FCPS

Python

Packets

pyKarel Programming

**August 2012**

## Authors: The teachers at

## [Thomas Jefferson High School for Science and Technology](http://www.tjhsst.edu/)

## Fairfax County Public Schools

## Fairfax, Virginia

These packets are used in a course given at TJHSST. The students' supporting web site, including code and packets, can be found at <http://academics.tjhsst.edu/compsci/index.html>. The teacher's FCPS Computer Science CD is available from Stephen Rose at [srrose@fcps.edu](mailto:srrose@fcps.edu)

### Contributing Authors

The lessons in this unit were first written for Java by teachers at TJHSST. The lessons were translated for Python by Shane Torbert and Marion Billington.

**Acknowledgements**

Many of the lessons in this unit were inspired by *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming* by Bergin, J., Stehlik, M., Roberts, J., and Pattis, R.

### License Information

This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

You are free:

\* to Share -- to copy, distribute, display, and perform the work, provided that appropriate credit is given and that the nature of your changes is clearly indicated.

Under the following conditions:

\* Attribution. You must attribute the work in the manner specified by the author or licensor.

\* Noncommercial. You may not use this work for commercial purposes.

\* For any reuse or distribution, you must make clear to others the license terms of this work.

\* Any of these conditions can be waived if you get permission from the copyright holder, [smtorbert@fcps.edu](mailto:smtorbert@fcps.edu)

**pyKarel Instruction Plan**

|  |  |
| --- | --- |
| Section One – Classes and Objects | Page |
| Lab00: Hello Robot . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-4 |
| Lab01: Students and Books . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-6 |
| Lab02: Escape the Maze . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-10 |
| Section Two – Inheritance and Methods |  |
| Lab03: Climb the Mountain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-14 |
| Lab04: Take the Field. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-16 |
| Lab05: Shuttle Run . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-21 |
| Lab06: A Half-DozenTasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-24 |
| Section Three – Algorithms |  |
| Lab07: Treasure Hunt. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-29 |
| Lab08: Yellow Brick Road . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-30 |
| Lab09: Shifting Piles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-31 |
| Lab10: Maze Escaping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-33 |
| Section Four – Polymorphic Behavior |  |
| Lab11: Climbing Up and Down . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-35 |
| Lab12: Hurdle Racing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-36 |
| Lab13: Harvesting and Planting. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-37 |
| Lab14: Republicans and Democrats . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-39 |
| **Section Five – Extending an Abstract Class** |  |
| Lab15: Synchronized Swimming . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-41 |
| Lab16: Dancing Robots . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-42 |
| Lab17: Your ID Number . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-43 |
| Lab18: An Original pyKarel Lab . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-45 |
| Appendix |  |
| Lab19: Lists of Robots . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-46 |
| Lab20: Recursive Racer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-47 |
| Lab21: Recursion: Seeking the Beeper . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-49 |
| Lab22: Seeking the Beeper, Part II . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-50 |
| **Glossary**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | pyKarel-51 |

Discussion

Welcome

Welcome to your first course in computer science. We hope you find something here to grab hold of, something to spark your interest, and something even to take your breath away.

Computer science teachers have always had to balance the big ideas against the details of syntax. We have tried to present representative computer science explorations with a minimum of overhead or tedium. For these reasons, we have chosen to work in Python 2.7. Especially at the beginning of the course, most of the background work has been done with a view to making the concepts accessible to first-year students.

Most programming languages require you to organize your files carefully. You must save your work in your personal folder, which might be on a network drive, a hard drive, or a removable drive. The pyKarel folder, with all its files and folders, should be copied from a CD or a network drive into your \personal\_folder\. Ask your teacher how to do this on your school’s system.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| personal\_folder | |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  | pyKarel |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  | east.gif |  | (Save your source code here.) | | |  |
|  | north.gif |  |  |
|  | south.gif |  |  |  |  |  |
|  | west.gif |  |  |  |  |  |
|  | Lab06.py |  |  |  |  |  |
|  | Lab07.py |  | world |  |  |  |
|  | . . . |  | \*.wld | |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

|  |
| --- |
| east.gif |

*Source code* is the Python program you write. Save your source code directly to your pyKarel folder; do not create sub-folders because your pyKarel programs and the gif image files must be

in the same folder.

This unit uses robots and worlds with walls and beepers to introduce students to the concepts of *object-oriented programming*. In object-oriented programming, both data and code are encapsulated in something called an “object.” Objects “know” how to do things. A complete program is usually composed of several objects that act on themselves as well as on each other.

Why are the robots called “karel” robots? According to *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming* by Joseph Bergin, Mark Stehlik, Jim Roberts, and Richard Pattis:

The name "karel" is used in recognition of the Czechoslovakian dramatist Karel Čapek, who popularized the word *robot* in his play *R.U.R.* (Rossum’s Universal Robots). The word *robot* is derived from the Czech word *robota*, meaning forced labor.

A helpful website for beginners is <http://wiki.python.org/moin/>

Lab00

Hello Robot

Objective

Classes, objects, and sending Robot commands. Program structure.

Background

Python programs are written in a text editor. A Python text editor for Windows computers is called IDLE. If you write code in IDLE, you can run your program by pressing F5. The results will show in one or more new windows. If you have any coding errors, those will also show. You will have to fix your errors before the program runs.

***Specification***

# Name: Torbert

# Date: 03/12/2008

from pyKarel import \*

wld = World("first")

karel = Robot(wld)

karel.move()

karel.pickBeeper()

karel.move()

karel.turnLeft()

karel.move()

karel.putBeeper()

karel.move()

karel.turnLeft()

karel.turnLeft()

wld.mainloop()

Go to File, New Window. Enter the *source code* as shown. Save it (as always) in the pyKarel folder as Lab00.py .

Lines 1 & 2: These lines state the programmer's name and date, beginning with #. The # starts a *comment* line, which is not executed; the computer ignores it.

Line 4: The line from pyKarel import \* makes available certain *classes* (World and Robot) that are located elsewhere, namely, in the pyKarel module. A class is a container for both data and code. We will be talking a lot about classes.

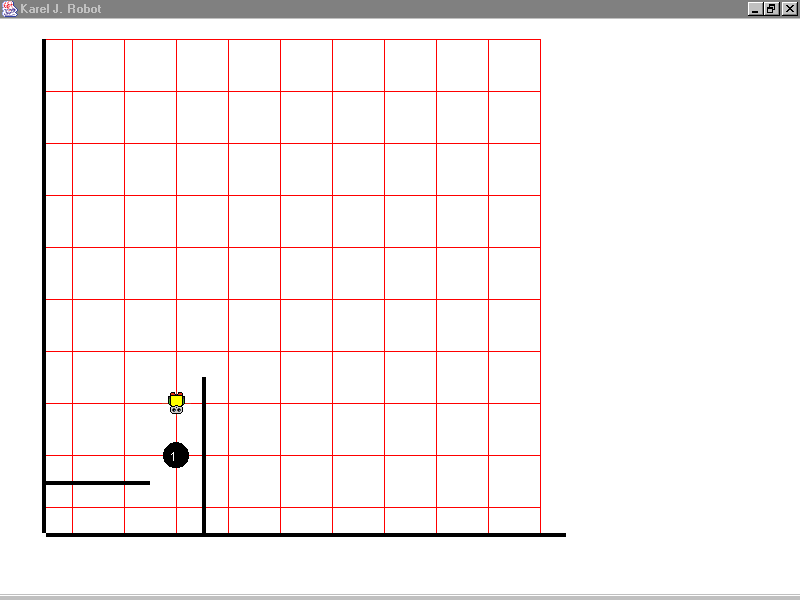
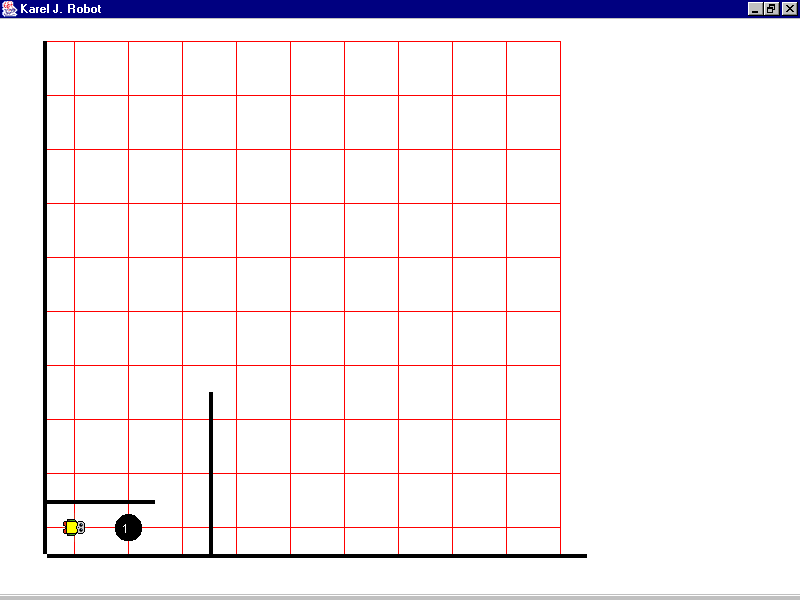
Line 6: The statements execute in order. This statement creates a World *object*, using the "first" map, and names that object as wld.

Line 7: The next statement creates a Robot object in the world, and names that object as karel.

Line 8: Next the code sends a message to karel, telling it to move. Every Robot object knows how to move because every Robot object was created using the Robot class as a template or blueprint. The parentheses are a marker for a *method.* In this case, the parentheses are empty because no information needs to be passed into move. Because of the code in move, when the program is run, karel will move forward one step.

Line 17: wld.mainloop is a mystery command that tells the wld object to keep the graphics window open on the screen until you press the red X. Don't worry about how it actually works.

Go to Save As. . . and save it as Lab00.py in the pyKarel folder. Run it and fix any errors. You will see a world and a karel robot following the commands, like this:

Sample Run

Start: End:

Exercises

Lab00

A Robot object’s *default* initial attributes are to start at (1, 1) facing east with zero beepers. If you don’t want the default settings, you may specify the x-coordinate, y-coordinate, direction, and number of beepers. These values are called *arguments.* In Python, arguments live inside parentheses. For example:

ophelia = Robot(wld); //uses the *default attributes*

horatio = Robot(wld, 5, 4, south, 37); //specifies the *initial attributes*

1. What attributes about herself does ophelia know?
2. What attributes about himself does horatio know?
3. What do both robots know how to do?

4) Write the commands to create a robot named pete starting at (4, 3) facing west with 50 beepers in the “home” world.

5) Write the commands to have lisa move one block, put down a beeper, then move another block. Since there are no arguments in the call to World(), the *default world* will be used. The default world is empty except for the two infinite-length walls bordering the south and the west.

wld = World();

lisa = Robot(wld, 7, 7, south, 15);

6) Write the code to make martha move forward five blocks and “hand-off” her beeper to george. Have george move forward two blocks and put the beeper down.

mv = World("MountVernon")

martha = Robot(mv, 1, 1, north, 1);

george = Robot(mv, 1, 6, east, 0);

Lab01

Students and Books

Objective

Learning about worlds and robots.

***Background***

Worlds have grids, numbered according to the Cartesian coordinate system. The corner at the bottom-left of the graphics window is (1, 1).

Worlds create the context for solving robot problems. Pre-defined worlds are stored in the folder pyKarel\worlds.

An *identifier* is the name of a class, an object, or a method. An identifier can be any unique sequence of numbers, letters, and the underscore character “\_”. An identifier cannot begin with a number. An identifier cannot be a Python keyword, such as class. Identifiers are case-sensitive, so lisa is not the same as Lisa. As a convention, only class names begin with an uppercase letter. Methods and variables by convention begin with a lowercase letter.

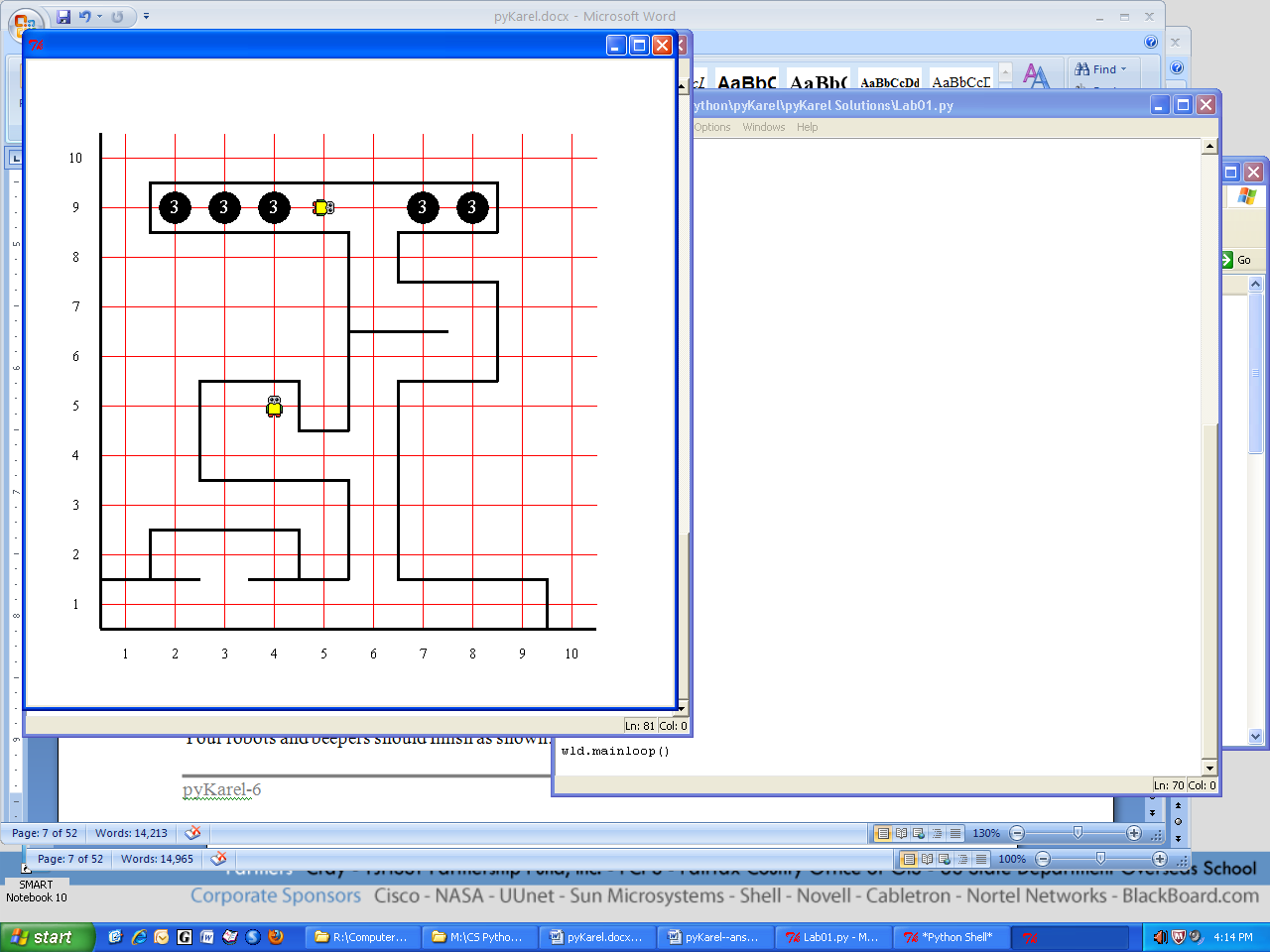
The word *instantiate* means to create an object from a class. We sometimes say the object is an *instance* of the class. For example, most of you are instances of the 9th grade class in this high school. In Python, we might instantiate three objects from two different classes with these commands:

high\_school = School("tjhsst")

you = 9thGrade(high\_school)

her = 9thGrade(high\_school)

The object “knows” how to do things in its *methods*. We can *call* or *invoke* a method, or *send a message* to a robot object, by using *dot-notation*. For example, a robot named pete will try to move forward with the call   
pete.move()



The identifier “pete”, before the dot, is the name of an object and the identifier “move()”, after the dot, is the name of a method. All methods in Python are marked by parentheses. Any data inside the parentheses, called an *argument*, is used by the method to do its job.

Specification

Create Lab01.py with the “school” world. Instantiate two Robot objects, one named lisa using the default constructor and the other named pete starting at (4, 5) facing south with zero beepers. Have lisa pick up the “book” from the math office and bring it to pete. Have pete take the book to the storage room and place it on the pile that currently has only two books. Then have pete step away from the pile.

***Sample Run***

Your robots and beepers should finish as shown:

Discussion

Errors

Whenever you press F5 in IDLE, the Python interpreter checks your work for errors and tells you the line on which your error occurs. One of the easiest errors to fix is a spelling error, such as Robt, robot, or Ribit. In IDLE, press ALT-G to help you go quickly to a specific line.

A second kind of error is a syntax error, meaning that it breaks the rules of forming valid commands in Python. It is like making a grammatical error. If you have studied a foreign language, you know that grammatical rules sometimes don’t make much sense. The rules are just the way it is. Accordingly, you must memorize the syntax for each Python command.

A third kind of error, a runtime error, occurs when the program runs. For example, telling a robot to walk through a wall, pick up a beeper that isn’t there, or place a beeper that it doesn’t have, will generate an error message and a line number. A good programmer learns to read and understand error messages.

A fourth kind of error, a *logic error*,occurs when your program executes all the commands, but still does not accomplish the task. You look at the screen and the results are not correct. A logic error is the programmer’s fault, usually due to unclear thinking, or to not understanding the problem deeply enough. You’ll be spending a lot of time correcting your logic errors.

It is very difficult to write a long program without any errors at all. To help yourself, get in the habit of pressing F5 after you write a few lines of code. Then you can more easily find and fix the syntax errors.

To help correct logic errors, you might try to *comment out* parts of the code. That way you can check portions of the code by itself. The commented out portions will turn red (in IDLE), and the compiler ignores those lines. You can either use # at the beginning of each line, or you can highlight some code and press Alt-3. To uncomment the code, highlight and press Alt-4. That’s nice.

Lab01 continued

Using Lab01, create these errors and write down what error messages, if any, they generate. Then decide if the error is a *syntax error* or a *runtime error*.

|  |  |  |
| --- | --- | --- |
| **Common Errors** | **Python's Error Message** | **Syntax or runtime error?** |
| press <space> or <tab> to indent one line |  |  |
| Misspell lsa |  |  |
| type lisa.move |  |  |
| type lisa.move( |  |  |
| type only move() |  |  |
| type only Robot(4, 5, south, 0) |  |  |
| Omit from pyKarel import \* |  |  |

Finally, comment out the lisa Robot and all lisa’s commands. What happens to pete?

The process of finding errors and correcting them is called “debugging” and the errors themselves are called “bugs.” In 1951, Grace Hopper, who would eventually rise to the rank of Rear Admiral in the United States Navy, discovered the first computer “bug.” It was a real moth that died in the circuitry. Hopper pasted the moth into her UNIVAC logbook and started to call all her programming errors “bugs.”

Exercises

Lab01

|  |  |  |  |
| --- | --- | --- | --- |
| mrsAdams | mr.chips | class | r2\_d2 |
| c3p-o | Hal9000 | 7\_of\_9 | robot |

1) Circle the identifiers (names for variables) that are legal. Put a star by the identifiers that by convention identify a class.

1. All the relevant information about a class is contained in its API (application programming interface) document, usually an html file that is on-line. For convenience, part of the pyKarel API is pasted here.
2. What modules are used by pyKarel?
3. What classes are defined in pyKarel?
4. What Robot commands have we used so far?
5. What Robot command looks interesting?
6. The \_\_init\_\_ method in Robot is called the *constructor*. We will talk about constructors soon. What do you think the Robot class does with self, world, x=1, y=1, direction= 0, beepers=0 ?

Discussion

Calling Constructors

We said that an object is a container for both data and methods. Each robot-object needs five items of data to specify itself. Those items are a world, an x-coordinate, a y-coordinate, a direction, and the number of beepers. A *constructor* is a special method in each object that gets the object up and running. In more formal language, we say that the constructor *initializes* the object’s attributes.

