Python3 応用

＜ 目次 ＞

1章 ファイル処理

1．1 ファイル入出力処理

ファイル操作の基本

Pytyon でファイルを操作するときには、open()という組み込み関数を利用します。
この関数を呼び出すとファイルを操作するためのファイルオブジェクトが返されます。

ファイルへの書き込みサンプル

```
f = open("test.txt","w")
f.write("Change!¥n")
f.write("Yes,we can.¥n")
f.close()
```

ファイルからの読み出しサンプル

```
f = open("test.txt","r")
for line in f.readlines():
   print(line, end="")
f.close()
```

open(ファイル名,モード)

モード	説明
w	書き込みモードで開きます 存在するファイルを指定すると中身を空にします
w+	読み書き両方のモードで開きます 存在するファイルを指定すると中身を空にします
r	読み込みモードで開きます（デフォルト） 存在しないファイルを指定するとエラーになります
r＋	読み書き両方のモードで開きます 存在しないファイルを指定するとエラーになります
a	ファイルの末尾に追記の書き込みモードで開きます 存在しないファイルを指定すると新規で作成します
a+	ファイルの末尾に追記の書き込みモードで開きますが、読み込みも可能です 存在しないファイルを指定すると新規で作成します
b	バイナリモード
t	テキストモード（デフォルト）

ファイルオブジェクトのメソッド

メソッド	説明
read(サイズ)	ファイルからサイズ分読む（サイズを省略するとすべて読む）
readline()	ファイルから１行読む
readlines()	ファイルからすべての行を読み、リストを返す
tell()	ファイルの中での現在位置を返す
seek(オフセット,起点)	現在位置を起点からオフセット指定された位置に変更する 起点：0＝ファイルの先頭／1＝現在位置／2＝ファイルの末尾
write(str)	str をファイルに書く
writelines(シーケンス)	文字列を要素に持つシーケンス（リストなど）をファイルに書く
close()	ファイルを閉じる

with 文を使ったファイルの操作（Python2.5 以降）

with 文を使ってファイルをオープンすると、with 文のブロックを抜けると自動的に close()メソッドが呼ばれます。

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

日本語文字コードの入ったファイルを扱う場合

Python3 の open 関数は Unicode 文字列とバイト文字列との変換を透過的に行うオプションがあります。

```
with open("test.txt","w",encoding="utf-8") as f:
  f.write("認めたくないものだな、")
  f.write("自分自身の若さ故の過ちというものを¥n")
```

次のようにファイルはバイナリモードでオープンし、encode メソッドや decode メソッドを使って明示的に変換する方法もあります。

```
with open("test.txt","ab") as f:
  f.write("戦いとは、常に".encode("utf-8"))
  f.write("二手三手先を読んで行うものだ¥n".encode("utf-8"))

with open("test.txt","rb") as f:
  for line in f.readlines():
    print(line.decode("utf-8"),end="")
```

CSV ファイルを扱う場合

CSV ファイルを扱う場合は、CSV モジュールを使うと簡単にできます。

ファイルへの書き込みサンプル

```
import csv
with open("test.csv","w") as f:
    w = csv.writer(f,delimiter=",")
    w.writerow([1,'yamada',171])
    w.writerow([2,'sugino',168])
```

ファイルからの読み出しサンプル

```
import csv
with open("test.csv","r") as f:
    r = csv.reader(f,delimiter=",")
    for row in r:
        print(row)
```

1．2 パスの処理

ファイルの操作で必要なものに、パスの処理があります。
Python ではパスを生成、分解、チェックなどの処理をするために os.path モジュールを使用します。

os.path モジュールの関数

関数	説明
exists(path)	引数で指定した path にファイルやディレクトリなどの種別を問わず、存在するかをチェックし、True/False を返します
isdir(path)	引数で指定した path がディレクトリかチェックし、True/False を返します
isfile(path)	引数で指定した path がファイルかチェックし、True/False を返します
islink(path)	引数で指定した path がリンクかチェックし、True/False を返します
join(path1,path2,…)	OS 依存のパスセパレータを使ってパス成分を結合したパスを返します
dirname(path)	引数で指定した path のディレクトリを返します
basename(path)	引数で指定した path の末端成分のファイル名またはディレクトリ名を返します
abspath(path)	引数で指定した path の絶対パスを返します
split(path)	引数で指定した path をパスセパレータで分解しタプルを返します
splitext(path)	引数で指定した path を拡張子とそうでない部分を分解しタプルで返します
splitdrive(path)	引数で指定した path をドライブとそうでない部分を分解しタプルで返します
normpath(path)	引数で指定した path を正規化（複雑な相対パスを整理）して返します
commonprefix(シーケンス)	引数に複数の path を要素として持つシーケンスを指定し、その共通成分を文字列で返します。

