Lecture 15, 25 September 2025

Class `Point` using (x, y) coordinates

In [1]:
```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b

  def translate(self,deltax,deltay):
    self.x += deltax
    self.y += deltay

  def odistance(self):
    import math
    d = math.sqrt(self.x*self.x +
                  self.y*self.y)
    return(d)

  def __str__(self):
    return('('+str(self.x)+','
           +str(self.y)+')')

  def __add__(self,p):
    return(Point(self.x + p.x,
                 self.y + p.y))

  def __lt__(self,p):
    return(self.x < p.x and self.y < p.y)

  def __le__(self,p):
    return(self.x <= p.x and self.y <= p.y)

  def __eq__(self,p):
    return(self.x == p.x and self.y == p.y)

  def __ne__(self,p):
    return(not(self == p))

  def __gt__(self,p):
    return(self.x > p.x and self.y > p.y)

  def __ge__(self,p):
    return(self.x >= p.x and self.y >= p.y)
```

In [2]:
```
p = Point(3,4)
q = Point(7,10)
r = Point(3,4)
s = Point(7,11)
```

In [3]:
```
p < q, q < p, p == q, p == r, p <= p, q <= s, s >= q, q > p
```

Out[3]:
```
(True, False, False, True, True, True, True, True)
```

Changing the implementation

- Change the definition of Point to use polar representation, (r, θ)

In [4]:
```
import math
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      if b >= 0:
        self.theta = math.pi/2
      else:
        self.theta = 3*math.pi/2
    else:
      self.theta = math.atan(b/a)

  def translate(self,deltax,deltay):
    x = self.r*math.cos(self.theta)
    y = self.r*math.sin(self.theta)
    x += deltax
    y += deltay
    self.r = math.sqrt(x*x + y*y)
    if x == 0:
      if y >= 0:
        self.theta = math.pi/2
      else:
        self.theta = 3*math.pi/2
    else:
      self.theta = math.atan(y/x)

  def odistance(self):
    return(self.r)

  def __str__(self):
    x = self.r*math.cos(self.theta)
    y = self.r*math.sin(self.theta)
    return('('+str(x)+','+str(y)+')')

  def __add__(self,p):
    sx = self.r*math.cos(self.theta)
    sy = self.r*math.sin(self.theta)
    px = p.r*math.cos(p.theta)
    py = p.r*math.sin(p.theta)
    return(Point(sx + px,sy+py))

  def __lt__(self,p):
    sx = self.r*math.cos(self.theta)
    sy = self.r*math.sin(self.theta)
    px = p.r*math.cos(p.theta)
    py = p.r*math.sin(p.theta)
    return(sx < px and sy < py)
```

- The interface still assumes (x, y) representation
- When constructing a point, convert (x, y) to (r, θ)
  - Be careful about the case where $x = 0$

- To translate a point, convert (r, θ) back to (x, y), translate, then convert back to (r, θ)
- Similar conversion for `__str__()`, `__add__()`, `__lt__()`

Repeat the examples above

- Observe that nothing changes for the user of the class

In [5]:
```
p = Point(3,4)
q = Point(7,10)
```

In [6]:
```
p.odistance(), q.odistance()
```

Out[6]:
```
(5.0, 12.206555615733702)
```

In [7]:
```
p.translate(3,4)
p.odistance()
```

Out[7]:
```
10.0
```

In [8]:
```
print(p) # Note some lack of precision going from (x,y) to (r,theta) and
```

```
(6.000000000000001,7.999999999999999)
```

In [9]:
```
str(p)
```

Out[9]:
```
'(6.000000000000001,7.999999999999999)'
```

In [10]:
```
print(p+q)
```

```
(13.000000000000002,18.0)
```

In [11]:
```
print(p,q)
```

```
(6.000000000000001,7.999999999999999) (6.999999999999999,10.0)
```

In [12]:
```
p < q, q < p
```

Out[12]:
```
(True, False)
```

A note about variables inside classes

- Without the prefix `self`, variables are internal to a function
- Variables with prefix `self` persist within the object

In [13]:
```
class Experiment:
    def __init__(self,a):
        x = a

    def __str__(self):
        return(str(x))
```

In [14]:
```
z = Experiment(5)
```

In [15]:
```
str(z)
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[15], line 1
----> 1 str(z)

Cell In[13], line 6, in Experiment.__str__(self)
      5 def __str__(self):
----> 6     return(str(x))

NameError: name 'x' is not defined
```

In [16]:
```
class Experiment2:
    def __init__(self,a):
        self.x = a

    def __str__(self):
        return(str(self.x))
```

In [17]:
```
y = Experiment2(7)
str(y)
```

Out[17]:
```
'7'
```

- The name `self` for the current object (first parameter) is only a convention
- Can use any other name

In [18]:
```
class Experiment3:
    def __init__(self,a):
        self.x = a

    def __str__(this):
        return(str(this.x))
```

In [19]:
```
x = Experiment3(17)
print(x)
```

```
17
```

- Classes and objects were grafted onto Python as an afterthought
- If we have a class `C`, and object `O` and a function `f(self,x,y,z)` inside `C`, we can replace `o.f(a,b,c)` by `C.f(o,a,b,c)`.
- In other words, `self` is actually a reference to the object on which `f` is being invoked.

