Classes and objects

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming and Data Structures with Python

Lecture 11, 13 Sep 2022

Classes and objects

- Abstract datatype
  - Stores some information
  - Designated functions to manipulate the information
  - For instance, list: `append()`, `insert()`, `delete()`, ...

Classes and objects

- Abstract datatype
  - Stores some information
  - Designated functions to manipulate the information
  - For instance, list: `append()`, `insert()`, `delete()`, ...
- Separate the (private) implementation from the (public) specification

Classes and objects

- Abstract datatype
  - Stores some information
  - Designated functions to manipulate the information
  - For instance, list: `append()`, `insert()`, `delete()`, ...
- Separate the (private) implementation from the (public) specification
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data

Classes and objects

- Abstract datatype
  - Stores some information
  - Designated functions to manipulate the information
  - For instance, list: `append()`, `insert()`, `delete()`, ...
- Separate the (private) implementation from the (public) specification
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data
- Object
  - Concrete instance of template

Example: 2D points

- A point has coordinates (x, y)
  - `__init__()` initializes internal values `x`, `y`
  - First parameter is always `self`

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

p1 = Point(5,4)

l = List([])

l = []

l = [1,2,3]

Default values

`int("36")` ⇝ 36

`int("ab")` ⇝ error

`int("ab", 16)` ⇝ 160 + 11 = 171

16 × 10 + 11

a b

base is optional — has default value 10

```
def int(s, b=10):
```

If a second arg is provided
then b is assigned that value

otherwise b is 10

Constraint Which arguments are missing?

Optional parameters must come at end

```
def __init__(self, a=0, b=0):
    self.x = a
    self.y = b
```

```
p1 = Point(2,3)
p2 = Point()
p3 = Point(6)
(6,0)
```

Example: 2D points

- A point has coordinates (x, y)
  - `__init__()` initializes internal values `x`, `y`
  - First parameter is always `self`
- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$

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

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

Example: 2D points

- A point has coordinates (x, y)
  - `__init__()` initializes internal values `x`, `y`
  - First parameter is always `self`
- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
- Distance from the origin
  - $d = \sqrt{x^2 + y^2}$

```
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)
```

p.translate(-,-)

p.odistance()

Polar coordinates

- (r, θ) instead of (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$


```
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)
```

Polar coordinates

- (r, θ) instead of (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r

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

Polar coordinates

- (r, θ) instead of (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
  - Convert (r, θ) to (x, y)
  - $x = r\cos\theta$, $y = r\sin\theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$

```
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:
    self.theta = math.pi/2
  else:
    self.theta = math.atan(y/x)
```

Polar coordinates

- (r, θ) instead of (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
  - Convert (r, θ) to (x, y)
  - $x = r\cos\theta$, $y = r\sin\theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$
- Interface has not changed
  - User need not be aware whether representation is (x, y) or (r, θ)

```
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:
    self.theta = math.pi/2
  else:
    self.theta = math.atan(y/x)
```

Special functions

- `__init__()` — constructor

Special functions

- `__init__()` — constructor
- `__str__()` — convert object to string
  - `str(o) == o.__str__()`
  - Implicitly invoked by `print()`

```
class Point:

  ...

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

Special functions

- `__init__()` — constructor
- `__str__()` — convert object to string
  - `str(o) == o.__str()__`
  - Implicitly invoked by `print()`
- `__add__()`
  - Implicitly invoked by `+`

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

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

p4 = p1 + p2

p1.__add__(p2)

p2 + p1

p2.__add__(p1)

x3 = self.x + p.x
y3 = self.y + p.y
p3 = Point(x3, y3)
return(p3)

Special functions

- `__init__()` — constructor
- `__str__()` — convert object to string
  - `str(o) == o.__str()__`
  - Implicitly invoked by `print()`
- `__add__()`
  - Implicitly invoked by `+`
- Similarly
  - `__mult__()` invoked by `*`
  - `__lt__()` invoked by `<`
  - `__ge__()` invoked by `>=`
  - ...

```
class Point:

  ...

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


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

Designing a flexible list

Madhavan Mukund

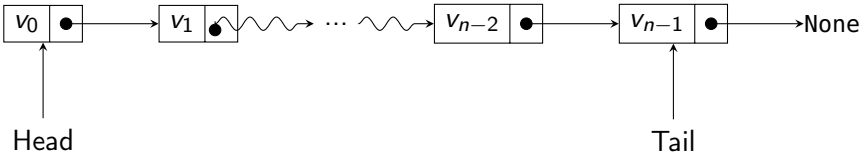
https://www.cmi.ac.in/~madhavan

Programming and Data Structures with Python

Lecture 11, 13 Sep 2022

Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
  - "Linked" list
- Easy to modify
  - Inserting and deletion is easy via local "plumbing"
  - Flexible size
- Need to follow links to access `A[i]`
  - Takes time $O(i)$



Implementing lists in Python

- Python class Node

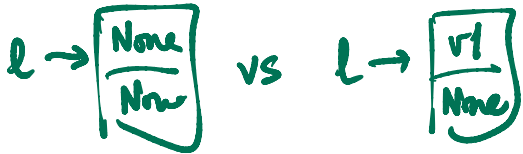
What is an empty list?

```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)
```

default

l = [] ≠ l = None



l.isempty()

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
  - `self.value` is the stored value
  - `self.next` points to next node

```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
  - `self.value` is the stored value
  - `self.next` points to next node
- Empty list?
  - `self.value` is `None`

```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
  - `self.value` is the stored value
  - `self.next` points to next node
- Empty list?
  - `self.value` is `None`
- Creating lists
  - `l1 = Node()` — empty list
  - `l2 = Node(5)` — singleton list


```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
  - `self.value` is the stored value
  - `self.next` points to next node
- Empty list?
  - `self.value` is `None`
- Creating lists
  - `l1 = Node()` — empty list
  - `l2 = Node(5)` — singleton list
  - `l1.isempty() == True`
  - `l2.isempty() == False`

```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)
```

Appending to a list

- Add `v` to the end of list `l`
- If `l` is empty, update `l.value` from `None` to `v`
- If at last value, `l.next` is `None`
  - Point `next` at new node with value `v`
- Otherwise, recursively append to rest of list

```
def append(self,v):
    # append, recursive
    if self.isempty():
        self.value = v
    elif self.next == None:
        self.next = Node(v)
    else:
        self.next.append(v)
    return
```

append(v, [3]) = [v]

l.append(v)

[l[0]] + app —

Appending to a list

- Add `v` to the end of list `l`
- If `l` is empty, update `l.value` from `None` to `v`
- If at last value, `l.next` is `None`
  - Point `next` at new node with value `v`
- Otherwise, recursively append to rest of list
- Iterative implementation
  - If empty, replace `l.value` by `v`
  - Loop through `l.next` to end of list
  - Add `v` at the end of the list

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

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

    temp.next = Node(v)
    return
```