1．3 ファイルシステムに対する操作

ファイルの操作で必要なものに、ファイルシステムに対する操作があります。
Python では基本的に os モジュールの関数を利用し、より高度な操作を行う場合は shutil モジュールを利用します。

os モジュールの関数

関数	説明
remove(path)	引数で指定した path のファイルを削除します
mkdir(path)	引数で指定した path にディレクトリを作成します
mkdirs(path)	引数で指定した path のすべてのパス成分のディレクトリを作成します
rmdir(path)	引数で指定した path のディレクトリを削除します
removedirs (path)	引数で指定した path のすべてのパス成分のディレクトリを削除します
rename(old,new)	引数で指定した old のファイルやディレクトリの名前を new で指定した名前に変更します
chdir(path)	カレントディレクトリを引数で指定した path に変更します
getcwd()	カレントディレクトリを文字列で取得します
listdir(path)	引数で指定した path の直下のファイルやディレクトリの一覧をリストで返します
walk(path)	引数で指定した path の直下のファイルやディレクトリ名を再帰的に取得します 戻り値はジェネレータで、イテレーションごとにディレクトリを辿って、そのパスとその直下のファイルやディレクトリのリストを返します
stat(path)	ファイルやディレクトリのメタデータを取得します
access(path,mode)	ファイルやディレクトリのアクセス権限を確認します
chmod(path,mode)	ファイルやディテクトリのアクセス権限を変更します

shutil モジュールの関数

関数	説明
rmtree(path)	引数で指定した path のディレクトリを中にファイルやディレクトリがあってすべて削除します
copyfile(src,dist)	引数で指定した src のファイルを dist としてコピーします
copy(src,dist)	copyfile 関数と同様にファイルをコピーしますが、dist にディレクトリを指定するとその直下にファイルをコピーします
copy2(src,dist)	copy 関数と同様にファイルをコピーしますが、最終アクセス時間や最終更新時間などのメタデータもコピーします
copytree(src,dist)	引数で指定した src ディレクトリをその内容を含めてすべてを dist ディレクトリにコピーします

2章 文字列処理と日付時間処理

2．1 文字列処理

入門編で Python の文字列について解説をしましたが、ここではもう少し深く文字列オブジェクトの便利なメソッドを紹介します。

str.count(*sub*[, *start*[, *end*]])	[*start*, *end*] の範囲に、部分文字列 *sub* が重複せず出現する回数を返します。
str.find(*sub*[, *start*[, *end*]])	文字列のスライス s[start:end] に部分文字列 *sub* が含まれる場合、その最小のインデックスを返します。オプション引数 *start* および *end* はスライス表記と同様に解釈されます。 *sub* が見つからなかった場合 -1 を返します。
str.index(*sub*[, *start*[, *end*]])	find() と同様ですが、部分文字列が見つからなかったときに ValueError を送出します。
str.isalnum()	文字列中の全ての文字が英数字で、かつ 1 文字以上あるなら真を、そうでなければ偽を返します
str.isalpha()	文字列中の全ての文字が英字で、かつ 1 文字以上あるなら真を、そうでなければ偽を返します。
str.isdecimal()	文字列中の全ての文字が十進数字で、かつ 1 文字以上あるなら真を、そうでなければ偽を返します。（全角半角の算用数字）
str.isdigit()	文字列中の全ての文字が数字で、かつ 1 文字以上あるなら真を、そうでなければ偽を返します。（isdecimal に加えてローマ数字も扱える）
str.isnumeric()	文字列中の全ての文字が数を表す文字で、かつ 1 文字以上あるなら真を、そうでなければ偽を返します。（isdigit に加えて漢数字も扱える）
str.join(*iterable*)	*iterable* 中の文字列を str で結合した文字列を返します。
str.replace(*old*, *new*)	文字列をコピーし、現れる部分文字列 *old* 全てを *new* に置換して返します。
str.split(*sep*)	文字列を *sep* をデリミタ文字列として区切った単語のリストを返します。
str.strip([*chars*])	文字列の先頭および末尾部分を除去したコピーを返します。引数 *chars* は除去される文字集合を指定する文字列です。 *chars* が省略されるか None の場合、空白文字が除去されます。

２．２ 日付時間処理

Python で日付時間処理を行う場合は datetime モジュールをインポートして使用します。

datetime モジュールには代表的な３つのクラスがあります。

datetime	日時を扱うためのクラスです。年月日時分秒のすべての時間に関する情報を扱います。
date	日付を扱うためのクラスです。年月日の情報を扱います。
time	時刻を扱うためのクラスです。時分秒の情報を管理します。

