

# Chapter 2

---

## Agents and Control

This implements the controllers described in Chapter 2.

In this version the higher-levels call the lower-levels. A more sophisticated version may have them run concurrently (either as coroutines or in parallel). The higher-levels calling the lower-level works in simulated environments when there is a single agent, and where the lower-level are written to make sure they return (and don't go on forever), and the higher level doesn't take too long (as the lower-levels will wait until called again).

### 2.1 Representing Agents and Environments

An agent observes the world, and carries out actions in the environment, it also has an internal state that it updates. The environment takes in actions of the agents, updates its internal state and returns the percepts.

In this implementation, the state of the agent and the state of the environment are represented using standard Python variables, which are updated as the state changes. The percepts and the actions are represented as variable-value dictionaries.

An agent implements the `go(n)` method, where  $n$  is an integer. This means that the agent should run for  $n$  time steps.

In the following code `raise NotImplementedError()` is a way to specify an abstract method that needs to be overridden in any implemented agent or environment.

```
agents.py — Agent and Controllers  
_____  
11 | import random  
12 |  
13 | class Agent(object):  
14 |     def __init__(self, env):
```

```

15     """set up the agent"""
16     self.env=env
17
18     def go(self,n):
19         """acts for n time steps"""
20         raise NotImplementedError("go") # abstract method

```

The environment implements a *do(action)* method where *action* is a variable-value dictionary. This returns a percept, which is also a variable-value dictionary. The use of dictionaries allows for structured actions and percepts.

Note that *Environment* is a subclass of *Displayable* so that it can use the *display* method described in Section 1.7.1.

---

agents.py — (continued)

---

```

22 from display import Displayable
23 class Environment(Displayable):
24     def initial_percepts(self):
25         """returns the initial percepts for the agent"""
26         raise NotImplementedError("initial_percepts") # abstract method
27
28     def do(self,action):
29         """does the action in the environment
30         returns the next percept """
31         raise NotImplementedError("do") # abstract method

```

## 2.2 Paper buying agent and environment

To run the demo, in folder "aipython", load "agents.py", using e.g., ipython -i agents.py, and copy and paste the commented-out commands at the bottom of that file. This requires Python 3 with matplotlib.

This is an implementation of the paper buying example.

### 2.2.1 The Environment

The environment state is given in terms of the *time* and the amount of paper in *stock*. It also remembers the in-stock history and the price history. The percepts are the price and the amount of paper in stock. The action of the agent is the number to buy.

Here we assume that the prices are obtained from the *prices* list plus a random integer in range  $[0, \max\_price\_addon)$  plus a linear "inflation". The agent cannot access the price model; it just observes the prices and the amount in stock.

---

agents.py — (continued)

---

33 | **class** TP\_env(Environment):

```

34     prices = [234, 234, 234, 234, 255, 275, 275, 211, 211, 211,
35     234, 234, 234, 234, 199, 199, 275, 275, 234, 234, 234, 255,
36     255, 260, 260, 265, 265, 265, 265, 270, 270, 255, 255, 260, 260,
37     265, 265, 150, 150, 265, 265, 270, 270, 255, 255, 260, 260, 265,
38     265, 265, 265, 270, 270, 211, 211, 255, 255, 260, 260, 265, 265,
39     260, 265, 270, 270, 205, 255, 255, 260, 260, 265, 265, 265, 265,
40     270, 270]
41 max_priceAddon = 20 # maximum of random value added to get price
42
43 def __init__(self):
44     """paper buying agent"""
45     self.time=0
46     self.stock=20
47     self.stockHistory = [] # memory of the stock history
48     self.priceHistory = [] # memory of the price history
49
50 def initialPercepts(self):
51     """return initial percepts"""
52     self.stockHistory.append(self.stock)
53     price = self.prices[0]+random.randrange(self.max_priceAddon)
54     self.priceHistory.append(price)
55     return {'price': price,
56             'instock': self.stock}
57
58 def do(self, action):
59     """does action (buy) and returns percepts (price and instock)"""
60     used = pickFromDist({6:0.1, 5:0.1, 4:0.2, 3:0.3, 2:0.2, 1:0.1})
61     bought = action['buy']
62     self.stock = self.stock+bought-used
63     self.stockHistory.append(self.stock)
64     self.time += 1
65     price = (self.prices[self.time%len(self.prices)] # repeating pattern
66             +random.randrange(self.max_priceAddon) # plus randomness
67             +self.time//2) # plus inflation
68     self.priceHistory.append(price)
69     return {'price': price,
70             'instock': self.stock}
71

```

The `pickFromDist` method takes in a `item : probability` dictionary, and returns one of the items in proportion to its probability.

