プログラムの書き方には手続き型とオブジェクト指向の2つがあります。皆さんが本書で学んできたプログラムは手続き型で書かれたものです。手続き型プログラミングで本格的なゲームを開発できることがお分かりいただけたと思いますが、大規模なプログラムを効率よく作るにはオブジェクト指向プログラミングの知識が役に立ちます。そこで本書では、最後にオブジェクト指向プログラミングについて説明します。

オブジェクト指向プログラミング

Chapter 13

Lesson 13-1

オブジェクト指向プログラミングについて

POINT

最初にお読みください

オブジェクト指向プログラミングは、Pythonに標準で備わったものです。この章では特別なモジュールは用いず、標準モジュールのみを使用します。Chapter 10～12で学んだPygameはオブジェクト指向プログラミングのために必要なモジュールではないことを、念のためお伝えしておきます。

Python、C/C++から派生したC系言語、Java、JavaScriptなど、ソフトウェア開発で広く使われているプログラミング言語はオブジェクト指向プログラミングをサポートしています。初めにオブジェクト指向とは、どのようなものなのかについて説明します。

オブジェクト指向プログラミングとは

オブジェクト指向プログラミングとは、複数のオブジェクトが係わり合う形でシステム全体を動かすという考え方です。オブジェクト指向プログラミングでは、**データ**（変数で扱う数値や文字列など）と**機能**（関数で定義した処理のこと）をひとまとめにした**クラス**というものを定義し、そのクラスから**オブジェクト**を作ります。そして複数のオブジェクトがデータをやり取りしたり、協調して処理を進めるようにプログラムを記述します。

なお、オブジェクトを**インスタンス**と呼ぶこともあります。インスタンスとは、クラスから作り出した**実体**という意味です。

クラスとオブジェクト

クラスとオブジェクトについて少し詳しく説明します。オブジェクト指向プログラミングでは、クラスから作ったオブジェクト（インスタンス）が処理を行うようにプログラムを記述します。クラスは機械の設計図、オブジェクトはその設計図から作った仕事をする機械に例えることができます。

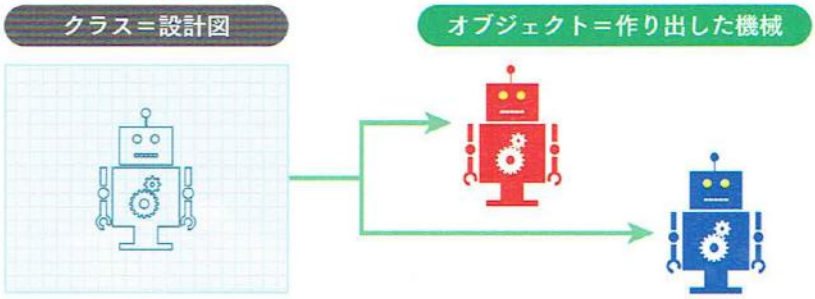
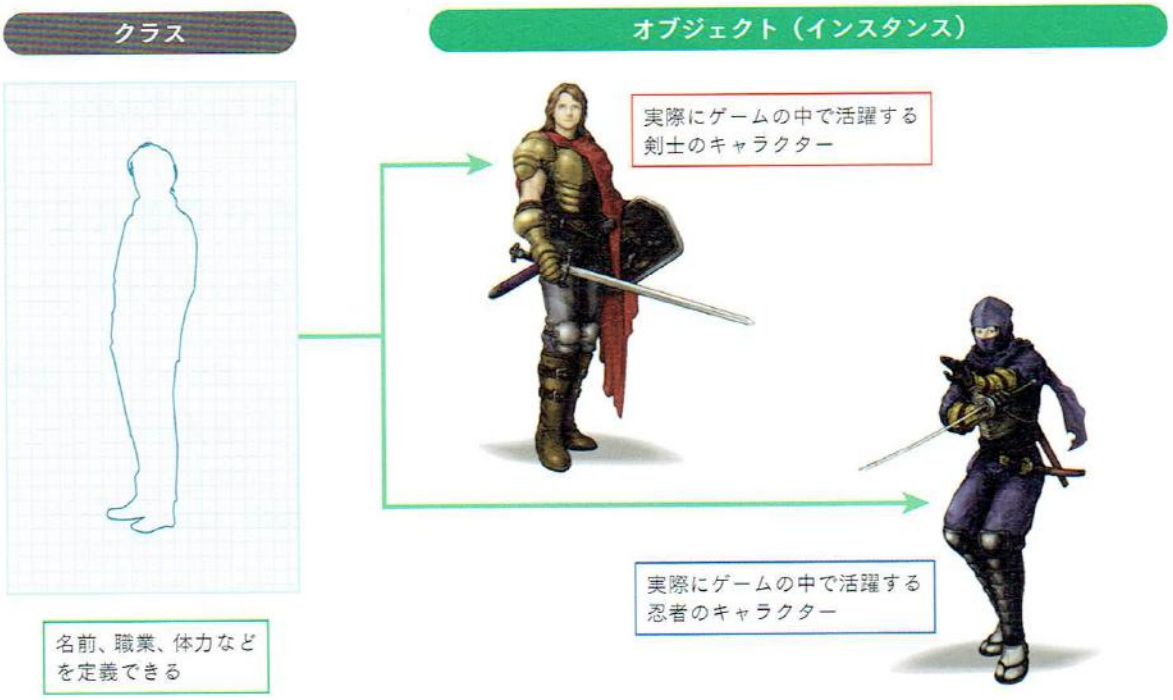


