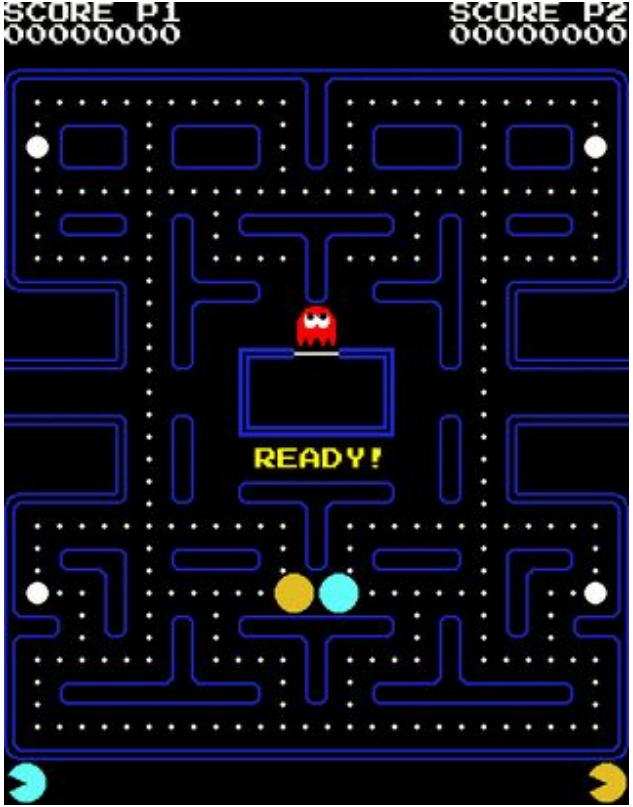
Philipps Universität Marburg

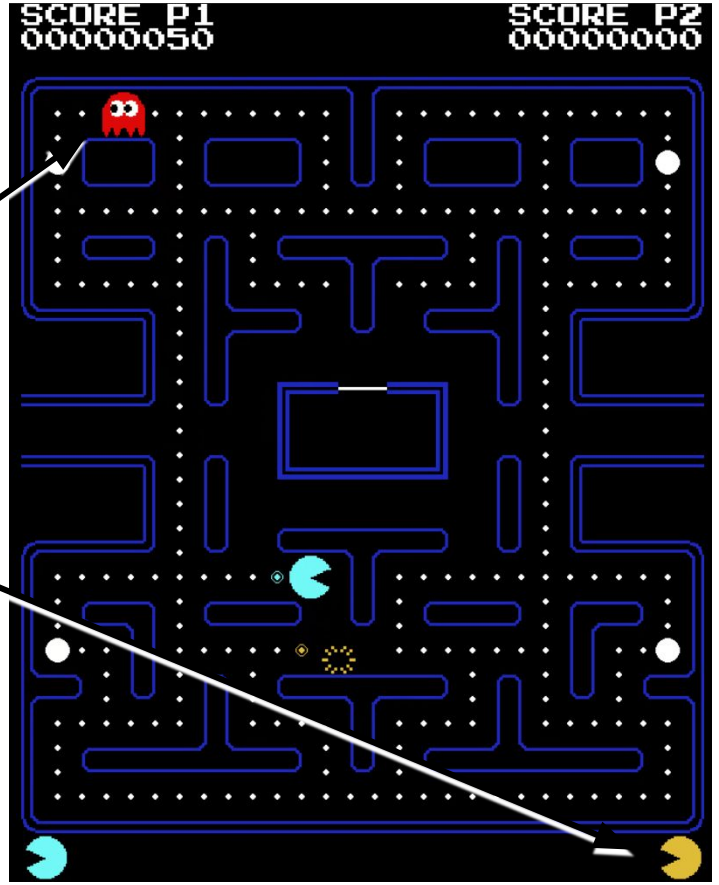
Making a Pacman Artificial Intelligence

Competing Pacmans

- You or your team will write a function to tell Pacman how to move
- You will choose a color
- We will match the Pacmans against each other to have them compete!
- You only have to edit mypacman.py



Game Components



Player scores

Enemy ghost (only 1)

Number of lives for Player 2

Number of lives for Player 1

The game ends when either:

- Both players have 0 lives
- All pellets have been eaten

Winner has the most points

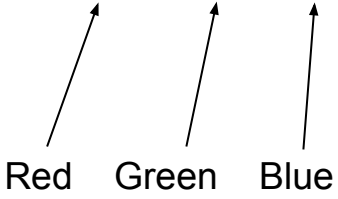
Getting started

Instructions for obtaining the starter code

Your Pacman: Choose the color

- Can be a color you choose
- Color can be YELLOW, WHITE, RED, PINK, TEAL, ORANGE, GREEN
- Or you can define a custom color with RGB values (0-255)

```
color = ( 255, 10, 0)
```




```
class MyPacmanAI(Pacman):
    def __init__(self, node, playerNum):
        color = GREEN
        Pacman.  init__(self, node,
color, playerNum)
```

Telling Pacman to Move

You will implement one function called move()

```
class MyPacmanAI(Pacman):
    def __init__(self, node, playerNum):
        color = GREEN
        Pacman.__init__(self, node, color, playerNum)

    def move(self, opponent, pellets, fruit, ghosts):
        direction = random.choice([UP, DOWN, LEFT, RIGHT])
        return direction
```

Telling Pacman to Move

You will implement one function called move()

```
class MyPacmanAI(Pacman):
    def __init__(self, node, playerNum):
        color = GREEN
        Pacman.__init__(self, node, color, playerNum)

    def move(self, opponent, pellets, fruit, ghosts):
        direction = random.choice([UP, DOWN, LEFT, RIGHT])
        return direction
```

Initially, we have a **list** of possible **directions** [UP, DOWN, LEFT, RIGHT] and the code is **randomly choosing one**

Telling Pacman to Move

You will implement one function called move()

```
class MyPacmanAI(Pacman):
    def __init__(self, node, playerNum):
        color = GREEN
        Pacman.__init__(self, node, color, playerNum)

    def move(self, opponent, pellets, fruit, ghosts):
        direction = random.choice([UP, DOWN, LEFT, RIGHT])
        return direction
```

Initially, we have a **list** of possible **directions** [UP, DOWN, LEFT, RIGHT] and the code is **randomly choosing one**

The Game updates several times per second and calls move() every time.

