"
r = os.path.split(path) 2
print("ディレクトリ:{0} ファイル:{1}".format(r[0], r[1])) 3
print("ディレクトリ:{0}".format(os.path.dirname(path))) 4
print("ファイル　　:{0}".format(os.path.basename(path))) 5
```

1でモジュールos.pathを読み込み、**2**～**5**でsplit、dirname、basename関数を実行した結果を表示しています。

■実行結果

```
ディレクトリ:/home/user1/data ファイル:sales.csv
ディレクトリ:/home/user1/data
ファイル　　:sales.csv
```

各関数が適切な値（ディレクトリ名、ファイル名）を返していることがわかります。

➡117

ファイル名と拡張子に分割するには

os.pathモジュールのsplitext関数を利用する

拡張子だけを取り出してファイルの種類に応じた処理を行うときは**splitext**関数を使用します。

◎関数　**splitext（path）**

カテゴリ	**os.pathモジュール**
引数	**パス名**
戻り値	**パスと拡張子からなるタプルを返す**
説明	**パスを拡張子とそれ以外の部分に分割して返す**
使用例	**p = os.path.splitext("/test/temp.txt")**

◎実行例

```
>>> import os.path Enter
>>> os.path.splitext("/home/taro/data/sales.json") Enter
('/home/taro/data/sales', '.json')
```

splitext関数を使って、拡張子以外の部分と拡張子部分からなるタプルが取得できていることを確認できます。

➡118

パス名（ディレクトリやファイル）を連結するには

os.path.join関数もしくはos.sep.join関数を使用する

ディレクトリ名とファイル名を結合する処理を行うために**os.path**モジュールの**join**関数が用意されています。

◎関数　ioin(path1, path2, ...)

カテゴリ	os.pathモジュール
引数	path1：パス名、path2：パス名
戻り値	複数のパスをディレクトリ区切り記号で連結した文字列
説明	パスを連結した文字列を取得する
使用例	p = os.path.join("folder1", "folder2", "test.txt")

　os.path.joinに似た処理は、os.sep（4-1で解説）で区切り文字を取得し、strクラスのjoinメソッド（015で解説）でパスを連結することによっても行えます。ただし、使用には注意が必要です。

◎実行例：Windowsの場合

```
>>> os.sep.join(["dir1", "dir2", "dir3"]) Enter
'dir1\\dir2\\dir3'
>>> os.path.join("dir1", "dir2", "dir3") Enter
'dir1\\dir2\\dir3'
```

◎実行例：macOS/Linuxの場合

```
>>> os.sep.join(["dir1", "dir2", "dir3"]) Enter
'dir1/dir2/dir3'
>>> os.path.join("dir1", "dir2", "dir3") Enter
'dir1/dir2/dir3'
```

　いずれもディレクトリ区切り文字で連結した結果を返しますが、前者は単なる文字列の連結、後者はよりパスの扱いを意識した挙動となります。たとえば、後者は途中で絶対パスが与えられたらそれ以前の内容は破棄します。

◎実行例

```
>>> os.sep.join(("/home/user1", "/home/user2")) Enter
```

```
'/home/user1\/home/user2'
>>> os.path.join("/home/user1", "/home/user2") Enter
'/home/user2'
```

➡119

パスを正規化するには

os.pathモジュールのnormpath関数を使用する

カレントディレクトリは「**.**」（ドット）で、親ディレクトリは「**..**」（ドット2つ）で表現されます。パスを表現するときに、これらの文字が使用されることがありますが、**normpath**関数はこれらの文字を**正規化**（パスの中に「.」や「..」を含まないように変換）してくれます。

◎関数　normpath(path)

カテゴリ	**os.pathモジュール**
引数	**パス**
戻り値	**正規化されたパス**
説明	**「.」や「..」を含んだパスを正規化する**
使用例	**p = os.path.normpath("/A/B/C/../D")**

◎実行例：Windowsの場合

```
>>> os.path.normpath("/A/B/C/../D") Enter
'\\A\\B\\D'
>>> os.path.normpath("/A/./B/./C") Enter
'\\A\\B\\C'
```

◎実行例：macOS/Linuxの場合

```
>>> os.path.normpath("/A/B/C/../D") Enter
```

```
'/A/B/D'
>>> os.path.normpath("/A/./B/./C") Enter
'/A/B/C'
```

NOTE normcase関数は大文字小文字を正規化します。

➡120

ファイルやディレクトリの有無を調べるには

isfile関数、isdir関数、exists関数を状況に応じて使い分ける

ディレクトリやファイルが存在しているか確認するには、**isfile**、**isdir**、**exists**といった関数を使用します。

◎関数　isfile(path)

カテゴリ	**os.pathモジュール**
引数	**調べたいファイルのパス**
戻り値	**パスが実在するファイルであればTrue、それ以外はFalse**
説明	**ファイルの有無を調べる**
使用例	**f = os.path.isfile("1.txt")**

◎関数　isdir(path)

カテゴリ	**os.pathモジュール**
引数	**調べたいディレクトリのパス**
戻り値	**パスが実在するディレクトリであればTrue、それ以外はFalse**
説明	**ディレクトリの有無を調べる**
使用例	**f = os.path.isdir("test")**

◎関数　exists(path)

カテゴリ	os.pathモジュール
引数	調べたいパス
戻り値	パスが実在すればTrueが返る
説明	パスはファイルでもディレクトリでも可
使用例	f = os.path.exists("test")

◎リスト os_path_check.py

```
import os, os.path, pathlib 1

pathlib.Path("test.txt").touch() 2
os.mkdir("testdir") 3

print("testdir    isfile =", os.path.isfile("testdir"))
print("testdir    isdir  =", os.path.isdir("testdir"))
print("test.txt isfile =", os.path.isfile("test.txt"))
print("test.txt isdir  =", os.path.isdir("test.txt"))
print("testdir    exists =", os.path.exists("testdir"))
print("test.txt exists =", os.path.exists("test.txt"))
print("test2       exists =", os.path.exists("test2"))

os.remove("test.txt")
os.rmdir("testdir")
```

1でos、os.path、pathlibの3つのモジュールを読み込んでいます。**2**でpathlib.Pathクラスのtouchメソッド（128で解説）でtest.txtというファイルを作成し、**3**でos.mkdir関数（121で解説）でtestdirというディレクトリを作成しています。あとは、それらのファイルやディレクトリに対してisfile、isdir、exists関数を呼び出しています。最後にremoveメソッド（123）とrmdirメソッド（122）でファイルとディレクトリを削除しています。

■実行結果

```
testdir    isfile = False
testdir    isdir    = True
test.txt isfile = True
test.txt isdir    = False
testdir    exists = True
test.txt exists = True
test2       exists = False
```

ディレクトリ(testdir)に対してはisfileがFalse、isdirがTrue、ファイル(test.txt)に対してはisfileがTrue、isdirがFalseになっていることが確認できます。

➡121

ディレクトリを作成するには

mkdir関数とmkdirs関数を階層に応じて使い分ける

ディレクトリを作成するには**os**モジュールの**mkdir**関数を使用します。

◎関数　mkdir(path)

カテゴリ	**osモジュール**
引数	**作成するディレクトリのパス**
戻り値	**なし**
説明	**引数で指定されたパス名のディレクトリを作成する**
使用例	**os.mkdir("test")**

「test/sample/work」のように一度に複数階層のディレクトリを作成する場合には**makedirs**関数を使用します。

◎関数　**makedirs(path)**

カテゴリ	**osモジュール**
引数	**作成するディレクトリのパス**
戻り値	**なし**
説明	**引数で指定されたパス名のディレクトリを作成する**
使用例	**os.makedirs(os.path.join("dir1", "dir2", "dir3"))**

これらの関数の使用例は、次のレシピで示します。

NOTE os.pathモジュール、osモジュールにはパスを扱う多くの関数がありますが、オブジェクト指向に慣れた人であればpathlibモジュールのPathオブジェクトを使ってもよいかもしれません。パスを作成したり、状態をチェックしたり、ディレクトリを作成したり、テキストとしてファイルを読み込んだりと、いろいろなメソッドが用意されています。
参考：https://docs.python.org/ja/3.6/library/pathlib.html

➡122

ディレクトリを削除するには

rmdir関数、removedirs関数、rmtree関数を使い分ける

ディレクトリを削除する場合は、**rmdir**や**removedirs**関数を使用します。

◎関数　**rmdir(path)**

カテゴリ	**osモジュール**
引数	**削除対象となるディレクトリ**
戻り値	**なし**

説明	ディレクトリ path を削除する。ディレクトリが空の場合だけ正常に動作する
使用例	os.rmdir("test")

◎関数　removedirs(path)

カテゴリ	osモジュール
引数	削除対象となるディレクトリ
戻り値	なし
説明	ディレクトリ path を再帰的に削除する
使用例	os.removedirs(os.path.join("dir1", "dir2", "dir3"))

◎リスト os_folders.py

```
import os

os.mkdir("temp") 1
os.rmdir("temp") 2

p = os.sep.join(["dir1", "dir2", "dir3"]) 3
os.makedirs(p) 4
os.removedirs(p) 5
```

1でディレクトリtempを作成し、2でそれを削除しています。3でdir1/dir2/dir3という深い階層を示すパスの文字列を作成し、4でそのディレクトリを作成し、5のremovedirs関数でそのディレクトリを削除しています。1階層ずつmkdir、rmdirでディレクトリを作成・削除する必要はありません。

ただし、rmdir関数、removedirs関数はディレクトリが空でないとエラーになります。ディレクトリが空でない場合でも強制的にディレクトリを削除する場合には、shutilモジュールのrmtree関数を使用します。

●ディレクトリごと削除するrmtree関数

◎関数　rmtree(path [,onerror=None])

カテゴリ	shutilモジュール
引数	削除対象のディレクトリ
戻り値	なし
説明	ディレクトリツリー全体を削除する
使用例	shutil.rmtree("A")

◎実行例

```
>>> import shutil Enter
>>> shutil.rmtree("dir1") Enter
```

◎リスト shutil_move.py

```
import os, shutil, pathlib

pathlib.Path("tmp.txt").touch()
dst = os.sep.join(["A","B","C"])
os.makedirs(dst)
shutil.move("tmp.txt", dst)
shutil.rmtree("A")
```

Pathクラスを使って「tmp.txt」というファイルを作成します。os.sep.joinでパスを作り、makedirsでディレクトリ階層を作ります。shutil.moveでそのディレクトリの最下層にtmp.txtを移動し、shutil.rmtreeでディレクトリ階層ごと削除しています。

rmtreeのonerror引数に関数を指定するとエラー時に通知を受け取ることができます。関数は、function、パス、例外の3つの引数を受け取る必要があります。

◎実行例

```
>>> shutil.rmtree("dummy", onerror=lambda x, y, z: print(z))
Enter
```

```
(<class 'FileNotFoundError'>, FileNotFoundError(2, '指定されたパスが見つかりません。'), <traceback object at 0x0000025769ACD788>)
(<class 'FileNotFoundError'>, FileNotFoundError(2, '指定されたファイルが見つかりません。'), <traceback object at 0x0000025769ACD788>)
```

存在しない"dummy"というパスを指定してrmtreeを実行しています。onerrorでは引数を3つとるlambda式を指定し、3番目の引数である例外をprintしていますが、指定されたパスが見つからない旨の例外が投げられていることがわかります。

➡123

ファイルを削除するには

osモジュールのremove関数を利用する

ファイルを削除するにはosモジュールの**remove**関数を使用します。

◎関数　remove(path)

カテゴリ	**osモジュール**
引数	**削除対象となるファイル**
戻り値	**なし**
説明	**引数で指定されたファイルを削除する**
使用例	**os.remove("test.txt")**

◎リスト file_touch_remove.py

```
import os, pathlib, tempfile 1
```

```
pathlib.Path("test.txt").touch() 2
os.remove("test.txt") 3
fd, temp_path = tempfile.mkstemp(prefix="hello") 4
os.close(fd) 5
os.remove(temp_path) 6
```

1でモジュールを読み込み、2でtest.txtというファイルを作成し、3でそのファイルを削除しています。4でテンポラリのファイルを作成し、5でそのファイルを閉じて、6でファイルを削除しています。tempfile.mkstemp関数については、レシピ129で取り上げています。

remove関数の引数には*などのワイルドカードを指定することはできません。ワイルドカードを使って複数のファイルを削除したい場合はglobモジュール（後述）を使って、以下のようにしてもよいでしょう。

◎実行例

```
>>> import glob Enter
>>> [os.remove(x) for x in glob.glob("test/*")] Enter
[None, None, None]
```

globモジュールを読み込み、リスト内包表記を使って複数のファイルを処理しています。globモジュールのglob関数で該当するファイルを列挙し、各ファイルに対してos.removeを呼び出しています。

➡124
作業ディレクトリを取得・移動するには

getcwd関数を使用する

ファイルにアクセスする場合、ルートからの絶対パス、もしくは、**作業ディレクトリ**からの相対パスのどちらかを使用します。相対パスを使う場合は、作業ディレクトリがどこか把握すること、また、その作業ディレクトリを移動することが重要になります。作業ディレクトリは「**カレントディレクトリ**」とも言われ、プログラム実行時の拠点となるディレクトリです。

◎関数　getcwd()

カテゴリ	**osモジュール**
引数	**なし**
戻り値	**現在の作業ディレクトリを表す文字列**
説明	**現在の作業ディレクトリを返す**
使用例	**p = os.getcwd()**

現在作業中のディレクトリを移動するには**chdir**関数を使用します。

◎メソッド　chdir(path)

カテゴリ	**osモジュール**
引数	**移動先のディレクトリへのパス**
戻り値	**なし**
説明	**作業ディレクトリを引数で指定されたパスへ変更する**
使用例	**os.chdir("test/sample")**

◎リスト os_dir.py

```
import os
p = os.getcwd() 1
print("before:"+p)

os.mkdir("test") 2
os.chdir("test") 3
```

```
p = os.getcwd() 4
print("after :"+p)
```

1で現在の作業ディレクトリを取得してprintで出力しています。2でtestディレクトリを作業ディレクトリの下に作成し、3で作業ディレクトリをtestに移動しています。4で移動後の作業ディレクトリを取得して出力しています。

■実行結果：Windowsの場合

```
before:c:\Users\Taro\Samples
after :c:\Users\Taro\Samples\test
```

■実行結果：macOS/Linuxの場合

```
before:/home/kenichiro/Samples
after :/home/kenichiro/Samples/test
```

➡125

ディレクトリのファイル一覧を取得するには

listdir関数を使用する

ディレクトリにあるファイル一覧を取得するには**listdir**関数を使用します。

◎関数　listdir(path)

カテゴリ	**osモジュール**
引数	**ディレクトリへのパス。省略時はカレントディレクトリ**
戻り値	**ディレクトリ内のファイルやディレクトリを含むリスト**

説明	ディレクトリ内のファイル一覧を返す。自分自身を意味する「.」、親ディレクトリを意味する「..」は一覧には含まれない
使用例	files = os.listdir()

listdir関数は、指定したディレクトリに含まれる、ファイルおよびディレクトリの名前を要素とするリストを戻します。

◎実行例

```
>>> import os Enter
>>> os.listdir("data") Enter
['sales1.csv', 'sales2.csv']
```

walk関数を使用すると、ディレクトリ、ファイル一覧を再帰的に取得できます。

◎リスト os_walk.py

```
import os
for root, dirs, files in os.walk('.'):
        print("root={0}".format(root))
        print("    dirs={0}".format(dirs))
        print(" files={0}".format(files))
```

1行目でosモジュールを読み込みos.walk関数で再帰的にファイルとディレクトリを取得しています。sampleディレクトリの下にworkディレクトリとtest.txtを、workディレクトリの下にsales.csvがあったときの出力を以下に示します。

■出力例：Windowsの場合

```
root=.
    dirs=['sample']
files=['os_walk.py']
```

```
root=.\sample          ←macOS/Linuxでは「root=./sample」
   dirs=['work']
files=['test.txt']
root=.\sample\work                    ←macOS/Linux で は
「root=./sample/work」
   dirs=[]
files=['sales.csv']
```

●ディレクトリのファイル一覧をイテレート可能なオブジェクトとして取得する

listdir関数はファイルやディレクトリのリストを返しました。イテレート可能なオブジェクトを返す**scandir**関数を使ってもディレクトリ内のファイルやディレクトリを列挙することができます。scandirはバージョン3.5から標準ライブラリに追加されました。

◎関数　scandir(path)

カテゴリ	**osモジュール**
引数	**ディレクトリへのパス。省略時はカレントディレクトリ**
戻り値	**ディレクトリ内のファイルやディレクトリをイテレート可能なオブジェクトとして返す**
説明	**ディレクトリ内のファイル一覧を得るためのイテレート可能なオブジェクトを返す**
使用例	**files = list(os.scandir())**

filesにはファイルやディレクトリの情報を含むDirEntryのリストが返されます。listdirよりも処理速度が速く、より多くの情報が取得できます。

➡126

条件に合致するファイルをリストアップするには

globモジュールのglob関数を使用する

前述のlistdir関数やscandir関数では特定のディレクトリに含まれるファイルやディレクトリを取得できますが、**glob**モジュールの**glob**関数では条件に合致したファイルのみを取得できます。任意の文字に合致する「？」、任意の文字列に合致する「＊」などのパターンを使用することができます。

◎関数　glob(pathname)

カテゴリ	**globモジュール**
引数	**検索対象となる条件を含むパス**
戻り値	**条件に合致したパス名からなるリスト**
説明	**特定のパターンにマッチしたパス名のリストを返す**
使用例	**files = glob.glob("test/*")**

仮に「1.txt」「1.png」「py.png」というファイルを含むディレクトリがあったとします。

◎リスト globe_globe.py

```
import glob

a = glob.glob('[0-9].*') 1
print(a)
b = glob.glob('*.png') 2
print(b)
c = glob.glob("?.png") 3
print(c)
```

1の「[0-9].*」はファイル名が0～9で拡張子が任意のファイルの指定です。「1.png」と「1.txt」が合致します。

2の「*.png」はファイル名が任意で拡張子が「png」のファイルの指定です。「1.png」と「py.png」が合致します。
3の「?.png」は、ファイル名が1文字で拡張子が「png」のファイルの指定です。「1.png」だけが合致します。

■実行結果

```
['1.png', '1.txt']
['1.png', 'py.png']
['1.png']
```

globメソッドの引数にrecursive=Trueを指定した場合、パス引数に含まれる「**」はあらゆるディレクトリにマッチして、再帰的に探すようになります。たとえば、次のような命令を実行すると、Windowsの場合はCドライブのUsers以下にあるすべてのpngファイル、macOS/Linuxの場合は/home/kenichiro以下にあるすべてのpngファイルを探しにいきます。ファイル数が多い場合は時間がかかるので注意してください。

◎実行例：Windowsの場合

```
>>> glob.glob("c:/Users/**/*.png", recursive=True) Enter
```

◎実行例：Linuxの場合（macOSの場合は"/Users/kenichiro//*.png"）**

```
>>> glob.glob("/home/kenichiro/**/*.png", recursive=True) Enter
```

➡127
ファイルやディレクトリの名前を変えるには

osモジュールのrename関数、shutilモジュールのmove関数を使用する

ファイルやディレクトリの名前を変える場合には、**os**モジュールの**rename**関数もしくは**shutil**モジュールの**move**関数を使用します。

◎関数　rename(src, dst)

カテゴリ	**osモジュール**
引数	**src：元のファイルもしくはディレクトリ名、dst：変更後のファイルもしくはディレクトリ名**
戻り値	**なし**
説明	**ファイルもしくはディレクトリの名前を変更する**
使用例	**os.rename("test.txt", "temp.txt")**

◎実行例

```
>>> os.listdir("data") Enter  1
['test1.txt']
>>> os.rename("data/test1.txt", "data/test2.txt") Enter  2
>>> os.listdir("data") Enter  3
['test2.txt']
```

1の「os.listdir("data")」でディレクトリdataの中に「test1.txt」が1つだけある状態であることを確認しています。その後**2**でrename関数を使って「test1.txt」を「test2.txt」に名前を変更し、最後にos.listdir関数で名前が変わったことを確認しています。

◎関数　move(src, dst)

カテゴリ	**shutilモジュール**
引数	**src：元ファイルやディレクトリ、dst：移動先のパス**
戻り値	**移動先のパス**

説明	移動先がファイル名の場合は、ファイル名が変更される。移動先がディレクトリの場合は、そのディレクトリへ移動する
使用例	p = shutil.move("test1.txt", "test2.txt")

os.renameとshutil.moveはどちらもファイル名の変更や移動を行うのでよく似ています。実際に、shutil.moveは、その内部でos.renameを呼び出しています。shutil.moveは、os.renameが失敗した場合に、ディレクトリごとコピーを行い、元のファイルを削除するよう動作します。

また、os.renameはdstが既存のディレクトリであるときにエラーが起きますが、shutil.moveはdstが既存のディレクトリのときには、そのディレクトリの下にファイルが移動するといった違いがあります。

➡128

空のファイルを作成するには

pathlibモジュールにあるPathクラスのtouchメソッドを使用する

open関数を使うとファイルを作成することができますが、ここでは別の方法でファイルを作成してみます。

◎メソッド touch()

カテゴリ	pathlibモジュールのPathクラス
引数・戻り値	なし
説明	macOSやLinuxなどのtouchコマンドは空のファイルを作成するが、それと同じ働きをする。まずPathオブジェクトを作成し、そのオブジェクトのtouchメソッドを呼ぶことで空のファイルを作成する

使用例	pathlib.Path("test.txt").touch()

◎実行例

```
>>> import pathlib Enter
>>> pathlib.Path("test.txt").touch() Enter
```

上記の例では「test.txt」という空のファイルが作成されます。

NOTE Pathクラスはバージョン3.4で追加されました。

➡129

一時ファイルを作成するには

tempfileモジュールのmkstemp関数を使う

一時ファイル（テンポラリファイル）を作る場合は、自分でファイル名を考えるよりも一時ファイルを作る関数を使ったほうがよいでしょう。

◎関数 **mkstemp (suffix=None, prefix=None)**

カテゴリ	tempfileモジュール
引数	prefix：接頭辞を指定したい場合に使用、suffix：接尾辞を指定したい場合に指定
戻り値	ファイルハンドルとファイルの絶対パスからなるタプル
説明	一時ファイルを作成する。作成したファイルは自分で削除する必要がある。一時ディレクトリを作成する場合にはmkdtemp関数を使用する
使用例	f = tempfile.mkstemp() f = tempfile.mkstemp(prefix="hello")

ＮＯＴＥ 一時ファイルを作るmktempという関数もありましたが、セキュリティホールになる可能性が見つかったため使用が非推奨となっています。

◎実行例：Windowsの場合

```
>>> import tempfile Enter
>>> tempfile.mkstemp() Enter
(6, 'C:\\Users\\KENICH~1\\AppData\\Local\\Temp\\tmp_nafp99_')
>>> tempfile.mkstemp(prefix="python", suffix="test") Enter
(7,
'C:\\Users\\KENICH~1\\AppData\\Local\\Temp\\pythond7k9l3g3te
st')
```

◎実行例：macOS/Linuxの場合

```
>>> import tempfile Enter
>>> tempfile.mkstemp() Enter
(4, '/tmp/tmp4ppk2gqf')
>>> tempfile.mkstemp(prefix="python", suffix="test") Enter
(5, '/tmp/python_03edx40test')
```

tempfileモジュールのTemporaryFileを使用すると、ファイルを閉じると**自動で削除**されます。

◎リスト temp_TemporaryFile.py

```
import tempfile
with tempfile.TemporaryFile(mode='w+t') as f: 1
    f.write("Hello Python")
    f.seek(0)
    print(f.read())
```

1のTemporaryFile関数でファイルを作成し、以下、writeで文字列を書き込み、seekで位置を先頭に戻し、readで読み出

しています。

■出力例

```
Hello Python
```

NOTE TemporaryFileはデフォルトで'w+b'モードで作成されるので、テキストを書き込む場合は**明示的にモードを指定**する必要があります。

4-2 多少高度なファイル操作

osモジュールやos.pathモジュールを使うと基本的なファイル処理を行うことができますが、コピーや削除などよく行われる操作、圧縮/展開などの高度な操作に関しては別のモジュールが用意されています。ここではそのようなモジュールについて説明します。

➡130

ファイルをコピーするには

shutilにあるコピー用の関数を利用する

shutilにはコピーをするための関数がいくつか用意されています。主なものを見てみましょう。

◎関数　copy(src, dst)

カテゴリ	**shutilモジュール**
引数	**src：コピー元ファイル、dst：コピー先ファイル、もしくは書き込み可能な既存のディレクトリ**
戻り値	**新しく作成されたファイルのパスを返す**
説明	**ファイルsrcをファイルdstとして、もしくはファイルsrcをディレクトリdstの下にコピーする**
使用例	**f = shutil.copy("test1.txt", "samples")**

「shutil.copy("test1.txt", "samples")」と実行したとします。「samples」というディレクトリがあると、その下に「test1.txt」という同名のファイルでコピーが作成されます。ディレクトリがない場合、ファイル名が「samples」となります。

◎関数　copyfile(src, dst)

カテゴリ	shutilモジュール
引数	src：コピー元ファイル、dst：コピー先ファイル
戻り値	新しく作成されたファイルのパスを返す
説明	ファイルsrcをファイルdstという名前でコピーする。dstがディレクトリだとエラーになる
使用例	f = shutil.copyfile("test1.txt", "test2.txt")

　copyはディレクトリの下にコピーできますが、copyfileはディレクトリの下にコピーができません。

◎リスト shutil_copy.py

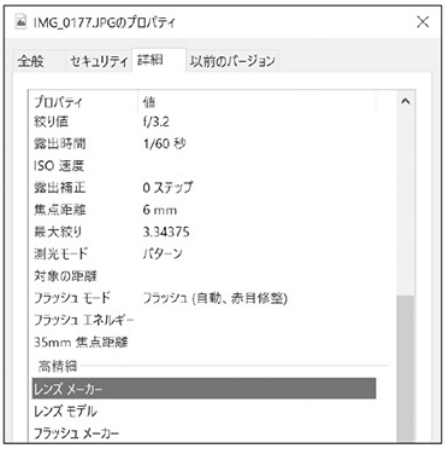
```
import shutil
shutil.copyfile("1.txt", "2.txt") 1
shutil.copy("2.txt", "2.copy.txt") 2
shutil.copy("2.txt", "tmp") 3
```

1で「1.txt」ファイルを「2.txt」にコピーしています。**2**で「2.txt」を「2.copy.txt」にコピーし、**3**で「2.txt」を「tmp」ディレクトリの下にコピーしています。

メタデータ

　ファイルにはテキストや画像などのデータの実体だけでなく、ファイル作成日時、所有者などさまざまな付属情報が付与されています。このような付属情報は「メタデータ」と呼ばれます。たとえば、Windowsで写真のファイルを選択してプロパティを見ると、図のような画面が表示されます。

⦿写真画像のプロパティ（Windows）



画像本体だけではなく、撮影時のカメラのパラメータ情報まで含まれていることがわかります。GPS機能付きのカメラの場合は、撮影場所まで特定できることがあるので、メタデータの取り扱いには注意が必要です。shutilモジュールにはファイルをコピーするためにcopyとcopy2という2つの関数が用意されています。copyは単なるコピー、copy2はメタデータも含めたコピーとなります。しかしながらメタデータの扱いはOSによって異なるため、どのメタデータがコピーされるかは実際に試して確認することをお勧めします。

ライブラリのソースコードを読む

shutilモジュールにはcopyfileobj、copy、copy2、copyfile、copymode、copystat、copytreeのようにコピー関連の関数が多数用意されています。copymodeはパーミッション情報（実行可否・読み取り可否・書き込み可否）もコピーします。リファレンスを見ても違いがわかりづらいか

もしれません。そのようなときはソースコードを直接見てみるのも1つの方法です。

◎ https://github.com/python/cpython/blob/3.6/Lib/shutil.pyより一部抜粋

```
def copyfileobj(fsrc, fdst, length=16*1024):
    while 1:
        buf = fsrc.read(length)
        if not buf:
            break
        fdst.write(buf)

def copyfile(src, dst, *, follow_symlinks=True):
...
        with open(src, 'rb') as fsrc:
            with open(dst, 'wb') as fdst:
                copyfileobj(fsrc, fdst)

def copy(src, dst, *, follow_symlinks=True):
...
    copyfile(src, dst, follow_symlinks=follow_symlinks)
    copymode(src, dst, follow_symlinks=follow_symlinks)
    return dst

def copy2(src, dst, *, follow_symlinks=True):
...
    copyfile(src, dst, follow_symlinks=follow_symlinks)
    copystat(src, dst, follow_symlinks=follow_symlinks)
    return dst
```

・copyfileobjはread、writeを使ってコピーしている
・copyfileは単にcopyfileobjを呼び出してコピーを作成している

・copyはcopyfileでコピーを作成し、copymodeを呼び出している
・copy2はcopyfileでコピーを作成し、copystatを呼び出している
といったことがわかります。ライブラリは良質なコードの宝庫です。shutilモジュールだけでなく、いろいろなモジュールの中をのぞいてみてください。

➡131
コマンドのパスを調べるには


shutilモジュールのwhich関数を使用する

コマンドプロンプトやターミナルを開いて、コマンドを入力している人であれば、コマンドがどこにあるか調べたことがあると思います。macOSやLinuxなどであればwhichコマンド、Windowsであればwhereコマンドが使えます。Pythonでもプログラムから調べることが可能です。

◎関数　which(cmd)

カテゴリ	**shutilモジュール**
引数	**実行コマンド**
戻り値	**実行コマンドへのパス。実行コマンドが見つからない場合はNone**
説明	**実行コマンドへのパスを返す**
使用例	**p = shutil.which("python")**

コマンドは環境変数**PATH**で指定されるディレクトリを順に辿ることによって探索を行います（ただしWindowsはカレントディレクトリが優先されます）。現在の環境変数PATHの内

容は**os**モジュールの**environ**オブジェクトから調べることができます。

◎実行例：Windowsの場合

