Python3 応用
（オブジェクト指向）

＜ 目次 ＞

セイ・コンサルティング・グループ株式会社

Self-Actualization for You SAY
セイ・コンサルティング・グループ

1章 正規表現

正規表現（Regular Expressions）とは文字列のパターンを表現するために利用される表現手法のことです。正規表現は文字列の検索や置換に大きな力を発揮します。
正規表現では通常の文字に加えてメタキャラクタという特殊文字を用いて文字列のパターンを表現します。

正規表現のメタキャラクタ

文字	説明
.	改行を除いた任意の文字にマッチします。
^	文字列の先頭にマッチします。
$	文字列の末尾、あるいは文字列の末尾の改行の直前にマッチします。
[ ]	"["と"]"の間に指定した文字のどれか一つにマッチします。 文字は個々に列記するか、最初と最後の文字の間に"-"を挟んで範囲指定できます。
[^]	"[^"と"]"の間に指定した文字以外の文字にマッチします。 文字は個々に列記するか、最初と最後の文字の間に"-"を挟んで範囲指定できます。
¥d ¥D	¥d は数字とマッチします。¥D は数字以外とマッチします。
¥s ¥S	¥s は空白やタブなど空白文字とマッチします。¥S は空白文字以外とマッチします。
¥w ¥W	¥w は大文字小文字を含む英数字とマッチします。¥W は英数字以外とマッチします。
*	直前にある正規表現の 0 回以上の繰り返し、最長パターンにマッチします。
+	直前にある正規表現の 1 回以上の繰り返し、最長パターンにマッチします。
?	直前にある正規表現の 0 回か 1 回の繰り返し、最長パターンにマッチします。
*? +? ??	"*"や"+"、"?"の直後に"?"を付けると、最短パターンにマッチします。
{m}	直前の文字の m 回の繰り返しにマッチします。
{m,n} {m,n}?	直前の文字の m 回以上 n 回以下の繰り返し、最長パターンにマッチします。 直後に"?"を付けると最短パターンにマッチします。
|	"|"の左右にあるパターンのどちらかにマッチします。
( )	"("と")"との間に指定したパターンをグループ化します。 グループとしてまとめた部分文字列は match オブジェクトの group()メソッドや groups()メソッドで取り出すことができます。

¥A	文字列の先頭だけにマッチします。
¥b	単語を構成する文字(¥w)と構成しない文字(¥W)の間の境界にマッチします。 つまり、単語の先頭あるいは末尾にマッチします。
¥B	単語の先頭あるいは末尾でない空文字列にマッチします。
¥d	¥d は数字とマッチします。[0-9]と同じ
¥D	¥D は数字以外とマッチします。[^0-9]と同じ
¥s	¥s は空白やタブなど空白文字とマッチします。[¥t¥n¥r¥f¥v]と同じ
¥S	¥S は空白やタブなど空白文字以外とマッチします。[^¥t¥n¥r¥f¥v]と同じ
¥w	¥w は大文字小文字を含む英数字およびアンダーバーとマッチします。 [a-zA-Z0-9_]と同じ
¥W	¥w は大文字小文字を含む英数字およびアンダーバー以外とマッチします。 [^a-zA-Z0-9_]と同じ
¥Z	文字列の末尾だけにマッチします。

１．１　re モジュール

Python で正規表現を使うには re モジュールを利用します。

```
import re
```

re モジュールにはさまざまな便利な関数が用意されています。

re モジュールの関数

re モジュールの関数	説明
match(pattern,str)	pattern が str の先頭にマッチするか調べて match オブジェクトを返す。（見つからない場合は None を返す）
search(pattern,str)	pattern が str のどこかにマッチするか調べて match オブジェクトを返す。（見つからない場合は None を返す）
split(pattern,str)	pattern で str を分割してリストを返す。
findall(pattern,str)	str の中で pattern にマッチするものをすべて探し、文字列のリストとして返す。
sub(pattern,repl,str)	str の中で pattern にマッチするものを repl に置換する。
compile(pattern)	pattern を予めコンパイルし正規表現オブジェクトを返す。

１．１．１　search 関数

search 関数を使って文字列がパターンに一致するかを判定するプログウラムは以下のように書きます。

```
str = "SMAP TOKIO V6 KAT-TUN 嵐"
pattern = r"[0-9]+"
mat = re.search(pattern,str)
if mat is None:
    print("Not Found")
else:
    print("Found")
```

正規表現のパターンを記述する際には普通の文字列ではなく、r"…"で表現する raw 文字列を使うことが勧められています。
通常の文字列で"¥"（または"\"）はエスケープ文字として扱われるため、"¥"（または"\"）自身を表現するには重ねて記述します。そのため正規表現パターン文字列が読みづらくなるので"¥"（または"\"）をそのままの文字として扱うことができる raw 文字列が適しています。