---

agents.py — (continued)

---

```

72 def pickFromDist(item_prob_dist):
73     """ returns a value from a distribution.
74     item_prob_dist is an item:probability dictionary, where the
75         probabilities sum to 1.
76     returns an item chosen in proportion to its probability
77     """
78     ranreal = random.random()
79     for (it,prob) in item_prob_dist.items():
80         if ranreal < prob:

```

```

81         return it
82     else:
83         ranreal -= prob
84     raise RuntimeError(str(item_prob_dist)+" is not a probability
distribution")

```

## 2.2.2 The Agent

The agent does not have access to the price model but can only observe the current price and the amount in stock. It has to decide how much to buy.

The belief state of the agent is an estimate of the average price of the paper, and the total amount of money the agent has spent.

agents.py — (continued)

```

86 class TP_agent(Agent):
87     def __init__(self, env):
88         self.env = env
89         self.spent = 0
90         percepts = env.initial_percepts()
91         self.ave = self.last_price = percepts['price']
92         self.instock = percepts['instock']
93
94     def go(self, n):
95         """go for n time steps
96
97         for i in range(n):
98             if self.last_price < 0.9*self.ave and self.instock < 60:
99                 tobuy = 48
100            elif self.instock < 12:
101                tobuy = 12
102            else:
103                tobuy = 0
104                self.spent += tobuy*self.last_price
105                percepts = env.do({'buy': tobuy})
106                self.last_price = percepts['price']
107                self.ave = self.ave+(self.last_price-self.ave)*0.05
108                self.instock = percepts['instock']

```

Set up an environment and an agent. Uncomment the last lines to run the agent for 90 steps, and determine the average amount spent.

agents.py — (continued)

```

110 env = TP_env()
111 ag = TP_agent(env)
112 #ag.go(90)
113 #ag.spent/env.time ## average spent per time period

```

### 2.2.3 Plotting

The following plots the price and number in stock history:

```
agents.py — (continued)
115 import matplotlib.pyplot as plt
116
117 class Plot_prices(object):
118     """Set up the plot for history of price and number in stock"""
119     def __init__(self, ag, env):
120         self.ag = ag
121         self.env = env
122         plt.ion()
123         plt.xlabel("Time")
124         plt.ylabel("Number in stock.
125             Price.")
126
127     def plot_run(self):
128         """plot history of price and instock"""
129         num = len(env.stock_history)
130         plt.plot(range(num), env.stock_history, label="In stock")
131         plt.plot(range(num), env.price_history, label="Price")
132         #plt.legend(loc="upper left")
133         plt.draw()
134
135 # pl = Plot_prices(ag,env)
136 # ag.go(90); pl.plot_run()
```

## 2.3 Hierarchical Controller

To run the hierarchical controller, in folder "aipython", load "agentTop.py", using e.g., ipython -i agentTop.py, and copy and paste the commands near the bottom of that file. This requires Python 3 with matplotlib.

In this implementation, each layer, including the top layer, implements the environment class, because each layer is seen as an environment from the layer above.

We arbitrarily divide the environment and the body, so that the environment just defines the walls, and the body includes everything to do with the agent. Note that the named locations are part of the (top-level of the) agent, not part of the environment, although they could have been.

### 2.3.1 Environment

The environment defines the walls.

---

agentEnv.py — Agent environment

---

```

11 | import math
12 | from agents import Environment
13 |
14 | class Rob_env(Environment):
15 |     def __init__(self,walls = {}):
16 |         """walls is a set of line segments
17 |             where each line segment is of the form ((x0,y0),(x1,y1))
18 |         """
19 |         self.walls = walls

```

### 2.3.2 Body

The body defines everything about the agent body.

