**Robots**

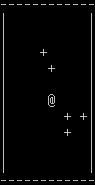
提出日　2018年6月19日

15-412　及川 裕公

1. Robotsとは

古くからUNIXオペレーティングシステムに付属してきた由緒正しきゲームである。

　プレイヤーは悪いロボットと戦っている。戦っている様子は以下の図１のようなゲームフィールドに、プレイヤーが’ @ ’、ロボットが’ + ‘で表されている。



凡例:　‘ @ ’...プレイヤー、’ + ’...ロボット、’ - ‘, ’ | ’...壁

図 1　Robotsのゲームボードのイメージ図

1.1　プレイヤーの操作

　プレイヤーは1回の動作で、上下左右と斜めの8方向、同じ場所で待機、ランダムな場所にテレポートの合計10種類の動作が選択可能である。

　ただし、以下の3つの条件のいずれかが当てはまる地点には、移動することができない。これはテレポートも同様で、以下の条件に当てはまらない場所からランダムに移動となる。

* フィールドの外
* ロボットがすでにいる場所
* スクラップがすでにある場所

プレイヤーが移動した座標に、ロボットが重なったらゲームオーバーである。

プレイヤーの操作は動く方向を0～9の数字で入力する。この数字と方向は、プレイヤーがテンキーの5の位置にいたと仮定したときの向きにそれぞれ対応している。

0: テレポート

1: 左下

2: 下

3: 右下

4: 左

5: その場で待機

6: 右

7: 左上

8: 上

9: 右上

* 1. ロボットの動き

ロボットはプレイヤーの移動後に行動を開始する。

　ロボットは、プレイヤーが移動した後の座標を基に、最接近するように行動してくる。

　そして、ロボット同士が衝突すると、ロボットが壊れスクラップとなる。また、ロボットはスクラップに衝突しても壊れてしまう。スクラップは’ \* ’で表される。

　すべてのロボットをスクラップにすると、ステージクリアとなり、次のステージが現れる。

* 1. ゲームの設定

　ゲーム開始時のレベルは1である。ステージクリアするごとに、レベルが1ずつ上がっていく。

　フィールドに設置されるロボットの数は、

Min (レベル \* 5, 40)

とする。ただし、Min (x,y)はx, yのうち、小さい方を取る関数である。すなわち、レベル8以降のロボット出現数は40固定となる。

　ロボットを1台スクラップにするごとに1点が与えられる。また、ステージクリアごとにレベル \* 10点のボーナスが与えられる。

　例えば、ステージ1をクリアすると、

Min (1 \* 5, 40) + 1 \* 10 = 5 + 10 = 15

となり、15点を得られることがわかる。

1. Robotsの使用方法

　今回作成したRobotsは、ターミナル上でソースコード「robots.py」があるディレクトリまで行き、「$python robots.py」と打ち込んで起動する。

　起動したと同時にゲームが始まるので、1.1で記したキーを入力することでプレイヤーが行動をする。また、操作がわからなくなったら、hキーを打つことでヘルプを閲覧することができる。

1. Robotsの機能

今回作成したRobotsは、ターミナルから実行することで遊ぶことができる、CLIアプリケーションである。コンティニュー機能を実装したことにより、何度も開く手間がなく遊び続けることができる。

1. Robotsのデータ構造

4.1　Gameクラス

　Gameクラスは、ゲームに関する情報をすべて管理し、このプログラムの中核とも言えるクラスである。Gameクラスは以下の要素で構成される。

表 1　Boardクラスのデータ構造

|  |  |
| --- | --- |
| 変数名・メソッド名 | 機能 |
| height | ゲームフィールドの高さ |
| width | ゲームフィールドの幅 |
| robot\_reft | 非スクラップ状態のロボットの数 |
| board | ゲームフィールド |
| score | 現在の得点 |
| level | 現在のレベル |
| MAX\_ROBOT ※定数 | フィールドに配置できるロボットの最大値 |
| setting() | ゲームフィールドの初期状態を生成する |
| show() | ゲームフィールドの情報を表示 |
| action() | ゲームフィールドを更新 |
| teleport() | プレイヤーがテレポートする座標の算出 |

