# Lecture 11, 13 September 2022

## Defining our own data structures

- In Lecture 8, we implemented a "linked" list using dictionaries
- The fundamental functions like `listappend`, `listinsert`, `listdelete` modify the underlying list
- Instead of `mylist = {}`, we wrote `mylist = createlist()`
- To check empty list, use a function `isempty()` rather than `mylist == {}`
- Can we clearly separate the **interface** from the **implementation**
- Define the data structure in a more "modular" way

## Object oriented approach

- Describe a datatype using a template, called a **class**
- Create independent instances of a class, each is an **object**
- Each object has its own internal *state* -- the values of its local variables
- All objects in a class share the same functions to query/update their state
- `l.append(x)` vs `append(l,x)`
  - Tell an object what to do vs passing an object to a function
- Each object has a way to refer to itself

## Basic definition of class `Point` using $(x, y)$ coordinates

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

  def translate(self,deltax,deltay):
    self.x += deltax  # Same as self.x = self.x deltax
    # In general, if we have a = a op b for any arithmetic operation op, can write a op= b
    # For example: a += 5 is a = a + 5, a -= 10 is a = a - 10 etc
    self.y += deltay

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

Create two points

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

Compute `odistance` for `p` and `q`

In [3]:
```
p.odistance(), q.odistance()
```

Out[3]:
```
(5.0, 12.206555615733702)
```

Translate `p` and check the distance

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

Out[4]:
```
10.0
```

- At this stage, `print()` does not produce anything meaningful
- `+` is not defined yet

In [5]:
```
print(p)
```

```
<__main__.Point object at 0x7f503001b1d0>
```

In [6]:
```
print(p+q)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_11157/751839466.py in <module>
----> 1 print(p+q)

TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

In [7]:
```
p  < q
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_11157/1530764535.py in <module>
----> 1 p  < q

TypeError: '<' not supported between instances of 'Point' and 'Point'
```

### Change the definition of `Point` to use $(r, \theta)$ representation

In [8]:
```
import math
class Point:
  def __init__(self,a,b):
    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)
```

### Repeat the examples above

- Observe that nothing changes for the user of the class

Create two points

In [9]:
```
p = Point(3,4)
q = Point(7,10)
```

Compute `odistance` for `p` and `q`

In [10]:
```
p.odistance(), q.odistance()
```

Out[10]:
```
(5.0, 12.206555615733702)
```

Translate `p` and check the distance

In [11]:
```
p.translate(3,4)
p.odistance()
```

Out[11]:
```
10.0
```

In [12]:
```
print(p)
```

```
<__main__.Point object at 0x7f503008cb90>
```

In [13]:
```
print(p+q)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_11157/751839466.py in <module>
----> 1 print(p+q)

TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

### Return to $(x, y)$ representation, adding `__str__` and `__add__`

In [14]:
```
class Point:
  def __init__(self,a,b):
    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))
  # Previous line is a concise way of saying
  #
  # newx = self.x + p.x
  # newy = self.y + p.y
  # newpt = Point(newx,newy)
  # return(newpt)
```

### Again, run the same examples

In [15]:
```
p = Point(3,4)
q = Point(7,10)
```

Compute `odistance` for `p` and `q`

In [16]:
```
p.odistance(), q.odistance()
```

Out[16]:
```
(5.0, 12.206555615733702)
```

Translate `p` and check the distance

In [17]:
```
p.translate(3,4)
p.odistance()
```

Out[17]:
```
10.0
```

In the following two cells, we see a difference

- Since `__str__` is defined, `print()` gives useful output
- `+` works as expected thanks to the definition for `__add__`

In [18]:
```
print(p)
```

```
(6,8)
```

In [19]:
```
print(p+q)
```

```
(13,18)
```