

前書き

本書は私がシスコシステムズという会社で働いていた際に提供していた社内トレーニングの資料、及びそれがマイナビニュースで連載となつた際の記事にもとづいて書かれています。シスコはIT業界において有名な会社であるため知っているかたも多いと思いますが、開発というよりネットワークの会社です(USやインドの開発部隊は除く)。つまり、本書のベースとなったトレーニングはITインフラのプロ(当然ながらプログラマーではない)に対して作られたものであるため、他のプログラミングの専門書が飛ばしてしまったり難解に説明したりしているプログラミングの概念をかなり丁寧に説明しています。本書を通してプログラミングがどのようなものか理解していただき、コードを書くことに抵抗がなくなつて頂ければ幸いです。なお、本書は前編・中編・後編のPython3シリーズの前編となり、プログラミングの基礎を丁寧に説明することに注力しています。オブジェクト指向の詳細やその他レベルの高い話題はそれほど深く扱わず、それらは本書の続編で扱います。

プログラミングは建築にたとえることができます。たとえば犬小屋を作ろうと思ったとき、木の柱や板を調達して適当にデザインして組み立てるすることができます。構造力学の計算をするどころか、建築について素人であっても作れます。ただ、犬小屋を作るには「犬小屋とはこういうものだ」ということと、ノコギリとハンマーの使い方は知っている必要があります。プログラミングもこれと同じで、比較的「小さい」プログラムはノコギリやハンマーに相当する最低限の文法と「作りたいものの仕組みをどう実現するか」ということさえ知つれば、そこそこ動くものを作ることができます。しかし、人が住む家を作るとなると、犬小屋とは話が変わってきます。頑丈であり住みやすい家を作るには、まず「家というものの仕組み」を犬小屋よりも深いレベルで知っている必要があります。なおかつ家の工作中に必要となるスキルは犬小屋より高度なものとなります。ビルの建設などになってくるとさらに高度な知識が必要になります。プログラミングも、単に文法や「言語や設計の思想」だけでなく周辺知識のようなものを身につけていないと、数千、数万行レベルのコードを1人で書いたり、プロジェクトを回したりすることは難しいと思います。より大きく複雑なものを作る場合ほど、より深いレベルの知識が必要になってきます。

まずは犬小屋を作るために、ハンマーやノコギリの使い方を正しく覚える。それがずばり本書の目的です。



初級者



中級者



上級者

本書のゴール

(本業プログラマー向け書籍)

ハンマーやノコギリの使い方を座学だけで学んで、「私はハンマーとノコギリを使えます」といったら、大工さんに笑われてしまします。使い方は教科書でも学べますが、使いこなそうと思うと、実際に木を切って、釘を打たないと、そのスキルは身につきません。使ってみてはじめてわかることが多いはずです。プログラミングも本で座学するだけでなく、実際にコードを書いて、動かすことで上達します。ゆっくりでもいいので、実際に手を動かしながら本を読み進めていってもらえば幸いです。

執筆書籍一覧 <http://www.yuichi.com/books.html>

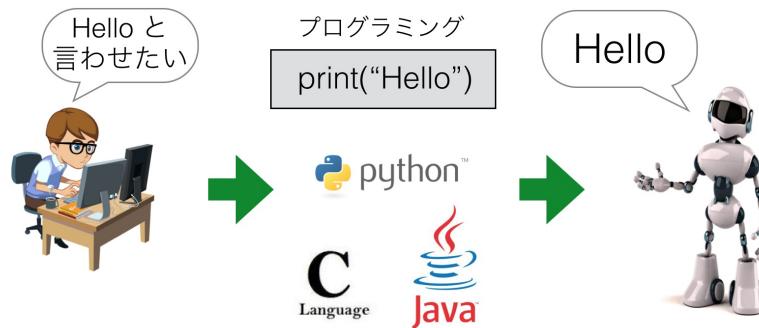
目次

1. プログラミングとはそもそもなにか
2. なぜPythonを学ぶのか
3. Python 3 のインストールと実行
4. 本書の概要
5. 型と変数
6. Python で重要な型
7. 制御構造
8. 関数
9. モジュール
10. オブジェクトとメソッド
11. テキスト処理
12. リスト再入門
13. タプル
14. セット
15. 辞書型
16. ファイル処理
17. 基数と文字コードの仕組み
18. 日本語の扱い
19. プログラムへの入力
20. 関数の高度な使い方
21. 終わりに

プログラミングとはそもそもなにか

機械は人間の言葉をそのまま理解することはできません。0,1のバイナリで書かれた命令をメモリに読み込み、CPUがそれを実行することで機械は人間の命令を実現します。PCのディスプレイに"Hello"と表示させるという命令も、つきつめると0,1で構成される機械が理解できる命令になります。ただ、「画面に Hello と出力することを機械が理解できるように0,1で命令して、実行せよ」といわれても、どういう0,1の羅列を書けばいいか分からずに途方にくれてしまいます。0,1には規則性があるため理論上は0,1で命令を書くことはできるはずですが、誰もそんなことはやりたくありません。

プログラミングはこの「機械が理解できるように0,1で命令を与える」という難しい作業を人間がより簡単に行うための方法です。ざっくりいってみると、人間と機械の両方にとってわかりやすい「中間言語」で人間は機械が解釈できる命令を書き、機械はそれを0,1に変換して実行するという流れです。このプロセスが「プログラミング」と呼ばれ、その人と機械の間の中間言語が「プログラミング言語」となります。以下にこの概念を示す図を記載します。



図の真ん中に位置する「C、Java、Python」がプログラミング言語です。これら以外にも多くの言語があります。図にあるように人間が「機械にHelloと言わせたい」と思ったらプログラミング言語で「Helloと出力しろ」と命令を書きます。上記図の「print("Hello")」がその命令です。機械はその「print("Hello")」は解釈できるので、命令されたように動きます。

"Hello"を表示するために「0,1の規則性を勉強して、0,1を延々と羅列して命令を書く」と「プログラミング言語で print("Hello")と書く」ことを比べたら、どう考えても後者のほうがずっと簡単で効率的です。そのためプログラミング言語を使います。

*広義では0,1で命令を羅列することもプログラミングといえます。ただ、今ではあまり一般的ではないと考えています。

なぜ Python を学ぶのか

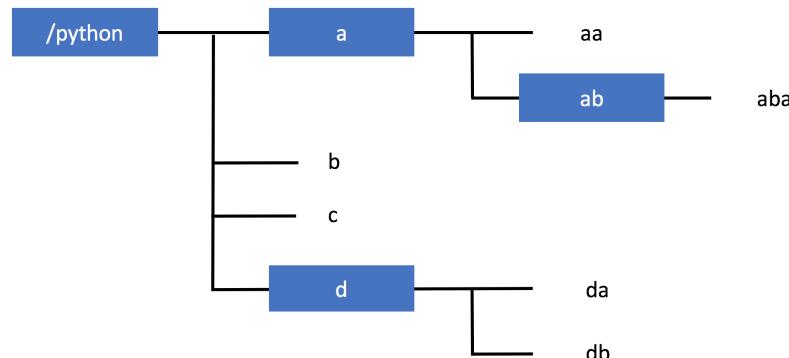
プログラミングを実現する手法や言語はさまざまです。どのプログラミング言語であっても自分がやりたいことはおおよそ実現可能です。ただ、重要なのはプログラミング言語にはそれぞれ得意な分野があるということです。つまり、プログラムで実現したいことにより、適切な言語は変わるということです。今回学ぶPythonは、そのようなプログラミング言語のひとつであり、他の言語よりも汎用的(なんにでも使える)であり簡単です。先程、プログラミング言語は人間と機械の中間にあると説明しましたが、他の言語よりも人間よりだということです。そのためプログラミング未経験者にとってPythonはかなりいい言語だと考えられています。例えば、有名なMIT(マサチューセッツ工科大学)などでもPythonを大学の最初のプログラミング授業で扱っています。

Pythonは汎用的であり簡単だと思いましたが、それは具体的にはどういうことでしょうか。まずは論より証拠を見せたほうがはやいと思いますので、「ディレクトリ(フォルダ)の階層を書き出すプログラムを作る」ということを複数の言語でを行い、それを比較してみます。同じ内容を複数の言語で書けば、それぞれの言語の特徴がよく分かります。

プログラムを書くにあたっては、まず「それをどう実現するか」について理解する必要があるため、「ディレクトリ(フォルダ)の階層を書き出す」方法について考えます。実現方法はいろいろありますが、今回は以下のルールに従って動かすことで実現させます。

1. ディレクトリの中身を順に見ていく
2. それがファイルであればそのままファイル名を表示。ディレクトリであれば、インデント(字下げ)してディレクトリ名を[]付きで表示し1.に戻る
3. ディレクトリ内のファイルをすべてチェックしたら終えたら終了

たとえば、以下のようなディレクトリ構造があるとしましょう。四角で囲われているのがディレクトリで、Pythonディレクトリが起点となっています。



プログラムでこれを表示させると次のようにになります。

```
[/python]
[ /python/a]
- aa
[ /python/a/ab]
- aba
- b
- c
[ /python/d]
- da
- db
```

上記のルールにしたがって、この構造のディレクトリを表示させた際のプログラムの動きを具体的に書きくだすと、以下のようになります。

1. pythonディレクトリの中を確認
2. aはディレクトリなのでインデントしてディレクトリを表示
3. aディレクトリの中を確認
4. aaはファイルなので表示
5. abはディレクトリなので、更にインデントしてディレクトリ名を表示
6. abディレクトリの中を確認
7. abaはファイルなので表示
8. abディレクトリの中身を確認し終えたので、aディレクトリの確認の続きを戻る
9. aディレクトリの中身を確認し終えたので、pythonディレクトリの中の確認の続きを戻る
10. bはファイルなので表示
11. 以下省略

ルールに従って規則的に動いていることがわかります。この自分で書いた「プログラムの挙動のルール」をアルゴリズムと呼びます。

では、実際にこれを実現するプログラムの例を、いくつかの有名な言語で記載していきます。各言語について知らない方も多いと思うので、プログラムが実際に何をしているかということよりも、ぼーっと眺めることで、各プログラミング言語の違いについていろいろと思いを馳せていただけたらよいと思います。なお、比較を簡単にするために、必ずしも教科書的な書き方をしていないので注意して下さい。

Pythonの例

まずPythonのプログラムを以下に記載します。日本語のコメントを除けば全部で約15行となっています。

```
import os

def list_file(path, indent_level):
    # ディレクトリ名を表示
    print('{0}{1}'.format(' '*indent_level, path))

    # ディレクトリ内のファイルとディレクトリを全てループで確認
    for file_name in os.listdir(path):
        if(file_name.startswith('.')): continue
        abs_filepath = path + '/' + file_name
        if(os.path.isdir(abs_filepath)):
            # ディレクトリだったので、そのディレクトリをチェックする
            list_file(abs_filepath, indent_level + 1)
        else:
            # ファイルだったので、ファイル名を表示
            print('{0}- {1}'.format(' '*indent_level, file_name))

list_file('/python', 0)
```

コード中にある「#」以降がプログラムのコメントであり、どのような処理をしているかメモしています。

Javaの例

Javaで同じことをするプログラムを書くと、以下のようになります。Pythonだと15行程度だったプログラムが、およそ30行になっています。

```
import java.io.File;
// クラス定義
public class Main {

    // エントリーポイント(プログラムの起点)
    public static void main(String[] args){
        Main m = new Main();
        m.listFiles("/python", 0);
    }

    // インデントのためのユーティリティ関数
    public void printIndent(int indentLevel){
        for(int i=0; i<indentLevel; i++){
            System.out.print("    ");
        }
    }

    public void listFile(String dirPath, int indentLevel){
        // ディレクトリ名を表示
        printIndent(indentLevel);
        System.out.printf("[%s]\n", dirPath);

        File dir = new File(dirPath);
        File files[] = dir.listFiles();
        for(int i=0; i<files.length; i++){
            // ディレクトリ内のファイルとディレクトリを全て確認
            if(files[i].getName().startsWith(".")) continue;

            if(files[i].isDirectory()){
                // ディレクトリだったので、そのディレクトリをチェックする
                listFile(files[i].getAbsolutePath(), indentLevel + 1);
            }else{
                // ファイルだったので、ファイル名を表示
                printIndent(indentLevel);
                System.out.printf("- %s\n", files[i].getName());
            }
        }
    }
}
```

"public void listFile(String dirPath, int indentLevel){"というものが真ん中にあります、これがディレクトリを走査するアルゴリズムです。Pythonと若干異なりますが、やっていることはほとんど同じです。プログラムに詳しい人が見ると、増えたのはJavaによる言語の制約(Classの宣言が必要なことなど)や、文字列処理あたりにあることがわかります。メインの処理である"public void listFile" の下にある処理は、Pythonとさほど変わっていません。同じことを実現するにしても Python のほうがずっと簡単に見えます。

プログラムの実行方法に関しても、Pythonのほうが Java よりも簡単です。Python であればこのプログラムファイルを「python プログラム名」というコマンドで呼び出すだけですが、Javaは「コンパイル」と呼ばれる作業で上記のテキストで書かれたプログラムを0、1のバイナリにしてあげて、それをコマンドで指定して実行するという方式になります。

C言語の例

最後にC言語のプログラムです。これは約50行となっています。どうです、もう読むのが面倒くさくなってきたのではないですか。書くのはもっと面倒でした(笑)。面倒なだけではなく、CはPythonやJavaと違い、プログラムが「実行するOS」に強く依存するという問題があります。下記のコードはMacで書いたものなので、Windowsでは動かないのではないかでしょうか。

```
#include <stdio.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/stat.h>
#include <string.h>

void printIndent(int indentLevel);
void listFile(char path[], int indentLevel);

// エントリーポイント(プログラムの起点)
int main(int argc, const char * argv[]) {
    listFile("/python", 0);
    return 0;
}

// インデントのためのユーティリティ関数
void printIndent(int indentLevel){
    int i;
    for(i=0; i<indentLevel; i++){
        printf("    ");
    }
}

void listFile(char dirPath[], int indentLevel){
    DIR *dir;
    struct dirent *dp;
    int i=0;
    struct stat stat_buf;

    printIndent(indentLevel);
    printf("[%s]\n", dirPath); // ディレクトリ名を表示

    dir = opendir(dirPath);
    // ディレクトリ内のファイルとディレクトリを全て確認
    for(i = 0; NULL != (dp=readdir(dir)); i++){
        const char* fileName = dp->d_name;
        if('.' == fileName[0]) continue;

        char cp[512];
        strcpy(cp, dirPath);
        strcat(cp, "/");
        strcat(cp, fileName);
        stat(cp, &stat_buf);
        if(S_ISDIR(stat_buf.st_mode)){
            // ディレクトリだったので、そのディレクトリをチェックする
            listFile(cp, indentLevel + 1);
        }else{
            // ファイルだったので、ファイル名を表示
            printIndent(indentLevel);
            printf("- %s\n", fileName);
        }
    }
    closedir(dir);
}
```

"void listFile(char dirPath[], int indentLevel){" の下にあるプログラムがディレクトリを走査するプログラムです。先に提示したPythonやJavaと同じアルゴリズムですが、ずいぶんと長くなっています。

言語の比較

いくつかのプログラミング言語で同一の処理を書きましたが、ここで知ってほしいことは各プログラミング言語の詳細というよりも「Aという処理を、Pythonだと2行で実現できるのに、Cだと5行必要」というような具合で、言語ごとにプログラムを書く労力が大きく違うということです。処理を書けば書くほどプログラムの文量がどんどん増えていくので、今回の例では Pythonと C 言語の3倍以上のコード行数が必要となってしまっています。

また、Cをわかる人が見ればすぐに見抜かれてしまいますが、上記は非常に問題の多いコードです。たとえばディレクトリが深くなったり、ファイル名が長かったりすると一気に破綻してしまいます。本来であれば、そのあたりも考慮したコードを書かなければいけないはずです。ディレクトリ構造を書き出すという処理をさせるのであれば、あまりC言語は使いたくないという感想を持っていただけたのではないか。

最後に、Python、Java、Cの特徴を比較した図を記載します。



シンプルで簡単

すぐ使えるようになる
初心者向き



覚えることが多い

オブジェクト指向を
本格的に学びたいなら
オススメ



難しい

低レイヤには必須

PythonはC言語やJavaに比べると、同じ処理を短いプログラムで実現することができます。短いということをいいかえると、PythonはCやJavaより「簡単」な言語であるともいえます。私の個人的な意見なのですが、初心者がまず学ぶべきことは、プログラミング言語の仕様詳細よりも、そのプログラミング作業を行う際の考え方や思想だと思います。複雑な処理を実現したいときに「処理を小さい単位に分解して、どのようなステップで実現するか順序立てて考える」ことが必ず必要となります。それを学ぶのであれば、Cのように余計な作法に振り回されて本質に注力できないプログラミング言語よりも、Pythonのような簡単な言語で「プログラミングの本質」に着目するほうがよいと思われます。そのため私は、プログラミング初心者にはPythonからはじめることを推奨しています。

余談ですが、Pythonは簡単なだけではなく、OpenStackをはじめとした多くの大規模なソフトウェアプロジェクトで採用されていたり、GoogleやCiscoといったメジャーなIT企業でも積極的に利用されています。「エキスパートたちがわざわざ選ぶのだから間違いない」と安直に考えることもできます。

Python 3 のインストールと実行

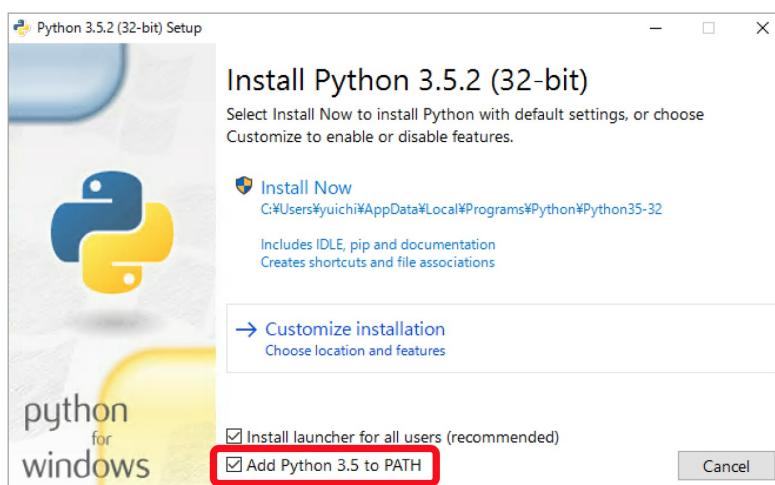
Windows 10 へのインストール

さっそく Python を使う準備にとりかかりましょう。Windows にはデフォルトで Python はインストールされていません。そのため Python を使うためにはインストールが必須です。インストールには Python の公式サイト (<https://www.python.org/>) から実行環境にあった Python のインストールイメージをダウンロードしてきます。



Python には現在 2.x と 3.x のバージョンがありますが、本書では Python 3 を利用するため、3.x の最新版をインストールしてください。Python 2.x と 3.x のプログラムには一部互換性がないので、2 で作ったコードを 3 で使うことや、その逆ができないことがありますので注意してください。ちなみに Python 2 と 3 の両方をインストールして使い分けることも可能ですので、最終的に 2.x を使いたいのかとも、3.x をインストールして大丈夫です。

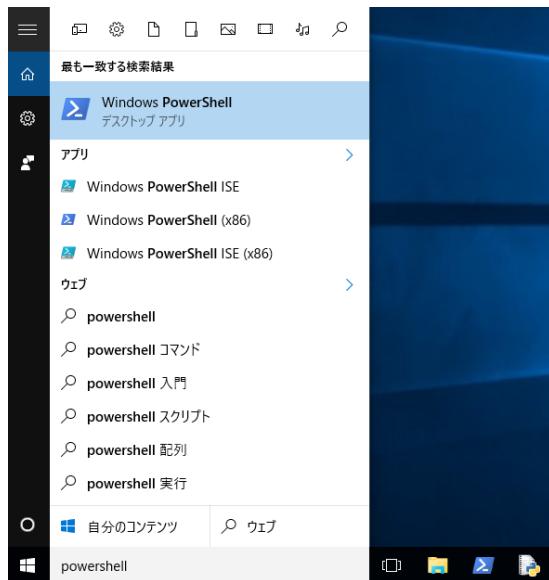
上記ページから落としてきたイメージからインストーラーを立ち上げると以下のような画面が現れると思います。



この選択画面で赤の枠囲みがされている場所をチェックすると、「環境変数」と呼ばれるアプリケーションの所在地を示す情報が Windows に自動で登録されますのでチェックをして下さい。チェックを忘れた場合は手動で「パスを通す」という作業が必要になります。検索してもらえればパスの通しかたは分かるはずです。

'Install Now' と 'Customize installation' の選択肢がありますが、この Windows ユーザだけを使うのであれば 'Install Now' で構いません。他のユーザーが使ったり、凝った使い方をしたりしたい場合は Windows のルートディレクトリ直下や Program Files にインストールする必要があります。インストール作業は難しくないので割愛します。

さっそくインストールを終えた Python を利用してみたいと思います。パワーシェルなりコマンドプロンプトなりをたちあげるとコンソールが現れますので、そこから python を使うことができます。パワーシェル及びコマンドプロンプトは場所がわからなくても Windows の検索機能を使うことで見つけられるかと思います。参考のために以下に画像を記載します。



パワーシェルやコマンドプロンプトは何度も使うことになるので、タスクバーにピン留めしてすぐに使えるようにしておくと今後が楽です。

プロンプトから以下のように“python --version”と打ってPythonを起動してみてください。先ほどのインストール作業時にパスが登録されていれば以下の出力が得られると思います。そのようなものがないと怒られた場合はインストールができていないかパスが通っていないかもしれません。パスの通し方は検索してもらえればすぐに分かるはずです。

```

選択 Windows PowerShell
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

PS C:\Users\yuichi> python --version
Python 3.5.2
PS C:\Users\yuichi>

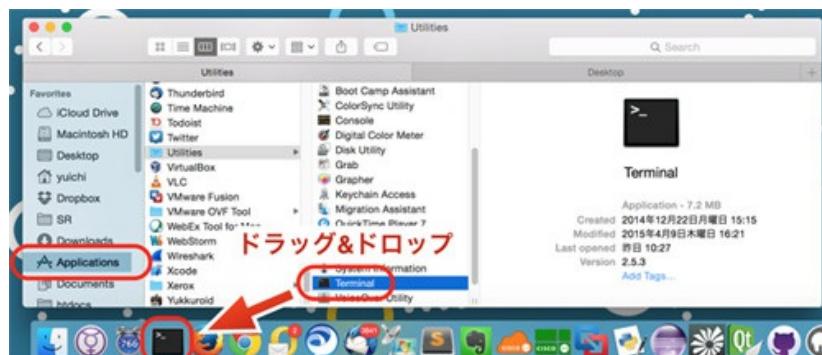
```

なお、本書のプログラムは Mac で書いて実行しているため、微妙に Windows と食い違いが発生するかもしれません。大きな差ですと明記しますが、コマンド名やOSのディレクトリ構造といった小さい箇所は適時読み変えて下さい。また、「ターミナル」と言った場合はパワーシェルやコマンドプロンプトのことを指します。

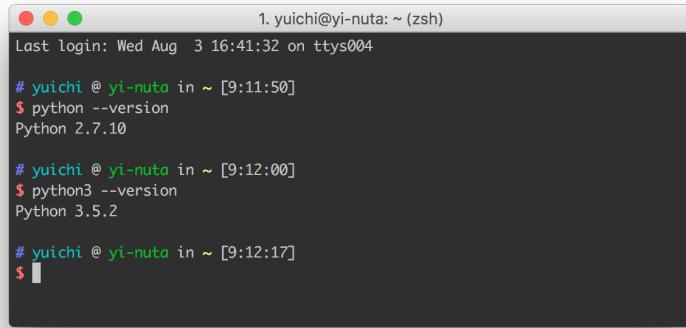
Macへのインストール

Windows と異なり、Mac にはデフォルトで Python がインストールされています。ただ、インストールされているものは Python2 のため、Windowsへのインストールと同じようにサイトへ行き、Python3 のイメージをダウンロードしてきてインストールを行って下さい。

Mac での Python の実行はターミナル(コマンドライン)から行います。ターミナルは「アプリケーション」フォルダ配下の「ユーティリティ(Utility)」の下にあります。今後頻繁に使うことになるのでドラッグ&ドロップでDockに登録しておいてもいいかもしれません。



ターミナルを起動して、“python --version”と打つと、おそらく Python2 のバージョンが表示されます。Python3 の確認には“python3 --version”と打つ必要があります。



1. yuichi@yi-nuta: ~ (zsh)
Last login: Wed Aug 3 16:41:32 on ttys004
yuichi @ yi-nuta in ~ [9:11:50]
\$ python --version
Python 2.7.10
yuichi @ yi-nuta in ~ [9:12:00]
\$ python3 --version
Python 3.5.2
yuichi @ yi-nuta in ~ [9:12:17]
\$

複数のバージョンの Python がインストールされているため、このような挙動となります。“python”の後にtabを打つと、インストールされているバージョンの一覧が確認できます。

```
$ python
python      python2.7-config  python3.5-32    pythonw2.6
python-config   python3        python3.5-config  pythonw2.7
python2.6       python3-32     python3.5m      python2.7
python2.6-config  python3-config  python3.5m-config
python2.7       python3.5      pythonw
```

Python3 を使うつもりで間違えて Python2 を使ったり、その逆が発生したりしないように注意をしてください。

Python の実行

インストールが完了したので、さっそくPythonを使ってみましょう。“--version”オプションなしで“python”と打つと、Pythonのプロンプト画面に入れます。

```
$ python3
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

画面に以下のプロンプトが表示されます。

```
>>>
```

それに続けてPythonのプログラムを書いて実行させます。とりあえず計算させたり、文字を画面に表示させたりしてみます。

```
>>> 1 + 1
2
>>> print('hello')
hello
>>>
```

1 + 1 を入力すれば、計算された値 2 が帰ってきて、hello を print しろと命令すれば hello と表示されました。これも立派なプログラミング作業です。このようにプロンプトでは「対話的」にプログラミングを行います。簡単なプログラムの挙動確認であればプロンプトを使うことが多いです。なお、この画面は「インタプリタ」とも呼ばれます。

プロンプトはプログラムのちょっとした挙動確認には便利なのですが、いわゆるプログラムのソースコード(コード)を書くためのものではありません。プロンプトの次は「IDLE」と呼ばれるPythonの開発環境を使ってコードを書いてみます。IDLEは先ほどのようなプロンプト画面も使えますし、Pythonを書くのに適したエディタの機能も持っています。では、さっそくIDLEを立ち上げてみましょう。

Windowsは、Windows メニューにある「すべてのプログラム」から「Python」を選択し、その下にある「IDLE (Python GUI)」を選べば起動します。Macは、アプリケーションフォルダ配下の「Python」ディレクトリの下に「IDLE」がありますので、それをダブルクリックして起動させます。パワーシェルやターミナルと同じように、IDLEもタスクバー や Dock に登録してしまってもいいかもしれません。

起動すると以下のような画面が立ち上がります。プロンプトと同じなので、計算をさせたり、文字を表示させたりすることもできます。

```

Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.