以下の解説では次のようにインポートされていることを前提とします。

```
from datetime import *
```

datetime クラス

現在の日時を取得するには、now メソッドまたは today メソッドを使用します。

```
datetime.now()
datetime.today()
```

現在の日時ではなく、特定の日時を表すオブジェクトを生成するには datetime クラスのコンストラクタを使用します。

```
datetime(年,月,日,時,分,秒,マイクロ秒)
```

日時の要素を取得するには datetime オブジェクトのプロパティを使用します。

year	西暦年を表します
month	月の値（１～１２）
day	日の値（１～３１）
hour	時の値（０～２３）
minute	分の値（０～５９）
second	秒の値（０～５９）
microsecond	マイクロ秒の値（０～９９９９９９）

date クラス

今日の日付を取得するには、today メソッドを使用します。

```
date.today()
```

今日の日付ではなく、特定の日付を表すオブジェクトを生成するには date クラスのコンストラクタを使用します。

```
date(年,月,日)
```

datetime オブジェクトから deta オブジェクトを取り出すには datetime オブジェクトの date メソッドを使用します。

```
a = datetime(2019,1,20,15,30,20)
b = a.date()
```

日付の要素を取得するには date オブジェクトのプロパティを使用します。

year	西暦年を表します
month	月の値（１～１２）
day	日の値（１～３１）

time クラス

特定の時刻を表すオブジェクトを生成するには time クラスのコンストラクタを使用します。

```
time(時,分,秒,マイクロ秒)
```

datetime オブジェクトから time オブジェクトを取り出すには datetime オブジェクトの time メソッドを使用します。

```
a = datetime.now().time()
```

時刻の要素を取得するには time オブジェクトのプロパティを使用します。

hour	時の値（０～２３）
minute	分の値（０～５９）
second	秒の値（０～５９）
microsecond	マイクロ秒の値（０～９９９９９９）

日時の演算

datetime オブジェクト、date オブジェクト、time オブジェクト同士の減算を行うと、結果は timedelta オブジェクトで返ってきます。

timedelta オブジェクトの要素を取得するにはプロパティを使用します。

days	日数を表す値
seconds	秒数を表す値
microseconds	マイクロ秒を表す値

例）生まれてから今日までの日数を算出する

```
delta = date.today() - date(1968,9,18)
print(delta.days)
```

datetime オブジェクト、date オブジェクト、time オブジェクトに timedelta オブジェクトを加算または減算すると、特定の日付時間が返ってきます。

例）今日の 1000 日後は何月何日？

```
print(date.today() + timedelta(days = 1000))
```

3章 Pythonの文法

3．1 クラス

Python ではクラスを扱うことができます。
クラスによってオブジェクト指向のプログラミングが可能になります。

オブジェクト指向とは

「オブジェクト」とは、現実世界に存在する「もの」や「概念」をシステム上で扱う単位です。
「オブジェクト指向」とは、システムに必要な「人・もの・概念」などをオブジェクトに置き換え、オブジェクトとオブジェクトの関係に着目する手法です。

オブジェクトの構成要素（メンバ）

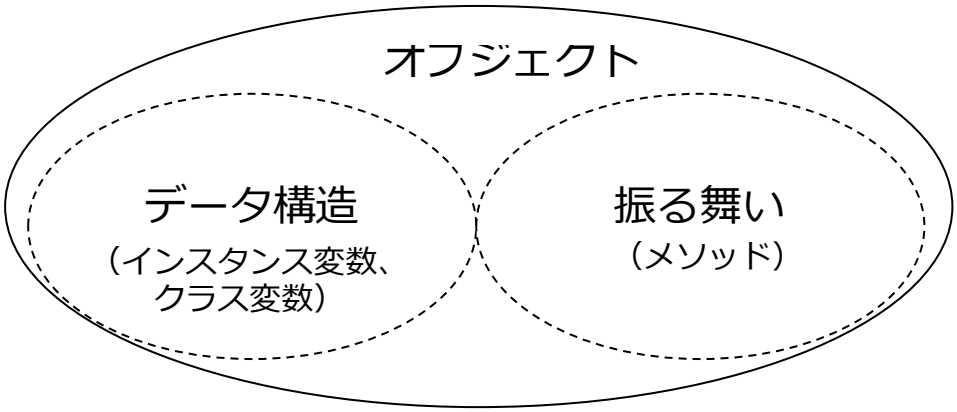
オブジェクトは、主に「データ構造」と「振る舞い」の２つの要素から成り立っています。
Python では、このような構成要素のことをオブジェクトの「メンバ」と呼びます。