---

agentEnv.py — (continued)

```

21 | import math
22 | from agents import Environment
23 | import matplotlib.pyplot as plt
24 | import time
25 |
26 | class Rob_body(Environment):
27 |     def __init__(self, env, init_pos=(0,0,90)):
28 |         """ env is the current environment
29 |             init_pos is a triple of (x-position, y-position, direction)
30 |                 direction is in degrees; 0 is to right, 90 is straight-up, etc
31 |         """
32 |         self.env = env
33 |         self.rob_x, self.rob_y, self.rob_dir = init_pos
34 |         self.turning_angle = 18 # degrees that a left makes
35 |         self.whisker_length = 6 # length of the whisker
36 |         self.whisker_angle = 30 # angle of whisker relative to robot
37 |         self.crashed = False
38 |         # The following control how it is plotted
39 |         self.plotting = True    # whether the trace is being plotted
40 |         self.sleep_time = 0.05  # time between actions (for real-time
41 |             plotting)
42 |         # The following are data structures maintained:
43 |         self.history = [(self.rob_x, self.rob_y)] # history of (x,y)
44 |             positions
45 |         self.wall_history = [] # history of hitting the wall
46 |
47 |     def percepts(self):
48 |         return {'rob_x_pos':self.rob_x, 'rob_y_pos':self.rob_y,
49 |             'rob_dir':self.rob_dir, 'whisker':self.whisker(),
50 |                 'crashed':self.crashed}
51 |         initial_percepts = percepts # use percept function for initial percepts
52 |             too
53 |
54 |     def do(self,action):
55 |         """ action is {'steer':direction}

```

```

52     direction is 'left', 'right' or 'straight'
53     """
54     if self.crashed:
55         return self.percepts()
56     direction = action['steer']
57     compass_deriv =
58         {'left':1,'straight':0,'right':-1}[direction]*self.turning_angle
59     self.rob_dir = (self.rob_dir + compass_deriv +360)%360 # make in
60         range [0,360)
61     rob_x_new = self.rob_x + math.cos(self.rob_dir*math.pi/180)
62     rob_y_new = self.rob_y + math.sin(self.rob_dir*math.pi/180)
63     path = ((self.rob_x,self.rob_y),(rob_x_new,rob_y_new))
64     if any(line_segments_intersect(path,wall) for wall in
65             self.env.walls):
66         self.crashed = True
67     if self.plotting:
68         plt.plot([self.rob_x],[self.rob_y],"r*",markersize=20.0)
69         plt.draw()
70     self.rob_x, self.rob_y = rob_x_new, rob_y_new
71     self.history.append((self.rob_x, self.rob_y))
72     if self.plotting and not self.crashed:
73         plt.plot([self.rob_x],[self.rob_y],"go")
74         plt.draw()
75         plt.pause(self.sleep_time)
76     return self.percepts()

```

This detects if the whisker and the wall intersect. It's value is returned as a percept.

---

agentEnv.py — (continued)

```

75 def whisker(self):
76     """returns true whenever the whisker sensor intersects with a wall
77     """
78     whisk_ang_world = (self.rob_dir-self.whisker_angle)*math.pi/180
79         # angle in radians in world coordinates
80     wx = self.rob_x + self.whisker_length * math.cos(whisk_ang_world)
81     wy = self.rob_y + self.whisker_length * math.sin(whisk_ang_world)
82     whisker_line = ((self.rob_x,self.rob_y),(wx,wy))
83     hit = any(line_segments_intersect(whisker_line,wall)
84             for wall in self.env.walls)
85     if hit:
86         self.wall_history.append((self.rob_x, self.rob_y))
87     if self.plotting:
88         plt.plot([self.rob_x],[self.rob_y],"ro")
89         plt.draw()
90     return hit
91
92 def line_segments_intersect(linea,lineb):
93     """returns true if the line segments, linea and lineb intersect.
94     A line segment is represented as a pair of points.
95     A point is represented as a (x,y) pair.