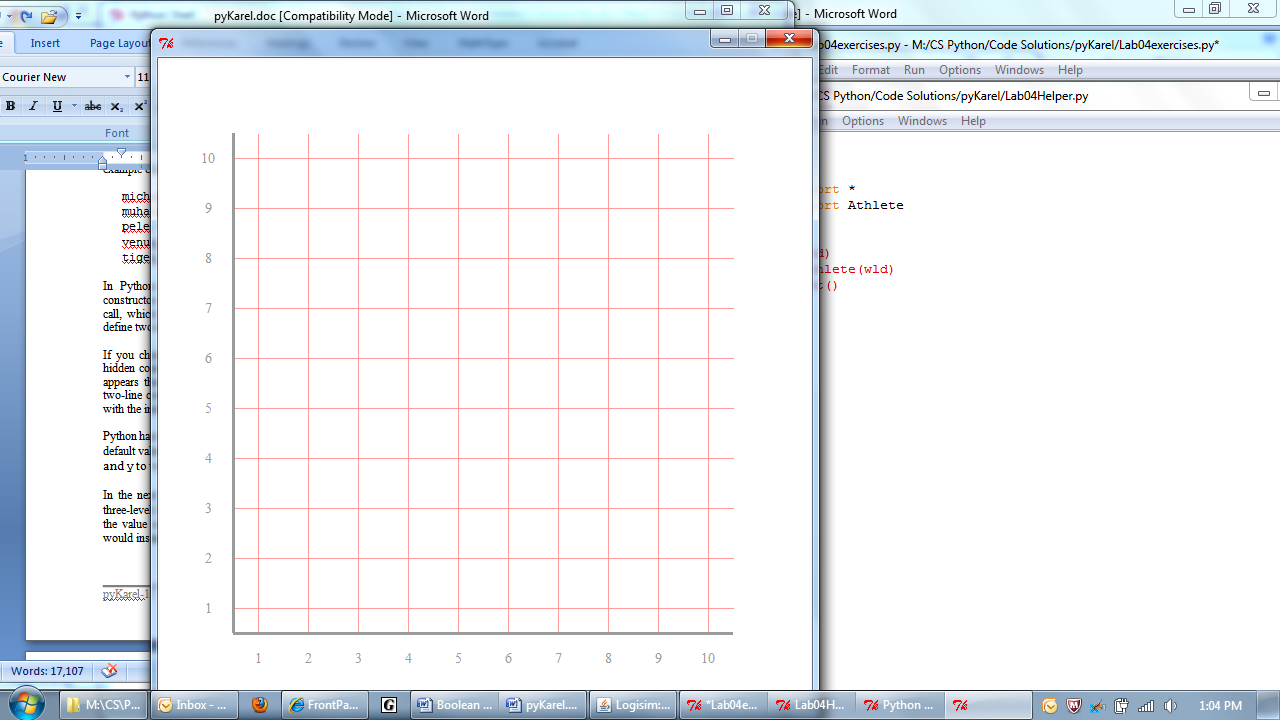
You have seen several *calls* to constructors:

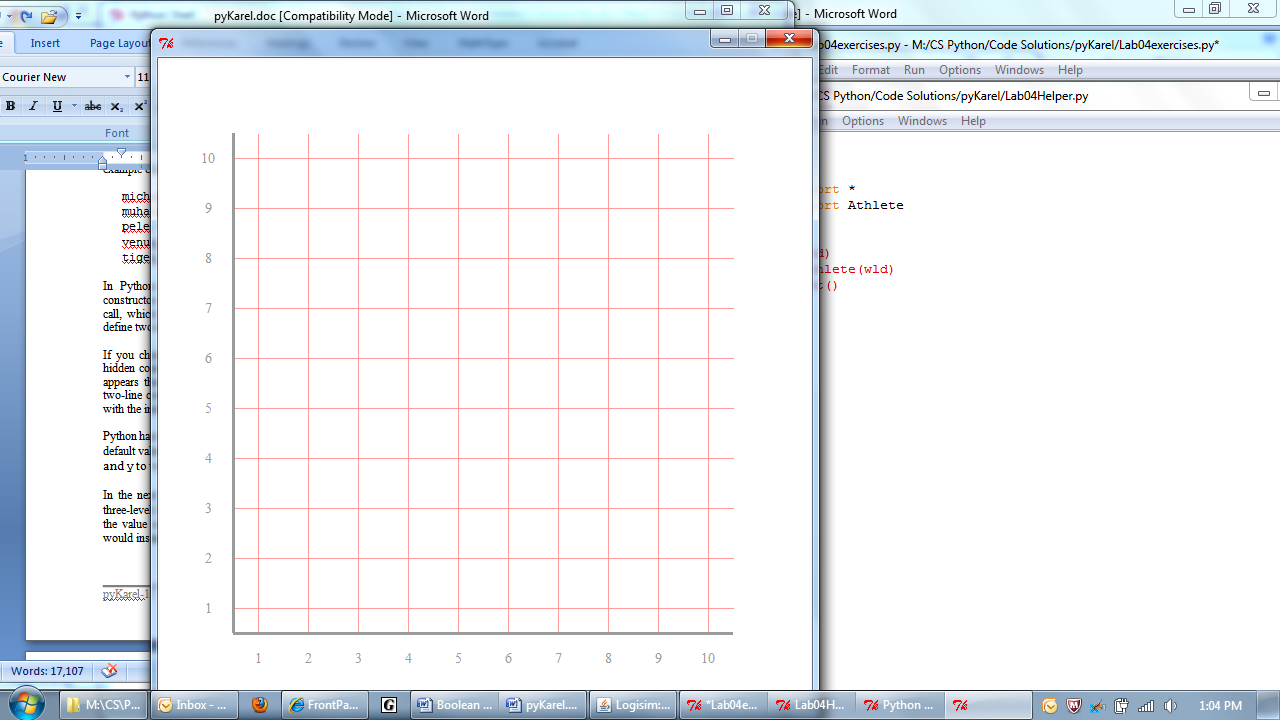
wld = World("first")

karel = Robot(wld)

horatio = Robot(wld, 5, 4, south, 37);

Calls to constructors are designed to be flexible and useful. Draw each Robot at its place in the correct world:



mv = World("first", width=3, height=6)

wld = World(width=8, height=8)

lisa = Robot(wld)

pete = Robot(wld, direction=north)

joan = Robot(wld, beepers=infinity)

rohan = Robot(wld, y=5)

martha = Robot(mv, 2, 1, north, 1);

george = Robot(mv, 2, 6, east, 0);

Every call to the Robot constructor works because of the code in the pyKarel module, which defines Robot and World. The Robot class defines x, y, direction, and beepers, which are the Robot *attributes*. The World class defines the attributes width and height, among others. Five pyKarel attributes are defined in the first line of code in pyKarel using this unusual syntax:

east, north, west, south, infinity = 0, 90, 180, 270, -1

The syntax means that variable east is assigned the value 0, north gets 90, and so on. Notice that you can use east, etc., without knowing why these numbers were chosen. The implementation details are hidden from you, which is a good programming principle. On the other hand, if you are curious, you can always look at the pyKarel code and try to figure out why the programmer made east=0, north=90, and so on.

All this useful code is made available to each robot program because each application begins with:

from pyKarel import \*

Exercises

Calling Constructors

1. Create a Lab01Exercises.py file. Experiment with calling some World constructors with different arguments. Then experiment with calling some Robot constructors with different arguments.

Lab02

Escape the Maze

Objective

Inheritance. Defining instance methods.

Background

How do we make Robots turn right? In object oriented programming, we usually do not change the Robot class. Instead, we extend Robot and define a new class. Then we give the new class the power to turn right by defining a method. In a sense, we “teach” an Athlete how to how to turn right. Study the syntax and structure of the Athlete class.

from pyKarel import **\***  #tells where Robot is

class Athlete(Robot): #adds new powers to Robot

def turnRight(self): #defines new power on self

#

# your code goes here

#

def turnAround(self):

#

# your code goes here

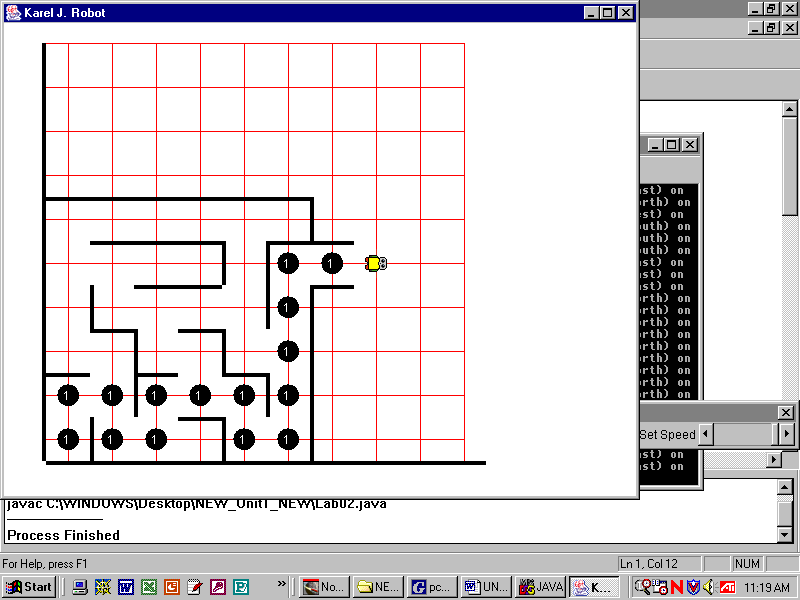
#

class Athlete(Robot): defines a new class that *extends* Robot, meaning that the Athlete class *inherits* the behaviors and attributes of the Robot class; the methods from Robot do not have to be re-written. When you see class Athlete(Robot): you may think of adding Athlete’s powers to the Robot class.

When you see def turnRight(self): you may think of adding new powers to the self object, which was created up in Robot. Since self.turnLeft() was also in Robot, we may call self.turnLeft() in our definitions of turnRight and turnAround. Indeed, because of inheritance, any Athlete object has access to all the Robot methods.

Athlete is a general *resource class* that will be useful in a wide range of applications. The *application* called Lab02.py will instantiate and use the Athlete class. We say that Lab02 *hasa* Athlete. On the other hand, because of inheritance, we say that Athete *isa* Robot. "Hasa" and "isa" are good Java jargon words.

Specification

Create Athlete.py as given above. *Implement* the methods turnRight and turnAround.

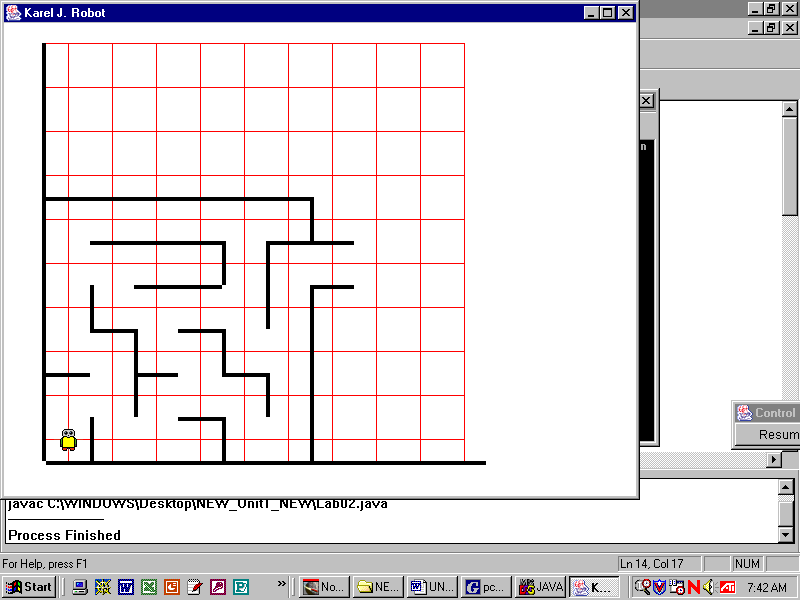
Pressing F5 (in IDLE) will check its code for errors, but nothing will run.

Create Lab02.py with “maze” world at width=8, height=8. Instantiate one Athlete object in this world with beepers=infinity and give it the commands to escape the maze. Leave a trail of beepers to mark the path, as shown.

Pressing F5 (in IDLE) will check and run the code.

***Sample Run***As shown.Exercises

Lab02

Examine the robot-map shown.

wld= World("maze")  
maria = Athlete(wld, beepers=infinity)

1) Write the commands to put one beeper at the indicated location.

2) Lab01 was written using an application and two classes. Label the boxes (it’s called a UML diagram), showing the relationships. Label the arrows with “hasa.”

3) Lab02 was written using an application and three classes. Label the boxes in the UML diagram, showing the relationships. Label the arrows with “isa” or “hasa.”

Discussion

Client vs. Server

In Lab01 you wrote an application that used two classes (World and Robot). In Lab02 you wrote an application that used three classes (World, Robot, and Athlete) that included inheritance. The application instantiated the objects and sent them commands while the classes contained the code executing those commands. Using a different terminology, we could say that the application is the *client* and the classes are the *server*. The client sends messages or commands while the server knows how to respond to those messages.

Notice also that the application did not have to be instantiated to be used. We could just run the application. We cannot just run classes and get behaviors, because classes have to be first instantiated as objects. After the application (or client) instantiates the objects, then it can send messages to that object (or server). The object executes the command according to its code. In a sense, the application does not “know” how the object will behave.

|  |  |  |
| --- | --- | --- |
| COUNT | 1 |  |
| x | 1 |  |
| y | 1 |  |
| direction | 0 |  |
| beepers | 0 |  |
|  | .x | 1 |
| self | .y | 1 |
|  | .d | 0 |
|  | .beepers | 0 |
| world |  |  |
|  |  |  |
| east | 0 |  |
| north | 90 |  |
| west | 180 |  |
| south | 270 |  |
| infinity | -1 |  |

Discussion

Data Types

A *data type* specifies how the different kinds of data, e.g., numbers, words, or Robots, are stored and manipulated. (This is a big topic in computer science. Here we just introduce the topic.) Obviously, we want Python to treat numbers differently from words. It is not so obvious that Python, like many other programming languages, treats numbers in two different ways, as integer for integers and float for decimals. Words or text are stored in a string. A boolean type stores true-false.

A *declaration* creates a *variable* and often assigns a *value.*  You can think of a variable as a name or label for the value. The assignment symbol is “=”. For example, if we look at the Robot class (either on page-8 or directly in pyKarel.py) we see:

COUNT = 1

self, world, x=1, y=1, direction= 0, beepers=0

Every Robot object keeps track of all its personal data with these other assignment statements:

self.x, self.y, self.d, self.beepers = x, y, direction%360, beepers

The pyKarel module also declares many *global variables*, including:

east,north,west,south,infinity=0,90,180,270,-1

Finally, here is an example of a float declaration (float is short for “floating-point”): pi = 3.14159

Classes are *user-defined types* that *encapsulate* data and methods, i.e., an object’s data and methods are private and wrapped up inside the object. An instantiation of an object creates a *variable* that points to the object. The code karel = Robot(wld); does three things: creates a variable (karel), instantiates an object (Robot()), and assigns (the "=" sign) the variable to point to the object. You should imagine the variable pointing to object, like this:

|  |  |  |  |
| --- | --- | --- | --- |
| karel |  |  | A Robot Object |

Two or more variables can point to a single object. We will see this in later labs.

|  |
| --- |
| Robot |
|  |
| Athlete |

Discussion

Writing Constructors

We have made Athlete inherit from, or extend, Robot. One way to visualize the relationship is by a UML diagram. We say that Athlete *isa* Robot. The Athlete inherits self, move, and turnLeft and then goes on to define two new methods turnRight and turnAround.

**Athlete**

turnRight()

turnAround()

**Robot**

move()

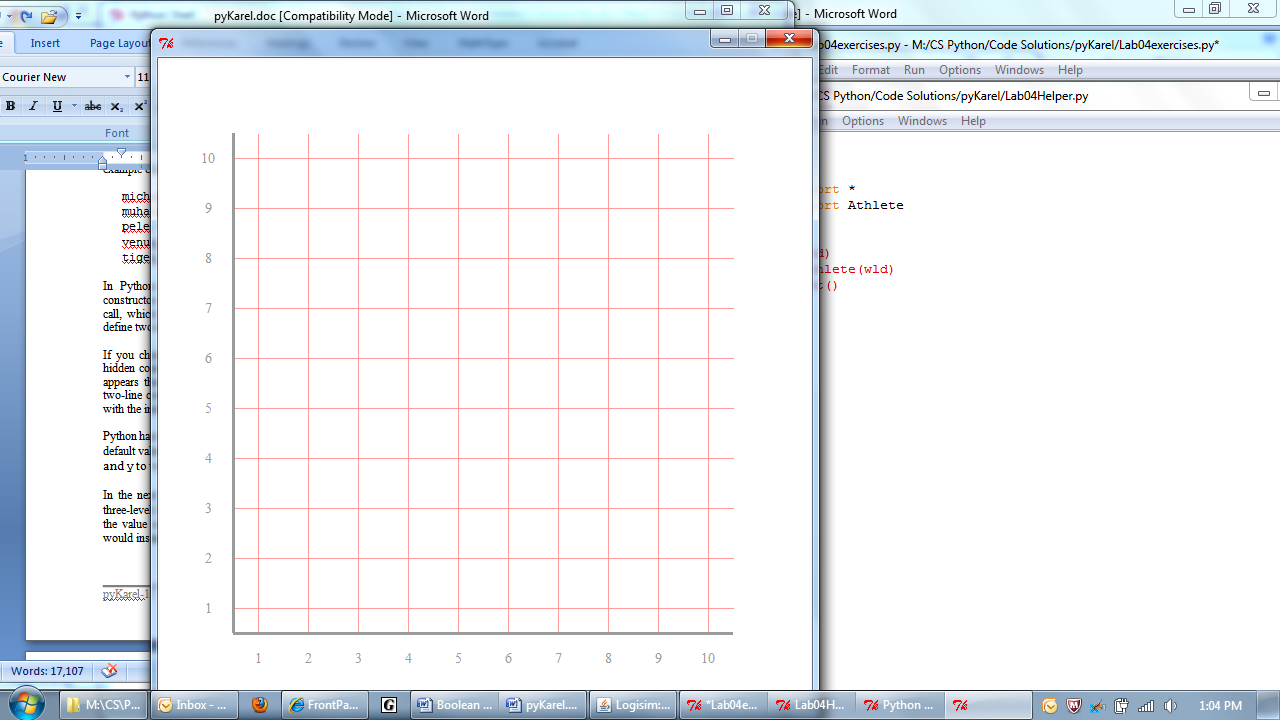
turnLeft()

Another way to visualize the relationship is to imagine a Robot object “inside” Athlete. When we construct an Athlete, the code has to construct a Robot first, then add the new methods. We can see that Athlete’s *constructor* calls Robot’s constructor:

def \_\_init\_\_(self, w, x=1, y=1, d=north, b=infinity):

Robot.\_\_init\_\_(self, w, x, y, d, b)

The *constructor* is the method that creates an object and *initializes its attributes*. Since Athlete extends Robot, all that those attributes are *passed* to the Robot.\_\_init\_\_ command, which then does a lot work setting up the object. The variables need to match up correctly, self with self, w with w, x=1 with x, and so on. As you see, the *default values* for an Athlete object are 1,1,north,infinity.

Now let’s call some constructors. What attributes do these Athletes have?

michael\_jordan = Athlete(wld)

muhammad\_ali = Athlete(wld, b=0)

pele = Athlete(wld, 2, 8)

serena\_williams = Athlete(wld,3,5,south)

These examples show that some or all of the Athlete's default values may be *overridden* in the call. Where is self? Every Athlete automatically has a self object that was created in Robot and never has to be passed here. On the other hand, w is a world and must get wld explicitly. Passing one world is all you need to get a default Athlete object.

If the programmer wants, it is possible to write the Athlete constructor which does not allow overriding values:

def \_\_init\_\_(self, w):

Robot.\_\_init\_\_(self, w, 1, 1, north, infinity)

Since self gets assigned automatically, but w needs to be passed explicitly, there is only one way to call this new constructor: tiger\_woods = Athlete(wld). As you see, the default values for this Athlete object are still 1, 1, north, infinity.

In the next lab we will write a Climber class, which extends Athlete. Then we will have a three-level hierarchy. Each constructor needs to call the \_\_init\_\_ of the class above it, passing the values up the hierarchy as needed. Eventually, all the \_\_init\_\_ calls in the hierarchy would create the self, the world, and all the variables and methods that make up the Climber object.

However, Python allows you not to worry about that, because constructors are inherited in Python. To see an example, you should “comment out” Athlete’s constructor, so that Athlete appears not have a constructor. When you run Lab02.py, notice that the Athlete still knows two new methods, but it starts out with the initial attributes of a Robot. To instantiate an Athlete in Athlete’s default attributes, we could override the Robot’s default attributes with the call: karel = Athlete(wld, 1, 1, north, infinity)

Indeed, no lab in this packet requires you to write an \_\_init\_\_ method. Your teacher can decide.

Lab03

Climb Every Mountain

Objective

Writing a new class in our Robot hierarchy.

|  |
| --- |
| Robot |
|  |
| Athlete |
|  |
| Climber |

***Background***

A Climber is a specialized Athlete which climbs mountains and finds treasure. Climbers always start on the row y=1 facing north with one beeper, but at any given x-position. In order to specify the starting x-position, the Climber class will define a constructor that requires two arguments. (Recall that constructors are special methods that create an object and initialize its attributes using the \_\_init\_\_ command.)

class Climber(Athlete):

def \_\_init\_\_(self, world, x, y=1, direction=north, beepers=1):

Athlete.\_\_init\_\_(self, world, x, y, direction, beepers)

def climbUpRight(self):

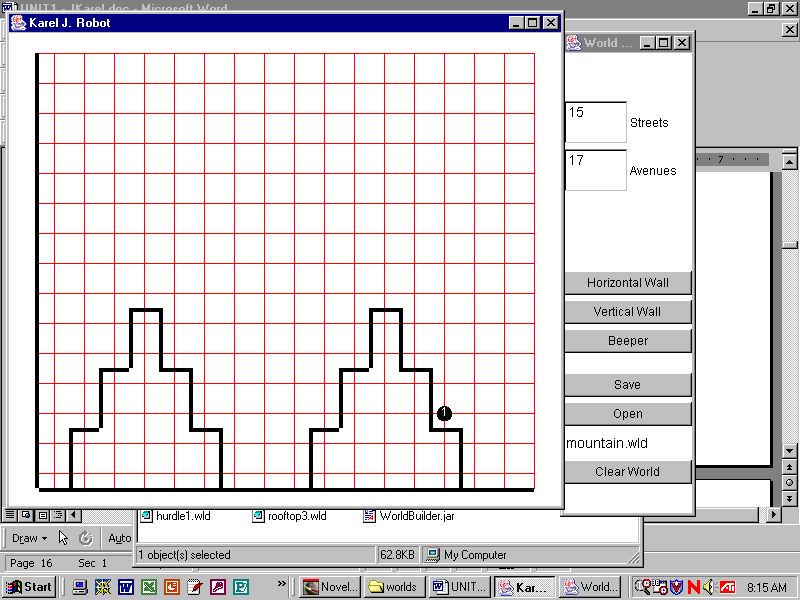
#

# your code goes here

#

# define more methods here

We might instantiate a climber with tenzing = Climber(wld, 100). Note that we must pass two arguments, the world and an x-position, but not self, to Climber's constructor.