図13-1-1
クラスとオブジェクトの関係

あるいはゲームソフトをイメージして、次のように考えることができます。クラスはキャラクターを作り出す素になるもの、オブジェクトが実際に作られたキャラクターです。

図13-1-2　ゲームのキャラクターに例えると



言葉による説明だけでは、オブジェクト指向プログラミングをイメージすることは難しいかもしれません。次のLesson 13-2からクラスを定義したり、オブジェクトを作るプログラムを確認しながら、オブジェクト指向プログラミングを学んでいきましょう。

本書はゲーム開発の解説書なので、**キャラクターを作るクラスを題材にしてオブジェクト指向プログラミングを学ぶ**内容になっています。

なぜオブジェクト指向なのか？

プログラムの学習に入る前に、オブジェクト指向プログラミングがもてはやされる理由を説明しておきます。

最初に述べたように、プログラムの書き方は**手続き型**による記述と**オブジェクト指向**による記述に分かれます。規模の大きなプログラムになると、手続き型よりもオブジェクト指向で書くほうが処理の内容や流れが分かりやすくなります。分かりやすいということは、プログラムのメンテナンスや改良がしやすいということです。これがオブジェクト指向プログラミングの人気が高まった大きな理由です。

オブジェクト指向プログラミングは、大規模なソフトウェア開発に向いています。趣味レベルのプログラミングならオブジェクト指向の習得は必須ではありませんが、本職としてゲームプログラマーを目指す方は、オブジェクト指向について学んでおくに越したことはありません。

さて、オブジェクト指向という言葉を初めて聞く方や、「どこかで聞いたかな？」という程度の方もいると思います。それもそのはずです。日常生活ではオブジェクト指向という言葉は使いません。

本書以外でプログラミングを学んだことのある方の中には、オブジェクト指向という言葉は知っているけど理解するのは難しいと考える方もいるでしょう。確かにオブジェクト指向はプログラミングの学習の中で難易度の高い部類に位置付けられますが、この章では短い行数のサンプルプログラムでオブジェクト指向の基礎を解説するので、気楽に読み進めてください。

Lesson
13-2 クラスとオブジェクト

Pythonでクラスをどのように定義するか説明します。そして定義したクラスからオブジェクトを作り、動作を確認します。

クラスの作り方

クラスを定義する基本の書式を見てみましょう。次のようになります。

書式：クラスの定義

```
class クラス名:
    def __init__(self):
        self.変数名 = 初期値
```

Pythonでは「class クラス名」としてクラスを宣言します。そしてdef __init__(self)に、このクラスから作ったオブジェクトで使う変数を記述します。この変数は**属性**と呼ばれます。属性についてはP.329で改めて説明します。

def __init__(self)は**コンストラクタ**と呼ばれ、クラスの中に1つだけ記述する特別な関数のようなものです。コンストラクタに記述した処理は、クラスからオブジェクトを作る時に1回だけ実行されます。コンストラクタはクラスに必ず記述すべきものではなく、コンストラクタを設けないクラスを定義することもできます。

Pythonでは、クラス内に記述するコンストラクタや関数の引数に**self**を入れる決まりがあります。このselfはオブジェクトを作る時、そのオブジェクト自身を意味するものになります。

