Outline

1. Encapsulation
   data hiding
   polynomials in one variable

2. Inheritance
   base classes and derived classes
   points and circles

3. Polymorphism and Wrapping
   builtin functions
   wrapping: Karl the Robot

4. Summary + Assignments

MCS 260 Lecture 26
Introduction to Computer Science
Jan Verschelde, 24 October 2008
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.
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.
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.
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

- data hiding
- polynomials in one variable

Inheritance

- base classes and derived classes
- points and circles

Polymorphism and Wrapping

- builtin functions
- wrapping: Karl the Robot

Summary + Assignments
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

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.
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
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. Encapsulation
   data hiding
   polynomials in one variable

2. Inheritance
   base classes and derived classes
   points and circles

3. Polymorphism and Wrapping
   builtin functions
   wrapping: Karl the Robot

4. Summary + Assignments
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 disadvantages.

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.: $p == 0$?
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 disadvantages.

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.: $p == 0$?
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 disadvantages.

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.: \(p == 0\)?
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 disadvantages.

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.: $p == 0$?
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 disadvantages.

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.: $p == 0$?
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 disadvantages.

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.: $p == 0$?
class Poly:
    "defines a polynomial in one variable"

def __init__(self, c=0, p=0):
    "monomial with coefficient c and power p"
    if c==0:
        self.__L = []
    else:
        self.__L = [(c,p)]

Notice:

① We hide the coefficient list via ___.
② We do not store monomials with zero coefficients.
The Class Poly

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

Notice:
① We hide the coefficient list via __.
② We do not store monomials with zero coefficients.
Encapsulation
data hiding
polynomials in one variable

Inheritance
base classes and derived classes
points and circles

Polymorphism and Wrapping
builtin functions
wrapping: Karl the Robot

Summary + Assignments

String Representations
overloading \_\_str\_

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

def \_\_str\_(self):
    "returns a polynomial as a string"
    s = ''
    for m in self.__L:
        s += '%+2.f*x^%d' % (m[0],m[1])
    if s == '':
        return '0'
    else:
        return s
```
String Representations

overloading `__str__`

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

def __str__(self):
    "returns a polynomial as a string"
    s = ''
    for m in self.__L:
        s += '%+2.f*x^%d' % (m[0],m[1])
    if s == '':
        return '0'
    else:
        return s
```
Adding Two Polynomials

relies on adding one monomial

def __add__(self, other):
    "adds two polynomials"
    p = Poly()
    for m in self.__L:
        p.addmon(m[0], m[1])
    for m in other.__L:
        p.addmon(m[0], m[1])
    return p

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

Avoid storing zero coefficients!
Adding Two Polynomials
relies on adding one monomial

```python
def __add__(self, other):
    "adds two polynomials"
    p = Poly()
    for m in self.__L:
        p.addmon(m[0], m[1])
    for m in other.__L:
        p.addmon(m[0], m[1])
    return p

>>> 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:

```python
def addmon(self, c, p):
    "adds a monomial"
    if c != 0:
        done = False
        for i in range(0, len(self.__L)):
            m = self.__L[i]
            if m[1] == p:
                nc = c + m[0]
                del(self.__L[i])
                if nc != 0:
                    self.__L.append((nc, p))
                done = True
                break
        if not done:
            self.__L.append((c, p))
```

Encapsulation
data hiding
polynomials in one variable

Inheritance
base classes and derived classes
points and circles

Polymorphism and Wrapping
builtin functions
wrapping: Karl the Robot

Summary + Assignments
Adding a monomial to a polynomial:

```python
def addmon(self, c, p):
    "adds a monomial"
    if c != 0:
        done = False
        for i in range(0, len(self.__L)):
            m = self.__L[i]
            if m[1] == p:
                nc = c + m[0]
                del(self.__L[i])
                if nc != 0:
                    self.__L.append((nc, p))
                    done = True
                break
        if not done:
            self.__L.append((c, p))
```
Adding a Monomial
avoiding to store zero

Adding a monomial to a polynomial:

```python
def addmon(self, c, p):
    "adds a monomial"
    if c != 0:
        done = False
        for i in range(0, len(self.__L)):
            m = self.__L[i]
            if m[1] == p:
                nc = c + m[0]
                del(self.__L[i])
                if nc != 0:
                    self.__L.append((nc, p))
                    done = True
                    break
        if not done:
            self.__L.append((c, p))
```
Adding a monomial to a polynomial:

```python
def addmon(self, c, p):
    "adds a monomial"
    if c != 0:
        done = False
        for i in range(0, len(self.__L)):
            m = self.__L[i]
            if m[1] == p:
                nc = c + m[0]
                del(self.__L[i])
                if nc != 0:
                    self.__L.append((nc, p))
                    done = True
                    break
        if not done:
            self.__L.append((c, p))
```
Adding a Monomial
avoiding to store zero

Adding a monomial to a polynomial:

```python
def addmon(self, c, p):
    "adds a monomial"
    if c != 0:
        done = False
        for i in range(0, len(self.__L)):
            m = self.__L[i]
            if m[1] == p:
                nc = c + m[0]
                del(self.__L[i])
                if nc != 0:
                    self.__L.append((nc, p))
                done = True
                break
        if not done:
            self.__L.append((c, p))
```

Adding a monomial to a polynomial:

```python
def addmon(self, c, p):
    "adds a monomial"
    if c != 0:
        done = False
        for i in range(0, len(self.__L)):
            m = self.__L[i]
            if m[1] == p:
                nc = c + m[0]
                del(self.__L[i])
                if nc != 0:
                    self.__L.append((nc, p))
                done = True
                break
        if not done:
            self.__L.append((c, p))
```
encapsulation and inheritance
polymorphism and wrapping

1. Encapsulation
   data hiding
   polynomials in one variable

2. Inheritance
   base classes and derived classes
   points and circles

3. Polymorphism and Wrapping
   builtin functions
   wrapping: Karl the Robot

4. Summary + Assignments
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.
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.
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.
encapsulation and inheritance
polymorphism and wrapping

1 Encapsulation
data hiding
polynomials in one variable

2 Inheritance
base classes and derived classes
points and circles

3 Polymorphism and Wrapping
builtin functions
wrapping: Karl the Robot

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

```python
class Point:
    "defines a point in the plane"

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

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

The string representation uses scientific notation for the coordinates, with 4 digits after the decimal point.
The definition of the class `Point` is placed in a separate file `classpoint.py`, available to import as a module.

```python
class Point:
    "defines a point in the plane"

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

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

The string representation uses scientific notation for the coordinates, with 4 digits after the decimal point.
The definition of the class Point is placed in a separate file `classpoint.py`, available to import as a module.

```python
class Point:
    "defines a point in the plane"

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

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

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 *
class Circle(Point):
    "defines a circle derived from point"
    def __init__(self,a=0,b=0,r=0):
        "the center is (a,b), radius = r"
        Point.__init__(self,a,b)
        self.r = r
    def area(self):
        "returns the area of the circle"
        from math import pi
        return pi*self.r**2
    def __str__(self):
        "returns string representation of a circle"
        s = 'center : ' + Point.__str__(self) + '
        s += 'radius : ' + '%.4e' % self.r
        return s
from classpoint import *
class Circle(Point):
    "defines a circle derived from point"
    def __init__(self,a=0,b=0,r=0):
        "the center is (a,b), radius = r"
        Point.__init__(self,a,b)
        self.r = r

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

    def __str__(self):
        "returns string representation of a circle"
        s = 'center : ' + Point.__str__(self) + '
        s += 'radius : ' + '%.4e' % self.r
        return s
The Class Circle
in the file classcircle.py

from classpoint import *
class Circle(Point):
    "defines a circle derived from point"
    def __init__(self, a=0, b=0, r=0):
        "the center is (a,b), radius = r"
        Point.__init__(self, a, b)
        self.r = r

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

    def __str__(self):
        "returns string representation of a circle"
        s = 'center : ' + Point.__str__(self) + '
        s += 'radius : ' + '%.4e' % self.r
        return s
from classpoint import *
class Circle(Point):
    "defines a circle derived from point"
    def __init__(self,a=0,b=0,r=0):
        "the center is (a,b), radius = r"
        Point.__init__(self,a,b)
        self.r = r
    def area(self):
        "returns the area of the circle"
        from math import pi
        return pi*self.r**2
    def __str__(self):
        "returns string representation of a circle"
        s = 'center : ' + Point.__str__(self) + '
        s += 'radius : ' + '%.4e' % self.r
        return s
Using the Classes

file pointcircle.py

# L-26 MCS 260 Fri 24 Oct 2008 : pointcircle

# This Python program uses the functionality
# of the classes circle and point.
# Note that Point is imported indirectly
# via the import of the class Circle.