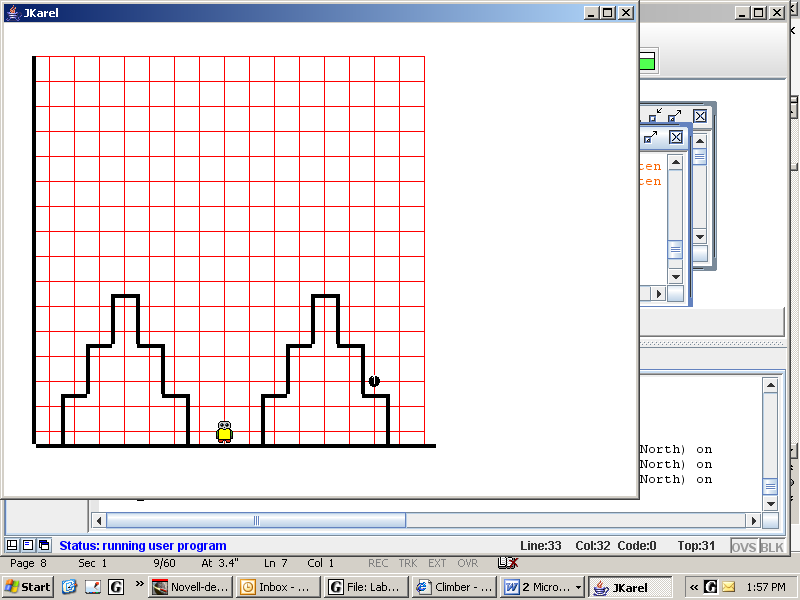
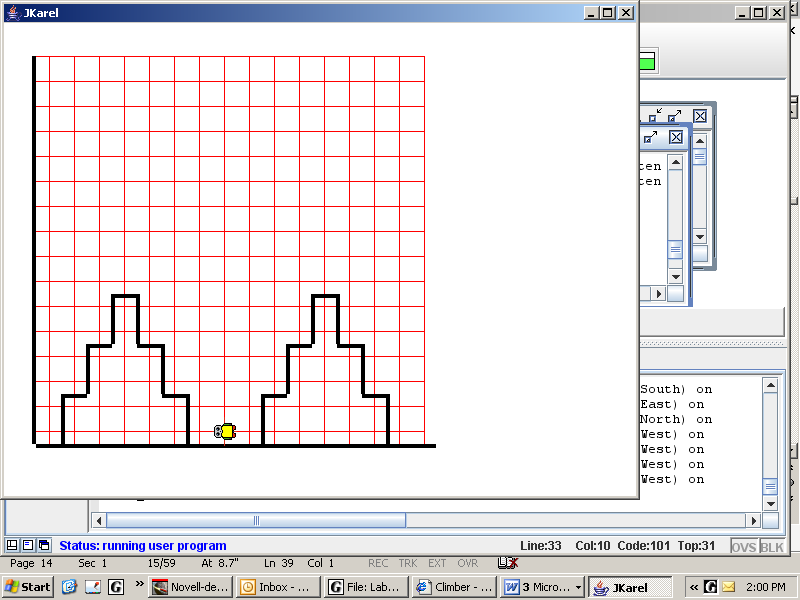
The Climber's instance methods all have to do with climbing the mountain. Each   
method should work for one step only. Plan how you will teach the climber to climb up one step, going to the right.

Specification

Create Climbers.py. Because we will eventually make several kinds of Climbers, let’s name the file “Climbers.py”. Import Athlete. Then define the Climber class with a constructor and four instance methods climbUpRight, climbDownRight, climbUpLeft, and climbDownLeft.

Create Lab03.py with the “mountain” world at width =16, height =8. Instantiate two climbers starting at x-coordinate 8. Have one climber put down a beeper, which marks the base camp. The beeper-treasure is always in the same place. Staying together, have the two climbers bring the treasure back to the base camp.

Sample Run:

 Start: End:

Exercises

Lab03

1. Notice the class from which Elf inherits. Write a constructor for Elf. Default elves begin at (1, 9), face south, and carry an unlimited supply of beepers.

**class** Elf(Robot):

2 Instantiate an Elf in the default state.

3. Notice which class Spartan extends. Write a constructor for Spartan that takes arguments specifying the position on which to start. Default Spartans face east with one beeper.

**class** Spartan(Athlete):

4**.** Instantiate a Spartan beginning at (3,3) and 1 beeper, but facing north.

Discussion

Passing Arguments

|  |  |
| --- | --- |
| wld |  |
| x | 100 |
|  |  |
|  |  |
| y | 1 |
| d | 90 |
| b | 1 |

We said above that we might instantiate a climber with tenzing = Climber(wld, 100);. The stuff inside the parentheses is called an *actual argument*. The application then goes to the constructor, sees the \_\_init\_\_(wld, x), creates room for wld and x, and assigns the 100 to the x. The variable x is called the *formal argument.* Whenever the x is used after that, its value will be 100. Similarly, when the call to Athlete.\_\_init\_\_(wld, 1, north, 1) is executed, then those four values are *passed* up to Athlete. There, the x is 100, the y is 1, the d is north, and the b is 1. The actual arguments must match up with the formal arguments, or Python will give an error. Whenever we write or call a method, we need to think about the method’s arguments, both their type and their number.

|  |  |  |
| --- | --- | --- |
| maria |  |  |
|  |  |  |
|  |  | Athlete |
|  |  |  |
| arg |  |  |

In the next lab, we will be *passing objects* as arguments with takeTheField(maria); The computer then goes to the method’s code, sees def takeTheField(arg): and creates and assigns a new *local variable*. Because of the way that Python treats objects, both maria and arg point to the same object. In effect, the athlete has two names. Commands sent to arg will behave just like commands sent to maria.

Lab04

Take the Field

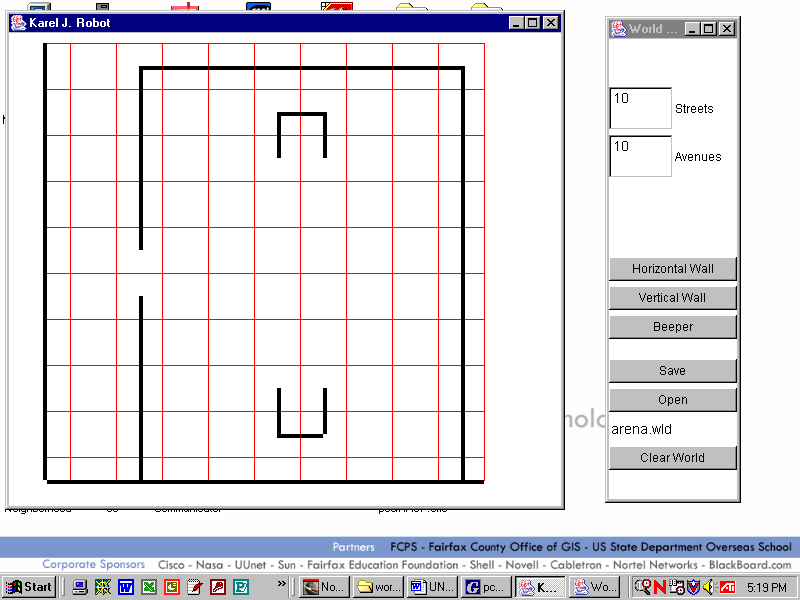
Objective

Instance methods vs. application methods

Background

One issue in object-oriented programming is to decide where to put the methods. Sometimes it is appropriate to put the methods in the object. We call these *instance methods*. The object “knows” how to do things. The object must be instantiated before being used. For the Athlete class, the programmer decided that every Athlete should know how to turnRight.

In contrast, in Lab04, the programmer decided that taking the field is something that is specific to this lab only. Therefore, the programmer chose to define takeTheField in the application code. All the athletes in Lab04 will get *passed* into takeTheField and follow the commands that are there.



A

B

**from** pyKarel **import** \*

**from** Athlete **import** Athlete

**def** takeTheField(arg):

arg.move()

arg.move()

arg.move()

arg.move()

arg.turnRight()

arg.move()

arg.move()

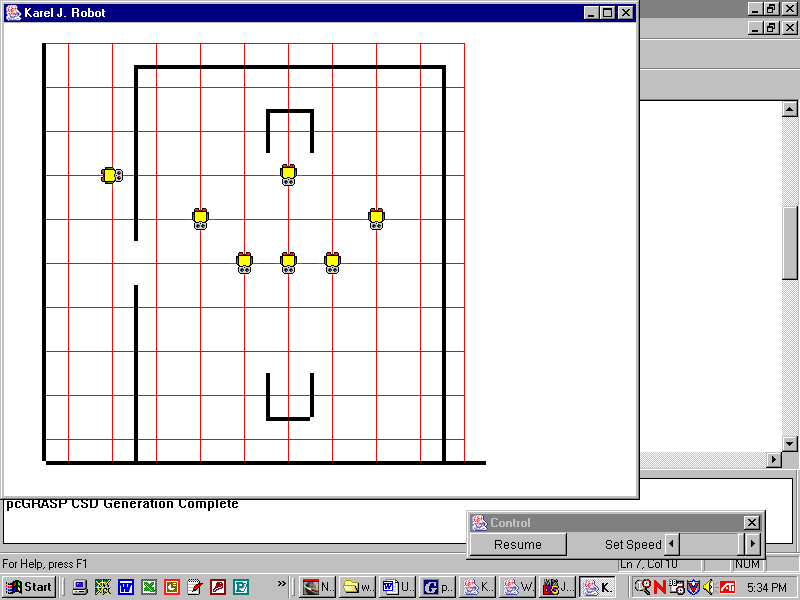
# start Lab04 application code here.

We can use this takeTheField with any athlete object by passing an athlete object. For example, after we instantiate athletes maria and gary, we can tell each to take the field with the commands:

takeTheField(maria);

takeTheField(gary);

Notice we do NOT say maria.takeTheField(). We do not use dot-notation because takeTheField is not an instance method in the Athlete object. Instead, we call takeTheField and pass the object to the argument arg, which points to whatever Athlete has been passed; first to maria, then to gary.



Specification

Create Lab04.py with “arena” world at the default size. Create six athletes starting in the locker room; call the default constructor. Create one coach at 2, 7, facing east. Position the athletes for the start of the game as shown.

Exercises, Part 1

Lab04

Each command #1 through #8 in Lab04Exercises.py will produce an error. In your imagination, "comment in" that line, then predict the error and tell how to fix it. Then "comment out" that line and "comment in" the next line. Predict the error and tell how to fix it. Repeat.

**Lab04Exercises.py**

**Lab04Helper.py**

**from** pyKarel **import** \*

**from** Athlete **import** Athlete

wld = World("arena")

c3p\_o = Robot(wld)

babe\_ruth = Athlete(wld)

#1 c3p\_o.turnRight()

#2 c3p\_o.putBeeper()

#3 babe\_ruth.pickBeeper()

#4 babe\_ruth.turnLeft()

#4 babe\_ruth.move()

#5 babe\_ruth.takeTheField()

#6 takeTheField(babe\_ruth)

#7 c3p\_o.turnLeft()

#7 takeTheField(c3p\_o)

#8 takeTheField(barry\_bonds)

#9 takeTheField(babe\_ruth, c3p\_o)

wld.mainloop()

**from** Athlete **import** Athlete

**def** takeTheField(arg):

arg.move()

arg.move()

arg.move()

arg.move()

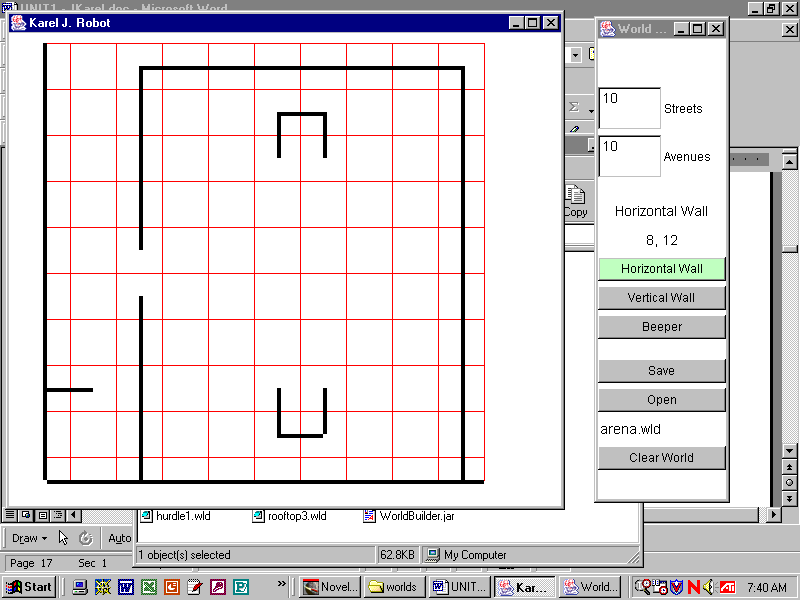
arg.turnRight()

arg.move()

arg.move()

Exercises, Part 2

Lab04

Examine the robot-map shown. Note that this is not exactly the same arena world from Lab04. Our facilities are being renovated and the construction crew has temporarily blocked off our normal passageway; there’s a horizontal wall north of (1, 2) that wasn’t there originally.

1. Describe how you would have to change your Lab04 application so that it would work in this world.

2. Write down the complete code of all methods that you changed, starting with the *header* for that method and including the entire method *body*.

3. *Modular design* is the practice of breaking larger problems into smaller, more manageable pieces. Explain how modular design helped us maintain our Lab04 program when the robot-map changed.

Discussion

Class Hierarchy

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | Robot | |  |  |
|  |  |  |  |  |  |
|  |  | Athlete | |  |  |
|  |  |  |  |  |  |
| Climber | |  |  | Racer | |

In the class hierarchy, or inheritance hierarchy, shown in the UML diagram at the right, we say:

* Robot is the superclass of Athlete and Athlete is the subclass of Robot.
* Athlete is the superclass of Climber and Climber is the subclass of Athlete.
* Athlete is the superclass of Racer and Racer is the subclass of Athlete

Other names for a superclass are base class or parent class. Other names for a subclass are derived class or child class. Since a climber isa athlete and an athlete isa robot, we can also say that a climber isa robot. Likewise, since a racer isa athlete and an athlete isa robot, we can say that a racer isa robot. Not only do the subclasses inherit methods and fields, but they also inherit the name of the superclass.

1) For each attribute, circle the classes that **must** have that attribute.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A. Athletes have green feet. | Robot | Athlete | Climber | Racer |
| B. Climbers have blue scales. | Robot | Athlete | Climber | Racer |
| C. Robots have curly tails. | Robot | Athlete | Climber | Racer |
| D. Racers have sharp teeth. | Robot | Athlete | Climber | Racer |

2) Given the declarations below, circle all the commands that will successfully execute the code.

karel = Robot(wld)

gary = Athlete(wld)

lisa = Climber(wld)

mary = Racer(wld) // Racer defines jumpEast(self)  
runAway(arg)

|  |  |  |  |
| --- | --- | --- | --- |
| karel.move() | gary.move() | lisa.move() | mary.move() |
| karel.turnRight() | gary.turnRight() | lisa.turnright() | mary.turnRight() |
| karel.turnLeft() | gary.turnLeft() | lisa.turnLeft() | mary.turnLeft() |
| karel.climbUpRight() | gary.climbUpRight() | lisa.climbUpRight() | mary.climbUpRight() |
| karel.jumpEast() | gary.jump() | lisa.jumpEast() | mary.jumpEast() |
| karel.runAway() | gary.runAway(gary) | lisa.runAway(self) | mary.runAway(arg) |
| runAway(karel) | runAway(gary) | runAway(self) | runAway(mary) |

Discussion

Loops

We often want to repeat (or *iterate*, or loop) either one command or a sequence of commands. For instance:

|  |  |
| --- | --- |
| karel.move()  karel.putBeeper()  karel.move()  karel.putBeeper()  karel.move()  karel.putBeeper()  karel.move()  karel.putBeeper()  karel.move()  karel.putBeeper()  karel.move()  karel.putBeeper() | pete.pickBeeper()  pete.pickBeeper()  pete.pickBeeper()  pete.pickBeeper()  pete.pickBeeper()  pete.pickBeeper()  pete.pickBeeper()  pete.pickBeeper()  pete.pickBeeper()  pete.pickBeeper() |

In the first example, we want to repeat moving and putting exactly 6 times. In the second example, we want to pick exactly 10 beepers. Since we know beforehand exactly how many times to iterate (*definite iteration*), we will use a *for-loop.* (We will study *indefinite iteration*, the while-loop, in Lab06.)

|  |  |
| --- | --- |
| for k in range(0,6):  karel.move()  karel.putBeeper()  Notice the indentation! This loop will cause karel to move and put down a beeper six times (once when k has each of the values 0, 1, 2, 3, 4, and 5). | for k in range(0,10):  pete.pickBeeper()  This loop will cause pete to pick up a beeper ten times (once when k has each of the values 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). |

Memorize the syntax of a for-loop. Each for-loop names a counter variable and specifies a range. If there are two numbers in the range (as shown above), the loop counts from the first number up to but not including the second number. If there is one number in the range (as shown below on the left), the loop counts from 0 up to but not including the second number. The for-loops shown below result in the same output as those shown above:

|  |  |
| --- | --- |
| for a in range(7):  karel.move()  karel.putBeeper() | n=100  for i in range(n,n+10):  pete.pickBeeper() |

However, you should of course write your for-loops to be a clear to the reader as possible.

Lab05

Shuttle Run

Objective

for-loops.

**class** Racer(Athlete):

**def \_\_init\_\_(**self,world,y):

Athlete**.\_\_init\_\_(**self,world,1,y,east)

**def** jumpEast(self):

# your code goes here

**def** jumpWest(self):

# your code goes here

**def** sprint(self, n):

**for** x **in range**(0,n):

self.move()

**def** put(self, n):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**def** pick(self, n):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Background

Racers always start on the left side at x=1, facing east, with an infinite number of beepers, but they may begin on any y-coordinate. Study the Racer constructor.

Racers also improve upon the powers of Athletes and Robots. Racers will be able to jump east or west over a hurdle, move many steps at a time, pick up piles of beepers, and put down piles. These three instance methods will all use for-loops and pass a variable. sprint is done for you; you must complete the others.

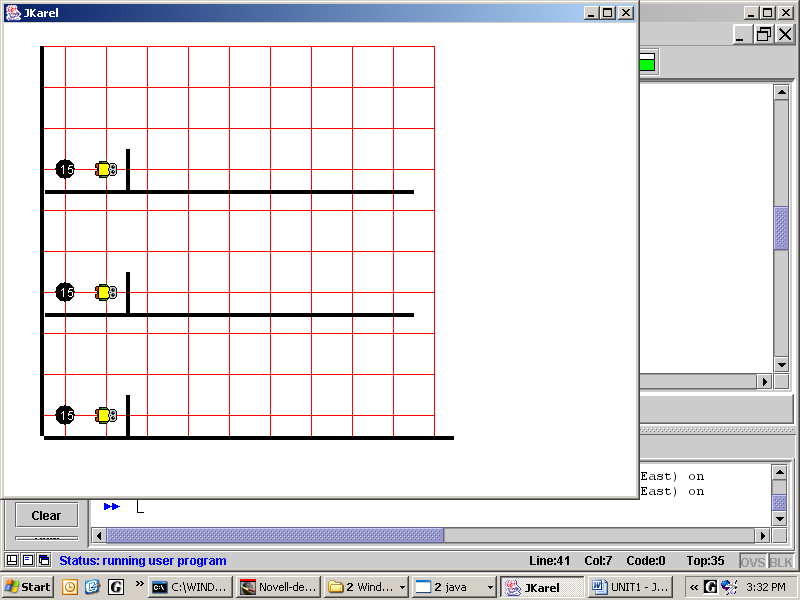
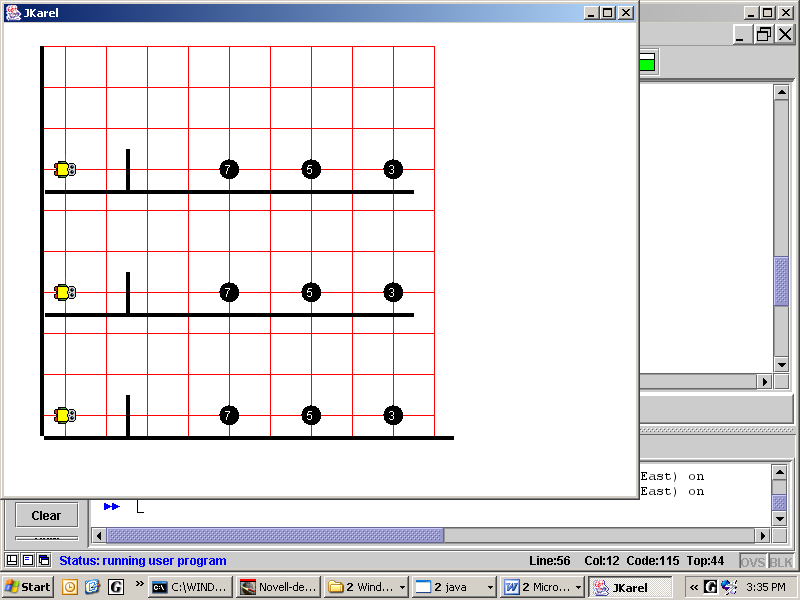
If pete is a Racer, then pete.sprint(100); will move pete way off the screen. How does the sprint method work?

Since the racers will complete the same task, which depends on the world, this is a perfect place to define a method in Lab05, not in the Racer class. Let's define runTheRace(arg)and pass each Racer to it.

Specification

Since we are eventually going to create several Racers, create Racers.py. Implement the Racer class.

Create Lab05.py with “shuttle” world at the default size. Instantiate three racers to run the shuttle run. It’s not a race. Let one racer finish the task, then send each of the others. Define a method in the application. Each pile of beepers must be brought individually back to the racer’s starting point. At the end each racer must step away from the pile to confirm that all 15 beepers have been retrieved.

***Sample Run*** Start: End:

Discussion

Loops and If

Any for-loop can be re-written as a while-loop.

j=0

**while** j<n:

self.turnLeft()

j+=1 # j = j + 1

**for** j **in range**(0,n):

self.turnLeft()

Both loops as written will try to make karel turn left for *n* times, no matter what. However, it has become conventional to use the for-loop for definite loops and the while-loop for indefinite loops, where you don’t know beforehand exactly how many times to loop.

The while-loop below is a better example of an indefinite loop, for we don’t know when it will stop. It keeps picking up a beeper as long as nextToABeeper returns True. The result is to make a robot that is standing on top of a pile to pick up that entire pile of beepers.

**while** karel.nextToABeeper():

karel.pickBeeper()

An if-statement is similar to a while-loop, but only makes one decision. The if-statement below will cause karel to pick up one beeper if the robot is currently standing on top of one. If karel is not currently standing on top of a beeper, the code just moves on. Thus, the if-statement acts as a guard to prevent the program from crashing in certain situations.

**if** karel.nextToABeeper():

karel.pickBeeper()

Warning

There is no such thing as an if-loop. An if-statement checks its condition only once.