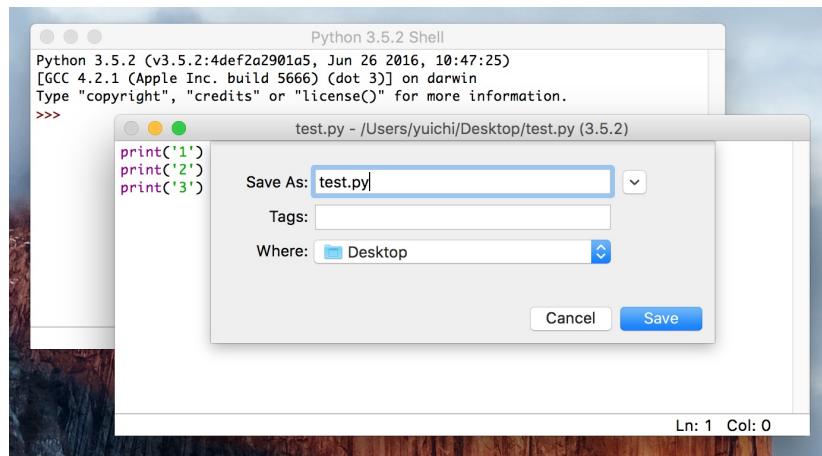
>>> 1+1
2
>>>

```

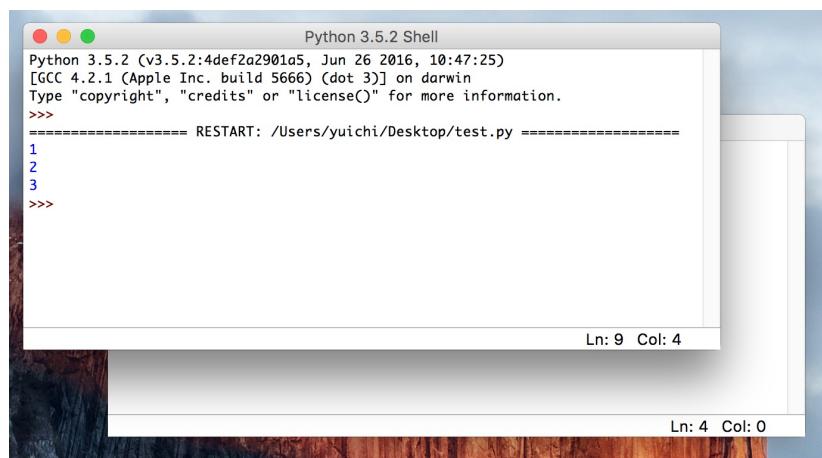
Ln: 9 Col: 4

プロンプトに1行1行書いて実行させるだけでなく、ファイルにPythonのプログラムを書いて実行させることもできます。IDLEのメニューから [File] -> [New File] とすることでエディタが開きます。これに以下の文を書き込んで「test.py」というファイル名で保存([File] -> [Save])してください。ファイルの名前はなんでもいいのですが、Pythonのプログラムの拡張子は「.py」であるということは覚えておいてください。

```
print('1')
print('2')
print('3')
```



プログラムの内容は1、2、3と順番に画面に出力させるだけの簡単なものです。IDLEのエディタが選択されている状態で [F5] ボタンを押すと、このファイルをPythonで実行できます(Macで「F5」を実行するには、「fn」キーを押しながら [F5] ボタンを押す必要があります)。実行するにはファイルが保存されている必要があります。F5を押すとプロンプトのウィンドウが前面に来て、プログラムの実行結果が出力されます。

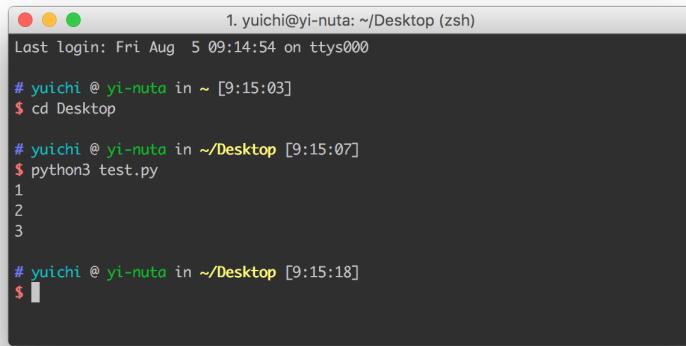


なお、先ほど作成したファイル「test.py」を開く際に、利用するアプリケーションとしてIDLEを選択することもできます。Windowsですと右クリックを押すと「IDLEで編集 (Edit with IDLE)」と出るので、そこからファイルを直接開けます。Macだと右クリック([Control] を押しながらクリック)して、「IDLEで開く」と指定することで実現できます。もちろん、IDELEのメニューから既存のファイルを開くこともできます。

先程はIDLEからファイルの実行をしましたが、プロンプトから python のプログラムファイルを実行することも可能です。ターミナルやコマンドプロンプトで、pythonコマンドに続けてファイルを指定することで、ファイルが実行されます。

先ほどIDLEで作成した「test.py」をターミナルから起動してみます。IDLEと同じ出力があることからプログラムがきちんと動いていること

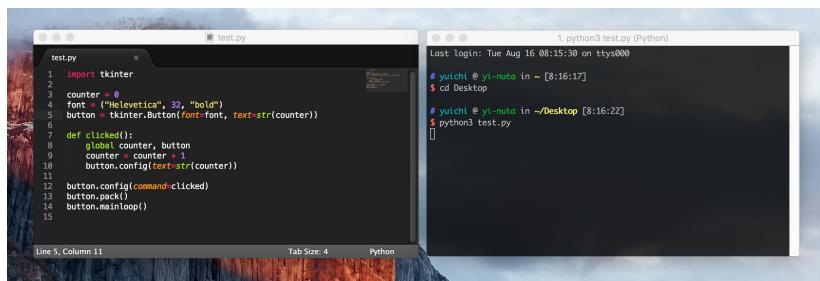
が分かります。



```
1. yuichi@yi-nuta: ~/Desktop (zsh)
Last login: Fri Aug  5 09:14:54 on ttys000
# yuichi @ yi-nuta in ~ [9:15:03]
$ cd Desktop
# yuichi @ yi-nuta in ~/Desktop [9:15:07]
$ python3 test.py
1
2
3
# yuichi @ yi-nuta in ~/Desktop [9:15:18]
$
```

また、プロンプトを見てもらうと分かるように「pythonファイルがあるディレクトリ」に移動してからファイル名を与えてプログラムを実行しています。違うディレクトリで実行するファイルを指定する場合は、相対パス(自分がいる場所を起点としたファイルやディレクトリの場所)なり、絶対パス(ファイルシステムのルートを起点としたファイルの場所)なりを指定する必要があります。

なお、以下のように自分の好きな高機能エディタとターミナルを並べて開発するスタイルや、PyCharmといったIDE(統合開発環境)を使った開発が現場では一般的です。ある程度コードが書けるようになったら自分にあった開発環境を模索してみてもいいかもしれません。



Python コマンドで対話型のインタプリタを呼び出したりプログラムファイルを実行したりすることがほとんどかと思いますが、1ライナーと呼ばれる一行のプログラムを実行することができます。それには -c オプションを使い、それに続けて実行したい Python のコードを書きます。

```
$ python3 -c "print(1+1)"
```

上記のダブルクオートに囲まれたpython のコードがその場で実行され、2という出力が得られています。あまり複雑なことには使えませんが、ある処理をコピー・ペーストで実行したり、SSH 経由で他のマシンに実行させたりする場合に便利です。

コメント

python のプログラムに注釈をつけたい場合があります。例えば「この処理はこれこれをしています」といったことを複雑なプログラムの最中に書いておけば、そのコードを読んだ人にとって非常に分かりやすくなります。Python では「プログラムとして解釈されないテキスト」を書くことでこれを実現し、それをコメントと呼んでいます。

```
# 5 + 5 の結果を表示する
print(5 + 5)

print(5 - 3) # 5 - 3 の結果を表示する

'''複数行の
コメントも
できる'''
print(5)
```

上記のプログラムでは # (シャープまたはハッシュと呼ばれる) から後はコメントとして Python に解釈されるため、実行されません。プログラムを読む「人」以外にとっては全く意味のないものです。

1つめの例のようにコメントに1行を割り当てることが一般的ですが、短いコメントであれば2つめのように Python の式のあとにコメントを書くこともできます。このとき、# 以降のみがコメントとして扱われる所以、# より前は Python に解釈されます。ちなみにこのような行に埋め込むコメントはインラインコメントと呼ばれています。

3つめの例は複数行にまたがるコメントです。# をたてに並べて複数行のコメントを書くこともできますが、シングルクオテーションまたはダブルクオテーション3つでテキストを囲むことでそれをコメントとして使うことができます。これは後ほど改めて紹介します。

本シリーズではプログラムの出力をコメントとして表現します。たとえば以下のプログラムがあるとしましょう。

```
print(5 + 5)
print(5 - 3)
print(5)
```

これを実行すると以下のような出力が得られます。

```
$ python3 test.py  
10  
2  
5
```

このとき、本書に記載するコードは以下のように出力をコードにコメントとして埋め込みます。

```
print(5 + 5)  
# 10  
print(5 - 3)  
# 2  
print(5)  
# 5
```

また複雑なコードを書く場合も本来の目的でコメントを使います。自分でコードを書く際も積極的にコメントを書くようにして下さい。ただ、コメントでわざわざ書かなくてもいい自明のことは書かないようにして下さい。

エラーの読み方

今後プログラムを書くにあたり、必ず間違いが発生します。そのとき Python はエラーを出力しますので、それを見て何が間違っているのか把握し、それを修正する必要がでできます。詳細は扱いませんが、簡単にエラーの読み方について紹介します。

まず以下のプログラム test.py があるとします。

```
5 + 5  
5 ~ 5  
5 - 5
```

これを実行すると以下のようなエラー出力が得られます。

```
$ python3 test.py  
File "test.py", line 2  
 5 ~ 5  
      ^  
SyntaxError: invalid syntax
```

エラーの出力には、「test.py」というファイルの「2行目」にある「5 ~ 5」で「SyntaxError: invalid syntax」というエラーが発生しているということが書かれています。これを見てプログラムの誤った箇所を修正できます。

エラーの意味が分からぬ場合はそれを Google などでそのまま検索してください。必要であれば python などのキーワードもつけて下さい。検索すれば丁寧に説明しているドキュメントやページが表示されるかと思います。

概要

プログラミングの学習の道のりは長いです。そのため細かいことを説明する前に、本章でざっと全体像を語ってしまいたいと思います。それぞれの細かい解説はありませんが、分からなくとも学習を進めると理解でききますので、まずは深いことは考えずに手を動かしてプログラミングでどのようなことができるのか感じてみてください。

では、さっそくはじめていきましょう。Pythonのプロンプトを起動してください。多くのプログラミング言語は、一行一行に順に命令を書いていくことで複雑な処理を実現します。Pythonもそれと同じです。まずは簡単な数値計算をさせてみます。計算式を書くと、その結果が返されます。

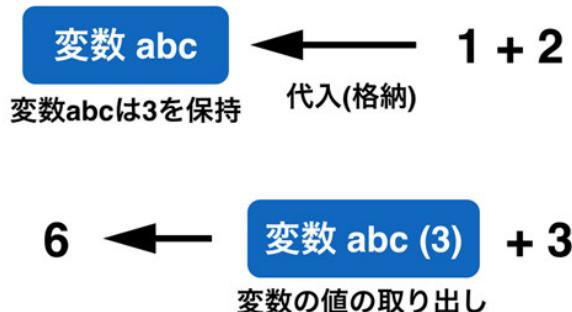
```
>>> 1 + 2  
3  
>>> 3 - 4  
-1  
>>> 5 * 6  
30
```

「*」は掛け算となります。

上記も立派なプログラムなのですが、一行でできることは限られていますので、単純な処理しか書くことができません。より複雑なことをさせる場合は、1行目の結果を2行目で利用するといった具合に、複数行を組み合わせて実行する必要があります。この場合、「処理の結果を保存する」ことが必要となり、そのために「変数」を使います。以下に具体例を示します。

```
>>> abc = 1 + 2  
>>> abc + 3  
6
```

上の例では $1 + 2$ の結果を「abc」に保存して、その結果に $+ 3$ して6を得ています。この保存に利用しているabcが「変数」で、=記号の右側の結果を、左側の変数に格納します。変数を利用する文字はabcでなくても構わず、名前付けのルールさえ守れば自分が好きなものを使うことができます。ちなみに、変数に値を格納することを「代入する」といいます。変数の概念を以下の図に記します。



次に「関数」について扱います。プログラムは、処理を一行一行書いていくことを繰り返して作られるのですが、全ての機能を「自分で書く」のには限界があります。たとえばファイルの内容を読み込む処理をしたいとした場合、その処理は「 $1 + 1$ 」とは根本的に違います。

「ファイルを開いて、それを読み込む(readする)」といった処理が必要となります。そのような処理は「すでに誰かが作った処理」を呼び出すことで実現します。具体的には、「ファイルを開く処理」を呼び出し、その後で「ファイルの中身を読む処理」を呼び出します。この呼び出しが「関数」と呼ばれる機能を使うことで実現できます。ファイルの読み出しだと話が難しいので、もっと簡単な「数の絶対値を得る」という関数を利用してみます。

```
>>> abs(-5)  
5  
>>> value = abs(5)  
>>> value  
5
```

「abs()」を使うことで-5の絶対値である5を得ています。またその次に、5の絶対値である5も得ています。absが関数の名前で()の中の数字が関数に与える値です。ここでは、関数への入力値として-5を与えて、出力値として5を得ています。例にあるように、関数から返された値を変数に格納することもできます。

この関数absを使うことで、自分で絶対値を得る具体的な処理を書かなくても絶対値を得ることができます。絶対値を得るぐらいの処理でしたら自分で書いてもいいでしょうが、先ほどのファイル処理やネットワークの利用などはそもそも関数を使わないと実現不可能です。なお、absはよく使われる処理なので「組み込み関数」という「すぐに利用できる特別な関数」として提供されています。

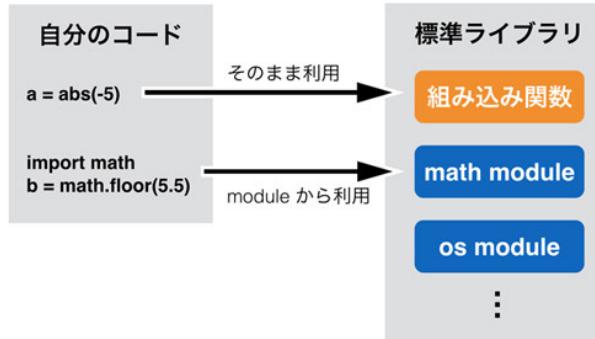
次に「小数点の切り捨て」をしたいと思います。小数点の切り捨ても関数として提供されていますが、絶対値を得るabsほどよく使われる処理ではないので「組み込み関数」としては提供されていません。「標準ライブラリ」という特定の処理をするためのツールセットのひとつの機能として提供されています。

標準ライブラリはそのなかに、特定機能を担当するモジュールをいくつも持っています。たとえば、数学関連の「mathモジュール」や、OSの機能を利用するための「osモジュール」などがあります。組み込み関数以外の「モジュールに属する関数」を使う場合は、まず「〇〇」というモジュールを使いますよ」という宣言をする必要があります。

切り捨ては「数学」的に利用される機能ですので、今回はmathモジュールを使います。具体的には以下のようになります。

```
>>> import math  
>>> math.floor(5.5)  
5.0
```

「mathモジュールをimportして使いますよ」と宣言したあとで、mathモジュールのfloor関数を呼び出しています。モジュールに属する関数は、“モジュール名.関数名”というスタイルで呼び出すことができます。5.5を与えて5.0を得ているのは先ほどのabsと同じですが、関数名の前にモジュール名がついているところが違います。組み込み関数とモジュールの利用方法を以下に図示します。



なおmathモジュールにはほかの機能もあり、floorはそのひとつにすぎません。切り捨てがあるのですから切り上げも当然ながらあります。

また、absやfloorのように「処理した結果を返す関数」だけでなく、「結果を返さない関数」もあります。たとえば今まで使っていたprint関数は、受け取った文字や数字を画面に出力するための関数です。この関数を呼び出すのは「画面に出力をする」ためであり、何か値を得ることを目的としたものではありません。

```
>>> print('hello')
hello
>>> abc = 3
>>> print(abc)
3
```

上のように変数に何が入っているか確認する用途に利用できるので、プログラムの挙動を確認するのに便利です。本書のコードがどういう動きをしているか分からぬ場合は自分でコードにprint文を挟むことで、どこをどう実行しているかといったことや、変数になにが入っているかということを確認してみてください。

さきほど絶対値を返す関数absを使ってみました。このような「関数」を自分で作ることも可能です。absのように誰にでも使われる処理だと、すでに提供されている可能性が高いのです。一方、自分だけが使うような処理だと自分で関数を作る必要がでてきます。関数を作成する理由は「プログラムを整理する」ためです。また「同じ処理を何度も呼び出す」ためでもあります。関数作成の方法を簡単に紹介します。関数作成の手始めに“hello”と出力する簡単な関数「print_hello」を作ってみます。複数行書く必要がでてきたため、プロンプトではなくファイルにコードを書いてためします。ここではtest.pyとしています。

```
def print_hello():
    print('hello')
```

上記は関数の宣言(名前の登録)と処理の実装(機能を作る)をしています。作った関数を使うためには、それを呼び出します。

```
def print_hello():
    print('hello')

print_hello()
```

これを実行すると以下のように出力されます。

```
hello
```

print_hello()を2回呼び出せば、helloが2回出力されます。当然ですが、関数を定義せずに関数を呼びだそうとすると「そんな関数ないよ」と怒られてしまいます。以前使った関数absやfloorはすでに定義されているので、自分で定義をしなくても呼び出すことができます。

次にprint_helloよりもう少し複雑な関数「add5」を作ってみます。名前からわかると思いますが、受け取った数字に5を加えたものを返す関数です。

```
def add5(x):
    y = x + 5
    return y
```

さっそくを使います。

```
def add5(x):
    y = x + 5
    return y

z = add5(5)
print(z)
# 10
```

先ほどのprint_helloと同じように、関数の定義をdefで行い、add5を呼び出すことで関数を利用しています。ただ、defで定義する際にadd5()の中にxという変数が書かれています。そして関数の定義の最後にreturnという命令が加えられています。このxは関数を呼び出す際に「値を受け取る」ことを意味しています。absが値を受け取るのと同じように、add5も値を受け取り、それがxに格納されるのです。そしてそのxに5を加えた値をyに格納し、それをreturnで関数の呼び出しもとに返しています。absが絶対値を返すように、add5は5を加えた値を返しています。

なお、わかりやすくするために、上の例では命令を一行にひとつしか書いていませんが、一行に複数の命令を書くことも可能です。上記のadd5の例では、以下のようになります。

```
def add5(x):
    return x + 5

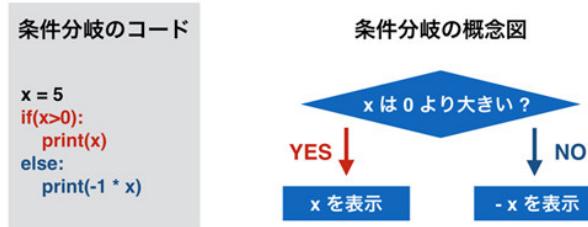
print(add5(5))
# 10
```

関数作成の最後にabsと同じ働きをする関数「my_abs」を作ってみます。まず絶対値を得るにはどういう手順で実現するか、考えてみましょう。

1. 数字Xを受け取る
2. その数字Xが0より大きければ、すでにXは絶対値。Xが0より小さければ、-1をかけて絶対値にする。

3. 絶対値を関数の呼び出しあとに返す

この2番目のステップで、「Xが0より大きければ、Aをする。そうでなければBをする」といったように、処理が条件分岐しています。このような条件分岐もプログラムで実現できます。



上記の図を参考にして、関数my_absを書いて利用してみます。

```
def my_abs(x):
    if(x>0):
        return x
    else:
        return x * -1

print(my_abs(-6))
# 6
```

先ほどのprint_hello、add5よりも複雑になっていますが、最初の5行で関数の定義をして、後半で定義した関数を呼び出しているのは同じです。-6という数字をmy_abs関数に渡して、返された絶対値をprint関数で画面に出力しています。

関数my_absの中の動きを見てみましょう。my_abs(x)のxは、呼び出しあとで渡された値を格納しています。今回は-6になります。その次に「xが0以上ならAをする、そうでないならBをする」という処理がきますが、それが「if else」の文です。ifの後ろの()の中の条件「x>0」が満たされたなら、return x、満たされないならelseの後の処理であるreturn x * -1が実行されます。つまりx>0のときはxをそのまま返して、x>0でないときは-1をかけて負数を正数にしたうえで値を関数の呼び出しあとに返すということです。これで正数であろうと負数であろうと、常に絶対値が返されることがわかります。

ifのような「条件分岐」や、特定の処理を繰り返す「ループ」と呼ばれるものを組み合わせることで、プログラムの動きをより複雑にすることができ、「特定の機能を実現するためのルール」が実現されます。そのルールのことを「アルゴリズム」と呼びます。

一行一行命令を書くことと、誰かが作った処理を呼び出すことでプログラムが作られるということはわかっていただけたかと思います。ただ、簡単な計算処理をさせるだけではプログラミングで何を実現できるか、いまいちイメージがつかめていないかもしれません。そのため、かなり駆け足となってしまいますがGUIのアプリケーションを実際に作ってみることで「一歩先のレベルで何ができるようになるか」を実際に味わってもらいたいと思います。なおGUIはオブジェクト指向の知識を必要とするため本書ではなく中編で扱っています。

まず以下のプログラムをコピペで結構ですので実行してみてください。

```
import tkinter
font=('Helvetica', 32, 'bold')
label = tkinter.Label(text='Hello Python', font=font, bg='red')
label.pack()
label.mainloop()
```

プロンプトへの貼り付けはうまく貼り付けられないことがあるので、前回話したようにファイルに記入してから実行したほうがいいかもしれません。起動すると以下のようなGUIの画面がでてきます。



GUIを利用する手法はさまざまですが、上の例では「tkinter」と呼ばれるPythonが提供しているGUIのモジュールを使っています。mathモジュールと同じように、tkinterをimportし、「tkinter.Label」という関数でGUIのパーツを作り、それを変数labelに格納しています。その際にフォントや背景色、テキストに表示する文字などを指定しています。少し複雑ですが、基本的には関数absに-5を与えたときに得ていたのと同じです。そして、そのlabelに格納されたパーツに対して、「pack()やmainloop()」という処理をしろ」と命令することで、実際に画面にGUIのパーツが画面に表示されます。

実は、tkinter.Labelという関数はLabelという「クラス」を「インスタンス化」するという処理を実行しています。ただ、その概念は少々ややこしいので、詳細は「中編のオブジェクト指向の書籍」で扱います。ただ、「クラスから作られたデータに対して、命令をすることで処理が実行され、データの状態を変化させる」ということは覚えておいてもらったほうがよいかもしれません。これについては後ほど簡単に解説します。

本章の最後にGUIのカウンターのアプリケーションを書いてみます。アプリケーションとしては非常に単純なもので「ボタンがクリックされた回数」を表示するだけのものです。以下のプログラムを実行してみてください。

```
import tkinter
counter = 0
font=('Helvetica', 32, 'bold')
button = tkinter.Button(font=font, text=str(counter))

def clicked():
    global counter
    counter = counter + 1
    button.config(text=str(counter))

button.config(command=clicked)
button.pack()
button.mainloop()
```

以下のような画面が立ち上がり、クリックすると表示される数が増えることがわかると思います。



上記のプログラムを細かく解説することはしませんが、このプログラムで重要なのは以下の2つです。

- `def clicked():` による関数の宣言
- `button.config(command=clicked)` による関数の登録

ボタンをクリックした際の処理を自分で関数「`clicked`」として定義し、それを「`button`」というGUIのパートに「ボタンがクリックされたらこの関数を実施して下さい」という形で登録をします。 そうすることで簡単にGUIのプログラムをカスタマイズして作ることが可能になります。

複雑なアプリケーションを書く際は、すべてを自分でプログラムをするということは実質的に不可能です。そのため、提供されるなんらかのライブラリやフレームワークを利用することになります。今回の「`clicked`関数の宣言とその登録」は、ライブラリやフレームワークを利用する際によく使われる手法のひとつです。ほかにはクラスを継承する方法などもありますが、これらについても中編のオブジェクト指向編にて扱います。

型と変数

型

プログラミングで使われるデータには「型」と呼ばれるものがあります。たとえば、1や2というものは「数値」という型で、“Hello”というテキストは「文字列」という型です。プログラミングをする際に、初心者の方はこの「型」について意識することが重要です。なぜかというと、「型」と「処理」は密接に結びついているためです。例をあげて説明します。Python のプロンプトを立ち上げて以下を実行してみてください。

```
>>> 3 + 3  
6  
>>> 3 - 1  
2
```

上記のように数値は足し算、引き算することができます。当たり前といえば当たり前です。

では、文字列はどうでしょうか。文字列は「」でアルファベットや記号を囲むことで作成できますので、先ほどと同じように足し算と引き算をさせてみます。

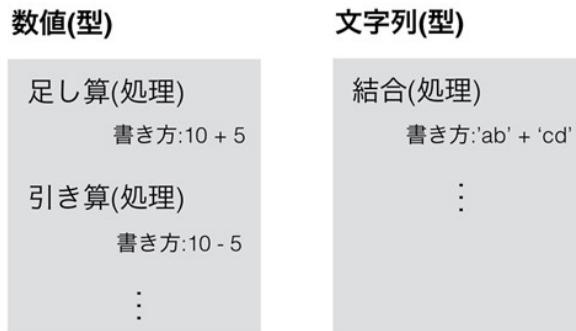
```
>>> 'hello' + 'python'  
'hellopthon'  
>>> 'hello' - 'python'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

文字列の足し算はできました。文字列の後ろに別の文字列をくっつけるという「結合」処理がされています。文字列の結合は数値の足し算とは違いますが、これも直感的な処理内容といえるのではないでしょうか。

ただ、文字列の引き算をしようとしたところ、エラーが表示されました。このエラーの内容は「operand(オペランド)」という言葉で小難しく説明されていますが、要するに「文字列 - 文字列」という処理はできないということを言っています。

最初にも説明しましたが、「型」と「処理」は密接に結びついています。数値であれば足し算も引き算もできますが、文字列は足し算(結合)しかできません。同様に文字列でしかできない処理というのも存在しています。すべてを暗記する必要はありませんが「どの型がどのような処理ができるか」「その処理をするにはどういう書き方をすればいいか」ということのある程度知っておくことは重要です。詳細はプログラミング言語ごとに若干異なりますが、他の言語でもそれほど大きな違いはありません。そのため、型と処理の関係の知識がたまると言語というよりプログラミング自体の知識が向上します。

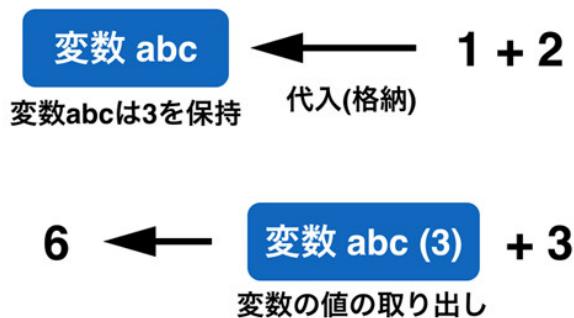
以下に型と処理の関係を示す図を記載します。



なお、ぱっと見て「同じに見えるデータ」であっても、型が違えば別物ですので注意が必要です。例えば数値の3と文字列の'3'は別物です。前者は引き算に使えますが、後者は使えません。後者はあくまでも文字列としての3であり、数値ではありません。

変数

型の続きを説明する前に変数の話をしません。変数はデータを格納するための箱のようなものです。プログラミングは1行1行に命令を書き連ねていくことで実現されますが、後ろの行で前の行の結果を利用する場面などがあります。そのような場合に、変数に命令の結果を保存して、それを後で使うといった利用方法がとられます。イメージとしては以下の図のようなものとなります。



変数の使い方は非常に簡単で、以下のように宣言するだけで変数に値が格納されます。

```
変数名 = 変数に入れたい値
```

変数名はアルファベットから始まり、特別なキーワードを避けた英数字と一部の記号("_"など)から構成されていれば、好きなものを使って

かまいません。特別なキーワードは次章以降で扱うPythonの文法で利用されるものです。たとえば、条件分岐のifやelseなどは変数名には使えません。なお、変数に値を入れることを「代入する」といいますので、それも覚えておいてください。

変数が利用される場合は、その中に実際に代入されている値が自動的に取り出されて使われます。以下に例を示します。

```
>>> a = 5
>>> print(a)
5
>>> b = a + 5
>>> print(b)
10
>>> b = a + 6
>>> print(b)
11
```

取り出しただけではデータは消失しないので、何度も利用できます。たとえば変数aに5を代入した後に、変数aから2回値を取り出していますが、2回とも5が取り出されています。一般的には、取り出すというよりも“変数aが5を‘返す’”というような言い方をします。

ただ、注意してほしいのは「なにか値が代入されている変数」に新しい値を代入してしまうと「昔の値」は上書きされてしまうということです。上記例で、bは10を保持していましたが、そこに11が代入されると10を消失してしまいます。また、代入されていない変数を使おうとするとエラーとなります。

少し高度になりますが、変数に同じ変数の値を加工して代入することも可能です。たとえば、変数aにすでに文字列が入っており、それに別の文字列を追加したいという場合は以下のように書きます。

```
>>> a = 'hello'
>>> a = a + 'python'
>>> print(a)
hellopython
```

上記の「a = a + 'python'」は以下の動きをします。

1. 右のaが'hello'という文字列を返す
2. それに'python'が結合されて'hellopython'になる
3. 'hellopython'が左のaに代入される

なお、「a = a」のように加工せずにそのまま代入することができますが、その処理にとくに意味はありませんので普通はしません。

上記のサンプルでは変数名に a というアルファベットを使いましたが、実際のプログラミングを行う際は以下のルールを守って命名するようになります。

- 分かりやすい変数名
- 大文字は使わずアンダーバーで単語を区切る
- 理由なくアンダーバーから始まる変数名は使わない(特別な意味を持つ)

以下に良い例と悪い例を示します。

```
# 1単語の場合
student = 'taro' # OK : 小文字のみで構成
Student = 'taro' # 大文字スタートなのでNG
a = 'taro' # a が何か分からないのでNG

# 2単語の場合
student_name = 'taro' # OK : 小文字の単語間を_で区切る
studentName = 'taro' # NG : 単語の区切りを大文字に
                      # するのは Python 流でない

# 先頭が_から始まる
_student = 'taro' # 他の人に「これを使わないで下さい」と示す
__student = 'taro' # 文法として外から使えないする
```

プログラミング言語によって変数名の名前の付け方は異なります。難しいことを考えなければ小文字の単語とアンダーバーのみで構成する覚えておいて下さい。なお、アンダーバーから始まる変数をどう利用するかは中編にて扱います。

また使いたい名前が予約後(Pythonの文法として特別な意味をもつワード)や、既に定義されている名前と重複する場合は名前の後ろにアンダーバーをつけるという慣習があります。たとえば後ほど説明する list という型がありますが、リストのデータを list という変数に代入することは名前が分かりやすいためついやりたくなりがちですが避けなければいけません。list という変数名の代わりに list_ という変数名を使います。

```
# NG
# list = [1,2,3,4]

# OK
list_ = [1,2,3,4]
print(list_)
# [1, 2, 3, 4]
```

ただ、本書で使われるようなサンプルのプログラムならともかく、現実のアプリケーションで用いられるコードの変数は何らかの役割があるはずです。そのため、list_ といったような名前は本来現れないことが望ましいです。

Pythonの変数の概念は非常に簡単です。「変数はどのような型の値でも格納する入れ物のようなもの」ということを理解していれば、しばらくは何も問題ありません。ただ、CやJavaなどのほかの言語だと、変数の概念はもう少し複雑なので簡単に説明します。興味がない人は読み飛ばしていただいてかまいません。

型と変数

Python以外の言語を少しでも学んだことのある人は、先ほどのPythonの変数の使い方に違和感を覚えられたかもしれません。CやJavaなどは「変数」と「型」が密接に結びついています。具体的にいうと変数にも型があり、変数の型と代入する値の型は同一である必要があります。

たとえば、Javaで変数xを宣言し、それを利用するには以下のように書きます。

// JAVA

```
int x;  
x = 5;      // OK  
x = "Java" // Error
```

Pythonで変数を利用する場合、変数xの前に「int」というキーワードは存在していませんでした。intは整数型のことなので、この場合は「整数型の変数x」を作っています。Javaの変数には必ず型があるので、変数を宣言する(作る)際には必ず「int」といった型を示す必要があります。

上記例では変数を作成した後で、その変数に整数値5と文字列"Java"を代入しています。変数xは整数型なので、同じ整数型である5は代入できるものの、文字列型の"Java"は代入できずにエラーとなります。

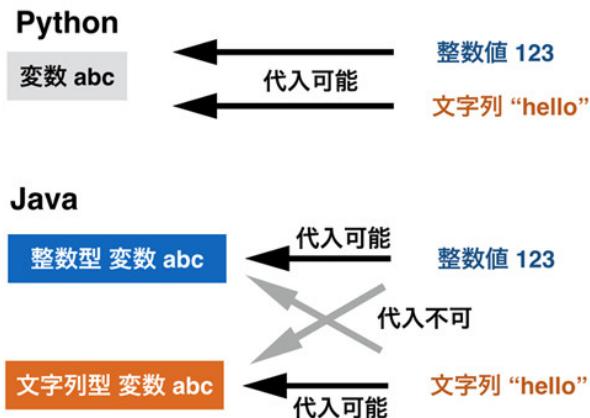
文字列型の変数もこれと同じです。文字列を格納するために作成された変数yに文字列を代入することは当然できますが、整数型である5を代入することはできません。

```
// JAVA  
String y = "Java"; // OK  
y = 5; // Error
```

一方、Pythonの変数にはどのような型でも入れることができます。以下のように「整数を代入した変数に文字列を代入する」ことも問題ありません。そもそもJavaでいう「int」や「String」といった変数の型宣言がPythonのコードにはありませんね。

```
# PYTHON  
x = 5  
x = 'python'
```

PythonとJavaの変数の使い方の違いを図にまとめます。



ただ、上記のような「変数を使いまわす」ようなコードは一般的には避けたほうがよいです。一度宣言された変数を「別の使い方で再利用」すると、その変数に何が入っているのかわかりにくくなるため、プログラミングの行儀がよくありません。専門的な言い方をすると「保守しにくいコード」といわれます。脱線はこのぐらいにして本題に戻ることにしましょう。