In other words, the ***update*** and ***move*** functions will occur in each ***iteration*** of the ***game loop***

What will Pacman do?

Telling Pacman to Move

You will implement one function called move()

```
class MyPacmanAI(Pacman):
    def __init__(self, node, playerNum):
        color = GREEN
        Pacman.__init__(self, node, color, playerNum)

    def move(self, opponent, pellets, fruit, ghosts):
        direction = random.choice([UP, DOWN, LEFT, RIGHT])
        return direction
```

Initially, we have a **list** of possible **directions** [UP, DOWN, LEFT, RIGHT] and the code is **randomly choosing one**

The Game updates several times per second and calls move() every time.

In other words, the ***update*** and ***move*** functions will occur in each ***iteration*** of the ***game loop***

What will Pacman do?

💡 Note: There is another direction called STOP, that Pacman's direction will be set to if he hits a wall

Run the Code

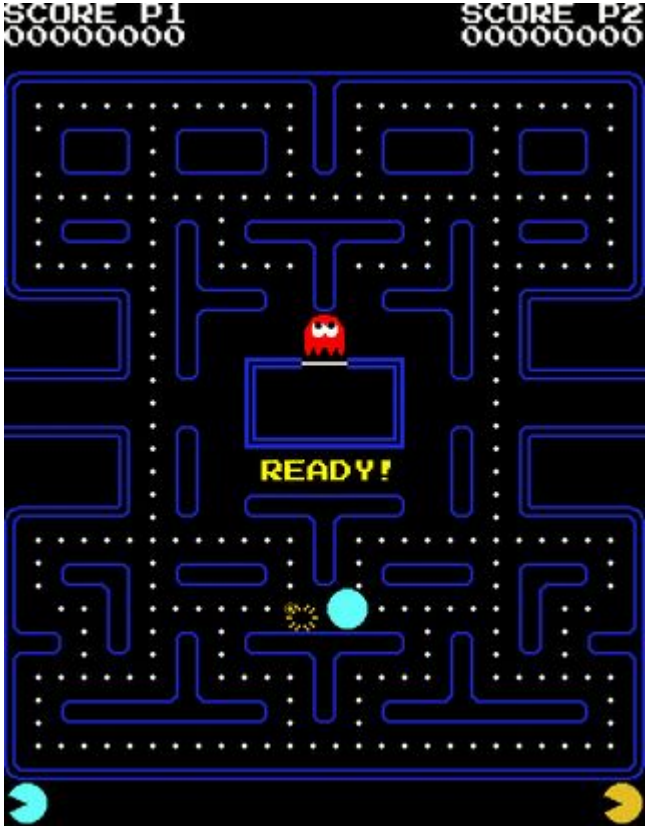
You can run the code by running

```
python run.py
```

- This will bring up the game window.
- The game will start when you press the space key.
- You can restart the game by closing the window and running the script again.

Telling Pacman to Move

- What will Pacman do?
- He jumps back and forth and does not move very far
- Pacman is changing his mind too quickly!
- We can fix this with timers



Using Timers

- We can fix this with timers

```
def move(self, opponent, pellets, fruit, ghosts):
        direction = self.direction
        if self.dt[0] > 2:
            self.resetTimer(0)
            direction = random.choice([UP, DOWN, LEFT, RIGHT])
        return direction
```

- Use self.dt[0] to refer to timer 0, self.dt[1] for timer 1, and so on
- Timers count seconds, so self.dt[0] > 2 means: “if two seconds have elapsed”
- self.resetTimer(0) will reset timer 0, telling it to start over

Using Timers

Time 1:
self.direction=RIGHT
direction=DOWN

Time 2:
self.direction=DOWN
direction=...

- We can fix this with timers

```
def move(self, opponent, pellets, fruit, ghosts):
        direction = self.direction
        if self.dt[0] > 2:
            self.resetTimer(0)
            direction = random.choice([UP, DOWN, LEFT, RIGHT])
        return direction
```

- Variables with “self.” belong to your Pacman
- Variables without it are temporary and do not exist outside of move()
- direction is where we **want Pacman to go**
- self.direction is where **Pacman is currently going**

- What will Pacman do now?

Pacman's Strategy

- What will Pacman do now?
- Problems:
  1. Pacman chooses directions but may choose to move into a wall
  2. Ignores the ghost
  3. Does not move toward pellets or fruit

First, problem 1 can be fixed with `self.validDirections()`

This will return only directions Pacman can move, for example, will return [UP, LEFT] here

Problem 1

Pacman chooses directions but may choose to move into a wall.

We need Pacman to know which directions he can move!

First, problem 1 can be fixed with `self.validDirections()`

This will return only directions Pacman can move, for example, will return [UP, LEFT] here

Problem 1

Pacman chooses directions but may choose to move into a wall.

If Pacman runs into a wall, we want him to keep moving. We can check for the STOP direction like this:

```
elif self.dt[0] > 3 or self.direction == STOP:
```

Now Pacman will change direction every 3 seconds OR when he runs into a wall.