It is useful to distinguish between methods that take action, such as turnRight, and those that provide information about a robot’s situation. The robot’s instance method nextToABeeper *returns* either True or False, depending on its situation. The values True or False are the two *boolean* values. Think of booleans as answering yes-or-no questions.

|  |
| --- |
| karel.frontIsClear() returns True |
| karel.frontIsClear() returns False |

Another useful boolean method defined in the Robot class is frontIsClear, which returns whether or not a wall blocks a robot’s path. Look at the examples:

There is no Robot method frontIsBlocked. However, Python has the not operator which follows the rules not true is false and not false is true. The not operator goes in front of the whole dot-notation expression. For example, in a while-loop, we might write:

while not karel.frontIsClear():

karel.turnLeft()

What does the code above do?

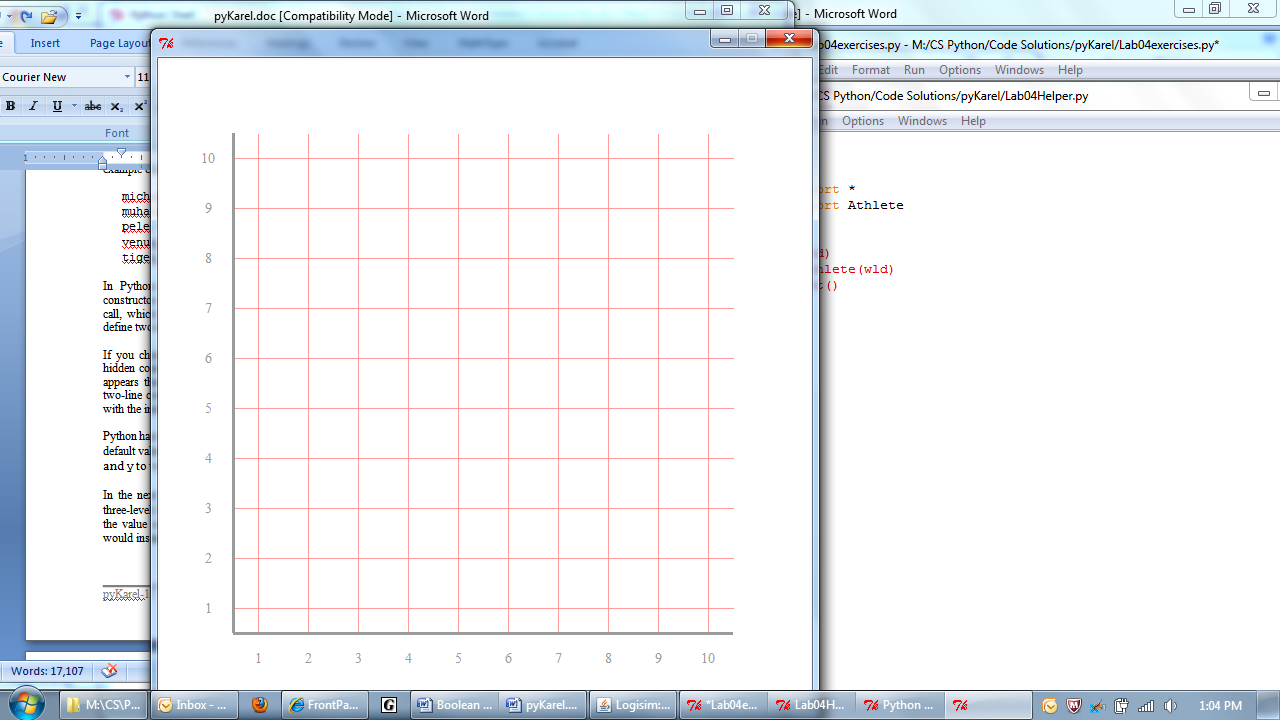
Exercises

Lab05

Look at the Robot’s API to see what boolean methods might be helpful here.

|  |  |
| --- | --- |
| 1. Write the commands to force a Robot object named karel to face west no matter what direction it is initially facing. | 2. Write the commands to make an Athlete, named ann, to put down all her beepers. |
| 3. Write the commands for Climber c to pick up a pile of beepers. | 4. Write the commands for Racer ray to stop moving when it is next to (on top of) another robot. |

5. Given the declarations and the default world, indicate the boolean value of each statement.

batman = Robot(wld, 5, 1, west, 37);

robin = Robot(wld, 2, 6, south, 0);

|  |  |  |
| --- | --- | --- |
| **not** batman.frontIsClear() | True | False |
| batman.hasBeepers() | True | False |
| **not** robin.hasBeepers() | True | False |
| **not** batman.facingWest() | True | False |
| robin.facingWest() | True | False |

1. Python provides a logical and and a logical or. Evaluate:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| True and True | 🡪 |  |  | True or True | 🡪 |  |
| True and False | 🡪 |  |  | True or False | 🡪 |  |
| False and True | 🡪 |  |  | False or True | 🡪 |  |
| False and False | 🡪 |  |  | False or False | 🡪 |  |

1. Extra Credit: make a truth table that shows that De Morgan’s Laws are true:

**not**(a **and** b) == (**not** a **or** not b) **not**(a **or** b) == (**not** a **and not** b)

Lab06

A Half-Dozen Tasks

Objective

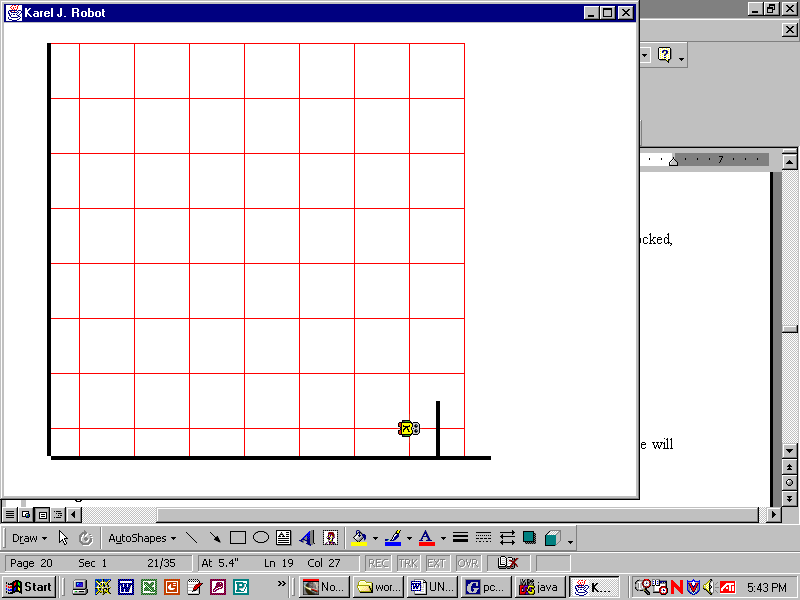
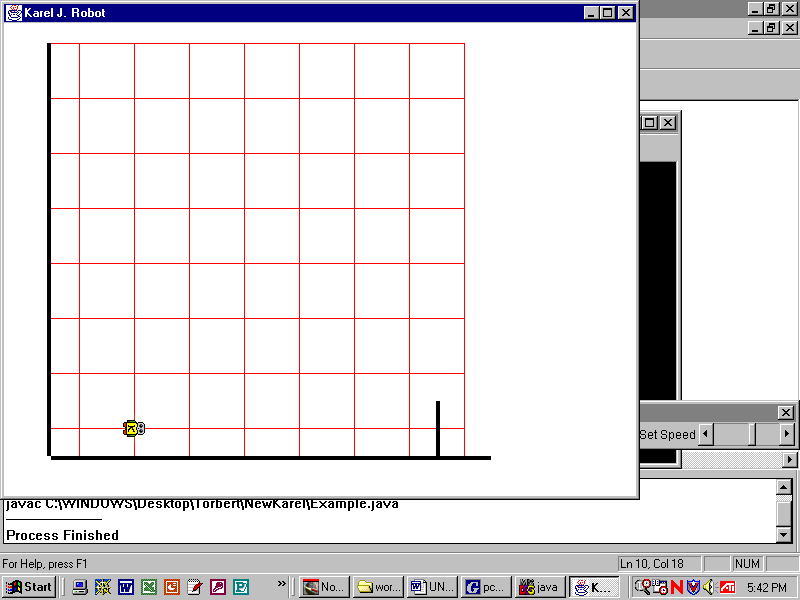
while-loops and if-statements

|  |  |
| --- | --- |
| Row | Task |
| 1 | Go to the end of the row of beepers. |
| 2 | Go to the beeper. |
| 3 | Go to the wall. |
| 4 | Go to the wall, picking up all the beepers (max one beeper per pile). |
| 5 | Go to the wall, picking up all the beepers. |
| 6 | Go to the end of the row of beepers; there is one gap somewhere. |

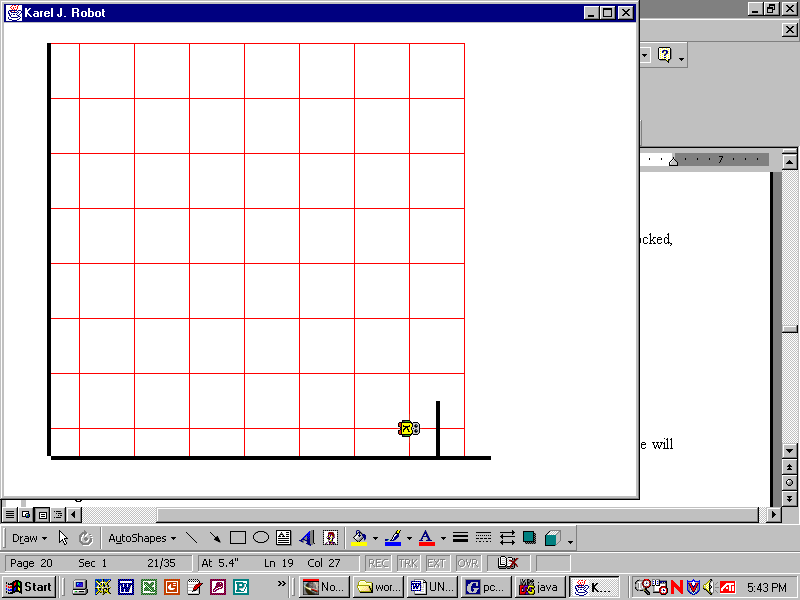
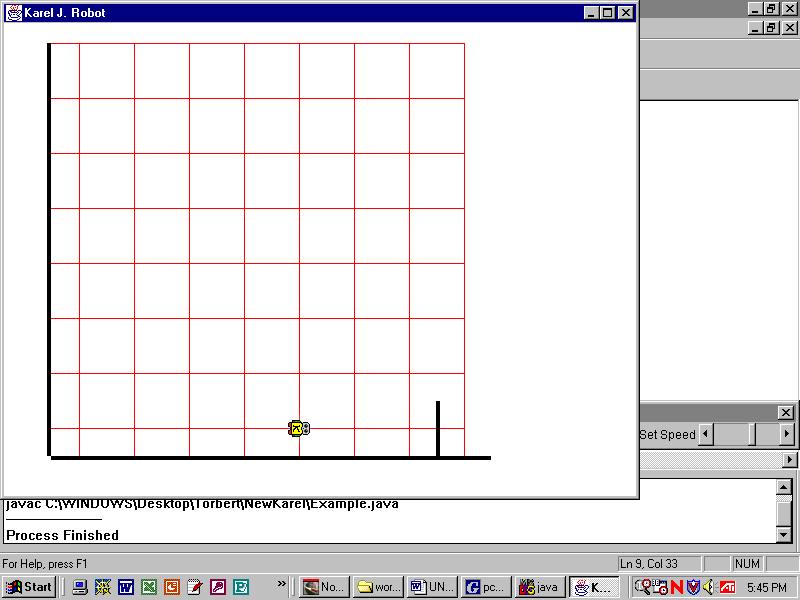
Background

This lab instantiates 6 Robots to perform 6 different tasks. For convenience, we code each task as a method. Each task uses one or more indefinite loops. Some tasks also use if-statements.

If temp (for temporary) is a Robot object, the solution to task number three is:   
 Before After



**while** temp.frontIsClear():

temp.move();

This loop will repeat an unknown number of times, as long as temp’s path is not blocked. The pictures show two such situations. Warning: if there is not a wall somewhere along the path then this segment of code will repeat forever; this is called an *infinite loop* and your program will never end.

This is the first lab in which your program must work correctly for many different robot-worlds. Thus, the name of the world will not be *hard-coded* as before: wld = World("tasks1.wld");

Instead the code will *prompt the user* to enter the name of the world.

name\_of\_world = raw\_input("Which 'tasks' world? ")

global wld

wld = World(name\_of\_world, delay=0.1)

Notice we defined a global variable wld, which makes it available outside the main method. This is the first lab in which we defined a main method and a strange if \_\_name\_\_=="\_\_main\_\_": statement. A main method and that if statement helps keep our code organized. Since this structure is standard in Python programs, we will use it from now on.

Specification

This is your first program that has shell code provided for you. Open Lab06.py. Study the import statement, the main, and the last if statement. Notice the 6 methods, each instantiating a Robot object. Accomplish all six tasks. Run your program three times, and when *prompted*, enter either “tasks1”, “tasks2”, or “tasks3”.

***Extensions***1. Modify task\_04 and task\_05 so that the robots count and print the number of beepers that they picked up. Look at the code on the next page for a hint.

2. Create a robot-world for this lab using worlds\MapBuilder.jar.

Lab06 Continued

|  |
| --- |
| count = 0  **while** karel.nextToABeeper():  karel.pickBeeper()  count+=1  print "The count is ", count |

A Dozen Tasks

A robot can count the beepers it picks up by using an integer variable. The variable starts at zero and then adds 1 each time. This code counts and prints the number of beepers in a pile. This print statement displays an informative label as well as the value in the variable count. Note the comma.

Complete the methods below.

**def task\_07(): //**go to the beeper or the wall. Count and report the number of steps you took.

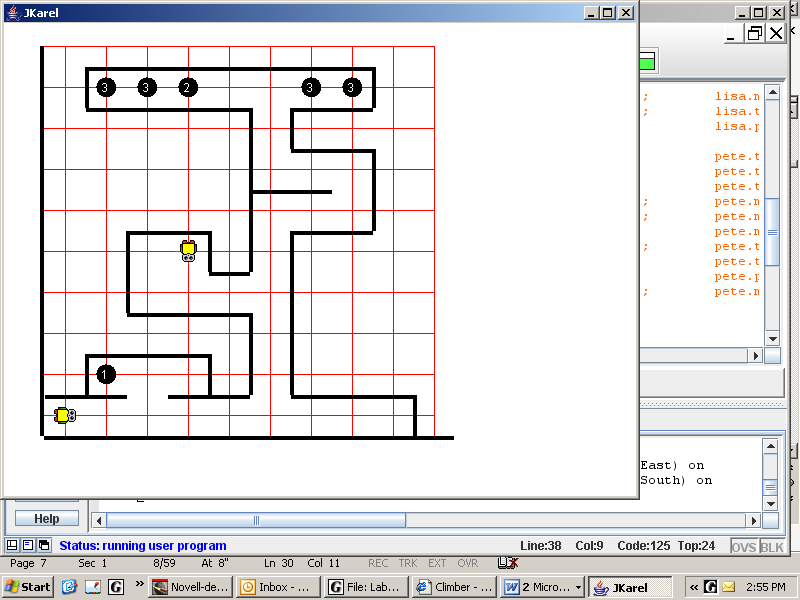
**def task\_08():** //go until you are next to another robot, then put all your beepers down.

**def task\_09():** //put down 5 different piles with 4 beepers in each pile. Use definite loops.

**def task\_10(): //**fill in gaps with a beeper. Stop when you reach a wall.

**def task\_11():** //while there is a wall to your right, put down one beeper at each step

**def task\_12():** //go until there is a wall to your right and you are standing on a beeper



Use worlds\MapBuilder.jar to make your own worlds for tasks 7 through 12 above.

Exercises

Lab06

1. Given the declarations shown, circle the commands which will run successfully.

wld=World(“Shakespeare”)

mercutio = Robot(wld)

juliet = Athlete(wld)

tybalt = Climber(wld, 4)

romeo = Racer(wld, 1, 1, east, 10)

|  |  |
| --- | --- |
| mercutio.climbUpRight() | romeo.pick() |
| juliet.facingEast() | romeo.sprint(-10) |
| juliet.move() | romeo.put(5) |
| tybalt.climbUpRight() | romeo.climbUpRight() |
| tybalt.jumpRight() | romeo.move(10) |
| takeTheField(mercutio) | race(romeo) |
| theFriar = Robot(1, 3, east, 0) | theNurse = Racer(wld) |

2. What is wrong with the call to the Racer constructor as it is defined in Lab05:

lord\_montague = Racer(wld, 1, 1, south, 10)

3. Write a constructor for a Capulet class that extends Robot. Capulets can start anywhere, but all Capulets face south and carry 100 beepers.

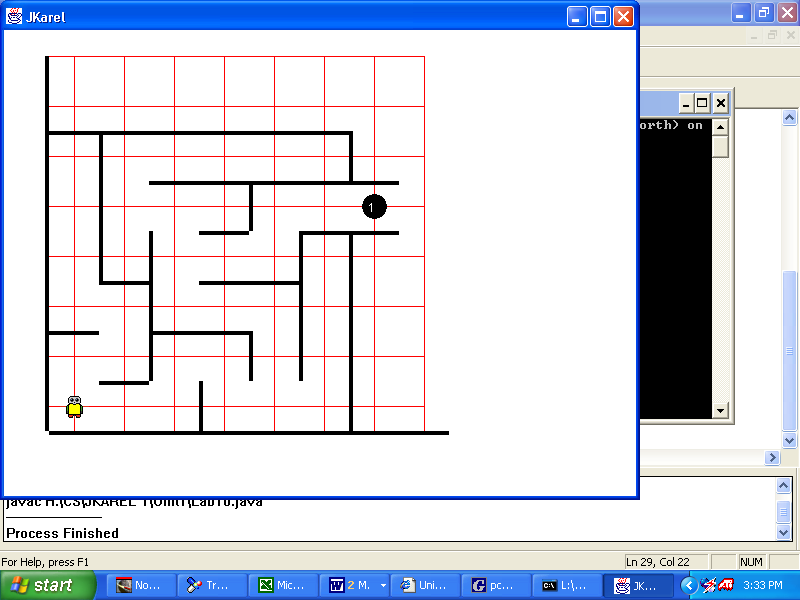
4. Write the code for: if you are on top of a beeper, pick them all up, and print how many there are.

5. Write the code for: if the Robot karel is not carrying beepers, and it is on top of a beeper, then pick it up.

Discussion

Algorithms

An *algorithm* is a sequence of steps, usually repeated, that eventually accomplishes the task. You have followed many algorithms in every-day life: baking a cake, installing an internal hard drive, doing long division, or knitting a scarf. Any list of how-to instructions is an algorithm. Do step 1, do step 2, and so on, without understanding, but in the end the task gets done. As you know, mechanically following a checklist without using your intelligence can become boring. On the other hand, designing and writing an interesting and intellectually challenging algorithm is a central part of what we call programming. (At least, we hope you will think so.) Indeed, many professional programmers write algorithms, not code. The algorithm-writers let someone else turn the algorithm into working code.

Human intelligence is different from computer intelligence. While humans can often glance at a situation and see an answer, computers must step-by-step compute an answer. For example, is there an exit to this 6x6 maze? You can find the path with your eyes very quickly and arrive at an answer. In contrast, a computer must rather blindly progress through a sequence of local decisions: is the robot at the end yet? If not, can the robot move forward? If not, should the robot turn, and which way? Do something, then repeat the whole sequence.

Now consider finding the exit to a 1000x1000 maze. A human would need a pencil, would make mistakes, would get tired, and no doubt would quit in frustration. A computer, of course, never suffers from those faults.

In the next few labs, we will concentrate on algorithms in our robot-worlds. We will give you a world and a task. The task will be complex, so that your job will be to break the large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed directly by the robots. The subtask will be a *command*, a *selection,* or a *repetition*, or, more likely, combinations of those. Selection statements (if or if-else) check for certain conditions and execute the appropriate sequence of commands. Repetition loops (while or for) perform some action repeatedly, usually with some variation. Pretty much every program you have ever used is made up of commands, selection, and repetition statements.

Programmers need to have strong visual imaginations. Students who can “see” the situation, both the whole and the detail, have a much better chance of solving the programming task. Pay close attention to the input and the output, because your task is to change the input into the output. The clearer your idea of what you have to do, the better. If it helps, draw pictures. If it helps, get up and act it out. The robot will be in a situation and need to do something: What does it need to decide? What does it do next? When do the repeated actions stop being repeated?

Discussion

Decision Structures

The simplest decision structure, which you have already seen, is the if-statement.

**if** karel.nextToABeeper():

karel.pickBeeper()

**if** total == 0: # tests for equality, i.e. "is equal to"

karel.turnRight()

karel.move()

(Given this code fragment, we may assume that karel is an Athlete and total has previously been declared as an integer.) The if-statement is useful when we have one branch that may or may not happen. When we have two branches, only one of which will happen, we use the **if-else** statement.

**if** karel.nextToABeeper():

karel.pickBeeper()

**else:**

karel.putBeeper() # only happens if !karel.nextToABeeper()

Here is an if-else example that where you must pay attention to indenting.

**if** total != 0: # tests for "is not equal to"

karel.turnRight()

karel.move()

**else:**

karel.turnLeft() # only happens if total == 0

karel.move()

In an if-else statement, exactly one of the choices will occur. It is impossible that neither will occur. It is impossible that both will occur. When three-way branching is needed, we use an if-else-if ladder. Note the colons, the indentation, and the new command elif:, which is Pythonese for else if . . . .

**if** total < 0:

karel.turnRight()

**elif** total > 0:

karel.turnLeft()

**else:**

karel.turnAround()

karel.move() # move goes here because it will

# definitely move no matter what

Of course, this three-way branching can be generalized to *n*-way branching by using as many if-else-if statements as are required. You will see this in future labs.