* 1. Objectクラス

　ObjectクラスはGameクラスのboardの中に配置されるオブジェクトが継承しているクラスである。現在どの座標にいるのかを保持している。

* 1. MoveObjectクラス

　Object クラスを継承して作成したクラス。座標の他に、moveメソッドを持ち、引数として指定した座標に移動することができるようになっている。

* 1. Playerクラス

　MoveObjectを継承して作成したクラス。Moveメソッドに移動の制約が追加され、移動できたか否かを返り値として持てるように変更してある。

ObjectクラスとMoveobjectとPlayerクラスの要素を以下に示す。

表 2　Objectクラス、MoveObjectクラス、Playerクラスのデータ構造

|  |  |
| --- | --- |
| 変数名・メソッド名 | 詳細 |
| x | オブジェクトが持つx座標 |
| y | オブジェクトが特つy座標 |
| Objectクラスは上記の要素を持つ | |
| move() | x, yを指定した座標に動かす |
| MoveObjectクラスは上記までの要素を持つ | |
| Playerクラスは、moveメソッドをオーバーライドしているだけのため、データ構造上は変わらない。 | |

* 1. Robotクラス

　MoveObjectを継承して作成したクラス。sclapという変数を加え、自分自身がスクラップ状態か否かを持てるようにしてある。また、Moveメソッドをオーバーライドし、ロボット同士が衝突したかどうかの判定などもできるようになっている。それに伴い、ロボットの状態をスクラップ状態にするkillメソッドも追加した。

　以下に、Robotクラスのデータ構造を示す。

表 3　Robotクラスのデータ構造

|  |  |
| --- | --- |
| 変数名・メソッド名 | 詳細 |
| MoveObjectクラスを継承しているため、基本は表2と同じである。ここでは、変わった部分のみを示す。 | |
| scrap | 自分がスクラップであるかを持つ |
| kill() | ロボットをスクラップにする |

* 1. Getchクラス

　Getchクラスは、Enterキーの入力無しで入力処理を行えるようにしたクラスである。

　以下にGetchクラスのデータ構造を示す。

表 4　Getchクラスのデータ構造

|  |  |
| --- | --- |
| 変数名・メソッド名 | 詳細 |
| impl | 呼ばれたときに行う処理 |
| call() | OSに合わせた1文字入力の処理 |

1. 主要なクラス・関数の仕様

　以下に、主要なクラス・関数の仕様を示す。その時、変数名が()で囲われているものは、省略可能であることを示している。

5.1　Boardクラス

5.1.1　コンストラクタ

・引数:(level)…ゲーム開始時のレベル、(height)…ゲームフィールドの高さ、(width)…ゲームフィールドの幅

・機能:ゲームに関する初期設定を行う。

・返り値:なし

5.1.2　board\_elementメソッド

・引数:pos…ゲームフィールド上の座標

・機能:座標を渡すと、その座標上にいるオブジェクトを返す。

・返り値:座標にいたオブジェクト

5.1.3 　settingメソッド

・引数:なし

・機能:ゲームフィールドの初期状態を生成する。

・返り値:なし

5.1.4　showメソッド

・引数:なし

・機能:ゲームフィールドを表示する。

・返り値:なし

5.1.5　teleportメソッド

・引数:なし

・機能:Objectがないマス(移動できないマス)を除いた場所からランダムな座標を選択する

・返り値:Objectがないランダムなマス

5.1.6　actionメソッド

・引数:command…どの数字が入力されたか

・機能:入力されたコマンドに対応した状態を反映させる

・返り値:何かしらのイベントがあるか否かをTrue, Falseで返す

* 1. read\_command関数

・引数:game\_master…Gameクラスのインスタンス

・機能:Enterキー入力無しで、コマンドを読み取る。その際、コマンドが移動以外のコマンドなら弾く。

・返り値:移動するコマンド(0～9のいずれか)

