Outline

1. Objects in Turtle
   - a satellite circulating an orbiting planet
2. Encapsulation
   - data hiding
   - polynomials in one variable
3. Inheritance
   - base classes and derived classes
   - points and circles
4. Polymorphism and Wrapping
   - builtin functions
   - wrapping: Karl the Robot

MCS 260 Lecture 26
Introduction to Computer Science
Jan Verschelde, 11 March 2016

Evolution of Computer Languages

We defined OOP as a new programming paradigm.
Learning a language is to learn idioms,
i.e.: how to express yourself properly.

1. procedure-oriented languages: C, FORTRAN 77
   Algorithms are central in program development.

   We use flowcharts or pseudo code to design programs.

2. object-oriented languages: Ada, C++, Smalltalk
   Objects belonging to ***classes*** organized in a ***hierarchy***
   are the main building blocks for programs.

   To design we use the Unified Modeling Language.

3. framework languages: Java, Python

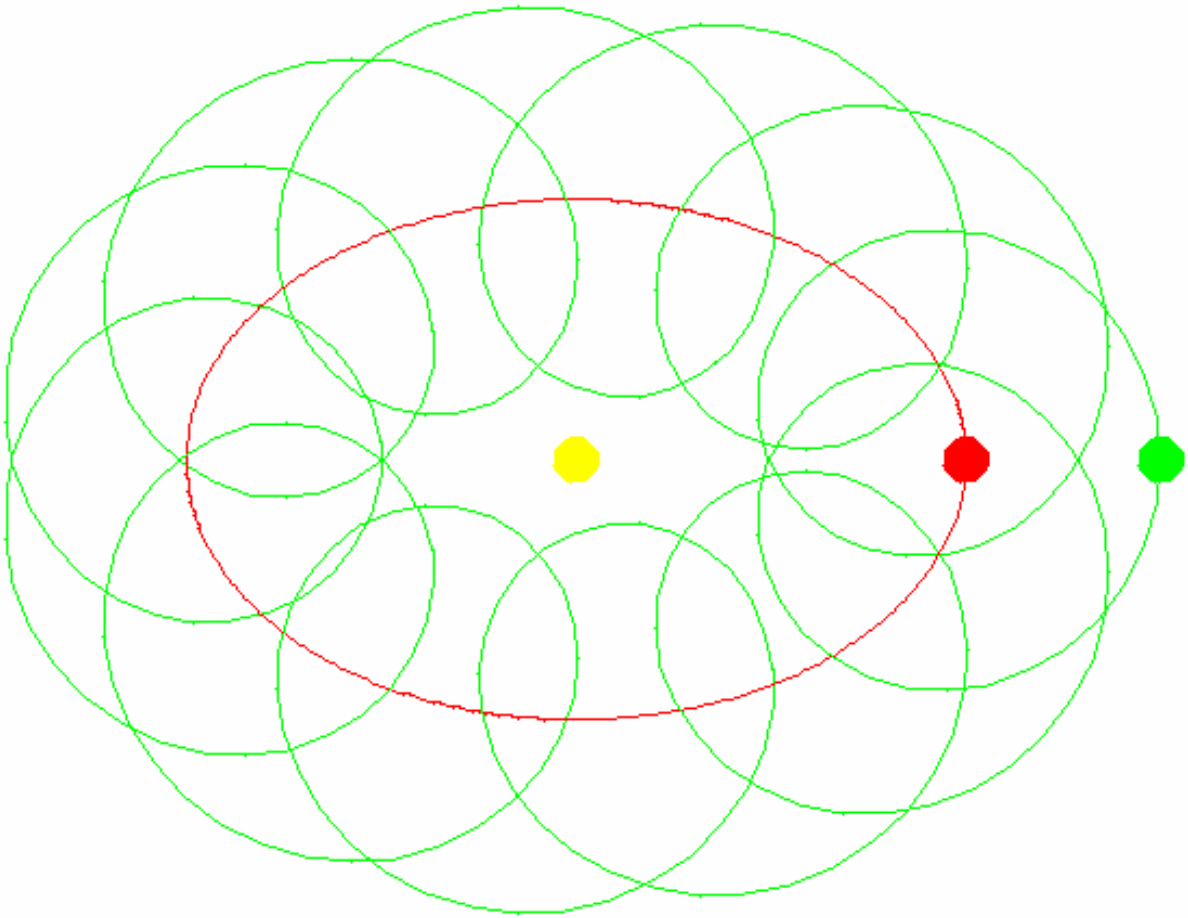
   A ***framework*** is a collection of classes
   that provides a set of services for a given domain.