```
>>> import shutil, os Enter
>>> shutil.which("notepad") Enter
'C:\\WINDOWS\\system32\\notepad.EXE'
>>> os.environ.get("PATH", os.defpath) Enter
'c:\\Anaconda3\\Library\\bin;C:\\Perl64\\site\\bin;C:\\Perl64\\bin;C:
\\WINDOWS\\system32;C:\\WINDOWS;C:\\WINDOWS\\System32\\
Wbem ～以下略～ ;C:\\Users\\kenichiro\\AppData\\Roaming\\npm'
```

Windows のメモ帳アプリ（notepad）がC:\Windows\system32\notepad.EXEにあることがわかります。

◎実行例：Linuxの場合

```
>>> import shutil, os Enter
>>> shutil.which("ls") Enter
'/bin/ls'
>>> os.environ.get("PATH", os.defpath) Enter
'/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/gam
es:/usr/local/games'
```

➡132

ファイルを圧縮／展開するには

shutilモジュールのmake_archive関数、unpack_archive関数を使用する

複数のファイルをまとめたり、ファイルサイズを小さくするために圧縮をしたり、**zipファイル**を展開してファイルを取

り出したりしたことがあると思います。**shutil**モジュールの**make_archive**関数や**unpack_archive**関数を使うと、プログラムから圧縮、展開といった操作を行うことができます。

◎関数　make_archive(basename, format, root_dir)

カテゴリ	shutilモジュール
引数	basename：作成するアーカイブファイルの名前 format：zip、tar、gztar、bztar、xztarのいずれか root_dir：アーカイブのルートとなるディレクトリ
戻り値	アーカイブされたファイル名
説明	root_dirにあるファイルをformat形式でアーカイブしてbasenameのファイル名で保存する
使用例	f = shutil.make_archive("archive_test", "zip", "archive_target")

◎ 関 数　unpack_archive(filename, extract_dir, format)

カテゴリ	shutilモジュール
引数	filename：展開するアーカイブファイルの名前 extract_dir：アンパックする先のディレクトリ、省略時は現在の作業ディレクトリformat：zip、tar、gztar、bztar、xztarのいずれか。省略時は拡張子から判断
戻り値	なし
説明	filenameを展開してextract_dirに保存する
使用例	shutil.unpack_archive("archive_test.zip", "unpack_target","zip")

◎リスト shutil_zipdownload.py

```
import shutil, urllib.request
zipfile = "sample_kisoPython.zip"
url = ("https://book.impress.co.jp/support/aftercare/"
      + "download/" + zipfile)
urllib.request.urlretrieve(url, zipfile)
```

```
shutil.unpack_archive(zipfile, "unarchive", "zip")
```

urllib.request.urlretrieveでネットからzipファイルを取得してダウンロードしています。そのファイルをshutil.unpack_archive関数で展開しています。

●zip形式でファイルを圧縮／展開するには

zipfileモジュールには、zip形式の圧縮／展開を扱うための専用の関数が用意されています。

◎コンストラクタ　ZipFile(path, mode)

カテゴリ	zipfileモジュール
引数	path：アーカイブファイルのパス
	mode：ファイルを開くときのモード。「r」でread（読み込み）、「w」でwrite（書き込み）。「r」がデフォルト
戻り値	ZipFileオブジェクト
説明	指定されたファイル名のzipアーカイブを作成する
使用例	z = zipfile.ZipFile("test.zip", "w")

◎メソッド　write(path)

カテゴリ	zipfileモジュールのZipFileクラス
引数	アーカイブに追加するファイル
戻り値	なし
説明	zipアーカイブにファイルを追加する
使用例	z = zipfile.ZipFile("test.zip", "w")
	z.write("1.txt")

◎メソッド　namelist()

カテゴリ	zipfileモジュールのZipFileクラス
引数	なし
戻り値	アーカイブに含まれるファイルのリスト
説明	zipアーカイブに含まれるファイルをリストとして返す
使用例	z = zipfile.ZipFile("test.zip", "w")

```
f = z.namelist()
```

◎メソッド　read(filename)

カテゴリ	zipfileモジュールのZipFileクラス
引数	zipアーカイブから取り出すファイルの名前
戻り値	取り出すファイルの内容
説明	zipアーカイブから指定されたファイルを取り出す
使用例	z = zipfile.ZipFile("test.zip", "r") b = z.read("1.txt")

◎リスト zipfile_archive.py

```
import zipfile
with zipfile.ZipFile("hello.zip", "w") as zf:    1
    for f in ["readme.txt", "helloworld.txt"]: 2
        zf.write(f)
```

1で「hello.zip」というzipファイルを作成し、**2**でその中に「readme.txt」と「helloworld.txt」という2つのファイルを追加しています。

◎リスト zipfile_extract.py

```
import zipfile
with zipfile.ZipFile("hello.zip") as zf:
    for f in zf.namelist():
        data = zf.read(f)
        print("{0}:{1}".format(f, data))
```

「hello.zip」というzipファイルからファイル一覧をnamelistメソッドで取り出し、それらのファイルの内容をreadメソッドで呼び出しています。

■実行結果

```
readme.txt:b'README'
helloworld.txt:b'HELLOWORLD'   ← ' ' 内にファイルの内容が表示される
```

ZipFileオブジェクトを作成した場合、オブジェクトを使用したあとcloseメソッドを呼ぶようにしてください。withを使って作成した場合はcloseメソッドが自動的に呼ばれるのでその必要はありません。

➡133

複数ファイルから読み込むには

fileinputモジュールのinput関数を使用する

複数のファイルを読み込んで処理する場合には**fileinput**モジュールが便利です。**input**関数にファイルのリストを与えると、それらのファイルから1行ずつ読み込みます。

◎関数　input(files)

カテゴリ	**fileinputモジュール**
引数	**ファイルリスト**
戻り値	**FileInputクラスのオブジェクト**
説明	**引数で与えられたファイルを順番に読み込む。引数を省略したときは標準入力から読み込む**
使用例	**f = fileinput.input(["1.txt", "2.txt"])**

◎リスト fileinput_input.py

```
import fileinput
import glob
files = glob.glob("archive_target/*") 1
for line in fileinput.input(files): 2
```

```
    print(line)
```

1でファイルのリストを取得しています。**2**でそれらのファイルから順番に１行ずつ読み込みます。

なお、ディレクトリにあるファイルの一覧を取得する場合、os.listdir関数も利用できますが、glob.glob関数と戻り値が異なることに注意してください。glob.glob関数の場合はディレクトリ名も戻り値に含まれますが、listdir関数の場合は単にファイル名が列挙されるだけです。

◎実行例

```
>>> glob.glob("archive_target/*") Enter
['archive_target\\test1.txt', 'archive_target\\test2.txt',
'archive_target\\test3.txt', 'archive_target\\test4.txt']
>>> os.listdir("archive_target") Enter
['test1.txt', 'test2.txt', 'test3.txt', 'test4.txt']
```

今回のサンプルでは、fileinput.input関数がファイルを開くためのパス名を渡す必要があったので、glob.glob関数を利用しました。状況に応じて使い分けてください。

➡134
オブジェクトをファイルに保存する／ファイルから読み込むには

pickleモジュールを使ってオブジェクトを保存する

pickleとは漬物という意味です。このモジュールは「オブジェクトをファイルに保存し、あとで取り出して利用できるようにするためのもの」です。オブジェクトを保存すること

を「**Serialize**（シリアライズ）」、復元することを「**DeSerialize**（デシリアライズ）」と言います。漬物にして保存して、食べるとき（使うとき）に取り出すイメージです。

◎関数　dump(obj, file)

カテゴリ	pickleモジュール
引数	obj：ファイルに保存するオブジェクト file：保存する対象となるファイルのファイルオブジェクト
戻り値	なし
説明	オブジェクトをファイルに保存する。ファイルはバイナリの書き込みモードで開いている必要がある
使用例	pickle.dump(mydict, fileObject)

◎関数　load(file)

カテゴリ	pickleモジュール
引数	pickleで保存されたファイルのファイルオブジェクト
戻り値	復元されたオブジェクト階層を返す
説明	ファイルからオブジェクトを復元する。ファイルはバイナリの読み込みモードで開いておく必要がある
使用例	dataObj = pickle.load(fileObject)

リスト　pickle_test.py

```
import pickle
mydict = {"apple":3, "orange":2}
with open("pickle.dat", mode="wb") as f:
    pickle.dump(mydict, f)
```

　pickleが保存するデータはバイナリ形式なのでテキストエディタで中身を見ることはできません。単純なデータであれば、コマンドラインから確認することができます。

◎実行例

```
>python pickle_test.py  Enter
>python -m pickle pickle.dat Enter
{'apple': 3, 'orange': 2}
```

　データの詳しい情報を取得するには以下のようにpickletoolsを使用します。

◎実行例

```
>python -m pickletools pickle.dat Enter
    0: \x80 PROTO      3
    2: }    EMPTY_DICT
    3: q    BINPUT     0
    5: (    MARK
    6: X        BINUNICODE 'apple'
   16: q        BINPUT     1
   18: K        BININT1    3
   20: X        BINUNICODE 'orange'
   31: q        BINPUT     2
   33: K        BININT1    2
   35: u        SETITEMS   (MARK at 5)
   36: .    STOP
highest protocol among opcodes = 2
```

　以下、pickleモジュールの関数の使用例を示します。

◎リスト pickle_basic.py

```
import pickle
import sys

class Person: 1
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
if len(sys.argv) > 1 and sys.argv[1] == "r": 2
    with open('pickle.dat', mode='rb') as f:    3
        obj = pickle.load(f) 4
        print("name={0} age={1}".format(obj.name, obj.age))
else:
    with open('pickle.dat', mode='wb') as f:    5
        obj = Person("Taro", 12) 6
        pickle.dump(obj, f)    7
```

1でPersonクラスを定義しています。このクラスにはnameとageという2つのフィールドがあります。

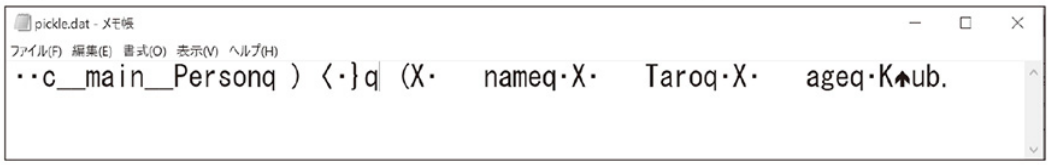
2でコマンド引数を調べて、rという文字が付与されていたら、3でpickle.datファイルをバイナリ読み込みモードで開きます。4でファイルオブジェクトfを引数としてpickle.loadを呼び出してオブジェクトを復元します。復元したオブジェクトはPerson型なので、print文でその内容を出力します。

コマンド引数がrでなかった場合、5でファイルをバイナリ書き込みモードで開きます。6でPersonオブジェクトを作成し、7でpickle.datファイルに保存します。

バイナリモードとテキストモード

pickleで保存されたファイルをWindowsのメモ帳やmacOSのテキストエディットで開いてみましょう。よくわからない暗号のような文字が表示されるはずです。

⦿pickleで保存されたファイルの内容（Windowsの場合）



コンピュータはすべての情報を0と1の列として扱います。ファイルの中に含まれているデータも例外ではありません。人間が読み書きする文字にはあらかじめコードが割り当てられています。たとえば、「@」は「0x40」、「A」は「0x41」、「P」は「0x50」、「Q」は0x51という具合です。「Hello」という文字をテキストエディタで保存すると、実際には「0x48、0x65、0x6c、0x6c、0x6f」というデータが保存されています。バイナリエディタという特別なエディタでファイルを開くとその様子がよくわかります。

⦿バイナリエディタ（Windowsの場合）

しかし、このような数値を直接扱うのは人間にとっては直感的ではありません。実は、テキストエディタは人間がわかりやすい文字にデータを変換しているのです。pickleで保存されているデータはバイナリ形式です。人間が読んだり直接編集したりすることは想定していないため、テキストエディタで開いたときには、人間が読めない状態になっていたのです。このように、テキストを読み書きする場合はテキストモード、数値データを直接扱う場合はバイナリモードと使い分ける必要があるのです。

ちなみに以下のコードを実行すると、文字コードと実際の文字の対応表が出力されます。

リスト ascii_code.py

```
for c in range(0, 127):
    if c % 16 == 0:
        print("{0:02x}".format(c), end="")
```

```
    fmt = "{0:3x}" if c < 16 else "{0:3c}"
    print(fmt.format(c), end="")
    if c % 16 == 15:
        print("")
```

■実行結果

```
00  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
10 Enter
20     !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
30  0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
40  @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
50  P  Q  R  S  T  U  V  W  X  Y  Z  [  <ct:>  ]  ^  _
60  `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
70  p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

4-3 timeモジュールによる時刻の取り扱い

時刻はあまりにも身近なので、プログラミングでの扱いも簡単だと思われるかもしれませんが、名前のよく似たいろいろなモジュールやクラスがあるので混乱しがちです。本節では時刻を扱うモジュールについて説明します。

➡135

現在時刻を取得するには

timeモジュールの各種関数を利用する

エポックというのは時刻の起点のことです。time.gmtime(0)と実行すると、そのプラットフォームでの起点となる時刻を調べることができます。

◎関数　time()

カテゴリ	**timeモジュール**
引数	**なし**
戻り値	**エポックからの秒数を浮動小数点数で返す**
説明	**一般的なプラットフォームでは1970年の1月1日0時0分0秒が起点となる**
使用例	**t = time.time()**

time.timeを使うと、エポックからの経過秒数を求めることができますが、「1502876167.644508」といった数値が得られるだけなので、この関数の戻り値から日付や曜日を計算することは大変です。日付や時刻などの情報を取得する場合には、以下の関数のほうが便利です。

◎関数　ctime([secs])

カテゴリ	timeモジュール
引数	エポックからの秒数、省略時はエポックから現在の時刻までの秒数
戻り値	ローカルの時刻を表現する文字列
説明	secsを省略した場合は現在時刻となる
使用例	t = time.ctime()

◎実行例

```
>>> time.time() Enter
1513506940.992682
>>> time.ctime() Enter
'Sun Dec 17 19:35:46 2017'
```

◎関数　localtime([secs])

カテゴリ	timeモジュール
引数	エポックからの秒数、省略時はエポックから現在の時刻までの秒数
戻り値	ローカルの時刻を表現するオブジェクト（名前付きタプル）
説明	戻り値は名前付きタプルなのでインデックスでも属性でもアクセス可能
使用例	t = time.localtime()

⊙localtimeの戻り値(time.struct_time)の内容

インデックス	属性	値
0	tm_year	例:2017
1	tm_mon	1～12の整数値
2	tm_mday	1～31の整数値
3	tm_hour	0～23の整数値
4	tm_min	0～59の整数値
5	tm_sec	0～61の整数値　※うるう秒を考慮して61まで
6	tm_wday	0～6の整数値、月曜が 0
7	tm_yday	1～366の整数値

localtimeを使用すると、年月日時分秒や曜日といった情報が簡単に入手できます。

◎リスト time_localtime.py

```
import time
t = time.ctime() 1
print(t)

t = time.localtime() 2
print("year={0}        month={1}        day={2}".format(t.tm_year,
t.tm_mon, t.tm_mday))
print("year={0} month={1} day={2}".format(t[0], t[1], t[2]))
```

1で現在時刻のローカル表現を取得してprintで出力しています。**2**でlocaltimeで現在時刻を取得し、年月日を属性とインデックスを使ってそれぞれ出力しています。

■実行結果

```
Wed Aug 16 19:39:52 2017
year=2017 month=8 day=16
year=2017 month=8 day=16
```

➡136

一定時間実行を停止するには

実行を一時的に停止したい場合はsleep関数を使用する

現在実行中のスレッドを一時停止するには**sleep**関数を使用します。一定の時間間隔をあけて処理を行うときなどに使用します。

◎関数　sleep(secs)

カテゴリ	**timeモジュール**
引数	**実行を一時停止する時間、単位は秒**
戻り値	**なし**
説明	**一定時間動作を停止する。浮動小数点での指定も可能**
使用例	**time.sleep(3)**

◎リスト time_sleep.py

```
import time
for i in range(5):
    print(i)
    time.sleep(0.5) 1
```

for文で文字を5回出力するような処理であれば一瞬で終わります。forループの中にsleep関数を挿入すると、その箇所でスレッドが一時停止するので、段階的に処理が行われている様子を見ることができます。この例では、**1**で0.5秒実行を停止しています。0.5秒おきに0～4までの数値が出力されます。

sleep関数は指定された秒数処理を停止します。一定間隔で処理を実行したい場合、処理に要する時間をsleepする時間から差し引く必要があります。

◎リスト time_sleep_interval.py

```
import time, math
for i in range(10):
    prev = time.time() 1
    for i in range(10000): 2
        math.sqrt(i)
    time.sleep(prev+1 - time.time()) 3
    print(time.time()) 4
```

1で現在時刻を求め、2で処理を行います。この例では平方根を10,000回求めています。3で1の時刻に1秒を加え、現在時刻を差し引いて、その時間分sleepしています（処理を1秒以内に抑えるよう2のfor文の回数を調整してください）。4で現在時刻を出力していますが、単にtime.sleep(1)としたときよりも正確なタイミングで実行されていることが確認できます。

➡137

経過時刻を計測するには

timeitモジュールを利用する

特定の処理を行うのにどの程度の時間がかかるのか計測するためには**timeit**モジュールの**timeit**関数が使用できます。

◎関数　timeit(stmt, setup, number=1000000)

カテゴリ	**timeitモジュール**
引数	**stmt：実行する処理、setup：初期化コード、number：繰り返し回数**
戻り値	**実行に要した時間を返す**
説明	**setupで指定された初期化コードを実行したのち、stmtで指定されたコードを1,000,000回繰り返し実行し、その処理に要した秒数を返す。繰り返しの回数を変更する場合にはnumber引数を指定する**
使用例	**t = timeit.timeit("a=math.sqrt(2)", "import math")**

◎実行例

```
>>> timeit.timeit("for i in range(1000): total+=i", "total=0", number=10000) Enter
0.6660786248936574
```

timeitモジュールには**Timer**クラスも用意されています。このクラスを使った実行例を以下に示します。

◎実行例

```
>>> import timeit Enter
>>> t = timeit.Timer("math.sin(10)", "import math") Enter
>>> print(t.timeit()) Enter
0.1606930545110572
```

mathモジュールのsin関数を1,000,000回実行するのに要した時間が表示されます。

timeit関数を使っても、Timerクラスのtimeitメソッドを使っても同じように計測ができます。オブジェクト指向に慣れた人であればTimerクラス、関数が好きな人であればtimeit関数を使うとよいでしょう。

処理時間の計測

timeitは短いコードの時間計測に適しています。ある程度の長さがあるコードの実行時間を計測する場合にはtime関数を使う方法もあります。

◎リスト time_time.py

```
import time

starttime = time.time() 1
total = 0
for i in range(100000):
      total += i
endtime = time.time() 2
elapsed = endtime - starttime 3
print("total={0} elapsed time={1}".format(total, elapsed))
```

1でプログラムの実行開始時の時刻をstarttimeに格納します。その後for文で計算を行い、**2**で計算終了の時刻をendtimeに格納します。**3**でそれらの差分をとり、計算に要した時間を求めています。実行の前後で時刻を計測し、その差分を求めるだけなのでシンプルです。ただし、この計測方法はばらつきが大きいことがあるので、何回か試してみるのがよいでしょう。

◎出力例

```
total=4999950000 elapsed time=0.015625
```

パフォーマンスに気を遣うあまり、コードが読みにくくなってしまうのは望ましくありません。まずは読みやすいコードを書くことに注力し、ある程度動いたあとで、プロ

ファイリングを行い、ボトルネックとなる箇所を特定し、
そこをチューニングするというスタンスがよいでしょう。

4-4 datetimeモジュールによる日付時刻の操作

timeモジュールを使うと年月日や時刻を扱うことができました。基本的な機能は提供されていますが、日付や時刻の経過時間を計算するのは大変なことも少なくありません。そんな状況に応えるため、datetimeモジュールが用意されています。

➡138

日付に関する情報を求めるには

dateオブジェクトを作成するとその日に関する情報が得られる

日付に関する情報を求めるには、まず**datetime**モジュールの**date**オブジェクトを作成し、そのオブジェクトのメソッドを適宜呼び出します。

◎コンストラクタ　date(year, month, day)

カテゴリ	**datetimeモジュール**
引数	**year：年（1から9999）、month：月（1から12） day：日（1から指定された年と月における日数）**
戻り値	**dateオブジェクト**
説明	**year、month、dayで指定された日のオブジェクトを作成する**
使用例	**t = datetime.date(2017, 11, 22)**

◎メソッド　isoformat()

カテゴリ	**datetimeモジュールのdateクラス**
引数	**なし**
戻り値	**"YYYY-MM-DD"のフォーマットで日付を表わす文字列**
説明	**dateオブジェクトの文字列表現と等しい**

使用例	a = datetime.date(2017, 11, 22) t = a.isoformat()

◎メソッド　timetuple()

カテゴリ	datetimeモジュールのdateクラス
引数	なし
戻り値	時刻情報を保持する構造体time.struct_time形式（名前付きタプル）で日時に関する情報を返す
説明	時分秒は0になる
使用例	a = datetime.date(2017, 11, 22) t = a.timetuple()

◎メソッド　weekday()

カテゴリ	datetimeモジュールのdateクラス
引数	なし
戻り値	月曜日を 0、日曜日を 6 として、曜日を整数で返す
説明	weekdayに似たメソッドにisoweekdayがあるが、こちらは月曜日を1、日曜日を7とした値を返す
使用例	a = datetime.date(2017, 11, 22) t = a.weekday()

◎リスト datetime_weekday.py

```
import datetime
d = datetime.date(2002, 12, 4) 1

i = d.isoformat() 2
print(i)

t = d.timetuple() 3
print(t)

w = d.weekday() 4
print(w)
```

1でdateオブジェクトを作成しています。**2**でiso形式の文字列を取得して出力し、**3**でtime.struct_time型のデータを取得して出力し、**4**で曜日を取得して出力しています。0が月曜日なので、2は水曜日です。つまり、2002年の12月4日は水曜日ということがわかります。

■実行結果

```
2002-12-04
time.struct_time(tm_year=2002, tm_mon=12, tm_mday=4,
tm_hour=0, tm_min=0, tm_sec=0, tm_wday=2, tm_yday=338,
tm_isdst=-1)
2
```

Calendarモジュールを使って確認してみましょう。この結果からも2002年の12月4日は水曜日ということが確認できます。

◎実行例

```
>>> import calendar Enter
>>> c = calendar.TextCalendar() Enter
>>> print(c.formatmonth(2002,12)) Enter
    December 2002
Mo Tu We Th Fr Sa Su
                   1
2  3  4  5  6  7  8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

dateクラスでは以下のようなプロパティ（リードオンリー）が利用可能です。

・year　年

・**month 月**
・**day　日**

●今日の日付のdateオブジェクトを生成する

◎メソッド　today()

カテゴリ	**datetimeモジュールのdateクラス**
引数	**なし**
戻り値	**現在のローカルな日付を示すdateオブジェクト**
説明	**クラスメソッドなのでオブジェクトの作成は不要**
使用例	**datetime.date.today()**

◎実行例

```
>>> import datetime      1
>>> datetime.date.today()      2
datetime.date(2017, 12, 22)
>>> type(datetime.date.today())      3
<class 'datetime.date'>
>>> datetime.date.today().isoformat()      4
'2017-12-22'
```

1でdatetimeモジュールを読み込んでいます。**2**でdatetime.dateクラスのtodayメソッドを呼んでいます。todayはクラスメソッドなのでdateオブジェクトを作成はしていません。**3**にあるように、todayメソッドの戻り値はdateオブジェクトとなります。**4**でdateオブジェクトのisoformatメソッドを呼び出しています。

➡139
datetimeモジュールで時刻に関する情報を求めるには

datetimeモジュールのtimeクラスを利用する

日付を扱うdateと対になるのが時刻を扱うtimeです。ここで説明するのはtimeモジュールや、その中にあるtime関数ではなく、**datetime**モジュールの**time**クラスです。混乱しやすいので注意してください。

◎コンストラクタ　time(hour, minute, second)

カテゴリ	**datetimeモジュール**
引数	**hour：時　0 <= hour < 24、minute：分　0 <= minute < 60** **second: 秒 0 <= second < 60**
戻り値	**timeオブジェクト**
説明	**引数を省略するとそのフィールドの値は0となる**
使用例	**d0 = datetime.time()** **d1 = datetime.time(12,34,56)**

作成されたtimeオブジェクトには、以下のようなプロパティ（リードオンリー）が利用できます。

- **hour　　時**
- **minute　　分**
- **second　　秒**
- **microsecond　マイクロ秒**

◎実行例

```
>>> import datetime Enter
>>> d = datetime.time(12, 34, 56) Enter
>>> d.hour Enter
12
>>> d.minute Enter
34
>>> d.second Enter
```

```
56
>>> d.microsecond Enter
0
```

datetimeモジュールとtimeモジュール

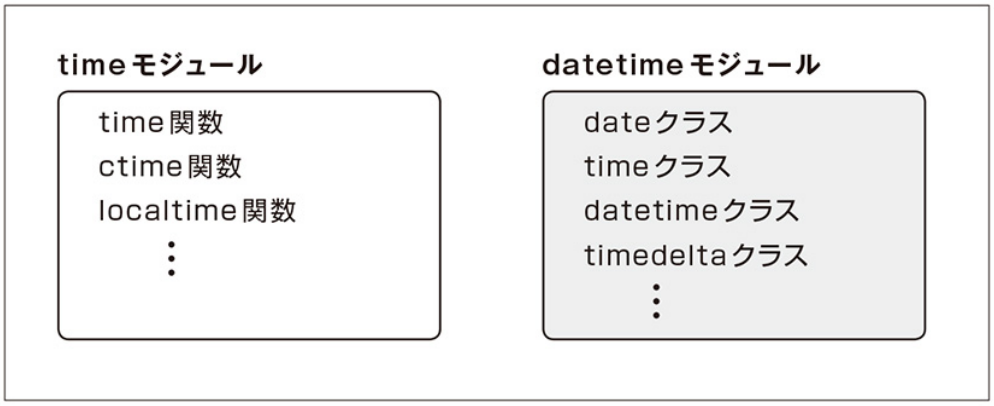
datetimeモジュールには以下のようなクラスが用意されています。

⊙datetimeモジュールのクラス

クラス	説明
date	日付を扱うクラス
time	時刻を扱うクラス
datetime	日付と時刻を扱うクラス、dateとtimeを包含する
timedelta	2つの時刻の差分を扱うためのクラス

timeモジュールとdatetimeモジュールには似たような関数、クラス名が使われているので混乱しがちです。以下の図をご覧ください。

⊙timeモジュールとdatetimeモジュール



timeモジュールに含まれているのは関数です。一方、datetimeモジュールに含まれているのはクラスが中心です。dateクラスは日付を、timeクラスは時間を扱います。datetimeはそれらを包含した関係にあります。これらの違いを意識して使い分けることが大切です。

➡140

経過日数をカウントするには

dateオブジェクトやdatetimeオブジェクトの差分をとる

datetimeやdateクラスを使うと日付を処理できます。これらのクラスを使うと2つの日付の間隔を調べることができそうです。しかしながら、うるう年などを考慮に入れる必要があるので自力で計算するのはとても大変です。**datetime**モジュールには経過日数、時刻を簡単に処理するための**timedelta**クラスが用意されています。

コンストラクタを使ってtimedeltaオブジェクトを作ることもできますが、一般的には2つのdateオブジェクトや2つのdatetimeオブジェクトの加減算の結果として、経過時刻を求めます。

◎リスト datetime_elapsed.py

```
import datetime

birthday = datetime.date(1992, 12, 15) 1
today = datetime.date.today() 2
newcentury = datetime.date(2100, 1, 1) 3
elapsed0 = newcentury - today 4
elapsed1 = today - birthday 5
days0 = elapsed0.days 6
```

```
days1 = elapsed1.days 7
print("{0} days to new century".format(days0))
print("You have lived {0} days so far".format(days1))
```

1で誕生日を、2で本日の日付を、3で2100年の元旦を示すdateオブジェクトを作成しています。4で本日から2100年の元旦までの日数、5で誕生日から今日までに経過した期間（timedeltaオブジェクト）を求めています。6と7でそれぞれの期間の日数を求め、print()でそれらを出力しています。

timedeltaオブジェクトには経過日数を表す**days**と経過秒数を示す**seconds**プロパティがあります。時、分、秒は以下のように計算できます。

◎実行例