定数とマジックナンバー

プログラムの中にはたまに「マジックナンバー」と呼ばれる正体不明の数字が現れことがあります。たとえば以下のコードを見て下さい。

```
a = 100 * 1.08  
print(a)
```

このコードが何をやっているか分かりません。結論を言ってしまうと、1.08は税率にあたり、税抜き100円が税込みいくらになるか確認しています。1.08という数字を見て勘のいいかたであれば「税率のようだ」と気づくかもしれません、プログラミングはそのような気づきには頼ってほしくないです。

このような場合、1.08というよくわからない「マジックナンバー」ではなく「定数」を使うことが望ましいです。定数は一回設定されたら変更されることがない特殊な変数であり、その名前は大文字とアンダーバーで宣言されます。例えば先程の税込み額の計算プログラムであれば以下のようになります。

```
TAX_RATE = 1.08  
a = 100 * TAX_RATE  
print(a)  
# 108
```

上記の TAX_RATE が定数にあたります。

この定数は一般的にプログラムのファイルの先頭で宣言されます。そしてパラメータとして使われることも多く、ファイルを開いて定数の値を変更することでプログラムの挙動を変えることがあります。例えば税率が10%に変更されたら、プログラム本体ではなく定数を書き換えることで対応します。

```
TAX_RATE = 1.1  
a = 100 * TAX_RATE  
print(a)  
# 110
```

1.08という税率がプログラムの各所に散っていたら、それを全て1.1に書き換えていく作業が必要になります。定数を使っていればそのような作業は不要になり、なおかつプログラムが分かりやすくなります。

ちなみに、Pythonの定数は厳密な意味では定数ではなく、実は文法としては単なる変数と変わりがありません。大文字で定数を作るというのはPythonのプログラマたちが決めたルールであり、文法的な拘束はありません。そのため再代入をしてもエラーとはなりません。ただ、

定数が変わることは誰も想定していないので、再代入は絶対にしないでください。

Python で重要な型

型と変数の基本的な使い方がわかったので、最初に知るべき重要な型とその利用法をいくつか紹介したいと思います。今回紹介するのは以下の4つとなります。

- 数値
- 文字列
- Bool(ブール)
- リスト(配列)

この4つの型を使わずにプログラムを書くことは不可能と言ってもよいほどです。これ以外にも重要な型はいくつかありますが、まずはこれら4つの型をしっかりと使いこなせるようになることが大事です。

数値型

いくつかのプログラミング言語では、同じ数値といってもそれが種類ごとに細かく分類されて別の型として扱われます。たとえば、CやJavaでは「整数」と「小数」は別物ですし、それらも表現できる範囲が決まっています。Javaの整数型であるintは32bitで整数を表現する型であるため、小数点は扱えない(切り捨て)ですし、32bitで表現できない非常に大きな数なども利用できません。

一方、Pythonで数値を使うのは非常に簡単です。正確にはPythonにも整数型や小数型は存在するものの、それらは区別なく同じ「数値型」のようなイメージで扱うことができます。また、Javaのintで表現できない非常に大きな桁も、Pythonでは特別な操作をせずに表現できます。

```
>>> 123456789 * 123456789 # 掛け算
15241578750190521
```

数値型でどのような処理ができるかという話に移りましょう。とくに断りなく使ってきましたが、数と数の計算に使用する「+」や「-」といった記号は「演算子」とよばれています。そして演算子の演算対象となる値を「オペランド」と呼びます。

```
>>> 1 + 2
3
>>> 5 - 3
2
```

たとえば、 $1 + 2$ の演算子は「+」であり、そのオペランドは「1」と「2」です。数値型に関しては、この演算子の種類を知ることが「処理」を知ることの第一歩といえます。

Pythonの数値計算で利用可能な演算子は以下となります。いくつかは算数で使われる記号なのでわかりやすいですが、プログラミング独自の記号の使い方や、Pythonだけしか使えない記号もあります。足し算、引き算、掛け算、割り算がメインとなる処理ですが、ほかの演算もときどき使うので覚えてしまってもいいかもしれません。

| 利用可能な演算子 | 説明 |:-----:| M + N | 足し算 || M - N | 引き算 || M * N | 掛け算 || M / N | 割り算 || M % N | 剰余(あまり) || M ** N | べき乗(M * M * M.. を N回) |

算数の授業で習ったかと思いますが、演算子にも優先順位があります。たとえば算数で「 $1 + 2 \times 3$ 」という計算をする場合、足し算よりも掛け算が優先されるため、 $1+2$ よりも先に 2×3 が計算されて、答えは7になります。Pythonでも同様に、上記の計算結果は7となります。掛け算よりも足し算を優先する場合は「足し算を()で囲む」ことをしますが、Pythonも同様です。

```
>>> 1 + 2 * 3
7
>>> (1 + 2) * 3
9
```

とりあえず数値型の処理の紹介はこれで終わりです。ただ、当然ながらほかにも多くの処理が存在しています。たとえば今まで利用していた「絶対値を得る方法」や、「文字列の数字を数値型に変換する方法」などもあります。よく行われる処理は調べればすぐわかるので、その都度ドキュメントをあたるなり、検索エンジンを使うなりして解決してください。

演算子の話をしたので、次に代入を行うための特別な演算子である「代入演算子」の紹介もします。名前からわかると思いますが、代入(=)と演算を同時に使うのが代入演算子です。

| 利用可能な演算子 | 説明 | M += N | M = M + N || M -= N | M = M - N || M *= N | M = M * N || M /= N | M = M / N || M %= N | M = M % N || M = N | M = M N |

演算子と代入の記号(=)がくっついているだけなので、規則性は見てとれます。

```
>>> a = 5
>>> a += 3
>>> print(a)
8
```

注意すべきなのは、PythonにはCやJavaでいうインクリメント/デクリメントが存在しないことです。インクリメントは変数の値に1を加えることで、そのためには特別な演算子である「++」を使います。

たとえばJavaの以下のコードでは、iは1になります。

```
int i=0;
i++;
```

Pythonで同様のことを行うには、以下のように書きます。

```
i = 0
i += 1
```

変数iに1を加えた値を、再度iに代入することは、インクリメントすることと実質的に同じです。デクリメントも同じように使えます。

必要であれば分数も利用されます。例えば「整数÷整数」も必要があれば分数になります。

```
>>> 5 / 3
```

```
1.6666666666666667
>>> 5 // 3 # // は整数になる除算(切り捨て)
1
```

数値を扱う関数も多数用意されており、例えば前章にて扱った絶対値を得る関数やリスト(後述)の合計値を得る関数があります。

```
>>> abs(-3)
3
>>> sum([1,3,5])
9
```

数値の操作はさまざまな処理で利用されます。たとえば解析ツールや統計処理を行うアプリケーションを作るのであれば、数学的な処理をする必要があります。演算子と数学的知識を使って、それを自力で実装する方法もありますが、可能であれば実装の労力と実効速度およびバグの少なさを考慮して「標準ライブラリ」や「外部のライブラリ」を使うべきです。ライブラリ(モジュール)の使いかたは本書で扱いますが、個別ライブラリに関しては主に下編で扱います。

文字列型

プログラミングでは、テキストデータを扱うことが非常に多いです。そのため、テキストデータを扱う「文字列型」を使いこなせるようになります。

数字はそのまま書けば認識されましたが、文字列は「特別な記号」でテキストを囲むことではじめて、Pythonで解釈できるようになります。今まで特にことわりなくシングルクオテーション「'」を使っていましたが、それも文字列を作る特別な記号のうちのひとつです。

たとえば以下のようにつかいます。

```
text = 'abcdefg'
```

テキストも数値と同じように演算することができます。先に示したように「+」で結合もできますし、あまり知られていませんが「*」で同じ文字列を繰り返すこともできます。

```
>>> text = 'hello' + ' python'
>>> print(text)
hello python
>>> text = 'hello' * 3
>>> print(text)
hellohellohello
```

プログラム中で文字列を作成する方法は主に3つあります。

Pythonで最もスタンダードなのは文字列型にしたいテキストをシングルクオテーション「'」で囲むというものです。空白文字もそのまま含めることができます。

```
>>> text = 'hello python'
>>> print(text)
hello python
```

タブや改行といった特殊な文字はエスケープ処理をすることで加えることができます。たとえば改行は、"\n"と表現されます。英語キーボードの半角の(バックスラッシュ)と日本語キーボードの¥は同じ意味なので、¥n(円記号は半角)も改行の意味を持ちます。

```
>>> print('hello \n python')
hello
python
```

わかりやすいように\nの前後に空白をいれましたが、改行させたいだけの場合は空白は不要です。'hello\npython'と書けば改行コードが入ります。

文字列を作る別の方法はテキストをダブルクオテーション「"」で囲むというものです。シングルクオテーションとほぼ同じです。ほかのプログラミング言語だとダブルクオテーションのほうがよく使われる所以、Pythonでもこちらを好んで使う人がいます。

シングルクオテーションとダブルクオテーションの使い分けは特に決まったものはないのですが、「シングルクオテーションをダブルクオテーションで囲むと文字として扱われる」というルールがあるので、文字列の中にシングルクオテーションを使いたい場合はダブルクオテーションを使うと便利です。

```
>>> print("it's nice!!")
it's nice!
>>> print('it\'s nice!!')
it's nice!!
```

2つめの例のようにエスケープ記号を使うことで、シングルクオテーションの中でシングルクオテーションを使うこともできます。ダブルクオテーションでシングルクオテーションを囲むのと同じように、シングルクオテーションでダブルクオテーションを囲むこともできます。その効果はまったく同じで、ダブルクオテーションをエスケープせずに文字列で使うことができるというものです。

最後の方法はトリプルクオテーションというものです。これはテキストをシングルクオテーションかダブルクオテーションの3つで囲むというものです。トリプルクオテーションで囲まれると、その中身が見たままにテキストとして表示されます。例えば以下のようにになります。

```
>>> text = '''it's nice!!!
>>> print(text)
it's nice!!
```

プロンプトでは使えませんが、ファイルにプログラムを書き込む場合は、改行も含めてひとつのテキストにすることが可能です。たとえば以下をPythonのプログラムファイルに書きます。

```
text = '''hello
python'''
print(text)
```

そしてそれを実行すると、以下のような出力がされます。

```
hello
python
```

一つのprint文で複数行の文字列が表示されていることが分かります。

このトリプルクオテーションは文字列の宣言としての利用よりも、複数行のプログラムをコメントアウトするときに使われることが多いかもしれません。たとえば以下のような例があげられます。

```
処理1
"""
処理2-1 # 古いコード。実行されない
処理3-1 # 古いコード。実行されない
"""
処理2-2 # 新しいコード。実行される
処理3-2 # 新しいコード。実行される
処理4
```

処理2,3の動きを変更したいと思ったので、昔の処理「2-1,3-1」を文字列にしてしまうことで実行されないようにして、新しく「処理2-2,3-2」を書いたものです。改良やバグ探しの場面では「昔の処理は実行させたくないのだけれども、消したくはない」ということが多く発生します。そのようなときにトリプルクオテーションで処理を文字列にしてしまい、処理を無視させてしまうと便利です。また、後ほど説明する関数やクラスにて「それが何をやっているか」ということをトリプルクオテーションで表明することもよくあります。なお、CやJavaでいうところの範囲指定のコメントアウト「`/** ハイフンアロー ハイフンアロー */`」はPythonでは使えない注意してください。

文字列の操作については後ほど詳細を扱いますので、ここでは簡単な説明にとどめておきます。演算子の利用は先に説明したとおりです。

```
>>> 'hello' + 'python'
'helloworld'
>>> a = 'hello'
>>> a += 'python'
>>> print(a)
'helloworld'
```

これに加えて関数を使う方法もあります。たとえば数値などの「文字列でない型」を文字列型にするには`str`関数を使います。この関数で囲ったすべての型は文字列に変換されます。

```
>>> str(5)
'5'
>>> str(5.5)
'5.5'
```

この関数は文字列に「文字列以外の型」を結合する際によく使われます。この変換をしないとエラーになる場合があります。たとえば、以下の1行目の「文字列 + 数値」はエラーとなりますが、2行目は「文字列 + 文字列」なので問題ありません。

```
>>> 'hello' + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> 'hello' + str(5)
'hello5'
```

Bool型

Boolは別名「真偽値」とも呼ばれる型です。真偽値という名前を聞くとなんだか難しそうに思えるかもしれません、要するにYES/NOに相当する「True/False」という「正か非」の2値しかない単純な型です。

Pythonのプログラム中で "True" と "False" は特別なキーワードとして扱われます。それぞれその言葉のとおりに、YES/NOとしてPythonに解釈されるのです。このようなキーワードのことを「予約語」と呼び、予約語は変数名や関数名に使うことはできません。

Boolは「比較演算子」と呼ばれる記号で2つの値を比較した際に返されます。例として数字の大小を比較してみます。

```
>>> 10 > 5
True
>>> 10 < 5
False
>>> a = 10 > 5
>>> a
True
```

特に難しいことはありませんね。「10は5より大きい -> True(YES)」とされていますし、「10は5より小さい -> False(NO)」とされています。Boolを変数に格納することも当然できます。

比較演算子の一覧を以下に記載します。

| 利用可能な演算子 | 説明 | A == B | AとBが同一ならTrue || A != B | AとBが異なればTrue || A > B | A が B より大きければ True || A >= B | A が B 以上なら True || A < B | A が B より小さければ True || A <= B | A が B 以下なら True |

以下に例を書いてみます。

```
>>> 'hello' == 'world'
False
>>> 'hello' != 'world'
True
>>> 5 > 4
True
>>> 5 > 5
False
>>> 5 >= 5
True
```

それほど難しくありませんね。以下のようなブール代数(True/False)も使えます。これは中学や高校の数学あたりで学んだものかと思います。

| 利用可能な演算子 | 説明 | not A | AがFalseならTrue || A and B | AもBもTrueならTrue || A or B | AかBがTrue |

```
>>> not True
False
>>> True and False
False
>>> True or False
True
>>> not (5>4)
False
```

他には集合やオブジェクトのチェックの演算子もあります。これらは追って説明します。

| A is B | A と B は同一オブジェクト || A is not B | A と B は異なるオブジェクト | | A in B | A は B に含まれる || A not in B | A は B に含まれない |

また、返り値がBool値である関数などもよく利用されます。

Bool型は後の章で扱う「ifやfor」といった制御文で利用されることが多いです。条件分岐のif文では、たとえば変数aがTrueなら処理Aを実行し、Falseなら処理Bをするといった使い方をします。具体的にBoolをどのように使うかは制御文を学ぶ際に理解できると思いますので、今回はここで解説を切り上げます。

リスト

リスト(List)が本章で紹介する最後の型です。名前からわかるようにデータを「リスト」状に複数個並べたような型です。今までの数値や文字列に比べると使いどころがよくわからないかもしれませんので、まずは例を示します。

たとえば生徒のテストの点数を管理するアプリケーションを書くとします。リストを使わずに3人の生徒の平均点を求めようとすると、以下のようなコードが書けます。

```
student1 = 68
student2 = 81
student3 = 49
average = (student1 + student2 + student3) / 3
print(average)
# 66
```

生徒ごとに変数を作って、そこから平均値を求めていました。それほど難しくはありませんね。ただ、上記のプログラムには問題があります。たとえば生徒の数が4人になった場合などに修正する箇所が多くなってしまうことです。生徒が40人だと変更はもっと大変です。

このような問題は、リストを使うことでかなり解消できます。リストは「リストというデータの中に複数のデータを格納できる」という型ですので、「生徒達の点数」というデータに「具体的な各生徒の点数」を格納します。リストを使うと先のプログラムは以下のようになります。

```
results = [68, 81, 49]
average = sum(results) / len(results)
print(average)
# 66
```

1行目では「生徒たちの点数」というリストを作成しています。見ればわかると思いますが、リストは[]の記号のなかにコンマ区切りでデータを羅列することで作成されます。2行目にあるsum()はリスト内にあるデータの合計値を算出する関数で、len()はリストに格納されるデータの数を返す関数です。ここでは平均値を求めるために「生徒の点数の合計値/生徒の人数」としています。

2つのコードは生徒一人ひとりの点数ごとに変数を作成していないので、「複数の生徒たちの点数の格納」も簡単ですし、なにより平均値の算出方法が生徒の数に依存していません。このようにリストを使うことで「ひとつのグループに属するデータ」を便利に扱うことができます。

リストの概念を以下に記します。



リストに含まれるデータは「要素(Element)」と呼ばれます。たとえば上記図でいうと1番目の要素は68であり、2番目の要素は81となります。

リストがどのようなものかわかっていただけたかと思いますので、操作方法について例を交えながら説明していきます。リストの作成は以下のように[]記号でリストの要素を囲むことで実現できます。

```
>>> a = []
>>> b = [1, 2, 3]
>>> c = ['A', 'B', 'C']
```

[]の中に何も入れない場合は空のリストを作成します。なかに数字や文字列、及びその他の型をコンマで区切って書くことにより、それらを要素とするリストを作成することができます。

次にリストの「要素」を取り出したり、書き換えたりする方法を示します。先の b=[1,2,3] では要素数が3つあるリストを作成していて、その中身は1、2、3となっています。この中身にアクセスするにはリスト内の「x番目の要素を指定」する必要があります。その方法は、「リスト名[要素の番号]」となります。

ただ、気をつけなければいけないのは、指定する順序は1からではなく0からということです。たとえば、以下のように使います。

```
b = [1,2,3]
print(b[0]) # -> 1
print(b[2]) # -> 3
b[1] = 5
print(b[1]) # -> 5
```

ちなみに、この「x番目」ということを「インデックス番号」と呼びます。b[2]は、「リストbのインデックス番号が2の要素」という意味になります。リストの値を参照するだけでなく、上書きもできています。

また、先ほど紹介したリスト長の取得関数 len() もよく使います。

```
>>> b = [1,2,3]
>>> b[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> len(b)
3
```

上記コードを見てもらうと分かりますが、リスト範囲外へのアクセスはエラーになっています。リスト長を超えたアクセスをしないためにもlen()でリスト長を得て、要素へのアクセスは「リスト長 - 1」番目までとする必要があります。

他にはリストに要素が含まれるかの確認もよく利用します。

```
>>> a = [5,7,3,9]
>>> 3 in a
True
>>> 8 in a
False
>>> 8 not in a
True
```

in という予約後を使って「要素 in リスト」とすることで、リストに要素が含まれていれば True、含まれていなければ False が返ってきます。

ちなみに Python のリストは「整数型だけ」「文字列型だけ」というように決まった型のデータだけでなく、なんでも格納することができます。ただ、管理しにくいのでこのような使い方はあまりしません。

```
>>> a = [1,'hello',False]
>>> a
[1, 'hello', False]
```

また、リストのなかにリストをいれて行列を作ることもよくあります。

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

a[1][1] は a というリストから1番目の要素のリストをとりだし、その取り出したリストから1番目の要素を取り出すということです。上記でいうと、まず [4,5,6] が取り出され、次に 5 といった具合です。行列を図で示すと以下のようになります。



リストに使える関数も色々ありますが、現時点で知つておいて欲しいのは先ほど紹介したリスト長を得る len() と、連番のリストをつくる range() です。これらは今後のサンプルコードでも利用されます。

```
>>> len([1,2,3])
3
>>> len([3,5,7,9,11])
5

# python2 より
>>> range(5) # 0 から 5 - 1 まで
[0, 1, 2, 3, 4]
>>> range(5,10) # 5 から 10 - 1 まで
[5, 6, 7, 8, 9]
```

なお、python3 では range() 関数は range 型という型を返すようになっていますが、上記の python2 のサンプルとほぼおなじ感覚で利用することができます。リストは奥が深いので、もう少しプログラミングの基礎を学習したあとで一章を割いて詳細を扱います。

型の確認 (type)

変数にどの型のデータが入っているか分からぬ場合はそれを type 関数に与えるとすぐに分かります。

```
>>> a = 'hello'
>>> type(a)
<class 'str'>
```

たとえば上記は変数 `a` に文字列型が入っていることがすぐに分かります。これは変数以外でも例えば関数の返り値の型を調べるといった目的でも便利です。C や Java といった言語であれば常に型を気にしてコードを書く必要があるため問題になりませんが、Python などの型を使っているか分からなくなることがあるため、その際は `type` 関数で調べて下さい。

型の変換(キャスト)

ある型を別の型に変換することを「キャストする」といいます。たとえば数字から文字列型、文字列型から数字への変換は以下のように行います。

```
>>> str(123)
'123'
>>> int('321')
321
>>> float('321.123')
321.123
```

まだ学んだ型が多くないため多くは語りませんが、基本的にキャストはコアとなる型(本章で学んだものや後ほど学ぶ `set` や `dict` など)の間で使うことがほとんどです。主要なものは都度覚えていくことが望ましいですが、必要以上に多くを覚える必要はありません。

制御構造

プログラムの基本的な流れは上から下へ一行ずつ実行していくというものです。単純なプログラムですと、テキストファイルに実施する処理を順番に羅列するだけで実現できます。いわゆる「バッチ処理」と呼ばれている処理方式です。

ただ、複雑なプログラムだと、このような「上から下へ順番に実行していく」というスタイルだけでは処理を実現できなくなってしまいます。たとえば、天気予報を確認するアプリケーションでは、「今日が晴れなら晴れマークを表示、雨なら雨マークを表示」といった具合に「あるものがAならBをする。そうでないならCをする」という複雑な処理が必要になってきます。「条件」に応じて処理が「分岐」しているので、こういった処理のことを「条件分岐」といいます。

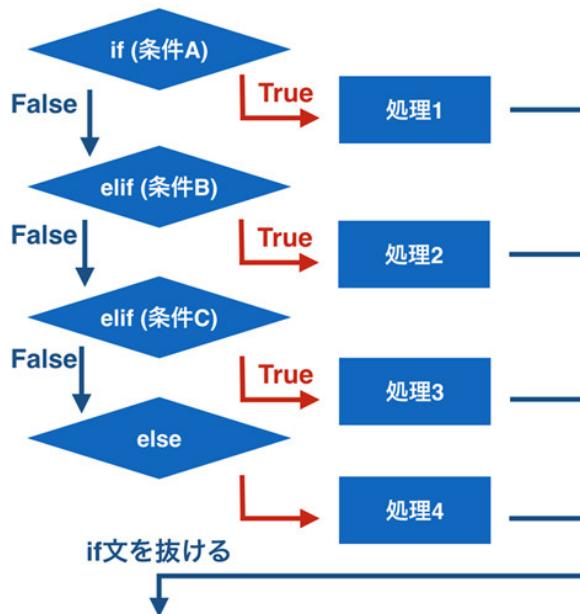
ほかには、同じ処理を何度も繰り返す「ループ処理」があります。たとえば、「クラス全員のテストの平均点を求め、その平均点と各生徒の点数の差分をチェックする」といった場合を考えてみましょう。平均を求めるには「生徒の点数の合計/人数」とする必要がありますが、この合計を求めるために「先頭の生徒から最後の生徒まで順番に点数を足していく」という「繰り返し(ループ)」が必要となります。平均との差分の算出も同様です。

本章ではこのような条件分岐やループ処理といった「プログラムの制御構造」について取り扱います。これらの処理を使うこと自体はそれほど難しくないので、何度も書いて慣れてしまえば、簡単に使いこなせるようになります。

if による条件分岐

条件分岐は、条件分岐の式を満たすか満たさないかで実行される処理が変わるという制御構造です。前章にて扱った Bool型が条件判定に利用され、その値がTrueかFalseかで実行するプログラムの挙動が変わります。

以下に条件分岐の仕組みを図で記します。



上記の図のうち、「if」は必ず必要です。そして「elif」は任意の数(0も含む)繰り返すことができ、「else」も省略することができます。if や elif、else にはそれぞれ決まった処理が書かれます。条件が合致した処理のみが実行され、他の条件に結びついた処理は一切実行されません。elseがないときは、どの条件にも合致しない場合は「何もしない」という動きになります。

この図をPythonのプログラムで書くと以下のようになります。

```
if(条件A):  
    処理1 # 条件 A が True の時に実行される処理  
elif(条件B):  
    処理2 # 条件 A が False で条件 B が True の時に実行される処理  
elif(条件C):  
    処理3 # 条件 A,B が False で条件 C が True の時に実行される処理  
else:  
    処理4 # 全ての条件が False の時に実行される処理
```

上記のifからelseの次の行までがひとつの「if文のカバー範囲」であり、そのなかにあるifやelif、elseが細かい処理の単位だと思っていただければ大丈夫です。条件ごとの処理はインデント(字下げ)されていることが分かります。

実際に条件分岐を行うプログラムを書くことで、条件分岐の使い方をイメージしてみましょう。プログラムは非常に簡単で、変数xの値が0より大きければ「+」と出力し、ピッタリ0なら「0」、0未満なら「-」と出力するというものです。これは以下のようになります。

```
x = 5  
if(x > 0):  
    print('+')  
elif(x == 0):  
    print('0')  
else:  
    print('-')
```

上記プログラムをIDLEのエディタに書いて実行してみてください。xに5が代入されているので、「+」と出力されたはずです。これはx > 0の条件式が満たされ(Trueとなり)、「print('+')」が実行されたからです。そして if else の条件文をぬけるため、print('0') や print('-') は実行されません。このxに代入する値をいろいろ変えて動かしてみると、どの条件式がチェックされ、「if、elif、else」のどの処理が実行されたのかイメージできるはずです。

インデントとコードブロック

条件分岐の話が終わったので、インデント(字下げ)についてもう少し詳しくお話ししましょう。先ほどのプログラムでif文に書かれた処理はかなりシンプルなものでしたが、実際にはif文は多くの処理を持ちます。すると、if文が「どこからどこまでをカバーしているか」をどのような形で表現するかが問題になってきます。このカバー範囲のことをコードブロックと読んでおり、インデント(字下げ)を揃えることで範囲を決めています。Pythonのインデントの仕方は「半角空白を4つ」が標準で、それに次いで半角空白2つも一般的です。タブの使用は推奨されていません。本書は紙面の都合上、半角空白2つとしています。

以下のコードを見て下さい。

```
処理1
if(条件A):
    # ここから
    処理2
    処理3
    # ここまでがコードブロック
else:
    # ここから
    処理4
    # ここまでがコードブロック
    処理5
```

上記のプログラムでは処理1、2、3、4、5があります。このうち処理1はif文の範囲外なので常に実行されます。そして処理2,3はif文のコードブロックに属するので、条件Aが満たされる場合に両方とも実行されます。処理2,3のインデントは同じ深さなので、同じコードブロックです。処理4はelseの場合のみ実行され、処理5は常に実行されます。字下げをすることでコードブロックを表現する。簡単です。

なお、CやJavaにもコードブロックはありますが、その書き方は異なっています。たとえばJavaだと上記のサンプルコードは以下のようなものとなります。

```
処理1
if(条件A){
    // ここから
    処理2 // 字下げは必須ではない
    処理3
    // ここまでがコードブロック
} else{
    // ここから
    処理4
    // ここまでがコードブロック
}
    処理5
```

{ }で囲むことでコードブロックを表しています。たいていは読みやすいように上記のようにインデントをしますが、プログラムとしてはインデントをする必要性はありません。

コードブロックの中にコードブロックを作ることも可能ですが、たとえば条件分岐の中に、さらに条件分岐を作ったりすることもできます。書き方は簡単で、コードブロックの内側にさらにコードブロックを作るというものです。その際、内側のコードブロックは外側のコードブロックに属しています。

サンプルコードをあげてみます。

```
if(条件A):
    処理1 # "if(条件1)" のコードブロックに属する
    if(条件B):
        処理2 # "if(条件1)" と "if(条件2)" の両方法のコードブロックに属する
        処理3
    else:
        処理4
```

処理1、2、3はすべて「if(条件1)」のコードブロックに属していますが、処理2だけではそれに加えて「if(条件B):」にも属しています。そのため、処理2が実行されるのは条件A、Bが共にTrueのときのみです。たとえ条件BがTrueであっても、条件AがFalseなら処理2は実行されません。コードブロックはifやループなどの制御構造だけではなく、関数やクラスでも利用されます。

なお、コードブロックに限らず、プログラミングで「入れ子」構造にすることを一般的に「ネストする」と言いますので覚えておいてください。ネストすること自体には問題はありませんが、その深さが増えてくるとプログラムが非常に読みにくくなります。深いレベルのネストが必要な状況になってきたら、アルゴリズムそのものを見直すか、後の章で扱う「関数」に処理を分割することで読みやすくすることが多いです。

なお、コードブロックの中には最低一つの式を書く必要があります。そのため、if文や関数を「とりあえず作って詳細は後で実装する」という場合に「なにもしない」ということをブロック内に書く必要があります。それにはpassキーワードを使います。

```
a = 'hello'
if(a == 'hello'):
    # 将来ここに処理を実装
    pass
else:
    # 将来ここに処理を実装
    pass

print(a)
```

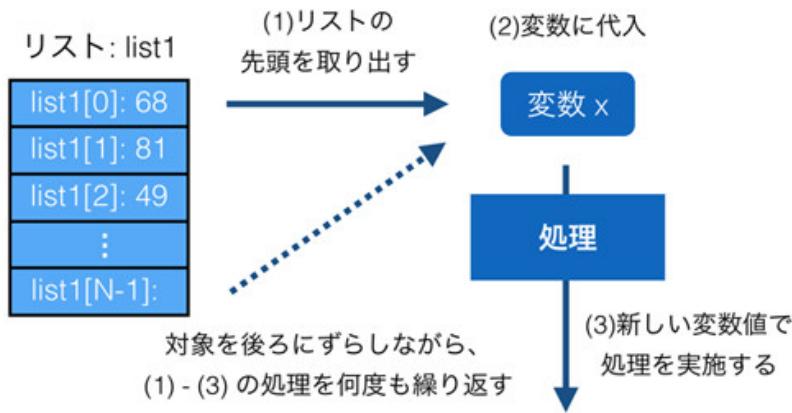
上記のpassはなにもしないためにこのif文は不要です。ただ、コメントにもあるように「とりあえずプログラムの構造を作っておきたい」という場合にpass文を利用することはよくあります。passを書かかずにブロックに何もないとプログラムはエラーとなります。

forによるループ

ループはその名前からわかるように「同じ処理を何度も繰り返す」という処理です。ループ処理の制御構造にはforとwhileの2つがあり、両者の使うべきポイントは若干異なっています。そのため、まずは利用頻度の多いforについて扱い、その後でwhileについて説明します。

forは「グループにある要素すべてを処理する」といったときに使われるループ構造です。一番よく使われるのが、前回お話したリスト(配列)に格納されている要素すべてをチェックするような処理です。JavaやCで使われるfor文と書き方はかなり異なるものの、ほとんど同じような場面で使います。

Pythonのfor文のイメージを以下の図に書きます。



難しい用語でいうと「イテレーター」と呼ばれる処理方式なのですが、ようするに「たくさんある集合の先頭ひとつを取り出して、それを処理する。それが終わったら、次を取り出して処理をする」ということを、集合が空になるまで繰り返すというイメージです。