encapsulation and inheritance polymorphism and wrapping

1. Objects in Turtle
   - a satellite circulating an orbiting planet
2. Encapsulation
   - data hiding
   - polynomials in one variable
3. Inheritance
   - base classes and derived classes
   - points and circles
4. Polymorphism and Wrapping
   - builtin functions
   - wrapping: Karl the Robot

Moon circles orbiting Earth

running `orbiting.py`



The sun is yellow, the earth is red, and the moon is green.

objects in turtle

Our sun-earth-moon simulation involves 3 turtles

The module `turtle` exports `Turtle` class:

```
>>> from turtle import Turtle
>>> sun = Turtle(shape='circle')
>>> sun.color('yellow')
>>> sun.position()
(0.00,0.00)
```

1. `sun` is an instance of the `Turtle` class, instantiated with value `'circle'` of `shape` attribute.
2. The method `color` changed the color of the object.
3. The method `position` returns the current position.

three turtles

```
from math import cos, sin, pi
from turtle import Turtle

def three_turtles(radius):
    """
    Returns a tuple of three turtles,
    the first turtle is centered at (0, 0),
    the second turtle is centered at (2*radius, 0),
    the third turtle is at (3*radius, 0).
    """
    green = Turtle(shape='circle')
    green.color('green')
    green.penup()
    green.goto(3*radius, 0)
    red = Turtle(shape='circle')
    red.color('red')
    red.penup()
    red.goto(2*radius, 0)
    yellow = Turtle(shape='circle')
    yellow.color('yellow')
    return (yellow, red, green)
```

the main program

```
def main():
    """
    Displays one turtle orbiting another one.
    """
    from turtle import window_height
    wdh = window_height()/6
    (distance, ndays) = (wdh, 365)
    day = 2*pi/ndays
    mth = 12*day
    (era, mna) = (day, mth)
    (sun, earth, moon) = three_turtles(distance)
    earth.pendown()
    moon.pendown()
    for _ in range(0, ndays):
        circulate(earth, (0, 0), 2*distance, era)
        pull_circulate(earth, moon, distance, mna)
        (era, mna)  = (era + day, mna + mth)
    input("hit any key to exit ")
```

orbit of the earth

Parametric trajectory of an ellipse centered at (c_x, c_y)
with rightmost point at $(r, 0)$ and highest point at $(0, 2r/3)$:

$$(x(\theta), y(\theta)) = (c_x + r\cos(\theta), c_y + 2r/3\sin(\theta))$$

where θ ranges from 0 to 2π.

```
def circulate(trt, ctr, rad, agl):
    """
    Places the turtle trt at angle agl,
    on an ellipse with center at ctr, (rad, 0)
    is the rightmost point and (0, 2*rad/3) is
    the highest point on the ellipse.
    """
    xpt = ctr[0] + rad*cos(agl)
    ypt = ctr[1] + (2*rad/3)*sin(agl)
    trt.goto(xpt, ypt)
```

orbit of the moon

If the earth is at (c_x, c_y), then the moon traces a circle as

$$(x(\theta), y(\theta)) = (c_x + r\cos(\theta), c_y + r\sin(\theta)),$$

for θ ranging between 0 and 2π.

```
def pull_circulate(tr0, tr1, rad, agl):
    """
    Pulls tr1 in the direction of tr0
    so the distance between tr1 and tr0 is rad.
    Then places tr1 on a circle centered at
    the position tr0, with angle agl.
    """
    ctr = tr0.position()
    xpt = ctr[0] + rad*cos(agl)
    ypt = ctr[1] + rad*sin(agl)
    tr1.goto(xpt, ypt)
```

top down versus bottom up design

Our design of the sun-earth-moon simulation is ***top down***:

1. There is a main program that *controls* the action.
2. For each day, we tell the earth to move.

An object-oriented design would be ***bottom up***:

1. The objects earth and moon have `move()` methods.
2. Instead of calling `earth.move()`, and `moon.move()`,
   - one thread of execution would be assigned to `earth`, and
   - another thread of execution would be assigned to `moon`,

   so that both object move on their own pace.