* 1. Playerクラス
     1. moveメソッド

・引数:x…移動先のx座標または、y…移動先のy座標、game\_master…Gameクラスのインスタンス、(absolute)…x,yが絶対指定か相対指定か

・機能:プレイヤーが動ける座標に移動しようとしているなら移動、動けない座標なら移動しない。また、その結果を返す。

・返り値:プレイヤーは正常に移動できたかをTrue, Falseで返す。

1. 検証

　このプログラムでは、通常終了されるものを除き、ほとんどの入力に耐えられるように設計を行った。検証した内容は、ゲーム中移動する時にどのような入力に耐えられるかである。以下に試した内容とその結果をまとめた表を示す。

表 5　様々な入力とその応答結果

|  |  |
| --- | --- |
| 入力した内容 | 結果 |
| h以外のアルファベットを入力(z) | Error. Input is integer. |
| 記号などを入力(@) | Error. Input is integer. |
| h | ヘルプを表示 |
| 座標外の入力(-1,-1に行くような入力) | Move error. Now pos is (-1, -1) |
| c-c(KeyboardInterrupt)を入力 | 通常、終了すべきなので、そのまま終了。 |
| c-d(EOF)を入力 | 通常、終了すべきなので、そのまま終了 |

また、ゲームオーバーの処理が正常に行われていること、レベルアップの処理が正常に行われていること、レベル8以降に登場するロボットの数は40固定になっていたことも確認した。

1. Robotsの計算量とメモリ使用量

　ここでは、Robotsのメインとなる処理にかかる時間と消費するメモリ量について考える。これ以下に出てくるn, mについてはそれぞれ、ロボットの数、フィールドサイズ(マス目の総数)に対応する。また、m\_widthならフィールドの幅、m\_heightなら高さに対応する。

7.1　プレイヤーの移動

　プレイヤーの移動は、テレポートを含む場合と含まない場合で大きく変わってくる。

7.1.1　テレポートをする場合

　テレポートをする場合、テレポート先の座標を算出するために、teleport関数内でO(m)の処理が発生し、move関数内でO(1)の処理が発生する。よって、テレポートをする場合のオーダーはO(m)であると言える。

　メモリ使用量は、teleport関数内でO(m-n)のメモリを消費し、move関数内でO(m\_height + m\_width)のメモリを消費している。

7.1.2　テレポートをしない場合

　テレポートをしない場合、テレポート先の座標は算出しないため、move関数内でO(1)の処理が発生するだけである。そのため、テレポートしない場合のオーダーはO(1)であるといえる。

* 1. ロボットの移動

　ロボットの移動は、ゲームフィールドを左上から順に参照していき、ロボットがいるなら動かすという処理をしている。そのため、ロボットの探索でO(m)の処理が発生する。

　ロボットのmove関数では、移動するべき座標を求めるのにO(1)の処理を、衝突の判定をするのにO(1)の処理をしているため、結果的にmove関数内ではO(1)の処理をしていると言える。よって、ロボットの移動のオーダーはO(m)であるといえる。