```

```

96     """
97     ((x0a,y0a),(x1a,y1a)) = linea
98     ((x0b,y0b),(x1b,y1b)) = lineb
99     da, db = x1a-x0a, x1b-x0b
100    ea, eb = y1a-y0a, y1b-y0b
101    denom = db*ea-eb*da
102    if denom==0: # line segments are parallel
103        return False
104    cb = (da*(y0b-y0a)-ea*(x0b-x0a))/denom # position along line b
105    if cb<0 or cb>1:
106        return False
107    ca = (db*(y0b-y0a)-eb*(x0b-x0a))/denom # position along line a
108    return 0<=ca<=1
109
110 # Test cases:
111 # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0,1)))
112 # assert not line_segments_intersect(((0,0),(1,1)),((1,0),(0.6,0.4)))
113 # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0.4,0.6)))

```

### 2.3.3 Middle Layer

The middle layer acts like both a controller (for the environment layer) and an environment for the upper layer. It has to tell the environment how to steer. Thus it calls *env.do(·)*. It also is told the position to go to and the timeout. Thus it also has to implement *do(·)*.

```

agentMiddle.py — Middle Layer
_____
11 from agents import Environment
12 import math
13
14 class Rob_middle_layer(Environment):
15     def __init__(self,env):
16         self.env=env
17         self.percepts = env.initial_percepts()
18         self.straight_angle = 11 # angle that is close enough to straight
19             ahead
20         self.close_threshold = 2 # distance that is close enough to arrived
21         self.close_threshold_squared = self.close_threshold**2 # just
22             compute it once
23
24     def initial_percepts(self):
25         return {}
26
27     def do(self, action):
28         """action is {'go_to':target_pos,'timeout':timeout}
29         target_pos is (x,y) pair
30         timeout is the number of steps to try
31         returns {'arrived':True} when arrived is true
32             or {'arrived':False} if it reached the timeout

```

```

31     """
32     if 'timeout' in action:
33         remaining = action['timeout']
34     else:
35         remaining = -1 # will never reach 0
36         target_pos = action['go_to']
37         arrived = self.close_enough(target_pos)
38         while not arrived and remaining != 0:
39             self.percepts = self.env.do({"steer":self.steer(target_pos)})
40             remaining -= 1
41             arrived = self.close_enough(target_pos)
42     return {'arrived':arrived}

```

This determines how to steer depending on whether the goal is to the right or the left of where the robot is facing.

---

agentMiddle.py — (continued)

```

44 def steer(self,target_pos):
45     if self.percepts['whisker']:
46         self.display(3,'whisker on', self.percepts)
47         return "left"
48     else:
49         gx,gy = target_pos
50         rx,ry = self.percepts['rob_x_pos'],self.percepts['rob_y_pos']
51         goal_dir = math.acos((gx-rx)/math.sqrt((gx-rx)*(gx-rx)
52                                     +(gy-ry)*(gy-ry)))*180/math.pi
53         if ry>gy:
54             goal_dir = -goal_dir
55             goal_from_rob = (goal_dir -
56                               self.percepts['rob_dir']+540)%360-180
57             assert -180 < goal_from_rob <= 180
58             if goal_from_rob > self.straight_angle:
59                 return "left"
60             elif goal_from_rob < -self.straight_angle:
61                 return "right"
62             else:
63                 return "straight"
64
65 def close_enough(self,target_pos):
66     gx,gy = target_pos
67     rx,ry = self.percepts['rob_x_pos'],self.percepts['rob_y_pos']
68     return (gx-rx)**2 + (gy-ry)**2 <= self.close_threshold_squared

```

---

### 2.3.4 Top Layer

The top layer treats the middle layer as its environment. Note that the top layer is an environment for us to tell it what to visit.