（１）データ構造

オブジェクトが持っているもので「量」や「数」のような値を格納するものです。
具体的には、テレビで例えると、画面サイズ、チャンネル、ボリューム量などとなり、車で例えると、排気量、ガソリンの残量、座席数、速度などに相当します。
Python では、「クラス変数」「インスタンス変数」が該当します。

（２）振る舞い（処理・動作）

オブジェクトの振る舞いのことです。
テレビでは、「電源を入れる・切る」「チャンネルを変える」「ボリュームを上げる」というような動きに相当し、車では、「始動する」「前進する」「ブレーキを踏む」「ハンドルを切る」というような動きに相当します。
Python では、「メソッド」と呼ばれます。



オブジェクトの２つの側面

オブジェクトは、「クラス」と「インスタンス」という２つの側面を持っています。
クラスは、オブジェクトの「設計図」のようなものです。
インスタンスとは、クラスから作成された、オブジェクトの「実体」のことです。

Python のクラス定義

Python では class 文を使ってクラスを定義します。

```
class クラス名:
    クラス変数
    def メソッド名(self):
```

クラス内には関数や変数など「メンバ」を定義することができます。

```
class SampleClass:
    a = 10
    def test(self,b):
        print(SampleClass.a+b)
```

メソッド

クラス内に定義した関数を「メソッド」と呼び def 文を使って定義します。
メソッドを定義するときには、必ず第 1 引数に self を指定しなければなりません。第 2 引数以降はメソッドを呼び出すときの第 1 引数以降に対応します。
self はそのインスタンス自身を表す変数名です。実は self という名前でなくても良いのですが、慣習で self が使用されています。

※ Java など他のオブジェクト指向言語ではメソッド名が同じでも引数の型や数が違う場合に、別メソッドとして定義することができ、呼び出すときには引数の状態で呼び分ける機能（オーバーロード）を持つものがありますが、Python はこれを許していません。

クラス変数

クラス内に定義した変数を「クラス変数」と呼びます。
クラス変数はインスタンスを生成しなくても、「クラス名.クラス変数名」でアクセスすることができます。

```
print(SampelClass.a)
```

インスタンスの生成

Python でインスタンスを生成するにはクラス名に括弧をつけて関数のようにして呼び出します。

変数＝クラス名（）

代入された変数でインスタンスを参照することができます。

```
x = SampleClass()
print(x.a)
x.test(10)
```

コンストラクタ

クラスをインスタンス化するときに呼び出される特殊なメソッドをコンストラクタと呼びます。
コンストラクタは__init__というメソッド名前で定義します。

```
class SampleClass:
    def __init__(self):
        初期化処理
```

コンストラクタに引数を持たせることもできます。
その場合インスタンスの生成のときに引数を与えることで値を渡します。

```
class SampleClass:
    def __init__(self,xyz):
        self.b = xyz

x = SampleClass(100)
```

デストラクタ

インスタンスを破棄するときに呼び出される特殊なメソッドをデストラクタと呼びます。
デストラクタは__del__というメソッド名前で定義します。

```
class SampleClass:
    def __del__(self):
        終了処理
```

デストラクタは明示的には del 関数を使って動作を確認することができます。

```
class SampleClass:
    def __del__(self):
        print("FINISH")

x = SampleClass()
print("CALL DELETE")
del x
print("RETURN DELETE")
```

```
--実行結果---
CALL DELETE
FINISH
RETURN DELETE
```

インスタンス変数

メソッド内で self.変数名として値を代入すると、自動的にインスタンス変数を生成したことになります。
インスタンス変数はクラスの外から「クラスのインスタンス.変数名」でアクセスすることができます。
また、クラスの中からは「self.変数名」でアクセスすることができます。

クラス変数とインスタンス変数

クラス変数は同じクラスから生成された全てのインスタンスで共通して使うことができる変数です。
それに対してインスタンス変数はそのインスンスだけで使用することができる変数です。

```
class SampleClass():
    a = 10
    def __init__(self,xyz):
        self.b = xyz
    def test(self):
        print(SampleClass.a)
        print(self.b)
```

```
>>> x = SampleClass(10)
>>> y = SampleClass(20)
>>> x.test()
>>> y.test()
>>> SampleClass.a = 100
>>> x.test()
>>> y.test()
```

クラス変数もインスタンス変数のように「クラスのインスタンス.変数名」やクラスの内部からであれば「self.クラス変数名」でも参照可能です。しかし、同名のインスタンス変数が存在する場合はインスタンス変数が優先されるので注意が必要です。