それほど難しくないので例で示しましょう。1、2、3、4、5という数字が格納されているリストの中身を、一つひとつすべてprint出力する処理をforで書くと以下のようにになります。

```
a = [1,2,3,4,5]
for i in a:
    print(i)
```

このfor文は以下のような動きをします。

1. リスト a から 1 を取り出して i に格納。それをprint出力
2. リスト a から 2 を取り出して i に格納。それをprint出力
3. ... (中略) ...
4. リスト a から 5 を取り出して i に格納。それをprint出力
5. リスト a からすべてを取り出したのでforのコードブロックを終了

すでに想像はついているかもしれません、出力は以下のようになります。

```
1
2
3
4
5
```

イテレーターを使っているので、Javaのfor文で使うような「インデックス(配列の何番目か)による制御」に比べて、間違った要素を指定するリスクが減っています。一方、インデックスを操作するような複雑な処理は python の for では実現できません。そのような場合は後述する while を使うことになります。

なお、for文はリスト以外の「シーケンス型」と呼ばれる「並びを持つ型」にも使えます。例えば今までに扱った文字列もシーケンス型なので for 文が使えます。

```
text = 'hello'
for c in text:
    print(c)

# h
# e
# l
# l
# o
```

ただ、for文は十中八九はリスト及びそれに似た型に対して利用することになると思います。それ以外の型で使うことはあまり多くありません。for文を書く際に注意すべきなのは取り出した値を格納する変数の名前付けです。プログラムとしては変数名として有効なものであれば何を書いても動くのですが、一般的には以下のようないルールがあります。

- リストなどのシーケンスの変数名に対応する単数形の名前
- 順番に並んだ数字の場合は i, j, k (i が既に使われていれば j といった具合)
- リストや文字列から取り出す「一文字」は c (character の c)

例えばリストとして students という複数形があればそれを取り出す変数は student といった具合です。自分でプログラムを開発する場合はこのような名前付けになるように注意を払って下さい。サンプルプログラムなどで数字の配列に対して for 文を回す場合は i という変数名を使うことが多いですが、これは index という単語から来ています。i が既に使われている場合は j, k と使っていくのですが、これは単に jk というアルファベットとしての並びがあるからです。l (小文字の L) は数字の 1 や大文字の I (I) と見分けが使いにくいので使用は避けることが望ましいとされています。変数名としてよい名前が浮かばないときに i が使われることが多いのですが、あまりよい使い方ではありません。

whileによるループ

whileもforと同じくループ処理のための制御構造です。ただ、whileは「ループを何周すればいいかわからない処理」に利用されます。

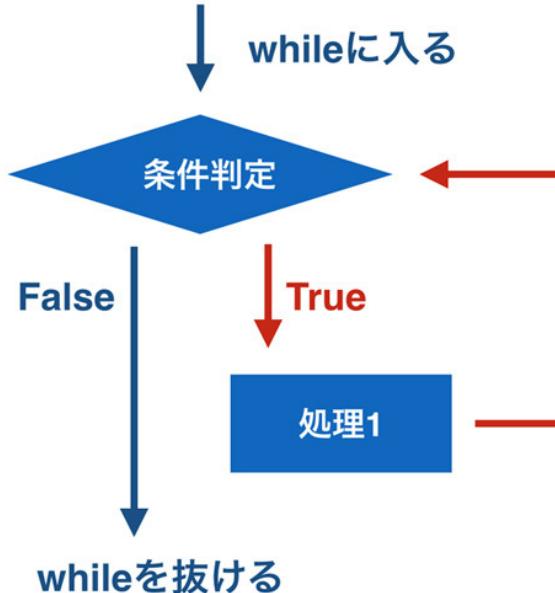
先ほどのforの例を思い出して下さい。forでのループ回数は「リストaに格納されている要素の数」と明確にわかります。このような場合はforで処理すべきです。一方、たとえば「123456789という数字を2進数で表現するのに必要な桁数を求める処理」が必要だとした場合、これをどうfor文で処理すればいいか、想像できますか。私はシンプルでスマートな実装は思いつかないです。

解き方はいろいろあると思いますが、一番簡単な解法の一つとして、以下のようなものが考えられます。

1. 2の1乗は 123456789 より大きいか -> No
2. 2の2乗は 123456789 より大きいか -> No
3. ..
4. 2のN乗は 123456789 より大きいか -> No
5. 2のN+1乗は 123456789 より大きいか -> YES
6. N+1桁あれば 123456789 を表現可能だとわかる

この処理では2を1乗、2乗とループ処理でどんどん大きくしていきますが、最終的に2の何乗になるかがわかりません。このようなときに「特定の条件をクリアするまでループを回す」ためにwhileを使うと便利です。

以下にwhile文の使い方のイメージ図をのせます。



上記の図を見てもうとわかるように、while文はループを回るごとに条件式をチェックして、それがTrueならループを継続して、Falseならループを抜けるという処理をします。これはJavaやCのwhileとまったく同じです。

先ほどの2進数の桁数を求めるプログラムをwhileで書いてみます。

```
a = 123456789
i = 1
while(2**i < a):
    i+=1
    print(i)
# 27
```

すでに扱った内容ですが、上記のプログラムを補足すると、2*i*は「2の*i*乗」を計算していて、*i*=1は*i*をインクリメント(*i*=*i*+1)しています。*2**i*が123456789より小さい間は*i*をインクリメントしていき、*2**i*が123456789より大きくなったらループを抜けるという動きをします。ループを抜けた際に入っている値が必要な桁数を表しています。

break と continue によるループ制御

ループ制御そのものの打ち切りや「ループのその回だけ」の打ち切りが必要な場面があります。たとえば以下のような偶数が含まれるかを判定するプログラムがあるとします。

```
a = [1,3,5,7,9,10,11,13,15]
has_even = False
for i in a:
    if(i%2 == 0):
        has_even = True
print("List has even: " + str(has_even))
# True
```

上記はリストをfor文で走査し、その要素が偶数であればhas_evenという変数にTrueをいれます。リストに偶数がなければFalseのままで、偶数があればTrueになります。今回はリストの中に10があるので、当然Trueとなります。

ただ、よく考えてみてください。なにか無駄な処理があると思いませんか。そう、ループが10になった回で偶数があることがわかったのに、さらに偶数が含まれるかチェックを繰り返しています。10が現れた時点で偶数があることはわかりきっているので、ループを回し続けるのは無駄です。

「break」を使ってループ処理を打ち切ることで、この問題を解決できます。breakを使ったプログラムは以下のようになります。

```
a = [1,3,5,7,9,10,11,13,15]
has_even = False
for i in a:
    print(i)      # NEW CODE
    if(i%2 == 0):
        has_even = True
        break     # NEW CODE
print("List has even: " + str(has_even))
```

確認のためにbreakだけでなく、print文も追加しています。これを実行すると以下のようになります。

```
3  
5  
7  
9  
10  
List has even: True
```

どうです、11以降のチェックをしなくなりました。iに10が代入されたタイミングでbreakが実行され、ループから抜けているためです。

一方「continue」ですが、正直こちらはbreakほど頻繁に利用されない気がします。ただ、ループで「特定の条件の場合だけ処理をしたい」というときに利用されることが多いです。

たとえば、数値1から99のリストのうち、3でも5でも割り切れるものだけを画面出力する必要があるとします。リストを使わないで愚直な書き方をすると以下のようになります(実際はcontinueを使わなくとも、もっとスマートに書けます)。

```
for i in range(1,100):  
    if(i%3 == 0):  
        if(i%5 == 0):  
            print(i)  
  
# 15 30 45 60 75 90
```

上記はrange関数で[1,2,3,...,98,99]というリストを作成し、それに対してforループを回しています。もし、iが3で割り切れたら、もしもiが5で割り切れたら……などというように条件分岐がどんどん深くなってしまいます。これをcontinueを使って書き直すと、次のようになります。

```
for i in range(1,100):  
    if(i%3 != 0):  
        continue  
    if(i%5 != 0):  
        continue  
    print(i)  
  
# 15 30 45 60 75 90
```

行数は増えてしましましたが、プログラムの見渡しはよくなりましたね。このように使いようによっては、breakとcontinueは便利です。

個人的に私がよく使うのは「whileの条件判定にTrueをいれた無限ループ」をbreakで抜けるというものです。たとえば以下のような構造です。

```
while(True):  
    処理  
    if(条件):  
        処理  
        break  
    処理
```

気をつけないと無限ループから抜けられなくなりますが、適切に使えばwhileの条件式に判定をいれるよりも綺麗なコードが書けます。

3項演算子

今まで扱ってきたif文やループに比べてあまり利用されないため後にもってきましたが、実はif文の亞種の3項演算子というものも存在します。

例えば以下のプログラムがあるとしましょう。

```
a = 5  
  
if(a % 2 == 0):  
    is_even = True  
else:  
    is_even = False  
  
print(is_even)  
# False
```

上記プログラムはaという変数に格納されている値が偶数であればis_evenという変数にTrueをいれ、そうでなければFalseをいれるというものです。プログラムを書いていると、時々こういうようなif文の結果に応じて同じ変数に入れる値が変わるという場面に出くわします。

3項演算子はまさにこういった処理をするために用意された文法で、以下のように書きます。

```
a = 5  
  
# (Trueのときに返る値) if (条件式) else (Falseのときに返る値)  
is_even = True if (a % 2 == 0) else False  
  
print(is_even)  
# False
```

1ライナーの条件式

最後に1ライナーと呼ばれるテクニックを紹介します。インデントとコードブロックで説明したように、基本的にはifやforの処理は字下げして書きます。ただ、1行だけの処理であれば条件式の直後に処理を書くことができます。

例えば以下のようになります。

```
a = 5  
if(a==5): print('a == 5')  
else: print('a != 5')  
# a == 5  
  
for i in [1,2,3]: print(i)  
# 1  
# 2  
# 3
```

Pythonの分かりやすさは綺麗なインデントにあるといつても過言ではないので、乱用は避けたほうがいいです。ただ、あまりにも短い単純

なコードであれば1ライナーで書いてしまったほうが見やすい場合もあります。

関数

Pythonに限らず、多くのプログラミング言語には「関数(Function)」という概念があります。関数は特定の機能を「呼び出す」ために使われます。たとえば今まで利用していたprint()も関数のひとつで、()の中に入れた変数や定数をスクリーンに出力するという処理を呼び出します。本章ではこの関数について扱います。

関数の必要性

プログラムは必ず、キーボード入力やファイルの読み込みといった「外からの情報の入力」と、ディスプレイヤスピーカー出力といった「外への情報の出力」を必要としています。入出力のないプログラムも作れますか、動かしても「プログラムの外の世界」に何も影響がないので、CPUとメモリを無駄に消費するという目的以外には使えません。変数や条件分岐といった制御はあくまでもデータを処理するための手段でしかなく、プログラムの外とやりとりするためには関数が必須です。先に説明した画面への文字出力を行うprint関数もその一例です。

そして「関数を使わなくても書けるコード」も、うまく関数を使うことで「分かりやすいコード」になります。例をあげて説明してみます。絶対値を得ようと思った場合、以下のようにif文で条件分岐させることで実現が可能です。

```
x = -5
if(x<0):
    x = x * -1
print(x)
# 5
```

2、3行目を見てもわかるが、「もし $x < 0$ なら x に-1をかける」という処理をしており、絶対値を得ているということが読み取れます。同じ処理を、絶対値を得る関数abs()を使って書くと、以下のようにになります。

```
x = abs(-5)
print(x)
# 5
```

前者と後者は同じことを実現していますが、どちらのほうがわかりやすいと思いますか。圧倒的に後者です。

関数は突き詰めるとその中身は処理の塊だといえます。ただ、関数の名前はその「処理の要約」なので、どういう実装で処理が実現されているのかは関数の利用者に隠蔽されています。たとえばabs()では、「どうやって絶対値を求めるか」を理解していないなくても、それを使えば絶対値が得られることが分かります。人間の思考能力には限界があるので、ゴチャゴチャとした実装を見せられるよりは、関数名という要約を見せられたほうが何をやっているか判断しやすくなります。複雑な処理を隠蔽して機能を分かりやすく提供する、これも関数の利点の1つです。

他にも関数の利用場面は多くあります。例えばプログラムを書いていると、「同じ処理を何度も使う」という場面が多々あります。たとえば「2つの数字の絶対値を比較する」というプログラムを作る場合、関数を使わないと以下のように絶対値を得るコードが2回出現する冗長なものとなります。

```
x = 5
y = -10

if(x<0):
    x = x * -1
if(y<0):
    y = y * -1

print('abs x > abs y ?')
print(x > y)
# False
```

x と y の絶対値を得る処理はほとんど同じなのにもかかわらず、2回書いています。2回程度なら書いてもいいような気もしますが、これが5回、10回となればどんどん面倒になっていきます。

先のコードを、関数を使って書き直すと以下のようにになります。

```
x = 5
y = -10

x = abs(x)
y = abs(y)

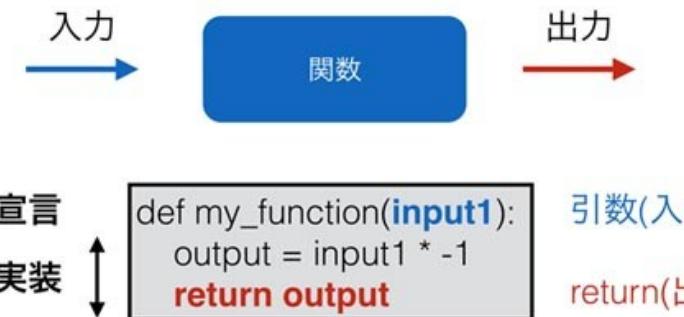
print('abs x > abs y ?')
print(x > y)
# False
```

冗長な要素が省かれてすいぶんと読みやすくなったのではないかでしょうか。今回はabsのような簡単な関数でしたが、これがたとえば100行以上必要なアルゴリズムだったら、関数で冗長性を減らすことによる多大な恩恵を得られます。

関数を使わず同じコードを何度もコピペで書いていると、そのなかに「バグ(プログラムのミス)」があることがわかった場合にすべての場所で修正を施す必要があります。一方、関数で同じ処理をまとめていると、理想的には一箇所のみの修正となります。これはプログラムの「保守性(メンテナンス性)」を向上させるというメリットがあります。

関数の定義と利用

関数のメリットがわかったところで、関数をどのように作って利用するかを具体的に説明していきます。以下の図を見てください。



これが関数の基本的な概念です。関数は入力を受け取り、それを加工して出力する。基本的にはこれだけです。先程の関数 `abs` を自分で作るとしたら、たとえば以下のようなものとなります。

```
def my_abs(x):
    if(x >= 0):
        return x
    else:
        return x * -1

print(my_abs(5))
# 5

print(my_abs(-3.3))
# -3.3
```

関数の入力と出力はそれぞれなくともかまわず、入力がない場合は関数の引数(入力の宣言)をなくし、出力が不要な場合はreturn文(出力の宣言)をなくします。

具体的には以下のように関数を定義します。

```
# 引数がない関数
def my_func1():
    return 0

# 戻り値がない関数
def my_func2(x):
    x = x * -1
```

関数をどう呼び出すかについては今までさんざん利用したのでなんとなくわかると思いますが、宣言した引数に対応する箇所に入力値を入れることで呼び出します。ひとつ目の変数は引数がないので、呼び出し時に()に何も入れていないものの、後者は引数をとるので()に値を与えています。返り値については `return` されたものが返ってきて、なにも `return` されていない場合は `None` が返ってきます。この `None` については後述しますがようするに「なにもない」ということです。

```
print(my_func1())
# 0

print(my_func2(5))
# None
```

ほかに知っておくべきこととしては以下のものがあります。引数は複数指定できる `return` 文は一度しか実行されない

これも具体例を示しましょう。

```
def my_func3(x, y):
    print('A')
    if(x > y):
        return x
    print('B')
    return y

print(my_func3(5,1))
# A
# 5

print(my_func3(2,4))
# A
# B
# 4
```

上記関数では入力値を2つとっています。コンマで区切られた引数の数のぶんだけ入力を受け付けるという簡単なルールです。そして、内部では2つの `return` 文が確認できます。注目して欲しいのは $x > y$ の条件を満たす場合は `print('B')` が実行されていないということです。`return` 文はいくつあっても構わないのですが、`return` されたあとの関数の処理は一切無視されます。

ほかには「デフォルト引数」や「可変長引数」といった関数の書き方もありますが、高度なトピックとなるため本書の最後の章にてアドバンスドトピックとして扱います。

ローカル変数

関数内では引数も含めて変数を宣言できます。例えば以下の関数があるとしましょう。

```
def test(x):
    a = x * 2
    return a

print(test(5))
# 10
```

これは与えられた引数を2倍して返すという単純な関数です。内部に引数(関数内部では変数と同じ扱い)である `x` と変数 `a` が存在しています。実はこれらの変数は関数の外からはアクセスできません。

なんとなくですが、上記の関数を呼び出す前に `x` や `a` にアクセスしたら問題が起きそうな気がするのですが分かります。なぜなら、一度も呼ばれていないから `x` や `a` になにも入っていないからです。ただ、実際には関数を呼び出した後であっても関数内の引数や変数に関数外からアクセスすることはできません。

具体的には以下のようなエラーとなります。

```
print(test(5))
# 10

print(x)
# Traceback (most recent call last):
# ...
# NameError: name 'x' is not defined

print(a)
# Traceback (most recent call last):
# ...
# NameError: name 'a' is not defined
```

重要なことなので覚えておいて下さい。

グローバル変数

さっそくですが次のコードを見てください。

```
x = 5

def add():
    x += 1

print(x)
add()
print(x)
```

このプログラムを実行するとどのようになると思いますか。`add`という関数が`x`に1をインクリメントするので、以下のようになるでしょうか。

```
5
6
```

実行してみると分かりますが、実際はエラーが出てしまいます。

```
Traceback (most recent call last):
  File "/Users/yuichi/Desktop/hello.py", line 7, in <module>
    add()
  File "/Users/yuichi/Desktop/hello.py", line 4, in add
    x += 1
UnboundLocalError: local variable 'x' referenced before assignment
```

上記エラーを見ると、関数の中で「変数`x`に値が与えられる前に参照した」というような内容となっています。結論から言いますと、プログラム1行目の`x`と関数内の`x`は別物です。そのため関数内の`x`を使おうとしたところ、エラーが出てしまったのです。関数内に存在しない1行目の変数`x`は「グローバルレベル」に存在する変数です。一方、関数内は「その関数のレベル」で変数などが定義されています。そのため、グローバルレベルの変数を関数内で使いたい場合は、関数がそれを利用できるように「global 変数名」と宣言をしてあげる必要があります。具体的には以下のようになります。

```
x = 5

def add():
    global x  # global宣言
    x += 1
```

こうすることで、関数の外で定義された変数`x`を関数の中で利用することができるようになります。他のプログラミング言語の経験者であればグローバルレベルから関数内の変数を見れないことを知っている人は多いのですが、Pythonにはその逆があることを知らないことが時々あるようです。グローバルレベルと先程説明した関数レベルの変数のアクセスについて以下の図にまとめます。



関数内からグローバルは `global` 宣言すれば見えるが、グローバルレベルから関数内の変数や引数は見えない。基本的にはこのルールさえ守れば問題ありません。

ただ、実は`global` 宣言しなくても関数内からグローバル変数にアクセスできる例外があります。それは「参照するだけ」の場合です。例えば以下のようなプログラムはエラーなく実行できます。

```
x = 5
def print_x():
    print(x)
```

```
print_x()  
# 5
```

見て分かるように関数内でグローバル変数xを「参照」のみしています。これは問題ありません。一方、関数内でグローバル変数に「代入」のコードが存在する場合、代入する前に参照するとエラーになります。

```
x = 5  
  
# OK  
def print_x1():  
    print(x)  
  
# OK  
def print_x2():  
    x = 3  
    print(x)  
  
#ERROR  
def print_x3():  
    y = x + 1  
    x = 3  
    print(x)
```

上記例だとprint_x1, print_x2は上記エラーとなる条件を満たしません。一方、print_x3及び最初に提示したx+=1は条件を満たすためエラーとなるのです。正直などころ、こんなことまでいちいち意識して関数は書いてほしくありません。global変数を参照するだけでもglobal宣言を書いたほうが分かりやすいです。

ちなみに「参照のみの場合はグローバル宣言が不要」というルールを生かして、グローバルレベルにある定数を使うことは全く問題はありません。なぜなら定数は値が再代入されることがないからです。ただ、繰り返しますが関数内でグローバルレベルの変数は利用しないべきです。2つの関数がなにか一つのものをともに利用しないといけないといったケースでは、ついグローバル変数を関数内で使ってしまったくなりますが、これは中編にて扱うクラスを使うことで簡単に解決できます。

副作用

難しい話となってしまうのですが、関数などを動かすことで「期待される範囲を超えた外の世界に影響がおよぶ」ことを「副作用」といいます。関数の章でも話しましたが、基本的に関数は「入力を受け取って出力を返す」ことをその役割としています。そのため、関数を動かすことでの「グローバルレベルの変数x」を変更するのは一般的には副作用だと言えます。

この副作用を減らすことがきれいなコードを書くコツですので、先ほどのグローバル変数xを変更するコードであれば、global宣言を使うより以下のようにするのが一般的です。

```
x = 5  
  
def add(x):  
    return x + 1  
  
print(x)  
# 5  
x = add(x)  
print(x)  
# 6
```

こうすることでグローバル変数xと関数の処理が切り離されます。最終的にグローバル変数xは更新されるという点では同じですが、副作用という点では全く異なります。繰り返しますが、関数内でのグローバル変数へのアクセスは可能な限り避けて下さい。

モジュール

Pythonのプログラムは書けば書くほど大きくなります。数百行のコードでしたらひとつのファイルに書けないことはないですが、何千行にもなってくるとコードを複数のファイルに分けたほうが管理はしやすいです。これは日常生活の整理整頓とまったく同じです。たとえば洋服ダンスがあるとすると、それを使いやすく使うためには下着、シャツ、ズボンといった種類ごとに引き出しを分けて使うと思います。ひとつの大好きなダンボール箱にすべての服をつっこんでしまうとどこに何があるかわからなくなり、なおかつ服もきれいに管理できずにシワシワになってしまいます。プログラムのファイルを分けないと、後者のような乱雑な服の管理法に近い形でコードを書くことになります。ひとつの大きなファイルのなかにさまざまなコードをゴチャゴチャと書くのでどこで何をやっているのかわからなくなってしまいます。

一方、特定の処理ごとにファイルを分けて「このファイルはXXの処理」「このファイルはYYの処理」などと整理すると、XXの処理を追加したり修正したりする際にすぐに場所がわかります。Pythonではこの「ファイルに分けられた各プログラム」のことをモジュールと呼んでいます。既存のモジュールを使ったり、自分で新しくモジュールを使ってプログラムを書いていったりすることになります。

本章ではこのモジュールを使ったり、作ったりする方法について学びます。



モジュールなし
コードが雑然とする



モジュールを利用
コードが整理される

まずはPythonが提供してくれているモジュールを利用するところからはじめていきましょう。モジュールを利用するには「import宣言」が必要です。たとえば、数学処理がまとめられたmathモジュールを利用するには以下のようにします。

```
# import モジュール名
import math
```

このようにimportをすると、mathモジュールに入っている関数などが利用できるようになります。たとえばmathモジュールの切り捨て関数を使うには以下のようにします。

```
>>> import math
>>> math.floor(5.5)
5.0
```

モジュール内の関数を呼び出すには"モジュール名.関数()"とします。mathモジュールに含まれる関数であれば全て「math.関数名()」という形で呼び出します。importされるモジュール名が長かったりわかりにくかったりする場合はそれに別名を付けることができます。それは以下のように行います。

```
# import A as B
# モジュール A を B として import
import tkinter as tk

font=('Helvetica', 32, 'bold')
# 呼び出しあは tk
label = tk.Label(text='Hello Python', font=font, bg='red')
label.pack()
label.mainloop()
```

上記例ではtkinterと呼ばれているGUIのモジュールをtkとしてimportしています。

モジュールの関数を呼び出すたびに毎回モジュール名を書くのが面倒であれば「from」を使うことで、モジュール内の関数をモジュール名なしで呼び出すことも可能になります。

```
# from モジュール名 import 関数名
from math import floor

print(floor(5.5))
# 5
```

頻繁に使う関数であればこのように使用しても構いませんが、一般的にはfromを使わずにimportのみを行うほうが分かりやすいコードになります。なぜならその関数がどのモジュールに属しているかひとめで分かるからです。

先ほどの「import A as B」とこのfromを組み合わせることで、あるモジュールに属する関数を別名で使うこともできます。

```
# math モジュールの floor 関数を f として import
from math import floor as f

print(f(5.5))
# 5
```

2つのモジュールが同名の関数を持っていて、2つとも使いたいといった場合に利用可能ですが、そのような状況ではモジュール名ごと関数を書くほうが賢明かと思います。

モジュール内の関数すべてをモジュール名なしで呼び出すには以下のようにワイルドカードを使います。ただ、このような乱雑なモジュールの利用法は「名前の衝突(同じ名前が2箇所で使われてしまっている)」などの問題が発生しかねないので、あまり推奨できません。

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

あるモジュールをテストするといったシナリオでは有効な使い方ですが、本番環境用のコードでは使用しないほうがよいです。

モジュールの作成

モジュールの作成は簡単です。本連載の最初に説明したように、「.py」という拡張子をつけたファイルにpythonのコードを書くだけです。モジュール名(ファイル名)はアルファベットの小文字と数字のみから構成されていることが望ましいとされていますが、それに加えてアンダーバーを使うこともあります。ここではモジュールutil.pyを作成し、それをmain.pyから呼び出す例を示します。両ファイルは同じディレクトリで作成し、そのディレクトリ内でプログラムを実行してください。

```
#util.py
def say_hello():
    print('hello!')
def say_python():
    print('python!')
```

上記が呼び出される側のPythonのプログラムです。2つの関数が定義されています。そして以下がそれを呼び出す側のPythonのプログラムです。

```
#main.py
import util
util.say_hello()
util.say_python()
```

main.pyを実行すると以下のような出力が得られ、Pythonのプログラムのファイルが別のPythonのプログラムのファイルを呼び出していることが分かります。

```
hello!
python!
```

特に難しいことはありませんね。

モジュールを書くにあたって注意すべきすることは、モジュールが以下の特性を持っているかということです。

- 再利用可能か
- 似た処理のみをまとめているか

たとえば標準ライブラリで提供されていない特殊な数値計算が必要なら、その計算のためのモジュールを作ってもよいでしょう。ただ、そこに特殊な文字列処理であったり、ネットワークの処理も書いたりするというのは誤った設計です。数値計算のモジュールであればそれに徹すべきです。文字列処理、ネットワークの処理についても同様です。また、そのモジュールを誰しもが簡単に使えるようにすることが理想です。例えばよくわからない名前の関数を作っていたり、変な副作用などがあったりすると扱いに困ります。実際は複雑なプログラムを分割するためだけにモジュール化することも多いのですが、それでも「使いやすい」ように書くことを心がけておくといいかもしれません。

なお、実行するプログラムからモジュールが見つからずエラーとなることがたまにあります。サンプルを試す際に「両ファイルは同じディレクトリで作成し、そのディレクトリ内でプログラムを実行してください」とお願いしたのはこの問題を防ぐためです。モジュールの探索には規則があり、それを知ることで別の場所にあるモジュールを読み込むこともできます。ただ、それらは難しいため本書ではなく本シリーズの下編にて扱います。

モジュールのリロード

モジュールを開発している際にモジュールのコードを更新しても、それをimportしている側は昔のコードのモジュールを保持し続けます。新しいコードのものを参照して欲しい場合はそのモジュールを「リロード」することが必要で、それは以下のように行います。

```
import imp
imp.reload(モジュール名)
```

モジュールimpのreloadという関数を使うことでモジュールのリロードができます。実際に先ほど作成したutilモジュール(util.py)を使いながらこの挙動を確認してみます。インタプリタを起動しモジュールの関数を呼び出します。

```
>>> import util
>>> util.say_hello()
hello[]
```

先ほど定義した通りの動きをしました。このutilモジュールのファイルを以下のように更新し保存します。

```
# util.py
def say_hello():
    print('hello!' * 3)
def say_python():
    print('python!' * 3)
```

モジュールのファイルを新しくしたあとで、インタプリタで再度同じ関数を呼び出してみます。

```
>>> util.say_hello()
hello!
```

出力を見てもうかるように「昔の定義」のまま動いています。試しにモジュールを再度importし関数を呼び出してみます。

```
>>> import util
>>> util.say_hello()
hello[]
```

再度importをしても挙動は変わっていません。次に先ほど紹介したモジュールのリロードを行い、関数を実行してみます。

```
>>> import imp
>>> imp.reload(util)
<module 'util' from '/Users/yuichi/Desktop/util.py'>
>>> util.say_hello()
hello[] hello[] hello[]
```

出力を見ると分かりますが、モジュールが更新されていることが分かります。

本番環境でモジュールをリロードすることはあまりありませんが、サーバープロセスなどにおいて動的に実行するコードを更新するといった目的にも使えるかもしれません。

モジュールの読み込みと実行

Python はモジュールを読み込む際に実行をしています。そのため、先程までのような関数のみのモジュールを import しても全く影響はありませんが、特定の処理をするコードを書くとそれが実行されてしまいます。例えば以下の関数を定義しているモジュール util.py ですが、これを import するだけで 4 行目が実行されて test と出力されてしまいます。

```
def test():
    print('test')
test()
```

そのため、モジュールとして読み込まれることを想定して開発された Python のプログラムファイルは、実行されるコードを含まないべきです。モジュールを読み込んだ際になんらかの初期化処理が必要な場合でも関数などとして提供するほうが行儀はいいです。なぜなら import をする側は import をするだけで勝手になんらかの処理をすると想定していないからです。

ただ、モジュールがプログラムの起点になることもあれば import されることもあるという場合は、「起点となる場合はある処理をする」一方、「モジュールとして呼び出される場合はそれをしない」という実装が必要なことがあります。これを実現するには特殊な変数 `__name__` を使います。これは特殊属性(詳細は本シリーズの下編にて扱います)と呼ばれる高度なトピックなのですが、難しいことは考えずにこれにはモジュール名が入っていると認識して下さい。たとえば util.py をモジュールとして読み込めば util となりますし、testmodule.py を読み込めば testmodule となります。ただし、一つ例外がありプログラムの起点となるプログラムは、モジュール名がファイル名ではなく `__main__` となります。