１．１．２　match 関数

match と search は似ていますが、match は必ず文字列の先頭から比較します。

```
strs = ["abc","ABC","012abc"]
pattern = r"[a-z]+"
for x in strs:
    m = re.match(pattern,x)
    if m is not None:
        print("match",x)
    m = re.search(pattern,x)
    if m is not None:
        print("search",x)

--実行結果--------------
match abc
search abc
search 012abc
```

１．１．３　split 関数

split では正規表現で区切り文字を表すので複数の区切り文字が混在しても切り出せます。

```
str = "SMAP,TOKIO:V6   KAT-TUN       嵐"
pattern = r"[ ,:¥t]+"
johnnys = re.split(pattern,str)
for group in johnnys:
    print(group)

--実行結果--------------
SMAP
TOKIO
V6
KAT-TUN
嵐
```

１．１．４　findall 関数

パターンに一致した部分の切り出しは findall メソッドでできます。

```
str = "SMAP TOKIO V6 KAT-TUN 嵐"
pattern = r"¥b[-A-Z]+¥b"
johnnys = re.findall(pattern,str)
for group in johnnys:
    print(group)

--実行結果--------------
SMAP
TOKIO
KAT-TUN
```

１．１．５　sub 関数

パターンに一致した部分の置き換えは sub 関数でできます。

```
str = "SMAP TOKIO V6 KAT-TUN 嵐"
pattern = r"¥b[-B-Z]+A[-B-Z]+¥b"
replace = r"***"
johnnys = re.sub(pattern,replace,str)
print(johnnys)

--実行結果--------------
*** TOKIO V6 *** 嵐
```

パターンに一致した部分の置き換え（後方参照）も sub 関数でできます。

```
str = "SMAP TOKIO V6 KAT-TUN 嵐"
pattern = r"¥b([-B-Z]+A[-B-Z]+)¥b"
replace = r"【¥1】"
johnnys = re.sub(pattern, replace,str)
print(johnnys)

--実行結果--------------
【SMAP】 TOKIO V6 【KAT-TUN】 嵐
```

１．１．６　compile 関数

同じパターンの処理を何度も繰り返す場合は、予めパターンをコンパイルしておく方法もあります。compile 関数の戻り値は正規表現オブジェクトです。

```
strs = ["abc","ABC","012abc"]
pattern = re.compile(r"[a-z]+")

for x in strs:
    m = pattern.search(x)
    if m is not None:
        print("match",x)
```

１．２　正規表現オブジェクト

re モジュールの compile 関数を利用することで正規表現オブジェクトを取得することができます。
この正規表現オブジェクトには、re モジュールの関数と同等なメソッドが提供されていて、同じパターンで繰り返し処理を行う場合など処理の効率を上げることができます。

正規表現モジュールのメソッド

メソッド	説明
match(str)	pattern が str の先頭にマッチするか調べて match オブジェクトを返す。 （見つからない場合は None を返す）
search(str)	pattern が str のどこかにマッチするか調べて match オブジェクトを返す。 （見つからない場合は None を返す）
split(str)	pattern で str を分割してリストを返す。
findall(str)	str の中で pattern にマッチするものをすべて探し、文字列のリストとして返す。
sub(repl,str)	str の中で pattern にマッチするものを repl に置換する。

1．3　match オブジェクト

re モジュールの match 関数と search 関数、正規表現オブジェクトの match メソッドと search メソッドの戻り値は match オブジェクトです。このオブジェクトから一致した詳細な情報や便利な取り出し方を利用できます。

match オブジェクトのメソッド

メソッド	説明
expand(template)	sub()メソッドと同様にマッチした文字列で template 文字列を置換
group([g])	マッチしたサブグループ g を返す
groups()	パターンにマッチしたすべてのサブグループ一覧を返す
groupdict()	名前付きのサブグループを辞書型で返す
start([g])	グループ g とマッチした部分文字列の先頭のインデックスを返す
end([g])	グループ g とマッチした部分文字列の末尾のインデックスを返す
span([g])	グループ g に関して(start,end)のタプルを返す

1．3．1　groups メソッド

match オブジェクトの groups メソッドでより詳細な一致情報が得られます。

```
str = "192.168.1.100"
pattern = r"(¥d+)¥.(¥d+)¥.(¥d+)¥.(¥d+)"
mat = re.search(pattern,str)
if mat is None:
    print("Not Found")
else:
    print(mat.groups())

--実行結果--------------
('192', '168', '1', '100')
```