from classcircle import *
print 'using classes point and circle'
x = input('give x : ')
y = input('give y : ')
p = Point(x,y)
print 'the point ' + str(p)
print 'has coordinates ', p.x, p.y
r = input('give r : ')
c = Circle(x,y,r)
print 'the circle :
' + str(c)
print 'has area ', c.area()
# L-26 MCS 260 Fri 24 Oct 2008 : pointcircle

# This Python program uses the functionality
# of the classes circle and point.
# Note that Point is imported indirectly
# via the import of the class Circle.

from classcircle import *
print 'using classes point and circle'
x = input('give x : ')
y = input('give y : ')
p = Point(x,y)
print 'the point ' + str(p)
print 'has coordinates ', p.x, p.y
r = input('give r : ')
c = Circle(x,y,r)
print 'the circle :
' + str(c)
print 'has area ', c.area()
# L-26 MCS 260 Fri 24 Oct 2008 : pointcircle

# This Python program uses the functionality
# of the classes circle and point.
# Note that Point is imported indirectly
# via the import of the class Circle.

from classcircle import *
print 'using classes point and circle'
x = input('give x : ')
y = input('give y : ')
p = Point(x,y)
print 'the point ' + str(p)
print 'has coordinates ', p.x, p.y
r = input('give r : ')
c = Circle(x,y,r)
print 'the circle :
' + str(c)
print 'has area ', c.area()
the program 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
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
Recall Python’s dynamic typing: 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.
Recall Python’s dynamic typing:

→ 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*. 
Polymorphism

method overriding

Recall Python’s dynamic typing:
→ 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`. 
Recall Python’s dynamic typing: 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*. 
Recall Python’s dynamic typing:

→ 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 Encapsulation
data hiding
polynomials in one variable

2 Inheritance
base classes and derived classes
points and circles

3 Polymorphism and Wrapping
builtin functions
wrapping: Karl the Robot

4 Summary + Assignments
Built-in Functions

to check types and relationships

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

```python
>>> 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:

```python
>>> from classcircle import *
>>> issubclass(Point, Circle)
False
>>> issubclass(Circle, Point)
True
```
We can check whether a variable is of a certain type:

```python
>>> 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:

```python
>>> from classcircle import *
>>> issubclass(Point,Circle)
False
>>> issubclass(Circle,Point)
True
```
Built-in Functions on Attributes

With `hasattr` we ask whether an attribute exists:

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

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

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

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

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

To delete an attribute:

```
>>> delattr(p,'x')
>>> hasattr(p,'x')
False
```
Built-in Functions on Attributes

With `hasattr` we ask whether an attribute exists:

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

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

```python
>>> getattr(p,'x')
2
```

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

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

To delete an attribute:

```python
>>> delattr(p,'x')
>>> hasattr(p,'x')
False
```
Built-in Functions on Attributes

With `hasattr` we ask whether an attribute exists:

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

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

```python
>>> getattr(p,'x')
2
```

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

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

To delete an attribute:

```python
>>> delattr(p,'x')
>>> hasattr(p,'x')
False
```
Built-in Functions on Attributes

With `hasattr` we ask whether an attribute exists:

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

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

```python
>>> getattr(p,'x')
2
```

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

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

To delete an attribute:

```python
>>> delattr(p,'x')
>>> hasattr(p,'x')
False
```
encapsulation and inheritance
polymorphism and wrapping

1 Encapsulation
data hiding
polynomials in one variable

2 Inheritance
base classes and derived classes
points and circles

3 Polymorphism and Wrapping
builtin functions
wrapping: Karl the Robot

4 Summary + Assignments
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. $(x, y)$ are the coordinates of the current position

2. $(vx, vy)$ is the orientation vector

3. $d$ is the step size (or size of the tiles)