せっかくなので実際に試してみます。以下に3つのファイルがあるとします。

hello.py

```
import nice
import world

print('hello.py: ' + __name__)
```

nice.py

```
def fun_nice():
    print('nice')

print('nice.py: ' + __name__)
```

world.py

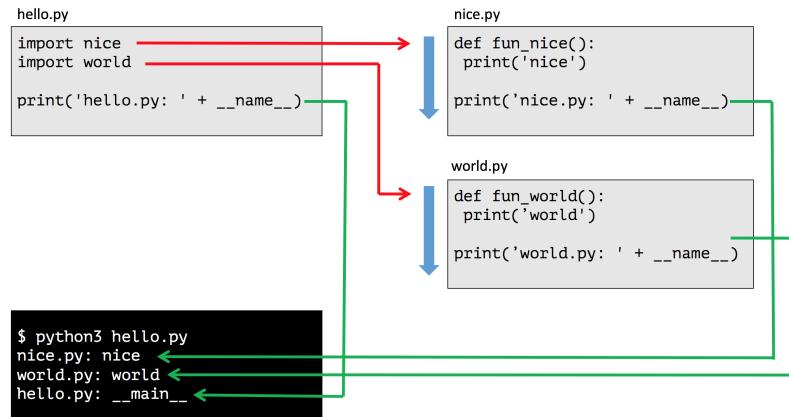
```
def fun_world():
    print('world')

print('world.py: ' + __name__)
```

コードを読んでもらえれば分かりますが、hello.py は nice.py と world.py を import しています。 nice.py 及び world.py はモジュールとして使われることを想定していますので、簡単な関数の定義を持っています。そしてそれらのファイルは最後にモジュール名のプリントをしています。 hello.py を起点に呼び出すと以下のようない出力が得られます。

```
$ python3 hello.py
nice.py: nice
world.py: world
hello.py: __main__
```

この際、Python は以下の図のようにコードを読み込み、print 出力をしています。



出力を見てもうと分かりますが、import されたモジュール内にある特殊変数 `__name__` には、ファイル名(モジュール名)が入っています。ただ、起点となるファイルの `__name__` だけはファイル名ではなく `__main__` となっていることが分かります。

この `__name__` の性質を利用して、そのモジュールが起点として呼ばれた場合のみ特定の処理をすることができます。想像がつくかもしれませんのが、以下のようなものとなります。

test.py

```
def fun_test(x):
    return x**2

if(__name__ == '__main__'):
```

```
print('start function tests')

if(4 != fun_test(2)):
    print('error')
if(9 != fun_test(-3)):
    print('error')
```

上記のコードは **name** がもし **main** だった場合、つまりこのモジュールが起点としてプログラムが起動された場合にモジュールで定義された関数をテストしています。このモジュールが呼び出されるたびにいちいちテストをするのは問題ですが、モジュールとして使われるプログラムを起点と呼び出した場合にだけテストをするというのは理にかなっています。この書き方は様々なところで使われているので、ぜひ覚えておいて下さい。

モジュール紹介

Python には数え切れないほどのモジュールがあります。モジュールには大きく分けて2つあり、標準モジュール、及び外部のモジュール(インストールなどが必要)があります。有用なモジュールは多くあるのですが、それらを多く語るのは本書の趣旨にそぐわないため下編に譲ります。ここでは重要なモジュールのみ駆け足で紹介します。

まず Python の環境自体に関わる **sys** モジュールがあります。インタプリタとしての挙動に関わる関数を多く持ち、たとえば **sys.exit()** はプログラムを終了させます。他には標準入出力や標準エラー出力を扱う機能も持っており、**sys.stdin.read()** や、**sys.stdout.write()**、そしてエラー出力の **sys.stderr.write()** があります。改行なしの **print** 文を使いたい場合は改行をしないオプションを加える(後述します)か、代わりに **sys.stdout.write()** を使うという方法があります。

```
import sys

print('hello')
print(' python')
print('\n')

sys.stdout.write('hello')
sys.stdout.write(' python')
print('\n')
```

上記コードを見てもらうと、同じ出力を **print()** と **sys.stdout.write()** で出力させています。このプログラムを実行すると以下のようない出力が得られます。

```
hello
python

hello python
```

出力を見てもうと分かりますが、**sys.stdout.write** は出力後に改行されていません。改行なしでコンソールに出力をしたい場合は使ってみて下さい。

次に OSに関わる処理として **os** モジュールがあります。OS モジュールはちょうどターミナルや DOS Prompt、power shell のような、シェルを使ったファイルやディレクトリの操作に近いことができます。例えば以下のようなものとなります。

```
# 現在のディレクトリを取得
>>> os.getcwd()
'/Users/yuichi/Desktop'

# ディレクトリを移動
>>> os.chdir('test')

# ディレクトリの中身を確認
>>> os.listdir('.')
['.DS_Store', '1.txt']

# ディレクトリを作成
>>> os.mkdir('hello')

# ディレクトリを削除
>>> os.rmdir('hello')
```

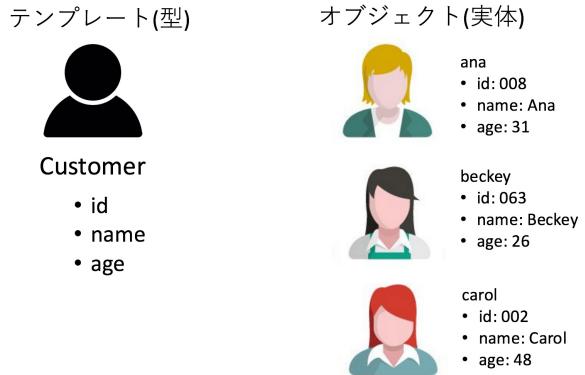
os モジュールは子モジュールも持っており、たとえば **os.path** モジュールはパス関係の処理をできます。他には時間関係の **time** モジュールや数学関係の **math** モジュールなどもよく利用されます。それらの使い方の詳細は適時Python の公式ドキュメントなり、webサイトや書籍を参照してください。

Python のモジュールの数は膨大ですが、プログラマが実現したい処理は一般的なことであればたいがいは既になんらかの形で提供されています。どういうものがあるか手を動かしながら試し、概観を掴んでしまえば簡単にそれらを利用できるようになるはずです。ただ、提供されるモジュールやライブラリはPythonやプログラミングの最低限の知識を必要とする場合が多いです。モジュールの詳細について学ぶ前に、まずは Python やプログラミングの基礎知識を身につけることに注力するのがよいと思います。

オブジェクトとメソッド

今まで型や変数、制御構造や関数といったテーマを扱ってきました。これらは昔から続く「手続き型言語」において重要な概念です。ただ、昨今のプログラミング言語は「手続き型言語」の進化系ともいえる「オブジェクト指向型言語」と呼ばれるジャンルに属していることがほとんどです。もちろんPythonもオブジェクト指向型言語です。この「オブジェクト指向」については話すことが山ほどあるため、本書の続編でメインに扱います。ただ、これを全く理解していないと本書(上編)の後半が理解しにくくなるため、少し背伸びをしてその概念の説明をしたいと思います。

まず「オブジェクト指向」における「オブジェクト」なのですが、これは簡単に言ってしまうと「データの塊 + a」です。例えばお客様の情報を扱うとしましょう。ここではお客様ID、名前、年齢を管理しているとします。オブジェクトはここでは「具体的な顧客情報」にあたり、それはテンプレートのような枠組み(型)から作成されます。



上記の図でいえば、まずCustomerという一般的な「型」があり、ana, beckey, carolはその型の具体的なデータ、つまりオブジェクトだといえます。anaのID、名前、年齢はあくまでanaのものなので、それぞれ個別にではなく「Customerという単位」でまとめて扱ったほうが簡単です。beckey、carolについても同じです。なお、上記のオブジェクトはインスタンスと呼ばれることもあり、idやnameといったオブジェクトに付随するデータはインスタンス変数と呼ばれています。

そしてオブジェクト指向の「データの塊 + a」のaの部分ですが、それはメソッドと呼ばれる「オブジェクトのデータを操作する処理」となります。つまりオブジェクトは「データの塊 + 処理」といえます。

たとえばanaの登録された年齢を変更したいとしましょう。この時、anaという「オブジェクトに対してメソッドset_ageを呼び出す」ことでanaの年齢を変更します。



図にかかれているように「オブジェクト.メソッド名(引数)」とすることでそのオブジェクトを操作することができます。このset_ageというメソッドはanaの年齢を変更するということが明白です。

ちなみに、オブジェクトのメソッドはなにもオブジェクトを変更するだけではなく、そのデータを「参照する」ためにも使われます。例えば「年齢を確認する」こと自体はオブジェクト自体には影響は与えません。



参考までにanaオブジェクトの操作イメージを以下にコードで示してみます。(実際には、私がCustomerというクラスを作成してそのデータやメソッドなども定義し、そこからanaを作成しています。ただ、作成は難しいので省略しています。あくまでも利用イメージだけに着目してください。)

```

# 現在のオブジェクトの状態をダンプ
>>> ana.dump()
id: 8
name: Ana
age: 31

# ana オブジェクトのメソッドを呼び出し年齢を更新
>>> ana.set_age(32)
# オブジェクトの中身が更新されている
>>> ana.dump()
id: 8
name: Ana
age: 32

# ana オブジェクトから値を取り出す(変更はなし)
>>> ana.get_age()
32
>>> ana.dump()
id: 8
name: Ana
age: 32

```

今回は例に ana を使いましたが、同様の操作は becky 及び carol に対して行うこともできます。これらのオブジェクトはどれも Customer という型から作成されたものであり、同じ型から作られたオブジェクトには同じ操作ができるからです。

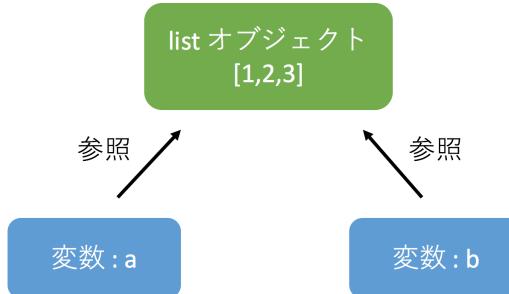
本章以降ではこのオブジェクトとメソッドが利用されますが、基本的には「オブジェクトはデータを内部に持つ」とこと「メソッドはオブジェクトを操作(中身を変更)もしくは参照(取り出すだけで変更しない)」するということを意識してもらえば、使っているうちに慣れてくると思います。なお、Python のデータは実は全てオブジェクトです。数字もオブジェクトであるため、実はメソッドを呼び出すこともできます。あえてそのような使い方はされませんが、全てはオブジェクトであるということは覚えておいて下さい。

オブジェクトの参照

オブジェクトはメモリ上に存在し、オブジェクトが代入された変数はそれを参照しています。言葉で書くと難しいですが、コードで見ると簡単です。

```
a = [1,2,3]
b = a
```

上記コードではまず、リストのオブジェクトである[1,2,3] が存在し、それが変数 a に代入されています。そして変数 a は変数 b に代入されています。この操作をすると以下の様な状態になっています。



図を見るとわかりますが、変数 a, b は共に同一のリストのオブジェクトを持っているといえます。そのため、変数 a に対して操作をすると、それは変数 b にも影響を与えます。

```

a = [1,2,3]
b = a

a.append(4)
print('a: ' + str(a))
# a: [1, 2, 3, 4]

print('b: ' + str(b))
# b: [1, 2, 3, 4]

```

リストのオブジェクトに対して append メソッドを呼び出すと、リストの末尾に要素を追加することができます。append したあとの print 出力を見ると変数 a に格納されているリストオブジェクトのデータがアップデートされています。そして変数 b に格納されているリストオブジェクトは変数 a のものと同じなので、変数 b 自体に対しては操作をしていないのに中身が変わっていることがわかります。

変数に対する操作は変数が格納しているオブジェクトに対する操作だといえます。その変数にどのオブジェクトが入っているかは常に意識するようにしてください。自分が意図していないところでオブジェクトが操作されてしまっていて、使うときに問題が起きるというのもよくあるトラブルです。

オブジェクトの属性の確認

先ほどの ana オブジェクトには name, age といったインスタンス変数や get_age() といったメソッドが存在しましたが、これらはオブジェクトが持つ「属性」と呼ばれるものです。この属性としてどのようなものがあるかは dir 関数を使うことで確認できます。dir 関数の引数としてオブジェクトを与えると、属性の一覧をリスト形式で返してくれます。たとえば文字列型の属性であれば以下のようにになります。

```

>>> dir('text')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__init__',
 '__getnewargs__', '__gt__', '__hash__', '__init_subclass__', '__iter__', '__le__',
 '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__rmod__', '__rmul__', '__setattro__', '__sizeof__', '__str__',
 '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
 'lstrip', 'maketrans', 'partition']

```

```
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

オブジェクトのメソッド名を忘れてしまった際に dir 関数を使ってどういうものがあるか確認できます。ぜひ活用してみてください。

テキスト処理

テキスト処理は要するに、文字列型の処理です。前の章で扱ったように文字列型もオブジェクトであるため、様々なメソッドが利用できます。テキストファイルの処理では文字列型の基本知識が必要なので、以前学んだことを少し発展させて復習します。

まず、文字列は以下のように定義するのでした。

```
text1 = 'hello python'  
text2 = '''hello  
world  
python'''
```

ひとつめに関しては今さらいうこともないですが、2つめに関しては複数行でテキストをプログラム内で定義する方法でした。記号「」の代わりに記号「"」を使うことも可能ですが、文字列の前後で統一されている必要があります。

文字列の結合に関しては「+」記号でできますが、数字などを結合するときは「文字列に変換」してから結合するのでした。ほかの型から文字列型への変換にはstr関数を使います。

```
print('hello ' + 'world')  
# hello world  
print('hello ' + str(5))  
# hello 5
```

文字列長は以下のように len() 関数を使うことで取得できます。

```
length = len('hello world')  
print(length)  
# 11
```

文字列中の「文字」の取得は以下のように [X] で位置を指定して行います。

```
text = 'hello world python'  
print(text[4])  
# o  
  
print(text[-4])  
# t  
  
print(text[100])  
# Traceback (most recent call last):  
# ...  
# IndexError: string index out of range
```

この位置の指定はリストの要素の数え方と同じで0から始まります。先頭から0、1、2……と数えていくと4はoに対応しています。面白いのがこの値をマイナスにできるところです。このように指定すると後ろ側から取得してきます。この際、0からではなく-1、-2、-3……とカウントすることに注意してください。文字列の範囲を超えてアクセスしようとするとエラーになります。

文字列から「文字列」を取得するには、以下のように行います。

```
text = 'hello world python'  
print(text[6:11])  
# world  
  
print(text[-12:-7])  
# world
```

これは「スライシング」と呼ばれるテクニックで、[X:Y]とあるとXからYまで取得という意味になります。[X:Y]と指定する際はX < Yとなるようにしてください。先ほどと同じように、範囲指定にもマイナス値を利用できます。

前と後ろを指定するのではなく、Xより前、Xより後という指定の仕方も可能です。

```
print(text[6:])  
# world python  
  
print(text[:11])  
# hello world  
  
print(text[:])  
# hello world python
```

見ていただくとわかるように [X:Y] の片方を省略しています。そうすると先頭から、もしくは末尾までという意味になります。あまり使いどころはありませんが、両方とも省略すると、文字列のすべてが取得されます。このスライシングはリストから複数の要素を取り出す際にも使えます。

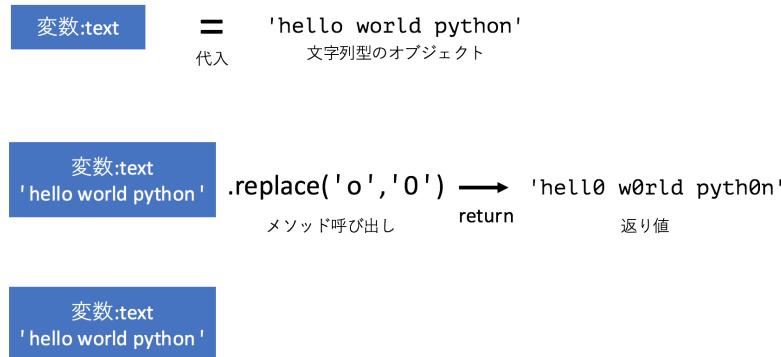
文字列型のメソッド

先にお伝えしたように文字列型もオブジェクトであるためメソッドが利用できます。ただ、文字列型に対してメソッドを呼び出した場合は「呼び出し元の文字列オブジェクトは変更されない」という規則があります。例えば次にお話する文字列の置き換えも、置き換えメソッドを呼び出した文字列は変更されず、メソッドの返り値として変更された文字列オブジェクトがかえってくるという動きをします。これは全ての文字列型のメソッドに共通した特徴なので必ず覚えておいて下さい。

文字列型のメソッドを使ってテキストの置き換え処理をしてみます。テキストエディタなどである特定のキーワードを別のキーワードに置き換えることがあるかと思いますが、それと同じ要領です。サンプルコードを以下に記載します。

```
text = 'hello world python'  
text1 = text.replace('o', '0')  
print(text1)  
# hello w0rld pyth0n  
  
print(text.replace('world', 'WORLD'))  
# hello WORLD python
```

文字列.replace(置き換える文字列, 置き換えられる文字列)とすると、変換された文字列が返されます。例にもあるように、元の文字列自体は変化していません。1文字だけを変更することもできますし、文字列を変更することもできます。不要かもしれませんのが、上記のコードのオブジェクトとメソッドの関係を図に示します。



'hello world python' という文字列型のオブジェクトを変数 text に格納し、そのオブジェクトに対してメソッド replace を呼び出しています。そのメソッドは変換した文字列を返り値として返しますが、元のデータ(オブジェクトが持つ'hello world python' というテキストデータ自体)は変更を加えません。

文字列の検索もそれほど難しくはありません。検索には「存在の確認」と「位置の確認」の2つの使い方があり、それぞれ次のようにあります。

```
text = 'hello world python'
print('wor' in text)
# True

print('w0r' in text)
# False

print(text.find('wor'))
# 6

print(text.find('w0r'))
# -1

print(text.find('o'))
# 4
```

inについてはlistでの使い方と同じで、「A in B」の場合は「AがBに含まれていれば Trueを返す」という動きをします。'hello world python' というテキストに'wor' は含まれているので True となっています。find メソッドについては最も左側にあるマッチした位置を返します。そのため、'o'は何個もありますが、一番左の位置となっています。マッチしない場合は-1が返ってきます。

find メソッドに似たメソッドで startswith と endswith があります。名前から分かるかもしれませんのが、前者は「この文字列から始まっているればTrue」、後者は「この文字列で終わっているればTrue」という動きをします。

```
text = 'hello world python'
print(text.startswith('hell'))
# True

print(text.startswith('hell0'))
# False

print(text.endswith('hon'))
# True

print(text.endswith('h0n'))
# False
```

それほど使う場面は多くないのですが、find メソッドのオプションである第二引数を指定することで前側を指定した数だけ飛ばして途中から検索することもできます。右側から探索をする rfind というメソッドも利用できます。

```
text = 'hello world python'
print(text.find('o', 10))
# 16

print(text.rfind('o'))
# 16
```

次に文字列の前後からの特定の文字の削除メソッド stripです。よく利用するのは、前後の空白や改行コード、タブなどを取り除く場合などでしょう。

```
text = ' hello world \n'
print(text.strip())
# 'hello world'

print(text.strip(' hell'))
# 'o world \n'
```

strip関数に引数を指定しないと文字列の前後の空白文字(空白とタブ、改行)が取り除かれます。引数に文字列を指定すると、その文字列が取り除かれます。このstrip()関数はファイル読み込み処理とともに「改行コードを取り除く」ことによく使われることがありますので、覚えておいて頂いたほうがいいかもしれません。左側の文字だけを取り除く場合は lstrip、右側だけの場合は rstrip メソッドを使います。

```
text = ' hello world \n'
print(text.strip())
# 'hello world'

print(text.lstrip())
# 'hello world \n'
```

```

print(text.rstrip())
# ' hello world'

text = '1, taro, 35, male'
print(text.split(','))
# ['1', 'taro', '35', 'male']

text = 'hello\nworld\npython'
print(text.splitlines())
# ['hello', 'world', 'python']

print(text.split('\n'))
# ['hello', 'world', 'python']

```

上記サンプルにあるようなコンマ「,」記号での文字列の分割は CSV(Excel出力)やログの解析あたりでよく使うテクニックです。たとえば以下ではトリプルくおテーションで作ったCSV形式のテキストを、まず改行コードで分割して1行ずつにして、各行においてコンマでテキストを区切っています。

```

text = '''1, taro, 35, male
2, jiro, 29, male
3, hanako, 23, female'''

for line in text.split('\n'):
    elems = line.split(',')
    print(elems[1].strip() + ':' + elems[2].strip())

# taro:35
# jiro:29
# hanako:23

```

この例のように CSV の要素に空白が含まれているのであれば、先ほどの strip 関数と組み合わせて前後の空白を取り除いて整形してあげてもいいかもしれません。

「オブジェクトを返す関数やメソッド」に対してメソッドを呼び出しているようなコードはよくありますが、それは上記と同じように「作成されたばかりのオブジェクトのメソッドを呼ぶ」ということをしています。例えば以下の様なコードです。

```

print('hello world python'.replace('he', 'HE').replace('py', 'PY'))
# HELLO world PYthon

```

これは分かりやすく書けば以下のようになります。

```

a = 'hello world python'
b = a.replace('he', 'HE')
print(b.replace('py', 'PY'))
# HELLO world PYthon

```

メソッドをチェーン上に並べるのは長くなりすぎなければ積極的に活用するべきだと思います。ただ、例えばマイナーなメソッドを呼び出したりバグを生んだりしそうな場所での利用は避けて頂いたほうが無難です。メソッドをチェーン上に並べた場所でエラーが発生すると、そのエラー原因の特定が困難な場合があります。

他には文字列の大文字、小文字の変換あたりもよく使います。

```

>>> a = 'Hello Python'
>>> a.lower()
'hello python'
>>> a.upper()
'HELLO PYTHON'

```

文字列の生成

テキストを整形して文字列を作ることはよくあります。たとえば、文字列型の名前と数値型の年齢を合わせて「名前：年齢」とするには以下のようにすればできます。

```

text = 'taro' + ' : ' + str(35)
print(text)
# taro : 35

```

上記は文字列の結合を使って文字列を生成しています。数値型と文字列は直接結合できないため、数値35はstr()関数を使って一旦文字列にしています。

3つの文字列ぐらいであれば結合を使うことで文字列を作れますか、あまりに長くなってくると結合で文字列を作るのは非常に見苦しくなってきます。結合の代わりに、文字列にテキストや数字を埋め込むという手法で文字列を生成することができます。これを実現する format メソッドを使うと非常にシンプルに文字列を作ることができます。

```

text = '{} : {}'.format('taro', 35)
print(text)
# taro : 35

```

文字列のformat関数(メソッド)の引数に {} に対応する文字列なり数値なりを与えていきます。ひとつ目の{}がformatメソッドの1つめの引数に対応し、2番目の{}が2番目の引数に対応、といった具合で文字列に引数を埋め込んでいきます。私はそれほど利用しませんが、このformatはもっと複雑な使い方もできます。たとえば {} の中に数字やキーワードを埋め込むことで引数の何番目をそこに埋め込むかを調整できます。

```

print('{0} {2} {1} {0}'.format('a', 'b', 'c'))
# a c b a

print('{user} : {age}'.format(user='taro', age=35))
# taro : 35

```

上記のように同じ引数を何度も埋め込むこともできます。キーワード引数を使うこともできます。

さらに format メソッドは文字列に埋め込む「フォーマット(形式)」も調整できます。フォーマットを指定するには「:」のあとにフォーマットを指定する書式を書きます。かなり細かく調整できますが、いくつか便利なものだけ紹介します。

まず数字の桁の調整です。

```
# 5桁
print('{:5}'.format(50))
# ' 50'

print('{:5}'.format(255))
# '255'

# 5桁(左寄せ)
print('{:<5}'.format(255))
# '255   '

print('{:<5}'.format(50))
# '50   '

# 5桁(0埋め)
print('{:05}'.format(50))
# '00050'

print('{:05}'.format(255))
# '00255'

# 位置指定やキーワードとの併用
print('{ab:05} {cd:05}'.format(ab=50, cd=255))
# '00050 00255'
```

コロンの後に数字を指定することで表示される桁数を調整できます。0を指定する桁数の前に付けることで0埋めされます。

数値型の表示を調整することもよくあります。

```
# 整数、 少数
print('{:d}, {:f}'.format(5, 5.5))
# '5, 5.50000'

print('{:.3f}, {:.5f}'.format(5.5, 5.5))
# '5.500, 5.50000'

# 3桁区切り
print('{:,}, {:10,}, {:010,}'.format(111111, 111111, 111111))
# '111,111, 111,111, 00,111,111'

# 基数変換(2進数、8進数、10進数、16進数)
print('{:b}, {:o}, {:d}, {:x}'.format(100))
# '1100100, 144, 100, 64'
```

format メソッドに似た書式化演算子(%)も Python3 では利用できます。書式化演算子は format メソッドより気軽に使えますが、そのぶん単純なことしかできません。以下のように使います。

```
ベースとなる文字列 % 埋め込む値
ベースとなる文字列 % (埋め込む値1, 埋め込む値2,...)
```

ベースとなる文字列に値を一つだけ埋め込む場合はそれを % 記号の後におきます。そして複数の値を埋め込む場合は % 記号のあとに埋め込む要素をタプルにまとめて配置します。例えば以下のようにになります。

```
text = 'hello %s' % 'HELLO'
print(text)
# hello HELLO

text = 'hello %s python %s' % ('HELLO', 1)
print(text)
# hello HELLO python 1
```

ベースとなる文字列の「%s」の箇所に値が埋め込まれていることがわかります。この %s はどのような形で値が埋め込まれるかを指定する「変換指定子」と呼ばれており、%s は文字列、%d は整数、%f は少數として利用されます。

```
print('hello %s python' % (5.5))
# hello 5.5 python

print('hello %d python %f' % (5.5, 5.5))
# hello 5 python 5.50000
```

format メソッドと書式化指定子は奥が深く、様々な使い方ができます。ただ、込み入った使い方は頻繁には利用されないため、必要になつたタイミングでドキュメントなどを参照すれば問題ないと思います。C言語などの printf に慣れているかたは書式化演算子のほうが書式を見慣れているため魅力的に見えるかもしれません。ただ、今後の Python は format メソッドを主流としていく見込みなので、可能であれば format メソッドを使うようにしてください。

リストに含まれる複数の文字列を「特定の文字列」で結合していくことも可能です。これはちょうど先程の split メソッドの逆です。2次元配列(リストにリストが入っている)に格納された情報を CSV 形式でファイルに書き出したりする際に便利な手法です。書式は「結合に使う文字列.join(文字列のリスト)」となります。空白で結合すればそのままつながります。

```
a = ['1', 'taro', '35', 'male']
print(', '.join(a))
# 1, taro, 35, male

print(''.join(a))
# 1taro35male
```

エスケープ記号

最後に文字列で使われる特殊記号についてお話しします。特殊記号はプログラム内で意味を持つてしまう特別な記号のことです。たとえば「\」という記号は文字列を作成する際に利用する特別な記号です。そのほかにはビープ音なども記号に分類されます。これらは文法的な理由やそもそもそれを表現する記号がキーボードのキーにないことから、「これは XX ですよ」という特別なルールにもとづいて文字列に表記します。そのルールに利用されるのがエスケープ記号と呼ばれるもので半角のバックスラッシュ「\」(英語キーボード)か、半角の円記号「¥」(日本語キーボード)を利用します。このエスケープ記号の後に特別な文字を続けることで、それが特別な意味を持つのです。

たとえば「\」とビープ音は以下のように記載できます。

```
print('escape sample1 \'\'')
print('escape sample2 \a.')
```

ほかには改行とエスケープ記号自身あたりをよく使います。

```
print('escape sample1 \n.')
print('escape sample1 \\.')
```

エスケープ記号一覧は調べてもらえばすぐに出できますので、興味がある人は検索してみてください。

リスト再入門

文字列と同じく、リストもオブジェクトです。そのため、まずは以前扱わなかったメソッドを使う処理を紹介します。

リストにデータを追加する方法です。追加するといつても「リスト末尾への追加」と「リストの途中への追加」でやりかたが異なります。末尾(一番最後)への追加は `append` メソッドを使い、間に追加するには `insert` を使用します。

```
a = [1,2,3]
a.append(4)    # 末尾への追加
print(a)
# [1, 2, 3, 4]

a.insert(1,10)  # 1番目の要素に10を追加
print(a)
# [1, 10, 2, 3, 4]
```

追加とくれば削除です。削除には `remove` メソッドを使います。

```
a = ['a', 'b', 'c', 'd']
a.remove('b')
print(a)
# ['a', 'c', 'd']

a.remove(1)
# Traceback (most recent call last):
# ...
# ValueError: list.remove(x): x not in list
```

`remove` メソッドはリストの中にある要素自体を指定してそれを消します。存在しない要素を指定するとエラーになりますので、含まれるかどうか分からぬ場合は `in` 演算子で存在を確認したあとで `remove` して下さい。`del` 演算子を使った要素の削除もできますが、こちらは `remove` メソッドとは異なり消す要素を「インデックス」で指定します。何番目の要素を消すか番号で指定するということです。これはスライスやリスト末尾からの指定などもできます。

```
a = [1,2,3,4,5]
del a[2]
print(a)
# [1, 2, 4, 5]

a = [1,2,3,4,5]
del a[1:3]
print(a)
# [1, 4, 5]

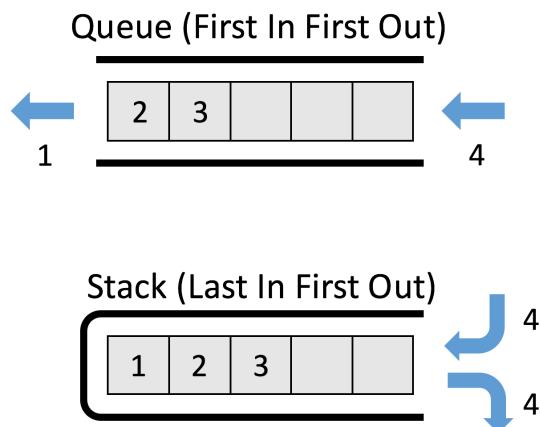
a = [1,2,3,4,5]
del a[-1]
print(a)
# [1, 2, 3, 4]
```

次に `pop` メソッドです。これは `del` 演算子に似ているのですが、消すだけではなく同時に値を取り出します。引数にインデックス番号(先頭が0)を渡せばそれに対応する要素を取得してリストから消します。引数を与えない場合はリスト末尾の要素を取り出します。

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

c = a.pop()
print(a)
# [1, 2, 3]
print(c)
# 5
```

様々なプログラミング言語でリストを使ってキューやスタックと呼ばれるデータ構造を実現することがよくあります。キュー及びスタックは複数のデータを管理するためのデータ構造です。キューは別名FIFO (First In First Out) とも呼ばれており、最初にいれたデータを最初に取り出すというデータ構造です。一方、スタックはLIFO (Last In First Out) と呼ばれており、その名前が示すように最後にいれたデータを最初に取り出すというデータ構造です。以下に両者がどのようなものか示す図を記載します。