最初のうちはselfの意味が難しいと思いますので、「こう記述する」と考えておけばOKです。

クラスを定義し、オブジェクトを作る

4つのステップでクラスとオブジェクトを確認していきます。「ゲームのキャラクターを作り出す」というプログラムになります。

第1ステップはクラスを定義します。クラス名はGameCharacterとします。クラス名の頭文字は大文字にすることが推奨されるので、Gを大文字にしています。コンストラクタにはself以外に2つの引数、job（職業）とlife（ライフ）を追加しています。次のプログラムを入力し、ファイル名を付けて保存し、実行しましょう。

リスト▶list1302_1.py

```
1  class GameCharacter:
2      def __init__(self, job, life):
3          self.job = job
4          self.life = life
```

```
クラスの宣言
    コンストラクタ
        jobという属性に引数の値を代入
        lifeという属性に引数の値を代入
```

このプログラムを実行しても何も起きません。実行して何も起こらないことを確認しましょう。**クラスを定義しただけでは処理は行われない**のです。

第2ステップではクラスからオブジェクトを作ります。次のプログラムを入力し、ファイル名を付けて保存し、実行しましょう。

リスト▶list1302_2.py

```
1  class GameCharacter:
2      def __init__(self, job, life):
3          self.job = job
4          self.life = life
5
6  warrior = GameCharacter("戦士", 100)
7  print(warrior.job)
8  print(warrior.life)
```

```
クラスの宣言
    コンストラクタ
        jobという属性に引数の値を代入
        lifeという属性に引数の値を代入

warriorというオブジェクトを作る
warriorのjob属性の値を出力
warriorのlife属性の値を出力
```

このプログラムを実行すると、IDLEのシェルウィンドウに「戦士」と「100」が出力されます。

```
戦士
100
>>>
```

図13-2-1
list1302_2.pyの実行結果

ゲームの世界で活躍する戦士（オブジェクト）を、魔法の粘土のようなキャラクターの素になるもの（クラス）から作り出すことを想像すると分かりやすいと思いますが、いかがでしょうか？　1～4行目でキャラクターの素を定義し、6行目のwarriorがその素から作り出した実体のある戦士のイメージです。

このプログラムではwarriorという変数がオブジェクトになります。オブジェクトは次の書式で作ります。

書式：オブジェクトを作る

```
オブジェクト変数 = クラス名()
```

コンストラクタにself以外の引数を設けた場合

オブジェクト変数 = クラス名(引数)

list1302_2.pyに定義したクラスでは、コンストラクタにselfの他に2つの引数を記述し、3～4行目のように引数で受け取る値を**self.変数**に代入しています。この変数のことを**属性**といい、オブジェクトにデータを保持させるために用います。属性は7～8行目のように**オブジェクト変数.属性**と記述してデータの値を参照したり、新たなデータを代入します。

クラスを定義しただけでは何も起きませんでしたが、クラスからオブジェクト（インスタンス）を作り、そのオブジェクトに属性の値（ここでは職業とライフ）を設定することができました。**第3ステップでは属性の値の出力をオブジェクト自身が行う**ようにします。次のプログラムを入力し、ファイル名を付けて保存し、実行しましょう。

リスト▶list1302_3.py

```
1  class GameCharacter:
2      def __init__(self, job, life):
3          self.job = job
4          self.life = life
5
6      def info(self):
7          print(self.job)
8          print(self.life)
9
10 warrior = GameCharacter("戦士", 100)
11 warrior.info()
```

```
クラスの宣言
    コンストラクタ
        jobという属性に引数の値を代入
        lifeという属性に引数の値を代入

    属性の値を出力する関数(メソッド)
        job属性の値を出力
        life属性の値を出力

warriorというオブジェクトを作る
warriorのinfo()メソッドを実行
```

このプログラムを実行すると、前のプログラムと同様にシェルウィンドウに「戦士」と「100」が出力されます。実行画面は省略します。

動作は前のプログラムと一緒ですが、11行目のwarrior.info()で職業名とライフの値を出力しています。info()は6～8行目でクラス内に記述した関数です。クラス内に定義した関数を**メソッド**と呼びます。メソッドは11行目のように**オブジェクト変数.メソッド**と記述して実行します。オブジェクト指向プログラミングでは、このようにオブジェクトの機能をメソッドで定義します。

オブジェクト指向のプログラムは「オブジェクトに仕事をさせる」というイメージで捉えると分かりやすいことがあります。戦士のオブジェクトwarriorにinfo()命令を実行させたイメージを思い浮かべてみましょう。