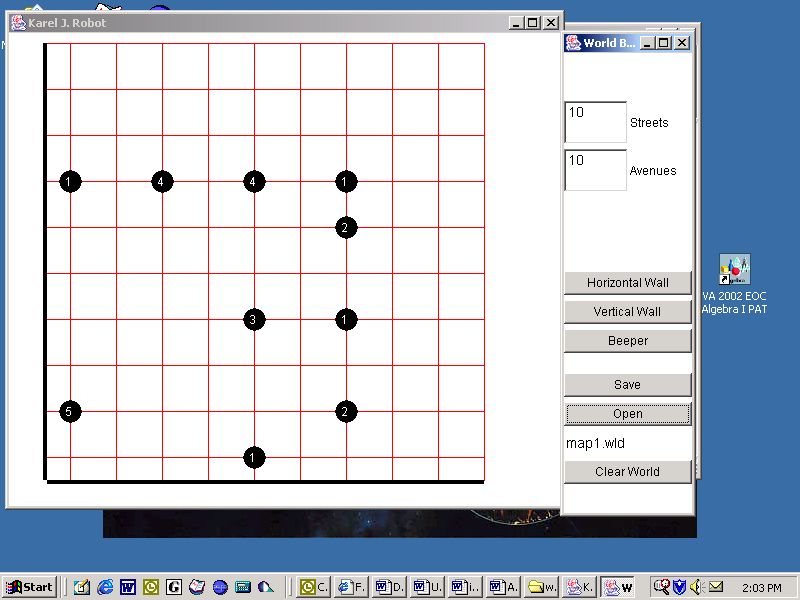
Lab07

Treasure Hunt

Objective

Implement an algorithm.

Background

There are treasures hidden in various robot-maps. Piles of beepers will guide your robot to the treasures. The treasure your robot is searching for is marked by a pile of exactly five beepers. To read the maps, your robot must walk forward until it encounters a pile of beepers. If that pile is the treasure, then you’re done. If not, have your robot turn a certain way based on the number of beepers in the pile. Then continue searching for the treasure.

|  |
| --- |
| Algorithm for Following the Treasure Map  1. Continue forward until you encounter a pile of beepers.  2. If you are at a pile with exactly five beepers, you’ve found the treasure.  3. Otherwise, if there is exactly one beeper, then turn left.  4. Otherwise, if there are exactly two beepers, then turn around.  5. Otherwise, if there are exactly three beepers, then turn right.  5. Otherwise, maintain your current heading.  6. Repeat as needed. |

Let’s start designing a solution. It looks like we don’t need to climb (Climber) or race (Racer). Robots can’t turn right. Probably an Athlete is a good place to start. What does an Athlete know how to do? Can we accomplish the task with those methods, or would it help to define new methods? If so, what new methods would help? Describe what each method should do. Think in terms of while, if, or for.

approachPile

numOfBeepersInPile

turnAppropriately

Specification

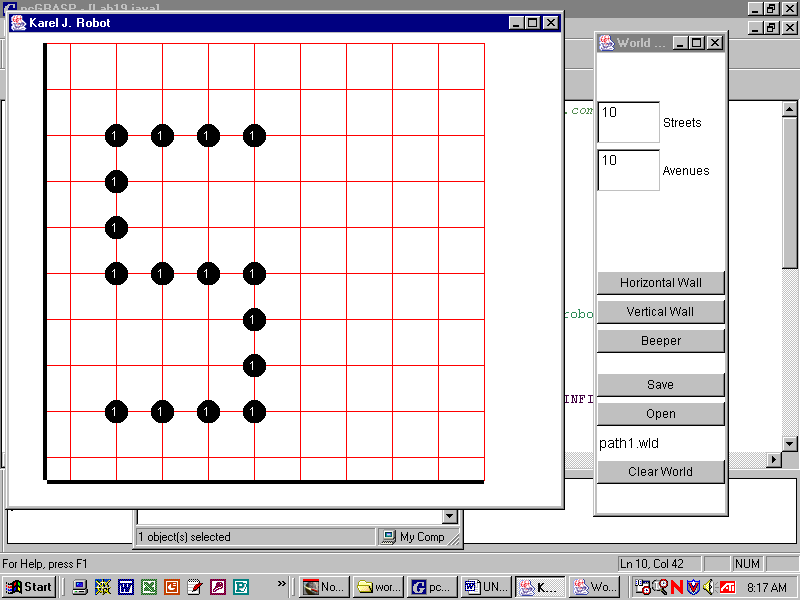
Open Pirate.py. Pirate extends Athlete. Pirates always start at 1, 1, east, 0. Implement the three methods in Pirate.

Open Lab07.py. Run it. When prompted, enter “map1”, “map2”, or “map3”.

Test Data

Look in the Python shell window when your program ends to make sure you followed the map correctly. There are 24 total beepers in map one, 32 total beepers in map two, and 33 total beepers in map three.

Lab08

Yellow Brick Road

Objective

Design and implement an algorithm.

Background

Follow, follow, follow, follow, follow the yellow brick road. The task is to get your robot from (2, 2) to the end of the path of beepers, wherever that may be. The robot must end standing on the last beeper.

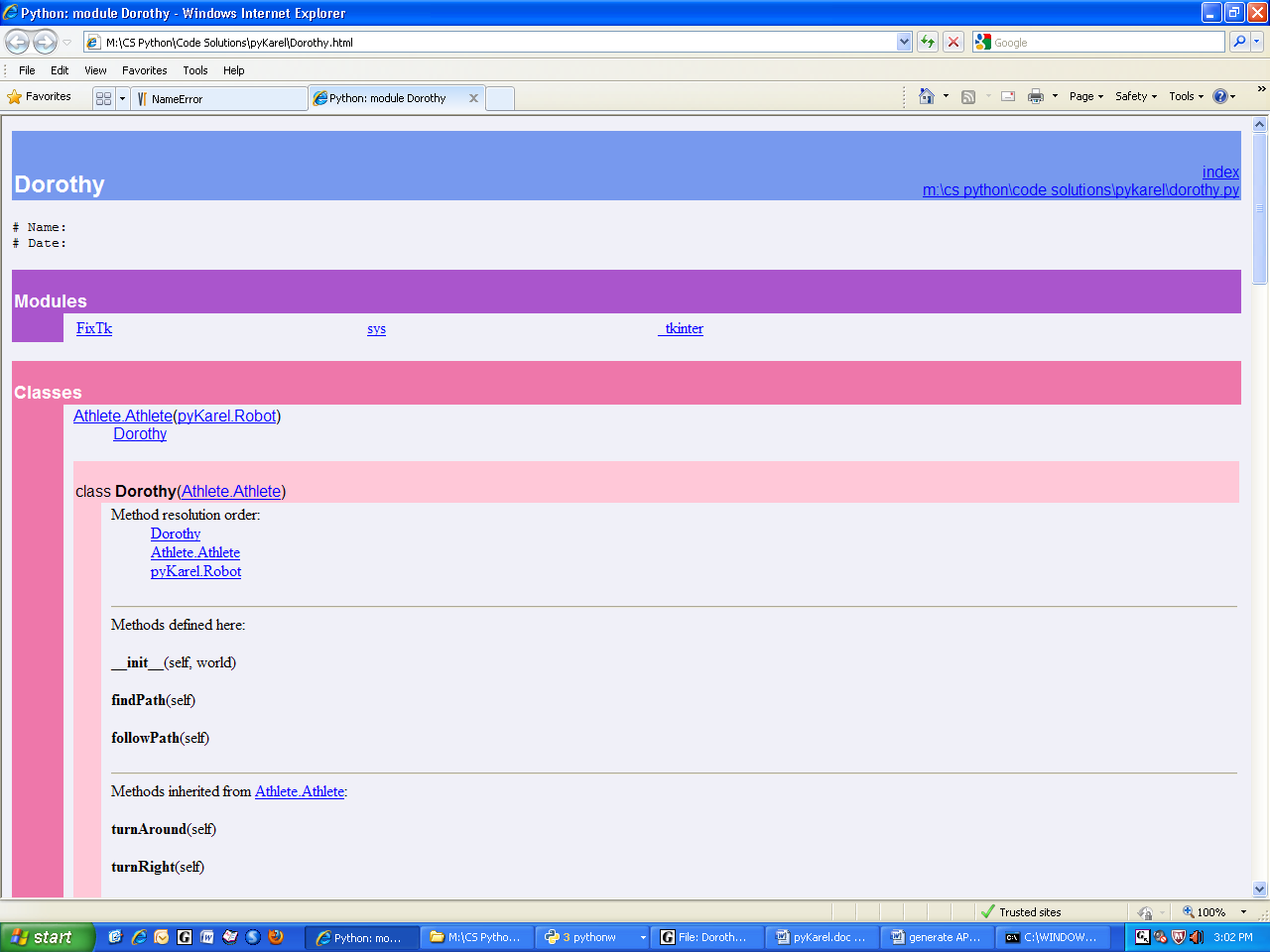
In words, write a good algorithm:

while

Some people prefer to think of it this way:

if

Look at the API. The teacher accomplished the task by writing findPath and followPath.

findPath returns a Boolean. Write the code for

def findPath(self):

Write the code for

def followPath(self):

Specification

Create Dorothy.py. Dorothys always begin at 2, 2, east, 0. Look at the API to see the arguments needed by the constructor.

Open Lab08.py. Instantiate a Dorothy object and send her on her way. When prompted, enter “path1”, “path2”, or “path3”. The "path3" world has an extra complication for you to deal with!

Extension

Use worlds\MapBuilder.jar to create other robot-worlds.**Lab09**

Shifting Piles

Objective

Design and implement an algorithm.

Background

|  |
| --- |
| pile1 before |
| pile1 after |
| pile2 before |
| pile2 after |
| pile3 before |
| pile3 after |

The task is to move piles of beepers one space to the east (right).

Since that task is a big vague, we first specify that task more precisely. The piles of beepers are all on row=1. There are no more than 7 piles. No pile has an infinite number of beepers. The robot begins at 1, 1, east, 0. The task is to shift each pile one space to the right.

(The piles on row=2 do not need to be moved! They only serve as a reference, so that when your program runs, you can easily check whether the piles have been properly moved over.)

There are two common ways to solve this task. Complete the algorithms below.

1) Going east, pick up each pile . . .

2) Move the robot 8 steps to the east, turn around, and . . .

Do you need to make a new class with new methods, or will an Athlete be good enough?

Specification

Create Lab09.py. Import from pyKarel and from Athlete. The world is width=8, height=3. Instantiate one Athlete at 1, 1, east, 0. Shift each of the piles on the first row one block to the right. Test the program with worlds “pile1”, “pile2”, and “pile3”.

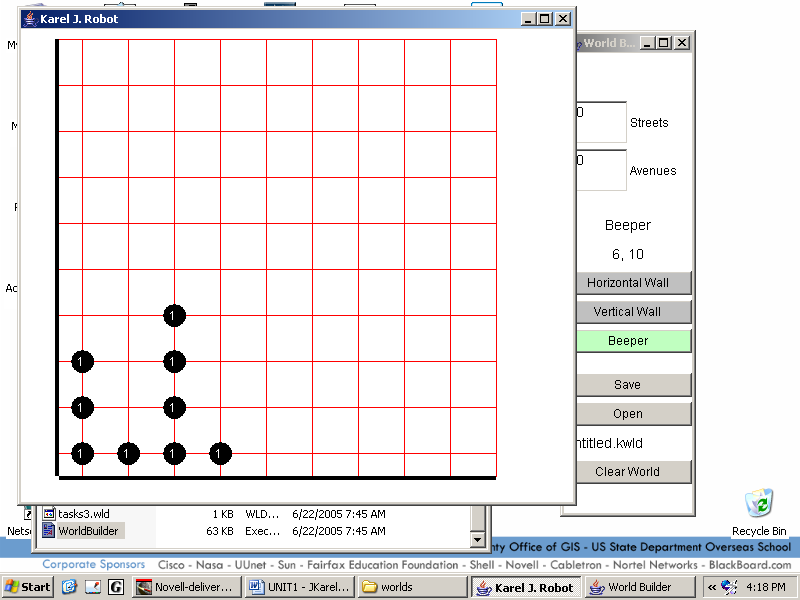
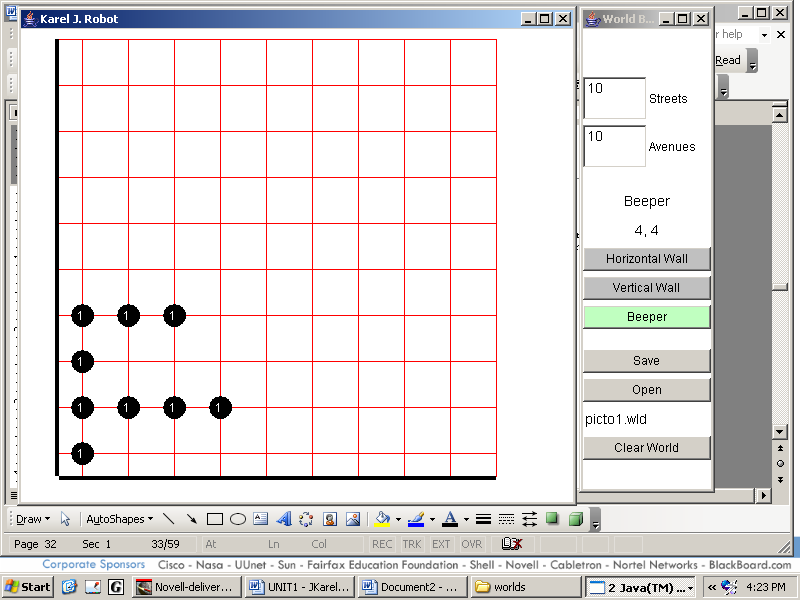
Extension

Open Lab09Extension.py. Create Shifter.py, which defines a Shifter class and its method shift\_piles. Move all the piles that way.

Exercises

Lab09

In this exercise you also have to pick up beepers, counting as you pick, then carry the beepers to another place and put them down. Change the vertical pictograph ("before") into the horizontal pictograph ("after"). There is only one beeper at each corner. There are no more than 4 columns of beepers. You do not know how tall each column is, but there are no gaps within a column.



Before After

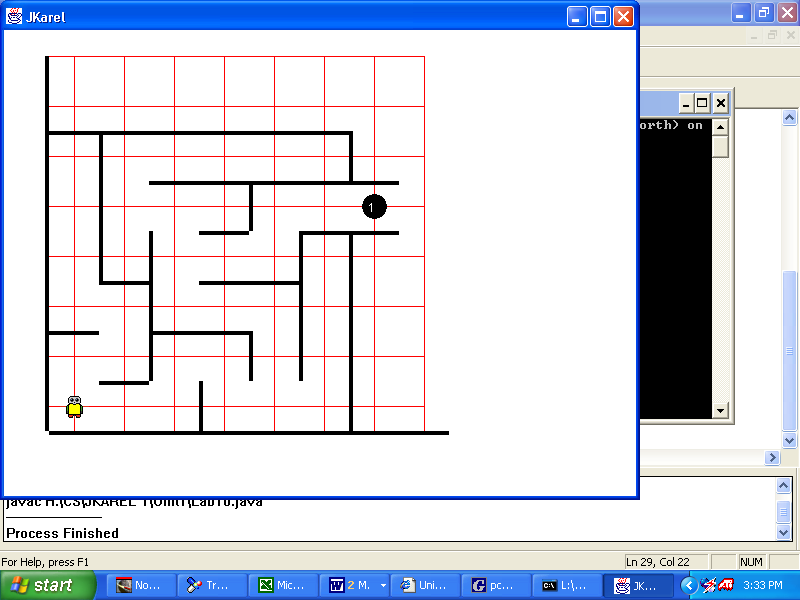
1. Plan the tasks, in English, not in code, which your Athlete must do.

2. Would it help to make a subclass of Athlete? Y/N. If so, what would you name it and what would it do?

1. Assume that the robot is in position, facing east. Assume that the robot has already picked up and stored the number of beepers in that row in the variable *rowCount2*. Now write the code to put down *rowCount2* number of beepers in a row.

4. Your teacher may wish to turn this exercise into a lab. If so, you may use worlds "picto1", "picto2", and "picto3".

Lab10

Maze Escaping

Objective

Design and implement an algorithm.

Background

The task is to escape a maze.

We first specify the task more precisely. The robot begins in the lower left-hand corner. Somewhere there is an exit, which is marked by 1 beeper. There are several different mazes. Each maze has a continuous sequence of walls connecting the start of the maze to the end of the maze.

The last specification has a very important consequence. It means that the robot will reach the exit by always keeping a wall on the right (or on the left).

A good problem-solving strategy is to imagine yourself in a corn maze. What do you do to exit the maze?

Let’s start designing the algorithm. Ask yourself the following questions:

1. What are the initial attributes of the Athlete? (What values do you pass to the constructor?)

2. What does an Athlete know about walls and beepers?

3. How will the Athlete know when to stop? while . . .

4. What different configurations of walls will the Athlete encounter? Draw some pictures:

1. At each step, the Athlete needs to make one decision and either move, turn left, or turn right. At each step, the Athlete needs to move so as to keep a wall on its right. if . . .

Specification

Create Lab10.py. Import Athlete. Prompt the user for a "maze" world, either “maze1”, “maze2”, or “maze3”. The world is width=10, height=10. Instantiate one Athlete with the default attributes. Escape the maze.

Extensions

1. Use worlds\MapBuilder.jarto create robot-worlds to test your program.

2. Mark the path from start to end. The final path should not mark detours that were tried and then later abandoned. Hint: The last sentence was another hint.

Lab11

Climbing Up and Down

Objective

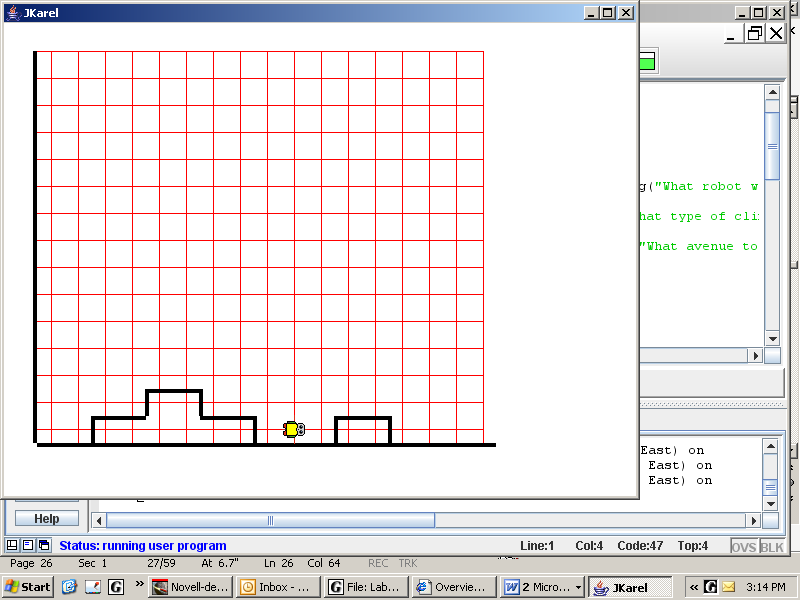
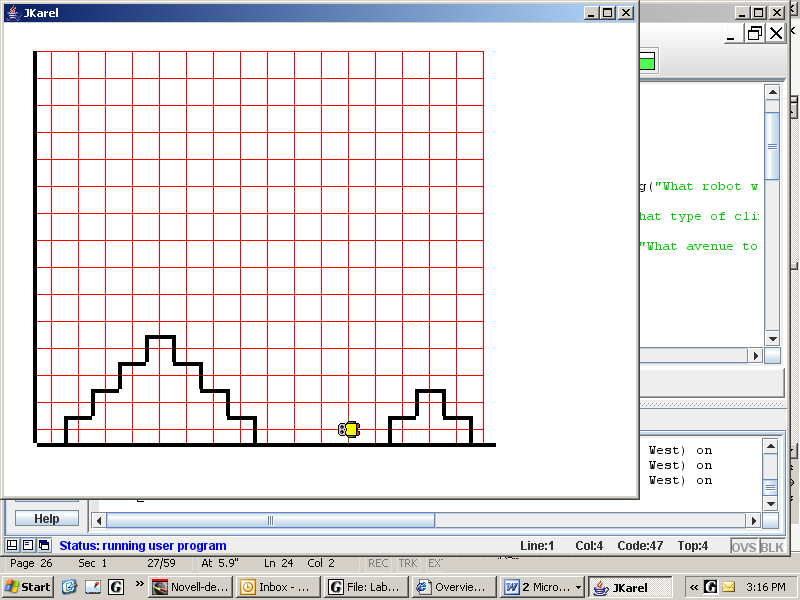
Polymorphic behavior.

Background

In object oriented programming, if we set things up correctly, we can call the “same” methods, but by passing different subclass objects, we can get (slightly) different behaviors. This is a very powerful technique. As a real-world example, suppose the student council tells the students to wear their class colors during Homecoming. Every subclass of student, freshman, sophomore, etc., acts on that command, but in its own way, because freshmen know to wear red and sophomores white, and so on. That is polymorphic behavior.

To get polymorphic behavior in Python, we call methods on a variable, using dot-notation, but the variable can point to different subclass objects. Since the subclass objects execute the methods differently, we get different, or *polymorphic*, behavior. Any time we have a variable that can hold different subclass objects, we have the possibility of polymorphic behavior.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | Climber | |  |  |
|  |  |  |  |  |  |
| HillClimber | |  |  | StepClimber | |

In this lab we are going to define two subclasses of Climber, one to climb hills and one to climb steps. Four instance methods (climbUpRight, etc.) in each subclass will *override* the four similar instance methods in Climber. We will be calling the "same" methods, but passing different objects, either a Climber, HillClimber, or StepClimber. Whatever object we pass,   
the arg will execute climbUpRight as it is defined in that class and we will see polymorphic behavior.

As a shorter example, this code produces polymorphic behavior:

**def** ascend(arg):

arg.climbUpRight()

How does it work? What will happen when we call ascend and pass each of these objects: a Climber, a HillClimber, a StepClimber, an Athlete, a Robot? Why is the result called "polymorphic behavior?"

Specification

Open Mountain.py and study its method explore. Complete the Exercises on the next page.

Open Climbers.py, which you last used in Lab03. Inside Climbers, define the HillClimber and StepClimber classes. The methods in HillClimber and StepClimber override the methods in Climber. Implement all the methods. Recall that the arguments of the \_\_init\_\_ must match the arguments accepted by the Climber constructor.

Open Lab11.java. Do not change this code! Run it. The application will prompt the user to specify the world, the kind of climber, and the x-coordinate starting position. When prompted, type in this information:

|  |  |  |
| --- | --- | --- |
| mountain1, mountain2, mountain3 | Climber | 8 |
| hill1, hill2, hill3 | HillClimber | 10 |
| step1, step2, step3 | StepClimber | 12 |
| mountain1 | Athlete | 8 |
| mountain1 | Robot | 8 |

Exercises

Lab11

Looking at Mountain.py, answer the following questions.