The main program in a bottom up design launches threads.

encapsulation and inheritance polymorphism and wrapping

1. Objects in Turtle
   - a satellite circulating an orbiting planet
2. Encapsulation
   - data hiding
   - polynomials in one variable
3. Inheritance
   - base classes and derived classes
   - points and circles
4. Polymorphism and Wrapping
   - builtin functions
   - wrapping: Karl the Robot

Encapsulation

data hiding

Information hiding is important in modular design.

In object oriented programming,
we can hide the representation of an object.

Hidden data attributes are called *private*,
opposed to *public* (the default).

In Python, starting a name with _ _
makes a data attribute private.

encapsulation and inheritance polymorphism and wrapping

1. Objects in Turtle
   - a satellite circulating an orbiting planet
2. Encapsulation
   - data hiding
   - polynomials in one variable
3. Inheritance
   - base classes and derived classes
   - points and circles
4. Polymorphism and Wrapping
   - builtin functions
   - wrapping: Karl the Robot

Polynomials in One Variable

the need for data hiding

Problem: design a class to manipulate polynomials.

Representing $2x^8 - 3x^2 + 7$:

1. as coefficient vector $c = [7, 0, -3, 0, 0, 0, 0, 0, 2]$,
   $c[i]$ is coefficient of x^i;
2. as list of tuples $L = [(2, 8), (-3, 2), (7, 0)]$
   $(c, i) \in L$ represents cx^i.

Both representations have advantages and disavantages.

Solution offered by *encapsulation*:

1. make representation of the object polynomial private;
2. access to the data only via specific methods.
3. resolve the problem of normalization, i.e.: `p == 0`?

the class `Poly` in the file `classpoly.py`

```
class Poly(object):
    """
    Defines a polynomial in one variable.
    """
    def __init__(self, c=0, p=0):
        "monomial with coefficient c and power p"
        if c == 0:
            self.__terms = []
        else:
            self.__terms = [(c, p)]
```

Notice:

1. We hide the coefficient list via _ _.
2. We do not store monomials with zero coefficients.

string representations: overloading __str__

```
>>> from classpoly import Poly
>>> p = Poly(2,3)
>>> str(p)
'+2*x^3'
```

```
    def __str__(self):
        "returns a polynomial as a string"
        prt = ''
        for mon in self.__terms:
            prt += '%+2.f*x^%d' % (mon[0], mon[1])
        if prt == '':
            return '0'
        else:
            return prt
```

adding a monomial to a polynomial

```
    def __add__(self, other):
        "adds two polynomials"
        result = Poly()
        for mon in self.__terms:
            result.addmon(mon[0], mon[1])
        for mon in other.__terms:
            result.addmon(mon[0], mon[1])
        return result
```

```
>>> p = Poly(2,4)
>>> q = Poly(3,4)
>>> s = p+q
>>> str(s)
'+5*x^4'
```

Avoid storing zero coefficients!

adding a monomial – avoiding to store zero

Adding a monomial to a polynomial:

```
def addmon(self, cff, deg):
    "adds a monomial"
    if cff != 0:
        done = False
        for i in range(0, len(self.__terms)):
            mon = self.__terms[i]
            if mon[1] == deg:
                newcff = cff + mon[0]
                del(self.__terms[i])
                if newcff != 0:
                    self.__terms.append((newcff, deg))
                done = True
                break
        if not done:
            self.__terms.append((cff, deg))
```

callable objects

```
>>> from classpoly import Poly
>>> p = Poly(2,4) + Poly(1,0)
>>> print(p)
+2*x^4+1*x^0
>>> p(3)
163
```

The `p(3)` is defined by

```
    def __call__(self, x):
        "returns the value at x"
        result = 0
        for (cff, deg) in self.__terms:
            result += cff*x**deg
        return result
```

encapsulation and inheritance polymorphism and wrapping

1. Objects in Turtle
   - a satellite circulating an orbiting planet
2. Encapsulation
   - data hiding
   - polynomials in one variable
3. Inheritance
   - base classes and derived classes
   - points and circles
4. Polymorphism and Wrapping
   - builtin functions
   - wrapping: Karl the Robot

Inheritance

base classes and derived classes

We can create new classes from existing classes.
These new classes are *derived* from *base* classes.