* 1. ロボットの衝突

　ロボットの衝突は7.2でも述べたようにO(1)の処理を行っている。よって、ロボットの衝突のオーダーはO(1)であるといえる。

1. プログラム

　今回作成したRobotsを次頁から示す。

リスト 1　robots.pyのソースコード

|  |
| --- |
| import random  # タプルの計算をするプログラム  # 初めに入った数からopeに従い計算していき、結果のタプルを返す。  def tuple\_calc(\*tuples, ope):  ans\_list = [0] \* len(tuples[0])  first\_flg = True  for tp in tuples:  for i in range(len(tp)):  if ope == 'add' or first\_flg == True: #初めの数から後ろを引いていくための準備  ans\_list[i] += tp[i]  elif ope == 'sub':  ans\_list[i] -= tp[i]  first\_flg = False  return tuple(ans\_list)  # Enter入力無しで入力処理を行うクラス。  # Win版とUNIX版の2種類が用意されていて、import状況に応じて自動的に分岐される。  class \_Getch:  """Gets a single character from standard input. Does not echo to the  screen."""  def \_\_init\_\_(self):  try:  self.impl = \_GetchWindows()  except ImportError:  self.impl = \_GetchUnix()  #この時、得られるのはb(バイト列)である。  #そのため、普通の文字として扱うため、.decode()を書いている。  def \_\_call\_\_(self): return self.impl()  # UNIX版  # stdinから1文字読み込んで、読み込んだ文字を返す。  class \_GetchUnix:  def \_\_init\_\_(self):  import tty, sys  def \_\_call\_\_(self):  import sys, tty, termios  fd = sys.stdin.fileno()  old\_settings = termios.tcgetattr(fd)  try:  tty.setraw(sys.stdin.fileno())  ch = sys.stdin.read(1)  finally:  termios.tcsetattr(fd, termios.TCSADRAIN, old\_settings)  return ch  # Win版  # msvcrt内蔵の関数を呼出して、その値を返すだけ。  class \_GetchWindows:  def \_\_init\_\_(self):  import msvcrt  def \_\_call\_\_(self):  import msvcrt  return msvcrt.getch().decode()  # ゲームボードの上に存在するオブジェクトは必ずこのクラスを継承して作る  # オブジェクトはx,yの座標を持っている。  class Object():  def \_\_init\_\_(self, x = -1, y = -1):  self.\_x, self.\_y = x, y  @property  def position(self):  return self.\_x, self.\_y  @position.setter  def position(self, pos):  if len(pos) != 2:  raise "test"  (x, y) = pos  self.\_x, self.\_y = x, y  @property  def x(self):  return self.\_x  @x.setter  def x(self, x):  self.\_x = x  @property  def y(self):  return self.\_y  @y.setter  def y(self, y):  self.\_y = y  # ゲームボード上を動き回るクラスはこのクラスを継承する。  class MoveObject(Object):  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y)  def move(self, x, y):  self.\_x = x  self.\_y = y  # MoveObjectを継承したPlayerクラス  # 移動するときに、オブジェクトが存在していたら動けない。  class Player(MoveObject):  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y)  # 移動したかどうかをTFで返す  def move(self, x, y, game\_master, absolute = False):  if absolute == False:  x += self.\_x  y += self.\_y  if x in range(game\_master.\_width) \  and y in range(game\_master.\_height) \  and type(game\_master.\_before\_board[y][x]) != Robot:  #移動ができる  game\_master.\_after\_board[y][x] = self  super().move(x, y)  game\_master.player\_pos = (y, x)  return True  else:  print('move error. now pos is ' + str((x, y)))  return False  class Robot(MoveObject):  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y)  self.\_sclap = False  @property  def sclap(self):  return self.\_sclap  #使う予定はないけど、一応用意しておく  @sclap.setter  def sclap(self, boolean):  self.\_sclap = boolean  # 返り値として、GameOverか否かを返す。  def move(self, game\_master):  if self.\_sclap == False:  #移動する座標の算出  pos = []  for p, e in zip(game\_master.player\_pos, (self.\_y, self.\_x)):  if p - e > 0:  pos.append(1)  elif p - e == 0:  pos.append(0)  else:  pos.append(-1)  y = self.\_y + pos[0]  x = self.