キュー及びスタックに1,2,3 というデータが入っているとします。キューにデータ4をいれると1,2,3,4 というデータを持ちます。そこからデータを取りだすと先頭(最初にいれたデータ)にある1を取り出し、2,3,4となります。一方、スタックは最後にいれたデータを最初に取り出します。そのため、4を入れられて1,2,3,4となっているデータから取り出されるのは4となります。リストへの追加には既に利用した `append` メソッドを利用し、データの取り出しには `pop` メソッドを使います。

以下に list を使ったキューの実現方法を記載します。

```
queue = [1,2,3]
queue.append(4)
print(queue)
#[1, 2, 3, 4]
a = queue.pop(0)
print(a)
# 1
print(queue)
#[2, 3, 4]
```

append でリストの最後にデータを追加し、pop(0) でリストの最初のデータを取り出しています。そして次にスタックです。

```
stack = [1,2,3]
stack.append(4)
print(stack)
#[1, 2, 3, 4]
a = stack.pop()
print(a)
# 4
print(stack)
#[1, 2, 3]
```

スタックからデータを取り出すということはつまり、リストの最後の要素を取り出すということです。そのため、引数なしの pop メソッドでリストの最後の要素を取得します。

次はリストとリストの結合です。これにはプラス演算子を使う方法と extend メソッドがあります。append を使うと、append されたリスト自体がメソッドを呼び出したリストの最後の要素になるので結合はされません。

```
a = [1,2,3,4]
b = [5,6,7,8]
print(a + b)
#[1, 2, 3, 4, 5, 6, 7, 8]

a.extend(b)
print(a)
#[1, 2, 3, 4, 5, 6, 7, 8]
```

プラス演算子とextend メソッドは一見すると同じに見えますが、もとのデータがどうなるかという点でかなり違います。まずプラス演算子ですが、結合されているリスト a,b はそのまま残り、両者が結合された新しいリストが作られています。一方、extend メソッドを使うと呼び出し元のリスト a の後ろに指定されたリスト b が結合されます。リスト b 自体は変更が加わりませんが、リスト a はもとのデータではなくなります。

他にはリストの順序を反転するreverse メソッドや、中身の順序をソートする sort メソッドあたりも利用されます。

```
a = [1,2,3,4,5]
a.reverse()
print(a)
#[5, 4, 3, 2, 1]

b = [1,3,5,2,4]
b.sort()
print(b)
#[1, 2, 3, 4, 5]
```

リストのコピーや初期化は以下のように行います。

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

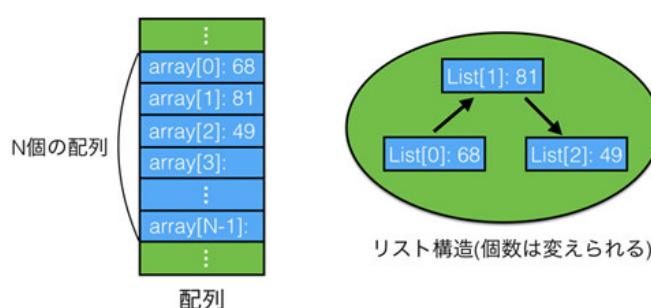
a.clear()
print(a)
[]
```

先にお伝えしたようにリストを格納する変数を別の変数に代入したとしても、それはコピーではなく同じオブジェクトを共有するだけです。つまり片方の変数の値を操作すると、もう片側の変数の値も変わっているということです。上記のようにコピーを行った場合は別々のオブジェクトとなるため片側の操作がもう片側に影響をあたえることはありません。

リストの仕組みと配列との違い

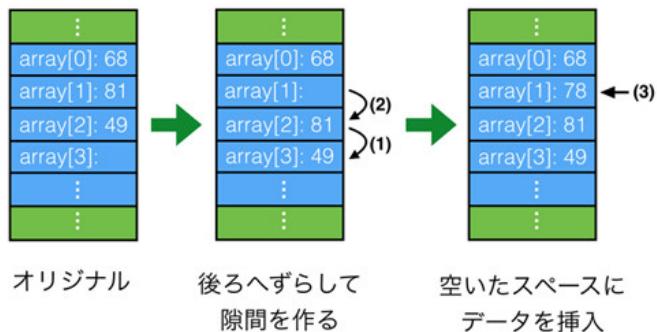
ここではCやJavaの「配列」とPythonのリストの比較を通して、Python のリストがどのようなものか説明します。Python では配列相当のものがないので必ずしも学ぶ必要はないかと思いますが、この違いはプログラミング一般において重要なため、C や Java を学ばれたことがないかたも一読していただくといいと思います。

生徒の点数を扱うことを題材にして配列とリストを比較してみます。まず簡単に両者のイメージ図を以下に記載します。

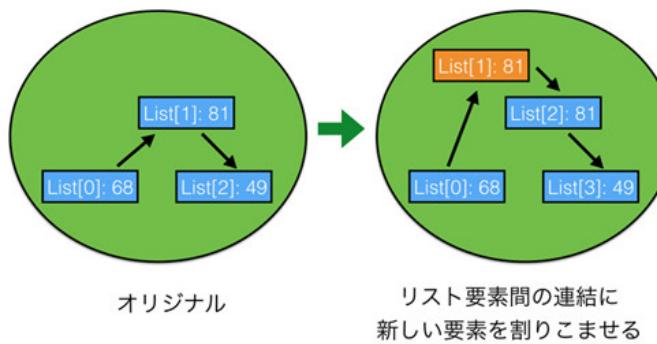


左側のCやJavaの配列は「メモリ」上に連番でデータを格納するスペースを用意するのに対し、右側の「リスト」はバラバラの複数のデータ間を順序を持って結びつけることで実現されています。

Pythonのリストは、CやJavaのVectorやListに相当する型です。Pythonのリストはまるで配列のように利用されますが、CやJavaの配列とは大きく異なります。たとえば純粋な配列では、「要素(配列やリストの中にあるデータ)」の間に新しいデータを挟み込むことはできません。そのため、配列に入っている要素を詰め替えるなどしてデータを追加します。



一方、リストはN-1番目の要素とN番目の要素の間に新しいデータを挟み込むことができます。配列ではメモリ上に要素を連番で格納するためのスペースを用意するのに対し、リストは以下の図のようにバラバラに用意された要素間を順に結びつけることで実現されているためです。



Javaの配列のコードを確認してみます。

```
int a[] = {0, 1, 2};  
System.out.println(a[1]); // 2番目の要素の値を取得 -> 1  
System.out.println(a.length); // a の配列長を取得 -> 3  
a[1] = 10;  
a[3] = 3; // Error
```

1行目では要素数3のint型の配列の変数を宣言し、それに代入しています。先の章でお伝えしたように、Javaの変数には型があるのでした。a[x]とすると配列aのx番目の要素にアクセスできます。そして、a.lengthとすることで配列長が取得できます。

4行目では配列の2番目の要素に値を代入しています。ただ、5行目では配列長3の4番目の要素に値を代入しようとしているのでエラーとなってしまいます。

次にPythonのリストを使ってみます。Pythonの変数には型がないので、特に型を指定していない変数aに[0, 1, 2]という3要素のリストをそのまま代入しています。2行目ではリストの中身を確認しています。そして3行目では配列長を取得しています。

```
a = [0, 1, 2]  
print(a[1])  
# 2番目の要素の値を取得 -> 1  
print(len(a))  
# a の配列長を取得 -> 3  
  
a[1] = 10;  
a.append(3)  
print(len(3))  
# 4
```

異なるのは5行目です。配列は配列長を超えて要素を代入することができませんが、リストはリスト長を伸ばすことができます。Pythonではリストをまるで配列のように使いますが、両者はあくまでも別物という認識を持っておく必要があります。

タプル

リストという型がなぜ存在するのか覚えていますか。ひとつの型のなかに任意の数の複数のデータを格納できると便利だからでした。たとえば「生徒たちの成績を格納する」といった目的で利用されます。タプルもリストと似ていて、ひとつの型のなかに複数のデータを格納します。ただ、その目的は異なっていて、「決まった数の複数のデータがひとつの意味を持つもの」にタプルは使われます。

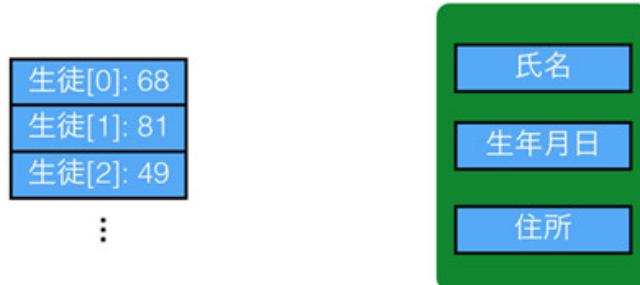
例をあげて説明しましょう。お店の会員情報をデータで表現することを考えてみます。例えば 'taro, 1986, tokyo' といった具合で単純に文字列としてすべてを含めてしまってもいいのですが、以下のような複数の要素を持つデータとして表現したほうがプログラムで使いやすそうです。

- 氏名
- 生年月日
- 住所

テキスト処理で文字列の中から生年月日を抜き出したりするよりも、「Aさんの住所 -> 東京都...」というようにぱっと取り出せる方が便利です。これはちょうど先ほど説明したオブジェクト指向におけるメソッドがないクラスのようなものです。

タプルは上記のような複数のデータをひとつにまとめるための型です。Python のリストは様々な種類のデータを格納することができるため、同じことをリストでも実現できます。ただ、リストは「可変長(長さが変わる)」なので、複数のデータが合わさってはじめて意味を持つ場合の利用は本来の用途ではなく、それよりも同じデータをいくつも格納する用途で使われます。

以下にリストとタプルの違いを図にまとめます。



リストのデータ
例: 生徒たちの点数

タプルのデータ
例: 会員情報

タプルがどういうものかわかつていただけたと思うので、具体的にどのように使うのか説明しましょう。タプルの作成はタプルの要素となる値を () で囲むことで作成できます。type関数は型を確認するために利用する関数です。

```
a = ('taro', '1986', 'tokyo')
print(type(a))
# <class 'tuple'>
```

上記は先ほどの会員情報のタプルです。その要素への参照はリストに似ています。

```
a = ('taro', '1986', 'tokyo')
print(a[0])
# 'taro'
print(len(a))
# 3

for i in a:
    print(i)
# taro
# 1986
# tokyo
```

ただし、リストと異なり一度作成されたオブジェクトは変更することはできません。

```
a = ('taro', '1986', 'tokyo')
a[0] = 5
# Traceback (most recent call last):
# ...
# TypeError: 'tuple' object does not support item assignment
```

少しトリッキーな見た目ですが、タプル内の要素を一気に取得することもできます。

```
a = ('taro', '1986', 'tokyo')
(b,c,d) = a
print(b) # taro
print(c) # 1986
print(d) # tokyo
```

タプルはC言語でいう「構造体」やJavaでいう「クラスのメンバ関数」の簡易版として使うのが主な用途です。ただし、構造体やメンバ関数はそれぞれ「変数名」を持っているのに対して、タプルは「何番目にある要素か」ということを基準にしてデータを管理します。

最後にタプルの便利な使い方を紹介します。あまり機会は多くないのですが、関数の返り値を2つ返したい場合がときどき発生します。一つひとつ別の関数に分けて値を取得するようにすることができますが、たとえばfor文での探索のようなプログラムだと計算コストが高いため、無駄に2周するのは避けたほうがよいです。そのようなときにタプルをreturnで使うと便利です。たとえば、リストの要素の最小値と最大値を取得する関数だと、返り値は2つ返したいところです。そのような場合は以下のようにすれば大丈夫です。

```
def get_min_max(list_):
    min_ = a[0]
    max_ = a[0]
    for i in list_:
        if(i < min_): min_ = i
        if(max_ < i): max_ = i
    return (min_, max_)

a = [5,8,1,4,10,3,7]
(min_, max_) = get_min_max(a)
print('min: {}'.format(min_))
# 1
print('max: {}'.format(max_))
# 10
```

返り値をタプルにした際は、タプルの構造に気をつけて利用してください。たとえ関数の返り値のタプルの形式が変更されたにもかかわらず、それを利用する側が変更されなければエラーとなります。複雑な形式をやりとりしたいのであれば、継編で扱うクラスなどにしてしまうほうがよいです。

セット

次は「セット」です。セットを一言で説明すると、「集合」という概念を実現するための型です。たとえば、ある空の集合にAを追加すると、その集合にはAがあります。さらにBを追加するとA、Bの2つが存在します。しかし、ここにさらにAを追加しても、「A、A、B」とはならずに「A、B」のままです。そして集合は「順序を持たない」ので「A、B」も、「B、A」も同じ意味を持ちます。セットの内部実装には非常に重要な概念があるのですが、とりあえず使い方を説明してしまいます。

空のセットのオブジェクトを作るにはset()関数を使います。空でない場合は[]に要素を並べることで作ることもできます。もしくはリストのデータをset関数に渡すことで、中身のあるセットを作ることもできます。

```
a = set()
print(a)
# set()

b = {1,2,3}
print(b)
# {1, 2, 3}

c = set([1,3,5,7])
print(c)
# {1, 3, 5, 7}
```

空のセットを作る際に注意して欲しいことは「変数 = {}」としてしまうと、セットではなく次に説明する「辞書型のオブジェクト」が作成されてしまうということです。

セットへの要素の追加にはadd、削除にはremoveメソッドを使います。

```
a = {1, 2, 3}
a.add(4)
print(a)
# {1, 2, 3, 4}
a.add(2)
print(a)
# {1, 2, 3, 4}

a.remove(1)
print(a)
# {2, 3, 4}
```

リストのデータ追加のメソッド名を覚えていますか。addではなく、appendやinsertでしたね。日本語で言うと同じ「追加」であっても、addだと「集合に加える」という感じで、appendだと「末尾に加える」という意味合いになります。順序を持たないset型なので、addというメソッド名となっています。そしてremoveメソッドを使うと指定した値をセットから取り除きます。存在しない値を指定するとエラーになるので注意してください。

セットの使い方はリストに似ています。

```
# in, not in を使える
a = {1,3,5,7}
print(1 in a)
# True
print(2 in a)
# False

# pop を使える(何が帰ってくるかは分からない)
b = a.pop()
print(a)
# {3, 5, 7}
print(b)
# 1

# for も使えるが順序は不明
for i in {1,3,5,7}:
    print(i)
# 1 3 5 7
```

リストと若干異なるのはセットの結合です。先程説明したようにセットは重複した要素を持たないため、結合される2つのセットが同じものを持っていれば1つだけになります。プラス記号を使った結合はできません。

```
a = {1,2,3}
a.union({3,4,5})
print(a)
# {1, 2, 3, 4, 5}

print(a + {5,6,7})
# Traceback (most recent call last):
# ...
# TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

あまり利用場面は多くないかもしれません、集合特有の演算をすることもできます。

```
# 比較
print({1,2,3} < {1,2,3,4})
# True

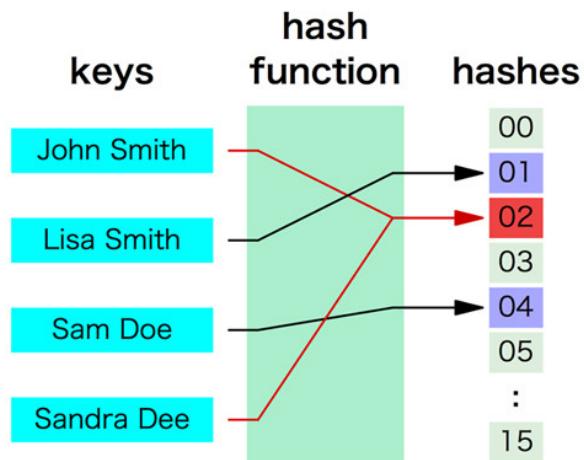
# AND(両方含むものののみ)
print({1,2,3} & {3,4,5})
# {3}

# OR(結合)
print({1,2,3} | {3,4,5})
# {1, 2, 3, 4, 5}
```

セットの使い方はわかっていただけでしょうか。追加や削除、有無のチェックなどの機能を見ると、なんだかリストに似ているような気がしたかもしれません、両者の仕組みはまったく異なっています。

ハッシュの仕組み

セットで使われている重要なコンピュータ技術に、ハッシュ(Hash)と呼ばれているものがあります。集合はハッシュを使わずにでも実現できるでしょうが、PythonのセットはJavaでいうところのHashSetに近いです。このハッシュの概念図を以下に示します。



ハッシュは「ハッシュ関数」と呼ばれるものに特定の値(キー)を与えて「ハッシュ値」を得ることで実現されています。ハッシュ値はある範囲のなかの数値(一般的には0からN)のどれかとなり、同じキーから生成されるハッシュ値は常に同じです。

上記の図でいうと、キーとして「John Smith」をハッシュ関数にかけるとハッシュ値「02」が得られています。同様に「Lisa Smith」をハッシュ関数にかけると「01」となります。そしてハッシュ値の範囲は00から15です。この性質を考慮したうえで「ある集合に要素Xはあるか」ということをどのようにして実現するか想像してください。

ハッシュを使う場合、たとえば「John Smith」を集合に加える際には、ハッシュ値「02」の場所に「John Smith」を格納します。そして「John Smith」が存在するかどうかのチェックはハッシュ値「02」の場所に「John Smith」がいるかどうか確認すればいいのです。一方、リストの探索であれば「先頭から末尾まで順に「John Smith」かどうかを確認していく」ことが必要です。リストのサイズが大きければこの探索コストは非常に大きくなります。「要素」の探索という面においてハッシュはリストに比べるとあるのに比べると随分スマートだと思いませんか。実際、ハッシュを使った要素の探索は非常に高速です。

ただ、ハッシュも使い方を間違えると効率が悪くなります。もう一度図を見てください。よく見ると「John Smith」と「Sandra Dee」は同じハッシュ値に割り当てられています。これはいわゆる「ハッシュの衝突」と呼ばれており、これが多発すると探索のスピードが遅くなります。なぜなら「Sandra Dee」の有無の確認をする際に「01」を見にいって、そこに「John Smith」やほかの要素がたくさん入っていると、「01」のなかで「リストの探索」のようにして全部をチェックしていかないといけないからです。

この問題を防ぐためにハッシュ値の範囲は十分な広さを持たせる必要があります。たとえば今回のように00から15などという範囲は狭すぎるので、これをもっと広げます。そうすると確率的には衝突は発生しにくくなります。ただ、通常はこんなことを気にしなくともPythonがよしなに処理してくれるので大丈夫です。

辞書型(マップ)

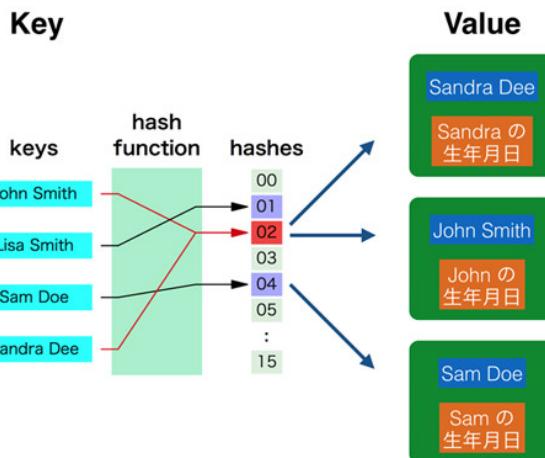
辞書型は、別名で連想配列やマップとも呼ばれている型です。簡単にいってしまえば、重複が許されないキー(Key)とその値(Value)が対応付けられたデータ型です。「キー」という名前からわかるように、これも内部的にハッシュを使っています。

例をあげて説明しましょう。果物(Key)と色(Value)の辞書オブジェクトを作るとすると、

- りんご(Key) : 赤色(Value)
- レモン : 黄色
- ぶどう : 紫
- さくらんぼ : 赤色

というペアが作れます。辞書型を使うと「りんご」と指定すれば「赤色」が得られ、「ぶどう」と指定すれば「紫」が帰ってきます。先ほどのセットと同じように「りんご」というキーは重複が許されずにひとつしか存在することができないため、「りんご : 緑色」というペアを改めて登録すると昔のデータは上書きされてなくなってしまいます。ただ、例にある「りんご」と「さくらんぼ」を見ればわかるように値(Value)の重複は許されています。

勘のいいかたであれば辞書型のしくみの想像がついたかもしれません、簡単にいってしまえば、セットにおけるハッシュの使い方に「Valueも追加」しているだけです。



「John Smith」をキーとして指定するとハッシュ関数で「02」が得られ、「02」のなかから「John Smith」のValueを取得してきます。

それではさっそく辞書型を利用するサンプルプログラムを書いてみます。まずは辞書オブジェクトの生成です。

```
a = dict()
print(type(a))
# <class 'dict'>

b = {}
print(type(b))
# <class 'dict'>

c = {"apple": "red", "lemon": "yellow"}
print(type(c))
# <class 'dict'>
```

辞書オブジェクトの生成にはdict()関数を使う方法と、リストにおける[]に近い{}記号を使うという方法があります。{}を使う場合はその内部で「key:value」という組み合わせをコンマ区切りで列挙すると、そのペアが追加された辞書オブジェクトが得られます。

次に辞書オブジェクトを操作してみます。

```
a = {"apple": "red", "lemon": "yellow"}

# バリューの取得
print(a['apple'])
# red

# キーを指定したバリューの更新
a['apple'] = 'green'
print(a['apple'])
# green

# 新しいキーとバリューの組み合わせの追加
a['orange'] = 'orange'
print(a['orange'])
# orange

# 存在しないキーへのアクセス
print(a['banana'])
# Traceback (most recent call last):
# ...
# KeyError: 'banana'
```

キーを使った値の取得、キーと値のペアの登録を行っています。リストにおけるインデックス番号がキー名に変わっているだけです。"辞書オブジェクト[キー]"としてキーを指定することで、対応する値(Value)を参照します。存在しないキーを参照しようとすると当然エラーとなります。

ただ、存在しないキーを参照した場合に「デフォルト値」を返したいという場合がたまにあります。そのような場合、getメソッドやsetdefaultメソッドを利用します。getメソッドの第一引数にキーを指定し、第二引数にキーが存在しなかった際に取得されるデフォルト値を指定します。第二引数がない場合はデフォルト値としてNoneが使われます。

```
a = {'apple':'red', 'lemon':'yellow'}
print(a.get('apple', 'green'))
# red

print(a.get('banana', 'green'))
# green

print(a)
{'lemon': 'yellow', 'apple': 'red'}
```

辞書オブジェクトにはキー 'apple' が存在するため、'apple' を get した際はそのバリューである 'red' が返されています。ただ、存在しないキー 'banana' を指定した場合はデフォルト値である 'green' が返されています。辞書オブジェクト自身は変更されません。

setdefault は get とほとんど同じですが、その名前からわかるように辞書オブジェクトがアップデートされます。

```
a = {'apple':'red', 'lemon':'yellow'}
print(a.setdefault('apple', 'green'))
# red

print(a.setdefault('banana', 'green'))
# green

print(a)
{'lemon': 'yellow', 'apple': 'red', 'banana': 'green'}
```

セットと同じように辞書型もリストと似た操作ができます。例えばキーの存在確認に in を使うことができます。

```
a = {"apple": "red", "lemon": "yellow"}
print('apple' in a)
# True
print('banana' in a)
# False
```

ほかにも、リストに似た特性があり、pop や for 文での利用も可能です。

```
a = {'lemon': 'yellow', 'apple': 'red', 'banana': 'green'}
print(a.pop('apple'))
# red
print(a)
# {'banana': 'green', 'lemon': 'yellow'}

print(a.popitem())
# ('banana', 'green')
print(a)
# {'lemon': 'yellow'}

a = {'lemon': 'yellow', 'apple': 'red', 'banana': 'green'}
for key in a:
    print(key)
# lemon apple banana
```

キー一覧の取得などもよく使います。値一覧の取得はそれほど使わないかもしれません。

```
a = {'lemon': 'yellow', 'apple': 'red', 'banana': 'green'}
print(a.keys())
# dict_keys(['apple', 'lemon', 'banana'])

print(a.values())
# dict_values(['red', 'yellow', 'green'])
```

ほかにも辞書型の使い方にはいろいろありますが、まずはこのあたりさえ使いこなせれば十分でしょう。

None 型

None 型は「値がない」ということを表明する特別な型です。たとえば返り値がない関数の結果を取得しようとしたら、以下のように None 型が取得できます。

```
a = print('hello')
# hello
print(a)
# None
```

None は C 言語や Java の null に相当するもので、たまに関数やメソッドから期待されるオブジェクトの代わりに返されることがあります。少し高度になるのですが、たとえばある関数内の複雑な処理(例えばネットワークやデータベースへの接続など)に失敗した際に、「Noneを返す」もしくは「例外を発生させる」とことで処理が失敗したこと示すことができます。None を返す実装の場合、返り値がNone であれば失敗したことがわかり、そうでなければ成功したということが分かります。None か否かの判定には is 演算子を使うことが推奨されています。理由が高度になるため説明は割愛しますが、== 演算子での判定は推奨されません。

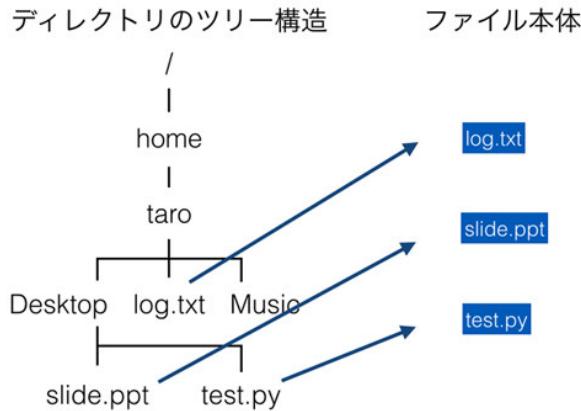
```
return_value = 関数()
if(return value is None):
    失敗した際の処理
else:
    成功した際の処理
```

ファイル処理

ファイル処理

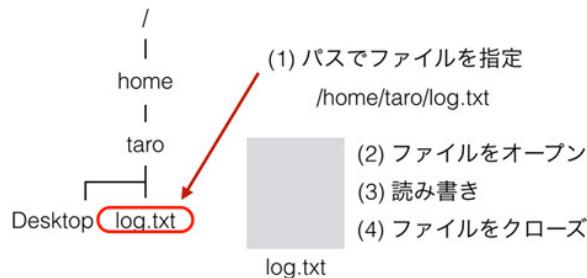
ファイル処理については、プログラミングというよりも「OSのファイル処理の方式」をまず理解しておく必要があります。そのため、最初にファイル処理の概念について説明します。これがわかつてしまえば、その利用はさほど難しくありません。なお、プログラムがどのようにファイルを扱うかは、OSの仕組みにもとづいているため、多くのプログラミング言語でさほど変わりません。

ファイル処理がOSにおいてどう実現されているかを抽象化すると以下の図のようになります。実際はもっと複雑ですが、通常のプログラミングではそこまで意識する必要はないので詳細は割愛します。



まずご存知のようにOSにはディレクトリがあり、それが階層構造を作っています。ファイルはそのディレクトリの中に保存されています。ディレクトリやファイルは、サイズなどの情報と共にポインタのようなものを持っていて、それがファイルの実体を指しています。

構造についての話はこれぐらいにして、実際にファイルをどのように処理するか話をしましょう。OSにおけるファイル処理は主に以下のような流れとなります。



まず絶対パス(ルートやCドライブなどからのパス)や、相対パス(現在いるディレクトリから指示するパス)を使ってファイルを指定します。それに対して読み、書き、読み書きなどのモードを指定してファイルをオープンします。そして読み書きなどの必要な処理を繰り返し、処理がすべて完了したらファイルをクローズして終わりです。

読み書きなどの具体的な処理はそれほど難しくありません。一言でいってしまえば、「テキストファイルは行ごとに処理する」「バイナリファイルは先頭から何バイトめか(位置)を指定して処理する」ことです。たとえば、テキストファイルで以下のものがあるとします。

```
world
python
java
```

この内容にすべて"hello "を加えて画面に表示するというプログラムを書く場合、ループ処理を利用して以下のことを繰り返して処理するのが一般的です。

1. 行の内容を取得
2. hello に行の内容を追加しprint
3. 次の行に進む

「テキストファイルは行ごとに処理する」のが基本であることを覚えておいてください。バイナリファイルの扱いは後ほど簡単に扱います。Pythonでそれを専用ライブラリなしにやることはかなり稀かと思います。私が過去にバイナリファイルを操作した際に利用した言語はCもしくはObjective-Cでした。

テキストファイルの読み書き処理

実際に python でテキストファイルの処理をどのようにするか紹介します。先ほどのテキストファイルの処理方法さえ理解してしまえば非常に簡単です。以下の内容が書かれたファイル text1.txt があるとします。

```
world
python
java
```

このファイルに書かれている各行にhelloを加えて表示するサンプルを書いてみます。

```
f = open('text1.txt', 'r')
print(type(f))
```

```
for line in f:  
    print('hello ' + line)  
f.close()
```

まずファイル 'txt.txt' をモード 'r(読み)' でオープンしています。オープンしたファイルオブジェクトに対してfor文を使うと1行1行取得できるので、行ごとにprintする処理をしています。これを実行すると以下のような出力となります。

```
<type 'file'>  
hello world  
hello python  
hello java
```

print文の改行に加えてもとのテキストの改行コードも表示されるので業の間にもう1行スペースがあいてしまっていますが、この回避方法については先に説明した sys.stdout.write() を使うか後述する print 文の出力オプションを使います。

ほかにはファイルを丸ごと読む方法もあります。

```
f = open('text1.txt', 'r')  
text = f.read()  
print(text)  
  
lines = text.split('\n')  
print(lines)  
  
f.close()
```

ファイルオブジェクトに対してreadメソッドを使うことで、その中身をすべて文字列として取得します。それを行ごとに処理したいのであれば、文字列を先に説明した改行コードで分割することで行ごとのリストになるので、それに対して処理を行うことができます。この処理方式はファイルを全て読み込むため、容量の大きなファイルで実行しようとするときリソースを使いすぎてエラーになる可能性があります。小さいファイルであれば問題無いですが、注意して利用して下さい。

次にファイルへの書き込み方法について説明します。書き込みも読み込みと大差ありませんが、ファイルをオープンする際に書き込みモードを指定します。以下のテキストが既に書かれているテキストファイルtext2.txtに 123456 と書き込みをするとします。

```
Hello
```

書き込みのコードは以下となります。

```
f = open('text.txt', 'w')  
f.write('123')  
f.write('456')  
f.close()
```

コードを見てもらうと想像がつくとは思いますが、openの第二引数が書き込みモードの 'w' となっています。そしてファイルオブジェクトにたいしてwriteすることで、実際にファイルに書き込み処理がされています。そして最後にクローズするのと同じです。