1．3．2　groupdict メソッド

match オブジェクトの groupdict メソッドは辞書形式で詳細な一致情報が得られます。

```
str = "192.168.1.100"
pattern = r"(?P<one>¥d+)¥.(?P<two>¥d+)¥.(?P<three>¥d+)¥.(?P<four>¥d+)"
mat = re.search(pattern,str)
if mat is None:
    print("Not Found")
else:
    print(mat.groupdict())

--実行結果--------------
{'four': '100', 'one': '192', 'three': '1', 'two': '168'}
```

2章 DB 操作

2．1 SQLite

SQLite は MySQL などと同じリレーショナルデータベースですが、サーバーとして動作させるのではなくライブラリをアプリケーションに組み込んで利用します。
SQLite3 と接続するときは Python 標準ライブラリの sqlite3 モジュールを使います。

このモジュールを使うには、connect 関数を使って Connection オブジェクトを作ります。

sqlite3.connect(DB ファイル)

connect 関数は指定した DB ファイルへの接続を実行して、Connection オブジェクトを返します。
Connection オブジェクトから Cursor オブジェクトを取得して、さまざまなクエリを実行します。

DB 操作サンプル
user テーブルを生成して、データを登録する。

```
import sqlite3

# DB ファイルへ接続
conn = sqlite3.connect("example.db")
# カーソルの取得
cur = conn.cursor()
# テーブルの生成
cur.execute("CREATE TABLE user (id integer,name text,tall real,weight real)")
# データの挿入
param = (1,'yamada',171.5,68.5)
cur.execute("INSERT INTO user VALUES (?,?,?,?)",param)
# 更新の確定
conn.commit()
# DB 接続を閉じる
conn.close()
```

Connection オブジェクト

メソッド	説明
cursor()	Cursor オブジェクトを返します。
commit()	現在のトランザクションをコミットします。 このメソッドを呼ばないと、前回 commit()を呼び出してから行ったすべての変更は、他のデータベースコネクションから見ることができません。
rollback()	最後に行った commit()後の全ての変更をロールバックします。
close()	データベースコネクションを閉じます。 commit()をせずにコネクションを閉じると、変更が消えてしまいます。

Cursor オブジェクト

メソッド	説明

execute(sql [,parameters])	SQL 文を実行します。SQL 文ではプレースホルダー記法をサポートしており、組み込むデータをパラメータで与えることがきます。
	
	sqlite3 モジュールは 2 種類のプレースホルダー記法をサポートします。
	一つは疑問符スタイル、もう一つは名前スタイルです。
	
	# 疑問符スタイル
	cur.execute("insert into people values (?, ?)", (who, age))
	
	# 名前スタイル
	cur.execute("select * from people where name_last=:who and age=:age", {"who": who, "age": age})

executemeny(sql [,parameters])	execute メソッドと同様に SQL 文を実行します。 execute メソッドとの違いは、複数のパラメータをリストで与えることができ、連続して SQL 文を実行することができます。
fetchall()	全てのクエリ結果の row をフェッチして、リストを返します。 これ以上の row がない場合は、空のリストが返されます。
fetchone()	クエリ結果から次の row をフェッチして、1 つのシーケンスを返します。 これ以上データがない場合は None を返します。

インデックスを指定して情報の取得

```
import sqlite3

# DB ファイルへ接続
conn = sqlite3.connect("example.db")
# カーソルの取得
cur = conn.cursor()
# user テーブルから情報取得
cur.execute("select * from user;")
# 情報の表示
for row in cur.fetchall():
    print(row[0],row[1],row[2],row[3])
# DB 接続を閉じる
conn.close()
```

列名を指定して情報の取得

```
import sqlite3

# DB ファイルへ接続
conn = sqlite3.connect("example.db")
#
conn.row_factory = sqlite3.Row
# カーソルの取得
cur = conn.cursor()
# user テーブルから情報取得
cur.execute("select * from user;")
# 情報の表示
for row in cur.fetchall():
    print(row['id'],row['name'],row['tall'],row['weight'])
# DB 接続を閉じる
conn.close()
```

SQLite コマンド

SQLite で使用するデータベースファイルを Python からではなく、SQLite3 というツールでコマンドから操作することもできます。
SQLite3 コマンドは Python には同梱されていませんので SQLite のサイト（http://www.sqlite.org）からダウンロードして使用することになります。

（１）コマンドプロンプトを起動しプログラムを配置したディレクトリへ移動します

```
C:¥> cd sqlite
```

（２）sqlite を起動する（情報を保存するファイルとして dbfile を指定した場合）

```
C:¥sqlite> sqlite3 dbfile
```