The derived class *inherits* the attributes of the base class and usually contains additional attributes.
Inheritance is a powerful mechanism to reuse software.
To control complexity, we add extra features later.

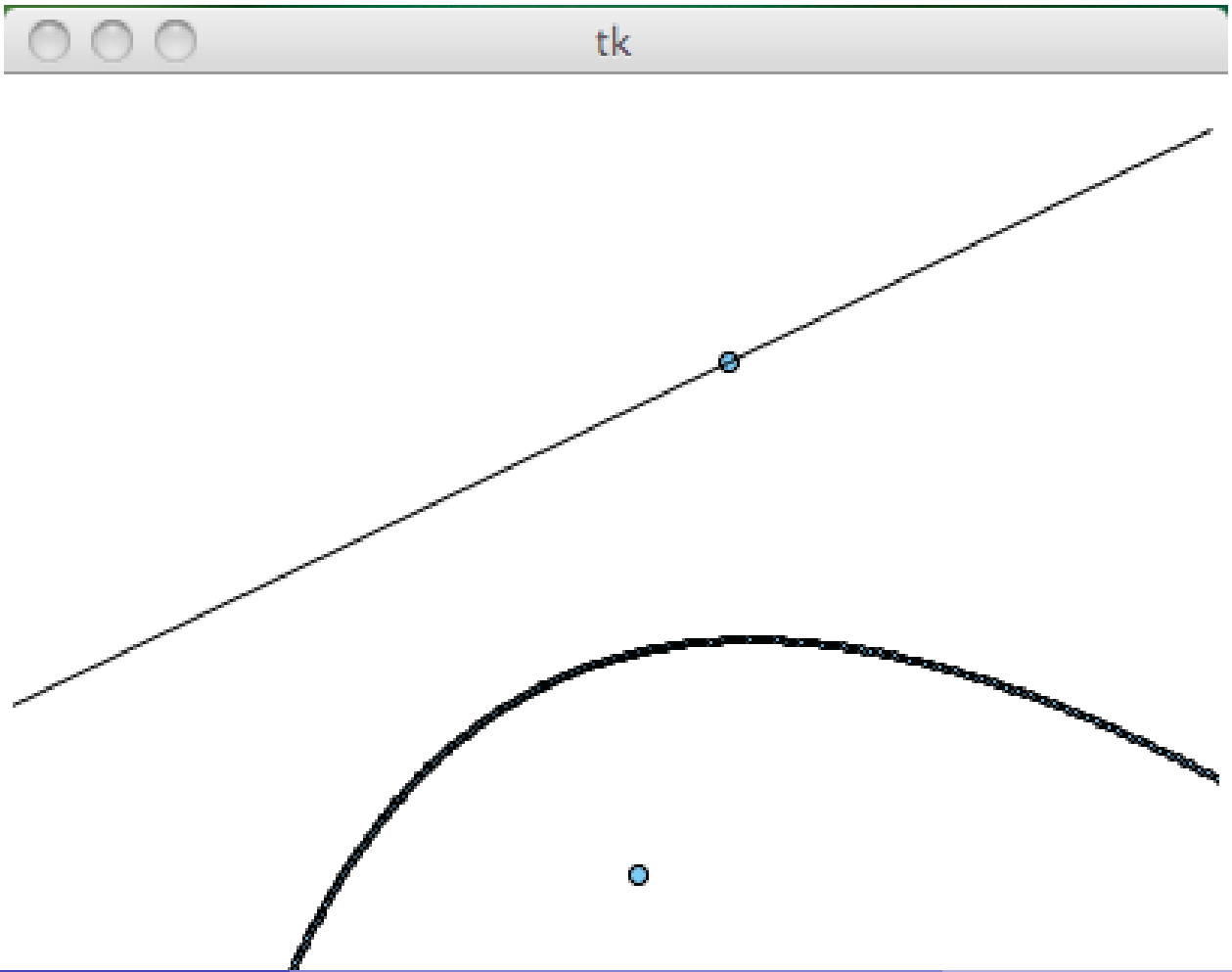
We distinguish between single and multiple inheritance:

- **single** a derived class inherits from only *one* class;
- **multiple** derivation from *multiple* different classes.

