21. Designing & Using Classes

Topics:

Methods
getters and setters
class invariants
More on assert and isinstance
Sorting w.r.t. an Attribute
Class Variables
Methods

Methods are functions that are defined inside a class definition.

We have experience using them with strings

\[
\text{s.upper()}, \text{s.find(s1)}, \text{s.count(s2)}, \\
\text{s.append(s2)}, \text{s.split(c)}, \text{etc}
\]

and lists

\[
\text{L.append(x)}, \text{L.extend(x)}, \text{L.sort()}, \text{etc}
\]
Methods

Now we show how to implement them.

We will revisit the Point class that we used earlier, and define methods for computing distance and midpoints.

Anticipate this:

\[
\begin{align*}
delta &= P.\text{Dist}(Q) \\
C &= A.\text{Midpoint}(B)
\end{align*}
\]
class Point:
    
    Attributes:
    
    x: float, the x-coordinate of a point
    y: float, the y-coordinate of a point
    d: float, distance to origin

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.d = sqrt(x**2 + y**2)

The constructor

Assume proper importing from math class
A Simple Method: Dist

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.d = sqrt(x**2 + y**2)

    def Dist(self, other):
        """ Returns the distance from self to P
        Prec: other is a point """
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx**2 + dy**2)
Let’s create two point objects and compute the distance between them. This can be done two ways...

```python
>>> P = Point(3,4)
>>> Q = Point(6,8)
>>> deltaPQ = P.Dist(Q)
>>> deltaQP = Q.Dist(P)
>>> print deltaPQ,deltaQP
5.0 5.0
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.d = sqrt(x**2 + y**2)

    def Midpoint(self, other):
        """ Returns the midpoint of the line segment that connects self to other ""
        PreC: other is a point
        """
        xm = (self.x + other.x) / 2.0
        ym = (self.y + other.y) / 2.0
        return Point(xm, ym)
Using the Midpoint Method

Let's create two point objects and compute the midpoint. This can be done two ways...

```python
>>> P = Point(1,2)
>>> Q = Point(3,4)
>>> MPQ = P.Midpoint(Q)
>>> MQP = Q.Midpoint(P)
>>> print MPQ
( 2.000, 3.000)     distance =  3.606
>>> print MQP
( 2.000, 3.000)     distance =  3.606
```
Recall: \texttt{\_\_str\_\_(self)}

```python
def \_\_str\_\_(self):
    s = '(%6.3f,%6.3f) distance = %6.3f'
    %(self.x,self.y,self.d)
```

With this method in place, we have a handy way of “printing out” an object:

```python
>>> P = Point(3,4)
>>> print P
( 3.0000, 4.0000) distance =  5.000
```
Method Implementation: Syntax Concerns

class Point:
    
    def Dist(self, other):
        """ Returns the distance from
        self to other
        PreC: other is a point
        """
        
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx**2 + dy**2)

Note indentation.
A class method is part of the class definition.
class Point:
    def Dist(self, other):
        """ Returns the distance from self to P
        PreC: P is a point
        """
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx**2 + dy**2)

Note the use of "self". It is always the first argument of a method.
class Point:
    
    def Dist(self, other):
        """ Returns the distance from self to P
        Prec: P is a point
        """
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx**2 + dy**2)

Think like this: “We are going to apply the method dist to a pair of Point objects, self and other.”
def Dist(self, other):
    dx = self.x - other.x
    dy = self.y - other.y
    D = sqrt(dx**2 + dy**2)
    return D

>>> P = Point(3, 4)
>>> Q = Point(6, 8)
>>> P.Dist(Q)
5.0

def Dist(P, Q):
    dx = P.x - Q.x
    dy = P.y - Q.y
    D = sqrt(dx**2 + dy**2)
    return D

>>> P = Point(3, 4)
>>> Q = Point(6, 8)
>>> Dist(Q, P)
5.0
Visualizing a Method Call Using State Diagrams

Let’s see what happens when we execute the following:

\[ P = \text{Point}(3,4) \]
\[ Q = \text{Point}(6,8) \]
\[ D = P.\text{Dist}(Q) \]
Visualizing a Method Call

- $P = \text{Point}(3,4)$
- $Q = \text{Point}(6,8)$
- $D = P.\text{Dist}(Q)$
Visualizing a Method Call

\[ P = \text{Point}(3, 4) \]
\[ Q = \text{Point}(6, 8) \]
\[ D = P.\text{Dist}(Q) \]
Visualizing a Method Call

\[ P = \text{Point}(3,4) \]
\[ Q = \text{Point}(6,8) \]
\[ D = P.\text{Dist}(Q) \]

\[ dx = \text{self}.x - \text{other}.x \]
\[ dy = \text{self}.y - \text{other}.y \]
\[ z = \sqrt{(dx^2 + dy^2)} \]
return \( z \)
class Point:
    
    def Dist(self, other):
        """ Returns the distance from self to P 
        PreC: other is a point
        """
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx**2 + dy**2)

Think of self and other as input parameters.
P = Point(3, 4)
Q = Point(6, 8)

D = P.Dist(Q)

dx = self.x - other.x
dy = self.y - other.y
z = sqrt(dx**2 + dy**2)
return z

Control passes to the method Dist
Visualizing a Method Call

P = Point(3,4)
Q = Point(6,8)
D = P.Dist(Q)

dx = self.x - other.x
dy = self.y - other.y
z = sqrt(dx**2 + dy**2)
return z
Visualizing a Method Call

\[
\begin{align*}
P &= \text{Point}(3,4) \\
Q &= \text{Point}(6,8) \\
D &= P.\text{Dist}(Q)
\end{align*}
\]

\[
\begin{align*}
dx &= \text{self}.x - \text{other}.x \\
\text{dy} &= \text{self}.y - \text{other}.y \\
\text{z} &= \sqrt{(dx^2 + dy^2)} \\
\text{return } \text{z}
\end{align*}
\]
Visualizing a Method Call

\[ P = \text{Point}(3,4) \]
\[ Q = \text{Point}(6,8) \]
\[ D = P.\text{Dist}(Q) \]

\[ dx = \text{self}.x - \text{other}.x \]
\[ dy = \text{self}.y - \text{other}.y \]
\[ z = \sqrt{dx^2 + dy^2} \]
\[ \text{return } z \]
Visualizing a Method Call

\[ P = \text{Point}(3,4) \]
\[ Q = \text{Point}(6,8) \]
\[ D = P.\text{Dist}(Q) \]
Let’s Turn Our Attention to Some Software Engineering Issues that Relate to Classes
Motivation

This becomes increasingly important as the problems get bigger and multiple software developers are on the scene.

At the CS 1110 level, we begin to practice these habits and motivate their relevance.
Setter and Getter Methods

Motivation:
Changing the attributes of an object by “freely” using the dot-notation is dangerous and short sighted.