```
>>> import datetime Enter
>>> newcentury = datetime.datetime(2100, 1, 1) Enter
>>> today = datetime.datetime.today() Enter
>>> elapsed = newcentury - today Enter
>>> elapsed Enter
datetime.timedelta(29950, 11176, 879676)
>>> "{0} days {1:02d}:{2:02d}:{3:02d}".format(elapsed.days,
elapsed.seconds//3600, elapsed.seconds//60%60,
elapsed.seconds%60) Enter
'29950 days 03:06:16'
```

date/time/datetimeクラスの基本メソッドとプロパティ

この節ではdatetimeモジュール内のクラスの使い方について説明しました。同じ名前のメソッドやプロパティがある

ので混乱しがちです。それぞれのクラスについて、メソッドを実行したとき、プロパティにアクセスしたとき、どのような値が得られるか以下の表に具体例を整理しました。

⊙datetimeモジュールの主要クラス比較

		dateクラス	timeクラス	datetimeクラス
メソッド	timetuple()	下記※1参照		下記※3参照
	weekday()	6		6
	isoformat()	'2017-01-01'	'12:34:56'	'2017-01-01T12:34:56'
	isocalendar()	(2016, 52, 7) ※2		(2016, 52, 7)
プロパティ	hour		12	12
	minute		34	34
	second		56	56
	year	2017		2017
	month	1		1
	day	1		1

※ 1　datetime.date.today().timetuple()を実行したときの出力例

time.struct_time(tm_year=2017, tm_mon=9, tm_mday=20, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=2, tm_yday=263, tm_isdst=-1)

※ 2　ISO年で週は月曜から始まります。2017年の元旦は日曜日で、その週の月曜日は2016年12月26日なのでiso.calendarの値は(2016, 52, 7)となります。

※ 3　datetime.datetime.today().timetuple()を実行したときの出力例

time.struct_time(tm_year=2017, tm_mon=9, tm_mday=20, tm_hour=18, tm_min=50, tm_sec=16, tm_wday=2, tm_yday=263, tm_isdst=-1)

オブジェクトのプロパティを調べるときは

メソッドや関数によっては、単なる値ではなくオブジェクトを返すものがあります。たとえば、datetime.datetime.today()の戻り値をprint命令で出力すると「datetime.datetime(2017, 8, 16, 20, 13, 51, 143712)」のような文字列が出力されます。この結果だけを見ると、todayメソッドは文字列を返しているように見えますが、実際に返されるのはdatetime型のオブジェクトです。printで出力する際に文字列に変換されているのです。type関数を使うと、戻り値の型を確認できます。

◎実行例

```
>>> type(datetime.datetime.today()) Enter
<class 'datetime.datetime'>
```

このようにtodayメソッドはdatetime型を返していることがわかります。では、datetime型にはどのようなプロパティがあるのでしょうか？　リファレンスやマニュアルを見ればわかりますが、**dir関数**を使うと簡単に調べることができます。

◎実行例

```
>>> dir(datetime.datetime) Enter
['__add__', '__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__ne__', '__new__', '__radd__',
'__reduce__', '__reduce_ex__', '__repr__', '__rsub__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', 'astimezone',
'combine', 'ctime', 'date', 'day', 'dst', 'fromordinal', 'fromtimestamp',
'hour', 'isocalendar', 'isoformat', 'isoweekday', 'max', 'microsecond',
```

'min', 'minute', 'month', 'now', 'replace', 'resolution', 'second', 'strftime', 'strptime', 'time', 'timestamp', 'timetuple', 'timetz', 'today', 'toordinal', 'tzinfo', 'tzname', 'utcfromtimestamp', 'utcnow', 'utcoffset', 'utctimetuple', 'weekday', 'year']

アンダースコア2つを前後に持つ属性は内部的に利用することを意図したものです。それ以外の属性に注目してください。date、minute、month、secondなど、どのような値が格納されているか予想がつきそうな属性が見つかると思います。

コンストラクタ、クラスメソッド、インスタンスメソッド

コンストラクタ、クラスメソッド、インスタンスメソッドを区別することはとても大切です。コンストラクタというのはオブジェクトを作るための関数で、コンストラクタの戻り値としてオブジェクトが返されます。オブジェクトとインスタンスは同じ意味で使われます。以下は、コンストラクタを使って、dateクラス、datetimeクラス、timeクラスの3つのオブジェクトを作成している例です。

◎実行例

```
>>> d0 = datetime.date(2018,1,1) Enter
>>> type(d0) Enter
<class 'datetime.date'>
>>> d1 = datetime.datetime(2018,1,1) Enter
>>> type(d1) Enter
<class 'datetime.datetime'>
>>> d2 = datetime.time() Enter
>>> type(d2) Enter
<class 'datetime.time'>
```

クラスメソッドは、クラスに共通の性質を返すものなので、インスタンスを作成する必要はありません。クラス名の直後に直接メソッドを記述することで呼び出すことができます。

◎実行例

```
>>> datetime.date.today() Enter
datetime.date(2017, 8, 19)
>>> datetime.date.min Enter
datetime.date(1, 1, 1)
>>> datetime.datetime.today() Enter
datetime.datetime(2017, 8, 19, 11, 5, 6, 645345)
>>> datetime.datetime.now() Enter
datetime.datetime(2017, 8, 19, 11, 5, 19, 613322)
>>> datetime.time.min Enter
datetime.time(0, 0)
```

todayメソッドやnowメソッドを実行するにあたり、オブジェクトを作成していないことに注意してください。

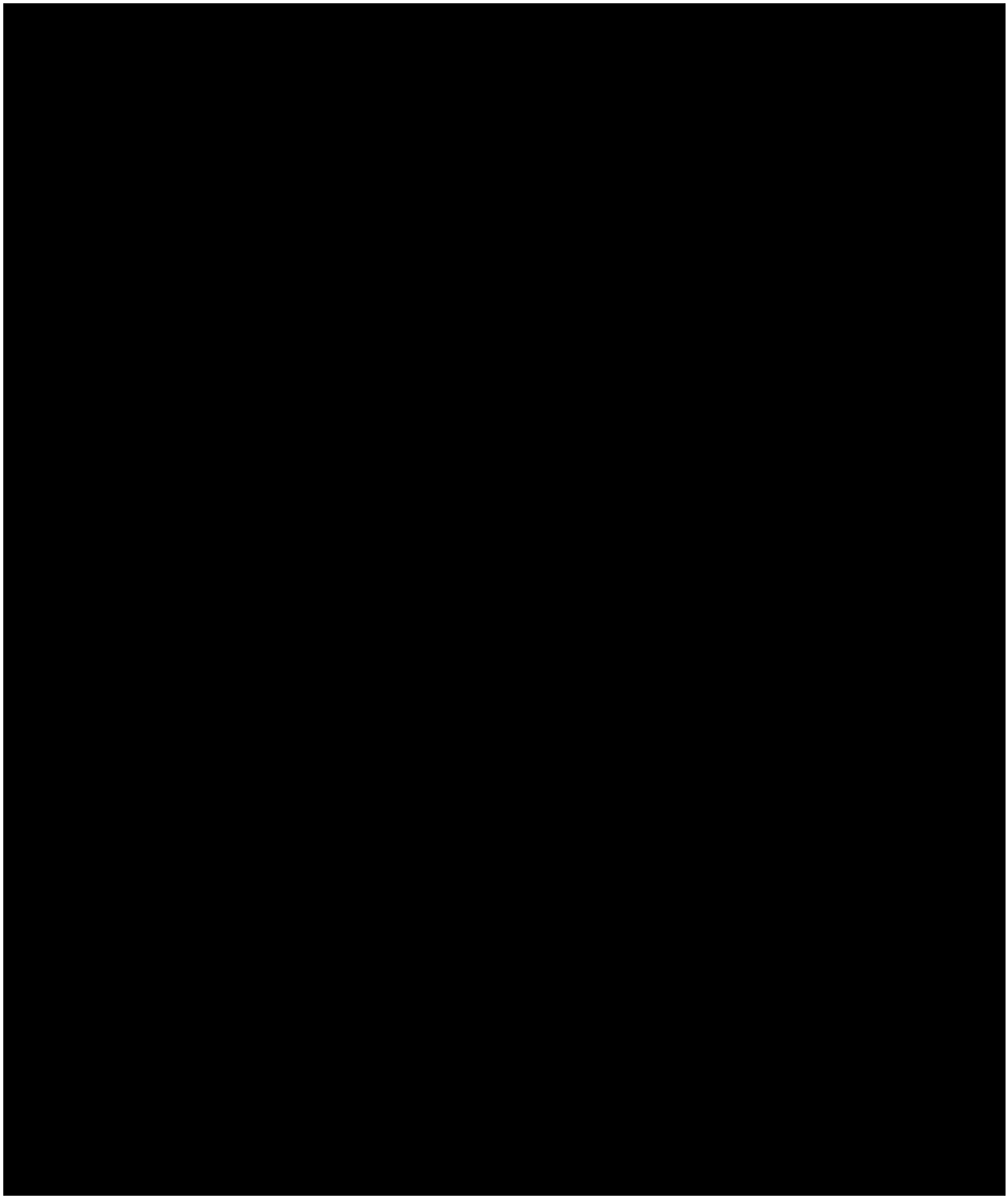
最後にインスタンスメソッドです。インスタンスメソッドの場合、オブジェクトを最初に作成し、その戻り値として得られたオブジェクトのメソッド（インスタンスメソッド）を呼び出します。以下の例ではd0、d1、d2がオブジェクトであり、それらのインスタンスメソッドを呼び出しています。

◎実行例

```
>>> d0 = datetime.date(2018,1,1) Enter
>>> d1 = datetime.datetime(2018,1,1) Enter
>>> d2 = datetime.time() Enter
>>> d0.isoformat() Enter
'2018-01-01'
>>> d1.isoformat() Enter
'2018-01-01T00:00:00'
>>> d2.isoformat() Enter
```

'00:00:00'

Chapter 5



数値演算と
乱数

数値演算はプログラミングの基本です。四則演算はもちろんですが、浮動小数点数を整数に変換したり、平方根を求めたりする必要性に迫られることもあるでしょう。ゲームなど偶発的な要素を実装するためには乱数が欠かせませんし、角度を扱うなら三角関数も必要になります。本章ではいろいろな数値演算処理と乱数の使い方について説明します。

5-1 さまざまな数値演算

Pythonの組み込み関数、mathモジュール、randomモジュールには、多くの数値演算用の関数が用意されています。本節ではこれらの中から使用頻度が高いと思われる関数を取り上げて説明します。

➡141

小数を整数に変換するには

ceil／floor／int／roundを状況に応じて使い分ける

小数を整数に変換する必要性に迫られるケースは少なくありません。小数を整数に変換するときには、**切り捨て**、**切り上げ**、**丸め**のどれを使うのか意識することが大切です。

◎関数　ceil(x)

カテゴリ	**mathモジュール**
引数	**数値**
戻り値	**x以上の最小の整数を返す。切り上げに相当する**
説明	**0.9、0.1、0.01、いずれの場合も1が返される**
使用例	**v = math.ceil(2.6)**

◎関数　floor(x)

カテゴリ	**mathモジュール**
引数	**数値**
戻り値	**x以下の最大の整数を返す。切り捨てに相当する**
説明	**1.9、1.0、1.0001、いずれの場合も1が返される**
使用例	**v = math.floor(2.6)**

floorは床、ceilは天井です。天井と床の間に浮動小数点数があると考えてください。天井は大きい値なので切り上げ、床は小さい値なので切り捨てと考えると覚えやすいでしょう。

int関数も引数を切り捨てて整数値にするので、floor関数と似た働きをします。

◎関数　int(x[, radix])

カテゴリ	**組み込み関数**
引数	**数値**
戻り値	**x:数値や数値を表す文字列、radix:基数**
説明	**引数を整数に変換する。浮動小数点数については小数点以下を切り捨て、文字列は数値に変換する（017で解説）**
使用例	**v = int(3.2)**

基数とは何進数かを表す数値で、デフォルトは10です。

◎実行例

```
>>> int(2.6) Enter       ←小数が切り捨てられて整数に
2
>>> int("-345") Enter          ←文字列-345が数値の-345に
-345
>>> int("FF", 16) Enter       ←基数16で整数に
255
>>> int("1111", 2) Enter    ←基数2で整数に
15
```

floorも切り捨てを行いますが、負の数を扱う場合、intとfloorは結果が変わってくるので注意してください。

◎実行例

```
>>> math.floor(-1.7) Enter
-2
```

```
>>> int(-1.7) Enter
-1
```

指定した桁で丸めるには、**round**関数を使います。

◎関数　round(number [, ndigits])

カテゴリ	**組み込み関数**
引数	**number: 数値、ndigits:小数点以下の桁数**
戻り値	**丸めた値**
説明	**指定した桁で丸める（多くの場合で四捨五入と同じ結果となる）**
使用例	**v = round(3.1415, 2)**

◎実行例

```
>>> round(3.14159) Enter
3
>>> round(3.14159, 2) Enter
3.14
>>> round(1.73205) Enter
2
```

指定した桁数で丸められた数値が返ってきています。桁数を省略した場合は一番近い整数となります。

◎実行例

```
>>> round(1.5) Enter
2
>>> round(0.5) Enter
0
```

1.5が2に変換されるのは自然ですが、0.5が0になることには違和感を覚えるかもしれません。これはPython 3から

roundの仕様が変わり、ちょうど真ん中の値は偶数のほうに丸められるようになったためです。0.5は0と1のちょうど真ん中なので、偶数の0に丸められています。このように、より誤差の少ないとされる数値に丸めることを**round-half-even方式**と呼びます。次に、これらの3つの関数を使ったスクリプトの例を示します。

◎リスト math_float2int.py

```
import math
print("input floating point number:", end="")
val = float(input())
v0 = math.ceil(val)
v1 = math.floor(val)
v2 = round(val)
print("ceil:{0} floor:{1} round:{2}".format(v0, v1, v2))
```

input関数で数値を受け取り、float関数で浮動小数点数に変換しています。その数をceil、floor、roundで整数に変換して出力しています。2行目のprint関数の最後にあるend=""は改行をしないための引数です。

■実行結果

```
input floating point number:3.14
ceil:4 floor:3 round:3
```

➡142
絶対値を求めるには
abs関数を使用する

絶対値とは数値のもつ値のことで正負といった符号は考慮しません。

◎関数　**abs(x)**

カテゴリ	**組み込み関数**
引数	**数値**
戻り値	**数の絶対値を返す**
説明	**引数は整数または浮動小数点数を指定**
使用例	**v = abs(-3)**

◎実行例

```
>>> abs(-19) Enter
19
```

➡143

べき乗を求めるには

pow関数もしくは**演算子を使用する

べき乗とは、同じ数を何回掛けるかという計算です。

◎関数　**pow(x, y)**

カテゴリ	**組み込み関数**
引数	**x：底、y：指数**
戻り値	**x の y 乗を返す**
説明	**pow関数はx**yと等価となる**
使用例	**v = math.pow(2, 3)**

◎実行例

```
>>> pow(2, 4) Enter    ←2の4乗
16
```

```
>>> 2 ** 4 Enter　←同じ計算を**演算子で記述
16
```

NOTE mathモジュールにもpow関数があります。この関数は常にfloatで計算します。

実行例

```
>>> math.pow(2, 4) Enter
16.0
```

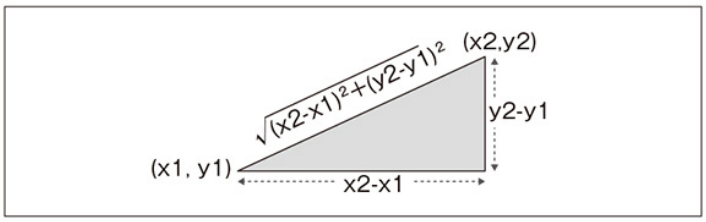
➡144

平方根を求めるには

sqrtを使用する。原点からの距離を求める場合にはhypotが便利

二乗したらxになる数のことを「**xの平方根**」といい、「**√x**」と表します。三平方の定理でも平方根が使われますが、ゲームにおいては2つの座標の距離を求めるときに平方根がよく使用されます。

◉平方根



◎関数　sqrt(x)

カテゴリ	mathモジュール
引数	数値

戻り値	**xの平方根を返す**
説明	**√xと同じ**
使用例	**v = math.sqrt(25)**

　原点(0, 0)から座標(x,y)までの距離を求める場合は、**hypot**関数が便利です。

◎関数　hypot(x, y)

カテゴリ	**mathモジュール**
引数	**x：x座標の値、y：y座標の値**
戻り値	**原点から座標(x,y)までの距離**
説明	**(sqrt(x*x + y*y))と同じ値が返される**
使用例	**v = math.hypot(1,2)**

◎実行例

```
>>> math.sqrt(2) Enter
1.4142135623730951
>>> math.hypot(1, 2) Enter
2.23606797749979
```

　sqrtで平方根を求めています。√2は1.1414...です。また、hypotで原点から(1,2)への距離を求めていますが、この値は√(1x1+2x2) = √5となります。

➡145
角度とラジアンを相互変換するには

角度をラジアンに変換するにはradians、逆はdegreesを使う

Pythonでは、一周は360度ではなく、**2×π**の「**ラジアン**」という単位を使用します。

◎関数　radians(x)

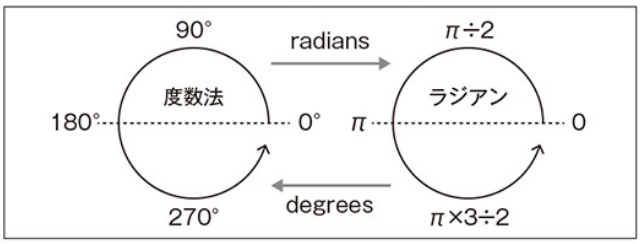
カテゴリ	**mathモジュール**
引数	**角度（度数法）**
戻り値	**角度（ラジアン）**
説明	**度数法による角xをラジアンの値に変換する**
使用例	**v = math.radians(90)**

◎関数　degrees(x)

カテゴリ	**mathモジュール**
引数	**角度（ラジアン）**
戻り値	**角度（度数法）**
説明	**ラジアンによる角xを度数法の値に変換する**
使用例	**v = math.degrees(3.141592)**

これら2つの式の関係は次のようになります。円周率πは**math.pi**と表せます。

⊙radians関数、degrees関数



◎実行例

```
>>> math.radians(180) Enter
3.141592653589793
>>> math.degrees(math.pi) Enter
180.0
```

math.radiansを使って、180度をラジアンの値3.14...に変換しています。math.piは円周率πで、その度数法の値をdegreesで求めると180という値が得られます。

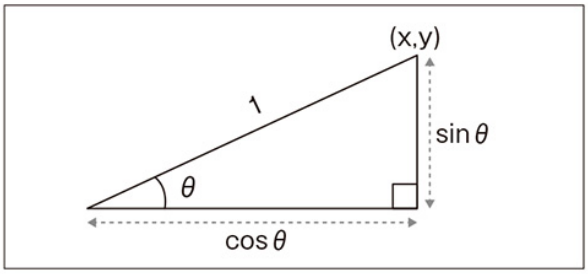
➡146
三角関数を使うには

正弦はsin、余弦はcos、正接はtanを使う

なめらかなカーブを描くとき、回転行列の計算をするとき、いろいろな場面で三角関数が活躍します。**math**モジュールを使うとさまざまな三角関数が計算できます。

グラフィックス計算を行うときには**ベクトル**（角度と大きさをもつ値）を多用しますが、ベクトルをx座標、y座標に変換するときにも**sin**関数、**cos**関数を使用します。この考え方の基本は、下記の図のとおりです。直角三角形の最も長い辺を1とすると、他方の2辺の長さはsinθとcosθになります。

⊙sin関数、cos関数のベースになる考え方



◎関数　sin(x)

カテゴリ	**mathモジュール**
引数	**角度（ラジアン）**
戻り値	**正弦の値を返す**
説明	**サインの値を返す。xの単位はラジアンである**

使用例	v = math.sin(math.radians(90))

◎関数　cos(x)

カテゴリ	mathモジュール
引数	角度（ラジアン）
戻り値	余弦の値を返す
説明	コサインの値を返す。xの単位はラジアンである
使用例	v = math.cos(math.radians(0))

◎関数　tan(x)

カテゴリ	mathモジュール
引数	角度（ラジアン）
戻り値	正接の値を返す
説明	タンジェントの値を返す。xの単位はラジアンである
使用例	v = math.tan(math.radians(45))

◎リスト math_sincos.py

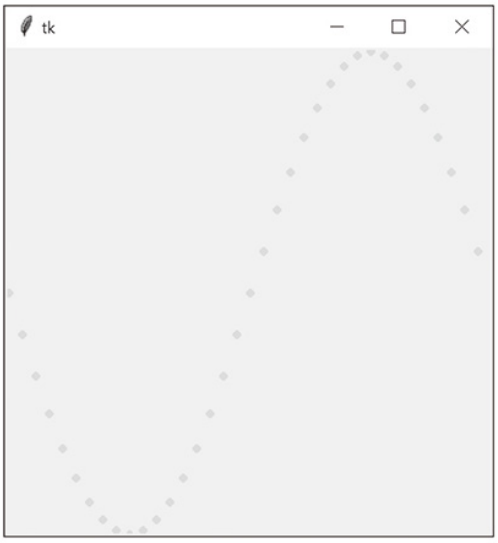
```
import math, tkinter as tk 1
root = tk.Tk()
canvas = tk.Canvas(root, width=360, height=360)
canvas.pack()
for i in range(36): 2
    y = math.sin(math.radians(i*10)) * 180 + 180 3
    canvas.create_oval(i*10, y, i*10+5, y+5,
        fill="#00FF00", outline="")
root.mainloop()
```

サインカーブを描画するためにtkinterモジュールを使用しています1。ここではmathモジュールのみに注目してください（tkinterモジュールはChapter 7で解説します）。

2のfor文のiは0から35まで変化します。3が座標を求める部分です。角度に10を掛けた値を、math.radians関数に渡しています。つまり、1周期分の角度を求めています。その値

をmath.sin関数に引数として与えています。math.sin関数は-1から+1の値域をとりますが、そのままCanvasに描画しても変化が見えないため180倍して座標を求めています。

◉出力例（緑の点で波形が描画される）



➡147

座標(x,y)とX軸のなす角度を求めるには

atan2を使うと座標から角度を求められる

sin関数、cos関数を使えば、角度からx座標とy座標を求めることができます。逆の処理を行うのが**atan2**関数です。

◎関数　atan2(y, x)

カテゴリ	**mathモジュール**
引数	**y：y座標の値、x：x座標の値**
戻り値	**逆正接の値を返す。atan(y / x)をラジアンで返す。戻り値は-piからpiの間になる**
説明	**座標(x, y)がx軸となす角度を返す。xとyの順番に注意**
使用例	**v = math.degrees(math.atan2(-1, 1))**

◎リスト math_atan2.py

```
import math, tkinter as tk 1
def mousemove(e): 2
    canvas.coords(line, (100,100, e.x, e.y)) 3
    theta = math.atan2((e.y-100)*-1, e.x-100) 4
    print("degree={}".format(math.degrees(theta))) 5
root = tk.Tk()
canvas = tk.Canvas(root, width=200, height=200)
canvas.pack()
canvas.create_line(0, 100, 200, 100)
canvas.create_line(100, 0, 100, 200)
line = canvas.create_line(100, 100, 100, 100)
canvas.bind('<Motion>', mousemove) 6
root.mainloop()
```

　このサンプルもtkinterを使っています。まず**1**でmathとtkinterモジュールを読み込んでいます。

　2がマウス移動時に呼び出されるコールバック関数です。**6**のbindメソッドにより、canvas上でマウスが移動したときに呼び出されるようになります。

　マウスの座標が「e.x, e.y」で与えられます。**3**では、Canvasのlineメソッドを使い、画面の中心座標(100,100)からマウスの座標まで線を描画しています。

　4が角度を求めている部分です。Canvasの中心座標が100、100なので、中心からの差分はx座標が「e.x-100」、y座標が「(e.y-100)*-1」となります。y座標に-1を掛けているのは、Y軸を反転するためです。

　その座標をmath.atan2関数に渡して角度を求めています。

　5でラジアンの角度を度数法の単位に変換し、print関数で出力しています。

マウスを移動すると、画面の中心から右方向を0とした場合の、角度が出力されます。

⦿出力例

コールバック関数とイベントハンドラ

コールバック（callback）とは「折り返し電話をする」という意味です。「今忙しいからあとで電話するね」といった状況で使用します。プログラミングにおけるコールバック関数もこれに似ています。何らかの処理を行うとき、完了までに時間がかかるだけでなく、いつ終わるのか予測できないことがあります。たとえば、ネットワークからファイルを取得したり、巨大なデータベースから検索するような状況が相当します。このようなときは処理の完了を待つのではなく、「終わったらこの関数を呼び出してね」と依頼することができます。このような関数をコールバック関数と呼びます。

イベントハンドラもコールバック関数の一種です。マウスがクリックされる、キーが押される、マウスが移動するといった処理はいつ起きるかわかりません。このようなイベントが起きたときに実行する関数を「イベントハンドラ」と呼びます。イベントハンドラといっても通常の関数と変わりません。ただし、自分で呼び出すのではなく、システムがイベントハンドラを呼び出してくれます。どのような引数が渡されるかはイベントによって異なるので、リファレンスやデバッガで調べるとよいでしょう。

➡148

１０進数と１６進数を相互に変換するには

10進数を16進数にするにはhexやformat、16進数を10進数にするにはintを使う

人間は10進数に慣れ親しんでいますが、コンピュータはそうとは限りません。16進数に変換したり、その逆を行ったりすることもあります。**hex**関数は10進数を16進数の文字列に変換する関数です。

◎関数　hex(x)

カテゴリ	**組み込み関数**
引数	**整数**
戻り値	**16進数を表す文字列**
説明	**先頭に0xが付与された形式の16進数文字列を返す**
使用例	**v = hex(255)**

◎実行例

```
>>> hex(255) Enter
```

```
'0xff'
```

　文字列のformatメソッドを使っても同様の値が取得できます。

◎実行例

```
>>> "0x{:04x}".format(255) Enter
'0x00ff'
```

　「{:04x}」の**0**は、0で**パディング**（ほかと桁数を合わせるために0で埋めること）、4は桁数、**x**は16進数を意味します。xの代わりに大文字の**X**と指定すると、戻り値の16進数のA～Fが大文字になります。また、16進数の数値のリテラルは0xを前につけて16進数であることを示します。たとえば、16進数で6Fは、10進数で111になります。通常の数値と同じように扱うことができます。

◎実行例

```
>>> 0x6f + 10 Enter
121
```

　16進数を10進数に変換するには、int関数（017　文字列を数値に変換するには）を使います。

◎実行例

```
>>> int("0x6F", 16) Enter
111
```

余りと商を一度に求めるには

divmodもしくは//や%演算子を使う

どの言語でも商を求めたり、余りを求めたりする演算子が用意されていますが、それらを一度に求められる関数は珍しいかもしれません。

◎関数　**divmod(a, b)**

カテゴリ	**組み込み関数**
引数	**a：割られる数、b：割る数**
戻り値	**aをbで割ったときの商と剰余からなるタプルを返す**
説明	**引数が整数の場合、結果は(a // b, a % b)と同じになる**
使用例	**v = divmod(7, 3)**

◎実行例

```
>>> divmod(7,3) Enter
(2, 1)      ←商が2で余りが1
>>> 7//3 Enter   ←//演算子は切り捨ての除算
2
>>> 7%3 Enter      ←%演算子は除算した余り
1
```

7を3で割ると商が2で余りが1となります。なお、Pythonでは**//**が切り捨ての除算、**%**が余りとなります。よって、7//3を実行すると2が、7%3を実行すると1が得られます。

➡150

複数の要素のTrue/Falseを一度に調べるには

すべての条件を調べる場合はall、どれか1つはanyを使う

たくさんのアイテムがあり、「それらがすべて条件を満たすか」「それらのどれか1つでも条件を満たすか」という処理を書くことは少なくありません。そんな状況に直面したときは**all**関数や、**any**関数を使うとシンプルに処理を記述できるかもしれません。

◎関数　all(a)

カテゴリ	**組み込み関数**
引数	**リストやタプルなどのイテレート可能なオブジェクト**
戻り値	**TrueかFalse**
説明	**リストなどに含まれる要素がすべてTrueのときに全体としてTrueを返す。それ以外のときはFalseを返す**
使用例	**f = all([True, False, True])**

◎関数　any(a)

カテゴリ	**組み込み関数**
引数	**リストやタプルなどのイテレート可能なオブジェクト**
戻り値	**TrueかFalse**
説明	**リストなどに含まれる要素がすべてFalseのときに全体としてFalseを返す。それ以外のときはTrueを返す**
使用例	**f = any([False, True, False])**

◎実行例

```
>>> data = [30, 20, 10] Enter  1
>>> [x > 10 for x in data] Enter  2
[True, True, False]
>>> any([x > 10 for x in data]) Enter  3
True
>>> all([x > 10 for x in data]) Enter  4
```

```
False
```

1でリストに「30, 20, 10」というデータを格納しています。**2**では、そのデータをもとにリスト内包表記を使い、xが10より大きいか否かを示すブール型のリスト「[True, True, False]」を作成しています。

3のanyは、1つでもTrueがあればTrueが返ります。ここではTrueが2つあるので、anyの戻り値はTrueになります。

一方、**4**のallは、すべてがTrueのときのみTrueが返されます。今回はすべてTrueではないので、allの戻り値はFalseになっています。

◎リスト all_any.py

```
ages = (19, 20, 23, 18, 25)
adults = [x >= 20 for x in ages]
print("adults:", adults)
print("all adults? ", all(adults))
print("any adults? ", any(adults))
```

5人分の年齢を示すagesタプルをもとに、リスト内包表記を使って、20歳以上のブール値を保持するadultsというリストを作成しています。あとは、printでadultsデータ、all、anyの戻り値を出力しています。全員20歳以上ではないのでallはFalseを、20歳以上の人が1人以上いるのでanyはTrueを返しています。

■実行結果