クラスメソッド

クラス変数のように、クラスから直接利用できるメソッドをクラスメソッドといいます。
クラスメソッドを定義するときはメソッドの前に「@classmethod」と書かなければなりません。（このように@の記号で始まるプログラムの印はアノテーションと呼ばれます）
また、クラスメソッドの引数は必ず第 1 引数に cls を指定しなければなりません。第 2 引数以降はメソッドを呼び出すときの第 1 引数以降に対応します。
cls はそのクラスを表す変数名です。実は cls という名前でなくても良いのですが、慣習で cls が使用されています。

```
@classmethod
def test(cls):
    print(cls.a)
```

非公開メンバ

クラスを利用する側からは隠蔽したい（クラスの内部から利用することはできる）非公開のメンバを作りたいときがあります。

Python のクラスにはメンバを完全に非公開にする機能はありません。
その代わりに、メソッド名や変数名が「アンダーバー（_）から始まっているものは非公開とする」という暗黙のルールがあります。

そうは言っても、それは「暗黙のルール」なので知らない人は操作してしまいます。
そこで、「マングリング」という機能を利用してクラス利用者から要素名を操作しづらくする手法が使われます。

「マングリング」とは、スーパークラスのメンバと同じ名前を、誤ってサブクラスのメンバに付けたときに問題が起きないようにする機能です。

具体的には隠蔽したいメンバの名前にアンダーバーを２つ（__）付けます。
この名前をつけられたメンバは、同じクラス内から呼び出されるときは、普通のメンバと同じように呼び出すことができますが、それ以外の場合は呼び出し方を変えなければならなくなります。それはアンダーバー（_）とクラス名を接頭して、その後にメンバ名で呼び出すという方法です。

```
class SampleClass:
    def __init__(self):
        self._a = 10
        self.__a = 20
```

```
>>> x = SampleClass()
>>> x._a
>>> x.__a
>>> x._SampleClass__a
```

プロパティ

クラスに値を保管する場合にインスタンス変数を使用しますが、クラス外からのアクセスが容易だとインスタンス変数を管理することが難しくなります。
もちろんインスタンス変数を非公開メンバとして定義すれば、クラス外からのアクセスを抑制できますが、その代わりにメソッドを介して変数へアクセスすることになるため、メソッド数が増えメソッドと変数の関係がわかりにくくなります。
この問題を解決するために、インスタンス変数へ直接アクセスするかのように記述しつつ、メソッドを介して変数を管理するプロパティという仕組みが用意されています。

プロパティは次のように定義します。
プロパティを使った値の取得メソッド

```
@property
def プロパティ名(self):
    return 値
```

プロパティを使った値の設定メソッド

```
@プロパティ名.setter
def プロパティ名(self,値):
    値を設定する処理
```

以下にサンプルを示します。

```
class SampleClass:
    def __init__(self):
        self.__val = 20

    @property
    def val(self):
        return self.__val

    @val.setter
    def val(self,value):
        self.__val = value
```

```
>>> a = SampleClass()
>>> print(a.val)
>>> a.val = 50
>>> print(a.val)
```

継承

継承とは既に定義してあるクラスを元にして新たなクラスを定義することです。
基になるクラスを「スーパークラス（基底クラス）」と呼び、新たに定義されたクラスを「サブクラス（派生クラス）」と呼びます。

```
Class スーパークラス:
    スーパークラスの定義

Class サブクラス名(スーパークラス名):
    サブクラスの定義
```

サブクラスをインスタンス化するとスーパークラスのメンバーはすべてサブクラスのメンバーとして利用することができます。

ただし、スーパークラスとサブクラスで同じ名前のメソッドが定義されている場合はサブクラスのメソッドが優先されます。このことをオーバーライドといいます。
メソッドをオーバーライドしてもスーパークラスのメソッドが消滅したわけではなく、サブクラスの中で「super()」の後にドットでスーパークラスのメソッド名を指定すると呼び出すことができます。

```
Class スーパークラス:
    スーパークラスの定義

Class サブクラス名(スーパークラス名):
    def __init__(self):
        super().__init__():    #スーパークラスのコンストラクタを呼び出す

    def メソッド名(self):
        super().スーパークラスのメソッド名(引数)
```

多重継承

Python のオブジェクト指向には多重継承があります。これは、複数のクラスをスーパークラスとして指定することができるというものです。

```
Class サブクラス名(スーパークラスA，スーパークラスB，スーパークラスC):
    サブクラスの定義
```

多重継承で問題となるのが、同名のメンバが存在したときの扱いです。
この問題を Python では、多重継承で指定したクラスの順番で左側で指定したものが優先されるというルールで解決しています。