```python
>>> P = Point(3,4)
>>> P.x = 0
>>> print P
( 0.000, 4.000)       distance =  5.000
```

The “class invariant” that $\sqrt{P.x^2 + P.y^2} = P.d$ is broken.
Getter Methods

Access attributes through getter methods.

def get_x(self):
    return self.x

def get_y(self):
    return self.y

def get_d(self):
    return self.d

Typically name these simple methods in this style.

```python
>>> P = Point(3,4)
>>> a = P.get_x()
>>> b = P.get_y()
>>> c = P.get_d()
>>> print a,b,c
3.0 4.0 5.0
```
Getter Methods—Why?

Access attributes through getter methods.

def get_x(self):
    return self.x

def get_y(self):
    return self.y

def get_d(self):
    return self.d

>>> P = Point(3,4)
>>> a = P.get_x()
>>> b = P.get_y()
>>> c = P.get_d()
>>> print a,b,c
3.0 4.0 5.0

You don’t want the user to “see” and work with attributes.
Setter Methods

```python
def set_x(self, x):
    self.x = x
    self.d = sqrt(self.x**2 + self.y**2)

def set_y(self, y):
    self.y = y
    self.d = sqrt(self.x**2 + self.y**2)

>>> P = Point(3, 4)
>>> P.set_x(0)
>>> print P
(0.000, 4.000) distance = 4.000
```
Setter Methods—Why?

Good:

```python
>>> P = Point(3,4)
>>> P.set_x(0)
>>> print P
( 0.000, 4.000)     distance =  4.000
```

Automatically maintains the required connection among the x, y, and d attributes

Bad:

```python
>>> P = Point(3,4)
>>> P.x = 0
>>> P.d = sqrt(P.x**2+P.y**2)
>>> print P
( 0.000, 4.000)     distance =  4.000
```

Requires programmer attentiveness. Don’t forget to update P.d!
Setter Methods Justification—A Tale of Two Software Engineers

Bob and Sue each develop a Point class with this constructor:

```python
def __init__(self, x, y):
    self.x = x
    self.y = y
    self.d = sqrt(x**2+y**2)
```

Sue uses setter methods. Bob does not.
Setter Methods Justification - A Tale of Two Software Engineers

Bob is very successful. Tons of python code is written that uses his stuff. Millions of references like this are out there:

\[
P.x = \text{blahblah}
\]
\[
P.d = \sqrt{P.x^2 + P.y^2}
\]

But then...
One day Bob’s boss says “we have a new definition of distance. Instead of

$$\sqrt{x^2+y^2}$$

we now have to use

$$\text{abs}(x) + \text{abs}(y)$$

Bob must direct customers to change those millions of P.d updates to reflect the new definition of distance.
One the other hand, to maintain Sue’s software, the customers just have change one line of code in the constructor:

```python
def __init__(self, x, y):
    self.x = x
    self.y = y
    self.d = abs(x) + abs(y)
```
Before...

def set_x(self,x):
    self.x = x
    self.d = sqrt(self.x**2+self.y**2)

def set_y(self,y):
    self.y = y
    self.d = sqrt(self.x**2+self.y**2)
def set_x(self, x):
    self.x = x
    self.d = abs(self.x) + abs(self.y)

def set_y(self, y):
    self.y = y
    self.d = abs(self.x) + abs(self.y)
Moral:

Bob is moved to an interior cubical with no window!
Reminder about assert and isinstance
Using Assert in the Class Setting

```python
def __init__(self,x,y):
    Bx = type(x)==float or type(x)==int
    assert Bx, 'x must be a number'

    By = type(y)==float or type(y)==int
    assert By,'y must be a number'

    self.x = x
    self.y = y
    self.d = sqrt(x**2+y**2)
```

The usual check-the-preconditions business
The function `isinstance` can be used to check for user-defined types.