プロンプトが「sqlite>」となっている間は sqlite のコマンドモードです。
この状態で sqlite コマンドや SQL を実行することができます。

（３）sqlite を停止する（「.quit」コマンド）

```
stqlite> .quit
```

主な sqlite コマンド

.databases	データベースを一覧表示する
.dump	データベースをダンプする
.exit	コマンドラインプログラムを終了する
.help	コマンドの使用方法を表示する
.quit	コマンドラインプログラムを終了する
.schema	テーブルスキーマを表示する
.tables	パターンにマッチするテーブル名を表示する

主な sqlite がカラムで扱うデータ型

INTEGER	符号付き整数型
TEXT	文字列型
REAL	浮動小数点数型
NUMERIC	固定小数点型
NONE	型指定なし

SQLite コマンド上で SQL での DB の操作

（１）テーブルを作成する

```
CREATE TABLE tbl(
  id    INTEGER,
  name  TEXT,
  home  TEXT,
  PRIMARY KEY(id)
);
```

（２）データを追加する

```
INSERT INTO tbl VALUES(6,'山田篤彦','三重県');
INSERT INTO tbl(id,name) VALUES(7,'松井秀喜');
```

（３）データを抽出する

```
SELECT * FROM tbl;
SELECT * FROM tbl WHERE id = 6;
```

（４）データを更新する

```
UPDATE tbl SET home = '愛知県' WHERE id = 6;
UPDATE tbl SET home = '石川県' WHERE name = '松井秀喜';
```

（５）データを削除する

```
DELETE FROM tbl WHERE id = 6;
```

（６）テーブルの削除

```
DROP TABLE tbl;
```

2．2 MySQL

MySQL と接続するためのモジュールは標準ライブラリにはありません。
そのためサードパーティ制のモジュールを PyPI からインストールします。
PyPI とは Python Package Index の略称で、Python のサードパーティソフトウェアリポジトリ（https://pypi.python.org/pypi）です。
PyPI には MySQL に接続するためのさまざまなライブラリがありますが、Python3 では MySQLdb からフォークされた mysqlclient モジュールがお勧めです。

mysqlclient モジュール

Anaconda で mysqlclient モジュールをインストールするには、Anaconda ナビゲーターの Environment からインストールするか、コマンドプロンプトから pip install mysqlclient を実行してインストールします。

mysqlclient モジュールの使い方は基本的に Python 標準ライブラリの sqlite3 と同です。
まずコネクションオブジェクトを作成し、そこからカーソルオブジェクトを作ります。
できたカーソルオブジェクトを使って様々なクエリを実行します。

```
MySQLdb.connect( user=接続ユーザ名,
                 password=接続ユーザーパスワード,
                 host=接続先ホスト,
                 db=接続データベース名)
```

connect 関数は指定した DB ファイルへの接続を実行して、Connection オブジェクトを返します。Connection オブジェクトから Cursor オブジェクトを取得して、さまざまなクエリを実行します。

データベースのテーブル作成サンプル

```
import MySQLdb

# DB ファイルへ接続
conn = MySQLdb.connect(
        user='user',
        passwd='tebasaki',
        host='localhost',
        db='mydb'
)

# カーソルの取得
cur = conn.cursor()
# テーブルの生成
sql = '''create table mytable (
id int primary key auto_increment,
name varchar(32),
price integer
)'''
cur.execute(sql)
```

```
# データの挿入
sql = 'insert into mytable(name,price) values (%s, %s)'
datas = [('hoge',100),('foo',250),('bar',120)]
cur.executemany(sql, datas)
# 更新の確定
conn.commit()
# DB 接続を閉じる
conn.close()
```

SQLite3 モジュールの execute,executemany メソッドのプレースホルダーでは「?」を使用しましたが、の MySQLdb モジュールの execute,executemany メソッドのプレースホルダーでは「%s」を使用します。

データベースのテーブル参照のサンプル

インデックスを指定して情報の取得

```
import MySQLdb

# DB ファイルへ接続
conn = MySQLdb.connect(
        user='user',
        passwd='tebasaki',
        host='localhost',
        db='mydb'
)
# カーソルの取得
cur = conn.cursor()
# データの取得
sql = 'select * from mytable'
cur.execute(sql)
for row in cur.fetchall():
    print('Id:', row[0], 'Name:', row[1], 'Price:', row[2])
# DB 接続を閉じる
conn.close()
```