\_x + pos[1]  if type(game\_master.after\_board[y][x]) == Robot:  #collision  self.kill(game\_master)  game\_master.after\_board[y][x].kill(game\_master)  elif type(game\_master.after\_board[y][x]) == Player:  #gameover  return True  game\_master.after\_board[y][x] = self  super().move(x, y)  else:  #先に移動してきたロボットの処理  if type(game\_master.after\_board[self.\_y][self.\_x]) == Robot:  game\_master.after\_board[self.\_y][self.\_x].kill(game\_master)  #スクラップだから動かない  game\_master.after\_board[self.\_y][self.\_x] = self  #まだスクラップになっていないなら、ロボ残数を減らし、スクラップ状態にする。  #delとかを使って書いたほうが良かったかも  def kill(self, game\_master):  if self.\_sclap == False:  self.\_sclap = True  game\_master.robot\_left -= 1  game\_master.score += 1  # ゲームの中心となる情報をすべて持っているクラス。  # 基本的に、この中からゲームを操作する。  class Game():  def \_\_init\_\_(self, level = 1, height = 10, width = 10):  self.\_level = level  self.\_height = height  self.\_width = width  self.\_robot\_left = level \* 5  self.\_board = [[None for i in range(height)] for j in range(width)]  self.\_before\_board = list(self.\_board)  self.\_after\_board = list(self.\_board)  self.\_score = 0  self.\_MAX\_ROBOT = 40  @property  def level(self):  return self.\_level  @level.setter  def level(self, level):  self.\_level = level  #残りロボ数も更新  self.\_robot\_left = level \* 5  if self.\_robot\_left > self.\_MAX\_ROBOT:  self.\_robot\_left = self.\_MAX\_ROBOT  @property  def score(self):  return self.\_score  @score.setter  def score(self, num):  self.\_score = num  @property  def height(self):  return self.\_height  @height.setter  def height(self, height):  self.\_height = height  @property  def width(self):  return self.\_width  @width.setter  def width(self, width):  self.\_width = width  @property  def robot\_left(self):  return self.\_robot\_left  @robot\_left.setter  def robot\_left(self, robot\_left):  self.\_robot\_left = robot\_left  @property  def board(self):  return self.\_board  @property  def before\_board(self):  return self.\_before\_board  @property  def after\_board(self):  return self.\_after\_board  @property  def player\_pos(self):  return self.\_player\_pos  @player\_pos.setter  def player\_pos(self, tp):  self.\_player\_pos = tp  #引数で(y, x)の順で座標を渡すと、その座標に何がいるかを返す関数  def board\_element(self, pos):  (y, x) = pos  return self.\_board[y][x]  #ゲームの初期状態を生成するクラス。  #レベルアップごとに呼び出される。  def setting(self):  self.\_board = [[None for i in range(self.\_height)] for j in range(self.\_width)]  temp\_list = []  center\_x = self.\_width // 2  center\_y = self.\_height // 2  center = (center\_y, center\_x)  for y in range(self.\_height):  for x in range(self.\_width):  if (y, x) != center:  temp\_list.append((y, x))  #it = itertools.product(range(self.\_height), range(self.\_width))  #temp\_list = [e for e in it if e != center]  random.shuffle(temp\_list)  for y, x in temp\_list[:self.\_robot\_left]:  self.\_board[y][x] = Robot(y = y, x = x)  self.\_board[center\_y][center\_x] = Player(y = center\_y, x = center\_x)  self.\_player\_pos = (center\_y, center\_x)  def show(self):  #ゲームボード表示  bug\_flag = False  print('-' \* (self.\_width + 2))  for y in range(self.\_height):  print('|', end = '')  for x in range(self.\_width):  if self.\_board[y][x] is None:  print(' ', end = '')  elif type(self.\_board[y][x]) == Robot:  if self.\_board[y][x].sclap == True:  print('\*', end = '')  else:  print('+', end = '')  elif type(self.