```
adults: [False, True, True, False, True]
all adults?   False
any adults?   True
```

5-2 乱数

ゲームやシミュレーションを実装するときに乱数は欠かせません。randomモジュールには乱数を扱うための関数が多数用意されています。

➡151

乱数を生成するには

乱数の分布に応じてrandrange、randint、random、uniformなどの関数を使い分ける

まずは、どの数も同じ頻度で生成する**一様分布**の乱数を見てみましょう。

◎関数　randrange(stop)

カテゴリ	**randomモジュール**
引数	**stop：乱数の上限**
戻り値	**0からstopまで（stopは含まない）の整数の乱数を返す**
説明	**range()と同じようにrandrange(start, stop[, step])と指定することも可能である。この場合は、startからstopまでの乱数を生成する**
使用例	**r = random.randrange(1, 7)**

randint(a, b)という関数もありますが、これはrandrange(a, b+1)と等価です。たとえば、randint(1, 6)はrandrange(1,7)と同じで、1～6までの乱数を生成します。

ほかにも次のような一様分布の乱数を発生させる関数があります。

◎関数　**random()**

カテゴリ	randomモジュール
引数	なし
戻り値	0から1までの浮動小数点の乱数を生成する
説明	seed関数で乱数生成器を初期化すると同じパターンの乱数を生成できる
使用例	r = random.random()

◎関数　**uniform(a, b)**

カテゴリ	randomモジュール
引数	a, b：乱数を生成する範囲を指定する
戻り値	a以上、b以下の浮動小数点数
説明	ランダムな浮動小数点数を返す
使用例	r = random.uniform(1, 100)

　乱数というと「どの数値も同じ確率で生成される」と思う方もいるかもしれませんが、必ずしも乱数は一様分布とは限りません。ここでは一様分布以外の関数を見ていきましょう。

◎関数　**normalvariate(mu, sigma)**

カテゴリ	randomモジュール
引数	mu：平均μ、sigma：標準偏差σ
戻り値	浮動小数点数
説明	（μ、σ）の正規分布に従った乱数を返す
使用例	r = random.normalvariate(50,10)

◎関数　**triangular(low, high, mode)**

カテゴリ	randomモジュール
引数	low：最小値、high：最大値、mode：最頻値
戻り値	浮動小数点数
説明	low以上、high以下で、modeを最頻値とする乱数を返す
使用例	r = random.triangular(0,100,50)

◎関数　betavariate(alpha, beta)

カテゴリ	randomモジュール
引数	alpha：α値、beta：β値
戻り値	浮動小数点数
説明	パラメータα、βで表されるベータ分布の乱数を返す
使用例	r = random.betavariate(0.5, 0.5)

　これらの関数がどんな分布をしているか、その様子を見てみましょう。

◎リスト math_random2.py

```
import random, collections 1
bucket = collections.Counter() 2
for _ in range(10000): 3
    #r = random.normalvariate(0, 3)
    r = random.betavariate(.5, .5) * 10 4
    #r = random.uniform(-10, 10)
    #r = random.triangular(-10, 10)
    v = round(r) 5
    bucket[v] += 1 6
for k in sorted(bucket.keys()): 7
    v = int(bucket[k] / 50) 8
    print("{0:3}:{1}".format(k, "*" * v)) 9
```

　1でrandomモジュールとcollectionsモジュールを読み込んでいます。2で出現回数を数えるカウンタを作成しています。3でfor文を使い、10000回ループを繰り返しています。4で乱数を生成しています。5のround関数で乱数を整数に変換し、6でその値をキーとして、出現頻度をカウントしています。
　7のfor文は出力用のループです。8で出現頻度を取り出して、9でその頻度を棒グラフにして表示しています。

コメントアウトを適宜外して、乱数を生成する関数が異なると、出現頻度がどのように変化するか確認してください。

⊙出力例1 正規分布（平均=0、標準偏差=3）

```
-8:
-7:*
-6:***
-5:******
-4:**********
-3:**************
-2:********************
-1:************************
 0:*************************
 1:************************
 2:********************
 3:**************
 4:**********
 5:******
 6:***
 7:*
 8:*
```

⊙出力例2 ベータ分布（α=0.5、β=0.5）

```
 0:*****************************
 1:**********************
 2:***************
 3:************
 4:************
 5:************
 6:************
 7:************
 8:***************
 9:**********************
10:*****************************
```

➡152

ランダムに並べ替え、もしくは1つを選択するには

並べ替えはshuffle、1つ選択するにはchoiceを使う

乱数を生成する場合、何らかの目的があるはずです。トランプなら、札から1枚ひいたり、並べ替えたりするでしょう。クイズなら、出題順序を変えたりするでしょう。乱数を使って自分で処理することも可能ですが、**random**モジュー

ルには単に乱数を生成するだけでなく、便利な関数が用意されています。

◎関数　choice(seq)

カテゴリ	randomモジュール
引数	seq：シーケンス型
戻り値	seqの中の値の1つ
説明	リストやタプルなどのシーケンス型からランダムに1つの要素を選択し、その値を返す
使 用 例	r = random.choice(["Spring", "Summer", "Autumn", "Winter"])

◎関数　shuffle(seq)

カテゴリ	randomモジュール
引数	seq：ミュータブルなシーケンス型
戻り値	なし
説明	リストなどのシーケンス型をランダムに並べ替える。引数で与えられたシーケンス自身が並べ替えられる。自身とは別の並べ替えた結果を返すのではないことに注意
使用例	random.shuffle(seasons)

　タプルは値を変更することができません。その場合は、**sample**関数を使用するとよいでしょう。

◎関数　sample(seq, k)

カテゴリ	randomモジュール
引数	seq：シーケンス型、k：選択する要素の個数
戻り値	タプルなどの母集団からk個の要素を選び、ランダムに並べ替えたリストを返す
説明	shuffleと違い、元のシーケンス型とは別の並べ替えられた値が戻り値として返される
使用例	r = random.sample(["apple", "banana", "orange"], 3)

◎実行例

```
>>> import random Enter
>>> seasons = ["Spring", "Summer", "Autumn", "Winter"] Enter
>>> random.shuffle(seasons) Enter
>>> seasons Enter
['Autumn', 'Spring', 'Winter', 'Summer']
>>> random.sample(seasons, len(seasons)) Enter
['Autumn', 'Winter', 'Spring', 'Summer']
```

randomモジュールを読み込み、seasonsという配列を作成し、random.shuffleで並べ替えをしています。seasonsの順序が入れ替えられていることがわかります。

一方、random.sampleを実行した場合は、戻り値としてランダムに並べ替えられた値が返されています。

次にトランプのカードをランダムに並べ替える例を示します。

◎リスト random_shuffle.py

```
import random
cards = [] 1
for i in "A23456789TJQK": 2
    for j in ("H", "D", "C", "S"): 3
        cards.append(j + "-" + i) 4
random.shuffle(cards) 5
for i, c in enumerate(cards): 6
    if i % 13 == 0: 7
        print()
    print("{0:3} ".format(c), end="") 8
```

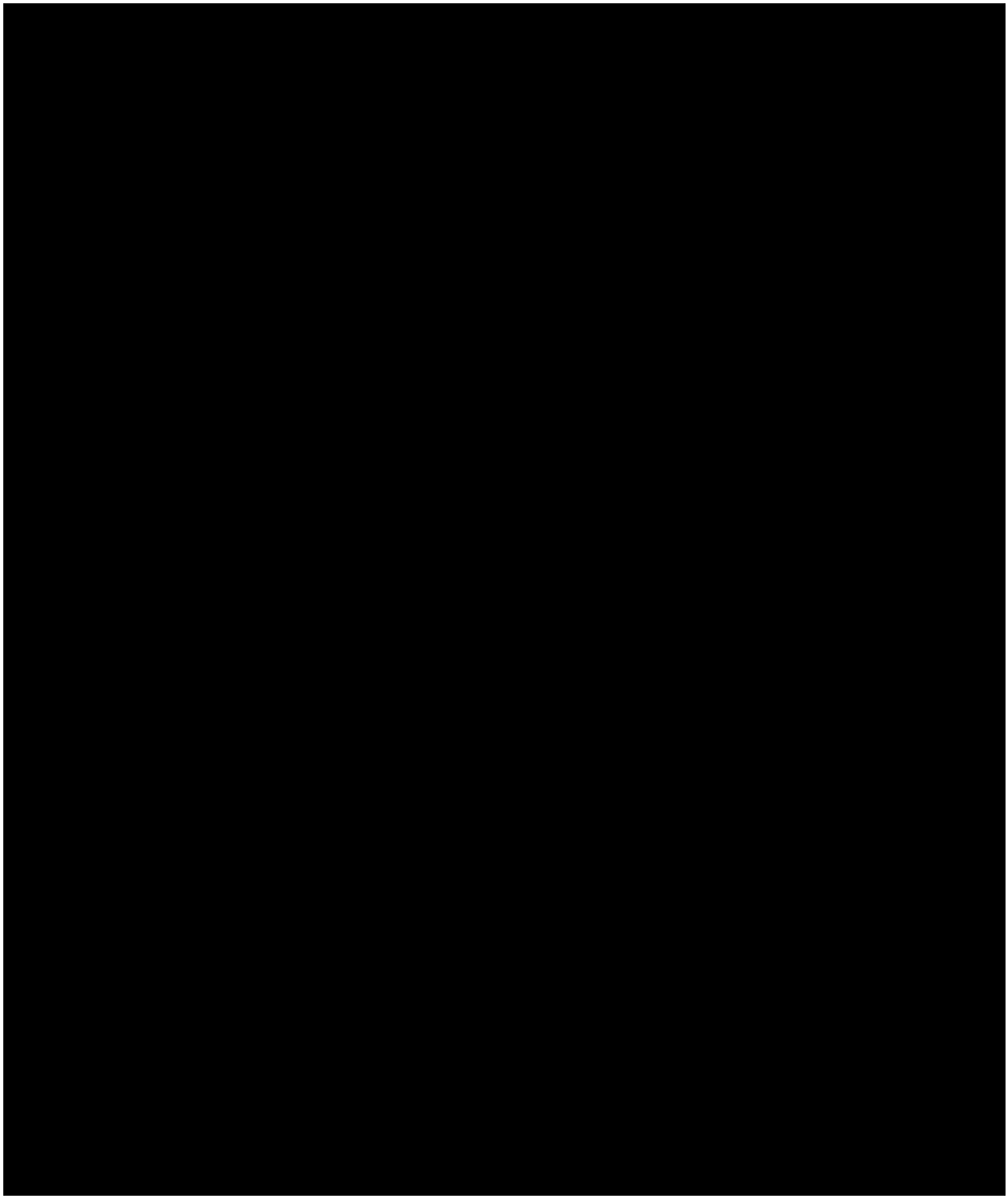
1でカードを格納するリストを初期化しています。2と3の2重ループで（A～Kまでの13回×H、D、C、Sのどれか4回）、合計52回ループを実行します。ループ中の4ではカー

ドをリストに追加し、**5**で順番の並べ替えを行います。**6**のfor文は出力用のループです。**7**で、13個ごとに改行を出力し、**8**でリストの中身を表示しています。

■実行結果

```
S-3 H-K C-T S-J S-4 C-5 S-2 S-6 H-2 C-4 S-Q S-T H-J
C-7 C-3 H-4 H-Q D-T H-T D-3 C-8 S-A S-5 S-9 C-6 C-2
D-7 H-7 D-K D-A H-9 H-A D-5 H-8 D-2 S-8 H-6 C-Q D-J
D-4 C-K H-3 C-9 H-5 S-K C-A S-7 C-J D-8 D-9 D-6 D-Q
```

Chapter 6



ネットワークへの
アクセス

20～30年ほど前、インターネットを使っている人はごく少数でしたが、今ではインターネットのない生活は考えられないほどに普及しました。Pythonでもネットワークやインターネットにアクセスする機能は充実しています。本章では、そのような機能について見ていきます。

6-1 URLの操作とアクセスの基本

Webサーバとやり取りをする際には、HTTPプロトコルが使用されます。本節では、HTTPの利用を前提とし、URLをどのように解析し操作するかをまず解説します。URLでアクセスしてページやファイル、JSONファイルを取得する方法を示します。

➡153

URLのフォーマットを解析するには

urllib.parseモジュールを使って解析する

日頃よく目にするホームページのアドレスは、正確にはプロトコル、ホスト、ポート、パス…などのフォーマットが定められています。**urlparse**関数はURLのフォーマットを解釈します。

◎関数　urlparse(urlstring)

カテゴリ	**urllib.parseモジュール**
引数	**解析対象となるURLの文字列**
戻り値	**URLを解析した結果を（スキーマ、アドレス、パス、パスに対するパラメータ、クエリ、フラグメント識別子）からなるタプルで返す**
説 明	**URL は scheme://host.domain[:port]/path?query#fragmentのようなフォーマットを取るが、与えられたURLを各部分に分割する**
使 用 例	**r = urllib.parse.urlparse("http://www.impress.co.jp/news.html?query=python")**

◎リスト urllib_parse0.py

```
import urllib.parse
r = urllib.parse.urlparse("https://www.impress.co.jp/business.html;hoge=1;?say=hello#01_digital")
print(r)
```

■実行結果

```
ParseResult(scheme='https', netloc='www.impress.co.jp', path='/business.html', params='hoge=1;', query='say=hello', fragment='01_digital')
```

スキーマ(scheme)がhttps、ネットワーク上のアドレスがwww.impress.co.jp、パスが/business.html、パラメータがhoge=1、クエリがsay=hello、フラグメントが01_digitalとURLが部分ごとに分割されていることがわかります。

➡154

URLで絶対パスを取得するには

urljoin関数を使って基準となるパスと相対パスを連結する

HTMLコンテンツを記述する際には「"../images/logo.png"」のような**相対パス**がよく使われます。このような相対パスから**絶対パス**を取得する場合には、**urljoin**関数を使用します。

◎関数　urljoin(base, path)

カテゴリ	urllib.parseモジュール
引数	base：基準となる絶対パス、path：相対パス
戻り値	絶対パス
説明	baseが基準の絶対パスであるとき、相対パスpathに対応する絶対パスを返す
使用例	r = urllib.parse.urljoin("http://www.hoge.com/news/index.html", "../images/logo.png")

　baseとなるURLが「/」で終わっているか否か、フラグメントの有無などによらず、絶対パスが取得できます。

◎リスト urllib_urljoin.py

```
from urllib.parse import urljoin
a = urljoin("http://www.impress.co.jp", "whatsup.html")
print(a)

b = urljoin("http://www.impress.co.jp/news/", "../weather/index.html")
print(b)

c = urljoin("http://www.impress.co.jp/search?key=hello", "../images/logo.png")
print(c)
```

　urljoin関数を使って、基準となるパスと相対パスから、絶対URLを作成しています。

■実行結果

```
http://www.impress.co.jp/whatsup.html
http://www.impress.co.jp/weather/index.html
```

http://www.impress.co.jp/images/logo.png

➡155

URLで示されるページを取得するには

urlopen関数でHTTPResponseオブジェクトを取得し、readメソッドを呼ぶ

最も基本的な操作として、インターネットからホームページを取得してみましょう。

◎関数　urlopen(url)

カテゴリ	**urllib.requestモジュール**
引数	**URLの文字列**
戻り値	**HTTPResponseオブジェクト**
説明	**urlへ接続して、コンテンツを取得する**
使用例	**res = urllib.request.urlopen('http://impress.co.jp/')** **html = res.read()**

◎リスト urllib_basic0.py

```
import urllib.request
res = urllib.request.urlopen('http://impress.co.jp/')
htmlbuf = res.read(300)
res.close()
print(htmlbuf)
```

urllib.requestモジュールを読み込み、urlopen関数を使って http://impress.co.jp に接続しています。 res は**HTTPResponse**オブジェクトです。そのオブジェクトの

readメソッドで300バイトのデータを読み込み、htmlbufという変数に格納し、その内容を出力しています。

■実行結果

```
b'<!DOCTYPE html>\n<html lang="ja">\n<head>\n <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />\n <meta name="viewport" content="width=device-width, initial-scale=1.0, user-scalable=no, minimum-scale=1.0, maximum-scale=1.0">\n <meta name="referrer" content="unsafe-url">\n<title>\xe6\xa0\xaa\xe5\xbc\x8f'
```

NOTE macOS用公式インストーラでPython 3.6をインストールした場合、HTTPS通信ができないことがあります。その場合は「アプリケーション」→「Python 3.6」→「Install Certificates.command」を実行して証明書をインストールしてください。

コンソールの出力を見ると、最初に"b'"という文字がありますが、これは出力がバイナリデータであることを意味します。また、出力の終端あたりに「<title>\xe6\xa0\xaa\xe5\xbc\x8f」という文字がありますが、これは日本語文字をバイナリで出力しているためです。

なお、resはHTTPResponseオブジェクトで、最後にcloseする必要がありますが、次のようにwith命令を使うとcloseを明示的に呼ぶ必要がなく便利です。

また、read部分を書き換えて、バイナリデータをデコードして文字列に変換できるようにしています。

◎リスト urllib_basic1.py

```
import urllib.request
with urllib.request.urlopen('http://impress.co.jp/') as response:
```

```
    html = response.read().decode("utf-8")
    print(html)
```

　取得したページから文字列を抽出するような場合、このようにdecodeも同時に処理してしまうとよいでしょう。ちなみにバイト数を指定せずにread（）を実行すると、ページのすべてが読み込まれます。

HTTPのステータスコード301

HTTPでは、クライアントがリクエストを送り、サーバがそのリクエストに応えてレスポンスを返します。レスポンスには必ず「ステータスコード」と呼ばれる情報が含まれ、番号によって意味が規定されています。100番台が処理中、200番台が成功、300番台がリダイレクト、400番台がクライアント側のエラー、500番台がサーバ側のエラーとなります。

301は「ページが恒久的に移動した」ことを意味します。urllib_basic0.pyでは「'http://impress.co.jp/'」というアドレスからページを取得したように見えたかもしれません。しかしながら、実際には「https://www.impress.co.jp」というアドレスからページを取得しています。wwwが付与されているだけでなく、「http」から「https」になっていることに注意してください。

urlopen(url)を実行すると、引数で与えられたサーバにアクセスします。正常にページが取得できればそれで処理は終了しますが、今回のケースでは、サーバはHTTPのステータスコード301（移動先：http://www.impress.co.jp）を返してきました。これを受けて、urlopenは「www.」を付けて「http://www.impress.co.jp」としてアクセスしましたが、再度301（移動先：https://www.impress.co.jp）が返されてきました。最終的に「http」を「https」に替えて「https://www.impress.co.jp」にアクセスし、ホームページを取得しました。単にurlopenを1回呼び出しただけですが、実はその裏ではこのような処理が行われていました。

⊙サーバとのやり取り

➡156

URLをファイルに保存するには

urllib.requestモジュールのurlretrieve関数を使う

URLからファイルを取得できたら次はそれを保存してみましょう。Google Chart APIを使って、helloという文字列のQRコードを作成・ダウンロードして、qrcode.pngというファイルに保存する例を以下に示します。

◎リスト urllib_saveimg.py

```
import urllib.request
url = "http://chart.apis.google.com/chart?cht=qr&chs=300x300&chl=hello"
with urllib.request.urlopen(url) as img:
    with open("qrcode.png", 'wb') as file:
        file.write(img.read())
```

urllib.request.urlopenでURLを読み込み、その内容を**バイナリ形式'wb'**でファイルに保存しています。この方法でもかまいませんが、ファイルを保存するために特化した関数も用意されています。

◎関数　urlretrieve(url, file)

カテゴリ	**urllib.requestモジュール**
引数	**url：取得対象となるurl、file：保存先**
戻り値	**ローカルのファイル名とhttp.client.HTTPMessageオブジェクトのタプル**
説明	**urlの内容をfileに保存する**
使用例	**fname, headers = urllib.request.urlretrieve('http://python.org/', 'python.txt')**

この関数を使って先ほどの例を書き換えてみましょう。

◎リスト urllib_saveimg2.py

```
import urllib.request
url = "http://chart.apis.google.com/chart?cht=qr&chs=300x300&chl=hello"
urllib.request.urlretrieve(url, "qrcode2.png")
```

URLからQRコードを取得し、その内容をファイル"qrcode2.png"に保存しています。

モジュールの変遷

Pythonでネットワークにアクセスする方法を検索したものの、いろいろな情報が遍在するために混乱する人も少なくないと思います。その理由として、Python 2とPython 3でモジュール名やその構造が変わったことが考えられます。以下のようにPython 3ではモジュール構造が整理されました。ネット上の記事にはPython 2を前提としたものも少なくありません。特にネット上の記事の場合、記事の内容がPython 2と3のどちらを前提にしているのか、意識して読むことが大切です。

⦿主なモジュール

Python 2	Python 3
import urllib	import urllib.request, urllib.parse, urllib.error
import urllib2	import urllib.request, urllib.error
import urlparse	import urllib.parse
import robotparser	import urllib.robotparser

⦿主な関数、クラス

Python 2	Python 3
from urllib import FancyURLopener from urllib import urlencode from urllib import quote	from urllib.request import FancyURLopener from urllib.parse import urlencode from urllib.parse import quote
from urllib2 import Request from urllib2 import HTTPError	from urllib.request import Request from urllib.error import HTTPError

参照：www.diveintopython3.net/porting-code-to-python-3-with-2to3.html

参照 ： www.diveintopython3.net/porting-code-to-python-3-with-2to3.html

➡157

URLからJSON形式のデータを取得するには

jsonモジュールを使ってjson文字列をオブジェクトに変換する

Web-APIとは、インターネット上に公開されているAPIのことです。多くの会社や団体からさまざまなサービスが提供されています。Web-APIを使うといろいろなサービスを簡単に実現できます。

Web-APIではHTTPでリクエストをサーバに送信し、サーバから返信をもらいますが、返信フォーマットの主流がJSONです。以下は書籍を検索するWeb-APIを使った例です。

◎リスト web_api.py

```
import urllib.request 1
import json
url = 'https://www.googleapis.com/books/v1/volumes?q=id:JPNhCAAAQBAJ' 2
response = urllib.request.urlopen(url) 3
content = json.loads(response.read().decode('utf-8')) 4
print(content)
```

まず**1**でurllib.requestとjsonモジュールを読み込んでいます。**2**のurlは書籍検索のWeb-APIです。**3**のurlopenでページを取得し、**4**のreadメソッドでページを読み取り、decodeメソッドでデコードします。その結果を**json.loads**に渡しています。

NOTE macOS用公式インストーラでPython 3.6をインストールした場合、HTTPS通信ができないことがあります。その場合は「アプリケーション」→「Python 3.6」→「Install

Certificates.command」を実行して証明書をインストールしてください。

6-2 Webサーバとのやり取りの基本

本節ではWebサーバにアクセスする方法について解説します。たとえば、URLエンコードを行ってからアクセスする方法、POSTによるデータ送信などを取り上げます。

➡158

テスト用Webサーバを用意するには

pythonを「-m http.server ポート番号」オプションで起動する

Pythonにはもともとテスト用のWebサーバが用意されており、コマンドラインから以下のように入力するだけでWebサーバが8000番ポートで起ち上がります。

◎実行例

```
>python -m http.server 8000 Enter
Serving HTTP on 0.0.0.0 port 8000 ...
```

あとは、ブラウザから「http://localhost:8000/」とアクセスすれば、コマンドを起動したディレクトリにあるファイル一覧が表示されます。ファイルの取得テストには十分なサーバですが、しかしサーバの中で何が起きているのかはわかりません。そこで、どんなリクエストがきたか表示する簡単なWebサーバを作ってみました。

◎リスト test_web_server.py

```
import threading
from socketserver import ThreadingMixIn
from http.server import BaseHTTPRequestHandler, HTTPServer
1

class TestServer(BaseHTTPRequestHandler): 2
    def handle_headers(self): 3
        for k, v in self.headers.items(): 4
            print(k , ":" ,v)
        self.rfile.close()
        self.send_response(200) 5
        self.end_headers()

    def do_GET(self): 6
        self.handle_headers()
        self.wfile.write(bytes(self.path, "utf-8")) 7
    def do_POST(self): 8
        content_length = int(self.headers['Content-Length']) 9
        post_data = self.rfile.read(content_length) 10
        self.handle_headers()
        self.wfile.write(post_data) 11

class ThreadedHTTPServer(ThreadingMixIn, HTTPServer): 12
    pass

httpd = ThreadedHTTPServer(("127.0.0.1", 8080), TestServer) 13
try:
    httpd.serve_forever() 14
except KeyboardInterrupt:
    pass
httpd.server_close()
```

まず 1 で必要なモジュールを読み込んでいます。BaseHTTPRequestHandlerは基本的な機能を持つHTTPサーバ

クラスで、カスタマイズしたい処理をオーバライドして使用します。

2でTestServerというBaseHTTPRequestHandlerを継承したクラスを作成しています。このテスト用WebサーバはHTTPヘッダの内容を出力しますが、その処理を行うために**3**のhandle_headersメソッドを定義しています。**4**でHTTPヘッダの内容（＝headersの値）をfor文を使って取り出し、printで出力しています。

5では応答メッセージのヘッダを返しています。

6はGETを処理するメソッドです。handle_headersでヘッダの内容を出力し、**7**で接続に使われたパスの内容を応答メッセージとして出力しています。

8がPOSTを処理するメソッドです。**9**でコンテンツの長さをContent-Lengthから取得し、**10**でその長さ分を入力ストリームrfileからreadしています。あとはhandle_headersメソッドでHTTPヘッダを出力し、**11**で受領したデータを送り返しています。

一度張ったコネクションを再利用するなど、最近のブラウザは高速化のためにさまざまな工夫をこらしています。そのような状況にも対応できるよう、**12**でHTTPサーバがマルチスレッドで動作するようにThreadingMixInを利用しています。

13でThreadedHTTPServerを作成し、**14**で実行を開始しています。

このテスト用サーバは以下のように実行することができます。

◎実行例

```
>python test_web_server.py Enter
```

終了するためにはCtrl+Cキーを押下してください。Ctrl+Cで終了しない場合はブラウザを終了する、もしくはコマンドを実行したウインドウを閉じてください。なお、macOSの終了キーは ＋．(ドット)キーとなります。

➡159
特殊文字をURLで使うには

quote、unquoteでURLエンコード、URLデコードを行う

URLに使える文字は制限されており、規定された文字以外はUTF-8で符号化したうえで%XX（XXは16進数）という形式に変換する旨が規定されています（https://ja.wikipedia.org/wiki/パーセントエンコーディング）。この変換は「**URLエンコード**」と呼ばれます。この変換を行う関数が**quote**、元に戻す関数が**unquote**です。

NOTE Webブラウザによってはアドレス欄にURLエンコードがデコードされた状態でアドレスが表示されます。たとえば「https://ja.wikipedia.org/wiki/パーセントエンコーディング」は「 https://ja.wikipedia.org/wiki/%E3%83%91%E3%83%BC%E3%82%BB%E3%83%B3%E3%83%88%E3%82%A8%E3%83%B3%E3%82%B3%E3%83%BC%E3%83%87%E3%82%A3%E3%83%B3%E3%82%B0 」と表示されることがあります。

◎関数　**quote(string)**

カテゴリ　**urllib.parseモジュール**

引数	変換対象の文字列
戻り値	URLエンコードされた文字列
説明	引数の文字列をURLエンコードした文字列を返す
使用例	r = urllib.parse.quote("あいう")

◎関数　unquote(string)

カテゴリ	urllib.parseモジュール
引数	変換対象の文字列
戻り値	URLエンコードされる前の文字列
説明	URLエンコードを元の文字列に戻す
使用例	r = urllib.parse.unquote("%E3%81%82%E3%81%84%E3%81%86")

◎リスト urllib_parse1.py

```
import urllib.parse
original = 'hi!はい'
quoted = urllib.parse.quote(original)
print("quote   : {0}->{1}".format(original, quoted))
unquoted = urllib.parse.unquote(quoted)
print("unquote: {0}->{1}".format(quoted, unquoted))
```

◎出力例

```
quote   : hi!はい->hi%21%E3%81%AF%E3%81%84
unquote: hi%21%E3%81%AF%E3%81%84->hi!はい
```

"はい"という文字列はUTF-8にすると0xE3,0x81,0xAF,0xE3,0x81,0x84となります。"!"は0x21です。これらの文字がURLエンコードの変換対象となるので、"hi!はい"をquoteでURLエンコードすると"hi%21%E3%81%AF%E3%81%84"となります。unquote

はその逆変換をする関数なので、元の"hi!はい"に戻っています。

➡160
辞書型データをURLエンコードするには

urlencodeを使って複数のパラメータをURLエンコードする

quote、unquoteを使うと、単一の文字列をURLエンコードできます。Webサーバにパラメータを送信するときには、複数の「**パラメータ＝値**」の組を送ることも少なくありません。このようなデータを保持するのには辞書型データが適していますが、**urlencode**は辞書型データに対して一括してURLエンコードを行い、URLとして送信できる単一の文字列に変換します。

◎関数　urlencode(query)

カテゴリ	**urllib.parseモジュール**
引数	**辞書型などのマッピングオブジェクト**
戻り値	**辞書のパラメータをURLエンコードして連結した文字列を返す**
説明	**引数で渡されたオブジェクトをKey=Valueの文字列にして返す。その際、日本語などのURL規定外の文字はurlencode（UTF-8で符号化し、%XXの形式に変換）して返す。それぞれのKey=Valueペアは&文字で連結される**
使用例	**r = urllib.parse.urlencode({'ja': "はい", 'en': 'Yes'})**

◎リスト urllib_parse2.py