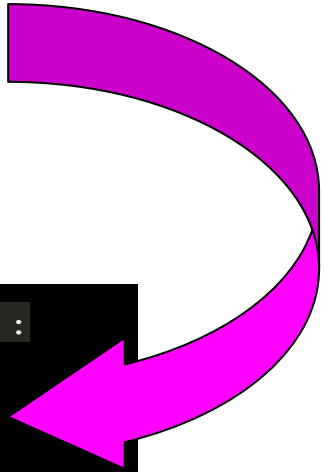
Moving Toward Objects

- To move toward something, we need to know its **position (location)**

```
def move(self, opponent, pellets, fruit, ghosts):
        directions = self.validDirections()
        self.goal = ghosts.blinky.position
        printDirections(directions)
        directions = self.goalDirection(directions)
        direction = random.choice(directions)
        return direction
```

Moving Toward Objects

- To move toward something, we need to know its position
- The ghost's location is stored in **ghosts.blinky.position**



```
def move(self, opponent, pellets, fruit, ghosts):
        directions = self.validDirections()
        self.goal = ghosts.blinky.position
        printDirections(directions)
        directions = self.goalDirection(directions)
        direction = random.choice(directions)
        return direction
```

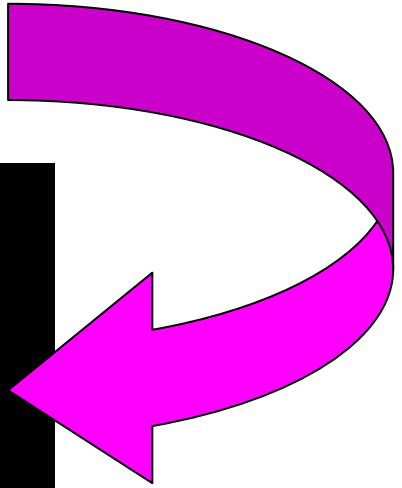
Moving Toward Objects

```
def goalDirection(self, directions, minimize=True):
        distances = []
        for direction in directions:
            goalDist = self.tileDistance(self.position + self.directions[direction]*TILEWIDTH, self.goal)
            distances.append(goalDist)

        matchValue = min(distances) if minimize else max(distances)
        indices = [i for i in range(len(directions)) if distances[i] == matchValue]
        goalDirections = []
        for i in range(len(directions)):
            if distances[i] == matchValue:
                goalDirections.append(directions[i])
        return goalDirections
```

Moving Toward Objects

- To move toward something, we need to know its position
- The ghost location is stored in ghosts.blinky.position
- Pacman has a function **self.goalDirection()** that returns the direction toward self.goal

```
def move(self, opponent, pellets, fruit, ghosts):
        directions = self.validDirections()
        self.goal = ghosts.blinky.position
        printDirections(directions)
        directions = self.goalDirection(directions)
        direction = random.choice(directions)
        return direction
```



Moving Toward Objects

- To move toward something, we need to know its position
- The ghost location is stored in ghosts.blinky.position
- Pacman has a function self.goalDirection() that returns the direction toward self.goal
- We can use these to move Pacman toward an object:

```
def move(self, opponent, pellets, fruit, ghosts):
        directions = self.validDirections()
        self.goal = ghosts.blinky.position
        printDirections(directions)
        directions = self.goalDirection(directions)
        direction = random.choice(directions)
        return direction
```

Moving Toward Objects

- To move toward something, we need it's position
- The ghost location is stored in ghosts.blinky.position
- Pacman has a function self.goalDirection() that returns the direction toward self.goal
- We can use these to move Pacman toward an object:

```
def move(self, opponent, pellets, fruit, ghosts):
        directions = self.validDirections()
        self.goal = ghosts.blinky.position
        printDirections(directions)
        directions = self.goalDirection(directions)
        direction = random.choice(directions)
        return direction
```

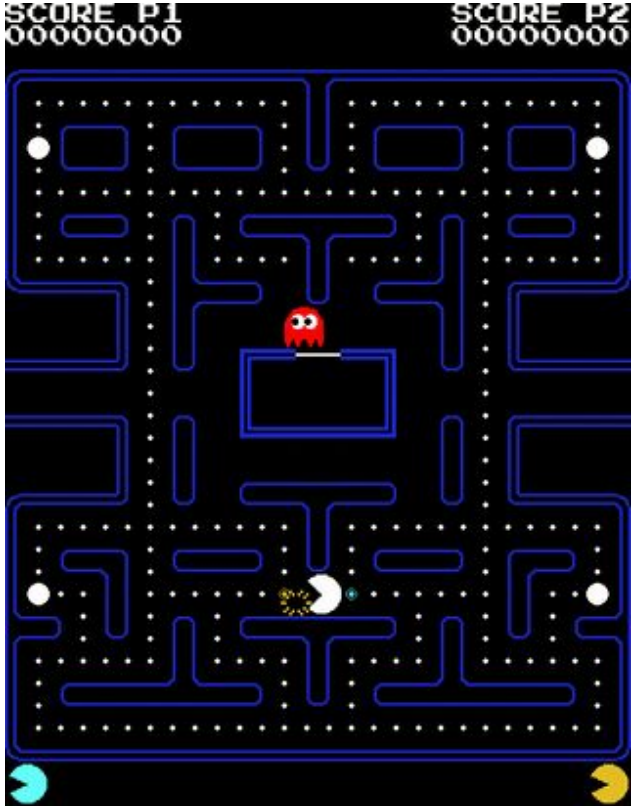
💡 Tips:
printDirections() will print direction names in the terminal

Note: self.goalDirection() takes a list of directions as input and returns the ones that will get it closest to the goal. This is often two directions, so we still need random.choice to choose one.