３．２ イテレータ

イテレータ（iterator）とは、日本語では「反復子」と訳され値を一つずつ順に取り出すための仕組みです。

これまで何度も使用してきた for 文はイテレータの機能を使って実現しています。
for 文の in の後で使うことの多いのが range 関数やリストですが、range 関数は、任意の個数の整数を取り出すことができるイテレータを生成する関数、リストはイテレータを生成可能なデータ型です。help 関数を使って確認すると、どちらも__iter__メソッドを持っていることがわかります。

iter 関数を使うと__iter__メソッドを持つオブジェクトからイテレータオブジェクトを取得することができます。
イテレータオブジェクトから順にデータを取り出すメソッドは__next__メソッドで、next 関数を使うことで__next__メソッドを呼び出すことができます。
イテレータオブジェクトが最後までデータを出し切った状態で__next__メソッドを呼び出すと StopIteration 例外が発生します。

```
>>> a = range(1,10,3)
>>> b = iter(a)
>>> next(b)
1
>>> next(b)
4
>>> next(b)
7
>>> next(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

for 文はこのような仕組みを利用することでループ処理を実現しています。

```
for x in range(1,10):
    print(x,end="")
```

```
a = iter(range(1,10))
while True:
    try:
        x = next(a)
        print(x,end="")
    except:
        break
```

クラスを使って、__iter__メソッドと__next__メソッドを実装すればイテレータを作成することができます。

クラスでイテレータの作成

```
class クラス名:
    def __iter__(self):
        return(self)
    def __next__(self):
        if 要素がまだあるか:
            return(次の要素)
        else:
            raise StopIteration()
```

イテレータ実装のサンプル

```
import random

class ShuffleIter:
    def __init__(self,*val):
        self.value = list(val)
        random.shuffle(self.value)
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count < len(self.value):
            data = self.value[self.count]
            self.count += 1
            return data
        else:
            raise StopIteration()

a = ShuffleIter('First','Second','Third','Fourth','Fifth','Sixth')
for x in a:
    print(x)
```

３．３ ジェネレータ

ジェネレータ（generator）は、イテレータを手軽に作成する仕組みのことです。
ジェネレータを使ってイテレータを作るには、一つ関数を定義して値を返すときに return ではなく yield 文を使うことで実現します。
yield 文を使うと処理はそこで終わりではなく、関数内の状態を保持して次回その関数が呼び出されたときに続きから処理を実行します。

サンプルとして'A','B','C'を順番に返すイテレータをジェネレータで作成してみます。

```
def genSample():
    yield 'A'
    yield 'B'
    yield 'C'

gens = genSample()    # イテレータを取得
for x in gens:
    print(x)
```

もう少し複雑な課題をジェネレータで作ってみます。
次のプログラムは素数を返すイテレータをジェネレータで作成したものです。

```
def genPrime(max):
    for i in range(2, max+1):
        for j in range(2, i//2+1):
            if i%j == 0:
                break
        else:
            yield i

genp = genPrime(100)    # イテレータを取得
for x in genp:
    print(x)
```

【演習】フィボナッチ数列を返すイテレータをジェネレータで作成してください。
フィボナッチ数列とは「２つ前の項と１つ前の項を足し合わせたものが項の値となる数列で、最初の２項は１、１」です。
１、１、２、３、５、８、１３、２１、３４、５５・・・となります。

３．４ デコレータ

デコレータ（decorator）は、関数の処理を装飾するための仕組みのことです。
デコレータを使うと、手軽に関数の前後に任意の処理を付け加えることができます。

デコレータ関数は、関数の中にラッパー関数を定義してラッパー関数のオブジェクトを戻り値として返すように定義します。

```
def デコレータ名(func):
    def wrapper(*args,**kwargs):
        前処理
        res = func(*args,**kwargs)
        後処理
        return res
    return wrapper
```

デコレータ関数を使う場合は、装飾したい関数定義の直前に「@デコレータ名」を記述します。

```
@デコレータ名
def 関数():
    処理
```

この関数を実行すると以下のように実行されます。

```
前処理
処理
後処理
```

デコレータのサンプルコーディング

```
def decoSample(func):
    def wrapper(*args,**kwargs):
        print("** PRE PROC **")
        res = func(*args,**kwargs)
        print("** POST PROC **")
        return res
    return wrapper

@decoSample
def test(s):
    print(s)

test("TEST")
```

```
--実行結果---
** PRE PROC **
TEST
** POST PROC **
```

３．５ 無名関数

無名関数とは、名前の無い関数のことで匿名関数ともいいます。
Python では lambda（ラムダ）を使って無名関数を作成します。

```
lambda 引数１,引数２…：式
```

lambda を使った無名関数は複雑な処理には向いていませんが、引数を括弧で括る必要もなく、return を書く必要もないのでシンプルに関数オブジェクトを作成できます。

無名関数サンプル