列名を指定して情報の取得

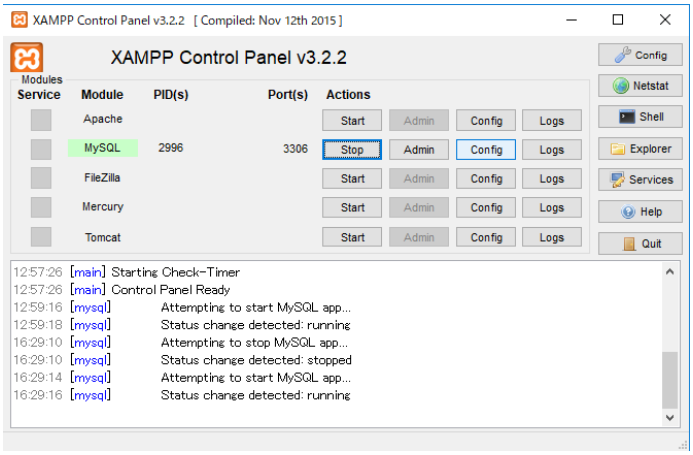
```
import MySQLdb

# DB ファイルへ接続
conn = MySQLdb.connect(
        user='user',
        passwd='tebasaki',
        host='localhost',
        db='mydb'
)
# カーソルの取得
cur = conn.cursor(MySQLdb.cursors.DictCursor)
# データの取得
sql = 'select * from mytable'
cur.execute(sql)
for row in cur.fetchall():
    print('Id:', row['id'], 'Name:', row['name'], 'Price:', row['price'])
# DB 接続を閉じる
conn.close()
```

MySQL データベース実習環境

MySQL データベース実習環境として「XAMPP」の MySQL を使用します。
「Apache Friends」のサイト（https://www.apachefriends.org/download.html）から、XAMPP for Windows をダウンードします。

データベースのサーバーが扱う文字コードを「UTF-8」に変更します。



（1）XAMPP Control Panel の MySQL の「Config」ボタンを押下し「my.ini」を選択する。
（2）my.ini 中の以下の行を探し
`#character_set_server=utf8`
先頭の「#」を取り除きファルを保存する。
（3）MySQL を起動（「Start」ボタンを押下）する。
（4）XAMPP Control Panel の「Shell」を起動し、コマンドプロンプトで「mysql -u root」と入力して MySQL に接続する。
（5）MySQL に対して「status」と入力して、サーバー文字コードが「utf8」になっていることを確認する。
MySQL のコンソールを抜けるには「exit」と入力する。

データベース接続ユーザーの作成

以下の作業は XAMPP Control Panel の「Shell」を起動し、コマンドプロンプトで「mysql -u root」と入力して MySQL に接続している状態で実行するものです。

SQL でユーザーの生成は CREATE USER 文を使用します。

```
CREATE　USER　'<ユーザー>'@'<接続元ホスト>'
 IDENTIFIED　BY　'<パスワード>'
```

例）DB が動作しているホストと同じホスト（localhost）からパスワード（tebasaki）で接続できるユーザー（user）を生成する

```
create user 'user'@'localhost' identified by 'tebasaki';
```

ユーザーに権限を与えるのはGRANT文を使用します。

```
GRANT ＜権限＞ ON　＜データベース＞.＜テーブル＞
 TO　'＜ユーザー＞'@'＜接続元ホスト＞'
```

例）DBが動作しているホストと同じホスト（localhost）から接続できるユーザー（taro）に対して、データベース（mydb）の全てのテーブル（＊）に対して、全ての権限（all）を与える。

```
grant all on mydb.* to 'taro'@'localhost';
```

データベース接続ユーザーの接続確認

XAMPP Control Panelの「Shell」ボタンで新しいコマンドプロンプトを開き、mysqlコマンドで生成したユーザーで接続できることを確認する。

```
mysql -u user -p
```

次に、演習で使用するデータベースを作成します。

```
create database mydb;
```

3章 オブジェクト指向

オブジェクト指向とはオブジェクトを中心に考える設計手法です。
オブジェクト指向以前の設計手法はデータ中心指向呼ばれており、これはデータ構造と手続き（処理）を別々に設計する手法です。オブジェクト指向はデータ構造と手続き（処理）をひとつの塊として設計します。
データ中心指向では処理を表現するためにフローチャートや PAD を使いましたが、オブジェクト指向では処理とデータ構想を一緒に表現できる図が必要となり、フローチャートの代わりに UML（Unified Modeling Lnaguage）を使用します。

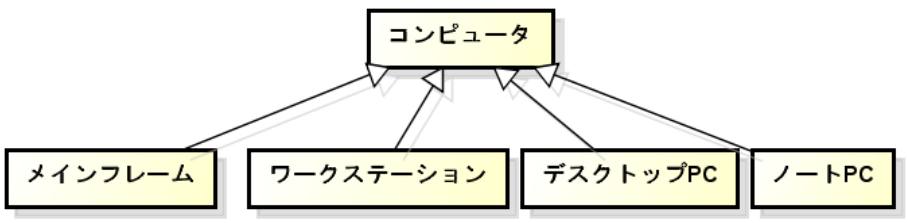
オブジェクト指向設計ではクラスが主役となります。
クラスはデータ構造（変数）と振る舞い（メソッド）を一つにまとめたものですが、それだけではなく「カプセル化」、「継承」、「ポリモーフィズム」の３つの機能を実現できることがオブジェクト指向には重要です。