第4ステップではクラスから複数のオブジェクトを作ります。次のプログラムを入力し、ファイル名を付けて保存し、実行しましょう。

リスト▶list1302_4.py

```
1  class GameCharacter:                          クラスの宣言
2      def __init__(self, job, life):               コンストラクタ
3          self.job = job                               jobという属性に引数の値を代入
4          self.life = life                             lifeという属性に引数の値を代入
5
6      def info(self):                              属性の値を出力する関数(メソッド)
7          print(self.job)                              job属性の値を出力
8          print(self.life)                             life属性の値を出力
9
10 human1 = GameCharacter("騎士", 120)          human1というオブジェクトを作る
11 human1.info()                                human1のinfo()メソッドを実行
12
13 human2 = GameCharacter("魔法使い", 80)       human2というオブジェクトを作る
14 human2.info()                                human2のinfo()メソッドを実行
```

このプログラムを実行すると、2つのオブジェクトの職業名とライフの値が出力されます。10行目のhuman1が騎士のオブジェクト、13行目のhuman2が魔法使いのオブジェクトです。

```
騎士
120
魔法使い
80
>>>
```

図13-2-2
list1302_4.pyの出力結果

Lesson 13-3

tkinterを使って オブジェクト指向を学ぶ

tkinterでウィンドウを表示し、画像を用いて、オブジェクト指向プログラムの動作を確認します。目に見える形でオブジェクトを確認することで、クラスやオブジェクトについての知識と理解を深めていきましょう。

tkinterを用いる

Lesson 13-2ではシェルウィンドウへの文字出力でオブジェクトを確認しましたが、次はtkinterを用いてウィンドウを表示し、クラスとオブジェクトを確認します。次の画像を用いるので、書籍サポートページからダウンロードして、プログラムと同じフォルダに入れてください。

次のプログラムを入力し、ファイル名を付けて保存し、実行しましょう。

リスト▶list1303_1.py

```
1  import tkinter
2  FNT = ("Times New Roman", 30)
3
4  class GameCharacter:
5      def __init__(self, job, life, imgfile):
6          self.job = job
7          self.life = life
8          self.img = tkinter.PhotoImage(file=imgfile)
9
10     def draw(self):
11         canvas.create_image(200, 280, image=self.img)
12         canvas.create_text(300, 400, text=self.job,
   font=FNT, fill="red")
13         canvas.create_text(300, 480, text=self.life,
   font=FNT, fill="blue")
14
15 root = tkinter.Tk()
16 root.title("tkinterでオブジェクト指向プログラミング")
17 canvas = tkinter.Canvas(root, width=400, height=560,
   bg="white")
18 canvas.pack()
```

tkinterモジュールをインポート
フォントを指定する変数

クラスの定義
　コンストラクタ
　　job属性に引数の値を代入
　　life属性に引数の値を代入
　　img属性に画像を読み込む

　画像と情報を表示するメソッド
　　画像の描画
　　文字列の表示(jobの値)

　　文字列の表示(lifeの値)

ウィンドウのオブジェクトを作る
タイトルを指定
キャンバスの部品を作る

キャンバスを配置する

```
19
20 character = GameCharacter("剣士", 200, "swordsman.png")  キャラクターのオブジェクトを作る
21 character.draw()                                        そのオブジェクトのdraw()メソッドを実行
22
23 root.mainloop()                                        ウィンドウを表示
```

このプログラムを実行すると、次のように剣士の画像と情報が表示されます。

図13-3-1　list1303_1.pyの実行結果

注意して見ていただきたいのがコンストラクタに記述した8行目の処理です。

```
self.img = tkinter.PhotoImage(file=
imgfile)
```

コンストラクタの引数で画像のファイル名を受け取り、self.imgに画像を読み込んでいます。**属性では数値や文字列だけでなく画像データを扱うこともできます**。

10～13行目に記述したdraw()メソッドを21行目で実行し、キャンバスに画像を表示していることも合わせて確認しましょう。

属性で画像を扱っているところがこのプログラムのポイントです。20行目で作ったcharacterというオブジェクトは、職業名、ライフの値、そして画像というデータを持っています。