---

agentTop.py — Top Layer

---

```

11 | from agentMiddle import Rob_middle_layer

```

---

```

12 | from agents import Environment
13 |
14 | class Rob_top_layer(Environment):
15 |     def __init__(self, middle, timeout=200, locations = {'mail':(-5,10),
16 |                                         'o103':(50,10), 'o109':(100,10),'storage':(101,51)}
17 |                                         ):
18 |         """middle is the middle layer
19 |             timeout is the number of steps the middle layer goes before giving
20 |                 up
21 |             locations is a loc:pos dictionary
22 |                 where loc is a named location, and pos is an (x,y) position.
23 | """
24 |         self.middle = middle
25 |         self.timeout = timeout # number of steps before the middle layer
26 |             should give up
27 |         self.locations = locations
28 |
29 |
30 |
31 |     def do(self,plan):
32 |         """carry out actions.
33 |             actions is of the form {'visit':list_of_locations}
34 |             It visits the locations in turn.
35 | """
36 |
37 |         to_do = plan['visit']
38 |         for loc in to_do:
39 |             position = self.locations[loc]
40 |             arrived = self.middle.do({'go_to':position,
41 |                                         'timeout':self.timeout})
42 |             self.display(1,"Arrived at",loc,arrived)

```

### 2.3.5 Plotting

The following is used to plot the locations, the walls and (eventually) the movement of the robot. It can either plot the movement if the robot as it is going (with the default *env.plotting = True*), or not plot it as it is going (setting *env.plotting = False*; in this case the trace can be plotted using *pl.plot\_run()*).

---

agentTop.py — (continued)

```

37 | import matplotlib.pyplot as plt
38 |
39 | class Plot_env(object):
40 |     def __init__(self, body,top):
41 |         """sets up the plot
42 |         """
43 |         self.body = body
44 |         plt.ion()
45 |         plt.clf()
46 |         plt.axes().set_aspect('equal')
47 |         for wall in body.env.walls:
48 |             ((x0,y0),(x1,y1)) = wall

```

```

49     plt.plot([x0,x1],[y0,y1],"-k",linewidth=3)
50     for loc in top.locations:
51         (x,y) = top.locations[loc]
52         plt.plot([x],[y],"k<")
53         plt.text(x+1.0,y+0.5,loc) # print the label above and to the
54         right
55     plt.plot([body.rob_x],[body.rob_y],"go")
56     plt.draw()
57
58     def plot_run(self):
59         """plots the history after the agent has finished.
60         This is typically only used if body.plotting==False
61         """
62         xs,ys = zip(*self.body.history)
63         plt.plot(xs,ys,"go")
64         wxs,wys = zip(*self.body.wall_history)
65         plt.plot(wxs,wys,"ro")
66         #plt.draw()

```

The following code plots the agent as it acts in the world:

```

agentTop.py — (continued)

67 from agentEnv import Rob_body, Rob_env
68
69 env = Rob_env({((20,0),(30,20)), ((70,-5),(70,25))})
70 body = Rob_body(env)
71 middle = Rob_middle_layer(body)
72 top = Rob_top_layer(middle)
73
74 # try:
75 # pl=Plot_env(body,top)
76 # top.do({'visit':['o109','storage','o109','o103']})
77 # You can directly control the middle layer:
78 # middle.do({'go_to':(30,-10), 'timeout':200})
79 # Can you make it crash?

```

**Exercise 2.1** The following code implements a robot trap. Write a controller that can escape the “trap” and get to the goal. See textbook for hints.

```

agentTop.py — (continued)

81 # Robot Trap for which the current controller cannot escape:
82 trap_env = Rob_env({((10,-21),(10,0)), ((10,10),(10,31)),
83                     ((30,-10),(30,0)),
84                     ((30,10),(30,20)), ((50,-21),(50,31)),
85                     ((10,-21),(50,-21)),
86                     ((10,0),(30,0)), ((10,10),(30,10)), ((10,31),(50,31))})
87 trap_body = Rob_body(trap_env,init_pos=(-1,0,90))
88 trap_middle = Rob_middle_layer(trap_body)
89 trap_top = Rob_top_layer(trap_middle,locations={'goal':(71,0)})
90
91 # Robot trap exercise:

```

```
90 | # pl=Plot_env(trap_body,trap_top)
91 | # trap_top.do({'visit':['goal']})
```