```
func = lambda x : x**2+2*x + 10

y = func(10)
print(y)
```

map 関数との組み合わせ

map 関数はリストのすべての要素に対して関数の処理を呼び出すときに使用します。

```
map(関数オブジェクト,iterable)
```

第一引数の関数オブジェクトを、第二引数の iterable のリストやタプルなどの要素すべてに対して適用し結果を返します。

```
>>> list(map(lambda x:x**2+2*x+10,range(1,10)))
[13, 18, 25, 34, 45, 58, 73, 90, 109]
```

filter 関数との組み合わせ

filter 関数はリストから任意の要素だけを取り出すときに使用します。

```
filter(関数オブジェクト,iterable)
```

第一引数の関数オブジェクトを、第二引数の iterable のリストやタプルなどの要素すべてに対して適用し結果が True の要素だけを返します。

```
>>> list(filter(lambda x:x%3 == 0 or x%10 == 3 or x//10 == 3,range(1,40)))
[3, 6, 9, 12, 13, 15, 18, 21, 23, 24, 27, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39]
```

３．６ 特殊変数__name__

特殊変数__name__は直接呼び出したスクリプトファイル内で参照すると'__main__'という値が設定され、直接呼び出しではなくインポートしたモジュール内で参照するとそのモジュールの名前が設定されています。

```
if __name__ == '__main__':
    このファイルを直接呼び出されたときの処理
```

nametest.py というファイルで以下の内容を記述して

```
if __name__ == '__main__':
    print("MAIN CALL!!")
else:
    print(__name__)
```

コマンドプロンプトから「python nametest.py」または「python -m nametest」で実行すると「MAIN CALL!!」と表示されます。

もう一つ calltest.py というファイルで以下の内容を記述して

```
import nametest
```

コマンドプロンプトから「python calltest.py」または「python -m calltest」で実行すると「nametest」と表示されます。

この変数を使うことでモジュールのテストなど、呼び出し方によって処理を変えたい場合に利用できます。

３．７ システムパラメータ（sys モジュール）

sys モジュールはシステム関連の情報を取得するための機能を提供する標準モジュールです。
いくつかの便利な機能を紹介します。
詳細については Python ドキュメント（https://docs.python.jp/3/library/sys.html）を参照のこと

sys.argv

コマンドを使ってプログラムを実行する場合に、コマンドの後にパラメータを指定してプログラムに渡したい時があります。

```
python コマンド名　パラメータ１　パラメータ２…
```

このときプログラムからは sys.argv にリストとしてコマンド名とパラメータが格納されています。コマンド名は sys.argv[0]に、それ以降にパラメータが格納されます。

例えば、以下の内容を argvtest.py というファイル名で作成して