複数のオブジェクトを作る

次はリストで複数のオブジェクトを作ります。swordsman.pngに加えて右の画像も用いるので、書籍サポートページからダウンロードして、プログラムと同じフォルダに入れてください。

ninja.png

次のプログラムを入力し、ファイル名を付けて保存し、実行しましょう。

リスト▶list1303_2.py

```
1  import tkinter                                                    tkinterモジュールをインポート
2  FNT = ("Times New Roman", 30)                                     フォントを指定する変数
3
4  class GameCharacter:                                              クラスの定義
5      def __init__(self, job, life, imgfile):                          コンストラクタ
6          self.job = job                                                   job属性に引数の値を代入
7          self.life = life                                                life属性に引数の値を代入
8          self.img = tkinter.PhotoImage(file=imgfile)                     img属性に画像を読み込む
9
10     def draw(self, x, y):                                            画像と情報を表示するメソッド
11         canvas.create_image(x+200, y+280, image=self.img)               画像の描画
12         canvas.create_text(x+300, y+400, text=self.                     文字列の表示(jobの値)
   job, font=FNT, fill="red")
13         canvas.create_text(x+300, y+480, text=self.                     文字列の表示(lifeの値)
   life, font=FNT, fill="blue")
14
15 root = tkinter.Tk()                                               ウィンドウのオブジェクトを作る
16 root.title("tkinterでオブジェクト指向プログラミング")                タイトルを指定
17 canvas = tkinter.Canvas(root, width=800, height=560,              キャンバスの部品を作る
   bg="white")
18 canvas.pack()                                                     キャンバスを配置する
19
20 character = [                                                     リストでオブジェクトを作る
21     GameCharacter("剣士", 200, "swordsman.png"),                     剣士のオブジェクト
22     GameCharacter("忍者", 160, "ninja.png")                          忍者のオブジェクト
23 ]
24 character[0].draw(0, 0)                                           剣士オブジェクトのdraw()メソッドを実行
25 character[1].draw(400, 0)                                         忍者オブジェクトのdraw()メソッドを実行
26
27 root.mainloop()                                                  ウィンドウを表示
```

このプログラムを実行すると剣士と忍者が表示されます。

基本的な処理は前のlist1303_1.pyと一緒ですが、今回のプログラムではdraw()メソッドにdraw(self, x, y)と2つの引数を追加し、画像と文字列の表示位置を指定できるようにしています。

20～23行目のように、2

図13-3-2　list1303_2.pyの実行結果

つのオブジェクトをリストで用意します。24～25行目でそれぞれのオブジェクトのdraw()メソッドを実行しています。

機能を定義し戦えるようにする

クラスに定義するメソッドは、オブジェクトに機能を持たせるためのものです。list1303_1.pyとlist1303_2.pyの2つのプログラムは、クラスにdraw()というメソッドを定義しています。オブジェクト指向プログラミングでゲームを開発するのであれば、例えば

- キャラクターを移動させる機能を持つメソッド
- キャラクターのライフの計算を行うメソッド

などを用意します。これらの機能に必要な変数（属性）があれば、コンストラクタにその変数を加えます。

オブジェクトの機能について学ぶために、前のlist1303_2.pyに相手のキャラクターと戦うメソッドを追加し、ゲームに近い動作をするプログラムに改良します。このプログラムでは前のプログラムまで使っていた職業名を入れる変数jobを、キャラクター名を入れる変数nameに変更しました。