Moving Toward Objects

- **What will Pacman do now?**

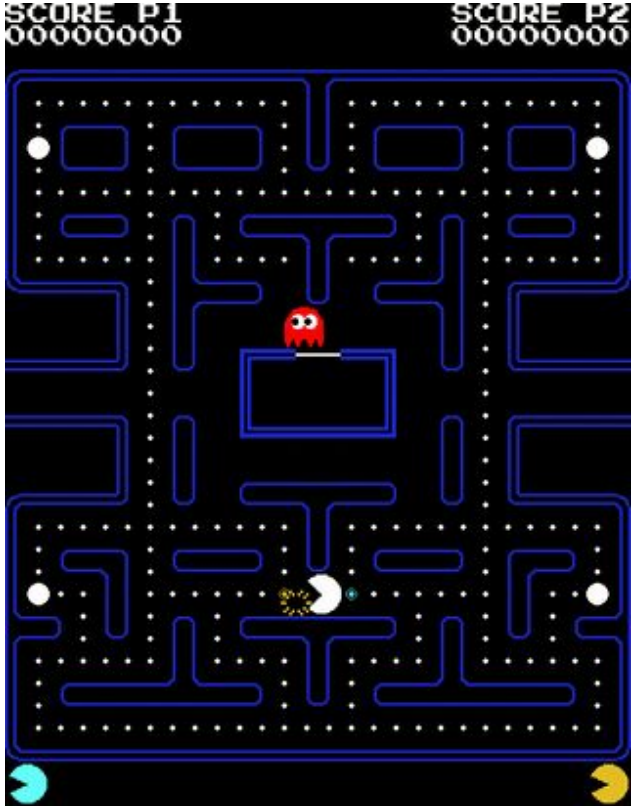
```
def move(self, opponent, pellets, fruit, ghosts):
        directions = self.validDirections()
        self.goal = ghosts.blinky.position
        printDirections(directions)
        directions = self.goalDirection(directions)
        direction = random.choice(directions)
        return direction
```



Moving Toward Objects

- What will Pacman do now?
- **He moves toward the ghost!**

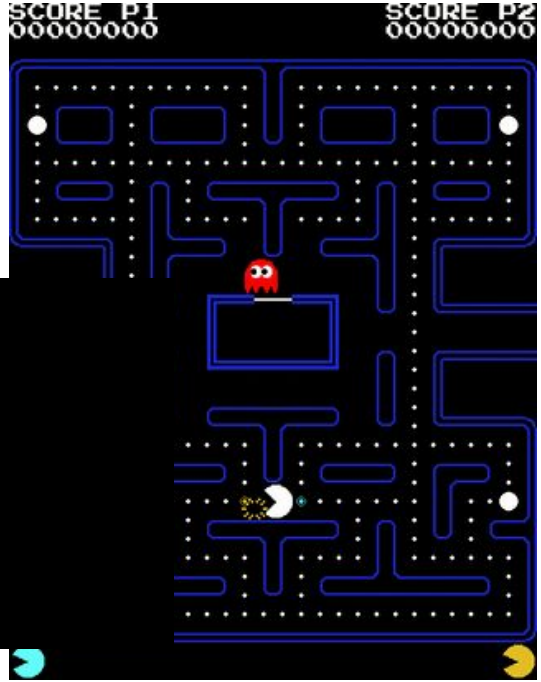
```
def move(self, opponent, pellets, fruit, ghosts):
        directions = self.validDirections()
        self.goal = ghosts.blinky.position
        printDirections(directions)
        directions = self.goalDirection(directions)
        direction = random.choice(directions)
        return direction
```



Moving Toward Objects

- What will Pacman do now?
- He moves toward the ghost!

```
def move(self, opponent, pellets, fruit, ghosts):
        directions = self.validDirections()
        self.goal = ghosts.blinky.position
        printDirections(directions)
        directions = self.goalDirection(directions,
minimize=False)
        direction = random.choice(directions)
        return direction
```

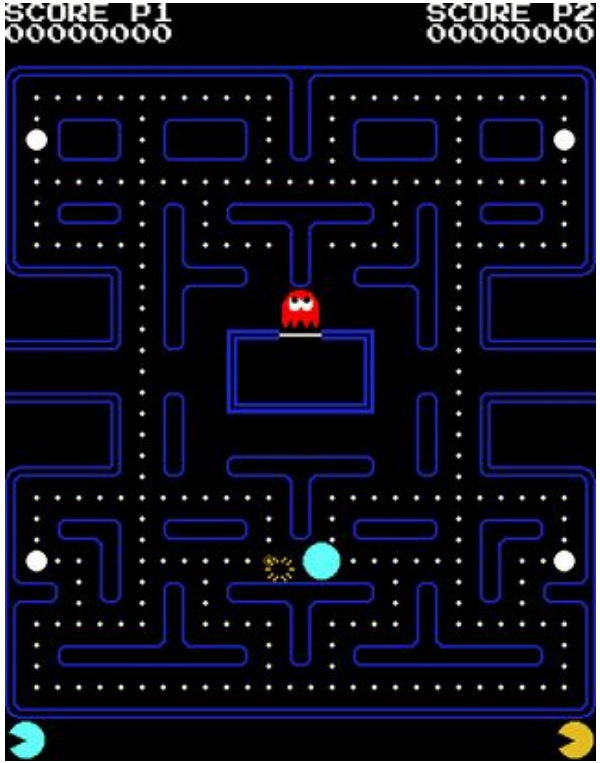


To move away instead of toward, we need to set minimize=False

This tells Pacman to find the direction that maximizes the distance instead of minimizing it.

Moving Away from Objects

- Pacman hides in the corner
- He does not need to worry about the ghost unless it is nearby
- Otherwise, he can still move randomly



Conditional Movement

To avoid ghosts only when nearby and otherwise move randomly we now have:

```
def move(self, opponent, pellets, fruit, ghosts):
        directions = [self.direction]
        d2ghost = self.tileDistance(ghosts.blinky)

        if d2ghost < 10:
            self.goal = ghosts.blinky.position
            directions = self.validDirections()
            printDirections(directions)
            directions = self.goalDirection(directions, minimize=False)
        elif self.dt[0] > 3 or self.direction == STOP:
            self.resetTimer(0)
            directions = self.validDirections()

        direction = random.choice(directions)
        return direction
```

Conditional Movement

self.tileDistance() will return the distance between Pacman and the target in tiles

Tip: You may want to also print this while testing

```
def move(self, opponent, pellets, fruit, ghosts):
        directions = [self.direction]
        d2ghost = self.tileDistance(ghosts.blinky)

        if d2ghost < 10:
            self.goal = ghosts.blinky.position
            directions = self.validDirections()
            printDirections(directions)
            directions = self.goalDirection(directions, minimize=False)
        elif self.dt[0] > 3 or self.direction == STOP:
            self.resetTimer(0)
            directions = self.validDirections()

        direction = random.choice(directions)
        return direction
```