1. Does Mountain define classes or methods? \_\_\_\_\_\_\_\_\_ How do you know?
2. Explain the difference in syntax between explore\_west(arg) and arg.move().
3. How does explore know not to search the eastern mountain if the treasure was on the western one?
4. Explain the purpose of the first arg.putBeeper and last arg.pickBeeper commands of explore.

1. In explore\_west, what is the variable n actually counting?
2. How does the climber know that it has reached the summit?
3. If the treasure isn’t on the east summit, how does the climber know not to try to pick it up?
4. How can the code use a for-loop going down the mountain but needed a while-loop going up?
5. What command causes the climber to stop at base camp?
6. Which specific method call(s) result in different behaviors, depending on the object?
7. Does the steepness of the mountain affect the code in Lab11.py? Why or why not?
8. How can different types of objects make use of a single, common code?

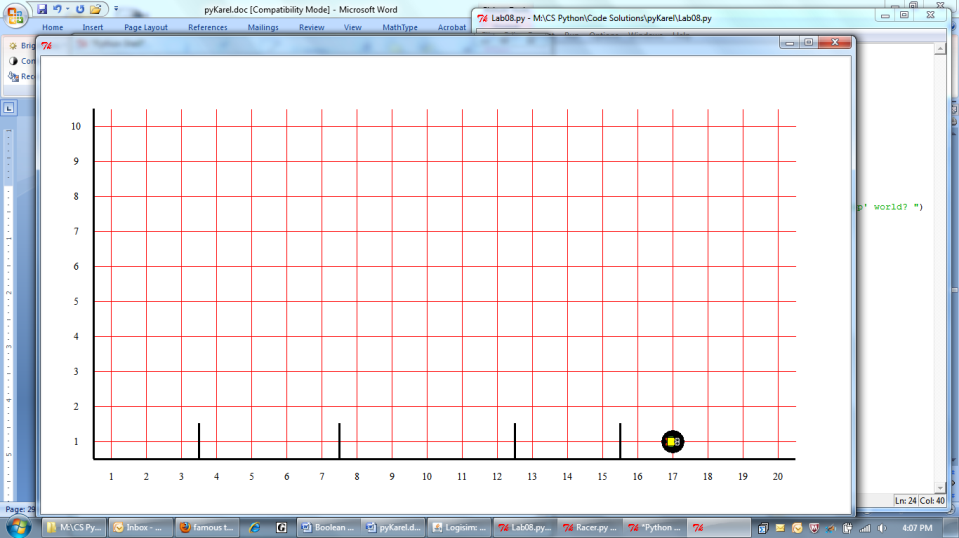
Lab12

Hurdle Racing

Objective

Polymorphic behavior.

Background

The hurdles shown are all one-block tall hurdles. A beeper marks the finish line. The algorithm is: if there’s no hurdle, then move, otherwise jump over the hurdle. Keep going until you hit a beeper.

In Python we can write the method shown in the box.

**def** race(arg):

**while not** arg.nextToABeeper():

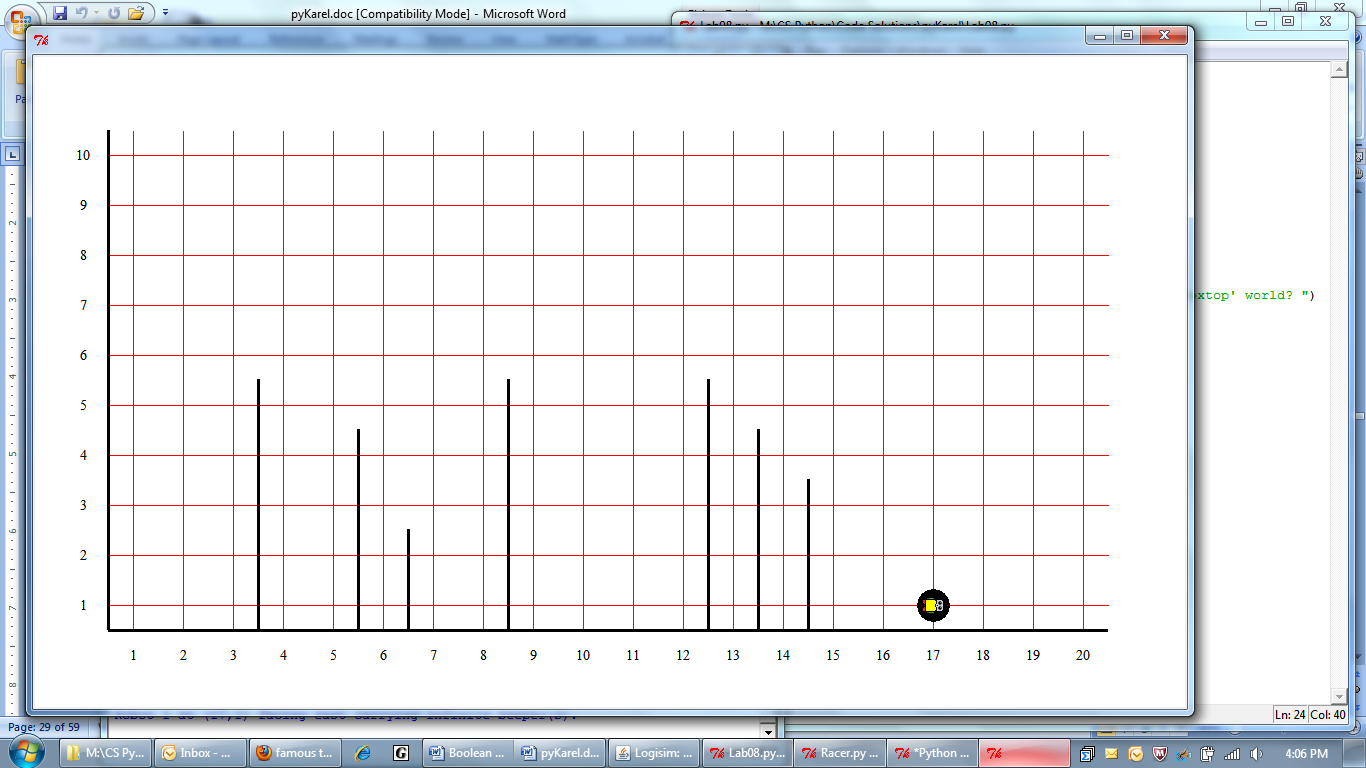
**if** arg.frontIsClear():

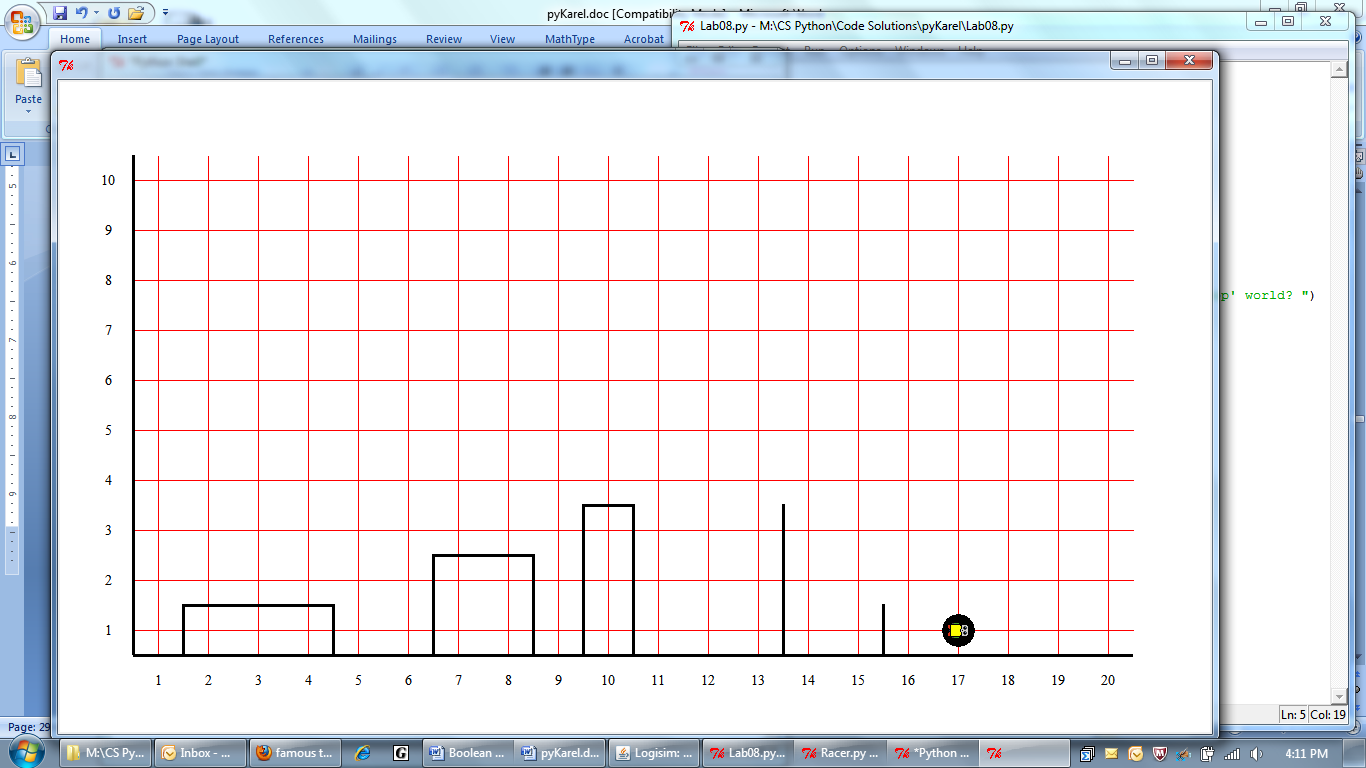
arg.move()

**else**:

arg.jumpEast()

This code implies that we will need to define a new resource class (let’s call it Racer) that inherits from Robot. A Racer will need to know how to jumpEast. That means you, the programmer, will have to define jumpEast in Racer.

Now let's change the race-course so that instead of one-block tall hurdles we must jump over hurdles of any height. We do not change the algorithm above! The method arg.jumpEast could be made to work to jump hurdles of any height. In object-oriented programming we look around for a class that does approximately what we want (i.e. Racer) and then extend it. When we extend a class, we give the new class the powers that we really want—namely, how to jump hurdles of any height.

What if we wanted to jump hurdles of any width and height? Again, our original algorithm still works! All we have to do is to write a new subclass that knows how to jump these new kinds of hurdles.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | Robot | |  |  |
|  |  |  | |  |  |
|  |  | Athlete | |  |  |
|  |  |  | |  |  |
|  |  | Racer | |  |  |
|  |  |  |  |  |  |
| SteepleChaseRacer | |  |  | BoxTopRacer | |

Each of the classes SteepleChaseRacer and BoxTopRacer will extend Athlete and override jumpEast. The new classes can inherit the Racer constructor.

Specification

Open Racers.py, which you used in Lab05, and define the SteepleChaseRacer and BoxTopRacer classes. Define one new method in each class, named jumpEast.

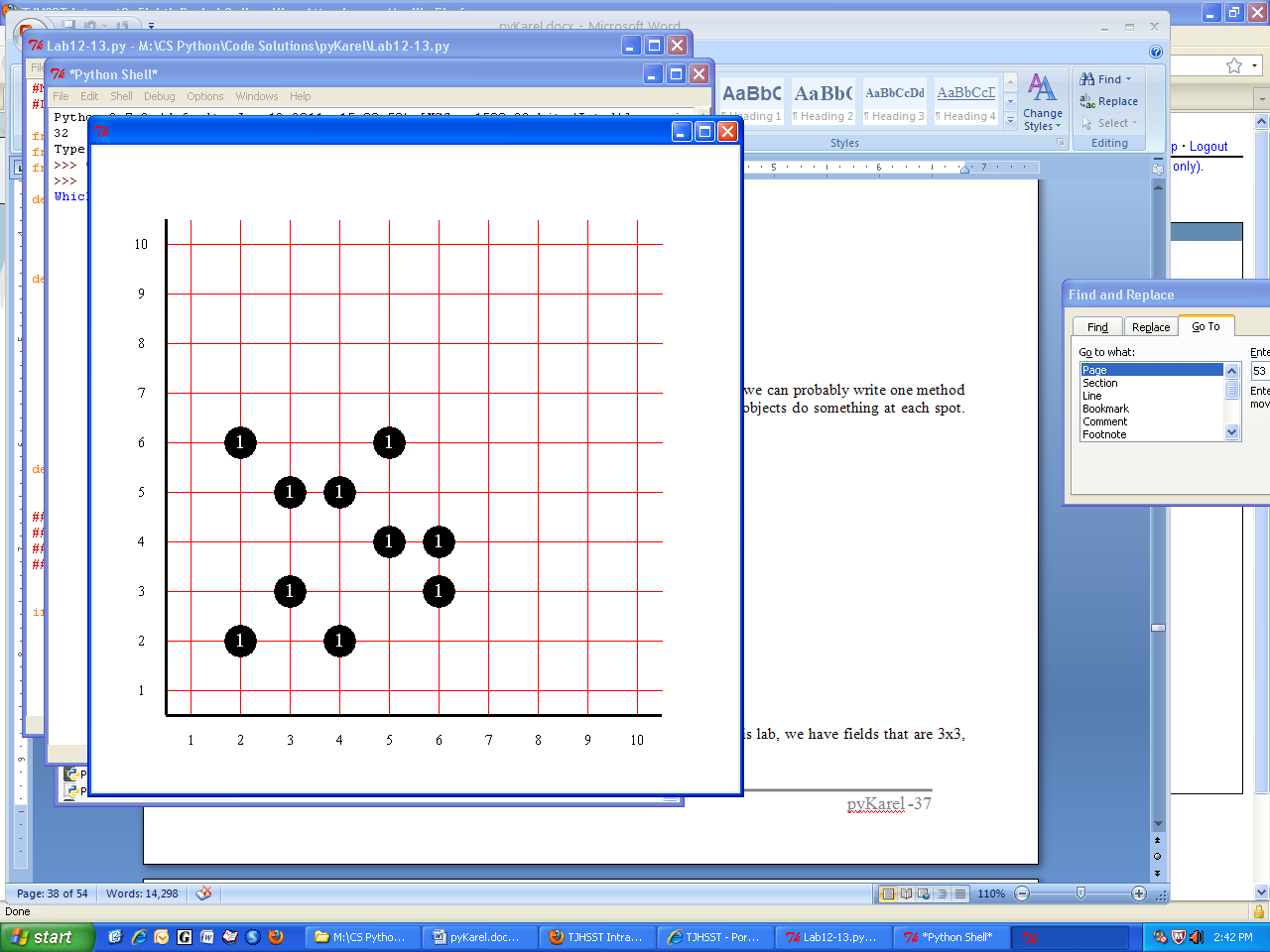
Open Lab12.py. Do not change this code! Run it. With worlds “hurdle1”, “hurdle2”, and “hurdle3”, use a Racer object. With worlds “steeple1”, “steeple2”, and “steeple3”, use a SteepleChaseRacer object. With worlds “boxtop1”, “boxtop2”, and “boxtop3”, use a BoxTopRacer object.

Extension

Explain how this lab is an example of polymorphic behavior.

Lab13

Harvesting and Planting



Objective

Design and implement an algorithm. Show polymorphic behavior.

Background

Let's consider a square field of beepers. You are going to write two similar-but-different classes. Each of these classes will execute the same command, but do it in different ways. Polymorphic behavior!

Consider a square field with an irregular scattering of beepers. Half the time, the task will be to cover the field with one beeper at each corner. The other half the time, the task will be to pick up all the beepers. How would you accomplish these two tasks?

In object-oriented programming, we naturally think in terms of objects. Since we have two tasks, we probably need to write two different classes.

From what class would the new classes inherit?

What would you call them?

What do the two classes do differently?

if . . .

if . . .

On other hand, since both tasks require that we visit each spot on the field, we can probably write one method that walks the field. We need to visit each spot on the field and tell the objects do something at each spot. Write a *pseudocode* solution. (Pseudocode is code that is half-English and half-Python, a series of method calls without all the strict syntax rules.)

def \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(arg):

How do you make the code work for different sizes of square fields? In this lab, we have fields that are 3x3, 5x5, and 8x8. The field shown above is the 5x5 field, with beepers covering columns and rows 2 through 6.

Lab13 Continued

Harvesting and Planting

**from** Workers **import** \*

**from** random **import** randint

**def** work\_one\_row(arg, m):

**for** k **in range**(m):

arg.doYourThing()

arg.move()

**def** work\_square(arg, n):

arg.turnToTheNorth()

work\_one\_row(arg, n)

**for** k **in range**(n, 1, -1):

arg.turnRight()

work\_one\_row(arg, k)

arg.turnRight()

work\_one\_row(arg, k)

arg.doYourThing()

**def** main():

size = raw\_input("Which 'field' (3, 5, 8)?")

wld=World("field"+size, delay=0.1)

**if** randint(0,1) == 0:

work\_square(Harvester(wld), int(size))

**else**:

work\_square(Planter(wld), int(size) )

wld.mainloop()

**if** \_\_name\_\_=="\_\_main\_\_":

main()

Examine this code. It will help you if you learn to scan code for important things, and ignore the things that are not. For now, ignore the import statements. Find the main and the names of the methods.

For some reason, the author decided to walk the field in a spiral. Explain how work\_square works.

Explain how this for-loop works:

for k in range(n, 1, -1):

Explain how work\_one\_row works.

Why do programmers make subtasks like work\_one\_row?

How does the application handle different sizes (3x3, 5x5, or 8x8) of fields? How do n, k, and m get their values?

From this code, what methods **must** be in the resource classes? Circle those method calls.

The main creates either a harvester or a planter with equal probability. The randint(0, 1) method returns an integer between 0 and 1, inclusive. So half the time the result will be 0 and half the time it will be 1. The main also shows how to turn a string into an integer. raw\_input returns a string, but we need to pass an integer into work\_square.

Specification

Create Workers.py. Import Athlete. Define the Harvester and Planter classes, which inherit from Athlete. Both workers start at 2, 2. Write all needed methods.

Create Lab13.py. Type in the code as shown. When you run it, you have a 50-50 chance of getting each kind of worker. Run it until you see that both kinds of workers work as they should.

Extension

The author chose to walk the field in a spiral pattern. Write your own work\_square to walk the field in a different pattern.

Lab14

Republicans and Democrats

Objective

Design and implement an algorithm. Use polymorphic behavior.

Background

Let's return to the maze. This time your boss has written the algorithm in *pseudocode*:

def escape

walk-down-current-segment

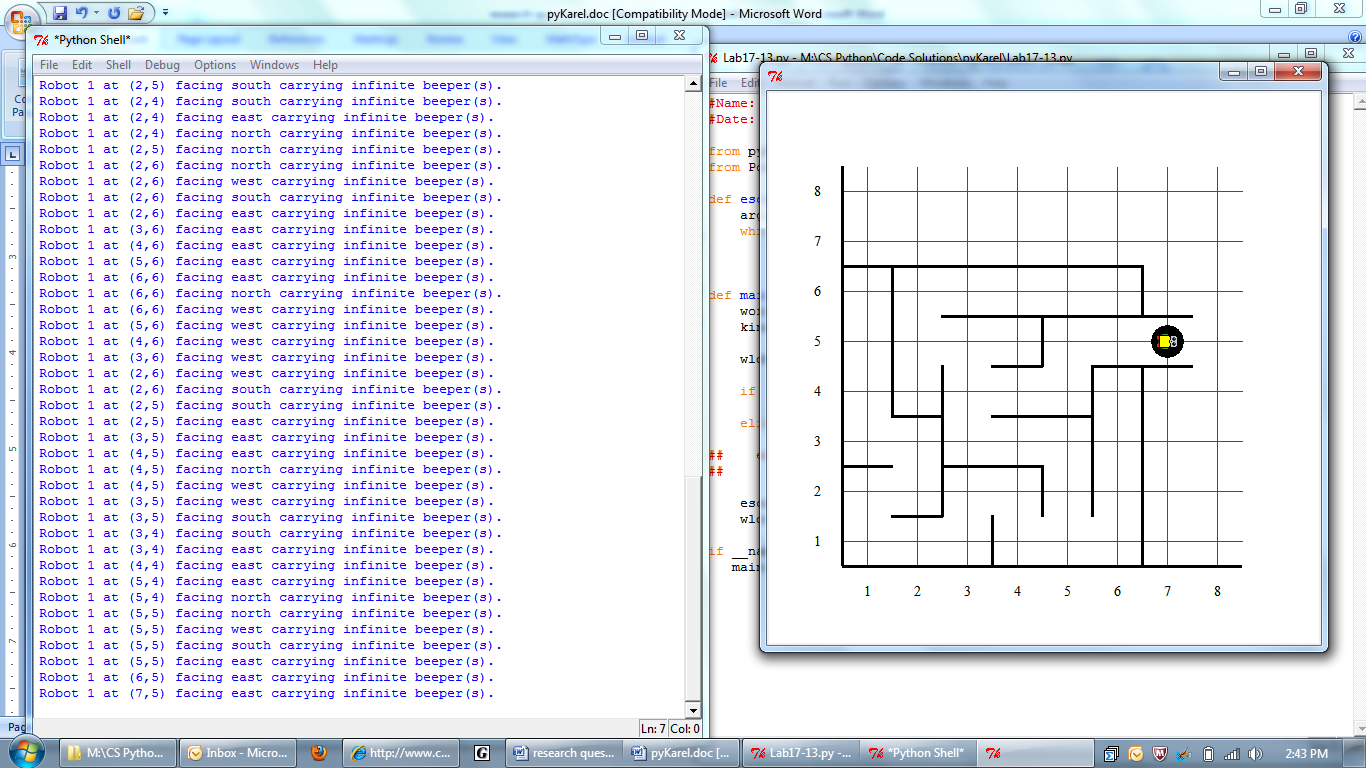
while not nextToABeeper

turn-to-the-next-segment

walk-down-current-segment

Your boss wants you to design and program two classes, a right-wall follower and a left-wall follower. Both of them use the same algorithm above.

First, make sure that the algorithm works. Get up and act out its directions. Does it work?