With turtle graphics we draw the path of 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. \((x, y)\) are the coordinates of the current position
2. \((vx, vy)\) is the orientation vector
3. \(d\) is the step size (or size of the tiles)

With turtle graphics we draw the path of the robot.
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. $(x, y)$ are the coordinates of the current position
2. $(vx, vy)$ is the orientation vector
3. $d$ is the step size (or size of the tiles)

With turtle graphics we draw the path of 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. $(x, y)$ are the coordinates of the current position
2. $(vx, vy)$ is the orientation vector
3. $d$ is the step size (or size of the tiles)

With turtle graphics we draw the path of the robot.
Encapsulation
- data hiding
- polynomials in one variable

Inheritance
- base classes and derived classes
- points and circles

Polymorphism and Wrapping
- builtin functions
- wrapping: Karl the Robot

Summary + Assignments
The Class Robot

```python
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 Class Robot

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.
                """
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.x = a  # x-coordinate of position
    self.y = b  # y-coordinate of position
    turtle.goto(a, b)
    self.h = d  # default step size is 10 pixels
    self.vx = 0  # direction for x-coordinate
    self.vy = 1  # direction for y-coordinate
    turtle.down()
the constructor

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.x = a  # x-coordinate of position
    self.y = b  # y-coordinate of position
    turtle.goto(a, b)
    self.h = d  # default step size is 10 pixels
    self.vx = 0  # direction for x-coordinate
    self.vy = 1  # direction for y-coordinate
    turtle.down()
Turning and Stepping

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

\[
\begin{bmatrix}
    v_x \\
    v_y
\end{bmatrix}
= \begin{bmatrix}
    0 & 1 \\
    -1 & 0
\end{bmatrix}
\begin{bmatrix}
    v_x \\
    v_y
\end{bmatrix}
= \begin{bmatrix}
    v_y \\
    -v_x
\end{bmatrix}
\]

```python
def turn(self):
    ""
    Makes one left turn of 90 degrees.
    ""
    (self.vx, self.vy) = (self.vy, -self.vx)
```

```python
def step(self):
    ""
    Does one step in the current direction.
    ""
    self.x = self.x + self.h*self.vx
    self.y = self.y + self.h*self.vy
    turtle.goto(self.x, self.y)
```
Turning and Stepping

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

\[
\begin{bmatrix}
    v_x \\
    v_y
\end{bmatrix} \rightarrow
\begin{bmatrix}
    0 & 1 \\
    -1 & 0
\end{bmatrix}
\begin{bmatrix}
    v_x \\
    v_y
\end{bmatrix} =
\begin{bmatrix}
    v_y \\
    -v_x
\end{bmatrix}
\]

```python
def turn(self):
    """
    Makes one left turn of 90 degrees.
    """
    (self.vx, self.vy) = (self.vy, -self.vx)

def step(self):
    """
    Does one step in the current direction.
    """
    self.x = self.x + self.h*self.vx
    self.y = self.y + self.h*self.vy
    turtle.goto(self.x, self.y)
```
def main():
    """
    Lets the robot draw a spiral path.
    """
    karl = Robot()
    karl.step()
    n = 1  # number of steps
    for k in range(0, 49):
        karl.turn()
        for i in range(0, n): karl.step()
        if k % 2 == 0: n = n + 1
    ans = raw_input('ok to close ? (y/n) ')
def main():
    
    
    """
    Lets the robot draw a spiral path.
    """
    karl = Robot()
    karl.step()
    n = 1  # number of steps
    for k in range(0, 49):
        karl.turn()
        for i in range(0, n):
            karl.step()
        if k % 2 == 0:
            n = n + 1
    ans = raw_input(’ok to close ? (y/n) ’)
the main program

def main():
    
    """
    Lets the robot draw a spiral path.
    """
    karl = Robot()
    karl.step()

    n = 1  # number of steps
    for k in range(0, 49):
        karl.turn()
        for i in range(0, n):
            karl.step()
        if k % 2 == 0:
            n = n + 1
    ans = raw_input('ok to close ? (y/n) ')
Summary + Assignments

Read chapters 9 & 10 in *Python Power!*

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 $\pi/2$.

Homework collected on Monday 3 November:
# 2, 5 of L-21; # 3, 4 of L-22; # 1 of L-23.