書き込みされたファイル text2.txt は以下のようになりました。

```
123456
```

見てもらうとわかるように、もともとのテキストであるHelloが消えています。ファイルが上書きされていることがわかります。

ただ、場合によっては「追記(もとの中身を残したまま後ろに加える)」しないといけないこともあります。その場合はファイルをオープンするモードを 'a' の「追記」にすれば実現できます。モードのみ修正して以下のコードにしてみます。

```
f = open('text.txt', 'a')  
f.write('123')  
f.write('456')  
f.close()
```

これを実行すると、以下のようになりました。

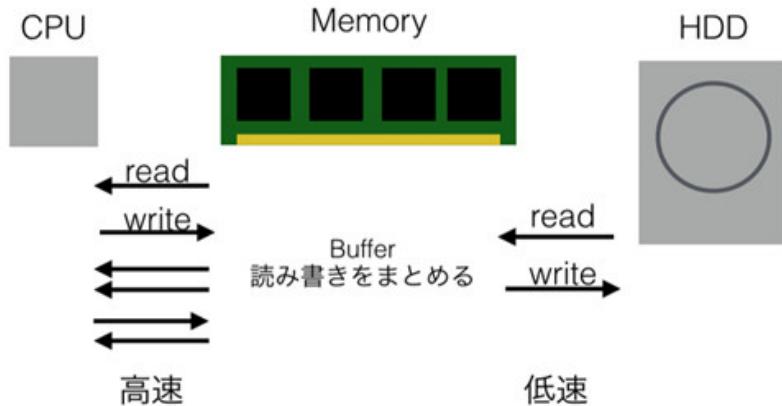
```
123456123456
```

もとの '123456' は残ったままで、その後ろに '123456' が新しく追加されています。ファイルのオープンごとに以前の内容が消えないので、アプリケーションのログなどを取る際に便利な手法です。なお、書き込みを「次の行」にする場合は改行コード"\n"をファイルに書き込めばそこで改行されます。

バッファリング

ファイルへの読み書きをする際に知っておいてもらいたいのが「バッファリング」という処理です。ご存知かもしれません、ハードディスクへのアクセス速度はメモリへのアクセス速度に比べて何桁も遅いです。そのため、ファイルを何度も細かく読み書きすることを繰り返しているとプログラムが非常に低速になってしまいます。この問題を防ぐために、出力があるたびに毎回ディスクに書き込むのではなく、メモリ上の高速な一時領域にデータをおいておき、まとめてそれを読み書きするという処理が行われます。こうすることで低速なディスクアクセスの回数が減らせるのでプログラムが高速化されます。これがバッファリングの基本的な概念です。

以下にこれを図で示します。



このディスクへの書き込みは Python が判断して適切なタイミングで発生するようですが、書き込みを強制的に行いたい場合はそこでflush()メソッドを使います。

```
f = open('text3.txt', 'w')
f.write('123')
f.flush()
f.write('456')
f.close()
```

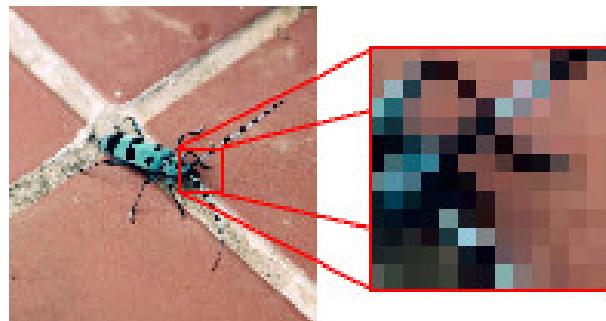
closeのタイミングで必ず書き込まれるので、今回のようにopenからcloseまで時間が短い場合はflushは不要です。ただ、openしっぱなしで、なかなかcloseしないようなプログラムは適切なタイミングでflushするように心がけてください。でないと、プログラムが強制終了されてしまった場合などに、ファイルに書き込みがされていない可能性があります。

バイナリファイルの読み書き処理

テキストファイルの主要な話を終えたため、次はバイナリファイルについて扱います。バイナリファイルは中身が01から構成されているファイルで、一般的には画像ファイルや音声ファイル、それに加えてアプリケーション特有のファイル(たとえばwordなど)があります。こちらはテキストと違うのでそもそも行という概念がありません。正直なことをいうと、テキスト処理よりもバイナリファイルの処理は骨が折れます。ただ、ファイルを読み書きできないかというと、そんなことはありません。そのバイナリファイルの構造を知ってさえいれば操作は可能です。

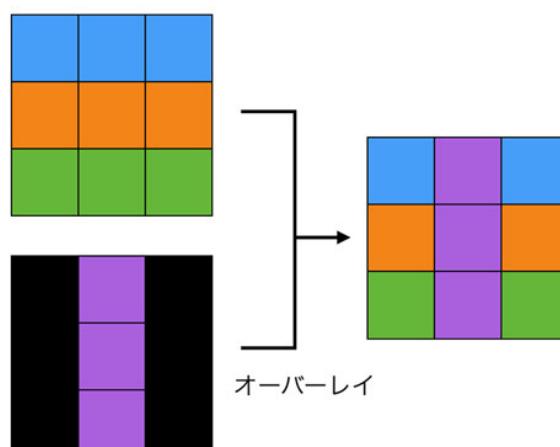
著者はビットマップ形式の画像ファイルの合成とWAV形式の音声データの加工の経験があるので、それをベースにしてバイナリファイルの処理についてお話をします。

ビットマップは以下の図のように、ピクセルから構成されている画像ファイルです。



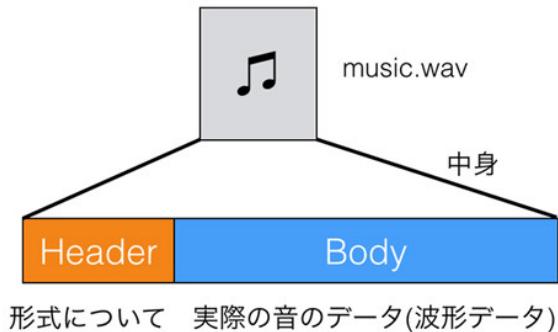
それぞれのピクセルはRGB(赤緑青)で表現されています。それぞれの色は1バイト(0~255)の容量があるので、ようするに1ピクセルは3バイトです。つまりファイルサイズは「縦のピクセル数×横のピクセル数×3」バイトになります。

ここまでわかってしまえば、あとは簡単です。たとえば画像Aに画像Bをオーバーレイ(一部上書き)するとします。この際、Bの画像の黒(RGBが0, 0, 0)は透過させます。すると、以下の図のようにして合成が可能です。



Bの左上は黒なのでAのものを合成画像に利用。その右隣は黒ではないのでBのものを利用。その右隣はA……といった感じでどんどん処理をしていくと、最終的に右の図のようになります。これをファイルに書き込めば、自分でバイナリファイルを作ったことになります。

次にWAV音声ファイルです。これも比較的わかりやすい形式ですが、先ほどのビットファイルと違って「ヘッダ」と「データ」に分かれています。データは先程のビットマップと同じく音声のデータ(波形)を含んでいるだけなので簡単ですが、ヘッダにはデータをどのように表現するかといった情報が含まれています。



後ろのデータを変えれば当然再生される音も変わりますが、その際に必要に応じてヘッダを変更する必要があります。

最後にバイナリデータの処理のコツを伝えます。それは「プログラムで処理しやすい生(raw)の形式に一旦戻す」ということです。たとえばビットマップであれば編集は簡単ですが、JPEGやPNGを編集するのは非常に難しいです。なぜならRAW形式に比べてJPEGやPNG形式が画像をどのように表現するかはるかに複雑だからです。JPEGを直接操作するのであれば、JPEGに関する深い知識が必要になります。その分野を専門としているプログラマ以外には実装することはできないでしょう。

まずJPEG → ビットマップに変換してやり、ビットマップで編集を行う。そしてビットマップ → JPEGに再度変換することでJPEGファイルを変換できます。音声も同じでmp3を直接編集するのではなく、mp3 → wav → 編集 → new wav → new mp3とすればよいです。これらの変換には組み込みもしくは外部のライブラリを利用することになると思います。

with/asによるファイル処理

withとasは「コンテキスト」を扱うための特別な構文です。ほかの言語ではあまり見られない少々独特な概念なのですが、知っておいて損はないと思うので取り扱いたいと思います。コンテキストの詳細については下編で扱います。

コンテキストという言葉は少しボヤッとしているのですが、「ある特定の処理」を実行するための状態(モード)だと言えるかもしれません。たとえばファイル処理なのですが、基本的には以下の流れとなります。

1. ファイルをオープンする
2. 読み書き
3. ファイルをクローズ

これは「ファイル処理のための状態(モード)に入っている」という状況です。当然ながら自分で書くプログラムなので、ファイルを意図的にオープンしっぱなしにして何もしないことも可能ですが。ただ、ほとんどの場合は単にファイルをクローズし忘れてはいるだけです。

withとasはPythonの文法として「ある一連の処理」を完遂することを目的に使うものです。具体的には以下のように使います。

```
with A as B  
    処理
```

Aはコンテキストをサポートする特別なオブジェクトであり、Bはそれが代入されているのですが、ここでは深いことは気にせずファイル処理をwithとasで行ってみます。以下のコードを見てください。

```
with open('text1.txt', 'r') as f:  
    for line in f:  
        print(line)
```

このプログラムはtext1.txtを読み取り専用モードで開き、その中身を一行ずつプリントするというプログラムです。注目して欲しいのは今までのようにopen関数で作ったファイルオブジェクトを代入するのではなく、withとasでコンテキストとして扱っていることです。ここではopen('hello.txt', 'r')がコンテキストをサポートするファイルオブジェクトを返し、それがfに格納されています。そしてこのfを使ってファイル処理をしています。

一般的なopen関数と代入によるファイルオブジェクトの取得ではなく、with/asを使うとファイルオブジェクトがクローズされることが保証されます。

基數と文字コードの仕組み

日本でプログラミングをする以上、プログラミングにおける「日本語の処理」は必須の知識です。ただ、日本語について扱うには「文字コード」と文字データがどのように表現されているかを知るための「基数の知識」が必要です。そのため、順を追って基数と文字コード、そして次の章にて日本語の処理を扱うという手順で説明します。

基数

私たちが普段使う10進数を思い出してください。0、1、2……と数が大きくなっている、8、9となると次は桁があがって10になります。このとき、ひとつ前の桁で表現できる数は0~9の10個であり、この1桁で表現できる数を「基数」と呼びます。そしてその基数Nで表現される数え方を「N進数」と呼びます。たとえば、0~9の10個であれば10進数、0~7の8個であれば8進数になります。

では、0~1で表現されるのは、基数が何で何進数と呼ばれているでしょうか。基数が2なので、2進数です。コンピュータが理解可能なのは01だけという話を何度もしていますが、要するにコンピュータは2進数を使うということです。

ただ、人間は01だけで構成された数字を見せられても、あまりピンときません。たとえば、適当に打った0100010001001110010という2進数の大きさがどれほどなのか、一瞬では把握できません。これを10進数に直すと279794になり、およそ28万であることがわかります。

2進数を人が読みやすい10進数に変換するのは結構骨が折れます。たとえば、01001101という2進数を10進数に変換するには、2進数の各桁の値(0 or 1)に2の「桁-1」乗をかけた値を足しあわせていくという処理を行います。試しに低い桁から順に足しあわせて計算してみます。

```
(1 × 2^0) + (0 × 2^1) + (1 × 2^2) + (1 × 2^3) + (0 × 2^4) + (0 × 2^5) + (0 × 2^6) + (1 × 2^7) + (0 × 2^8)  
= 1 + 0 + 4 + 8 + 0 + 0 + 64 + 0  
= 77
```

補足
N^MはNのM乗という意味。べき乗をテキストで表現できるので便利な書き方
Python的に書くとN ** M
2^0 は1, 2^1は2, 2^2は4, ..., 2^8は128

2進数 01001101 から 10進数の 77 を求められています。疑い深い人は2進数から10進数に変換できるアプリケーションの電卓などもあるでしょうから、確認してみてもいいかもしれません。0進数から2進数への変換はこの逆で、割り算を繰り返すようなことをします。

以上のように「2進数は桁が大きくなりわかりづらい」ものの「10進数は2進数と相性が悪い(変換しにくい)」という問題があるので、コンピュータでは2進数と相性がよい16進数がよく使われます。16進数は0~9までの数字だと16パターンを表現できないので、アルファベットを使って、0から15までを1桁で表現します。具体的には以下のようになります。

```
16進数 : 10進数  
0 : 0  
1 : 1  
2 : 2  
3 : 3  
4 : 4  
5 : 5  
6 : 6  
7 : 7  
8 : 8  
9 : 9  
A : 10  
B : 11  
C : 12  
D : 13  
E : 14  
F : 15  
10 : 16  
11 : 17
```

■ 桁が増える

この16進数は2進数と非常に相性がよく、2進数の4桁をちょうど16進数だと1桁で表現できます。0000~1111は10進数でいう0~15なので、ちょうど16進数の0~Fにピッタリあてはまるのです。1バイト(0から255)は8ビット、つまり2進数が8桁なので、ちょうど16進数2桁で表現できます。

基数変換に便利なPythonの関数を紹介します。10進数から2進数、8進数、16進数への変換は専用の関数を使います。

```
print(bin(100)) # 2進数  
# 0b1100100  
  
print(oct(100)) # 8進数  
# 0144  
  
print(hex(100)) # 16進数  
# 0x64
```

先頭に何かついていますが、0bは2進数、0xは16進数という表明に使います。8進数は0だけを先頭につけるというルールもありますが、桁埋めの0と混同しないように注意してください。

次に、N進数から10進数への変換です。実はこれはすでに使ったことがある関数 int を使います。

```
print(int('100'))  
# 100  
  
print(int('1100100', 2)) # 2進数  
# 100  
  
print(int('0144', 8)) # 8進数  
# 100  
  
print(int('64', 16)) # 16進数  
# 100  
  
print(int('6B', 16)) # A,B,C,...も使える  
# 107  
  
print(int('212', 3)) # あまり使わないN進数も一応使える  
# 23
```

intの第一引数に数字のものとなる文字列を入れ、第二引数に基数を指定します。第二引数を省略すると10進数として扱われます。基数とN進数に関しては以上です。

文字コード

コンピュータは、突き詰めると01しか理解できないので、文字も最終的には01に対応付けられます。その「文字と01の対応関係」を決めるのが文字コードと呼ばれているものです。アルファベットと数字のみを利用する場合は「ASCIIコード」と呼ばれる文字コードが使われることが多いです。たとえばASCIIコードだと「01100001」は「a」に対応し、その次の「01100010」は「b」に対応しています。ただ、先にお伝えしたように2進数だと桁が長いので、一般的には16進数2桁で文字コードを表現します。aとbはそれぞれ以下になります。

```
>>> hex(int('01100001', 2)) # a  
'0x61'  
>>> hex(int('01100010', 2)) # b  
'0x62'
```

以下にASCIIコードの一部を記載します。

ASCII (American Standard Code for Information Interchange) 印字可能文字															
-#	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
2-	½	!	"	#\$	\$	£	¤	‘	()	*	+	,	-	/
	8x18	8x21	8x22	8x23	8x24	8x25	8x26	8x27	8x28	8x29	8x2A	8x2B	8x2C	8x2D	8x2E
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
	8x48	8x41	8x42	8x43	8x44	8x45	8x46	8x47	8x48	8x49	8x4A	8x4B	8x4C	8x4D	8x4F
3-	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>
	8x38	8x31	8x32	8x33	8x34	8x35	8x36	8x37	8x38	8x39	8x3A	8x3B	8x3C	8x3D	8x3E
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62
	8x68	8x61	8x62	8x63	8x64	8x65	8x66	8x67	8x68	8x69	8x70	8x71	8x72	8x73	8x74
4-	Ø	A	B	C	D	E	F	G	H	I	J	K	L	M	N
	8x48	8x41	8x42	8x43	8x44	8x45	8x46	8x47	8x48	8x49	8x4A	8x4B	8x4C	8x4D	8x4E
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78
	100	101	102	103	104	105	106	107	110	111	112	113	114	115	116
5-	Ø	Q	R	S	T	U	V	W	X	Y	Z	[\]	^
	8x58	8x51	8x52	8x53	8x54	8x55	8x56	8x57	8x58	8x59	8x5A	8x5B	8x5C	8x5D	8x5E
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94
	128	121	122	123	124	125	126	127	130	131	132	133	134	135	136
6-	-	a	b	c	d	e	f	g	h	i	j	k	l	m	n
	8x68	8x61	8x62	8x63	8x64	8x65	8x66	8x67	8x68	8x69	8x6A	8x6B	8x6C	8x6D	8x6F
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110
	148	141	142	143	144	145	146	147	150	151	152	153	154	155	156

見てもらうとわかりますが、文字1文字が1バイトに対応しています。1バイトは8ビットなので2の8乗パターンの組み合わせ、つまり0～255の256パターンが存在します。アルファベットや数字、改行などのいくつかの特殊記号だけであれば256パターンもあれば表現できます。

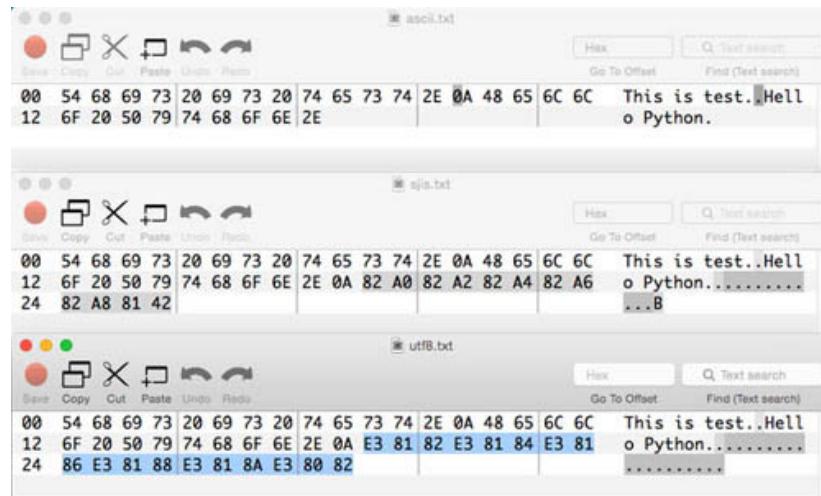
ただ、よく考えてみてください。日本語はどう考えても1バイト=256個じゃ足りません。そこで日本語を扱うときは複数バイトを使います。2バイトにするだけでも65536(256^2)パターン、3バイトにすれば16777216(256^3)パターンの組み合わせが表現できます。

この複数バイトの01と文字のマッピングをする文字コードにはいくつかの種類があります。日本で有名なのは、Shift-JISやUTF-8、EUC-JPあたりでしょうか。文字コードが違えば、01に変換したデータも変わってきます。

以下のテキストがあるとします。

This is test.
Hello Python.
あいうえお。

文字コードとバイト配列を比較するためにこれをASCII、Shift-JIS(SJIS)、UTF-8でファイルに書き込み、それをバイナリエディタを使って01(実際は16進数)で見てみます。なお、日本語あいうえおはASCIIにはそもそも対応していないので、ASCIIの例はアルファベットのみから構成されています。上からASCII、SJIS、UTF-8という順です。



英語の部分は変わっていませんが、強調している日本語の部分は文字コードごとに違っているのがわかります。

ここからわかることは「どの文字コードで書かれているか」ということがわからないと、エディタやPythonは適切に文字を扱うことができないということです。たとえば、SJISのファイルをUTF-8として読み込もうとすればコードを解釈できずに文字化けが発生したりエラーとなったりします。当然、ひとつのファイルのなかでさまざまな文字コードを織り交ぜるということはできません。ファイルのなかで利用する文字コードは必ず統一して下さい。文字コードがどのようなものか、正しく文字コードを認識できることがいかに大切かということを理解してもらえたなら幸いです。なお、自分で文字コードの変更を試されたい場合は文字コードを変更できるエディタやnkfコマンドなどを利用すればよいと思います。

余談ですが、もしどの文字コードを使っててもよいのであれば、UTF-8が今だと一般的かもしません。10年前だとShift-JISだとかEUC-JPあ

たりも見たのですが、今のプログラマはあまり好んで使わない傾向があります。私は制約がない限り、すべてUTF-8でコードもドキュメントも統一するようにしています。実際、次の章でも説明しますが python3 はUTF-8 をデフォルトの文字コードにしています。可能であればUTF-8 でコードを書くようにして下さい。

バイト配列

文字コードだけでなく様々なデータは全て「0,1 の羅列」で表現されています。この0,1 の羅列のことをバイト列といい、Python はそれを bytes という型で扱います。バイト列は文字列やリストと似たようなシーケンスとして処理ができますが、C 言語などと違ってPython でバイト配列を直接扱う機会は多くないと思います。ただ、プログラミングの基礎概念として知っておくべき内容であるため、簡単に説明します。バイト列の宣言は文字列とほぼ同じように行いますが、シングルクオテーション等の前に b と宣言します。例えば以下のようにになります。

```
b = b'hello world'  
print(type(b))  
# <class 'bytes'>
```

上記例では 'hello world' を文字列としてではなく、バイト列として扱います。そのため、変数 b は文字列型ではなくバイト型のオブジェクトを持っています。バイト型のオブジェクトも当然ながらメソッドを持っており、hex を使うとバイト列を 16 進数の文字列で返します。

```
print(b.hex())  
# 68656c6c6f20776f726c64
```

68 -> h, 65 -> e, 6c -> l, 6c -> l というように Ascii 表に沿って、上記の16進数をアルファベットに変換していくと hello world になります。16進数の文字列をbyte 型に変換するには bytes クラスの fromhex メソッドを使います。

```
print(bytes.fromhex('68656c6c6f20776f726c64'))  
# b'hello world'
```

文字列とバイト列の変換は encode ⚡ decode メソッドを使います。

```
print('hello world'.encode())  
# b'hello world'  
  
print(b'hello world'.decode())  
# hello world
```

文字列に対してメソッド encode() を呼び出すとバイト列を返し、バイト列に対して decode メソッドを呼び出すと文字列を返します。

日本語の扱い

そもそもPythonのプログラムファイルもプログラムが書かれたテキストファイルです。そのため、それがどの文字コードを利用しているかということを「プログラムを読み込むPython」に正しく認識させる必要があります。ブラウザやテキストエディタが文字化けしてしまうのと同じように、Pythonも文字コードが分からないと内部に書かれているプログラムを解釈できずに実行できません。

Python3はUTF-8をデフォルトの文字コードとしているため、UTF-8以外を使う場合は「この文字コードを使います」とファイルの先頭で宣言します。英数字のみで構成されるASCIIも宣言は不要です。その宣言は以下のように行います。なお、本書執筆時点においてIDLEでの日本語の扱いは正直あまりよいとはいえません。もし日本語を使いたいのであればなんらかの高機能なテキストエディタを利用して記述し、適切な文字コードに設定されたプロンプトなり、ターミナルなりで実行してください。

```
# coding: utf-8

print('hello python')
print('あいうえお python')
```

上記の「utf-8」と書かれている場所が文字コードの宣言です。utf-8と宣言しているので、utf-8以外で書かれているとトラブルが発生します。ここを「shift-jis」や「euc-jp」などに変えると、その文字コードとして解釈されます。宣言の先頭が#から始まっていることから理解してもらえると思いますが、この行はコメントアウトされているのでPythonはプログラムとしては解釈しません。なお、先にも述べたようにUTF-8はデフォルトであるため、あえて宣言する必要はありません。

このUTF-8で書かれたプログラムを実行すると以下のように表示されます。

```
$ python3 test.py
hello python
あいうえお python
```

試しにファイルで利用される文字コードはUTF-8のままで、宣言をshift-jisに変更してみます。

```
# coding: shift-jis

print('hello python')
print('あいうえお python')
```

これを実行してみます。

```
$ python3 test.py
File "test.py", line 1
SyntaxError: encoding problem: shift-jis
```

エラーが出て、「shift-jisのエンコーディングの問題がある」と怒られています。このように文字コードAを文字コードBとして読み込もうとするとトラブルが発生してしまいます。トラブルはこのようなエラーであったり、場合によっては文字化けだったりします。

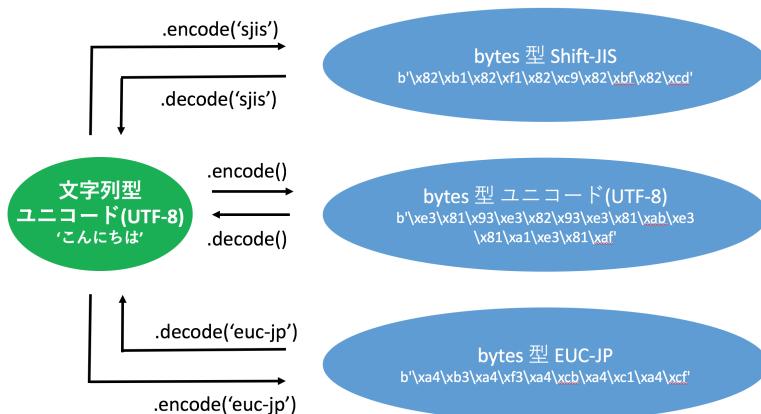
ではファイルの文字コードをShift-JISに変更して実行してみます。これでファイルの文字コードと冒頭の宣言の文字コードが一致します。

```
$ python3 test.py
hello python
あいうえお python
```

実はPython3はプログラムファイルの文字コードとその宣言の指定さえ間違えなければ、日本語などのマルチバイト文字をほとんど意識せずにアルファベットと同じ感覚で使えます。Python2では文字列型の亞種である「Unicode文字列型」を使うことで日本語を扱っていたのに比べると随分と簡単になりました。

文字コードの変換

先ほど見たように文字コードもバイト列で作られています。例えばUTF-8の文字列をShift-JISに変換するといった場合、このバイト列の処理が必要になります。実際はPythonのライブラリ任せになるのでそれほど難しくありませんが、おおまかに以下の図のような文字列とバイト列の関係の認識を持ってもらうと分かりやすいです。



まずPythonは文字列としてUnicode(UTF-8)を持ちます。一方、バイト列は単なる0,1の組み合わせですのでどのような文字コードでも持つことができます。先程説明した文字列のencode及びバイト列のdecodeメソッドを、文字コードを引数として与えて呼び出して変換ができます。文字列からShift-JISの変換は簡単で、単にencodeする際に引数にShift-JISを指定してShift-JISのバイト列に変換することができます。Shift-JISからEUC-JPへの変換も、Shift-JISを引数shift-jisでdecodeし、Unicodeの文字列にし、それから引数euc-jpでencodeしてあげればEUC-JPのバイト列が得られます。上記変換のサンプルを以下に記載します。

```
text = 'こんにちは'
unicode_bytes = text.encode()
```

```

print(unicode_bytes)
# b'\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\x81\x81\xaf'
print(unicode_bytes.decode())
# こんにちは

sjis_bytes = text.encode('sjis')
print(sjis_bytes)
# b'\x82\xb1\x82\xf1\x82\x9\x82\xbf\x82\xcd'
print(sjis_bytes.decode('sjis'))
# こんにちは

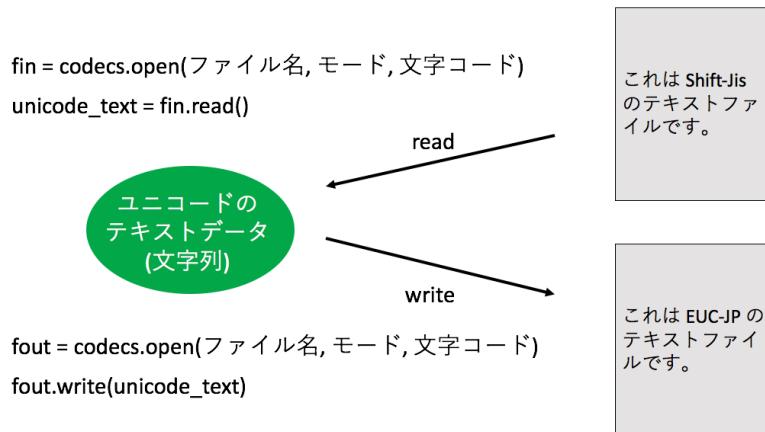
eucjp_bytes = text.encode('euc-jp')
print(eucjp_bytes)
# b'\xa4\xb3\x4\xf3\x4\xcb\x4\xc1\x4\xcf'
print(eucjp_bytes.decode('euc-jp'))
# こんにちは

```

文字列はユニコードであり、エンコードを指定してバイト列を書き出すことで様々な文字コードを表現できる。それらをエンコードの指定をして読み込むことでPythonのユニコード文字列に変換することができる。この流れは日本語以外にもありますので覚えておいて下さい。

日本語ファイルの読み書き

ファイルの入出力にはcodecs/パッケージのopen関数を使うのが簡単です。これを使って文字コードを指定してファイルをオープンすると、通常のファイル入出力の手順と大差なくマルチバイト文字を扱えます。具体的にはreadをすると文字コードを意識してファイルのテキストを読み込んでユニコードの文字列を返し、writeでユニコードの文字列を書き出すとそれが指定された文字コードでファイルに書かれます。以下の図にこの流れを記載します。



上記の「ファイル名」は読み書きするファイル名を相対パスなり絶対パスなりで指定し、「モード」は通常のopen関数と同じでr,w,rw,aなどを指定します。そして最後の「文字コード」で読み書きするファイルのエンコードを指定します。このopen関数のみ文字コードを意識する必要がありますが、それ以外のreadやwriteは今までのファイルの読み書きと同じです。Shift-Jisとして読み込みを行えば、readをすればShift-Jisとしてファイルの中身を読み込んで、それをユニコードの文字列として返します。書き出しも同様です。

実際にコードで確認してみます。文字コードutf8で書かれた以下のファイルutf8.txtを作成してください。

```

あいうえお
abcde
かきくけこ

```

このファイルを読み込み処理するコードは以下となります。

```

import codecs
f = codecs.open('utf8.txt', 'r', 'utf-8')
for line in f:
    print(line, end='')
f.close()

```

先程説明したようにcodecs.open関数の第一引数でファイル名を指定し、第二引数でオープンのモード(今回はread)を選択、そして第三の引数でファイルの文字コードを指定しています。それ以外は通常のファイル読み込みと同じです。これを出力してみます。

```

$ python3 test.py
あいうえお
abcde
かきくけこ

```

変数lineに格納されているファイルから読み込まれたデータは通常の文字列型です。読み込んだあとは文字コードを気にする必要は一切ありません。次に書き込みをしてみます。このutf8のファイルをShift-JISで書きだしてみます。

```

import codecs
fin = codecs.open('utf8.txt', 'r', 'utf-8')
fout = codecs.open('sjis.txt', 'w', 'sjis')
for line in fin:
    fout.write(line)
fin.close()
fout.close()

```

ファイルのオープン時にオープンモードをwにすることで書き込みファイルとして開いています。オープンしたファイルに対してUnicode文字列をwriteしてあげればファイルに文字列が追加されます。

結果を確認してみます。ファイルのエンコーディングの判定をしてみます。

```

$ nkf --guess utf8.txt
UTF-8 (LF)