Conditional Movement

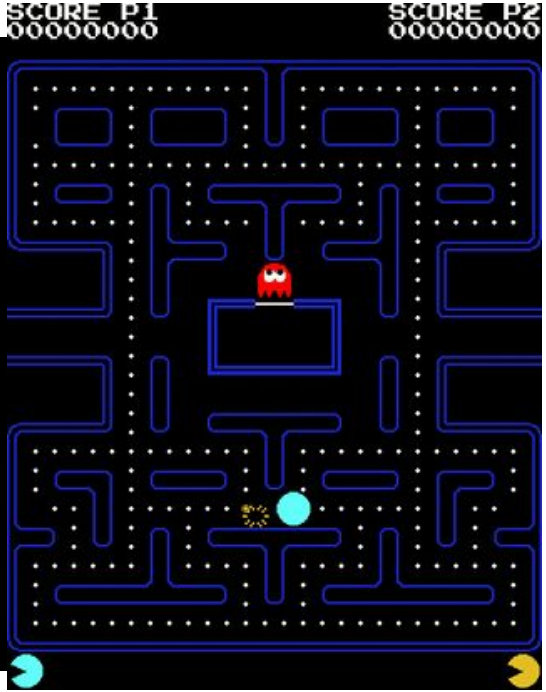
If ghost is less than 10 tiles away, move away.

Otherwise, choose a new direction every 3 seconds or when Pacman runs into a wall.

```
def move(self, opponent, pellets, fruit, ghosts):
        directions = [self.direction]
        d2ghost = self.tileDistance(ghosts.blinky)

        if d2ghost < 10:
            self.goal = ghosts.blinky.position
            directions = self.validDirections()
            printDirections(directions)
            directions = self.goalDirection(directions, minimize=False)
        elif self.dt[0] > 3 or self.direction == STOP:
            self.resetTimer(0)
            directions = self.validDirections()

        direction = random.choice(directions)
        return direction
```

Conditional Movement

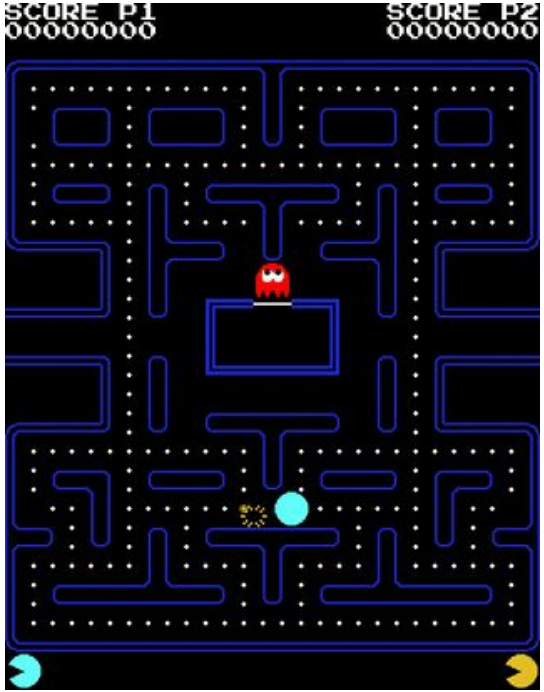
```
def move(self, opponent, pellets, fruit, ghosts):
        directions = [self.direction]
        d2ghost = self.tileDistance(ghosts.blinky)

        if d2ghost < 10:
            self.goal = ghosts.blinky.position
            directions = self.validDirections()
            printDirections(directions)
            directions = self.goalDirection(directions, minimize=False)
        elif self.dt[0] > 3 or self.direction == STOP:
            self.resetTimer(0)
            directions = self.validDirections()

        direction = random.choice(directions)
        return direction
```



Do you understand how the code is causing the Pacman's behavior here?

Moving toward the Pellets

- Great, but we don't want to move randomly. We want to eat the pellets.
- Remember: self.goalDirection() helps us determine the directions toward self.goal, which represents the location of the target object.
- We can use self.goalDirection() to move toward Pellet objects. But which pellet do we move toward?

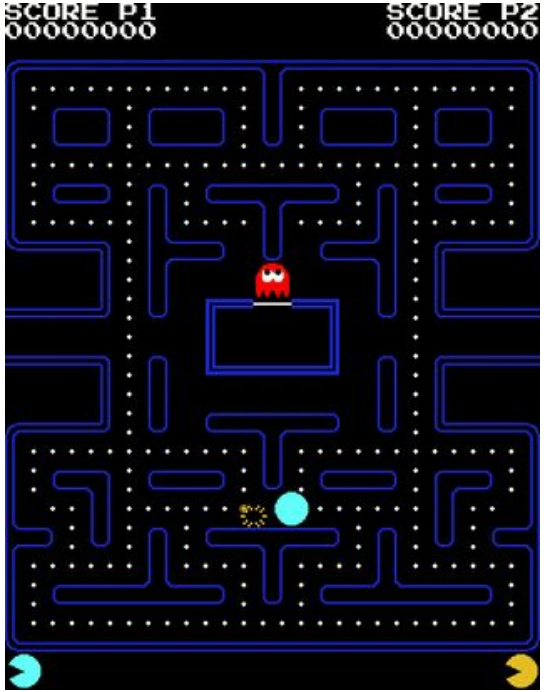


Moving toward Pellets

We can use self.goalDirection() to move toward objects. But which pellet do we move toward?