Multiple inheritance may lead to name clashes,
in case the parents have methods with same name.

an example of multiple inheritance

- A line inherits from a point,
  as a line is a point and a direction.
- A parabola has a focus and a directrix,
  so a parabola inherits from a point and a line.



encapsulation and inheritance polymorphism and wrapping

1. Objects in Turtle
   - a satellite circulating an orbiting planet
2. Encapsulation
   - data hiding
   - polynomials in one variable
3. Inheritance
   - base classes and derived classes
   - points and circles
4. Polymorphism and Wrapping
   - builtin functions
   - wrapping: Karl the Robot

Points and Circles

a first example of inheritance

We represent a point in the plane by the values for its coordinates, usually called x and y.

The class `Point` has attributes `x` and `y`
and a string representation.

A circle is determined by a center and radius.

The class `Circle` will inherit from the class `Point` to represent its center. The radius of the circle is an additional object data attribute. The function `area` is an additional functional attribute.

The class `Circle` will use the string representation of `Point` for its center and extend the `__str()__` function for its radius.

the class Point in the file classpoint.py

The definition of the class Point is placed in a separate file `classpoint.py`, available to import as a module.

```
class Point(object):
    "defines a point in the plane"

    def __init__(self, a=0, b=0):
        "constructs a point in the plane"
        self.xpt = a
        self.ypt = b

    def __str__(self):
        "returns string representation of a point"
        return '( %.4e, %.4e )' % (self.xpt, self.ypt)
```

The string representation uses scientific notation for the coordinates, with 4 digits after the decimal point.

the class Circle in the file classcircle.py

```
from classpoint import Point

class Circle(Point):
    """
    The circle class inherites from Point.
    """
    def __init__(self, a=0, b=0, r=0):
        "the center is (a,b), radius = r"
        Point.__init__(self, a, b)
        self.rad = r

    def __str__(self):
        "returns string representation of a circle"
        prt = 'center : ' + Point.__str__(self) + '\n'
        prt += 'radius : ' + '%.4e' % self.rad
        return prt

    def area(self):
        "returns the area of the circle"
        from math import pi
        return pi*self.rad**2
```

using the classes in the script `pointcircle.py`

```
from classpoint import Point
from classcircle import Circle

print('using classes point and circle')
XPT = float(input('give x : '))
YPT = float(input('give y : '))
PNT = Point(XPT, YPT)
print(('the point ' + str(PNT)))
print(('has coordinates ', PNT.xpt, PNT.ypt))
RAD = float(input('give the radius : '))
CRC = Circle(XPT, YPT, RAD)
print(('the circle :\n' + str(CRC)))
print(('has area ', CRC.area()))
```

running the script `pointcircle.py`

Executing at the command prompt `$`:

```
$ python pointcircle.py
using classes point and circle
give x : 2.3
give y : -0.2
the point ( 2.3000e+00, -2.0000e-01 )
has coordinates  2.3 -0.2
give r : 0.762
the circle :
center : ( 2.3000e+00, -2.0000e-01 )
radius : 7.6200e-01
has area  1.82414692475
```

Polymorphism

method overriding

Recall Python's dynamic typing:
$\rightarrow$ during run time Python determines the type of a variable to know which methods may be applied.

Calling `str()` on `x` gives different strings, depending on whether `x` is an instance of the class `Point` or `Circle`.

In `x.method()`, the meaning of the `method` depends on the type (or class) of `x`.

Polymorphism allows objects of different classes related by inheritance to respond differently to the same method.

Giving new definitions for methods in the derived class with the same name as in the base class is *method overriding*.

encapsulation and inheritance polymorphism and wrapping

1. Objects in Turtle
   - a satellite circulating an orbiting planet
2. Encapsulation
   - data hiding
   - polynomials in one variable
3. Inheritance
   - base classes and derived classes
   - points and circles
4. Polymorphism and Wrapping
   - builtin functions
   - wrapping: Karl the Robot

builtin functions to check types and relationships

We can check whether a variable is of a certain type:

```
>>> isinstance('z',int)
False
>>> isinstance(3,int)
True
```

In the OOP parlance, we check whether an object is an instance of another object or class.