カプセル化（隠蔽性、可視性）

カプセル化とは、クラスのメンバ（変数やメソッド）にアクセスできる段階をコントロールすることができることです。
ＵＭＬでは、オブジェクトのメンバの隠蔽レベルを private 可視性（－）、パッケージ可視性（～）、protected 可視性（＃）、public 可視性（＋）4 つの段階に分けて表現します。
Python ではメンバの名前の付け方よって、どこからでもアクセス可能（public 相当）と、クラス内からだけアクセス可能（private 相当）を実現しています。またプロパティ機能によって細かい制御も可能です。

継承

あるクラスをもとに、新たなクラスを作成することです。
もとになっているクラスのことをスーパークラスと呼び、新たに出来たクラスのことをサブクラスと呼びます。継承させることを「派生」させるとも言います。
言語によっては、スーパークラスのことを「基本クラス」や「基底クラス」などと呼び、サブクラスのことを「派生クラス」などとも呼びます。
ＵＭＬでは、「継承」のことを「汎化（Generalization）」と呼んでいます。
「汎化」は、サブクラスからスーパークラスに向かって延びる白抜き三角形の矢印で表記します。



サブクラスでは、スーパークラスのメンバが全て継承されます。
これは、サブクラスの側に一切操作を記述しなくても、単にサブクラスという関係にしただけで、自動的に、スーパークラスの操作が受け継がれるということです。
従って、サブクラス側では、スーパークラスには存在しない新しい機能（属性や操作）だけを記述すればよいことになり、開発効率が向上することになります。

オーバーライド

サブクラスは、スーパークラスのメンバが全て継承されると述べましたが、それだけでなくスーパークラスの操作の内容を、サブクラス側で変更することもできます。
これを操作のオーバーライド（Override）と呼びます。

Python では「継承」も「オーバーライド」の機能も実装されています。

ポリモーフィズム

ポリモーフィズムは、「多形態」「多態性」「多様性」などとも訳されます。
ポリモーフィズムとは、「スーパークラスは、サブクラスの代わりになることができる」という概念です。

具体的には、サブクラスのインスタンスをスーパークラスの変数に代入することができます。
このとき、その変数を使ってスーパークラスのメソッドを呼び出したとき、そのメソッドがサブクラスでオーバーライドされている場合、スーパークラスのメソッドでなく、インスタンスのサブクラスのメソッドが使用されます。

ソースコードではスーパークラスのメソット呼び出しをする記述をしておけば、サブクラスの種類を意識することなくインスタンスに応じた処理を実行することができるので、プログラムをシンプルに記述することができます。
ポリモーフィズムはサブクラスの数が多い場合にプログラムで処理をサブクラスの種類を意識せずに記述することができ、威力を発揮します。

厳密にいうと Python にはポリモーフィズムの機能はありません。
しかし Python の変数は型宣言が不要（動的型付）なため、サブクラスで作ったインスタンスでなくても変数に代入できます。これは非常に制限の緩いポリモーフィズムと考えることもできます。
継承の関係がなくても同じメソッド名の処理を持っているクラスであれば、ポリモーフィズムのような呼び出しが実装できるということです。

しかし、たとえスーパークラスを使用しなかったとしても、一つのスーパークラスから複数のサブクラスを作成するようにした方が良いでしょう。
それは、メソッドを明示的にオーバーライドさせておくことによって、実装忘れによる例外を抑止する効果がありますし、スーパークラスをインターフェースとしてプログラマの認識を合わせる役目をもたせる意味があります。
サブクラスでメソッドの実装を強制する抽象メソッドは Python の基本文法ではありませんが、abc モジュールを組み込むことによって実現可能です。

abc モジュールによる抽象メソッドの使い方は、
まず、以下のように abc モジュールから ABCMeta クラスと abstractmethod デコレータをインポートします。
抽象クラスとして定義したいクラスの引数として「metaclass = ABCMeta」を定義します。
次に抽象メソッドとして扱いたいメソッドに「@abstractmethod」デコレータを宣言します。
以上です。

```
from abc import ABCMeta,abstractmethod

class MyAbsClass(metaclass = ABCMeta):
    @abstractmethod
    def method_a(self):
        pass
    @abstractmethod
    def method_b(self):
        pass
```

この抽象クラスを継承してサブクラスを作成したとき、メソッドの実装がされていないと

```
class MyClass(MyAbsClass):
    def method_a(self):
        return "ABC"

a = MyClass()
```