\_board[y][x]) == Player:  print('@', end = '')  else:  #本来ありえない表示  print('#', end = '')  bug\_flag = True  print('|')  print('-' \* (self.\_width + 2))  if bug\_flag == True:  #あり得ない表示が出た時のメッセージ  print('予期せぬエラーが発生しています。')  print('直ちに開発者に連絡してください。')  #ゲームステータス表示  print('lv:'+str(self.\_level)+', score:'+str(self.\_score))  #ロボットがいない場所からランダムに1地点を選び、座標を(y, x)の順で返す関数  def teleport(self):  pos\_list = []  for y in range(self.\_height):  for x in range(self.\_width):  if isinstance(self.\_board[y][x], Object) == False:  pos\_list.append((y, x))  pos = random.choice(pos\_list)  return tuple\_calc(pos, self.\_player\_pos, ope='sub')  #なにかしらのイベントが有る場合、Trueで返す  def action(self, command):  self.\_before\_board = list(self.\_board) #中身のコピー  self.\_after\_board = [[None for i in range(self.\_height)] \  for j in range(self.\_width)]  #各オブジェクトの移動  #プレイヤー  #(y, x)の順で入れる  if command == 0:  tp = self.teleport()  else:  tp = (None, None)  cmd\_to\_move = [tp,(1,-1),(1,0),(1,1),(0,-1),(0,0),(0,1),(-1,-1),(-1,0),(-1,1)]  (move\_y, move\_x) = cmd\_to\_move[command]  if self.board\_element(self.\_player\_pos).move(move\_x, move\_y, game\_master):  #プレイヤーが正常に移動した場合、敵の移動  for enemy\_pos\_y in range(self.\_height):  for enemy\_pos\_x in range(self.\_width):  if type(self.board\_element((enemy\_pos\_y, enemy\_pos\_x))) == Robot:  gameover = self.board\_element((enemy\_pos\_y, enemy\_pos\_x)).move(game\_master)  if gameover == True:  #game over!  return True  else:  #行動せずに終了 = before状態のまま  self.\_after\_board = list(self.\_before\_board)  if self.\_robot\_left <= 0:  #level up!  return True  #最後にafter状態を現在の状態にして終了  self.\_board = list(self.\_after\_board)  #正常終了はFalse  return False  #どの方向に動くかを読み取る関数  def read\_command(game\_master):  print('press "h" key to open help')  while True:  try:  getch = \_Getch()  x = getch()  command = int(x)  except ValueError:  #hキーならヘルプを表示  if x == 'h':  print('\nhow to operate')  print('7 8 9')  print('4 @ 6')  print('1 2 3')  print('\n0 ... random teleport')  print('5 ... stand-by')  #ctr-c、EOFが入力されたら、強制終了  elif x == '\x03':  # ctr-c  raise KeyboardInterrupt  elif x == '\x04':  # EOF  raise EOFError  #数字を読み込んだ直後の末端記号が入ったときは読み飛ばす  #それ以外の文字は入力範囲外のものだから、エラーメッセージを表示  elif x != '\x00':  print('error. input is integer')  else:  #コマンドは0～9、つまりrange(10)  if command in range(9+1):  break  else:  print('error. range is 1-9')  return command  #ゲーム開始から終了までを司る関数  def main\_game(game\_master):  game\_master.setting()  while True:  game\_master.show()  command = read\_command(game\_master)  event = game\_master.action(command)  if event == True:  if game\_master.robot\_left <= 0:  #level up  print('CLEAR! LEVEL' + str(game\_master.level) + ' => ' + str(game\_master.level + 1))  game\_master.score += game\_master.level \* 10  game\_master.level += 1  game\_master.setting()  else:  #game over  print('GAME OVER')  return None  #ゲームに必要な前準備を書いておく  #今回みたいに1行だったら関数にする必要ないが、拡張したときにわかりやすくなるように  def init\_game():  game\_master = Game()  return game\_master  #ゲーム終了後の処理を行う関数  #主に、続けてプレイするかを聞いている。  def postprocess\_game():  print('continue? y/n')  while True:  getch = \_Getch()  command = getch()  if command in {'y', 'n'}:  break  return command  if \_\_name\_\_ == '\_\_main\_\_':  game\_master = init\_game()  while True:  main\_game(game\_master)  cont = postprocess\_game()  if cont == 'n':  break |