```
import urllib.parse
s = urllib.parse.urlencode({'ja': "はい", 'en': 'Yes'})
print(s)
```

■出力例

```
ja=%E3%81%AF%E3%81%84&en=Yes
```

「ja=はい」と「en=Yes」という2つのパラメータを送る例です。"はい"という文字列はUTF-8で符号化すると、0xE3、0x81、0xAF、0xE3、0x81、0x84となります。よって、ja=%E3%81%AF%E3%81%84となります。一方、英語のYesはエンコード不要です。それらの値が＆で連結されていることがわかります。この関数を使うとGETによるデータ送信が簡単に実装できます。

◎リスト urllib_get0.py

```
import urllib.request
import urllib.parse
url = "http://localhost:8080/test?"
param = urllib.parse.urlencode({'ja': "はい", 'en': 'Yes'})
with urllib.request.urlopen(url + param) as res:
    print(res.read().decode('utf8'))
```

urlencodeで辞書型のデータをURLエンコードし、urlopenを使ってサーバに接続しています。test_web_server.pyを動作した状態でurllib_get0.pyを実行すると、サーバのコンソールに以下のような出力が表示されます。

■出力例

```
127.0.0.1 - - [10/Sep/2017 11:21:39] "GET /test?ja=%E3%81%AF%E3%81%84&en=Yes HTTP/1.1" 200 -
```

このように、ja=%E3%81%AF%E3%81%84とurlencodeされた文字列がGETでサーバに届いていることがわかります。

➡161

POSTでデータを送信するには

urlopenにRequestオブジェクトを渡す

GETでデータを送る場合、その**データはURLの一部**として送信されます。一方、**POST**でデータを送るとデータは**HTTPのボディ**としてサーバに送られます。GETでデータを送るときには、送信するデータをURLエンコードしましたが、POSTの場合は、別の方法を使用します。

◎コンストラクタ　Request(url, data)

カテゴリ	**urllib.requestモジュール**
引数	**url：サーバのURL、data：データ転送するバイト列、ファイルオブジェクト**
戻り値	**Requestオブジェクト**
説明	**urlで示されるサーバへPOSTメソッドでデータを送信する**
使用例	**req = urllib.request.Request(url, buf)**

◎リスト urllib_post0.py

```
import urllib.request
url = "http://localhost:8080/test"
```

```
buf = "Hello World".encode() 1
req = urllib.request.Request(url, buf) 2
with urllib.request.urlopen(req) as res: 3
        print(res.read().decode())
```

1のbufは「Hello World」をバイト列にエンコードしたものです。2では、urlとbufを引数にRequestオブジェクトを作成しています。そのオブジェクトを3でurlopenの引数に渡しています。この例では、URL以外に送るべきデータがあるため、GETではなくPOSTが使用されます。テストサーバでのコンソール出力は以下のようになります。POSTと出力されていることが確認できます。

NOTE http.serverモジュールはデフォルトでPOSTに対応しておりません。POSTやHTTPヘッダの挙動を確認するには「158　テスト用Webサーバを用意するには」のtest_web_server.pyをご利用ください。

■出力例

```
Accept-Encoding : identity
Content-Type : application/x-www-form-urlencoded
Content-Length : 11
Host : localhost:8080
User-Agent : Python-urllib/3.6
Connection : close
127.0.0.1 - - [17/Jan/2018 02:35:04] "POST /test HTTP/1.1" 200 -
```

urlopenでのGETとPOST

urlopenではGETとPOSTの両方が使われます。URL文字列とは別に何らかのデータを送信する場合にはPOSTが使われます。

```
1 r = urllib.request.urlopen("http://localhost:8080/") ➡GET
```

```
2 r = urllib.request.urlopen("http://localhost:8080/",
  "hello".encode()) ➡POST
```

```
3 req = urllib.request.Request("http://localhost:8080/")
  r = urllib.request.urlopen(req) ➡GET
```

```
4 req = urllib.request.Request("http://localhost:8080/",
  "hello".encode())
  r = urllib.request.urlopen(req) ➡POST
```

1と**2**がurlopenに文字列を渡した場合、**3**と**4**がRequestオブジェクトを渡した場合です。**2**と**4**では"hello”という文字列を送信していますが、これらの場合にPOSTが使用されます。

➡162

HTTPヘッダを指定するには

Requestオブジェクトのadd_headerメソッドを使用する

HTTPはヘッダとHTTP本体から構成されます。HTTPヘッダを自分で追加する必要性に迫られることも少なくありませ

ん。そんなときは、**Request**オブジェクトの**add_header**メソッドを使用します。

◎メソッド　add_header(key, val)

カテゴリ	**urllib.requestモジュールのRequestクラス**
引数	**key：ヘッダ名、val：ヘッダの値**
戻り値	**なし**
説明	**HTTPヘッダに追加する**
使用例	**req.add_header('Content-Type', 'application/json')**

◎リスト urllib_addheader.py

```
import urllib.request

req = urllib.request.Request("http://localhost:8080/")
req.add_header("X-MY-HEADER", "python_standard_library")
with urllib.request.urlopen(req) as res:
    html = res.read().decode("utf-8")
```

Requestオブジェクトを作成し、add_headerメソッドでHTTPヘッダを追加します。テスト用のサーバに接続すると、ヘッダが追加されていることが確認できます。

■出力例

```
Accept-Encoding : identity
Host : localhost:8080
User-Agent : Python-urllib/3.6
X-My-Header : python_standard_library
Connection : close
127.0.0.1 - - [10/Sep/2017 22:12:21] "GET / HTTP/1.1" 200 -
```

NOTE このサンプルはローカルのWebサーバと通信をします。「158　テスト用Webサーバを用意するには」の

test_web_server.pyを動かした状態でサンプルを実行してください。

➡163
HTTPやHTTPSで通信を行うには

http.client.HTTPConnection、HTTPSConnectionを使用する

Requestを使ってHTTP(s)通信を行う方法を紹介してきましたが、urllib.requestモジュールはその中で**http.client**モジュールを使っています。ここでは、直接http.clientを使って通信を行う方法を紹介します。

◎コンストラクタ　HTTPConnection(host)

カテゴリ	**http.clientモジュール**
引数	**ホスト名**
戻り値	**HTTPConnectionオブジェクト**
説明	**ホストと通信するためのオブジェクトを作成する。httpsで通信する際にはHTTPSConnectionオブジェクトを作成する**
使用例	**conn = http.client.HTTPConnection("localhost", 8080)**

◎リスト httpconnection.py

```
import http.client
conn = http.client.HTTPConnection("localhost", 8080)

headers = {
      "X-MY-HEADER": "python_standard_library"
}
body = "{'hello':'world'}"
```

```
conn.request("POST", "/", body, headers)
response = conn.getresponse()
data = response.read()

print(data)
```

HTTPConnectionオブジェクトを作成します。HTTPヘッダは連想配列の形で指定します。HTTPConnectionオブジェクトのrequestメソッドで、メソッド（POST、GETなど）、パス、ボディ、ヘッダなどの情報を渡し、getresponseでサーバとの通信を行います。サーバからの応答は、レスポンスのreadメソッドを呼び出して取得しています。

NOTE このサンプルはローカルのWebサーバと通信をします。「158　テスト用Webサーバを用意するには」のtest_web_server.pyを動かした状態でサンプルを実行してください。

Requestsモジュール

ここまで標準モジュールを使ってネットワークにアクセスする方法について説明してきました。実際には、標準モジュールではなく**Requests**モジュールを使う人のほうが多いかもしれません。Requestsモジュールは標準ライブラリに満足できない人が実装しただけあって使用方法がシンプルです。

GETでページを取得するコードは以下のようになります。

◎**リスト requests_get.py**

```
import requests
res = requests.get('http://localhost:8080/test')
print(res.status_code)
print(res.text)
```

POSTのコードは以下のようになります。

◎**リスト requests_post.py**

```
import requests
res = requests.post('http://localhost:8080/',
      data={'ja': "はい", 'en': 'Yes'})
print(res.status_code)
print(res.text)
```

どちらも標準ライブラリを使ったときよりも直観的な記述になっています。ただし、Requestsは標準ライブラリではないので「pip install requests」を実行してパッケージを追加する必要があることに注意してください。

NOTE 公式のドキュメントにもRequestsモジュールの使用を推奨する記載があります。
http://docs.python-requests.org/en/master/

➡164

ブラウザを起動するには

webbrowserモジュールのopen関数で開きたいページURLを指定する

Webブラウザを起動するには**webbrowser**モジュールの**open**関数を使います。

◎関数　open(url)

カテゴリ	**webbrowserモジュール**
引数	**ページのURL**
戻り値	**なし**
説明	**デフォルトのブラウザを起動して指定されたURLのページを開く**
使用例	**webbrowser.open("http://impress.co.jp")**

◎リスト webbrowser0.py

```
import webbrowser
webbrowser.open("http://impress.co.jp")
```

●新しいタブで開く

すでにブラウザが起動しているときには、新しいタブを開いてそこにページを表示することもできます。

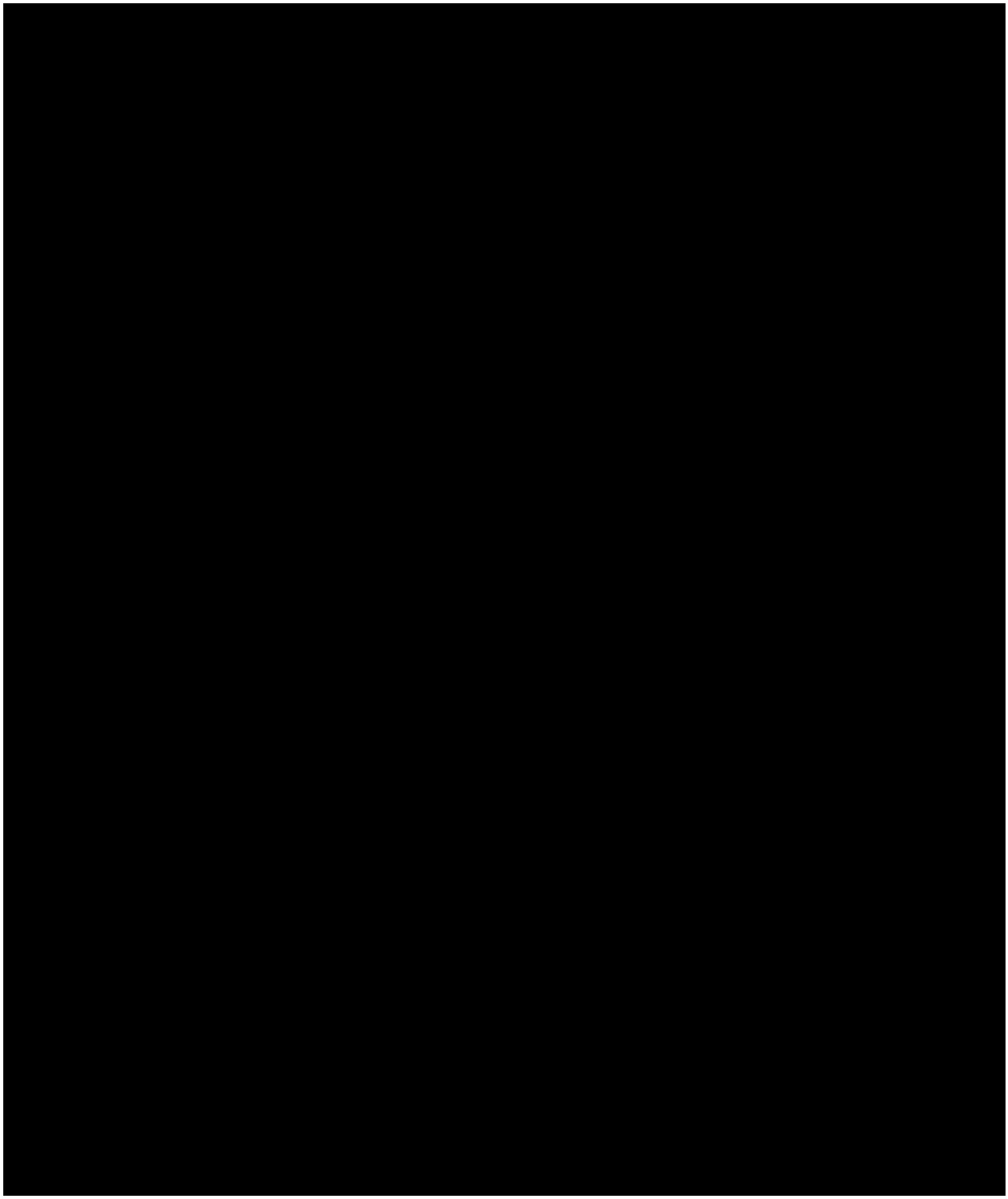
◎関数　open_new_tab(url)

カテゴリ	**webbrowserモジュール**
引数	**ページのURL**
戻り値	**なし**
説明	**新しいタブで指定されたURLのページを開く**
使用例	**webbrowser.open_new_tab("http://impress.co.jp")**

◎リスト webbrowser1.py

```
import webbrowser
webbrowser.open_new_tab("http://impress.co.jp")
```

Chapter 7



描画とGUI

ここまでコマンドラインから実行するプログラムについて説明してきました。この章ではGUI（Graphical User Interface）について説明します。Pythonの標準モジュールにはturtleやtkinterといったGUIモジュールが含まれています。turtleは主にプログラミング学習用に、tkinterは簡易GUI作成用に用いられます。

7-1 Turtleグラフィックスを使用する

turtleモジュールでは、ペンの上下、前進後退、右折左折といった基本的な操作を通して描画を行います。その内容は、プログラミングの基礎を学ぶのに適しています。

➡165

基本的な描画を行うには

forwardで前進、leftとrightで向きを変える

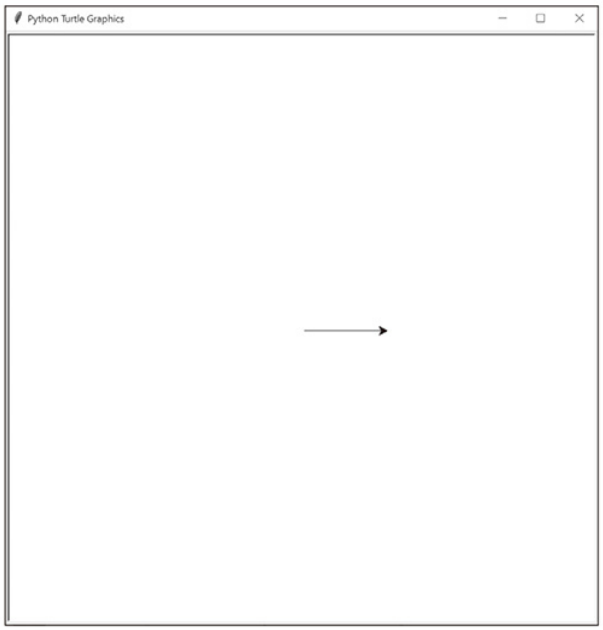
Turtleグラフィックスの基本に慣れるには、Pythonコマンドで対話的に操作するのがよいでしょう。

◎実行例

```
>>> from turtle import * Enter
>>> forward(100) Enter
```

この2行を入力するだけで、ウィンドウが現れて右方向に矢印（カーソル）が描画されます。

⦿出力結果

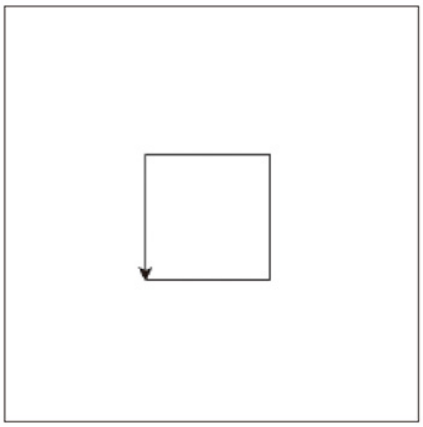


このあとに、90度左折するleft(90)、さらに100進むforward(100)、と繰り返し実行すると四角形を描画できます。**forward**は**fd**と記述することも可能です。

◎実行例

```
>>> reset() Enter
>>> forward(100) Enter
>>> left(90) Enter
>>> forward(100) Enter
>>> left(90) Enter
>>> forward(100) Enter
>>> left(90) Enter
>>> forward(100) Enter
```

⦿出力結果



◎関数　forward(distance)

カテゴリ	turtleモジュール
引数	距離
戻り値	なし
説明	カーソルが向いている方向にdistanceの距離分を進む
使用例	forward(100)

◎関数　left(angle)

カテゴリ	turtleモジュール
引数	角度
戻り値	なし
説明	カーソルの向きをangleの角度で左に回す
使用例	left(90)

主な命令を次に列挙します。

⊙基本的な描画を行うための主な関数

関数	説明	例
forward(x)、fd(x)	前進する(x:距離)	forward(100)
backward(x)、bk(x)、back(x)	後退する(x:距離)	backward(50)
right(x)、rt(x)	右折する(x:角度)	right(90)
left(x)、lt(x)	左折する(x:角度)	left(-60)
home()	ホームポジションに戻る	home()
circle(x)	円を描く(x:半径)	circle(50)
speed(x)	スピードを変える(x:0～10)	speed(10)
setx(x)	x座標をセットする	setx(100)
sety(y)	y座標をセットする	sety(200)
pendown()、pd()、down()	ペンを下げる	down()
penup()、pu()、up()	ペンを上げる	up()
pensize(x)、width(x)	ペンの太さ	width(3)
colormode(x)	色の範囲を1か255のどちらかに指定	colormode(255)
pencolor(r, g, b)	ペンの色を変える	pencolor(0,0,255)
fillcolor(r, g, b)	塗りつぶし色を変える	fillcolor(10,20,50)

ペンの現在位置を表すカーソルはデフォルトでは矢印の形状ですが、**shape**関数を使うことでカーソルの形状を変更でき、**shapesize**でカーソルのサイズを変更することができます。

⊙カーソルの形状

arrow	turtle	circle	square	triangle	classic
					

◎実行例

```
>>> from turtle import * Enter
```

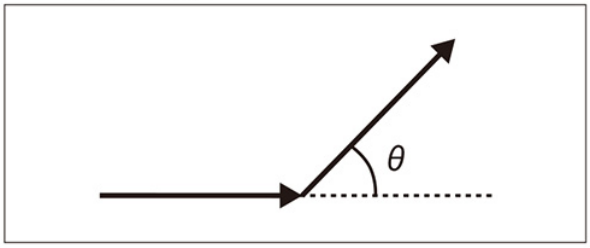
```
>>> shape("turtle") Enter
```

➡166
多角形の描画を行うには

「fdで進み、left、もしくはrightで曲がる」という処理を繰り返す

多角形を描画する場合、すべての頂点で同じ角度で曲がる必要があります。

◉多角形の角度



n角形の場合、各頂点で曲がる角度θは、360度÷nとなります。あとは、前進して曲がってという処理をn回繰り返せば多角形が描画できます。以下の例ではペンの上げ下げに**up**と**down**を、ペンの移動に**setpos**を使っています。

◎関数　up()

カテゴリ	**turtleモジュール**
引数	**なし**
戻り値	**なし**
説明	**ペンを上げる**
使用例	**up()**

◎関数　down()

カテゴリ	**turtleモジュール**
引数	**なし**
戻り値	**なし**
説明	**ペンを下げる**
使用例	**down()**

◎関数　setpos(x, y)

カテゴリ	**turtleモジュール**
引数	**x座標、y座標**
戻り値	**なし**
説明	**ペンをx, y座標の場所に移動する**
使用例	**setpos(100, 200)**

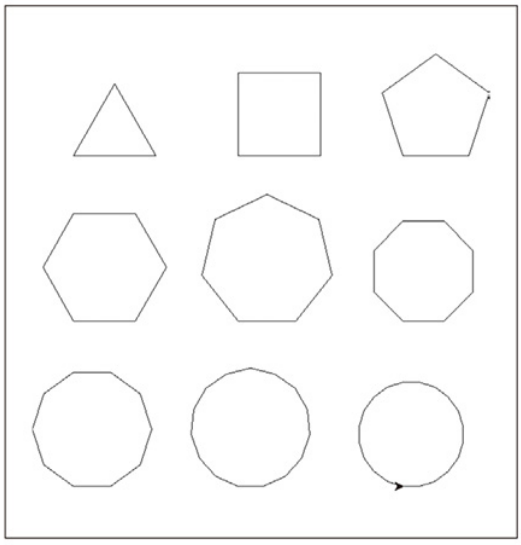
◎リスト turtle_polygon.py

```
import turtle as t
from math import sin, cos, radians

def draw_poly(x, y, s, n): 1
    t.up() 2
    t.setpos(x, y)
    t.down()
    for _ in range(n): 3
        t.fd(s) 4
        t.left(360/n) 5

draw_poly(-200, 200, 100, 3)
draw_poly(0, 200, 100, 4)
draw_poly(200, 200, 80, 5)
draw_poly(-200, 0, 75, 6)
draw_poly(0, 0, 70, 7)
draw_poly(200, 0, 50, 8)
draw_poly(-200, -200, 45, 10)
draw_poly(0, -200, 30, 15)
draw_poly(200, -200, 20, 20)
```

```
t.done()
```

今回のリストではdraw_poly(x, y, s, n)関数を使って多角形を描画しています。関数は**1**で定義されています。引数のx、yは図形を描き始める場所、sは1辺のサイズ、nは頂点の数です。**2**でペンを上げて、setposでペンを移動します。そのあとでdownで書き始めます。**3**でn角形分のループを開始します。**4**で1辺分進んで、**5**において、各頂点で曲がる処理を行います。このような処理で多角形を簡単に描画することができます。

⊙出力結果



➡167

幾何学模様を描画するには

2重ループを使って、多角形を回転して描画する

直前の項では1重ループを使って多角形を描画しました。2重ループを使うといろいろな幾何学模様が簡単に描画できま

す。

◎関数　colormode(cmode)

カテゴリ	turtleモジュール
引数	1か255のどちらかの値
戻り値	なし
説明	色指定をする際、最大値を1にするか255にするか選択する
使用例	colormode(1)

◎関数　pencolor(r, g, b)

カテゴリ	turtleモジュール
引数	r：赤、g：緑、b：青
戻り値	なし
説明	ペンの色を設定する。colormodeに応じて0～1もしくは0～255の範囲で指定
使用例	pencolor(255,255,0)

◎リスト turtle_pattern.py

```
from turtle import *
from math import sin, cos, radians

def draw_poly(x, y, s, n): 1
    up()
    setpos(x, y)
    down()
    for _ in range(n):
        fd(s)
        left(360/n)

speed(0) 2
colormode(1) 3

for i in range(90): 4
    rad = radians(i*4) 5
```

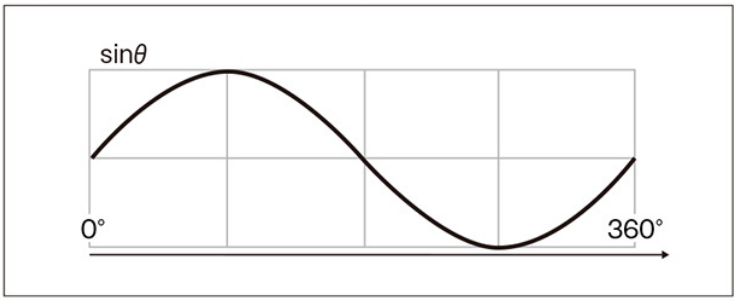
```
        r = (sin(rad) + 1) / 2 6
        g = (cos(rad) + 1) / 2 7
        b = (sin(rad*2) + 1) / 2 8
        pencolor(r, g, b) 9
        draw_poly(0, 0, 100, 4) 10
        right(4) 11

done()
```

1の関数は前項で使ったものをそのまま流用しています。**2**で描画のスピードを最大にしています。**speed**の引数は1が最も遅く、10が速くなります。0はアニメーション効果をなくす指定で、最も描画が速くなります。**3**でカラーモードを設定します。引数は1か255のどちらかです。色を設定するときのパラメータの範囲を、0～1にするか0～255にするかを指定します。今回は三角関数を使って色の値を指定するため引数に1を渡しています。

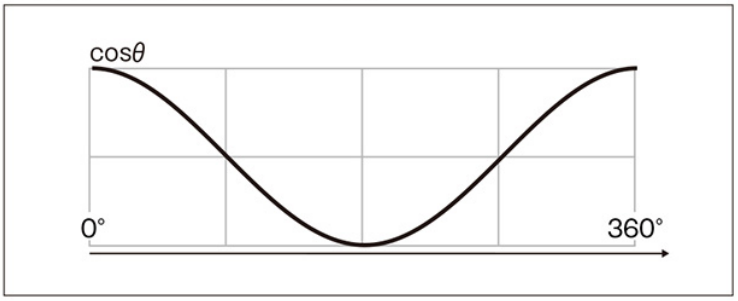
4でfor文を90回繰り返しています。**5**では0～360度の角度をラジアンに変換しています。**6**～**8**でR（赤）、G（緑）、B（青）の色を指定しています。**9**でR、G、Bの値を指定し、**10**で一辺が100ピクセルの四角形を描画しています。**11**で角度を4度右にずらしてfor文を繰り返します。このようにすることで、四角形が4度ずつずれた幾何学模様が描画されます。

ここで色の指定に三角関数を使った理由について説明しましょう。今回の模様は1回転、すなわち360度移動しながら色を変化させますが、つなぎ目を目立たなくするためには、最初と最後の色味をスムーズにつなげる必要があります。この用途には三角関数が適しています。sin、cosどちらも360度で元の値に戻るからです。sin、cosともに値の取りうる範囲は-1から+1です。これを0～1の範囲にするために、1を加えて2で割るという計算をしました。

⦿赤色の描画（前掲リスト6）

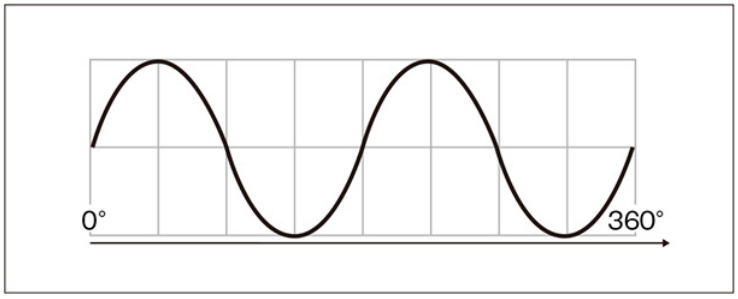


⦿緑色の描画（前掲リスト7）

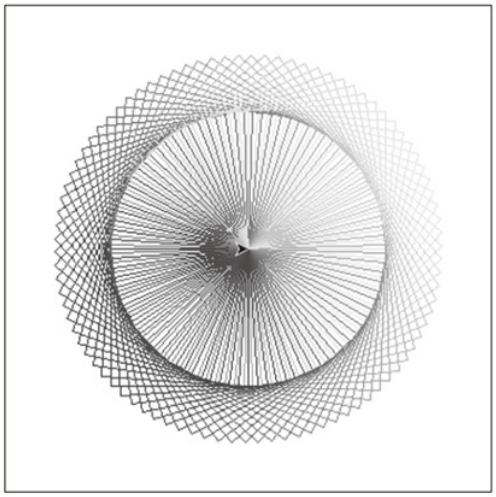


　青色に関してはさらに変化をつけるため、角度radを2倍にすることで周期を2倍にしています。

⦿青色の描画（前掲リスト8）



⊙出力結果



以下のリストはクリックした場所に星を描画します。

リスト: turtle_star.py

```
import turtle as t
def star(x, y):
	t.up()
	t.setpos(x,y)
	t.down()
	for _ in range(5):
		t.forward(100)
		t.right(144)
t.onscreenclick(lambda x, y: star(x, y))
t.done()
```

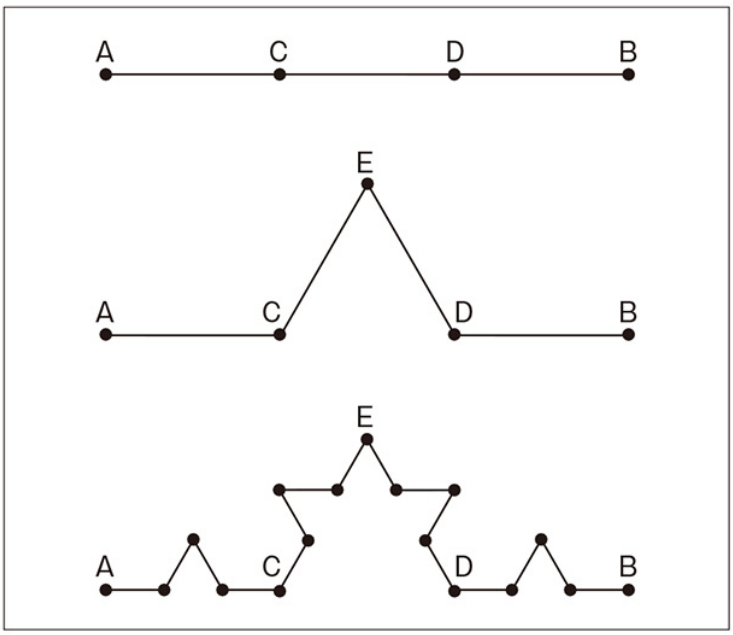
➡168
フラクタルを描画するには

再帰関数を使う

図形の一部分を取り出しても、それが全体と似ている図形を「**フラクタル**」といいます。Turtleモジュールを使ってフラクタル図形を描画してみましょう。

線分ABがあり、それを三等分する点CとDがあったとします。CDを一辺とする正三角形を描き、その三角形の新たな頂点をEとします。さらに線分AC、CE、ED、DBに対しても、同じように三等分して正三角形を描画すると以下のようになります。

⊙三等分して正三角形を描画していく



これをずっと繰り返していくと面白い図形ができあがります。

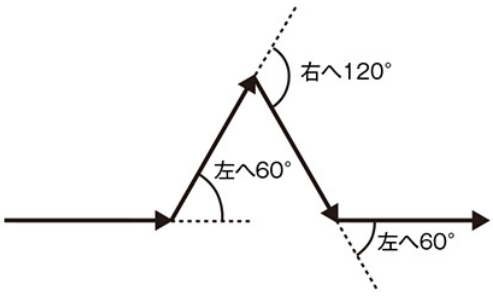
◎リスト turtle_fractal.py