実行時に以下のようなエラーとなります。
「TypeError: Can't instantiate abstract class MyClassA with abstract methods method_b」

４章　デザインパターンによる設計モデリング

デザインパターン

「デザインパターン」は、オブジェクト指向型のプログラミング設計（Design）の定石（Pattern)を体系化したものです。

当初、Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides の 4 人が 1995 年に 23 個のパターンを体系化しました。（当時は、まだ Java が登場したばかりだったため、C++で解説されています）
なお、この４人のことを、GoF（Gang of Four：四人組）と呼んでいます。

その後、何人かによって追加され、現在多くのパターンが知られていますが、公的機関に認定されなければいけないというような性質ものもではなく、各自が自由にパターンを考案してもまったくかまいません。

デザインパターンを習得するメリット

概念が共有できる

デザインパターンを使うと設計手法の概念を共有することが簡単になります。

デザインパターンには、さまざまありますが、例えば「今回のシステムでは、ファクトリ・メソッドとシングルトンを使って開発しようよ」などと言えば、自分の考えている設計手法について、簡単に相手に伝えることができるようになります。

もちろん、相手も、デザインパターンの名前とその内容を理解していなければなりませんが、今までは、このようなパターンの名前さえ統一されたものがなかったので、デザインパターンを覚えるということは、システム設計・開発を効率化するために大きく寄与することが期待できます。

基本的なスキルが向上する

デザインパターンの習得によって、設計者やプログラマのスキルが向上することです。

デザインパターンは「再利用性を高める」ためのパターンとか「交換性を高める」ためのパターン、そして「ソースコードの汚染を少なくする」パターンなどというように、バグが少なく高品質になるために工夫され開発されたものです。
従って、デザインパターンを習得することによって、デザインパターンを適切に使うことが増えてくれば、必然的にソフトウェアの品質も高まってきます。
デザインパターンにはオブジェクト指向的な発想や、設計・開発手法のエッセンスが凝縮されているので、習得するメリットは非常に大きいでしょう。

パターン一覧

GoF パターン（1995 年）

	パターン名	特徴	分類	Python
1	Abstract Factory	関連する部品を組み合わせる。	生成	
2	Adapter	一皮かぶせて再利用する。	構造	
3	Bridge	機能と実装の階層を分ける。	構造	
4	Builder	複雑なオブジェクトを組み立てる。	生成	
5	Chain of Responsibility	責任をたらい回しにする。	振舞い	
6	Command	命令をクラスにする。	振舞い	
7	Composite	容器と中身を同一視する。	構造	
8	Decorator	飾り枠と中身を同一視する。	構造	実装済
9	Façade	窓口をシンプルにする。	構造	
10	Factory Method	生成部と使用部を分離する。	生成	
11	Flyweight	同じものを共有し無駄をはぶく。	構造	
12	Interpreter	文法規則をクラスで表現する。	振舞い	
13	Iterator	ひとつひとつ順に取り出す。	振舞い	実装済
14	Mediator	複雑な相談は窓口を一本化する。	振舞い	
15	Memento	状態を保存する。何回でも UNDO。	振舞い	
16	Observer	状態の変化を観察し、通知する。	振舞い	
17	Prototype	コピーしてオブジェクトを生成する。	生成	
18	Proxy	必要になってからオブジェクト生成。	構造	
19	Singleton	唯一のオブジェクト生成を保証する。	生成	
20	State	状態をクラスとして表現する。	振舞い	
21	Strategy	ロジックをごっそり切り替える。	振舞い	
22	Template Method	処理をサブクラスにまかせる。	振舞い	
23	Visitor	構造を渡り歩きながら処理を行う。	振舞い	

Grand によるパターン（1998 年）の一部

	パターン名	特徴	分類	Python
1	Interface	実装部と使用部を分離する。	基本	abc モジュール
2	Delegation	メソッド呼び出しを委譲する。	基本	
3	Immutable	変更不可能なオブジェクト。	基本	実装済
4	Marker Interface	印付けのためだけのインタフェース。	基本	

Factory Method

あるサブクラスと別のサブクラスを必ずペアにして使う場合に便利なパターンです。

既存のクラスを繰り返し拡張して新しいサブクラスの種類を増やしても、必要なサブクラスの関連を意識しなくて済むのでプログラマの負担を減らすことができます。

「ファクトリ・メソッド・パターン」を利用しない例

```
from abc import ABCMeta,abstractmethod

class MyClassA(metaclass = ABCMeta):
    @abstractmethod
    def getName(self):
        pass
    @abstractmethod
    def getAge(self):
        pass

class MyClassB(metaclass = ABCMeta):
    @abstractmethod
    def getAddress(self):
        pass
    @abstractmethod
    def getFloor(self):
        pass

class MyClassA1(MyClassA):
    def getName(self):
        return "Yamada"
    def getAge(self):
        return 50

class MyClassB1(MyClassB):
    def getAddress(self):
        return "Nagoya"
    def getFloor(self):
        return 2


myA = MyClassA1()
myB = MyClassB1()
print(myA.getName(),myA.getAge(),myB.getAddress(),myB.getFloor())
```