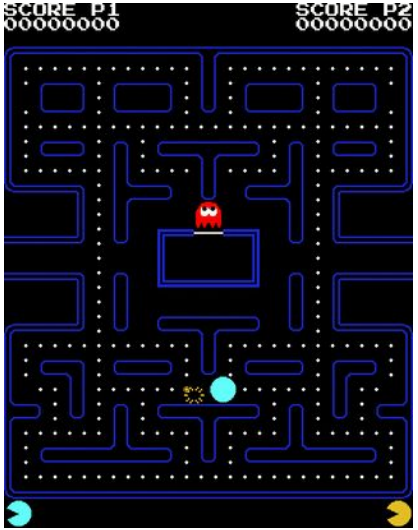
We need to:

1. Find distances to all pellets
2. Get the pellet with the shortest distance
3. Set that pellet's location as the goal



Moving toward Pellets

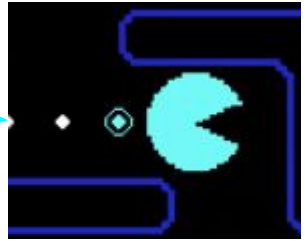
We can use self.goalDirection() to move toward objects. But which pellet do we move toward?



We need to:

1. Find distances to all pellets
2. Get the pellet with the shortest distance
3. Set that pellet's location as the goal

💡 The closest pellet to Pacman is highlighted in your color



Moving to Pellets

We need to:

1. Find distances to all pellets
2. Get the pellet with the shortest distance
3. Set that pellet's location as the goal

```
pelletDists = []
for pellet in pellets.pelletList:

pelletDists.append(self.tileDistance(pellet)
)
mindist = min(pelletDists)
index = pelletDists.index(mindist)
self.goal =
pellets.pelletList[index].position
directions = self.validDirections()
directions = self.goalDirection(directions)
```

Moving toward Pellets

We need to:

1. **Find distances to all pellets**
2. Get the pellet with the shortest distance
3. Set that pellet's location as the goal

```
pelletDists = []
for pellet in pellets.pelletList:
    pelletDists.append(self.tileDistance(pellet))
mindist = min(pelletDists)
index = pelletDists.index(mindist)
self.goal = pellets.pelletList[index].position
directions = self.validDirections()
directions = self.goalDirection(directions)
```

Loop over pellets and append the distances to a list called pelletDists

Moving toward Pellets

We need to:

1. Find distances to all pellets
2. **Get the pellet with the shortest distance**
3. Set that pellet's location as the goal

```
pelletDists = []
for pellet in pellets.pelletList:
    pelletDists.append(self.tileDistance(pellet))
mindist = min(pelletDists)
index = pelletDists.index(mindist)
self.goal = pellets.pelletList[index].position
directions = self.validDirections()
directions = self.goalDirection(directions)
```

Loop over pellets and append the distances to a list called pelletDists

Set mindist to the smallest number in the list

Moving toward Pellets

We need to:

1. Find distances to all pellets
2. **Get the pellet with the shortest distance**
3. Set that pellet's location as the goal

```
pelletDists = []
for pellet in pellets.pelletList:
    pelletDists.append(self.tileDistance(pellet))
mindist = min(pelletDists)
index = pelletDists.index(mindist)
self.goal = pellets.pelletList[index].position
directions = self.validDirections()
directions = self.goalDirection(directions)
```

Loop over pellets and append the distances to a list called pelletDists

Set mindist to the smallest number in the list

Set index to the place in the list that contains the smallest value

Moving toward Pellets

We need to:

1. Find distances to all pellets
2. Get the pellet with the shortest distance
3. **Set that pellet's location as the goal**

```
pelletDists = []
for pellet in pellets.pelletList:
    pelletDists.append(self.tileDistance(pellet))
mindist = min(pelletDists)
index = pelletDists.index(mindist)
self.goal = pellets.pelletList[index].position
directions = self.validDirections()
directions = self.goalDirection(directions)
```

Loop over pellets and append the distances to a list called pelletDists

Set mindist to the smallest number in the list

Set index to the place in the list that contains the smallest value

Set the goal to the position of this pellet

Moving toward Pellets

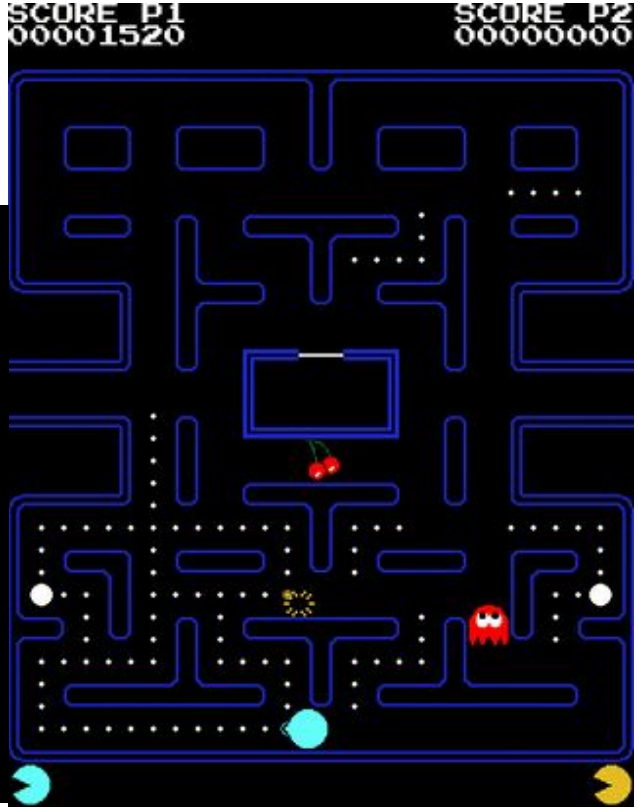
```
if d2ghost < 10:
    self.goal = ghosts.blinky.position
    directions = self.validDirections()
    directions = self.goalDirection(directions, minimize=False)
elif len(pellets.pelletList) > 0:
    pelletDists = []
    for pellet in pellets.pelletList:
        pelletDists.append(self.tileDistance(pellet))
    mindist = min(pelletDists)
    index = pelletDists.index(mindist)
    self.goal = pellets.pelletList[index].position
    directions = self.validDirections()
    directions = self.goalDirection(directions)
```

We can put this code in the move function, in place of what follows `elif self.dt[0] > 3 or self.direction == STOP:`

We also change the elif statement to check if there are any pellets on screen. This comes after checking the distance to the ghost, so it is the 2nd priority.