Let's consider the right-wall follower first. When is it okay to take one step? Write a while-loop in

**def** walkDownCurrentSegment(self):

|  |  |
| --- | --- |
| Turning the corner to the right. | |
|  |  |
| Turn left. | |
|  |  |

When the robot must turn, it has to choose the correct turn. If it can turn right, it . . . .

Otherwise, . . .

Write a three-way if-statement for

**def** turnToTheNextSegment(self):

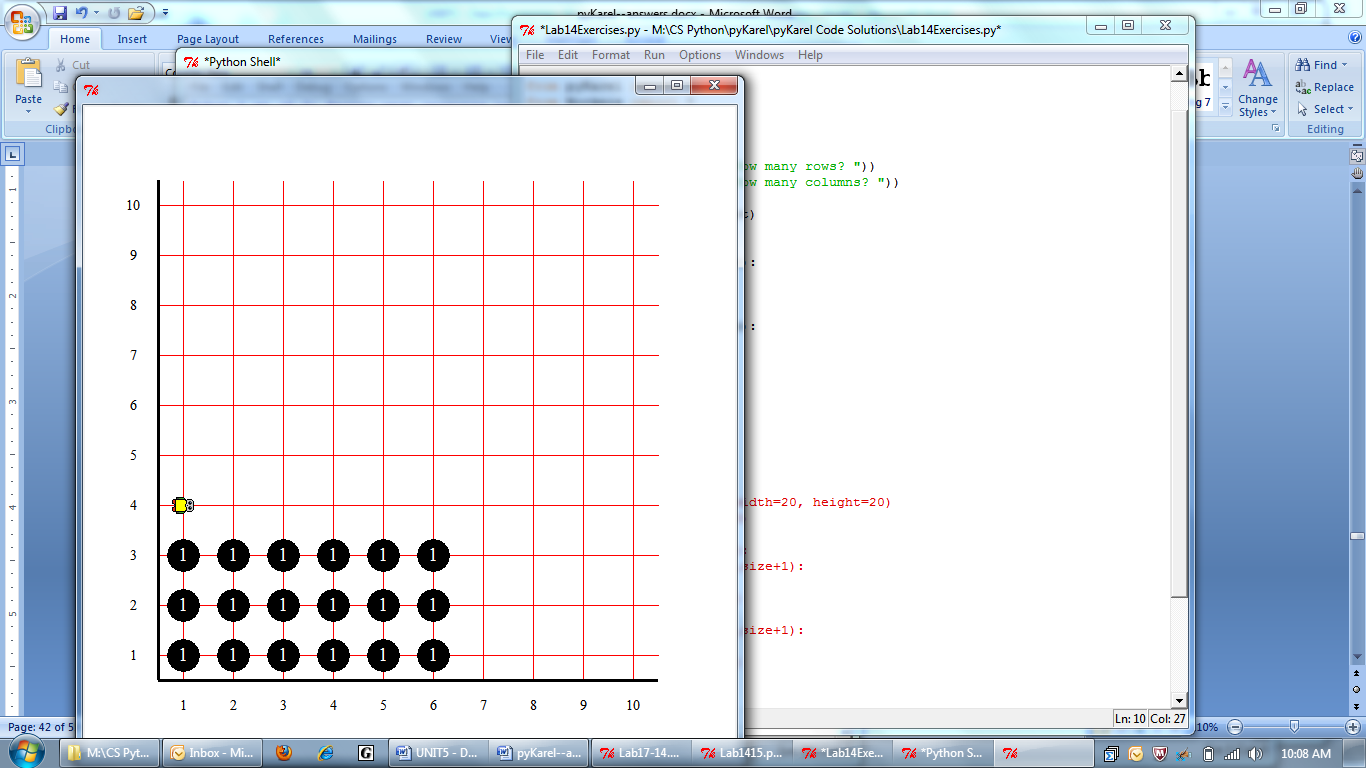
***Specification***

Create Politicians.py. Define two classes that extend Athlete, a Republican who follows walls right and a Democrat who follows walls left. You don't need to write constructors, for they are inherited from Athlete.

Open Lab14.py. Study the class method escape. When prompted, enter “maze1”, “maze2”, “maze3,” or "maze4” and the kind of politician.

Extension

Explain why this lab is another example of polymorphic behavior.

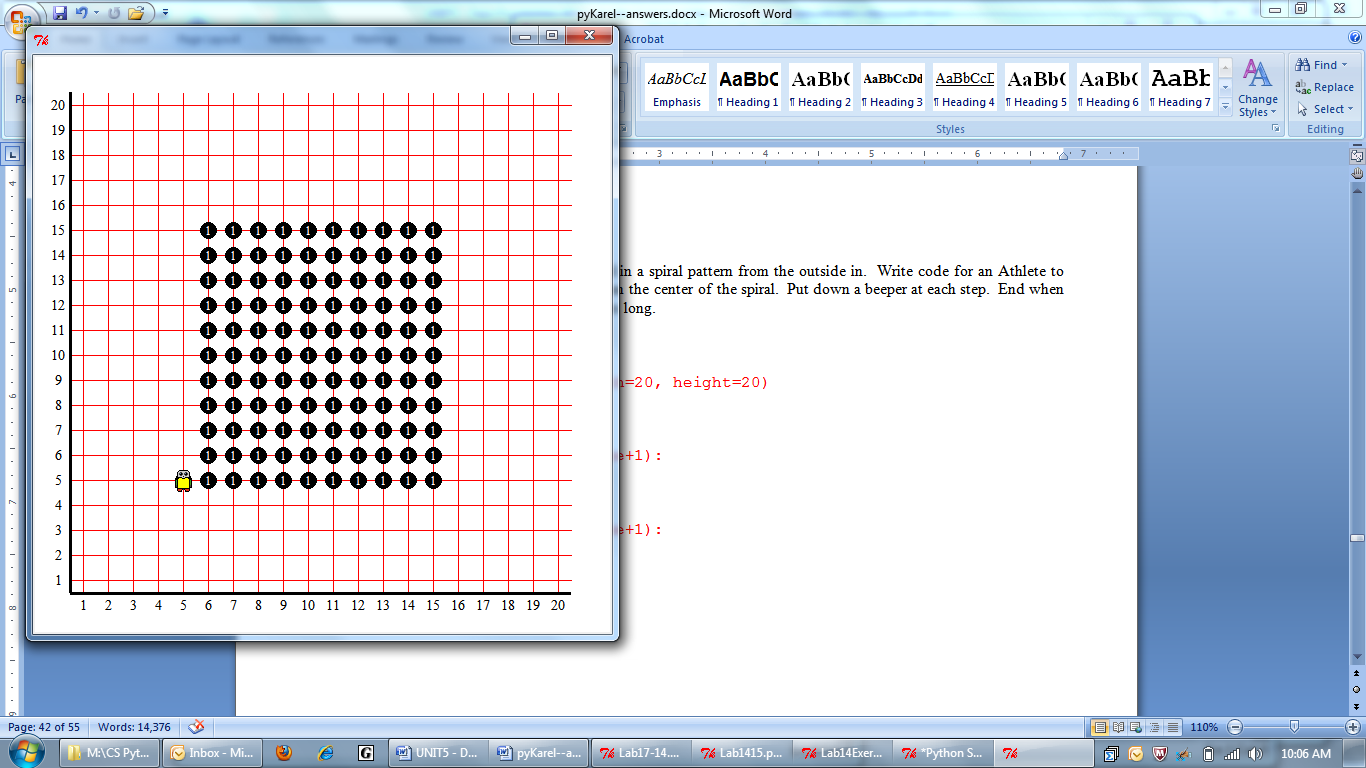
Exercises

Lab 14

1. Prompt the user to enter two numbers for rows and columns. Write code to make an Athlete drop beepers in rows and columns starting from 1, 1.

How many rows? 3

How many columns? 6



2. In Lab13, the robots walked the field in a spiral pattern from the outside in. Write code for an Athlete to walk a square spiral, this time starting from the center of the spiral. Put down a beeper at each step. End when at least one side of the square is 10 beepers long.

Lab15

Synchronized Swimming

Objective

Use threads to achieve parallel programming.

***Background***  
Is Lab15 a robot lab? Where are the robots and the pyKarel.py? All we see are World, Swimmer, and Thread objects. We have to look at the Swimmer class (or its API) to see that Swimmer extends Athlete. We have to look at the Athlete class (or its API) to know that it extends Robot and imports pyKarel.py.

**from** Swimmer **import** \*

**from** threading **import** Thread

**def** myfunc(arg):

arg.swim\_laps()

**def** main():

wld=World()

weismuller = Swimmer(wld, 2);

fraser = Swimmer(wld, 4);

spitz = Swimmer(wld, 6);

phelps = Swimmer(wld, 8);

t1 = Thread(target=myfunc, args=(weismuller, ))

t2 = Thread(target=myfunc, args=(fraser, ))

t3 = Thread(target=myfunc, args=(spitz, ))

t4 = Thread(target=myfunc, args=(phelps, ))

t1.start()

t2.start()

t3.start()

t4.start()

wld.mainloop()

**if** \_\_name\_\_=="\_\_main\_\_":

main()

Thread is a class defined in threading.py. The Thread constructor’s arguments specify a target method and an object. (The syntax is that of a “tuple,” which we have not studied yet.) Then the Thread object takes care of the details of running the object’s method. The start method starts it all off. The result will be parallel processing, i.e., different robots will perform their own methods seemingly at the same time.

As you should be able to tell by now, Swimmer has to define a method called swim\_laps.

***Specification***

Create Swimmer.py. Swimmer extends Athlete. Implement swim\_laps so that a swimmer moves forward eight steps, twirls around, then moves back to its starting position to prepare for the next iteration—like one lap in a pool. Make a swimmer do ten laps.

Create Lab15.py. Type in the code above. Run.

Extension

Return to Lab05.py, the shuttle run. Modify Lab05.py so that the racers run in parallel.

Lab16

Dancing Robots

Objective

Abstract classes, parallel programming.

Background

So far all the classes you have created have been *concrete* classes, meaning they were meant to be instantiated. Programmers sometimes create a hierarchy with *abstract* classes, meaning they are not meant to be instantiated. The reason to create an abstract class is for design purposes. The programmer wants to define the shared methods in the superclass, but leave other methods to be defined in the subclasses.

Unlike some other languages, Python has no special syntax for abstract classes. Therefore, Python programmers usually include a comment and list what methods need to be implemented later.

**from** Athlete **import** \*

**class** Dancer(Athlete):

##this class is abstract

##myDanceStep should be implemented in subclasses

**def** dance(self):

**for** k **in** **range**(10):

self.myDanceStep()

Specification

Dancers.py contains the abstract Dancer class, as shown.

Write at least three different dancer  
subclasses. For example, you

might make a square dancer,

a spinner, and a waltzer.

Create Lab16.py which makes

all your dancers dance

simultaneously. Use Lab15 as a model.

Fill in the UML diagram showing

the hierarchy you created:

Here is a syntax point. We did not have to give our dancers names. We could have started the threads with *anonymous* dancer objects, as follows:

t1 = Thread(target=myfunc, args=(SquareDancer(wld, 2), ))  
t1.start();

In fact, we didn’t even have to name the threads:

Thread(target=myfunc, args=(SquareDancer(wld, 2), )).start()

Some students really like the second version. Which do you think is the clearest and most readable?

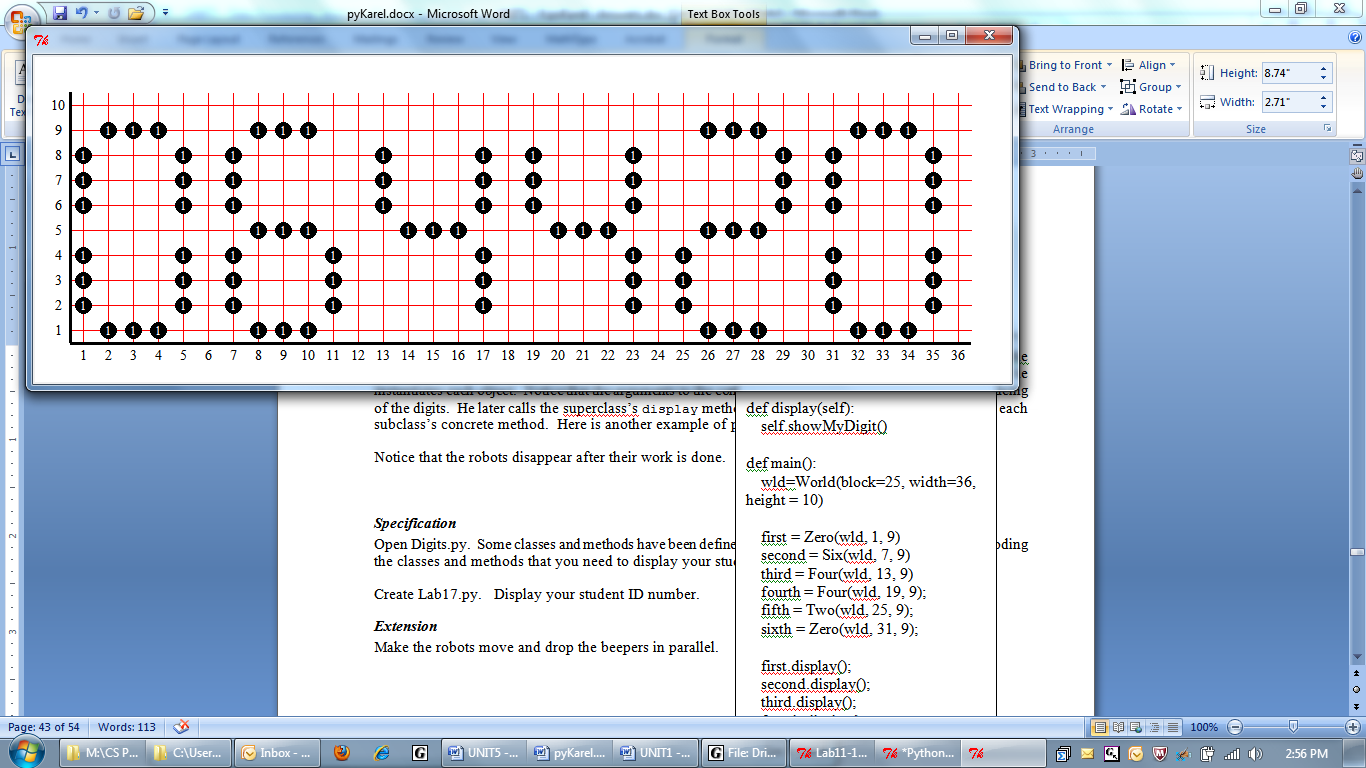
On the other hand, anonymous worlds will create multiple new worlds, which we do not want. Since we are going to pass wld four times, we must instantiate one World object, save it, and pass it by its name wld.

Lab17

Your ID Number

Objective

Extending an abstract class, polymorphic behavior, and parallel programming.

Background

Mr. Torbert’s student ID number is 064420. How would you display his ID number using beepers?

Each digit is modeled by its own class, Zero, Six, Four, Two.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 |  |  |  |  |  |
| 6 |  | 2 |  |  |  |  |
|  | 7 |  |  |  |  |  |
| 5 |  | 3 |  |  |  |  |
|  | 4 |  |  |  |  |  |

Shared code, such as dropping the beepers, is placed in the Digit class. Thus, each digit extends Digit, but defines showMyDigit differently.

Each segment is made of three moves. Let’s always complete the segments in the same order, segment 1, segment 2, segment 3, etc. Segments 1, 3, and 4 make three moves and turn. Segments 2, 5, and 6 make three moves without a turn. Segment 7 is a special case.

The robot always moves, sometimes dropping beepers and sometimes not. Maybe we could control the beepers with a Boolean (True or False). Then we need two versions of the same method for each segment, either dropping beepers or not:

**def** segment1\_On(self):  
 self.threeAndTurn(True)

**def** segment1\_Off(self):   
 self.threeAndTurn(False)

What segments are turned on or off for Zero? Write the code.

**class** Zero(Digit):

**def** showMyDigit(self):

Notice that the robots disappear after their work is done.

**from** Digit **import** \*

**from** threading **import** Thread

**def** display(self):

self.showMyDigit()

**def** main():

wld=World(block=25, width=36, height = 10)

first = Zero(wld, 1, 9)

second = Six(wld, 7, 9)

# more objects here

first.display();

second.display();

# more method calls here

wld.mainloop();

Specification

Open Digits.py. Some classes and methods have been defined for you. Study them carefully. Finish coding the classes and methods that you need to display your student ID number.

Create Lab17.py. Display your student ID number.

Extension

Make the robots display in parallel.

Exercises

Lab 17

1. Look back at all the labs in this unit. Draw the hierarchy showing the subclasses of Robot.

Robot

1. Why did we make all those different classes? Why didn’t we make just keep adding the new methods to Robot?

**Lab18--Project**

An Original pyKarel Lab

Use your imagination and your experience from this unit to create an original pyKarel lab. In formulating your own pyKarel problem, consider the following:

1. How many robots are involved?
2. What are the initial conditions?
3. What is the goal?
4. How many robot-maps do you need to make, and what do they look like?
5. What can the programmer assume? For instance, will there always be a beeper at (5, 4) or do we have to be flexible?
6. What control structures (while, for, if, if-elif) are needed?
7. What existing pyKarel class or classes may be appropriate to use in the solution?
8. What extended classes may be helpful (or necessary) in solving the problem?
9. What abstract classes or interfaces are helpful?
10. What story can you devise to suit this lab?

You may use previous labs as guides to choosing an appropriate problem, but try to be creative as well. You should code your own solution that works before writing up the lab assignment for your peers.

Here is one possible grading rubric. Check with your teacher to be sure.

Lab19

Word document looks just like a pyKarel Lab: 0----1----2----3----4----5

lab assignment for peers is appropriate 0----1----2----3----4----5

lab is at an appropriate level of difficulty: 0----1----2----3----4----5

UML class diagram is correct: 0----1----2----3----4----5

some elements of the task require the

robot to make decisions, i.e., the program

works in different maps: 0----1----2----3----4----5

maps are appropriate: 0----1----2----3----4----5

uses a class method to solve the problem: 0----1----2----3----4----5

uses inheritance appropriately: 0----1----2----3----4----5

uses polymorphism appropriately: 0----1----2----3----4----5

uses parallel programming: 0----1----2----3----4----5

the solution works: 0----1----2----3----4----5

the lab is creative and original: 0----1----2----3----4----5

Lists of Robots

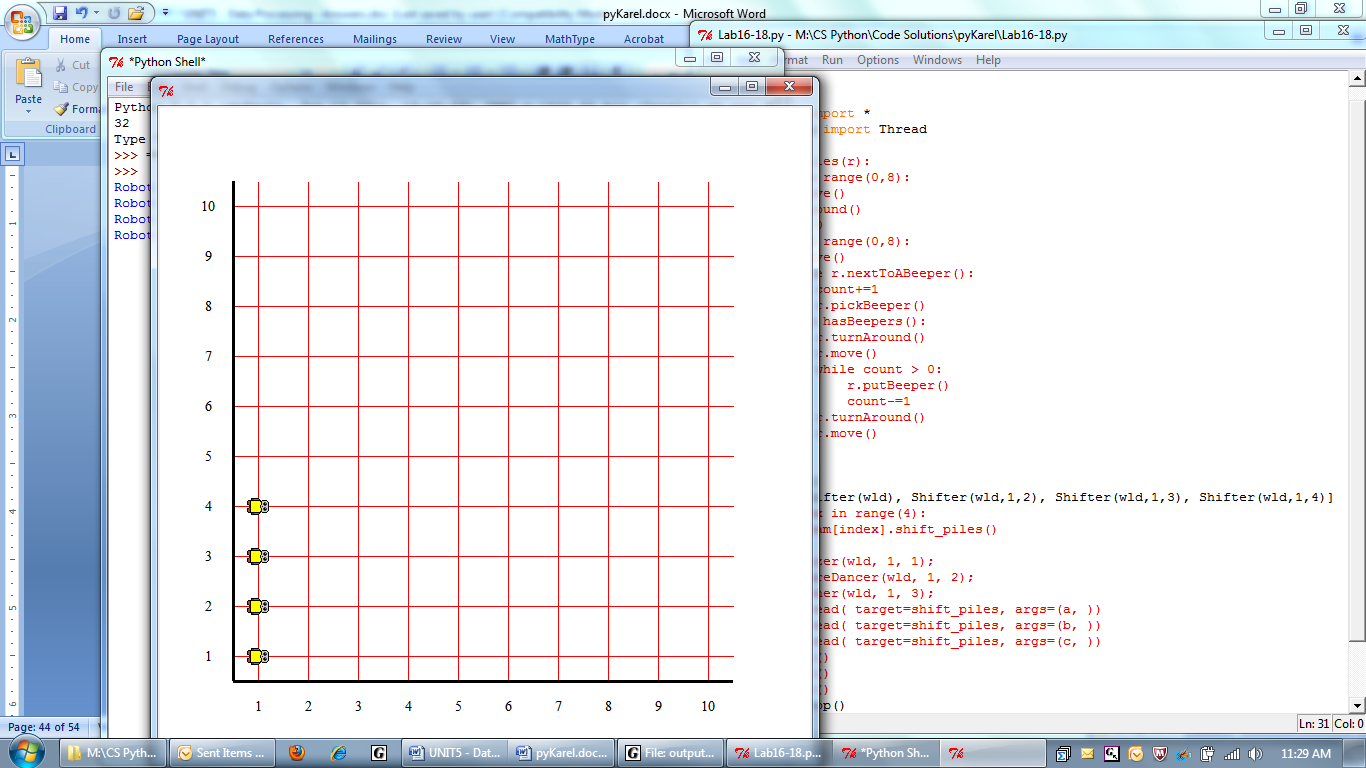
Objective

Using lists.

Background

Multiple objects can be stored with a single identifier in a *list*. A Python list is a linear data structure in which the elements are (sometimes) accessed by *index numbers*. Lists use [ ]. For example:

myTeam=[Athlete(wld), Athlete(wld,1,2), Athlete(wld,1,3), Athlete(wld,1,4)]



creates a list of four Athletes, stored like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| myTeam |  |  |  |  |  |
|  |  | [0] | [1] | [2] | [3] |

but positioned in the default world as shown.