次のプログラムを入力し、ファイル名を付けて保存し、実行しましょう。

リスト▶list1303_3.py

```
1  import tkinter                                          tkinterモジュールをインポート
2  import time                                             timeモジュールをインポート
3  FNT = ("Times New Roman", 24)                           フォントを指定する変数
4
5  class GameCharacter:                                    クラスの定義
6      def __init__(self, name, life, x, y, imgfile,           コンストラクタ
   tagname):
7          self.name = name                                    name属性に引数の値を代入
8          self.life = life                                    life属性に引数の値を代入
9          self.lmax = life                                    lmax属性に引数の値を代入
10         self.x = x                                          x属性に引数の値を代入
11         self.y = y                                          y属性に引数の値を代入
12         self.img = tkinter.PhotoImage(file=imgfile)         img属性に画像を読み込む
13         self.tagname = tagname                              tagname属性に引数の値を代入
14
15     def draw(self):                                     画像と情報を表示するメソッド
16         x = self.x                                          変数xに表示位置(X座標)を代入
17         y = self.y                                          変数yに表示位置(Y座標)を代入
18         canvas.create_image(x, y, image=self.img,           画像の描画
   tag=self.tagname)
19         canvas.create_text(x, y+120, text=self.name,        文字列の表示(nameの値)
   font=FNT, fill="red", tag=self.tagname)
20         canvas.create_text(x, y+200, text="life{}/          文字列の表示(lifeとlmaxの値)
```

```
    {}".format(self.life, self.lmax), font=FNT, fill=
    "lime", tag=self.tagname)
21
22      def attack(self):                                   攻撃処理を行うメソッド
23          di = 1                                              画像を動かす向き
24          if self.x >= 400:                                       右側のキャラは
25              di = -1                                                 動かす向きを左とする
26          for i in range(5): # 攻撃動作(横に動かす)           繰り返しで
27              canvas.coords(self.tagname, self.x+i*               coords()命令で表示位置を変更
    10*di, self.y)
28              canvas.update()                                     キャンバスを更新
29              time.sleep(0.1)                                     0.1秒待つ
30          canvas.coords(self.tagname, self.x, self.y)         画像を元に位置に移動
31
32      def damage(self):                                   ダメージを受ける処理を行うメソッド
33          for i in range(5): # ダメージ(画像の点滅)           繰り返しで
34              self.draw()                                         キャラを表示するメソッドを実行
35              canvas.update()                                     キャンバスを更新
36              time.sleep(0.1)                                     0.1秒待つ
37              canvas.delete(self.tagname)                         画像を削除(一旦消す)
38              canvas.update()                                     キャンバスを更新
39              time.sleep(0.1)                                     0.1秒待つ
40          self.life = self.life - 30                          ライフを30減らす
41          if self.life > 0:                                   ライフが0より大きければ
42              self.draw()                                         キャラを表示する
43          else:                                               そうでなければ
44              print(self.name+"は倒れた...")                      倒れたとシェルウィンドウに出力
45
46  def click_left():                                       左側のボタンをクリックした時の関数
47      character[0].attack()                                   剣士の攻撃処理のメソッドを実行
48      character[1].damage()                                   忍者のダメージ処理のメソッドを実行
49
50  def click_right():                                      右側のボタンをクリックした時の関数
51      character[1].attack()                                   忍者の攻撃処理のメソッドを実行
52      character[0].damage()                                   剣士のダメージ処理のメソッドを実行
53
54  root = tkinter.Tk()                                     ウィンドウのオブジェクトを作る
55  root.title("オブジェクト指向でバトル")                  タイトルを指定
56  canvas = tkinter.Canvas(root, width=800,                キャンバスの部品を作る
    height=600, bg="white")
57  canvas.pack()                                           キャンバスを配置する
58
59  btn_left = tkinter.Button(text="攻撃→", command=        左側のボタンを作り
    click_left)
60  btn_left.place(x=160, y=560)                            配置する
61  btn_right = tkinter.Button(text="←攻撃", command=       右側のボタンを作り
    click_right)
62  btn_right.place(x=560, y=560)                           配置する
63
```

```
64  character = [                                    リストでオブジェクトを作る
65      GameCharacter("暁の剣士「ガイア」", 200, 200, 280,    剣士のオブジェクト
    "swordsman.png", "LC"),
66      GameCharacter("闇の忍者「半蔵」", 160, 600, 280,      忍者のオブジェクト
    "ninja.png", "RC")
67  ]
68  character[0].draw()                              剣士オブジェクトのdraw()メソッドを実行
69  character[1].draw()                              忍者オブジェクトのdraw()メソッドを実行
70
71  root.mainloop()                                  ウィンドウを表示
```

このプログラムを実行すると、剣士と忍者の足元に攻撃ボタンが表示され、それをクリックすると相手を攻撃します。ライフが0以下になると画像が消え、シェルウィンドウに「○○は倒れた……」と出力されます。

図13-3-3
list1303_3.pyの
実行結果

GameCharacterクラスに定義したコンストラクタとメソッドの処理を説明します。

表13-3-1　GameCharacterクラスのコンストラクタとメソッド

行番号	コンストラクタ/メソッド	処理
6～13行目	コンストラクタ __init__()	キャラクターの名前、ライフ、表示位置、画像ファイル名、タグ名を引数で受け取り、変数(属性)に代入する。
15～20行目	draw()メソッド	キャラクターの画像、名前とライフを表示する。
22～30行目	attack()メソッド	画像を左右に移動させ、相手を攻撃する演出を行う。
32～44行目	damage()メソッド	画像を点滅させ、ダメージを受ける演出を行う。ライフの値を減らし、ライフが残っていれば画像を再表示し、0以下になった場合はシェルウィンドウに負けた旨を出力する。

59行目のボタンを作る記述で、剣士の足元のボタンをクリックするとclick_left()関数を呼び出すようにしています。この関数は47～48行目のように、剣士のオブジェクトが攻撃するメソッド、忍者のオブジェクトがダメージを受けるメソッドを実行します。忍者の足元のボタンも同様で、クリックすると忍者が攻撃するメソッドと剣士がダメージを受けるメソッドを実行するようにしています。