Moving toward Pellets

```
if d2ghost < 10:
    self.goal = ghosts.blinky.position
    directions = self.validDirections()
    directions = self.goalDirection(directions, minimize=False)
elif len(pellets.pelletList) > 0:
    pelletDists = []
    for pellet in pellets.pelletList:
        pelletDists.append( self.tileDistance(pellet))
    mindist = min(pelletDists)
    index = pelletDists.index(mindist)
    self.goal = pellets.pelletList[index].position
    directions = self.validDirections()
    directions = self.goalDirection(directions)
```

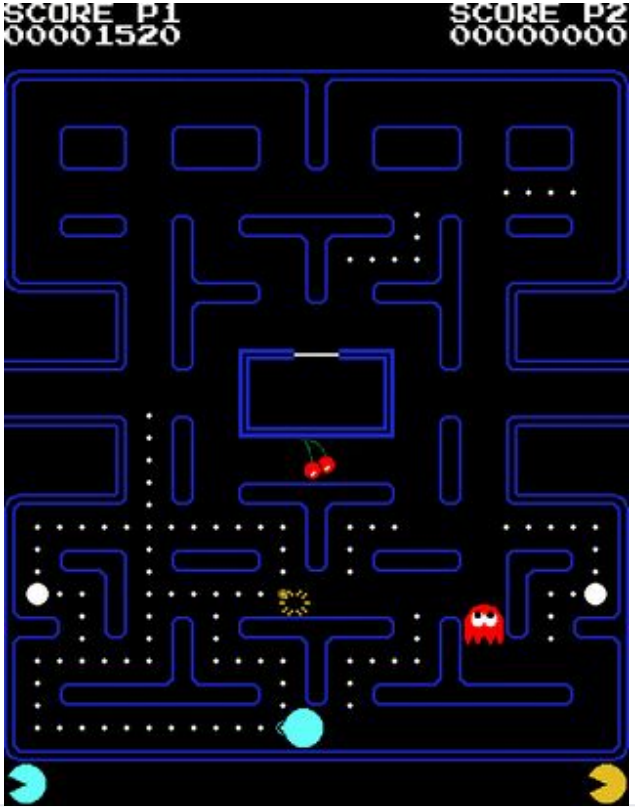


Do you understand how the code is causing the Pacman's behavior here?

Pacman's strategy

Pacman still avoids the ghost when it is blue. This is called FREIGHT mode.

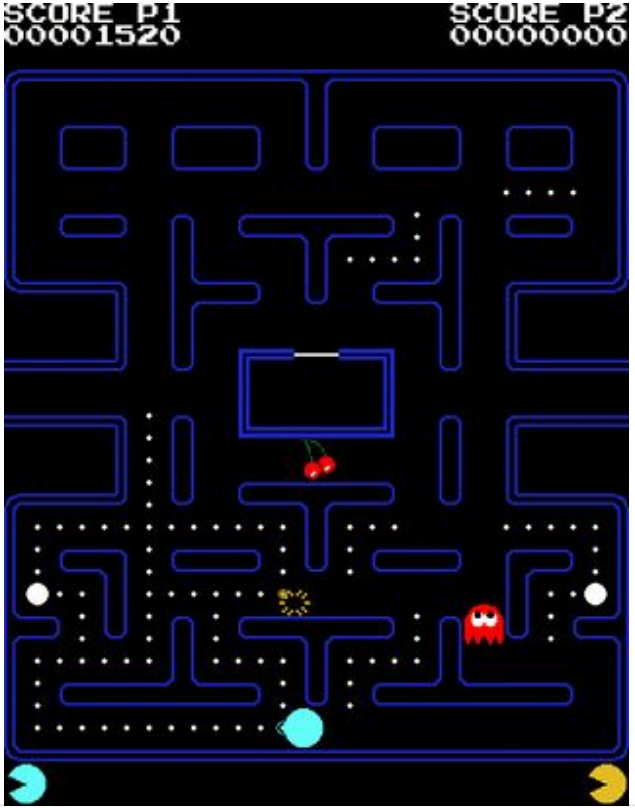
```
if d2ghost < 10:
    self.goal = ghosts.blinky.position
    directions = self.validDirections()
    directions = self.goalDirection(directions, minimize=False)
elif len(pellets.pelletList) > 0:
    pelletDists = []
    for pellet in pellets.pelletList:
        pelletDists.append(self.tileDistance(pellet))
    mindist = min(pelletDists)
    index = pelletDists.index(mindist)
    self.goal = pellets.pelletList[index].position
    directions = self.validDirections()
    directions = self.goalDirection(directions)
```



Pacman's strategy

We can tell Pacman to only avoid the ghost if it is not in this mode!

```
if d2ghost < 10 and ghosts.blinky.mode.current != FREIGHT:
    self.goal = ghosts.blinky.position
    directions = self.validDirections()
    directions = self.goalDirection(directions, minimize=False)
elif len(pellets.pelletList) > 0:
    pelletDists = []
    for pellet in pellets.pelletList:
        pelletDists.append(self.tileDistance(pellet))
    mindist = min(pelletDists)
    index = pelletDists.index(mindist)
    self.goal = pellets.pelletList[index].position
    directions = self.validDirections()
    directions = self.goalDirection(directions)
```



Advanced: Scoring Directions

```
def move(self, opponent, pellets, fruit, ghosts):
    d2ghost = self.tileDistance(ghosts.blinky)
    scores = {UP: 0, DOWN: 0, RIGHT: 0, LEFT: 0}

    if d2ghost < 10:
        self.goal = ghosts.blinky.position
        directions = self.validDirections()
        if ghosts.blinky.mode.current == FREIGHT:
            directions = self.goalDirection(directions)
            for d in directions:
                scores[d] += 20
        else:
            directions = self.goalDirection(directions,
minimize=False)
            for d in directions:
                scores[d] += 10

    if len(pellets.pelletList) > 0:
        pelletDists = []
        for pellet in pellets.pelletList:
```