```
import turtle as t

def fractal_line(size, depth): 1
    if depth == 0: 2
        t.fd(size)
```

```
        else:
            fractal_line(size/3, depth-1) 3
            t.left(60) 4
            fractal_line(size/3, depth-1) 5
            t.right(120) 6
            fractal_line(size/3, depth-1) 7
            t.left(60) 8
            fractal_line(size/3, depth-1) 9
t.speed(0) 10
t.up() 11
t.setx(-200)
t.sety(200)
t.down()
N = 4
fractal_line(400, N) 12
t.right(120)
fractal_line(400, N) 13
t.right(120)
fractal_line(400, N) 14
t.done()
```

ある線分が与えられて、それを 3 等分して正三角形を描く場合、turtleモジュールを利用したコードは以下のようになります。

⊙線分を3等分して正三角形を描く




```
fd(size/3)
left(60)
fd(size/3)
right(120)
fd(size/3)
left(60)
fd(size/3)
```

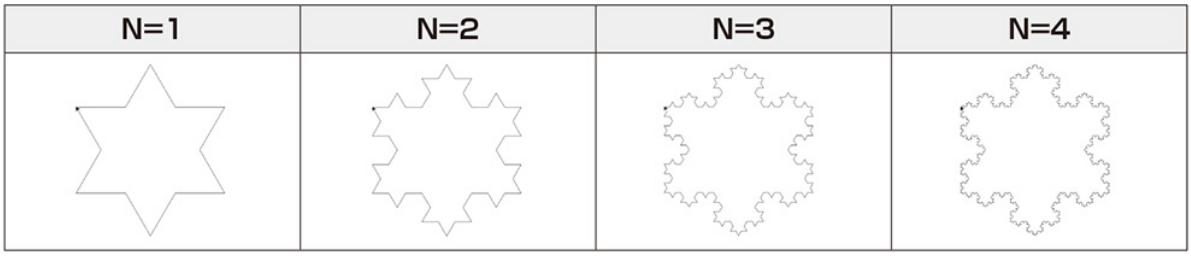
「1/3進んで左へ60度、1/3進んで右に120度、1/3進んで左へ60度、最後に1/3進む」という具合です。ただ、これではフラクタルになりません。fdで1/3進む代わりに、自分自身の関数を呼び出すようにします。関数が自分自身を呼び出すことを「**再帰**」と呼びますが、fractal_lineは再帰関数にほかなりません。それでは詳しく見ていきましょう。

1にあるように、fractal_lineは1辺の長さsizeと深さdepthを引数にとります。depthは自分自身を呼び出すたびに減らしていき、0になった時点で再帰を終了します。**2**でdepthが0か調べ、0の場合は単にsizeの線を引きます。1以上の場合は**3**～**9**で、「1/3進んで左へ60度（または右へ120度）…」という処理を行いますが、進むときにfractal_lineを使います。depthの値を1減らしていることに注意してください。

10でスピードを最速にし、**11**でペンを上げて最初の描画座標を設定し、**12**～**14**でfractal_lineを使って三角形を描画しています。

Nの値を変化させていくと、模様が緻密になっていく様子が観察できます。このような図形を考案した数学者コッホにちなんで、「コッホ曲線（コッホ雪片）」と呼びます。

⊙出力結果



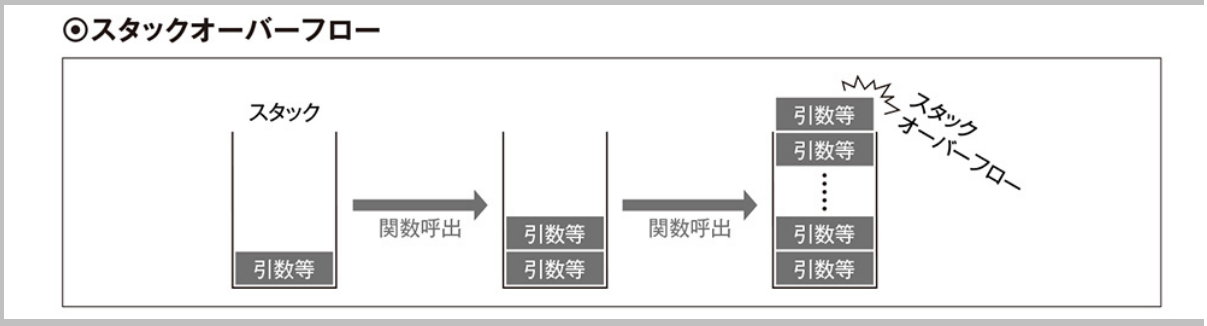
スタックオーバーフロー

再帰関数で自分自身を永遠に呼び出すとどうなるでしょうか？fractal_lineでdepthを減らさずに呼び出してみると、しばらく実行したあとに以下のようなエラーが表示され処理が終了します。

■実行結果

```
RecursionError: maximum recursion depth exceeded in comparison
```

一般的に、プログラムは関数を呼び出すときに、引数などの情報をスタックと呼ばれる記憶領域に格納します。関数をずっと呼び出し続けると、この領域をすべて占有してしまい、関数をこれ以上呼び出せなくなってしまいます。これがスタックオーバーフローという状態です。



普通にプログラムを組んでいるだけなら、スタックオーバーフローに直面することはあまりないでしょう。しかしながら、再帰関数を使用すると、少し間違えただけで自分自身を永遠に呼び続けるという状況に陥ってしまい、簡単にスタックオーバーフローを引き起こしてしまいます。今回は変数depthを使って、そのような状況を回避しました。

7-2 Tkinterでウィジェットをレイアウトする

GUIを構築する際には、まずウィンドウを生成し、そこに表示する部品（ウィジェット）を配置しなくてはなりません。TkではOSによって部品の大きさが微妙に変化するので、厳密な値で配置するのではなく、ざっくりと部品を並べるというアプローチのほうがよいでしょう。

➡169

ウィンドウを表示するには

Tkオブジェクトを作成し、mainloopメソッドを呼び出す

TkinterではTkオブジェクトを作成することで、アプリケーションのメインとなるトップレベルウィンドウを作成することができます。

NOTE TkinterとはTkをPythonから呼び出せるようにするためのモジュールです。TkはもともとTclというスクリプト言語用に開発されたGUI構築用のツールキットでしたが、手軽に使えるためファンも多く、現在ではいろいろな言語でサポートされています。

◎コンストラクタ　Tk()

カテゴリ	**tkinterモジュール**
引数	**なし**
戻り値	**Tkオブジェクト**
説明	**トップレベルのウィンドウを作成する**
使用例	**root = tk.Tk()**

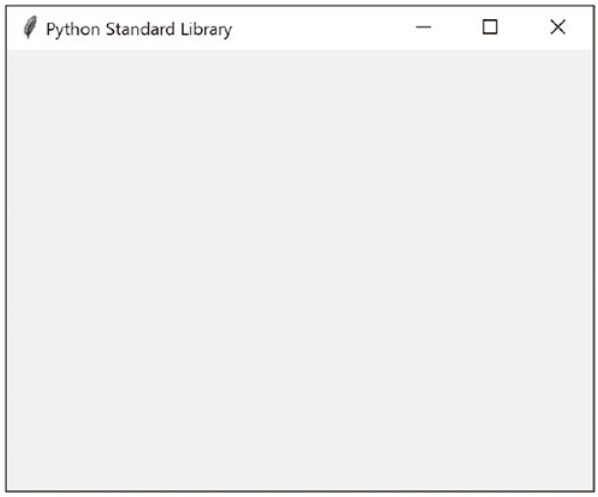
◎リスト tkinter_basic.py

```
import tkinter as tk
root = tk.Tk()
root.title("Python Standard Library") 1
root.geometry("400x300") 2
root.mainloop() 3
```

1では、作成されたオブジェクトのtitleメソッドを使ってウィンドウタイトルを設定しています。また、**2**のgeometryメソッドでウィンドウのサイズを設定しています。サイズ指定を省略したときは、子要素を格納するのに必要なサイズが自動的に計算されます。

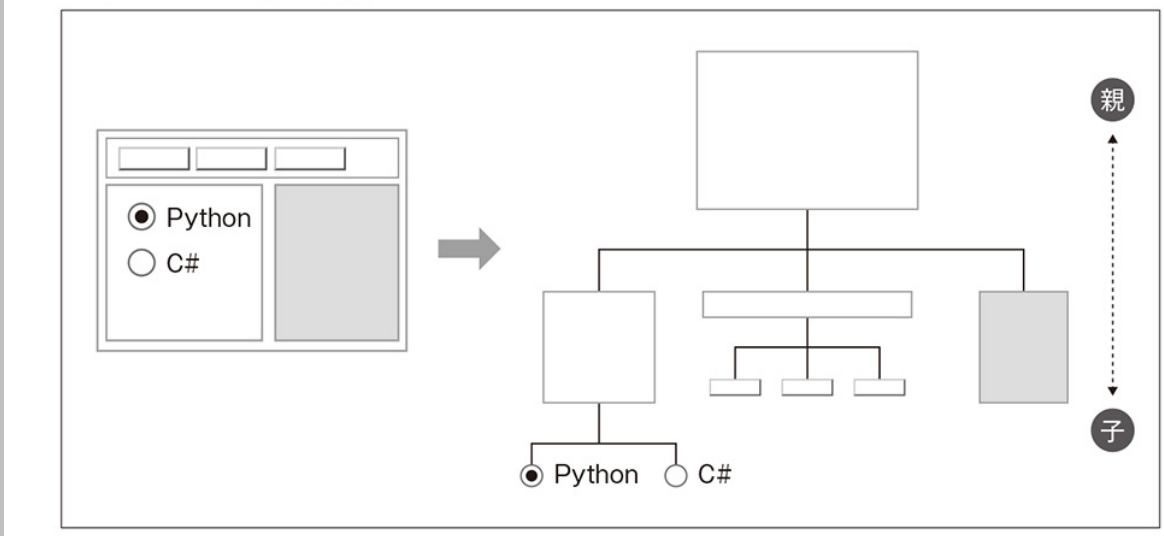
ウィンドウアプリケーションは、再描画を行ったり、ユーザーの操作に反応したりと、いろいろな事象に対応するために、メインループといわれるループ処理を実行するのが普通です。tkinterではトップレベルのウィンドウの**mainloop**メソッドを呼び出すことで、メインループを実行します**3**。

◉出力結果:ウィンドウが表示される



GUIの親子関係

tkinterにはボタン、ラベル、スクロールバー、キャンバスなどさまざまな部品が用意されており、それらを配置することでGUIを構築します。Tkではこれらの部品を「**ウィジェット**」と呼びます。ウィジェットは親子関係を形成します。



メインのウィンドウをつくって、その子要素として画面上部にメニューを配置し、メニューの子供にボタンを配置し、という具合です。Tkでは、ウィジェット作成時の最初の引数に親ウィジェットを指定することで親子関係を記述します。

➡170
縦方向（横方向）にボタンを配置するには

ウィジェットのpackメソッドを呼び出す

まずは最も基本的なウィジェットであるボタンを複数配置してみましょう。

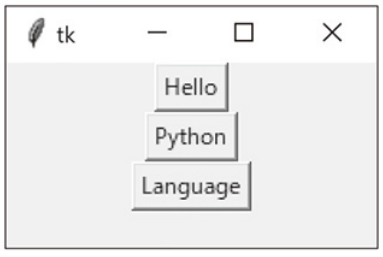
◎コンストラクタ　Button(parent, text=...)

カテゴリ	**tkinterモジュール**
引数	**parent：親ウィジェット、text：ボタンに表示する文字列**
戻り値	**Buttonオブジェクト**
説明	**親がparentであるButtonウィジェットを作成する**
使用例	**button0 = tk.Button(root, text="Hello")**

◎リスト tkinter_layout_basic0.py

```
import tkinter as tk
root = tk.Tk() 1
button0 = tk.Button(root, text="Hello") 2
button1 = tk.Button(root, text="Python") 3
button2 = tk.Button(root, text="Language") 4
button0.pack() 5
button1.pack() 6
button2.pack() 7
root.mainloop()
```

◎出力例



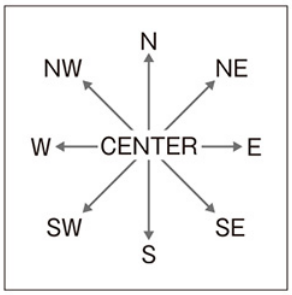
1でメインウィンドウを作成し、**2**～**4**でボタンを作成しています。第1引数がrootでメインウィンドウを指定しています。つまり、これらのボタンはメインウィンドウの直下に配置されます。Tkでは部品を作成しただけでは画面に表示されません。「親ウィンドウの中でどのように配置するか」を子供からリクエストを出す必要があります。そのリクエストを依頼するのが**5**～**7**の**pack**メソッドです。特に引数を指定しない場合、縦方向にレイアウトされます。

packメソッドには柔軟な配置を実現するために、さまざまな引数が用意されています。

◎メソッド　**pack(side=, fill=, anchor=, ...)**

カテゴリ	**tkinterモジュールの各種ウィジェットクラス**
引数	**名前付き引数sideがとり得る値は、TOP、BOTTOM、LEFT、RIGHTのどれか** **名前付き引数fillがとり得る値は、NONE、X、Y、BOTHのどれか** **名前付き引数anchorのとり得る値は、以下の図の通り**
戻り値	**なし**
説明	**side　どの方向から詰めていくかを指定する** **上から：TOP、左から：LEFT、右から：RIGHT、下から：BOTTOM** **fill　空きスペースがある場合に、どのように隙間を埋めるかを指定する** **横方向：X、縦方向：Y、縦横両方向：BOTH** **anchor　空きスペースがある場合に、どの方向に寄せて配置するかを指定する。東西南北（EWSN）を組み合わせて指定**
使用例	**button0.pack(side=tk.LEFT)** **button0.pack(fill=tk.X)** **button0.pack(anchor=tk.E)**

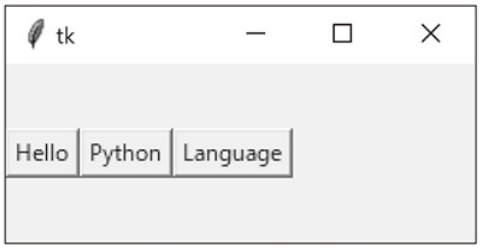
⦿引数anchorのとり得る値



◎リスト tkinter_pack0.py

```
import tkinter as tk
root = tk.Tk()
for txt in ("Hello","Python","Language"):
	b = tk.Button(root, text=txt)
	b.pack(side=tk.LEFT)
root.mainloop()
```

⦿出力結果



リスト：tkinter_pack1.py

```
import tkinter as tk
root = tk.Tk()
for txt in ("Hello","Python","Language"):
	b = tk.Button(root, text=txt)
	b.pack(fill=tk.X)
root.mainloop()
```

⦿出力結果

リスト：tkinter_pack2.py

```
import tkinter as tk
root = tk.Tk()
anchors = [tk.E, tk.W, tk.S]
texts = ["Hello","Python","Language"]
for i in range(3):
	b = tk.Button(root, text=texts[i])
	b.pack(anchor=anchors[i])
root.mainloop()
```

⦿出力結果

➡171
格子状にウィジェットを配置するには

ウィジェットのgridメソッドを使い、配置する行・列を指定する

電卓や表など、格子状に部品を配置する場合には**grid**レイアウトを使います。

◎メソッド　grid(row=, column=, padx=, pady=, ...)

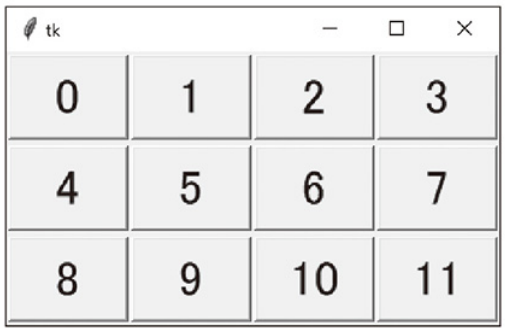
カテゴリ	**tkinterモジュールの各種ウィジェットクラス**
引数	**row：行、column：カラム、padx：横方向の余白、pady：縦方向の余白** **columnspan：横方向に何列にまたがって配置するか** **rowspan：縦方向に何行にまたがって配置するか**
戻り値	**なし**
説明	**行row、カラムcolumnの場所にウィジェットを配置する**
使用例	**button.grid(row=3, column=2)**

◎リスト tkinter_layout_grid0.py

```
import tkinter as tk
root = tk.Tk()
for i in range(12): 1
    button = tk.Button(root, text=i, width=4, font=("",25)) 2
    button.grid(row=i//4, column=i%4, padx=1, pady=2) 3
root.mainloop()
```

1で12回ループを回しています。**2**でボタンを作成しています。textにループの回数を指定し、ボタンの幅として**width**パラメータに4を指定。フォントの種類はデフォルトとし、そのサイズには25を指定しています。**3**が格子状にレイアウトする部分です。行を**row**で、列を**column**で指定します。//は切り捨ての除算、%は剰余を求める演算子です。これらの演算子を使って、ループカウンタのiから行と列の番号を求めています。padxは横方向、padyは縦方向の余白です。

⊙出力結果



➡172

位置を指定してウィジェットを配置するには

placeメソッドを使って位置指定を行う

　指定した位置と場所にウィジェットを配置するには**place**メソッドを使います。

◎メソッド　place(x, y, width, height, relx, rely, relwidth, relheight)

カテゴリ	**tkinterモジュールの各種ウィジェットクラス**
引数	**x：部品の左端の位置（ピクセル）、デフォルトは0** **y：部品の上端の位置（ピクセル）、デフォルトは0** **width：部品の幅、height：部品の高さ** **relx：部品の左端の位置（親に対する相対値0～1.0の実数）、デフォルトは0** **rely：部品の上端の位置（親に対する相対値0～1.0の実数）、デフォルトは0** **relwidth：部品の幅（親に対する相対値0～1.0の実数）** **relheight：部品の高さ（親に対する相対値0～1.0の実数）**

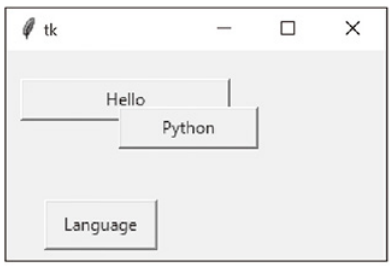
戻り値	**なし**
説明	**指定された場所にウィジェットを配置する（原点は左上隅）**
使用例	**button.place(x=10, y=20, width=150, height=30)**

◎リスト tkinter_layout_place0.py

```
import tkinter as tk
root = tk.Tk()
root.geometry("200x150")
button0 = tk.Button(root, text="Hello")
button1 = tk.Button(root, text="Python")
button2 = tk.Button(root, text="Language")
button0.place(x=10, y=20, width=150, height=30)
button1.place(x=80, y=40, width=100, height=30)
button2.place(relx=0.1, rely=0.6, relwidth=0.3, relheight=0.2)
root.mainloop()
```

button0とbutton1はピクセル単位の絶対値で場所と大きさを指定しています。button2は親に対する相対値で場所と大きさを指定しています。

◉出力結果



いろいろな設定を行うには

Tkinterではいろいろな**GUI**部品が用意されています。主なものを以下に列挙します。

⊙主なウィジェット

Button	LabelFrame	Radiobutton
Canvas	Listbox	Scale
Checkbutton	Menu	Scrollbar
Entry	Menubutton	Spinbox
Frame	Message	Text
Label	PanedWindow	

たくさんの部品が用意されていることがわかります。その上、それぞれの部品には数多くの設定内容があります。たとえば、今までボタンの作成は以下のように行ってきました。最初の引数は親ウィジェットを指定していますが、それ以外の引数にはキーワード引数を使用しています。

```
button = Button(root, text="Hello")
```

ボタンを作成する場合、文字列以外にも、フォント、色、ボーダー幅、余白、縦・横サイズ、コールバック関数などさまざまな設定項目があります。このような設定項目には数多くの種類があるので、すべてを順番通りに指定するのは現実的ではありません。

そこでTkinterではキーワード引数を使用します。たとえば以下のようになります。

```
button = Button(root, text=i, width=4, command=clicked)
button = Button(root, command=clicked, text=i)
```

このように名前付き引数を使用すれば、必要な項目のみを設定できるようになります。これはButtonだけでなく、ほかのウィジェットでも共通です。どのウィジェットも、コンストラクタ（ウィジェットを作る関数）の第1引数には親ウィジェットを指定し、それに続くオプション引数で詳細な指定を行います。

主なオプションに次のようなものがあります。ウィジェットによって指定できる内容は変わるので注意してください。以降、具体例の中で主要なウィジェットを作成していきます。

◉主なオプション

オプション	意味	Button	Frame	Label	Entry	Scale	Canvas
anchor	アンカー	○		○			
bg	背景色	○	○	○	○	○	○
bitmap	ビットマップ	○		○			
bd	ボーダー幅	○	○	○	○	○	○
command	コマンド	○				○	
font	フォント	○		○	○	○	
fg	前景色	○		○	○	○	
height	高さ	○	○	○			○
justify	左右の配置	○		○	○		
padx	横方向余白	○	○	○			
pady	縦方向余白	○	○	○			
relief	ボーダー装飾	○	○	○	○	○	○
text	テキスト	○		○			
width	幅	○	○	○	○	○	○

ウィジェットのkeys()オプションを使用するとオプション一覧が表示されます。この表は以下のプログラムを使って作成しました。

◎リスト tkinter_options.py

```
from tkinter import *
widgets = [Button, Frame, Label, Entry, Scale, Canvas]
options = ["anchor", "bg", "bitmap", "bd", "command",
      "font", "fg", "height", "justify", "padx", "pady",
      "relief", "text", "width"]

print(" " * 7, end='')
for w in widgets:
      print("{0:7}".format(w.__name__), end='')
print()

for o in options:
      print("{0:10}".format(o), end='')
      for w in widgets:
            flag = "o" if o in w().keys() else "-"
            print("{0:7}".format(flag), end='')
      print()
```

➡173

フォントを指定するには

ウィジェットのfont引数で指定する

ほとんどのウィジェットでフォントを指定できます。フォントの指定には**font**オプションを使用します。指定可能なパラメータは以下の通りです。フォントタイプを明示的に指定せず、デフォルトのフォントを使用する場合は空文字列を指定します。

⦿fontオプションのパラメータ

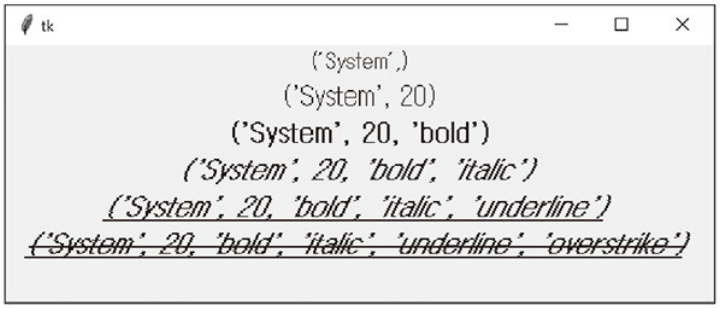
パラメータ	意味	省略	具体例
family	フォントタイプ	不可	"","System", "Helvetica", "FixedSys", …
size	サイズ	可	10, 15, …
weight	重さ	可	normal, bold
slant	斜体	可	roman, italic
underline	下線	可	normal, underline
overstrike	打消し線	可	normal, overstrike

◎リスト tkinter_font_labels.py

```
import tkinter as tk
root = tk.Tk()
params = ("System", 20, "bold", "italic", "underline", "overstrike")
for i in range(len(params)):
        s = params[0:i+1]
        tk.Label(root, text=str(s), font=s).pack(fill=tk.X)
root.mainloop()
```

params[0:i+1]で引数を順番に切り出しています。
i=0のときは("System",)、i=1のときは("System", 20)、i=2のときは("System", 20, "bold")、とパラメータの内容が徐々に増えていきます。

⦿出力例



どのようなフォントタイプが利用できるかを調べるには、tkinter.fontモジュールのfamilies関数を使用します。

◎リスト tkinter_font_list.py

```
import tkinter as tk
import tkinter.font
root = tk.Tk()
print(tkinter.font.families())
```

　現在の環境で利用できるフォント一覧が表示されます。ほかの環境でも意図したフォントを表示するには、一般的なフォント名を使うのがよいでしょう。

◉出力例

NOTE 多くのウィジェットではreliefオプションで3Dボーダーを描画できます。

◎リスト

```
import tkinter as tk
root = tk.Tk()
```

```
for r in (tk.SUNKEN, tk.RAISED, tk.GROOVE, tk.RIDGE,
tk.FLAT):
        tk.Label(root, text=r, relief=r,
                bd=5, font=("", 24)).pack(side=tk.LEFT)
root.mainloop()
```

⊙出力結果

➡174

複雑な配置をするには

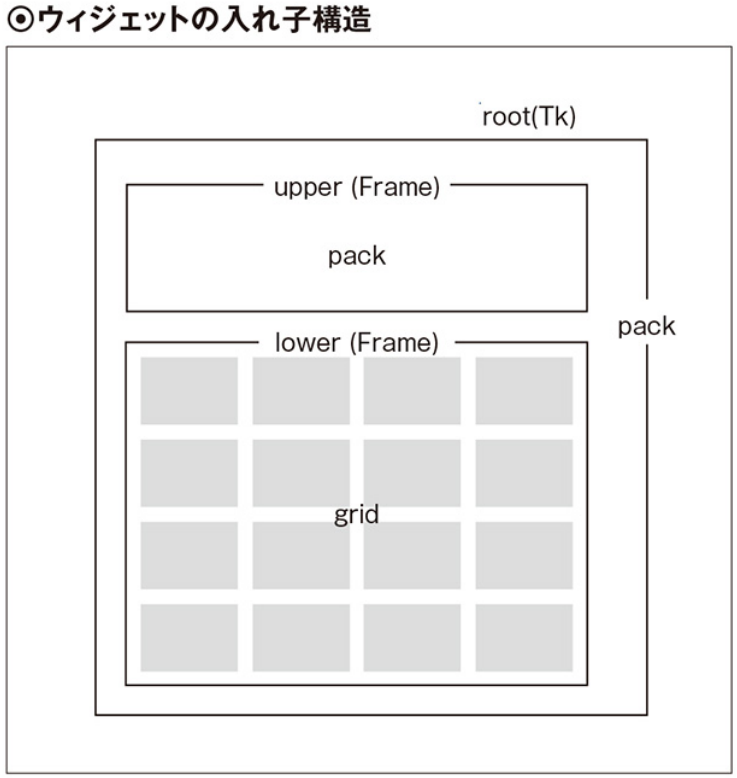
ウィジェットを使って入れ子構造を作り、それぞれに適切なレイアウトを設定する

ここまで説明したように、Tkでのレイアウトは、縦（横）方向に1列に配置する**pack**、格子状に配置する**grid**、任意の場所に配置する**place**の3つが用意されています。**1つの親ウィンドウの中にこれら複数のレイアウトを混在させることはできません。**よって、複雑なGUIを作るときには、**Frame**ウィジェットを入れ子にして、それぞれのFrameに別のレイアウトを適用することになります。

たとえば電卓を考えてみましょう。一般的な電卓では上部に結果表示領域があり、下部にボタンが格子状に並んでいます。このようにある程度複雑なレイアウトを実現したい場合、**root**の直下にすべての部品を配置するのではなく、ウィジェットを入れ子にすると配置が容易になります。

結果表示領域の**Entry**を含むFrameと、キー入力用のButtonを格子状に配置するFrameを用意し、rootは単にFrame2個を縦

方向に配置します。Frameやrootは**直下の子供を配置することに集中**できるので、FrameやRootでの管理がしやすくなります。



テキストを入力したり、表示したりするにはEntryウィジェットが便利です。以下のサンプルでは計算結果を表示するためにEntryウィジェットを使っています。

Entryにテキストを挿入するには**insert**メソッドを使用し、テキストの削除には**delete**メソッドを使用します。

◎メソッド **insert(index, string)**

カテゴリ	tkinterモジュール、Editクラス
引数	index：挿入する場所、string：挿入する文字列
戻り値	なし
説明	indexで指定した場所にstringで指定した文字列を挿入する
使用例	entry.insert(tk.END, "hello")

◎メソッド　delete(first, last=None)

カテゴリ	tkinterモジュール、Editクラス
引数	first：削除する文字の先頭位置 last：削除する文字の最後の位置、ENDとすると末端まで
戻り値	なし
説明	Entryウィジェットにおいてfirstからlastまでの範囲の文字を削除する
使用例	entry.delete(0, tk.END)

　また、以下のサンプルではEntryに入力された値を計算するためにeval関数（030で解説）を使用しています。

◎リスト tkinter_calc.py