In [20]:
```
pnew = Point(5,7)
pnew.odistance()
```

Out[20]:
```
8.602325267042627
```

In [21]:
```
Point.translate(pnew,4,7)
```

In [22]:
```
print(pnew)
```

```
(9.0,14.0)
```

- This is also true for built in datatypes like `list`

In [23]:
```
l = [1,2,3]
l.append(4)
```

In [24]:
```
l
```

Out[24]:
```
[1, 2, 3, 4]
```

In [25]:
```
list.append(l,5)
```

In [26]:
```
l
```

Out[26]:
```
[1, 2, 3, 4, 5]
```

- Python also has no mechanism to ensure privacy of implementation
- We cannot prevent code outside the class from accessing internal fields `p.x` and `p.y` for a `Point` `p`
- In fact, we can even add new internal fields!

In [27]:
```
pnew.z = 7
```

Linked lists

- An implementation using classes and objects; compare with our earlier implementation using nested dictionaries
- An empty list has a single node with `value` and `next` both `None`
- Last node in the list has `next` set to `None`

In [28]:
```
class List:
    def __init__(self):
        self.value = None
        self.next = None
        return

    def isempty(self):
        return(self.value == None)

    def append(self,v):   # append, iterative
        if self.isempty():
            self.value = v
            return

        temp = self
        while temp.next != None:
            temp = temp.next

        temp.next = List()
        temp.next.value = v
        return
```

```
    def insert(self,v):
        if self.isempty():
            self.value = v
            return

        newnode = List()
        newnode.value = v

        # Exchange values in self and newnode
        (self.value, newnode.value) = (newnode.value, self.value)

        # Switch links
        (self.next, newnode.next) = (newnode, self.next)

        return


    def __str__(self):
        # Iteratively create a Python list from linked list
        # and convert that to a string
        selflist = []
        if self.isempty():
            return(str(selflist))

        temp = self
        selflist.append(temp.value)

        while temp.next != None:
          temp = temp.next
          selflist.append(temp.value)

        return(str(selflist))
```

In [29]:
```
l = List()
l.append(5)
print(l)
```

```
[5]
```

In [30]:
```
l.append(7)
print(l)
```

```
[5, 7]
```

In [31]:
```
l.append(9)
print(l)
```

```
[5, 7, 9]
```

In [32]:
```
l.insert(4)
print(l)
```

```
[4, 5, 7, 9]
```

- Change the constructor
- Can create a non-empty list to start with
- Default is to create an empty list

In [33]:
```
class List:
    def __init__(self,initlist = []):
        self.value = None
        self.next = None
        for x in initlist:
            self.append(x)
        return

    def isempty(self):
        return(self.value == None)

    def append(self,v):   # append, iterative
        if self.isempty():
            self.value = v
            return

        temp = self
        while temp.next != None:
            temp = temp.next

        temp.next = List()
        temp.next.value = v
        return

    def insert(self,v):
        if self.isempty():
            self.value = v
            return

        newnode = List()
        newnode.value = v

        # Exchange values in self and newnode
        (self.value, newnode.value) = (newnode.value, self.value)

        # Switch links
        (self.next, newnode.next) = (newnode, self.next)

        return


    def __str__(self):
        # Iteratively create a Python list from linked list
        # and convert that to a string
        selflist = []
        if self.isempty():
            return(str(selflist))

        temp = self
        selflist.append(temp.value)

        while temp.next != None:
          temp = temp.next
          selflist.append(temp.value)

        return(str(selflist))
```

Some performance measurements

In [34]:
```
l = List([11,12,13])
print(l)
```

```
[11, 12, 13]
```

In [35]:
```
l.append(14)
print(l)
```

```
[11, 12, 13, 14]
```

In [36]:
```
l.insert(10)
print(l)
```

```
[10, 11, 12, 13, 14]
```

In [37]:
```
import time
```

- Insert items at the start of a linked list, multiples of 10^5, linear blowup

In [38]:
```
for i in range(1,5):
    l1 = List()
    start = time.perf_counter()
    for j in range(i*100000):
        l1.insert(j)
    elapsed = time.perf_counter() - start
    print(i*100000,elapsed)
```

```
100000 0.03644516799977282
200000 0.08824444399942877
300000 0.12094478699873434
400000 0.1904817329996149
```

- Insert items at the start of a Python list, multiples of 5×10^4, quadratic blowup

In [39]:
```
for i in range(1,5):
    l2 = []
    start = time.perf_counter()
    for j in range(i*50000):
        l2.insert(0,j)
    elapsed = time.perf_counter() - start
    print(i*50000,elapsed)
```

```
50000 0.17767955300223548
100000 0.7180923010018887
150000 1.6151041690027341
200000 2.9811208480023197
```

- Append items at the end of a linked list, multiples of 10^4, quadratic blowup

In [40]:
```
for i in range(1,5):
    l1 = List()
    start = time.perf_counter()
    for j in range(i*10000):
        l1.append(j)
    elapsed = time.perf_counter() - start
    print(i*100000,elapsed)
```

```
100000 1.0824261469970224
200000 4.68311660499603
300000 10.58243091499753
400000 17.945564321998972
```

- Append items at the end of a Python list, multiples of 10^6, linear blowup

In [41]:
```
for i in range(1,5):
    l2 = []
    start = time.perf_counter()
    for j in range(i*1000000):
        l2.append(j)
    elapsed = time.perf_counter() - start
    print(i*50000,elapsed)
```

```
50000 0.042161138997471426
100000 0.08379059300204972
150000 0.12394860199856339
200000 0.15941382599703502
```