Python allows access to the objects a list using a for-loop:

**for** object in myTeam:

object.putBeeper();

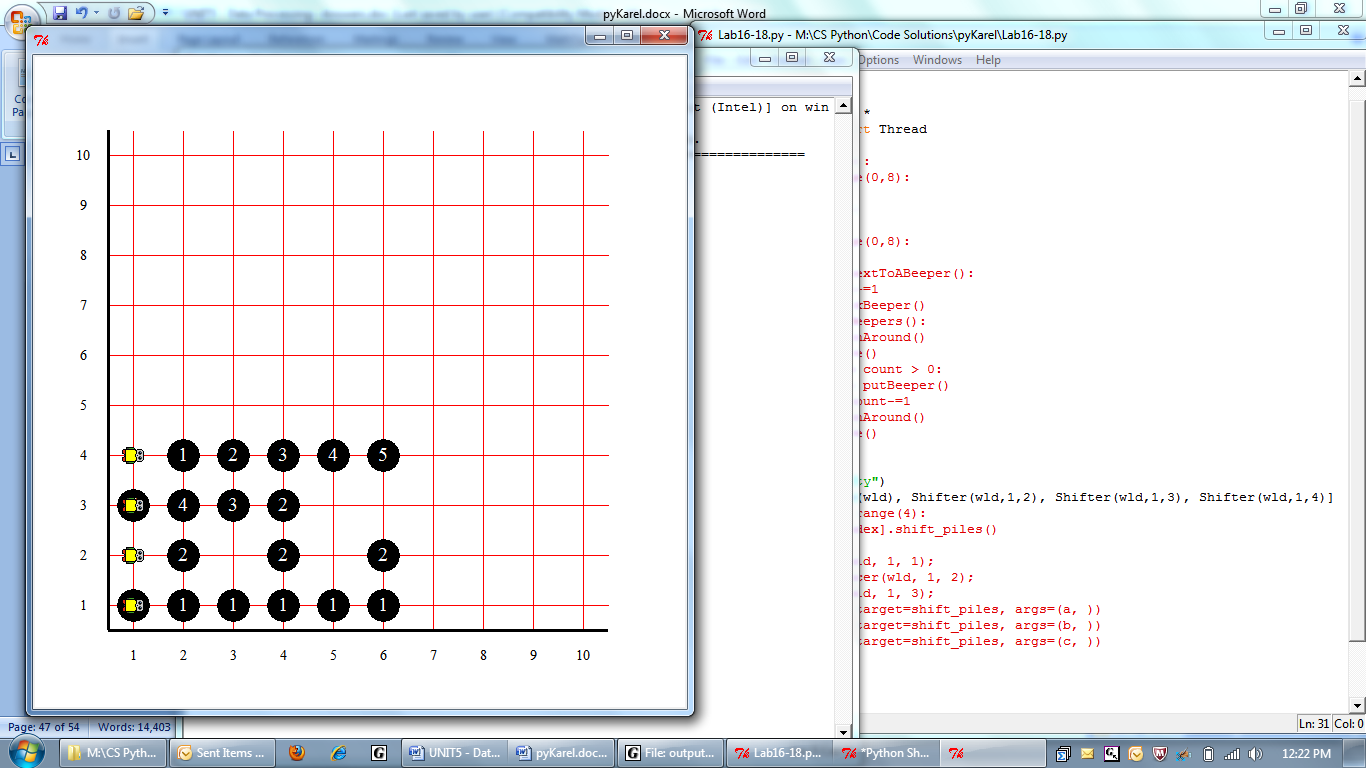
Suppose we want all of the Athletes to move forward five blocks:

**for** object in myTeam:

**for** count in range(5):

object.move();

Note that these robots will not all move at the same time. Rather, they will take turns as the outside for-loop repeats. In order to execute in parallel, you must use threads.



Specification

Create Lab19.py. Import Shifter (see Lab09). Use the “shifty” world. Make myTeam, a list of Shifter robots. Tell each object to shift the piles of beepers. Don’t execute in parallel.

Extension

Return to Lab05.py, the shuttle run. Rewrite it using a list of three Racers. Run the shuttle run in parallel.

Extension 2

Modify Lab19 to use Swimmer objects. Define the same myfunc method that you did in Lab15. (Alternatively, from Lab15 import myfunc.)

def myfunc(arg):

arg.swim\_laps()

Make a list of 30 swimmers, side-by-side on row=1. You will need to use a for-loop and list's append command. Make all 30 swimmers take laps in parallel.

Lab20

Recursive Racer

In Racer we defined an *iterative* method sprint as follows:

**def** sprint(self, n):

**for** x **in range**(0,n):

self.move()

The following solutions are *recursive* because the method sprint calls the method sprint—it calls a copy of itself. The two solutions below act exactly the same, but the code on the right is often easier to read. Since the **return** statement forces the method to end early, the **else:**  is not required.

|  |  |
| --- | --- |
| **def** sprint(self, n):  **if** n<=0: # base case  **return**  **else**:  self.move()  self.sprint(n-1) # recursive call | **def** sprint(self, n):  **if** n<=0:  **return**  self.move()  self.sprint(n-1) |

Let's trace the call sprint(2). It is useful to imagine the recursive calls as stacking on top of one another, as shown. Eventually, the code reaches the base case and all the calls come back out, in reverse order.

sprint(0)

sprint(1)

sprint(2)

Since n = 2, which is not equal to zero, the robot will move and then call sprint(1). In sprint(1) the value of n is 1, which is also not equal to zero, so the robot will move and call sprint(0). In sprint(0) the value of n is 0, which is equal to zero, so the robot would do nothing and the call to sprint(0) would return, thus ending the method. Then the call to sprint(1) would end, then the call to sprint(2) would end. In all, our robot moved twice.

A common beginner’s mistake with recursive algorithms is to recur infinitely. (In IDLE, infinite recursion gives you lots of red error messages, ending with "RuntimeError: maximum recursion depth exceeded".) The following code results in such an error:

**def** sprint(self, n):

self.sprint(n-1) #ERROR: makes the recursive call before

**if** n<=0: # checking the base case!

**return** # Keeps calling sprint() "forever"

To avoid infinite recursive calls, it is best to think of recursion as a two-step process: 1) check the base case first to see when the recursive calling should stop, then 2) if the base case is not met, call the recursive method again with a "one-step smaller" argument so that the base case will eventually be reached.

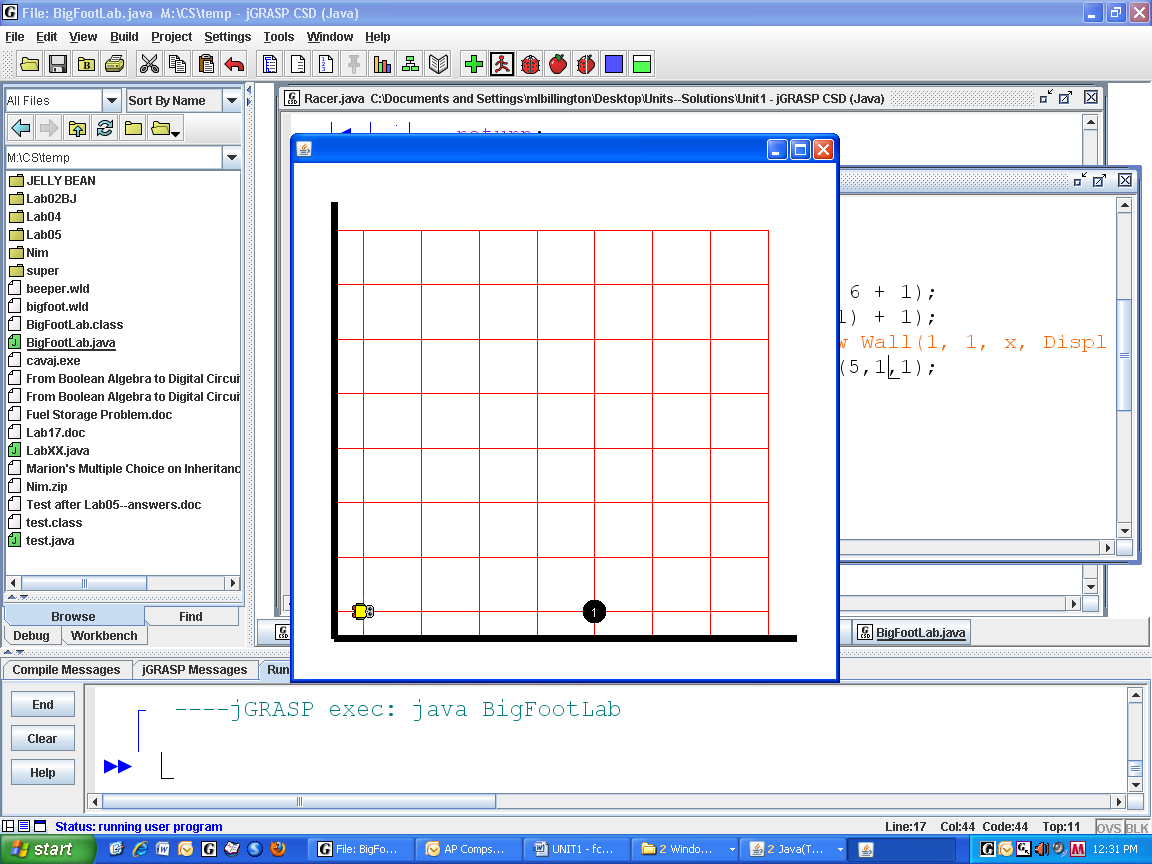
***Assignments:***

Create RecursiveRacer.py. Define RecursiveRacer by copying the Racer class from Racers.py and changing its name. Re-write the sprint, put, and pick methods as recursive methods. Copy Lab05.py and save it as a new application, Lab20.py. Modify the code to use a main method and RecursiveRacer objects. Run.

Return to the Pirate class and Lab07. All three instance methods in Pirate can be written recursively. Rewrite them recursively, then run Lab07.py.

Discussion

Stored Recursive Calls



Let a beeper be placed on the same y-coordinate as your robot, some unknown distance due east of your robot’s current location. The task is to retrieve the beeper, turn left, move north the same distance, and drop the beeper. How would you accomplish this task?

The following *iterative* solution counts how many blocks we travel east during the while-loop, then uses that number to control the for-loop, which travels north:

count = 0

**while** **not** karel.nextToABeeper():

karel.move()

count = count + 1

karel.pickBeeper()

karel.turnLeft()

**for** k **in range**(count):

karel.move()

karel.putBeeper()

A more elegant solution uses a *recursive* method:

**def** recur(arg):

**if** arg.nextToABeeper(): # base case

arg.pickBeeper()

arg.turnLeft()

**else**:

arg.move()

recur(arg) # recursive call

arg.move() # this command is stored;

# eventually it is executed

As long as nextToABeeper is false, we move and call recur. Each call to recur will test nextToABeeper on the block that is one step further east. At some point nextToABeeper will be true. The robot picks the beeper up and turns left—now facing north. The most recent call to recur, the one that found the beeper, will end. When it ends, the previous call to recur, the call to the call that found the beeper, will continue where it was interrupted, i.e., there is a second move that has to execute.

This move will now be made toward the north. When each call to recur ends, the one before it continues, moves one block farther north, and ends itself. Every move before the recursive call goes east and every move afterwards goes north. Since there is the same number of moves before and after the recursive call, the robot travels the same distance north as it had traveled east.

If this doesn’t make sense, imagine that the beeper was originally located only one block in front of karel and trace through the execution of the method. What if the beeper was originally located at the exact same intersection as karel?

Use the technique of stored recursive commands for the next lab.

Lab21

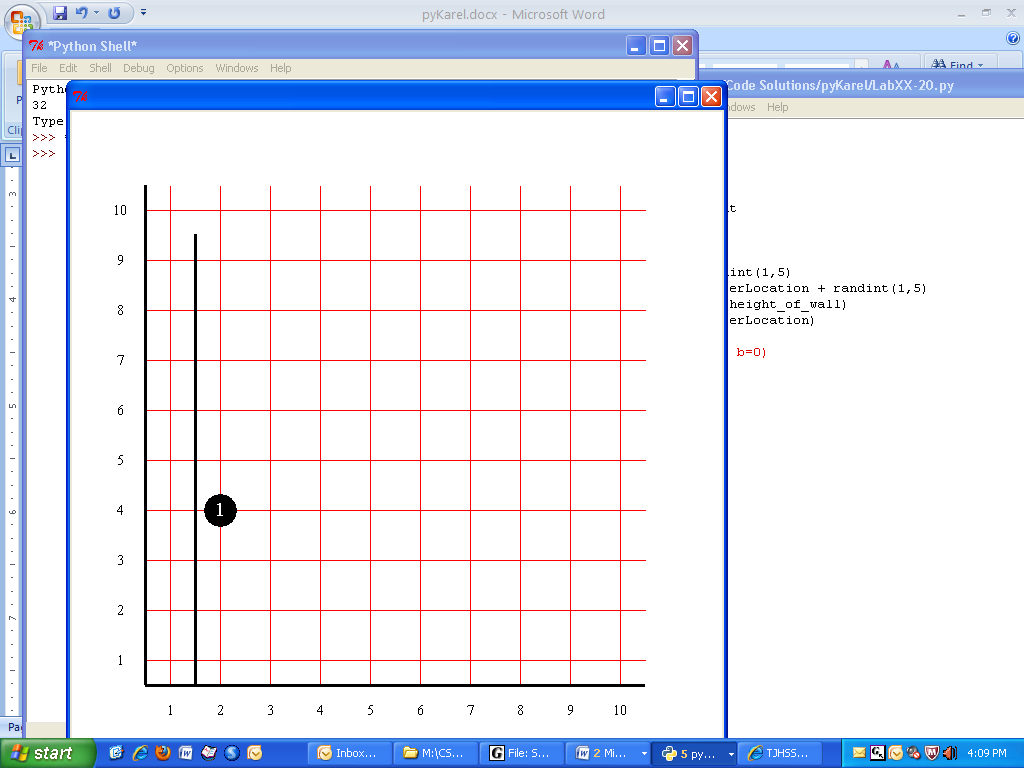
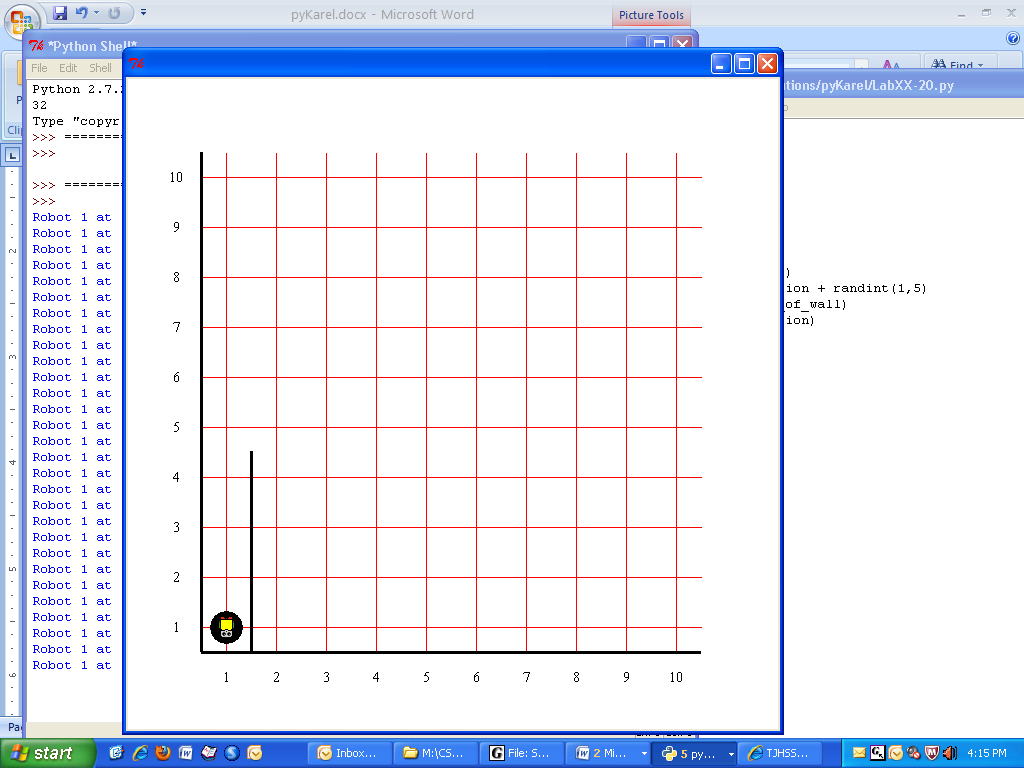
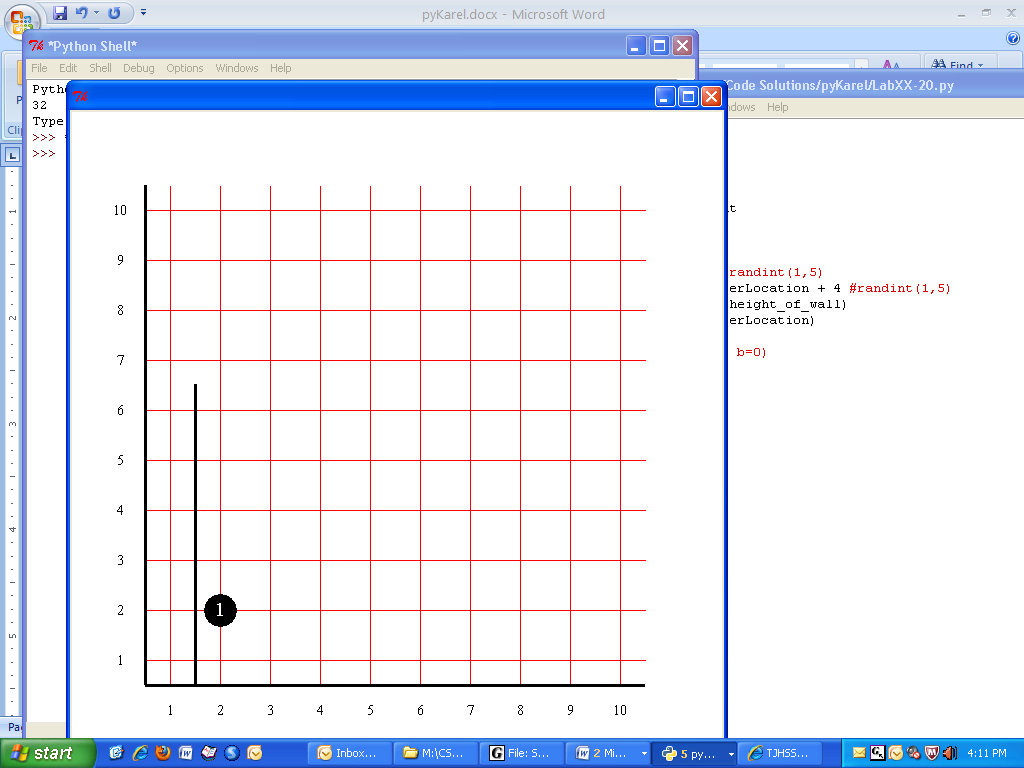
Recursion: Seeking the Beeper

Objective

To implement stored recursion.

Background

Your Seeker begins at (1, 1) facing north. There is a north-south wall of unknown length blocking your robot’s path. Directly on the other side of that wall, on the 2nd x-coordinate, there is a beeper. The beeper is against the wall but at a random y-coordinate. You must recursively retrieve the beeper, bring it back to (1, 1), and put it down. Here are three possible worlds:



The third screen-shot shows the end result, regardless of the height of the wall or the original placement of the beeper on the 2nd x-coordinate.

Specification

Create Seeker.py. Import Athlete. Define the Seeker class which extends Athlete. Seekers start out with no beepers. Define a recursive fetchBeeper.

Open Lab21.py. Notice how the beeper is placed at a random location and how the height of the wall is chosen at random. Run.

Test Data

As shown above in the far right image.

Lab22

Seeking the Beeper, Part II

Objective

Design and implement an algorithm.

Background

What does this code do?

random\_width = randint(1,50) + 25

random\_height = randint(1,25) + 12

random\_x = randint(1, random\_width)

random\_y = randint(1, random\_height)

wld=World(block = 20, width=random\_width, height=random\_height)

wld.addBeeper(random\_x, random\_y)

The task is to get the beeper.

There are a number of different ways to approach this problem. You could have a robot start at (1, 1) and search the entire world in some sort of pattern. You could have a bunch of robots, maybe even an array, start along 1st and march east. You could use iteration or recursion.

Specification

Open Lab22.py. Add code and pick up the beeper.

Test Data

However you decide to approach this problem, if the beeper gets picked up then your program works.

Glossary

**Abstract**: A class or method that does not have a definition and must be implemented in a subclass.

**Actual Argument**: A data item passed in a method call. Arguments appear inside parentheses. An actual argument can be a literal value, a variable, or an expression.

**Attributes:** The data that an object “knows” about itself. The values that specify the internal state of an object. Robots have four major attributes, x, y, direction, and beepers.

**Anonymous**: An object that is instantiated but not named. For example, Robot(wld).move()

**Body**: The commands that actually get executed when a method is called.

**Class**: A user-defined data type that specified a blueprint for objects. A class may define both class and instance methods. A class usually specifies a constructor method.

**Class methods**: Methods that are defined in an application. Class methods are called in the application, not by dot-notation on the object.

**Constructor**: the method that instantiates the object and initializes its attributes. Constructors may provide for default arguments, variable length arguments, and keyword arguments.

**Formal Argument (parameter)**: A variable or placeholder for a value in the definition of a method.

**Hasa**: a word expressing the relationship between an application and an instantiated object. The application “has a” object.

**Header**: The first line of a class or method definition.

**Inherited Method**: A method defined in a superclass that is available, through inheritance, to a subclass.

**Instance methods**: Methods that are defined in a class. Instance methods are called by dot-notation on the object (an instance).

**Instantiate:** To create an object from the blueprint or template that is specified in the class.

**Isa**: A word expressing the inheritance relationship of classes in a hierarchy. E.g., Athlete “is a” Robot.

**Module**: The building blocks of Python applications. Each module consists of data and functionality.

**Override**: A method in a subclass with the same name as that in the superclass, but a different definition.

**Polymorphic behavior**: Calling the "same" method, but resulting in different behavior. When a method defined in a superclass is overridden in different subclasses, then the method call executes one of the subclass's methods.

**Subclass**: A class that inherits from another class. Also known as a derived class or a child class.

**Superclass**: A class from which another class inherits. Also known as a base class or a parent class.