画像の表示演出では、一定時間、処理を止めるために**time.sleep()**命令を用いています(29、36、39行目)。この命令を使うには**timeモジュール**をインポートし、sleep()の引数で何秒間、処理を停止するか指定します。

剣士と忍者の画像を、移動や点滅させる処理でタグ名を使っています。リストでオブジェクトを作る際、65～66行目のようにタグ名を引数で渡し(剣士の画像はLC、忍者の画像はRCというタグ名)、オブジェクトの属性でそのタグ名を保持しています。

このプログラムを改良していけば、本格的なゲームが作れそうです。オブジェクト指向プログラミングを学ぼうという方は、ゲーム作りに挑戦することでも知識を深められると思います。

Lesson 13-4

オブジェクト指向プログラミングをもっと学ぶ

Lesson 13-1から13-3まで、オブジェクト指向プログラミングの基礎知識を説明しました。オブジェクト指向プログラミングをさらに学びたい方のために、クラスの継承とオーバーライドという2つの知識を説明します。難しい内容ですが、頑張って読み進めてください。

クラスの継承について

オブジェクト指向プログラミングでは、あるクラスを元に、新たな機能を加えた新しいクラスを作ることができます。これをクラスの**継承**といいます。

ロールプレイングゲームを例に、クラスの継承を説明します。まずキャラクターの名前とライフを扱うための基本的なクラスを用意します。これをCという名のクラスとしましょう。クラスCを元に、パーティメンバーを作り出すためのPというクラスと、敵のモンスターを作り出すためのMというクラスを用意します。

パーティメンバーは武器と防具を装備するので、クラスPには装備品を管理するweaponとarmorという変数を追加します。モンスターはゲームでよく使われる六大要素の火、風、土、水、闇、光のいずれかに属するものとし、クラスMにはどの要素に属するかを管理する変数elementを追加します。これを図示すると次のようになります。

図13-4-1　クラスの継承

継承元となるクラスを親クラスやスーパークラスといいます。親クラスを継承して作ったクラスを子クラスやサブクラスといいます。以上のように、あるクラスを元に新たなクラスを作ることが継承の概念です。

Pythonでは次の書式で親クラスを継承した子クラスを作ります。

書式：子クラスを作る

```
class 子クラス名(親クラス名):
    子クラスの定義内容
```

オーバーライドについて

子クラスでは、親クラスのコンストラクタやメソッドを上書きして機能を充実させることができます。コンストラクタやメソッドを上書きすることを**オーバーライド**といいます。

継承とオーバーライドをプログラムで確認します。プログラムの内容はロールプレイングゲームをイメージし、一般人（町の人々）を作るためのクラスを定義します。そして冒険に出る戦士を作るためのクラスを、一般人のクラスを継承して定義します。一般人のクラスが親クラス、戦士のクラスが子クラスです。戦士クラスではコンストラクタとメソッドをオーバーライドし、機能を充実させます。

画像は用いません。次のプログラムを入力し、ファイル名を付けて保存し、実行しましょう。

リスト▶list1304_1.py

```
1  class Human:                                          Humanクラスの定義（これが親クラス）
2      def __init__(self, name, life):                       コンストラクタ
3          self.name = name                                      name属性に引数の値を代入
4          self.life = life                                      life属性に引数の値を代入
5
6      def info(self):                                       属性の値を出力するメソッド
7          print(self.name)                                      name属性の値を出力
8          print(self.life)                                      life属性の値を出力
9
10
11 class Soldier(Human):                                  Humanクラスを継承しSoldierクラスを定義
12     def __init__(self, name, life, weapon):               コンストラクタをオーバーライド
13         super().__init__(name, life)                          親クラスのコンストラクタを実行
14         self.weapon = weapon                                  weapon属性に引数の値を代入
15
16     def info(self):                                       info()メソッドをオーバーライド
17         print("私の名前は"+self.name)                          文字列とname属性の出力
18         print("私の体力は{}".format(self.life))                文字列とlife属性の出力
19
20     def talk(self):                                       Soldierクラスで新たに定義したメソッド
21         print(self.weapon + "を携え、冒険に出発します")          台詞を出力
22
23
24 man = Human("トム(一般人)", 50)                          一般人のオブジェクトを作る
25 man.info()                                             そのオブジェクトのinfo()メソッドを実行
26 print("----------")                                     出力内容を----------で区切る
27 prince = Soldier("アレクス(王子)", 200, "光の剣")          戦士のオブジェクトを作る
28 prince.info()                                          そのオブジェクトのinfo()メソッドを実行
29 prince.talk()                                          そのオブジェクトのtalk()メソッドを実行
```