```python
def Midpoint(self, P):
    B = isinstance(P, Point)
    assert B, 'P must be a Point'

    xm = (self.x + P.x) / 2.0
    ym = (self.y + P.y) / 2.0
    return Point(xm, ym)
```
Sorting Lists of Objects
A Sorting Problem

Suppose we have a list of Points, i.e., a list of references to Point objects.

Let’s sort the list based on distance from origin.

It involves writing a getter function.
Before

\[ L: \]

\[
\begin{array}{c|c}
\text{Point} & \text{x} \\
\hline
3 & 3 \\
4 & 4 \\
5 & 5 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{Point} & \text{x} \\
\hline
1 & 1 \\
0 & 0 \\
1 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{Point} & \text{x} \\
\hline
2 & 2 \\
1 & 1 \\
2.3 & 2.3 \\
\end{array}
\]
<table>
<thead>
<tr>
<th></th>
<th>Point</th>
<th></th>
<th>Point</th>
<th></th>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
<td>2</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>y</td>
<td>1</td>
<td>y</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>d</td>
<td>2.3</td>
<td>d</td>
<td>5</td>
</tr>
</tbody>
</table>
How to Do It

Write a “getter” function that takes a point and returns the value of its d attribute:

```python
def getD(P):
    return P.d
```

Now use the sort method as follows

```python
L.sort(key = getD)
```

This will permute the references in L so that they refer to point objects in the required order, i.e., in order of distance from origin.
A New Example to Illustrate the Notion of a Class Variable
Class Variables

Class variables are shared among all instances of the class.

We illustrate with an example.

Then we will formally distinguish between class variables and instance variables.
The Class SimpleDate

We define a class that can be used to carry out certain computations with dates. For example:

1. Cornell was founded on 4/27/1865. Today is 4/14/2015. How many days has Cornell been around?

2. What’s the date 1000 days from now?
1. A “date string” looks like this: ‘4/14/2015’.

2. Assume the availability of

```python
def isLeapYear(y):
    """ Returns True if y is a leap year. Otherwise returns False """
```

- y is not a century year and is divisible by 4
- or
- y is a century year and is divisible by 400.
Four Attributes

m: int, index of month

d: int, the day

y: int, the year

s: str, a date string

Creating a SimpleDate Object:

D = SimpleDate('4/14/2015')
Visualizing a SimpleDate

```python
>>> D = SimpleDate('4/14/2015')
```

![Diagram showing a SimpleDate object with components m=4, d=14, y=2015, and s='4/14/2015']
Methods in SimpleDate

`__str__(self)`
pretty prints the date encoded in self

`Tomorrow(self)`
returns a SimpleDate object that encodes the day after self

`dateIndex(self)`
returns number of days from 1/1/1600 to the date encoded in self

`FutureDate(self, n)`
returns the SimpleDate encoding of the date that is n days after self
The Method Tomorrow

```python
>>> D = SimpleDate('4/14/2015')
>>> T = D.Tomorrow()
>>> print T
April 15, 2014
```

Pretty printing via `__str__`
Useful Class Variables

These variables house handy data:

```
TheMonths = ['', 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December']
nDays = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, 31]
```

Methods can access this data via self and the dot notation, e.g.,
```
self.TheMonths[self.m]
```
Visualizing the Overall Class

class SimpleDate:

    TheMonths = blah
    nDays = blah

    def blah blah
    def blah blah
    def blah blah
Referencing a Class Variable

def Tomorrow(self):
    m = self.m
    d = self.d
    y = self.y
    Last = self.nDays[m]
    if isLeapYear(y) and m==2:
        Last+=1
    
Creating andPrinting a
SimpleDate Object

```python
>>> Today = SimpleDate('4/14/2015')
>>> print Today
April 14, 2015
>>> T = Today.Tomorrow()
>>> print T
April 15, 2015
```
The `isequal` Method

```python
def isequal(self, other):
    B1 = self.m == other.m
    B2 = self.d == other.d
    B3 = self.y == other.y
    return B1 and B2 and B3
```

Can be used to check if two `SimpleDate` objects represent the same date.
Method `dateIndex`

```python
def dateIndex(self):
    idx = 1
    Day = SimpleDate('1/1/1600')
    while not Day.isequal(self):
        idx+=1
        Day = Day.Tomorrow()
    return idx
```

1 = Jan 1, 1600. Count forward from this baseline
How Old is Cornell in Days?

```python
>>> Today = SimpleDate('4/14/2015')
>>> nToday = Today.dateIndex()
>>> Founding = SimpleDate('4/27/1865')
>>> nFounding = Founding.dateIndex()
>>> CornellDays = nToday - nFounding
>>> print CornellDays
54773
```