`issubclass()` verifies relationships between classes:

```
>>> from classcircle import *
>>> issubclass(Point,Circle)
False
>>> issubclass(Circle,Point)
True
```

functions `hasattr()` and `getattr()`

```
>>> from classpoint import Point
>>> p = Point(2,3)
>>> hasattr(p,'xpt')
True
>>> hasattr(p,'z')
False
```

With `getattr` we get the value of an attribute:

```
>>> getattr(p,'xpt')
2
```

With `setattr` we set the value of an attribute"

```
>>> setattr(p,'xpt',10)
>>> str(p)
'( 1.0000e+01, 3.0000e+00 )'
```

To delete an attribute:

```
>>> delattr(p,'xpt')
>>> hasattr(p,'xpt')
False
```

encapsulation and inheritance
polymorphism and wrapping

1. Objects in Turtle
   - a satellite circulating an orbiting planet
2. Encapsulation
   - data hiding
   - polynomials in one variable
3. Inheritance
   - base classes and derived classes
   - points and circles
4. Polymorphism and Wrapping
   - builtin functions
   - wrapping: Karl the Robot

Wrapping: Karl the Robot

customized interfaces

Wrapping consists in

1. making a derived class of a class or module;
2. adding, modifying, or removing functionality.

Imagine a robot with two primitive operations:

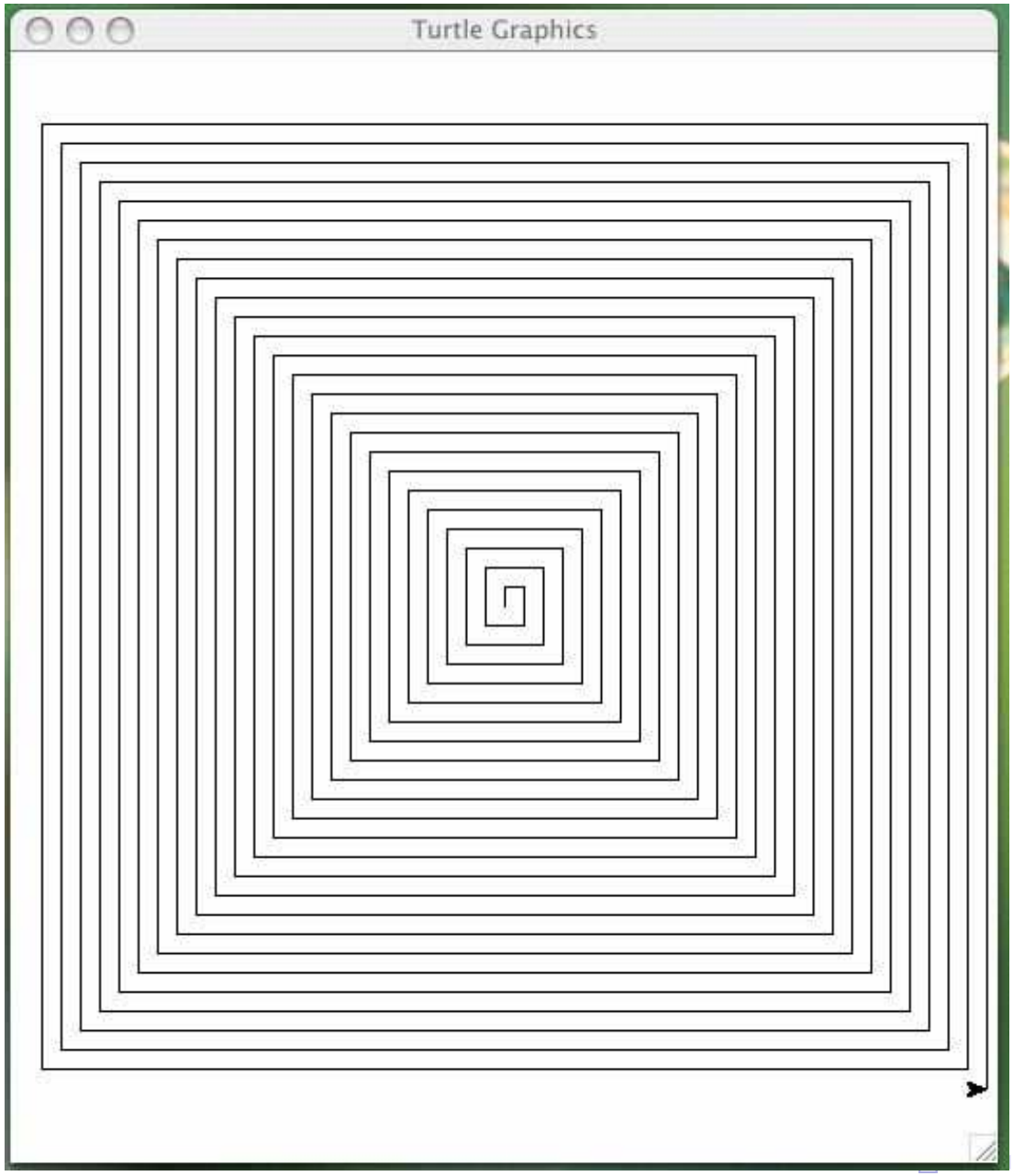
1. turn right using angle of 90 degrees,
2. do one step forward of distance d.