```
import tkinter as tk

labels = ("7","8","9","*","4","5","6",
      "-","1","2","3","+","C","0",".","=") 1

def clicked(e): 2
      btn = e.widget["text"] 3
      if btn == "C":
            entry.delete(0, tk.END) 4
      elif btn == "=":
            result = eval(entry.get()) 5
            entry.delete(0, tk.END) 6
            entry.insert(tk.END, result) 7
      else:
            entry.insert(tk.END, btn) 8
root = tk.Tk()
upper = tk.Frame(root) 9
entry = tk.Entry(upper, font=("", 25), justify=tk.RIGHT)
entry.pack(fill=tk.X)
upper.pack(padx=20, pady=20)
```

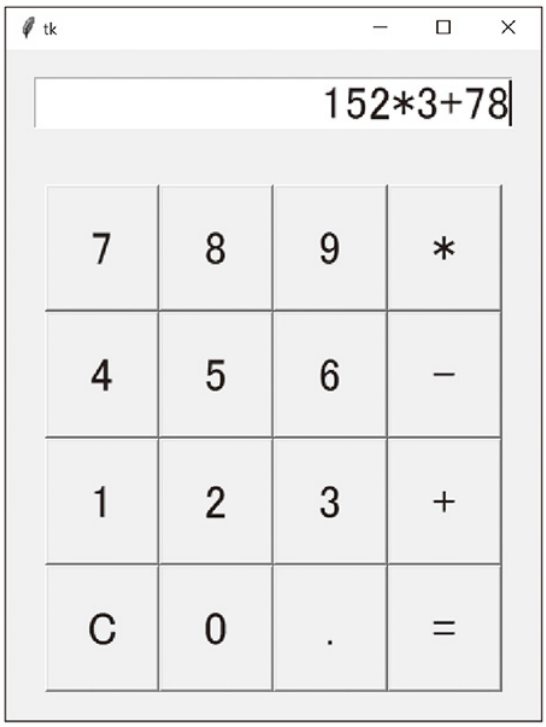
```
lower = tk.Frame(root) 10
lower.pack(pady=20)
for i, txt in enumerate(labels): 11
    b = tk.Button(lower, width=4, height=2, font=("",25), text=txt)
12
    b.grid(row=int(i/4), column=i%4) 13
    b.bind("<Button-1>", clicked) 14

root.mainloop()
```

1は電卓のボタン用のラベルです。**2**はボタン押下時のイベントハンドラです。**3**で押下されたボタンのtextを取得します。「C」キーが押下されたときは、**4**で入力内容をクリアします。

「=」キーが押下されたときは、**5**で入力内容を取得してevalで計算をします。

6で入力内容をクリアし、**7**で計算結果を表示します。それ以外のときは**8**で押下されたキーを入力領域に追加します。**9**で電卓上部のFrameを作成します。**10**で電卓下部のFrameを作成し、**11**でラベルを取り出し、**12**でボタンを作成します。**13**でボタンを格子状に配置し、**14**でイベントハンドラを登録しています。

⦿出力結果



7-3 Tkinterでイベントを処理する

部品を並べたら操作したくなるはずです。本節ではマウスやキーボードに反応するコードの実装方法を見ていきます。

➡175

クリックイベントに反応するには

ウィジェットのcommand引数、もしくはbindメソッドを使用する

ボタンやスライダバーなどはユーザー入力に反応して何か処理を行うことが一般的です。ボタンが押下されたときにコマンドを実行するには、Buttonウィジェット作成時の**command引数**にコールバック関数を指定します。

◎リスト tkinter_button0.py

```
import tkinter as tk
def clicked():
    print("clicked")

root = tk.Tk()
button0 = tk.Button(root, text="Hello", command=clicked)
button1 = tk.Button(root, text="World", command=clicked)
button0.pack()
button1.pack()
root.mainloop()
```

command引数でコールバック関数を登録する方法はシンプルですが、コールバック関数に値を渡すことができないので、複数のボタンを1つのコールバック関数で扱うような状況には適していません。そのようなときには**bind**メソッドを使用してください。イベントに関する情報が引数としてコールバック関数に渡されます。

◉出力結果

◎メソッド　bind(event, handler)

カテゴリ	**tkinterモジュールの各種ウィジェットクラス**
引数	**event：イベント文字列、handler：イベントハンドラ**
戻り値	**バインドを示す文字列**
説明	**ボタンなどイベントを発生するウィジェットにおいて、イベントとそれを処理するイベントハンドラを紐付ける**
使用例	**b = tk.Button(root, text="click")** **b.bind("<Button-1>", clicked)**

bindメソッドでは、最初の引数にイベントの種類を文字列で指定します。マウスのクリックやリリースを検出する場合には、以下の文字列に指定します。昔のUNIX端末では3つボタンマウスを使うのが一般的でしたが、そのような背景があったために3つのボタンに対応できるようになっています。

⊙マウスボタンのイベント

イベントの文字列	イベントの内容
<Button-1>	左マウスボタンが押されたとき
<Button-2>	真ん中マウスボタンが押されたとき
<Button-3>	右マウスボタンが押されたとき
<ButtonRelease-1>	左マウスボタンが離されたとき
<ButtonRelease-2>	真ん中マウスボタンが離されたとき
<ButtonRelease-3>	右マウスボタンが離されたとき

bindメソッドの第2引数にはコールバック関数を指定します。

◎リスト tkinter_button1.py

```
import tkinter as tk

def clicked(e):
      print("x:{0} y:{1} text:{2}".format(e.x, e.y, e.widget["text"]))

root = tk.Tk()
button0 = tk.Button(root, text="Hello")
button1 = tk.Button(root, text="World")
button0.pack()
button1.pack()
button0.bind("<Button-1>", clicked)
button1.bind("<Button-3>", clicked)
root.mainloop()
```

今回の例ではclicked(e)をコールバック関数として定義しています。**実際に発生したイベントの情報はコールバック関数の引数として渡されます。**イベントeのxプロパティがマウスのx座標、yプロパティがy座標、widgetプロパティがイベント発生元になったウィジェットを保持しています。この例では、e.widget["text"]で、ボタンのtextプロパティを取得してい

ます。マウスの座標位置などの詳細情報が不要なときは、以下のようにlambda式を使ってもよいでしょう。

```
def clicked(s):
      print(s)

button0.bind("<Button-1>", lambda e:clicked("hello"))
button1.bind("<Button-3>", lambda e:clicked("python"))
```

テーマ付きtk

tkのバージョン8.5からテーマ付きtkが導入されました。テーマ付きtk用のPythonモジュールがtkinter.ttkです。下記のリストは通常のtkinterのサンプルです。2行目に以下の を挿入すると、tkinterのウィジェットがtkinter.ttkのウィジェットに置き換えられます。たとえばtkinter.Buttonはtkinter.ttk.Buttonとなります。これによりウィジェットの外観が変わります。

ただし、ttkはtkinterとの互換性は意識されているものの、すべてのウィジェットで互換性は保証されていないので注意してください。また、ttkは単に部品の見た目を変えるだけでなく、TreeViewなど新しい部品が追加されたり、部品の役割とスタイルが分離しやすくなっていたりといった違いがあります。

◎リスト ttkinter_basic.py

```
from tkinter import *
root = Tk() ←1行目と2行目の間に下記1を挿入
Button(root, text="Hello").pack()
Label(root, text="World").pack()
```

```
Entry(root).pack()
Checkbutton(root).pack()
Scale(root).pack()
root.mainloop()
```

```
from tkinter.ttk import *   1
```

◉出力結果

➡176

キー入力に反応するには

bindメソッドのKeyPress、KeyReleaseイベントを使用する

キー入力を処理する場合にも**bind**を使用します。キー入力はOSで検出され、**フォーカス**を持っているウィジェットへ送られます。

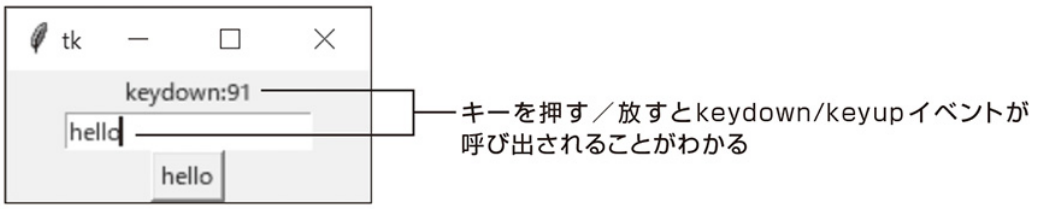
◎リスト tkinter_keyevent0.py

```
import tkinter as tk

def keydown(e):
    label["text"] = "keydown:" + str(e.keycode)   1
def keyup(e):
```

```
    label["text"] = "keyup:" + str(e.keycode)   2

root = tk.Tk()
label = tk.Label(root)
label.pack()
entry = tk.Entry(root) 3
entry.pack()
tk.Button(root, text = "hello").pack() 4
entry.bind("<KeyPress>", keydown) 5
entry.bind("<KeyRelease>", keyup) 6
root.mainloop()
```

1がキー押下、**2**がリリース時のコールバック関数です。**3**で1行入力用のEntryウィジェットを作成しています。**4**のようにウィジェットの作成とpackによるレイアウトを1行で記述することも可能です。**5**と**6**でbindメソッドを使って<KeyPress>と<KeyRelease>をイベントハンドラに紐付けています。

⊙実行結果



なお、Tabキーを押すと、フォーカスが入力欄からボタンに移ります（入力欄が非アクティブになる）。ボタンにフォーカスがある状態でキーを押しても、ボタンにはキー押下のコールバック関数が紐付けられていないのでイベントは検出されません。その状態でキーを押しても反応しません。

それでもキーを反応させたい場合は、前述のリスト**5** **6**のbindメソッドを、すべてのイベントを受け取るbind_allメソッドに書き換えます。

```
entry.bind_all("<KeyPress>", keydown)
entry.bind_all("<KeyRelease>", keyup)
```

あるいは、root.bindメソッドで、ルートウィンドウにイベントハンドラを紐付けるとよいでしょう。

```
root.bind("<KeyPress>", keydown)
root.bind("<KeyRelease>", keyup)
```

どのキーが押されたかを調べるには、イベントハンドラの引数として渡されるオブジェクトの**keycode**プロパティを見ます。ここにキーコードが格納されます。**keysym**プロパティにはキーの名前が格納されます。どちらか使いやすいほうを選択してください。

また、単にキーのイベントを検出するだけであれば、次の<Key>イベントも利用できます。

```
root.bind("<Key>", keyhandler)
```

➡177

マウスイベントを取得するには

bindメソッドの<Motion>イベントを使用する

マウスの移動を検出するには**<Motion>**イベントを使用します。これと、「175　クリックイベントに反応するには」の<Button-1> ～ <Button-3> 、 <ButtonRelease-1> ～

<ButtonRelease-3>を組み合わせると簡易ドローアプリが作成できます。

◎リスト tkinter_canvas_mousedraw.py

```
import tkinter as tk
prev = [0, 0] 1
isMouseDown = False 2

root = tk.Tk()
canvas = tk.Canvas(root, width=500, height=500)
canvas.pack()

def mousemove(event):   3
    if isMouseDown:
        canvas.create_line(prev[0], prev[1], event.x, event.y)    4
        prev[0], prev[1] = event.x, event.y 5
def mousedown(event): 6
    global isMouseDown
    prev[0], prev[1] = event.x, event.y
    isMouseDown = True

def mouseup(event): 7
    global isMouseDown
    isMouseDown = False

canvas.bind('<Button-1>' , mousedown) 8
canvas.bind('<ButtonRelease-1>' , mouseup) 9
canvas.bind('<Motion>', mousemove) 10
root.mainloop()
```

1と**2**で、以前のマウスの座標を保持するprevとマウスの押下状態を保持するisMouseDownというグローバル変数を宣言しています。

3はマウス移動時のコールバック関数で、マウスが押下状態、すなわちisMouseDownがTrueのときは、canvasのcreate_lineメソッドを使って、以前の座標から現在の座標まで線を引きます。

4のcreate_lineはCanvas上に線を描画するメソッドです。4つの引数（点1のx座標、点1のy座標、点2のx座標、点2のy座標）を引数にとり、点1と点2を結ぶ線を描画します。**5**でprevに現在のマウスの位置を格納します。

6はマウス押下時のコールバック関数で、現在のマウス位置をprevに格納し、isMouseDownをTrueに設定します。

7はマウスリリース時のコールバック関数で、isMouseDownをFalseにします。

8～**10**でcanvasウィジェットにイベントハンドラを登録しています。

◎出力結果

グローバル変数とローカル変数

関数の中からグローバル変数の値を書き換える場合には注意が必要です。グローバル変数はいろいろな場所からアクセスできます。これは便利な反面、不用意に変更するとバグの原因になってしまいます。そこで、Pythonではグローバル変数の値を関数の中から変更する場合には、その意

図を明確に表明するためにglobalと宣言する必要があります。この指定がないと、全く同一の変数名を使用していたとしても、ローカル変数として扱われるので、グローバル変数の値は更新されません。
　前出のリスト「tkinter_canvas_mousedraw.py」のisMouseDownはマウスの押下状態を保持するグローバル変数ですが、その値を関数の中から変更するためにglobalを指定しています。

➡178

生成後にウィジェットのオプションを設定するには

configureメソッドをキーワード引数で呼び出し、各種オプションを設定する

　各種ウィジェットを作成するときに、引数のオプションを使ってカスタマイズすることが可能です。生成したあとから変更する場合には**configure**メソッドを使います。

◎メソッド　configure()

カテゴリ	**tkinterモジュールの各種ウィジェットクラス**
引数	**引数名＝引数値の形式で指定**
戻り値	**なし**
説明	**ウィジェットの設定内容を更新する**
使用例	**label.configure(text="hello")**

◎リスト tkinter_configure.py

```
import tkinter as tk

def textUpdate():
    label.configure(text=entry.get()) 1
```

```
def scaleUpdate(e):
    label.configure(font=("", e)) 2
root = tk.Tk()
label = tk.Label(root)
label.pack()
entry = tk.Entry(root)
entry.pack()
tk.Button(root, text="Update", command=textUpdate).pack()
tk.Scale(root, orient = 'h', from_ = 10, to = 50,
    command=scaleUpdate).pack(fill=tk.X)
root.mainloop()
```

1はボタンが押下されたときのコールバックです。entryに入力された値をgetメソッドで取り出し、configureメソッドを使ってラベルに設定しています。**2**はScaleのコールバック関数です。引数eにはスライダバーの数値が渡されます。configureメソッドを使って、ラベルに表示されているフォントのサイズを変更しています。

◉出力結果

➡179

GUIの状態を変数にバインドするには

StringVar、IntVar、BooleanVarなどのオブジェクトをウィジェットにバインドする

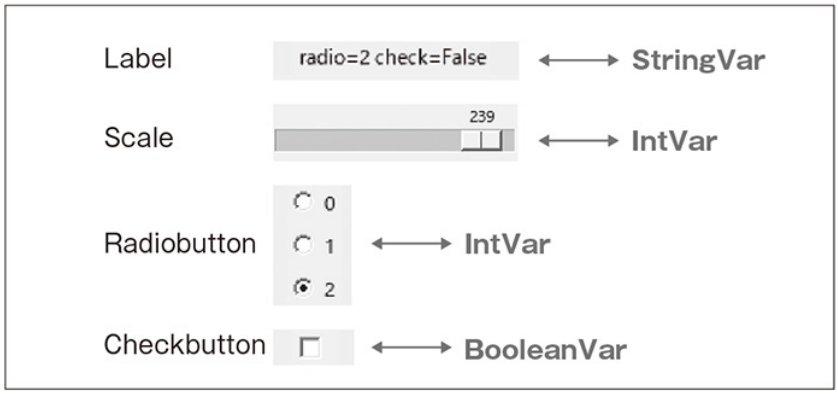
GUIの部品によってはその状態を変数と連動させると便利なものがあります。このような仕組みを**バインド**といいます。たとえば、次のようなものです。

- ●スライダバーが操作されると自動で数値が更新される
- ●チェックボックスが操作されると自動で状態変数が更新される
- ●ラジオボタンの選択肢が変わると自動で変数が更新される
- ●エディットの文字が更新されるとデータ用文字列が連動して更新される

このとき、連動する変数は、int、str、boolといった組み込み型ではなく、**IntVar**、**StringVar**、**BooleanBar**といったオブジェクトになります。IntVarはintを包んだような（Wrapした）オブジェクトで、StringVarはstrを、BooleanVarはboolを、といった具合です。

ScaleとIntVarを関連付けておけば、Scaleを操作しただけでIntVarの値が自動で更新されます。Radiobuttonの場合はIntVarを、Checkbuttonの場合はBooleanVarを使います。LabelとStringVarを関連付けておけば、StringVarの値を更新するだけでLabelの表示が自動的に更新されます。GUIの内容によって変数の型が変わってくることに注意してください。

⊙GUIの状態に連動する変数



関連付けはウィジェットを作成するときのオプションで行います。

⊙ウィジェットとオプション

ウィジェット	オプション	オブジェクト
Label	textvariable	StringVar
Entry	textvariable	StringVar
Scale	variable	IntVar
Radiobutton	variable	IntVar
Checkbutton	variable	BooleanVar

◎リスト tkinter_variable_bind.py

```
import tkinter as tk
def scaleUpdate(e):
    color = "#{0:02x}{1:02x}{2:02x}".format(R.get(), 1
        G.get(), B.get())
    root.configure(bg = color) 2

def btnUpdate():
    txt = "radio={0} check={1}".format(RadioVar.get(), 3
        CheckVar.get())
    TextVar.set(txt) 4

root = tk.Tk()
TextVar = tk.StringVar() 5
label = tk.Label(root, textvariable=TextVar) 6
label.pack()
R, G, B = tk.IntVar(), tk.IntVar(), tk.IntVar() 7
for color in [R, G, B]:
    tk.Scale(root, orient = 'h', from_ = 0, to = 255, 8
        variable = color, command = scaleUpdate).pack(fill=tk.X)

RadioVar = tk.IntVar() 9
for num in range(3):
```

```
        tk.Radiobutton(root, text = num, value = num,          10
                variable = RadioVar, command = btnUpdate).pack()
CheckVar = tk.BooleanVar() 11
tk.Checkbutton(root, variable = CheckVar,    12
        command = btnUpdate).pack()

root.mainloop()
```

1はスライダバーが操作されたときのコールバック関数です。R.get()、G.get()、B.get()で赤・緑・青の数値を取得し、＃RRGGBB形式の文字列を構築しています。**2**でメインウィンドウの背景色を設定しています。

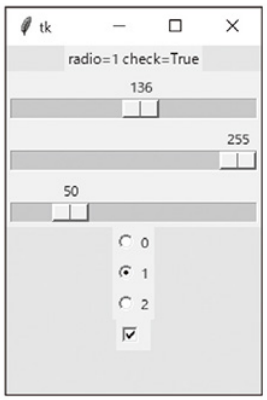
3はラジオボタンやチェックボックスの状態が変化したときに呼ばれるコールバック関数です。RadioVar.get()でラジオボタンの選択肢を取得し、CheckVar.get()でチェックボタンの状態を取得して、文字列txtを作成しています。**4**でその文字列をTextVarにセットしています。これでLabelの表示内容が自動で更新されます。

5でLabelのバインド変数StringVarを生成し、**6**でLabelのオプションとして登録しています。**7**でR、G、Bの3つの数値用のIntVarを作成し、**8**でそれぞれのScaleオブジェクトのvariableにセットすることでバインドしています。**9**でラジオボタン用のIntVarを作成し、**10**でラジオボタンにその変数を紐付けています。

11ではチェックボックス用のBooleanVarを作成し、**12**でチェックボタンに紐付けています。

スライダバーを操作すると背景色が変わり、ラジオボタンやチェックボックスを操作すると画面上部のラベルが変化します。

⦿出力結果



7-4 TkinterのCanvasに描画する

Canvasウィジェットを使うと、線・矩形・円・画像などを自由に描画することができます。本節では、それらを描画する方法のほか、描画したものを操作する方法を説明します。

Canvasについて

HTML5には**Canvas**という要素があり、自由に描画することができます。TkinterのCanvasも要素の上に描画できるという点は共通していますが、描画対象の扱いに大きな違いがあります。HTML5のCanvasでは絵筆のように描画するのに対し、TkinterのCanvasでは**描画内容をオブジェクトとして扱います。**ちょうど、ペイントツールとドローツールのような関係です。TkinterのCanvasではいったん描画したものを、あとから場所やサイズを変更したり、色を変えたりといったことが可能です。

➡180

Canvasに描画するには

Canvasクラスのメソッドで線や矩形、文字などを描画する

それでは、さっそくTkinterのCanvasにいろいろ描画していきましょう。四角形、円、線、文字、画像といった基本的な描画をマスターしましょう。

●四角形

◎メソッド **create_rectangle(x0, y0, x1, y1, option)**

カテゴリ	tkinterのCanvasクラス
引数	x0, y0：左上の座標、x1, y1：右下の座標、option：さまざまな設定
戻り値	作成した四角形のIDを返す
説明	矩形を描画し、そのオブジェクトの識別子を返す。x1, v1が幅と高さではない点に注意
使 用 例	id = canvas.create_rectangle(10, 50, 200, 100, fill="#FFFF00")

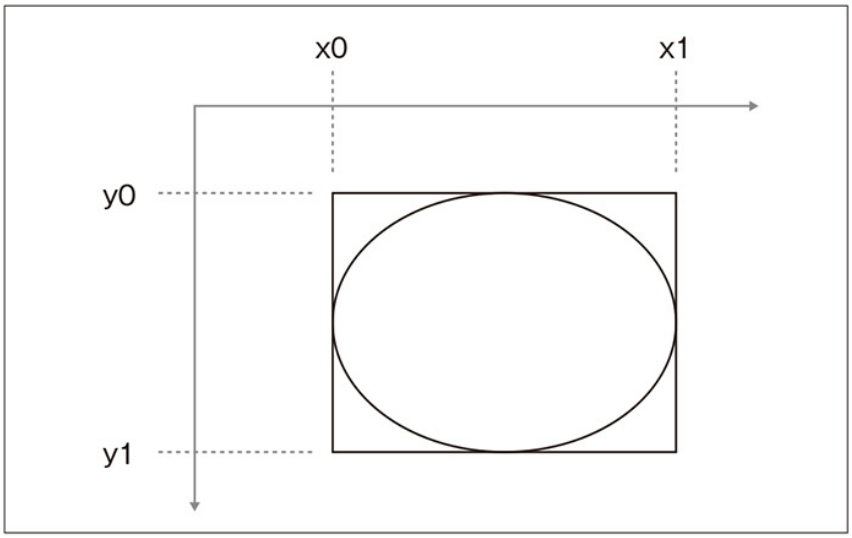
●楕円

◎メソッド **create_oval(x0, y0, x1, y1, option)**

カテゴリ	tkinterのCanvasクラス
引数	x0,y0：左上の座標、x1, y1：右下の座標、option：さまざまな設定
戻り値	作成した楕円のIDを返す
説明	x0, y0, x1, y1で表される矩形に内接する楕円を描画し、そのオブジェクトの識別子を返す。(x1-x0)と(v1-v0)が等しいとき正円になる
使 用 例	id = canvas.create_oval(10, 50, 200, 100, fill="#FFFF00")

create_rectangleと**create_oval**で描画される様子を次に示します。

⊙create_rectangleとcreate_oval



●線

◎メソッド　**create_line(x0, y0, x1, y1, option)**

カテゴリ	tkinterのCanvasクラス
引数	x0, y0：点1の座標、x1, y1：点２の座標、option：さまざまな設定
戻り値	作成した線のIDを返す
説明	点1と点2を結ぶ線を描画し、そのオブジェクトの識別子を返す
使用例	id = canvas.create_line(300, 50, 400, 100, width=3)

●文字

◎メソッド　**create_text(x, y, option)**

カテゴリ	tkinterのCanvasクラス
引数	option：描画文字はtextオプションで、フォントはfontオプションで指定する
戻り値	作成した文字のIDを返す
説明	x, yの座標にテキストを描画する
使用例	id = canvas.create_text(350, 180, text="Hello", font=("", 30))

●画像

◎メソッド create_image(x, y, option)

カテゴリ	tkinterのCanvasクラス
引数	x, y：画像を描画する場所、option：画像オブジェクトはimageオプションで指定する
戻り値	作成した画像のIDを返す
説明	画像からPhotoImageオブジェクトを作成し、そのオブジェクトをimageオプションで指定することで、座標(x,y)の場所に画像を描画する。デフォルトでは、anchorがCENTERなので(x,y)が画像の中心となる。画像の左上を(x,y)にする場合にはoptionとしてanchor=NWを指定する
使用例	img = tk.PhotoImage(file = 'logo.png') id = canvas.create_image(100, 250, image=img, anchor=tk.NW)

このほかにもCanvasにはポリゴンを描く**create_polygon**、円弧を描く**create_arc**などのメソッドが用意されています。

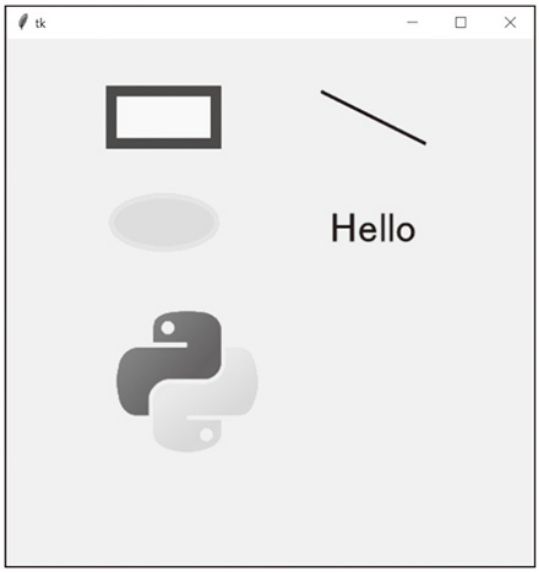
◎リスト tkinter_canvas_drawings.py

```
import tkinter as tk
root = tk.Tk()
canvas = tk.Canvas(root, width=500, height=500)
canvas.pack()

canvas.create_rectangle(100, 50, 200, 100,    1
      fill="#FFFF00", outline="#0000FF", width=10)
canvas.create_oval(100, 150, 200, 200,   2
      fill="#00FF00", outline="#00FFFF", width=5)
canvas.create_line(300, 50, 400, 100, width=3) 3
canvas.create_text(350, 180, text="Hello", font=("", 30)) 4
img = tk.PhotoImage(file = 'logo.png') 5
canvas.create_image(100, 250, image=img, anchor=tk.NW) 6
```

```
root.mainloop()
```

1で矩形を描き、**2**で楕円を描き、**3**で直線、**4**でテキストを描画しています。**5**で「logo.png」というファイルから画像オブジェクトを作成し、**6**で画像を描画しています。それぞれオプションで色や線の太さなどを指定しています。

⊙出力結果



画像のリサイズ

Tkinterだけで画像のリサイズを効率よく行うことはできません。以下のようにzoomとsubsampleを組み合わせることである程度の拡縮はできますが、これらのメソッドは引数に整数しか取れないこともあり、リサイズ後の画像品質はだいぶ劣化してしまいます。

```
img = img.zoom(2, 1)              #横2倍, 縦1倍
```

```
img = img.subsample(2, 2)       #横1/2倍, 縦1/2倍
```

リサイズが必要な場合は、PIL（Python Imaging Library）などほかのモジュールを使うのがよいでしょう。

➡181

Canvasに描画した内容を操作するには

描画時に取得したidを引数にCanvasの各種メソッドを実行する

前項でCanvasへの描画方法について説明しましたが、このままではHTMLのCanvasとあまり変わりません。draw_rectangleではなくcreate_rectangleというメソッド名だったことにも象徴されていますが、これらのメソッドは描画するだけでなく、オブジェクトを作成し、その識別子を戻り値として返します。そして、その識別子を使用すれば、いったん描画した内容を変更することができます。

◎メソッド　coords(id [, coods])

カテゴリ	tkinterモジュールのCanvasクラス
引数	id : canvasに描画したオブジェクトのid、coods: 更新後の座標
戻り値	オブジェクトの座標
説明	オブジェクトが占める矩形の座標を返す。coordsにx0,y0,x1,y1を指定するとオブジェクトの座標が更新される
使用例	obj_id = canvas.create_rectangle(10,10, 100,100) r = canvas.coords(obj_id)

◎メソッド　move(id, x, y)

カテゴリ	tkinterモジュールのCanvasクラス

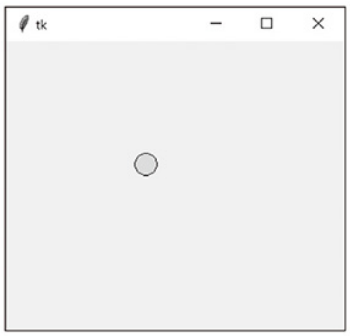
引数	id：canvasに描画したオブジェクトのID、x：横方向の移動量、y：縦方向の移動量
戻り値	なし
説明	オブジェクトを現在の位置からx、y分移動する
使用例	ball = canvas.create_oval(100, 100, 120, 120) canvas.move(ball, 10, 20)

◎リスト tkinter_canvas_objects.py

```
import tkinter as tk

def keyhandler(e):
    r = canvas.coords(ball) 1
    print(r)
    x, y = 0, 0
    if e.keysym == "Up":
        y = -5
    elif e.keysym == "Down":
        y = +5
    elif e.keysym == "Right":
        x = +5
    elif e.keysym == "Left":
        x = -5
    canvas.move(ball, x, y) 2

root = tk.Tk()
canvas = tk.Canvas(root, width=300, height=300)
canvas.pack()
ball = canvas.create_oval(100, 100, 120, 120,
    fill="#00FF00")
root.bind("<Key>", keyhandler)

root.mainloop()
```

1でballオブジェクトの座標を取得して、コンソールに出力しています。その後、押下されたキーの種類をkeysymプロパティで調べ、その値に応じ、**2**でballを移動しています。

⊙出力結果



■実行結果（ボールを移動したときにコンソールに表示されるメッセージ）