Advanced: Scoring Directions

Here we make a dictionary of scores for each direction

```
def move(self, opponent, pellets, fruit, ghosts):
    d2ghost = self.tileDistance(ghosts.blinky)
    scores = {UP: 0, DOWN: 0, RIGHT: 0, LEFT: 0}

    if d2ghost < 10:
        self.goal = ghosts.blinky.position
        directions = self.validDirections()
        if ghosts.blinky.mode.current == FREIGHT:
            directions = self.goalDirection(directions)
            for d in directions:
                scores[d] += 20
        else:
            directions = self.goalDirection(directions,
minimize=False)
            for d in directions:
                scores[d] += 10

    if len(pellets.pelletList) > 0:
        pelletDists = []
        for pellet in pellets.pelletList:
```

Advanced: Scoring Directions

Here we make a dictionary of scores for each direction

At the end, we will move in the direction with the highest score. key=scores.get gets the key instead of the value.

```
def move(self, opponent, pellets, fruit, ghosts):
    d2ghost = self.tileDistance(ghosts.blinky)
    scores = {UP: 0, DOWN: 0, RIGHT: 0, LEFT: 0}

    if d2ghost < 10:
        self.goal = ghosts.blinky.position
        directions = self.validDirections()
        if ghosts.blinky.mode.current == FREIGHT:
            directions = self.goalDirection(directions)
            for d in directions:
                scores[d] += 20
        else:
            directions = self.goalDirection(directions,
minimize=False)
            for d in directions:
                scores[d] += 10

    if len(pellets.pelletList) > 0:
        pelletDists = []
        for pellet in pellets.pelletList:
```

Advanced: Scoring Directions

Here we make a dictionary of scores for each direction

If the ghost is near, moving away gives 10 points

At the end, we will move in the direction with the highest score. key=scores.get gets the key instead of the value.

```
def move(self, opponent, pellets, fruit, ghosts):
    d2ghost = self.tileDistance(ghosts.blinky)
    scores = {UP: 0, DOWN: 0, RIGHT: 0, LEFT: 0}

    if d2ghost < 10:
        self.goal = ghosts.blinky.position
        directions = self.validDirections()
        if ghosts.blinky.mode.current == FREIGHT:
            directions = self.goalDirection(directions)
            for d in directions:
                scores[d] += 20
        else:
            directions = self.goalDirection(directions,
minimize=False)
            for d in directions:
                scores[d] += 10

    if len(pellets.pelletList) > 0:
        pelletDists = []
        for pellet in pellets.pelletList:
```

Advanced: Scoring Directions

Here we make a dictionary of scores for each direction

But if the ghost is in FREIGHT mode, moving TOWARD gives 20 points

At the end, we will move in the direction with the highest score. key=scores.get gets the key instead of the value.

```
def move(self, opponent, pellets, fruit, ghosts):
    d2ghost = self.tileDistance(ghosts.blinky)
    scores = {UP: 0, DOWN: 0, RIGHT: 0, LEFT: 0}

    if d2ghost < 10:
        self.goal = ghosts.blinky.position
        directions = self.validDirections()
        if ghosts.blinky.mode.current == FREIGHT:
            directions = self.goalDirection(directions)
            for d in directions:
                scores[d] += 20
        else:
            directions = self.goalDirection(directions,
minimize=False)
            for d in directions:
                scores[d] += 10

    if len(pellets.pelletList) > 0:
        pelletDists = []
        for pellet in pellets.pelletList:
```

Advanced: Scoring Directions

Here we make a dictionary of scores for each direction

Moving toward pellets is always worth 5 points

At the end, we will move in the direction with the highest score. key=scores.get gets the key instead of the value.

```
def move(self, opponent, pellets, fruit, ghosts):
    d2ghost = self.tileDistance(ghosts.blinky)
    scores = {UP: 0, DOWN: 0, RIGHT: 0, LEFT: 0}

    if d2ghost < 10:
        self.goal = ghosts.blinky.position
        directions = self.validDirections()
        if ghosts.blinky.mode.current == FREIGHT:
            directions = self.goalDirection(directions)
            for d in directions:
                scores[d] += 20
        else:
            directions = self.goalDirection(directions,
minimize=False)
            for d in directions:
                scores[d] += 10

    if len(pellets.pelletList) > 0:
        pelletDists = []
        for pellet in pellets.pelletList:
```

Advanced: Scoring Directions

Here we make a dictionary of scores for each direction

Note that these are “if” and not “else” or “elif” statements. This is because we want the score to combine all relevant information

At the end, we will move in the direction with the highest score. key=scores.get gets the key instead of the value.

```
def move(self, opponent, pellets, fruit, ghosts):
    d2ghost = self.tileDistance(ghosts.blinky)
    scores = {UP: 0, DOWN: 0, RIGHT: 0, LEFT: 0}

    if d2ghost < 10:
        self.goal = ghosts.blinky.position
        directions = self.validDirections()
        if ghosts.blinky.mode.current == FREIGHT:
            directions = self.goalDirection(directions)
            for d in directions:
                scores[d] += 20
        else:
            directions = self.goalDirection(directions,
minimize=False)
            for d in directions:
                scores[d] += 10

    if len(pellets.pelletList) > 0:
        pelletDists = []
        for pellet in pellets.pelletList:
```

Other Options

- Can access fruit.position when fruit exists (if fruit:)
- Can access opponent.position and opponent.alive
- Can access pellets.powerpellets
- Use more timers: self.dt[1], refers to timer 1, and so on

- Create your own “modes” for Pacman

```
def __init__(self, node, playerNum):
    color = TEAL
    Pacman.__init__(self, node, color, playerNum)
    self.mode = 'powerpellets'
```

self.mode can then be accessed in the move() and used in your logic