The robot can walk over a tiled grid (checkerboard).
Data attributes for the robot:

1. the coordinates of the current position,
2. the orientation vector,
3. the step size (or size of the tiles).

With turtle graphics we draw the path of the robot.

a spiral path



the class Robot

```
from turtle import Turtle
class Robot:
    """
    Plots the path of a robot in a turtle window.
    """
    def __init__(self, a=0, b=0, d=10):
        """
        Initializes the robot at position (a,b)
        and sets the step size to d pixels.
        """
    def turn(self):
        """
        Makes one right turn of 90 degrees.
        """
    def step(self):
        """
        Does one step in the current direction.
        """
```

the constructor

The turtle, called `bob`, is a data attribute.

```
def __init__(self, a=0, b=0, d=10):
    """
    Initializes the robot at position (a,b)
    and sets the step size to d pixels.
    """
    self.bob = Turtle()
    self.bob.penup()
    self.bob.goto(a, b)
    self.stp = d    # step size
    self.xdr = 0    # x-direction
    self.ydr = 1    # y-direction
    self.bob.pendown()
```

The coordinates of the position of the robot can be obtained via `self.bob.position()`.

turning and stepping

Rotating the direction (check $(0,1)$, $(0,-1)$, $(1,0)$, $(-1,0)$):

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} \mapsto \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v_y \\ -v_x \end{bmatrix}$$

```
def turn(self):
    """
    Makes one right turn of 90 degrees.
    """
    (self.xdr, self.ydr) = (self.ydr, -self.xdr)

def step(self):
    """
    Does one step in the current direction.
    """
    (xps, yps) = self.bob.position()
    xps = xps + self.stp*self.xdr
    yps = yps + self.stp*self.ydr
    self.bob.goto(xps, yps)
```

the main program

```
def main():
    """
    Lets the robot draw a spiral path.
    """
    karl = Robot()
    karl.step()
    nbs = 1  # number of steps
    for k in range(0, 50):
        karl.turn()
        for _ in range(0, nbs):
            karl.step()
        if k % 2 == 0:
            nbs = nbs + 1
    input('hit enter to exit')
```

Summary + Assignments

Read chapter 10 in *Python Programming in Context.*

Assignments:

1. An additional invariant to the representation of a polynomial in one variable is that its monomials are ordered along descending degree. Describe how you would implement this in the class `Poly`.
2. Derive a class `Triangle` from the class `Point`.
   The constructor should take three points as argument.
   Write an `area` function.
3. Write Python code to define a class Queue (FIFO).
   It should export `enqueue` and `dequeue`.
4. Give code so the robot draws a pentagon.
5. Extend the turn method of the robot class with the rotation angle.
   By default, the angle is $\pi/2$.

More Assignments

6. Change the `orbiting.py` program to simulate the sun-earth-moon orbits with a class `Planet`. Specify what data attributes are relevant to display the simulation. Add a functional attribute `Circulate` to the class `Planet`. Write a main program to run the simulation with multiple planets.
7. Extend the program of the previous exercise with a class `Satellite`. The class `Satellite` inherites from the class `Planet`. Write a main program to run the sun-earth-moon simulation with at least two moons.