上記のようなコードは、新たな条件が増えた場合や新たな種類のインスタンスを生成する必要が発生した場合にコードの汚染が発生しやすくなります。そこでインスタンスの生成部分を「Factory Method」を持ったクラスに任せるようにします。

「ファクトリ・メソッド・パターン」を利用した改善例

```
from abc import ABCMeta,abstractmethod

class MyClassA(metaclass = ABCMeta):
    @abstractmethod
    def getName(self):
        pass
    @abstractmethod
    def getAge(self):
        pass
    @abstractmethod
    def createMyClassB(self):
        pass

class MyClassB(metaclass = ABCMeta):
    @abstractmethod
    def getAddress(self):
        pass
    @abstractmethod
    def getFloor(self):
        pass

class MyClassA1(MyClassA):
    def getName(self):
        return "Yamada"
    def getAge(self):
        return 50
    def createMyClassB(self):
        return MyClassB1()

class MyClassB1(MyClassB):
    def getAddress(self):
        return "Nagoya"
    def getFloor(self):
        return 2

myA = MyClassA1()
myB = myA.createMyClassB()
print(myA.getName(),myA.getAge(),myB.getAddress(),myB.getFloor())
```

Abstract Factory Method

クライアントプログラムを修正することなく、一連のサブクラス群を実行環境の変更などに伴って交換することができるようにします。

Factory Method を進化させたもので、サブクラスのオブジェクトの生成を専門に引き受けるクラスを一つ用意してしまって、各サブクラスのファクトリメソッドをそのクラスにまとめて持たせることを考えます。

```
from abc import ABCMeta,abstractmethod

class MyFactory(metaclass = ABCMeta):
    @abstractmethod
    def getMyClassA(self):
        pass
    @abstractmethod
    def getMyClassB(self):
        pass

class MyClassA(metaclass = ABCMeta):
    @abstractmethod
    def getName(self):
        pass
    @abstractmethod
    def getAge(self):
        pass

class MyClassB(metaclass = ABCMeta):
    @abstractmethod
    def getAddress(self):
        pass
    @abstractmethod
    def getFloor(self):
        pass

class MyFactory1(MyFactory):
    def getMyClassA(self):
        return MyClassA1()
    def getMyClassB(self):
        return MyClassB1()

class MyClassA1(MyClassA):
    def getName(self):
        return "Yamada"
    def getAge(self):
        return 50
```

```
class MyClassB1(MyClassB):
    def getAddress(self):
        return "Nagoya"
    def getFloor(self):
        return 2

class MyFactory2(MyFactory):
    def getMyClassA(self):
        return MyClassA2()
    def getMyClassB(self):
        return MyClassB2()

class MyClassA2(MyClassA):
    def getName(self):
        return "Suzuki"
    def getAge(self):
        return 40

class MyClassB2(MyClassB):
    def getAddress(self):
        return "Tokyo"
    def getFloor(self):
        return 17

for myF in [ MyFactory1(),MyFactory2() ]:
    myA = myF.getMyClassA()
    myB = myF.getMyClassB()
    print(myA.getName(),myA.getAge(),myB.getAddress(),myB.getFloor())
```

Adapter

既存のクラスを修正することなく、適切なインターフェース（メソッド）を追加することができます。

既存のクラス MyPerson とそのクラスを閲覧する関数 MyViewer が存在するとします。

```
class MyPerson():
    def __init__(self,name):
        self.name = name
    def getName(self):
        return self.name

def MyViewer(person):
    print(person.getName())
```

また、同様に既存のクラス MyPersonX が存在するとします。

```
class MyPersonX():
    def __init__(self,firstname,lastname):
        self.firstname = firstname
        self.lastname = lastname
    def getFirstName(self):
        return self.firstname
    def getLastName(self):
        return self.lastname
```

それぞれのクラスは昔からある別システムで多くの開発者によって利用されているとします。

今回このシステムの結合することになり、クラスの違いを吸収する MyAdapter クラスを使って解決します。

```
class MyAdapter(MyPerson):
    def __init__(self,personX):
        self.psnX = personX
    def getName(self):
        return self.psnX.getFirstName()+" "+self.psnX.getLastName()

person = MyPerson("yamada atsuhiko")
MyViewer(person)

person = MyAdapter(MyPersonX("Yamada","Atsuhiko"))
MyViewer(person)
```