```

```
$ nkf --guess sjis.txt  
Shift_JIS (LF)
```

書き込みファイルsjis.txtの文字コードがShift-Jisと判定されています。UTF8のファイルを読み取り、それを解釈、Shift-Jisとして書き込むという動作がうまく動いています。なお上記のnkfコマンドはインストールをしないと使えないで注意してください。

プログラムへの入力

WindowsやMacのGUIアプリケーションを使うときに、さまざまな入力を求められることはあります。書き出すファイル名の入力や、利用するサーバを一覧から選択するといった場合などです。このようにプログラムがユーザーになんらかの入力を求めるることは一般的です。本章では今までのようにテキストベースのプログラムで「ユーザーからプログラムへの入力」を扱う方法についてお話しします。

ユーザーからの入力の方法にはいくつかありますが、ここではプログラムの起動時に指定する「コマンドライン引数」と、プログラム内でユーザーに入力を求める「標準入力」、そして標準入力をを使ったインタラクティブなプログラムの書き方についてお話しします。

コマンドライン引数の使い方についてお話しする前に、それがなぜ必要なのか説明しておきます。まず以下のような「一人の生徒の成績表を表示するプログラム」のshow_score_sheet.pyがあるとします。

```
student = 'taro'
score_sheet = get_score_sheet(student) # get_score_sheetは実装済みとする
print(score_sheet)
```

上記の例では'taro'の成績を取得しています。では、'taro'の代わりに'jiro'の成績を取得したいと思ったらどのようにすればよいと思いませんか。成績取得をする関数に与える生徒名の'taro'を'jiro'にしてやればよいです。

```
student = 'jiro' # 生徒の名前を変更
score_sheet = get_score_sheet(student)
print(score_sheet)
```

ただ、40人の生徒の成績を表示したい場合、プログラムのファイルを40回開いて、その都度生徒の名前を変更してプログラムを動かすのは正直面倒くさいです。それにそもそも「プログラムを開いてそれを修正する」などということは、そのプログラムの開発者でなければできません。要するにプログラムファイルをいちいち書き変えるという対応での解決策はなしということです。

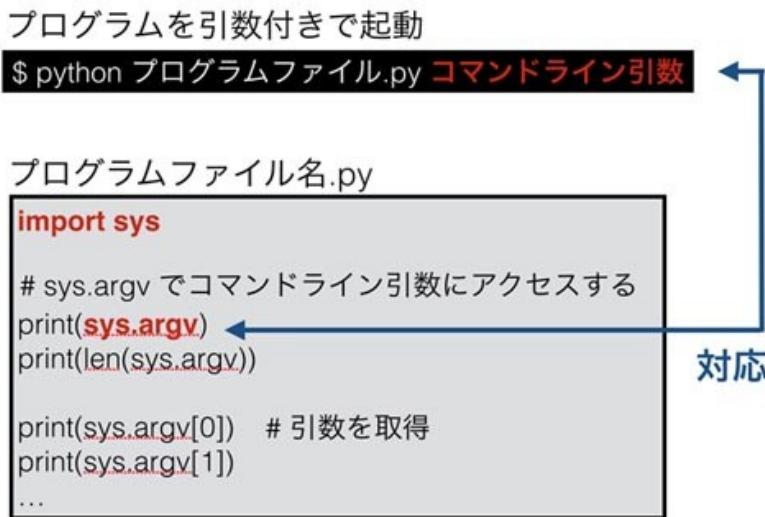
コマンドライン引数

解決方法はいろいろ考えられますが、最も一般的なものは、コマンドライン引数を利用することです。コマンドライン引数とは、Pythonコマンドでプログラムを起動する際に与えるパラメータのことです。たとえば以下のようなものになります。

```
python3 show_score_sheet.py taro
```

注目して欲しいのは3番目のキーワードのtaroです。今までPythonプログラムを実行するときは"python ファイル名"だけでしたが、そこにtaroが追加されています。このtaroという与えられたキーワードを、Pythonのプログラムが内部で利用することで、生徒の名前をプログラム中に直接書き込まなくても指定した生徒の成績を取得できるようになります。今回は生徒名ですが、プログラムによってはファイル名であったり接続するサーバ名であったり、いろいろな利用方法があります。

コマンドライン引数の使い方は以下の図のようになります。



直感的に動きをつかんでもらえればよいのですが、起動時に与えたコマンドライン引数を、プログラム中のsys.argvでアクセスしています。図の細かい説明をするより、実際にコマンドライン引数を利用するプログラムを見たほうが早そうなので、以下に記載します。

```
# sysモジュールをimport
import sys

# sys.argvにコマンドライン引数が「リスト」で格納されている
print(sys.argv)
print(len(sys.argv))
```

これを実行すると以下のようになります。

```
% python3 test.py taro
['test.py', 'taro']
2

% python3 test.py taro jiro
['test.py', 'taro', 'jiro']
```

sys.argvをprintしているのでわかると思いますが、これは「リスト」です。そのリストの中の最初の要素は必ずPythonの実行プログラムとなります。今回は同じディレクトリのプログラムを相対パスで呼び出したのでファイル名だけですが、絶対パスなどで呼び出すと要素も絶対パスとなります。

2番目以降の要素はコマンドライン引数に与えられた入力値と対応します。上記例を見てもらうとわかりますが、引数のn番目がsys.argvのn+1番目の要素になっています。sys.argvはリストですので、その長さはlen()関数で取得できます。

このコマンドライン引数を先ほどの生徒の成績を取得するプログラムに組むことはそれほど難しくありません。

```
import sys
# 誤った入力値の場合はメッセージとともにプログラム中断
if(len(sys.argv) < 2):
    print('usage: student.py student_name')
    exit()

# sys.argvよりもわかりやすい変数名で代入して使う
student = sys.argv[1]
score_sheet = get_score_sheet(student)
print(score_sheet)
```

コマンドライン引数の長さを調べて、2未満であれば使い方を表示して終了するようにしています。無言で終了するのも何が原因なのかプログラムの利用者にわからないのでやめたほうがよいです。コマンドライン引数に指定された内容は、プログラムの途中で都度チェックするよりも、このように最初に調べてしまって問題があれば終了するとしたほうがきれいにコードを書けるかもしれません。必要になった場所でチェックをするという実装だと、プログラムのコアとなるロジックに余計なものが食い込み、汚くわかりにくいコードになりがちなので気をつけてください。

その後はコマンドライン引数の値を「わかりやすい名前の変数」に格納しています。sys.argvの何番目という表現を延々とプログラム中で使い続けるとわかりにくく、なおかつ引数の順番を変えたときなどの修正が面倒になるため避けたほうがいいです。それ以降のコードは先ほどとまったく一緒ですので解説は割愛します。

なお、UnixやLinuxコマンドの「オプション(-v や --help など)」相当のことを実装したいのであれば、sys.argvを使って根性で作りこむよりも専用のパッケージ「argparse」などを利用したほうがよいかと思います。

標準入力

ユーザーがプログラムに入力を与えるのはコマンドライン引数だけではありません。標準入力も用います。先ほどの生徒の成績を取得するプログラムを例に、標準入力がどのようなものか説明します。

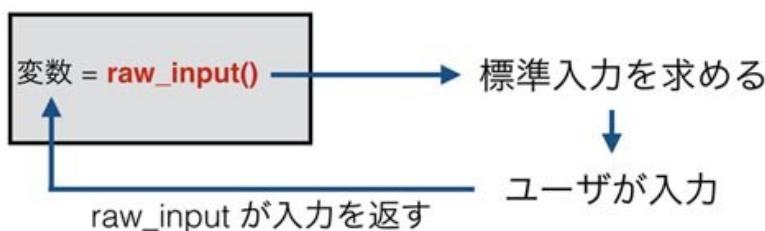
コマンドライン引数はプログラムの「起動時」に入力値を指定しますが、標準入力はプログラムの「起動後」に入力値を与えるものです。さっそくですが、生徒の成績表示プログラムを標準入力のものに書き換えてみます。

```
print('please input student name.')
# 標準入力
student = input()
print('your input is: ' + student)
```

1~2行目が変更されています。1行目は入力を促すテキストを出しているだけなのでたいしたことはないのですが、重要なのは2行目です。これは以下のように動いています。

1. input()関数が実行される
2. pythonはユーザーからのキーボード入力を待つ
3. ユーザーがキーボードでテキストを入力し、Enter(Return)を打つ
4. pythonがユーザーからの入力を読み取り、input()関数がそれを文字列として返す
5. 変数 studentが返された文字列を格納

長々と書きましたが、要するにinput()を書いた場所で「ユーザー入力」が求められて、ユーザーが入力した値がinput()から返されるということです。図にまとめると以下のようになります。



上記プログラムを実行すると次のようにになります。

```
python test.py
please input student name.
taro
your input is: taro
```

なお、この標準入力を読み取る関数ですがpython2の頃は別名でraw_input()関数でした。間違えないように注意してください。

ほかにはsysモジュールのreadline関数も同じ目的で利用できます。詳しくは書きませんが、以下のように使うことができます。

```
>>> import sys
>>> line = sys.stdin.readline()
hello
>>> print(line)
hello
>>> line = input()
hello
>>> print(line)
hello
>>>
```

ほとんどinput()と同じですが、着目して欲しいのは改行コード“Enter(Return)”も取得されているということです。上記サンプルを見ると、前者のreadline()は改行コードを含んでいますが、後者のinput()は省かれています。そのため、readline()を使う場合、必要であれば“文字

列.strip()などとして行末の改行コードを削ってください。

なお、かなりの小ネタですが、input()などの標準入力を挟むことで、指示があるまでプログラムをわざと中断させておくという使い方もあります。たとえばデモプログラムを実行する際に、デモとデモの間にinput()を入れておくと、ひとつめのデモが終わったあとに、すぐに2つめに入らざる入力で待ちに入ることができます。意外と便利な使い方です。

さて、ちょっと余談です。コマンドライン引数と標準入力の2つのユーザー入力方法を示しましたが、利用するとしたらどちらが優れているでしょうか。これは個人の好みによるとは思いますが、どちらでもよい場合、私は「Unixのコマンドの思想に沿っている」という点からコマンドライン引数を支持します。ここではUnixとしていますが、この思想は、LinuxはおろかWindowsであってもCUI(テキストのコンソール)を使う限りあてはまる大事な考え方です。当然、Pythonで作成したプログラムも基本はテキストベースなので当てはまります。Unixの思想にはいろいろあるのですが、「入力」に関しては以下が当てはまるでしょう。

- 一つひとつのコマンド(プログラム)が小さい範囲で完璧に仕事をこなすべき
- 大きなプログラムになんでもやらせるのは基本的に間違い
- 小さいプログラムを組み合わせることで大きな仕事を実現する

ここで重要なのは「小さいプログラムを組み合わせる」ことです。組み合わせる際に「あるプログラムから別のプログラムを呼び出す」ということをよくするのですが、この場合、標準入力よりもコマンドライン引数のほうが圧倒的に簡単に使えます。標準入力を使ってしまうと、プログラムが別のプログラムを「インタラクティブ(この出力がされたら、これを入力するといった具合)」に制御する必要があります。人間がプログラムとインタラクティブに対話するのは簡単ですが、プログラムにそれをやらせるのは結構苦労することが多いです。一方、コマンドライン引数だと、プログラムを呼び出す際にパラメータを与えるだけで簡単に期待どおりの使い方をすることが可能です。なので私は、コマンドライン引数のほうを好んで使っています。

ただ、標準入力が有効な場面というのも存在します。たとえば以下の場合は。

- ユーザーがインタラクティブに操作するプログラムを組む場合
- コマンドの履歴を残したくない場合(例えばパスワード入力など)

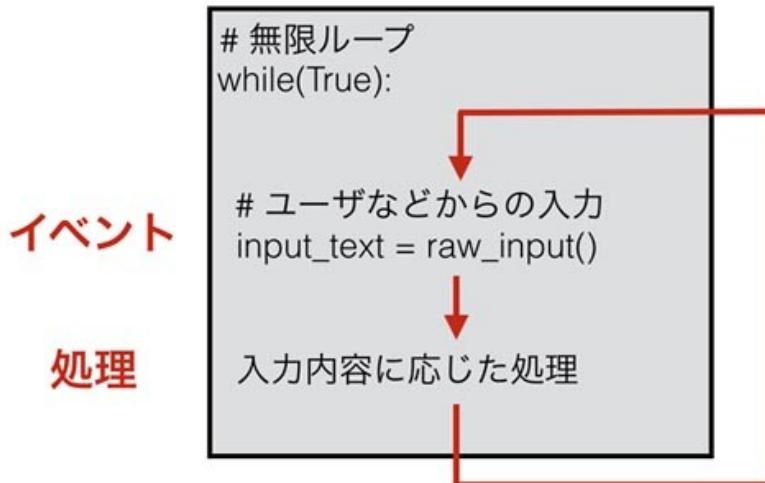
適切なものは時と場合によりますので、どちらが最適なのかよく考えて利用してください。なお、パスワード入力に関してはgetpassというモジュールもありますので、パスワード入力を求める場合は、それを使ってもよいでしょう。

インタラクティブなプログラムの作成

今までのプログラムは「ユーザーがプログラムを起動したら処理を実行し、それが終われば終了」というものでした。ただ、なかにはこれにそぐわないプログラムもあります。たとえばGUIのアプリケーションを想像してください。だいたいはボタンを押したりテキスト入力をしたりして使い続けて、必要がなくなった時点でウィンドウを閉じるなどして終了します。これは「インタラクティブなプログラム」と呼ばれており、以下の処理を繰り返すことで実現されています。

1. ユーザーからの入力をアプリケーションが待つ
2. ユーザーからの入力に応じてアプリケーションがなんらかの処理を行う
3. 処理が終わると1に戻る

これと同じことは、CLIのコンソールでもできます。その一番簡単な仕組みは以下の図のようなものとなります。



これも実例を用いて説明したほうがはやうなので、簡単なサンプルプログラムを使います。以下のサンプルでは「あるプログラムの設定ファイルを書き出すプログラム」を作成します。設定ファイルは以下のようなものとします。

```
username = taro
password = my_password
server = 10.0.0.1
```

プログラムの流れは以下のようになります。

1. 最初に入力可能なオプションを示し、input()で待機
2. ユーザーが入力
3. ユーザー入力を読み取り、適切な入力であればそれを設定。不適切であればエラー表示
4. exitと入力されれば終了し、内容をファイルに書き出す(終了条件)

これをプログラムにすると以下のようなものとなります。

```

username = ''
password = ''
server = ''

while(True): # 無限ループ
    print('please input option and its value.')
u USER_NAME
p PASSWORD
s SERVER_IP
exit''')

line = input() # ユーザーからの入力を取得

if(line == 'exit'): # 無限ループから離脱する条件
    break

words = line.split() # ユーザーからの入力内容をチェック
if(len(words) != 2):
    print('Error')
    continue

if(words[0] == 'u'):
    username = words[1]
elif(words[0] == 'p'):
    password = words[1]
elif(words[0] == 's'):
    server = words[1]
else:
    print('Error')
# ループ終わり

print('username = ' + username)
print('password = ' + password)
print('server = ' + server)

```

今回は書き出す代わりにprint出力させています。先に提示した処理手順と完全に同じではありませんが、ユーザーが入力した内容に応じて処理を行うということを繰り返します。

実際のプログラムでは処理をマルチスレッド(複数の処理を別々のタイムラインで実行)などとすることもありますが、この「入力 -> 処理 -> 入力 ->」という処理の流れは非常に重要なので覚えておいてください。バッチ処理以外のGUI(ボタンなどが押される -> なんらかのアクション)や、サーバ(ネットワーク越しにクライアントの要求を受け取る -> アクション)のプログラミングも基本的にはこの流れとなり、このような処理方式を「イベントドリブン」と呼びます。

このイベントドリブン型のプログラミングは、ある程度プログラムが書けるようになると頻繁に使うロジックになるはずです。ただ実際に利用するフレームワーク(自分のプログラムを呼び出す親分プログラムみたいなもの)などに隠蔽されてたりしているので、あまり意識しないことが多いかもしれません。

関数の高度な使い方

本章では関数の高度なトピックについて扱います。正直なところ本章の内容は初心者にはかなり難しいのですが、Pythonの初心者であっても既存ライブラリなどを利用する場面は多くあるため、「こういう機能がある」というレベルでは知っておいたほうがいいです。本トピックは下編で扱うか悩んだのですが、難易度が高いわりに利用する頻度が高いためここで扱わせていただきます。難しければ斜め読みしていただき、ある程度 Python のプログラムが書けるようになってから読み直してみて下さい。

関数内の関数

Python の関数は実は関数の中に入れ子構造にすることができます。例えば以下のような使い方ができます。

```
def test1():
    a = 10

    def test2(x, y):
        return x + y

    return a + test2(5, 6)

print(test1())
# 21
```

上記の例では関数 test1() の中に関数 test2() を定義し、それを使っています。今回のサンプルのような短い関数であればこのような使い方は不要ですが、より複雑な処理を関数で行う場合は処理を綺麗に分類するために内部で関数を定義して使うことがあります。

関数オブジェクト

先程もお伝えしましたがPython は関数自体もオブジェクトです。そのため、変数に関数を代入し、それを利用することができます。

```
def test1():
    print('test1')

a = test1
a()
# test1
```

変数に代入できるということは関数の引数として使えるということです。詳細は中編にて取り扱いますが、関数を使う関数である高階関数というものもあります。

```
def apply_list(fun, list_):
    for i in list_:
        fun(i)

def print_double(x):
    print(x * 2)

list_ = [1,2,3,4,5]
apply_list(print_double, list_)
# 2
# 4
# 6
# 8
# 10
```

位置引数とキーワード引数

関数の引数の使い方については既に何度も説明しています。例えば以下の関数があるとしましょう。

```
def test(a, b, c):
    print(a)
    print(b)
    print(c)
```

この関数は引数として a, b, c を受け取ります。この関数を呼び出す際に a, b, c に対応した順序で引数を与えます。

```
test(1,2,3)
# 1
# 2
# 3
```

出力みると分かりますが呼び出しの引数の順序と関数の定義の引数の順序が合致していることが分かります。このように関数の引数をその位置を使って指定することから、このような引数の使い方を「位置引数」と呼んでいます。

実は関数の引数は位置引数以外に「キーワード引数」と呼ばれる使い方ができます。具体的には関数を呼び出す際に仮引数(定義された引数名)に対して実引数を与えるとキーワード引数として使われます。具体的には以下の様な使い方です。

```
def test(a, b, c):
    print(a)
    print(b)
    print(c)

test(b='B', c='C', a='A')
# A
# B
# C
```

関数の定義自体は最初の例と全く代わりませんが、呼び出し時に「b='B'」といった形で値を与えています。指定する引数の順番としては b, c, a なのですが、呼び出し側では定義された順序の a, b, c で解釈されていることが分かります。位置で引数を指定するのではなく、引数名という「キーワード」で引数を指定するのでキーワード引数と呼ばれています。今回のようなシンプルな関数の定義だとキーワード引数で呼び出す必要はありませんが、次に説明する「デフォルト引数を使う関数」や「キーワード引数を使うことを前提に定義された関数」を使う関数を使う場合に利用されます。

デフォルト引数

デフォルト引数は引数定義時にデフォルトの値を決めてしまい、呼び出し時に対応する引数が与えられなければデフォルト値が適用されるというものです。デフォルト引数は関数定義の引数の箇所に「引数名 = 値」と定義することで実現できます。以下にサンプルを記載します。

```
def test(a, b='B', c='C'):
    print(a)
    print(b)
    print(c)
```

引数 a,b,c を見ると、b と c にはデフォルト引数が定義されていることが分かります。この関数を呼び出すときに引数を3つ与えればそれ a, b, c に与えられますが、それ未満の数で呼び出すと足りないぶんはデフォルト値が使われます。以下にこの関数の利用例を示します。

```
test(1,2,3)
# 1
# 2
# 3

test(1,2)
# 1
# 2
# C

test(1)
# 1
# B
# C
```

出力を見ると引数が足りない場合はデフォルト値が使われていることが分かります。デフォルト値を使う場合に気をつけてほしいことは、デフォルト引数は必ず引数の後部でなければならないということです。例えば以下のようないい方ではありません。

```
def test(a='A', b='B', c):
    print(a)
    print(b)
    print(c)

#     def test(a='A', b='B', c):
# #SyntaxError: non-default argument follows default argument
```

大量のデフォルト引数を持つ関数はキーワード引数との相性がいいです。例えば以下の例を見て下さい。

```
def test(a, b='B', c='C', d='D', e='E', f='F'):
    print(a)
    print(b)
    print(c)
    print(d)
    print(e)
    print(f)

test('AA', d='DD')
# AA
# B
# C
# DD
# E
# F
```

上記サンプルの関数は非常に高度なもので、引数が6つとれるとしましょう。ただ、細かい調整を引数で調整できるだけであり、最初の引数以外はそれほど重要でないとなります。全ての引数を指定するのは大変ですが、デフォルト引数の機能を使って必要な箇所のみピンポイントで指定し、あとはデフォルト値を使うようにすれば呼び出しが簡単になります。

可変長引数

可変長引数は名前の通り「引数の長さ(数)が変わる」ということです。先ほどのキーワード引数も良い出典の引数の数はデフォルト値を使うか否かという観点では可変でした。ただ、可変長引数はそれとは異なり「好きなだけ」関数の定義をすることができます。可変長引数を使う際は、関数を定義する際に引数の前に「*」をつけます。引数名はなんでもよいのですが、慣習的に args (日本語だと引数)といった名前を付けることが多いです。読む人にとって「これは可変長引数」と認識しやすいです。

```
def test(*args):
    print(type(args))
    print(args)

test(1)
# <class 'tuple'>
# (1,)

test(1,2,3)
# <class 'tuple'>
# (1, 2, 3)

test()
# <class 'tuple'>
# ()
```

上記のサンプルを見てもらうと分かるように可変長引数はタプルとして渡されています。呼び出し時にあたえられる引数の数は任意の数が使え、0 であっても問題ありません。

定義する可変長引数の位置を調整することで必要最低限の引数の数が決まります。例えば def test(a, *args): とした場合、最初に与えられた引数が a に入り、のこりが args に入れます。そのため、最低一つの引数が必要です。

```
def test(a, *args):
    print(a)
    print(args)

test(1)
# 1
# ()

test(1,2,3)
# 1
# (2, 3)

test()
```

```
# Traceback (most recent call last):
# ...
# TypeError: test() missing 1 required positional argument: 'a'
```

2つの引数が必要であれば `def test(a, b, *args):` とするなどして、必要な数にマッチした関数の定義をしてください。

なんでも受け取るキーワード引数

位置指定引数のところで紹介したキーワード引数は関数の引数の順序を無視して呼び出す目的で使いました。実はこのキーワード引数は自分が定義した引数以外にも使えます。以下のサンプルで関数定義の際に `**kargs` という引数を使っています。これを使うと定義されてないキーワード引数を辞書型として扱うことができます。

```
def test(**kargs):
    print(type(kargs))
    print(kargs)

test(a='A', hello=3)
# <class 'dict'>
# {'hello': 3, 'a': 'A'}

test()
# <class 'dict'>
# {}

test('A', 'B')
# Traceback (most recent call last):
# ...
# TypeError: test() takes 0 positional arguments but 2 were given
```

上記の呼び出し結果と出力を見てもう分りますが、キーワード引数の引数名が辞書型の `key` になり、それに対応する値が `value` となっています。`**kargs` はあくまでもキーワード引数専用であるため、キーワードを指定しない位置引数を使った呼び出しをするとエラーとなっています。なお、`kargs` は keyword args の省略形であり、先ほどの `args` と同様に慣習としてこの名前を使うことが多いです。他の変数名も使えますが、あえて使う理由はありません。

かなり高度な使い方となるのですが、可変長引数とキーワード引数を併用することで「どんな引数でも使える関数」を定義することができます。

```
def test(*args, **kargs):
    print(args)
    print(kargs)

test(1, 2, 3, a='A', b='B', c='C')
# (1, 2, 3)
# {'a': 'A', 'b': 'B', 'c': 'C'}
```

この使い方で便利なのは関数を「呼び出す」場合にも `*args` と `**kargs` を使えるということです。それをする「受け取った引数をそのまま別の関数に渡す」ということができます。以下のサンプルを見ると `test2` 関数が受け取った引数を `test1` 関数に丸投げしていることがわかります。

```
def test1(*args, **kargs):
    print('test1')
    print(args)
    print(kargs)

def test2(*args, **kargs):
    print('test2')
    # 可変長引数とキーワード引数をそのまま呼び出す関数に渡す
    test1(*args, **kargs)

test2(1, 2, 3, a='A', b='B', c='C')
# test2
# test1
# (1, 2, 3)
# {'a': 'A', 'b': 'B', 'c': 'C'}
```

今回は自分で作った `test1` を呼び出していますが、たとえばここに既存の関数をいれることもできます。

```
def test(*args, **kargs):
    print('args:', args)
    print('kargs:', kargs)
    result = sum(*args, **kargs)
    print('result:', result)
    return result

result = test([1, 2, 3, 4, 5])
# sum([1, 2, 3, 4, 5])
print(result)
# args: ([1, 2, 3, 4, 5],)
# kargs: {}
# result: 15
# 15
```

上記のサンプルは `sum` 関数になにを与えるかをリッチに出力させています。面白いのは関数 `sum` はキーワード引数を受け取らないのに呼び出し時にそれを与えても問題が発生していないところかと思います。

再帰関数

再帰関数は自分自身を呼び出す関数です。`for` や `while` 文でループ処理をしますが、再帰関数も似たように「同じ処理を何度も繰り返す」場面で使われます。たとえば配列から一番大きな要素を取得する処理は以下のように書けます。

```
def get_max(list_, max_):
    # リスト長が0なら最大値を返す
    if(len(list_) == 0):
        return max_

    # リストから値を取り出し最大値の更新
    value = list_.pop()
    if(value > max_):
        max_ = value

    # 次のリストの要素をチェック。
    return get_max(list_, max_)
```

```

list_ = [5,9,10,3,5]
max_ = get_max(list_, 0)
print(max_)
# 10

```

コメントを読んでもらうと何をやっているか分かるかと思いますが、ようするにリストから1つの要素を取り出して、それが現在の最大値より大きければ最大値を更新する作業を繰り返し実行しています。関数が同じ関数をどんどん呼んでいき、深くまで戻っていくようなイメージです。最終的に探索し終えたら、最大値を return 文で返し、深くもぐった関数呼び出しを今度は上に戻っていき、最初の get_max 関数の呼び出しもとに値を返します。

はっきり言うとこのコードは悪い再帰関数です。再帰関数としては分かりやすいのでとりあげたのですが、同じことを for 文で実現した以下のコードのほうがはるかに分かりやすいです。

```

list_ = [5,9,10,3,5]
max_ = 0
for i in list_:
    if i > max_:
        max_ = i
print(max_)
# 10

```

このコードがどういう処理をしているかについてあえて説明する必要はないと思います。

実はこの再帰関数なのですが、ループ文にはない特徴があります。それは繰り返しではなく「木構造の呼び出し」に向いているということです。本書の冒頭で説明した「あるディレクトリ配下を書き出す」という再帰の例を思い出して下さい。それは以下のようないい處です。

```

import os

def list_file(path, indent_level):
    # ディレクトリ名を表示
    print('{0}{1}'.format(' ' * indent_level, path))

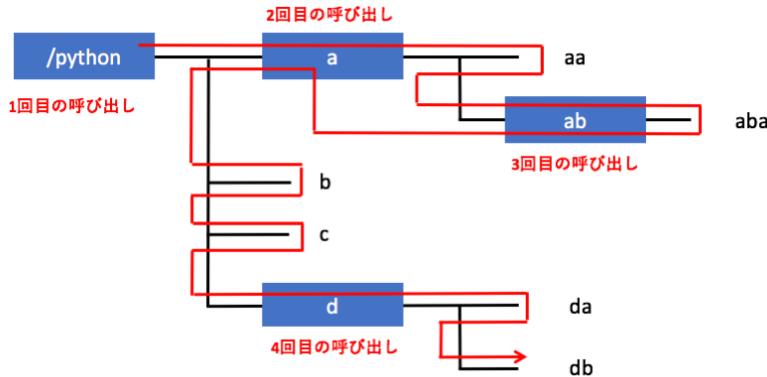
    # ディレクトリ内のファイルとディレクトリを全てループで確認
    for file_name in os.listdir(path):
        if(file_name.startswith('.')): continue
        abs_filepath = path + '/' + file_name
        if(os.path.isdir(abs_filepath)):
            # ディレクトリだったので、そのディレクトリをチェックする
            list_file(abs_filepath, indent_level + 1)
        else:
            # ファイルだったので、ファイル名を表示
            print('{0}-{1}'.format(' ' * indent_level, file_name))

list_file('/python', 0)

```

先程の最大値を得る再帰関数と同様に関数 list_file 内で関数 list_file を呼び出しています。ただ、両者の大きな違いは list_file 内での list_file 関数の呼び出しは必ずしも1回ではなく、状況に応じて任意の数に変わることです。そして呼び出された毎の各関数はそれぞれ状態を維持し続けています。

たとえば以下のディレクトリ構造に対してこのプログラムを走らせると、以下のような呼び出しかたをします。



そして出力は以下のようなものとなります。

```

[ /python ]
[ /python/a ]
- aa
[ /python/a/ab ]
- aba
- b
- c
[ /python/d ]
- da
- db

```

着目してほしいのはループと違って同じ関数を呼び出した時点で、その関数の仕事が終了していないという点です。たとえばディレクトリ /python に対して関数が呼ばれた際に、その子要素のディレクトリ a に対して関数を呼び出したあとも、ファイル b, c を表示したり、さらに別のディレクトリ d を呼び出したりしています。これは各関数の呼び出しがそれぞれに状態を持っているため簡単に実現できます。一方、同じことをループ文でやろうとすると「3周目の処理がおわって2周目の処理を継続」ということを実現するのに一苦労します。

単純にグルグル回す場合はループを使い、複雑な木構造のような処理をしないといけない場合は再帰関数を使うというのが一般的な使い分けとなります。それほど利用する機会は多くないとは思いますが、覚えておいて下さい。

終わりに

本書では python を通してプログラミングの基本的な概念の説明をしました。それほど大きなプログラムを書かないのであれば本書にある知識だけでもそこそこなものが書けると思います。ただ、本書では取り扱わなかった「オブジェクト指向」と「例外処理」は複雑なプログラムを書くには必須の概念となります。まだプログラミングのいろはも学べていない段階でこれらを前面に出すと混乱を招くと考えたため、本書では必要最低限の箇所を除いてオブジェクト指向に関する話題が取り除かれています。本書の内容を理解出来たと思ったら、ぜひ中編の「オブジェクト指向と例外処理」をお手にとってプログラミングのパラダイムチェンジをしてください。なお、後編では様々な Python のライブラリを駆使して「ネットワーク」や「データベース」といったものをどう操作するかといったアドバンストなトピックを扱います。前編と中編で学んだプログラミングの概念はそれらを使う前提条件となります。