このプログラムを実行すると次のように出力されます。

```
トム(一般人)
50
----------
私の名前はアレクス(王子)
私の体力は200
光の剣を携え、冒険に出発します
>>>
```

図13-4-2
list1304_1.pyの実行結果

11～21行目に記述したSoldierクラスが、1～8行目のHumanクラスを継承して作ったクラスです。Soldierクラスではコンストラクタをdef __init__(self, name, life, weapon)と記述してオーバーライドし、武器の名称を引数で渡せるようにしています。またnameとlifeの値を代入するために、13行目のようにsuper().__init__(name, life)として親クラスのコンストラクタを実行します。super()は「スーパークラス（親クラス）の」という意味です。

Soldierクラスではinfo()メソッドもオーバーライドしているので、24行目で作ったトムのオブジェクトでinfo()メソッドを実行した結果と、27行目で作ったアレクスのオブジェクトでinfo()メソッドを実行した結果が違います。アレクスのinfo()メソッドは「私の名前は○」「私の体力は□」と出力する機能を持つように上書きされています。

それからSoldierクラスにはtalk()メソッドを追加しています。親クラスを継承して作る子クラスでは、このように機能を充実させていくことが一般的です。

以上が継承とオーバーライドの学習ですが、難しい内容なのですぐに理解できなくても大丈夫です。オブジェクト指向プログラミングは一朝一夕に習得できるものではないので、焦らずにコツコツと学習を進めていただければと思います。

COLUMN

筆者も苦労したオブジェクト指向プログラミング

筆者（以下、私）がプログラミングを学び始めた1980年代、世の中にはまだオブジェクト指向プログラミングは浸透していませんでした。当時学生だった私が読むコンピュータ関連の書物には、記憶にある限り、オブジェクト指向の話は出てこなかったと思います。ただ私はホビー（ゲーム）寄りのコンピュータ誌ばかり読んでいたので、ビジネス寄りのコンピュータ雑誌や書物にはオブジェクト指向の話が出ていたかもしれません。いずれにしても当時はまだオブジェクト指向必須という時代ではありませんでした。

1990年代になると、コンピュータの急速な進歩とインターネットの普及により、大規模なソフトウェア開発が多く行われるようになります。そのような開発の場では複数のプログラマーが共同作業をしますが、オブジェクト指向でプログラミングすると、高度なプログラムを作業分担して作りやすいのです。またオブジェクト指向のプログラミング言語には、ソフトウェアの不具合の発生を抑制できる仕組みが備わっています。そのような理由から、90年代以降、Javaなどのオブジェクト指向を前提に設計されたプログラミング言語が普及していきました。

オブジェクト指向プログラミングは難しいと感じる方も多いことでしょう。オブジェクト指向を学び始めた当初、私にとってもその概念は難しく、「どうもイメージがつかめない……」という悩ましい日々が続きました。ある日、「これまで自分が書いてきた手続き型のプログラムをオブジェクト指向にするには、どう記述すればよいのだろう？」と考え、それがオブジェクト指向を理解する突破口になりました。手続き型のプログラムの一部をオブジェクト指向の書き方に置き換えていくことで、オブジェクト指向プログラミングへの理解が進んだのです。

プログラミング初心者や趣味のプログラマーの方は、オブジェクト指向に頭を悩ませる必要はありません。手続き型のプログラムでゲームを作ることができるのですから、この章の内容は全体を眺めれば十分です。

将来ゲームプログラマーとして活躍したい方は、ある程度プログラミングの技術が身についたら、ぜひオブジェクト指向プログラミングにも挑戦してください。この章のサンプルプログラムを改良するところからスタートしてもよいでしょう。サンプルプログラムに手を加えることも、オブジェクト指向プログラミングを理解する方法の1つになると思います。継承やオーバーライドは難しい概念なので、すぐに理解できなくても大丈夫です。プログラミングの学習を続けていけばオリジナルゲームを作れるようになりますし、やがてオブジェクト指向プログラミングも理解できるはずです。

みなさんはたくさんの知識を学びました。きっとオリジナルのゲームが作れようなります。頑張ってください。

みなさん、またどこかでお会いしましょう。
お疲れさまでした！