```
[100.0, 100.0, 120.0, 120.0]
[105.0, 100.0, 125.0, 120.0]
[110.0, 100.0, 130.0, 120.0]
```

キーを押下するたびにballが移動します。またコンソールにはballの占める領域が出力されます。

➡182

mainloop以外の処理を実行するには

afterメソッドを使って関数を実行する

TkinterでGUIを構築したあとで、何らかの処理を実行したい場合もあるでしょう。以下のようなイメージです。

```
root.mainloop()
```

```
while True:
    some_func()
```

しかし、Tkinterではmainloopメソッドを実行することでメインループに入ってしまいます。つまり、mainloopより後ろに記述した内容は、ウィンドウが閉じられるまで実行されません。このような状況に対処するために用意されているのが、指定した時間後に処理を実行する**after**メソッドです。また、このあとに示すサンプルでは、オブジェクトと矩形の重なりを検出する、次の**find_overlapping**メソッドを使用します。

◎メソッド　after(msec, func)

カテゴリ	**tkinterのTkクラス**
引数	**msec：ミリ秒、func：関数**
戻り値	**識別子を示す文字列**
説明	**msecで指定した時間のあとにfuncを実行する。funcの中でafterを呼べば、mainloopに入ったあとでも、定期的な処理を行える**
使用例	**id = root.after(10, tick)**

◎メソッド　find_overlapping(x1,y1,x2,y2)

カテゴリ	**tkinterモジュールのCanvasクラス**
引数	**重なりを検出する矩形領域**
戻り値	**衝突しているオブジェクトIDのタプル**
説明	**引数で指定した矩形が重なっているオブジェクトを検出する**
使用例	**collide = canvas.find_overlapping(x0,y0,x1,y1)**

canvasのまとめとして、迷路の壁にぶつからないように上下左右で矩形を動かすサンプルを次に示します。

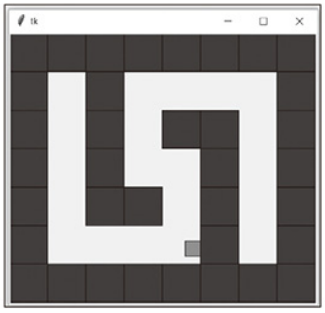
◎リスト tkinter_canvas_tick.py

```
import tkinter as tk
import collections 1
keymap = collections.defaultdict(bool) 2
speed = [0, 0] 3
MAP = ( (1,1,1,1,1,1,1,1),   4
        (1,0,1,0,0,0,0,1),
        (1,0,1,0,1,1,0,1),
        (1,0,1,0,0,1,0,1),
        (1,0,1,1,0,1,0,1),
        (1,0,0,0,0,1,0,1),
        (1,1,1,1,1,1,1,1))
def keypress(e): 5
    keymap[e.keycode] = True
def keyup(e): 6
    keymap[e.keycode] = False
def tick(): 7
    if keymap[37]: speed[0] -= 1
    if keymap[39]: speed[0] += 1
    if keymap[38]: speed[1] -= 1
    if keymap[40]: speed[1] += 1
    canvas.move(you, speed[0], speed[1]) 8
    speed[0] *= 0.95 9
    speed[1] *= 0.95
    b = canvas.coords(you) 10
    collide = canvas.find_overlapping( 11
        b[0], b[1], b[2], b[3])
    if len(collide) <= 1: 12
        root.after(30, tick)

root = tk.Tk()
canvas = tk.Canvas(root, width=400, height=350)
canvas.bind_all("<KeyPress>", keypress)
canvas.bind_all("<KeyRelease>", keyup)
canvas.pack()
for j, row in enumerate(MAP):
```

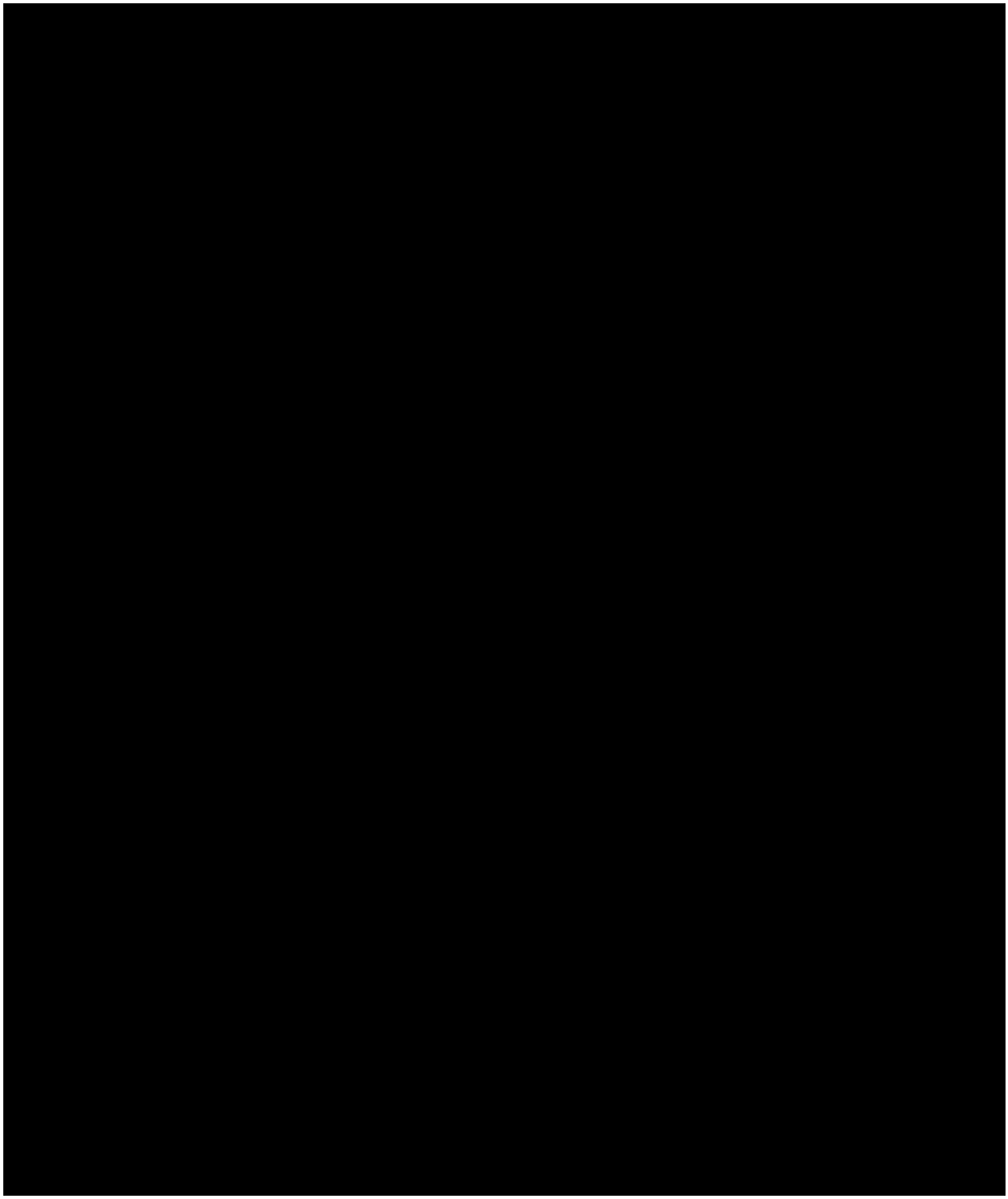
```
        for i, col in enumerate(row):
            if MAP[j][i] == 1:
                canvas.create_rectangle(i*50, j*50,   13
                    (i+1)*50, (j+1)*50, fill="#00c")
you = canvas.create_rectangle(70,70,90,90,fill="#F0F")   14
root.after(30, tick) 15
root.mainloop()
```

1と**2**でキー押下情報を保持する辞書を作成しています。**3**では移動速度を、**4**では地図のデータを保持する変数を初期化しています。**5**と**6**でキー押下状態をkeymap辞書に反映しています。**7**ではキーの状態に応じて移動速度を変化させ、**8**でキャラクタを移動させています。**9**は減速処理です。**10**でキャラクタの座標を取得し、**11**で衝突判定をしています。衝突していない場合は**12**で処理を継続しています。**13**ではMAPから地図を、**14**では操作対象となるキャラクタを作成しています。**15**ではでは、afterメソッドにより、メインループが開始された30ミリ秒後にtickメソッドを呼び出しています。

⊙出力結果



Chapter 8



そのほかの
便利なモジュール

Pythonには非常に多くのモジュールがあります。ここまでのカテゴリには分類されなかったものの、覚えておくと便利なモジュールをいくつか集めてみました。

8-1 そのほかのモジュール

ここでは、知っておくと便利なクラス、関数などをランダムに取り上げます。

➡183

コマンドラインツールの引数処理を簡単に行うには

argparseモジュールのArgumentParserを使用する

Pythonでコマンドラインツールを作成する場合、各種引数を受け取る必要があるかもしれません。引数のチェックやヘルプの表示に便利なクラスを見てみましょう。

◎コンストラクタ　ArgumentParser(description)

カテゴリ	argparseモジュール
引数	description：コマンドの使い方を伝えるメッセージ文字列を指定する
戻り値	Parserオブジェクト
説明	コマンド引数のチェックなどの処理を簡易化するとともに、コマンド使用方法を出力する
使用例	parser = argparse.ArgumentParser(description="multiple two values")

◎メソッド　add_argument(name or flags [,help] [,type] [, required] [,default][, nargs]...)

カテゴリ	argparseモジュールのArgumentParserクラス
引数	name or flags：引数の名前、もしくはオプション文字列のリスト help：引数の説明 type：引数が変換されるべき型

	required: コマンドラインオプションが省略可能かどうか **default : 引数がなかった場合のデフォルト値** **nargs : 受け取るコマンドライン引数の数を指定**
戻り値	**ArgumentParserが引数を管理するために使うオブジェクト**
説明	**引数を追加する**
使 用 例	**a = parser.add_argument("-a", help="value1", type=float, default=0)**

◎リスト argparse_test.py

```
import argparse

def mul(a, b, c):
    return a * b * c 1

if __name__ == "__main__":
    parser = argparse.ArgumentParser("multiply three values")
        parser.add_argument("-a",  help="value1",  type=float,
required=True)
        parser.add_argument("-b",  help="value2",  type=float,
required=True)
        parser.add_argument("-c",  help="value3",  type=float,
default=1)
    args = parser.parse_args() 2
    print(mul(args.a, args.b, args.c))
```

　このスクリプトは3つの引数をかけ合わせた結果を出力します**1**。最初の2つは必須の引数です。最後の1つはオプション引数で、デフォルト値が１です。parse_argsは引数の文字列をオブジェクトに変換するメソッドです**2**。 まずは引数を指定しなかった場合です。引数-a、-bが必須の旨のエラーが表示されます。

◎実行例

```
>python argparse_test.py Enter
usage: multiply three values [-h] -a A -b B [-c C]
multiply three values: error: the following arguments are required: -
a, -b
```

-hオプションを指定した場合の出力は以下の通りです。

◎実行例

```
>python argparse_test.py -h Enter
usage: multiply three values [-h] -a A -b B [-c C]

optional arguments:
  -h, --help   show this help message and exit
  -a A           value1
  -b B           value2
  -c C           value3
```

次のように引数を3つ指定した場合は正しい値が出力されることがわかります。

◎実行例

```
>python argparse_test.py -a 1 -b 2 -c 3   Enter
6.0
```

以前はoptparseというモジュールがコマンドライン解析用に用いられていましたが、廃止される予定です。今回紹介したargparseモジュールを使ってください。

●オプションをつけずに引数を指定できるようにする

「python prog -a 3 -b 4 -c 5」のようにオプションを指定するのではなく、python 「prog 3 4 5」のように引数を処理した

い場合は以下のサンプルのように記述します。

◎リスト argparse_test2.py

```
import argparse

def mul(a, b, c):
	return a * b * c

if __name__ == "__main__":
	parser = argparse.ArgumentParser("multiply three values")
		parser.add_argument("numbers", help="float values", type=float, nargs=3) 1
	args = parser.parse_args()
			print(mul(args.numbers[0], args.numbers[1], args.numbers[2])) 2
```

1の「nargs=3」で、コマンドライン引数の数を指定しています。受け取った引数はリストに格納され、**2**で各要素をかけ合わせています。

引数の個数が想定と異なる場合にはエラーが出力され、正しい場合は正しく処理が行われていることがわかります。

◎実行例

```
>python argparse_test2.py 1 2 Enter
usage: multiply three values [-h] numbers numbers numbers
multiply three values: error: the following arguments are required: numbers

>python.exe argparse_test2.py 1 2 3 Enter
6.0
```

 プログラムからコマンドライン引数を受け取る方法

argparseモジュールを使用しないで、プログラムからコマンドライン引数を受け取るにはsysモジュールのリストargvを参照します。argvにはコマンドライン引数がリストとして格納されます。

◎リスト sys_argv.py

```
import sys
for i, v in enumerate(sys.argv):
    print("argv[{0}]={1}".format(i, v))
```

◎実行例

```
>python sys_argv.py a b 3 Enter
argv[0]=sys_argv.py
argv[1]=a
argv[2]=b
argv[3]=3
```

コマンドラインの値はすべて文字列になるので数値として扱う際には変換する必要があります。

➡184

ログを処理するには

 loggingモジュールを使用する

かなり多くの開発者がprint関数を使ったデバッグをしたことがあると思います。print関数は簡単でよいのですが、コメ

ントアウトをON/OFFする必要があったり、多くのprint関数でソースコードが散らかってしまったりと問題点も少なくありません。そんなときはloggingモジュールの使用を検討してみましょう。

◎関数　getLogger(name=None)

カテゴリ	**loggingモジュール**
引数	**ログの名前**
戻り値	**指定された名前のロガーオブジェクトを返す**
説明	**名前省略時はrootという名前が使用される**
使用例	**logger = logging.getLogger()**

◎　関　数　basicConfig(level=level, filename=filename)

カテゴリ	**loggingモジュール**
引数	**level：ロギングレベルを指定する。DEBUG、INFO、WARNING、ERROR、CRITICALから選択する** **filename：ログを出力するファイルを指定する**
戻り値	**なし**
説明	**ロギングレベルを指定する。レベルを設定することで出力情報を制御できる**
使用例	**logger = logging.basicConfig(level=logging.CRITICAL)**

　ロギングレベルは以下のように設定されています。設定したロギングレベル以上の情報のみが出力されるようになります。

⊙ロギングレベル

CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

⊙ログ出力メソッド

debug(str)
info(str)
warning(str)
error(str)
critical(str)

◎リスト logging_test.py

```
import logging
logger = logging.getLogger()
logging.basicConfig(level=logging.CRITICAL) 1

logger.debug("hello")
logger.info("world")
logger.warning("python")
logger.error("standard")
logger.critical("library")
```

basicConfigで設定した以上の重要度レベルを持つメッセージのみが表示されます。上記の例ではCRITICAL 1 と指定しているので、criticalレベルのメッセージしか出力されません。

■実行結果

```
CRITICAL:root:library
```

loggerオブジェクトは単にエラーを出力するレベルを制御するだけでなく、コードの作成時の意図を第三者に伝える役目もあります（単なる情報なのか、警告なのか、エラーなのか）。効果的に使用する習慣を付けるとよいでしょう。

 モジュール名と同じ名前のファイル名を付けないこと

Pythonではよく使う関数やクラスなどをまとめてモジュールにすることができます。デフォルトでモジュール名＝ファイル名となります。ディレクトリをパッケージとして

扱うには__init__.pyを用意する必要があります。いずれにせよファイル名やディレクトリ名を標準モジュールと同じ名前にしてしまうと、自分が作ったファイルやディレクトリが優先されてしまうことがあるので注意してください。os.pyやlogging.pyのように標準モジュールと同名のファイルを自分で作り、実行しようとしても、意図した挙動にならなくなることがあります。

➡185

オブジェクトを手軽に永続化するには

shelveモジュールを使用する

　オブジェクトをファイルに保存したり、読み込んだりするにはshelveモジュールが便利です。shelveはその実装の中でpickle（134で解説）を利用しています。shelveは保存するデータを辞書のようにアクセスできるので使い方が簡単です。ちょっとしたデータを保存したい場合に適しています。

◎関数　open(filename)

カテゴリ	**shelveモジュール**
引数	**ファイル名**
戻り値	**値を格納するための辞書オブジェクトが返される**
説明	**通常の辞書のようにアクセスするだけで、格納された情報がファイルに保存される**
使用例	**with shelve.open("data.db") as f:**

◎リスト shelve_test.py

```
import shelve

with shelve.open('shelve.db') as db:
```

```
        db['name'] = 'python'
        db['dict'] = {"A": 3, "B": (2, 5), "C": "Hello"}
        db['list'] = [2,4,5,6,1,2,3]

with shelve.open('shelve.db') as db:
        print(db['name'])
        print(db['dict'])
        print(db['list'])
```

◎実行例

```
>python shelve_test.py Enter
python
{'A': 3, 'C': 'Hello', 'B': (2, 5)}
[2, 4, 5, 6, 1, 2, 3]
```

　前半部分でshelve.dbに値を書き込んで、後半部分で値を読み込んでいます。実行したディレクトリにはshelve.dbというファイルが生成されています。

➡186

ユニットテストを実行するには

unittestモジュールを使用する

　テスト駆動開発にはユニットテストが欠かせません。Pythonの標準ライブラリにもユニットテスト用のunittestモジュールが含まれています。
　テスト駆動開発では一般的に以下のような開発サイクルを繰り返します。

　1) 機能を検証するためのテストを書く（テストはFail）

2) テストをパスするための最小限のコードを書く（テストはPass）
3) 必要に応じてリファクタリングを行う

unittestを使いながら、足し算addと引き算subを行うCalcクラスを作ってみましょう。まずは、機能を検証するためのテストを作成します。

◎リスト calc_unittest.py

```
import unittest
import calc

class CalcTest(unittest.TestCase): 1
    def test_add(self):
        c = calc.Calc()
        expected = 5
        actual = c.add(2, 3) 2
        self.assertEqual(expected, actual) 3

    def test_sub(self):
        c = calc.Calc()
        expected = 3
        actual = c.sub(5, 2) 4
        self.assertEqual(expected, actual) 5

if __name__ == '__main__':
    unittest.main() 6
```

unittestを記述するには、unittest.TestCaseを継承するクラスを作成します。今回の場合はCalcTest1です。あとは、普通にメソッドを記述していくだけです。unittest.main()5を実行すると、testから始まる名前を持つメソッドが自動で呼び出されるので、その中でself.assertEqual3などの検証用のメソッドを呼び出します。test_addではCalcクラスのaddメソッド2を、

test_subではCalcクラスのsubメソッド**4**を呼び出して、その戻り値が予期した値か否かを検証しています。
　このテストを実行するとcalcモジュールがないというエラーが出力されるので、calcモジュールを作成します。

◎リスト calc.py

```
class Calc:
	def add(self, a, b):
		pass
	def sub(self, a, b):
		pass
```

　エラーが出ることを確認したいので、メソッドの中身は意図的に空にしています。この段階でテストを実行すると、以下のようなエラーが出力されます。

■実行結果

```
FF
======================================================================
FAIL: test_add (__main__.CalcTest)
...
self.assertEqual(expected, actual)
AssertionError: 5 != None
======================================================================
FAIL: test_sub (__main__.CalcTest)
...
self.assertEqual(expected, actual)
AssertionError: 3 != None
----------------------------------------------------------------------
Ran 2 tests in 0.006s
FAILED (failures=2)
```

expectedとactualの値が同じではないためAssertionErrorでテストがFailしていることがわかります。ここまで確認できてから、ようやくCalcを実装します。

◎リスト calc.py (修正後)

```
class Calc:
    def add(self, a, b):
        return a + b
    def sub(self, a, b):
        return a - b
```

再度テストを実行するとパスすることがわかります。

■実行結果

```
Ran 2 tests in 0.004s
OK
```

unittestではassertEqualのように値を比較するだけでなく、いろいろな検証用のメソッドが用意されています。主なメソッドを以下の表に列挙します。

⦿unittestで利用できる検証用メソッド

メソッド	確認事項
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)
assertAlmostEqual(a, b)	round(a-b, 7) == 0
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertRegex(s, r)	r.search(s)
assertNotRegex(s, r)	not r.search(s)
assertCountEqual(a, b)	aとbに、順番によらず同じ要素が同じ数だけある
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds)が例外excを送出する

　assertAlmostEqualは"ほぼ同じ値"を検証するために使います。浮動小数点を扱う場合などで使用します。各テストを実行する前後、クラスを初期化する前後に特別な処理を実行したい場合は、以下のメソッドを必要に応じて実装します。

⊙クラスを初期化する前後のメソッド

メソッド	説明
setUp()	個々のテスト実行直前に呼び出される
tearDown()	個々のテスト実行後に呼び出される
setUpClass()	指定したクラスのテスト実行前に呼び出される。引数にはクラスが渡される。classmethodで修飾されている必要がある
tearDownClass()	指定したクラスのテスト実行後に呼び出される。引数にはクラスが渡される。classmethodで修飾されている必要がある

これらのメソッドを使ってテストの挙動を確認してみます。前の例では各テストでCalcオブジェクトを作成していましたが、以下の例ではsetUpメソッドでCalcオブジェクトを作成しています**1**。

◎リスト calc_unittest1.py

```
import unittest
import calc

class CalcTest(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        print("setUpClass")

    @classmethod
    def tearDownClass(cls):
        print("tearDownClass")

    def setUp(self):
        self.calc = calc.Calc() 1
        print("setUp", id(self))

    def tearDown(self):
        print("tearDown", id(self))

    def test_add(self):
```

```
            expected = 5
            actual = self.calc.add(2, 3)
            print("test_add", id(self))
            self.assertEqual(expected, actual)

        def test_sub(self):
            expected = 3
            actual = self.calc.sub(5, 2)
            print("test_sub", id(self))
            self.assertEqual(expected, actual)

if __name__ == '__main__':
        unittest.main()
```

■実行結果

```
setUpClass
setUp 2653252992808
test_add 2653252992808
tearDown 2653252992808
.setUp 2653252992920
test_sub 2653252992920
tearDown 2653252992920
.tearDownClass
----------------------------------------------------------------
Ran 2 tests in 0.009s
OK
```

最初にsetUpClassが呼ばれ、最後にtearDownClassが呼ばれています。今回のテストケースはtest_addとtest_subの2つですが、それらが実行される直前にsetUpが、実行後にtearDownが実行されていることがわかります。また、それぞれのテストケースでオブジェクトのidを表示していますが（上記例では2653252992808と2653252992920）、別のインスタンスが作成されていることがわかります。

コンソールに色付きの文字を出力する

Pythonでは、ちょっとしたコマンドラインインターフェース用のスクリプトを作ることが少なくありません。多少色を付けるだけでもグッと印象が変わります。coloramaモジュールを使ってコマンドプロンプト（ターミナル）への出力に色を付けてみましょう。

NOTE coloramaモジュールは標準モジュールには含まれていないので、使用する際には「pip install colorama」（macOS/Linuxでは「pip3 ～」）でモジュールをインストールしてください。

◎関数　init(autoreset=False)

カテゴリ	**coloramaモジュール**
引数・戻り値	**なし**
説明	**coloramaモジュールを初期化する**
使用例	**colorama.init()**

coloramaモジュールを使うときは最初にinit関数を実行します。あとは、print文で文字を出力する際に、前景色、背景色のキーワード（制御コード）を文字の前に付与するだけです。Fore、Backともに以下のキーワードの色を指定できます。RESETは設定を解除するものです。

'BLACK', 'BLUE', 'CYAN', 'GREEN', 'LIGHTBLACK_EX', 'LIGHTBLUE_EX', 'LIGHTCYAN_EX', 'LIGHTGREEN_EX', 'LIGHTMAGENTA_EX', 'LIGHTRED_EX', 'LIGHTWHITE_EX', 'LIGHTYELLOW_EX', 'MAGENTA', 'RED', 'RESET', 'WHITE', 'YELLOW'

◎実行例

```
>>> from colorama import init, Fore, Back, Style Enter
>>> init() Enter
>>> print(Fore.GREEN + "Hello" + Back.RED + "World") Enter
HelloWorld
>>> print(Fore.YELLOW + "Hello" + Back.BLUE + "World") Enter
HelloWorld
>>> print(Fore.RESET) Enter

>>> print(Back.RESET) Enter
```

◎リスト colorama_test2.py

```
from colorama import init, Fore, Back
init()
for attr in dir(Fore):
      if attr[0] != "_":
                  print(Back.WHITE + getattr(Fore, attr) + "
{0:20}".format(attr), end='')
                  print(Fore.WHITE + getattr(Back, attr) + "
{0:20}".format(attr))
```

◎出力例

ファイルの作成や更新を検出する

サーバやサービスなどには、設定ファイルの更新を反映するために、設定ファイルを再読み込みしなければならないものがあります。そのような場合、ファイルの作成や変更を自動的に検出できれば、再読み込みといった手順を自動化することが可能です。watchdogというモジュールを使って、そのような自動化を実現してみましょう。watchdogモジュールは、一定時間ごとにファイルを監視するのではなく（Pollingするのではなく）、OSレベルで変更を検出するので、処理の負担が少ないのが特徴です。

NOTE watchdogモジュールは標準モジュールには含まれていないので、使用する際には「pip install watchdog」（macOS/Linuxでは「pip3 ～」）でモジュールをインストールしてください。

◎コンストラクタ　Observer()

カテゴリ	**watchdogモジュール**
引数	**なし**
戻り値	**Observerオブジェクト自身**
説明	**Observerオブジェクトを作成する**
使用例	**observer = Observer()**

◎メソッド　schedule(event_handler, path, recursive)

カテゴリ	**watchdogモジュール**
引数	**event_handler：イベントを処理するHandlerオブジェクト** **path：監視するパス、recursive：階層ごと監視するか否か**
戻り値	**WatchDogオブジェクト**

説明	Observerオブジェクトの監視内容の設定を行う。このあとstartメソッドを呼び出すことで実際の監視が開始される

◎リスト watchdog_test.py

```
from watchdog.observers import Observer
from watchdog.events import PatternMatchingEventHandler

observer = Observer() 1

class Handler(PatternMatchingEventHandler):
    def on_created(self, event):
        print("on_created:{}".format(event.src_path))
    def on_modified(self, event):
        print("on_modified:{}".format(event.src_path))
    def on_deleted(self, event):
        print("on_deleted:{}".format(event.src_path))

observer.schedule(event_handler=Handler("*"), path=".",
recursive=True) 2
observer.start() 3
try:
    while True:
        observer.join(1) 4
except KeyboardInterrupt:
    observer.stop()
```

Observerオブジェクトを作成し1、変化を処理するためのハンドラ、モニタするパスを指定してscheduleメソッド2を呼び出します。実際の監視はstartメソッド3で開始します。監視を継続するためにjoinメソッド4を呼び出しています。Ctrl+Cキーでの中断を受信するために、joinをタイムアウト値付きで呼び出しています。

◎実行例

```
C:\Samples\python watchdog_test.py Enter
on_created:.\test\新しいテキスト ドキュメント.txt
on_modified:.\test\新しいテキスト ドキュメント.txt
on_modified:.\test
on_deleted:.\test\test.txt
```

■ 商品に関する問い合わせ先
インプレスブックスのお問い合わせフォームより入力してください。
https://book.impress.co.jp/info/
上記フォームがご利用頂けない場合のメールでの問い合わせ先
info@impress.co.jp

● 本書の内容に関するご質問は、お問い合わせフォーム、メールまたは封書にて書名・ISBN・お名前・電話番号と該当するページや具体的な質問内容、お使いの動作環境などを明記のうえ、お問い合わせください。
● 電話やFAX等でのご質問には対応しておりません。なお、本書の範囲を超える質問に関しましてはお答えできませんのでご了承ください。
● インプレスブックス（https://book.impress.co.jp/）では、本書を含めインプレスの出版物に関するサポート情報などを提供しておりますのでそちらもご覧ください。
● 該当書籍の奥付に記載されている初版発行日から3年が経過した場合、もしくは該当書籍で紹介している製品やサービスについて提供会社によるサポートが終了した場合は、ご質問にお答えしかねる場合があります。

■ 落丁・乱丁本などの問い合わせ先
TEL　03-6837-5016
FAX　03-6837-5023
service@impress.co.jp
（受付時間／10:00-12:00、13:00-17:30　土日、祝祭日を除く）
● 古書店で購入されたものについてはお取り替えできません。

■ 書店／販売店の窓口
株式会社インプレス 受注センター
TEL　048-449-8040
FAX　048-449-8041
株式会社インプレス 出版営業部
TEL　03-6837-4635

逆引き（ぎゃくびき）Python標準（パイソンひょうじゅん）ライブラリ
目的別（もくてきべつ）の基本（きほん）レシピ180+（ぷらす）!

2018年　2月21日 初版第1刷発行
2018年　6月　1日 第1版第2刷発行

著　者　大津真（おおつまこと）、田中賢一郎（たなかけんいちろう）
発行人　土田米一
編集人　高橋隆志
発行所　株式会社インプレス
〒101-0051　東京都千代田区神田神保町一丁目105番地
ホームページ　https://book.impress.co.jp/

印刷所　株式会社廣済堂

ISBN978-4-295-00310-6 C3055

Printed in Japan