```
import sys
print(argv)
```

コマンドプロンプトから次のように実行すると

```
python argvtest.py 111 222 333
```

結果は以下のようににになります。

```
['argvtest.py','111','222','333']
```

sys.path

モジュールを検索するパスを示す文字列のリストを格納しています。

sys.platform

実行しているプラットホーム（OS）の情報を格納しています。

platform の値	システム
'linux'	Linux
'win32'	Windows
'cygwin'	Windows/Cygwin
'darwin'	Mac OS X

sys.version

使用している Python のバージョンを格納しています。
sys.version は文字列として情報を格納しています。
'3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]'

sys.version_info

使用している Python のバージョンを格納しています。

sys.version_info はオブジェクトとして情報を格納しています。

sys.version_info.major	メジャーバージョン	整数
sys.version_info.minor	マイナーバージョン	整数
sys.version_info.micro	マイクロバージョン	整数
sys.version_info.releaselevel	リリースレベル	'alpha','beta','candidate','final'
sys.version_info.serial	シリアル番号	整数

sys.exit()

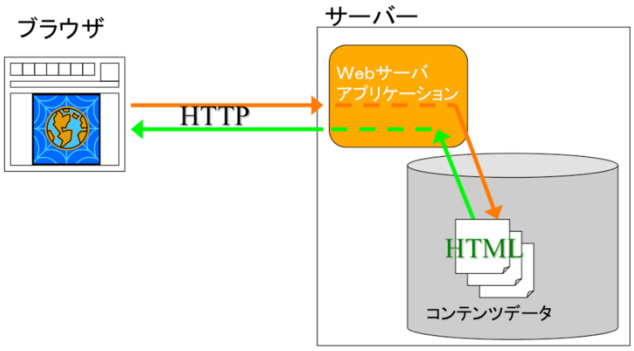
プログラムを終了するメソッドです。

引数として終了ステータス（０～１２７）を渡すことができます。（０は正常終了）

4章 urllib モジュールによる HTTP 通信

4．1 HTTP 通信

HTTP とは、HTTP サーバとクライアントがデータを送受信するプロトコルです。



HTTP の通信はクライアントからサーバに向かう情報をリクエストメッセージ、サーバからクライアントに返送される情報をレスポンスメッセージと呼びます。

リクエストメッセージはリクエストライン、HTTP ヘッダ部、HTTP ボディ部とで構成されています。

リクエストラインはメソッド、URI、クライアントの HTTP バージョンで構成されています。それぞれの情報は空白によって区切られます。

HTTP ヘッダ部と HTTP ボディ部は空行によって区切られます。

〔リクエストメッセージ〕

GET　/aaa/bbb.html　HTTP/1.1	←リクエストライン
HTTP ヘッダ部	
空行	
HTTP ボディ部	

リクエストラインのメソッドには次に示す情報を記述することができます。

メソッド	意味	HTTP
GET	URI で指定した情報を要求する	1.0～
POST	フォームの情報をサーバに送信する	1.0～
HEAD	HTTP ヘッダ情報だけを要求する	1.0～
OPTIONS	通信オプション情報を通知／要求する	1.1～
PUT	URI で指定した情報を登録する	1.1～
DELETE	URI で指定した情報を削除する	1.1～
TRACE	サーバで受け取ったリクエストを送り返すように指示する	1.1～
CONNECT	プロキシに転送する場合に使用する	1.1～

これらのメソッドの中でも Web システムではサーバサイドに情報を送ることができる GET メソッドと POST メソッドがよく使われます。

GET メソッドは、サーバサイドへの送信情報を HTTP ヘッダ部の URI に付加して送ります。URI に付加して送るためサーバによってはサイズが大き過ぎると受け付けられない場合があります。
また、ブラウザ上の URL の欄にその情報が表示されます。

URL?[パラメータ１]=[パラメータ１の値]&[パラメータ２]=[パラメータ２の値]&…
「？」がＵＲＬとＨＴＴＰパラメータの区切り、「＆」が複数のパラメータの区切りです。

POST メソッドは、サーバサイドへの送信情報を HTTP ボディ部に格納して送信します。ボディ部なのでデータのサイズ制限はありません。

レスポンスメッセージはステータスラインと、HTTP ヘッダ部、HTTP ボディ部で構成されています。
ステータスラインはサーバの HTTP バージョン、ステータスコード、レスポンスフレーズで構成されています。それぞれの情報は空白によって区切られます。

〔レスポンスメッセージ〕

HTTP/1.1　200　OK	←ステータスライン
HTTP ヘッダ部	
空行	
HTTP ボディ部	

ステータスラインのステータスは次の情報を通知します。

コード	説明	備考
1xx	処理の経過状況などを通知する	
2xx	正常終了	200 OK
3xx	何らかの別のアクションが必要であることを通知する	301 Moved Permanently 302 Found 303 See Other 304 Not Modified 307 Temporary Redirect
4xx 401 403 404	クライアント側のエラー リクエストはユーザー認証を必要とする リクエストを理解したが実行を拒絶した URI に一致するものが無かった	 Unauthorized Forbidden Not Found
5xx	サーバ側のエラー	500 Internal Server Error

４．２ urllib.request モジュール

urllib.request モジュールは URL による任意のリソースへのアクセスを提供します。
このモジュールは HTTP を介してデータを取り寄せるための高レベルのインタフェースを提供します。特に、関数 urlopen()は組み込み関数 open()と同様に動作し、ファイル名の代わりに URL を指定することができます。

GET リクエストを送信する

```
from urllib.request import urlopen

with urlopen("http://www.example.jp/?q=keyword") as res:
    data = res.read().decode("utf-8")
    print(data)
```

urlopen(url [,data])	url に接続します。 data パラメタを指定が指定された場合、HTTP リクエストは GET でなく POST になります。 ファイル類似のオブジェクトが返されます。 このオブジェクトは以下のメソッド: read(), readline(), readlines(), fileno(), close(), info(), getcode(), geturl() をサポートします。

POST リクエストを送信する

```
from urllib.request import urlopen
from urllib.parse import urlencode

params = {'q':'keyword','page':1}
enc_params = urlencode(params).encode("utf-8")
with urlopen("http://www.example.jp/",enc_params) as res:
    data = res.read().decode("utf-8")
    print(data)
```

HTML データの解析には html.parser モジュールを使うと便利です。

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    # 開始タグの検出
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)
    # 終了タグの検出
    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)
    # データの検出
    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>テスト</title></head>'
            '<body><h1>これはテストデータです</h1></body></html>')
```