Lecture 12, 24 September 2024

Default values for function parameters

- Can provide a default value for a parameter
- If no argument is passed, default value is used
- Recall `int(s)` converts a string `s` into an `int` if it makes sense

In [1]:
```
int("7")
```

Out[1]: 7

In [2]:
```
int("AB")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[2], line 1
----> 1 int("AB")

ValueError: invalid literal for int() with base 10: 'AB'
```

- The error message above refers to `base 10`
- `int()` has an optional second parameter, which is the base
- If we specify base 16, the previous conversion works

In [3]:
```
int("AB",16)
```

Out[3]: 171

- The way such a function is defined is as follows

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

- If we call this function with only one argument, the default value of `b` is used, so the conversion is in base 10
- If we pass two arguments, the second value is used to set `b`

- Here is another example, to illustrate the point

In [4]:
```
def checkdef(x = 0, y = 0):
    print("x:",x,"y:",y)
```

In [5]:
```
checkdef()
```

```
x: 0 y: 0
```

In [6]:
```
checkdef(7)  # 7 is used for the first parameter x, y gets the default value
```

```
x: 7 y: 0
```

- We can also pass parameters by name, ignoring the order

In [7]:
```
checkdef(y=17,x=12)
```

```
x: 12 y: 17
```

- Using this, we can pass an argument for `y` and use the default `x`

In [8]:
```
checkdef(y=9)
```

```
x: 0 y: 9
```

Defining our own data structures

- Earlier, 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 [9]:
```
class Point:
  def __init__(self,a=0,b=0):
    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
    # No return is same as empty return: return()

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

Create two points

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

Compute `odistance()` for `p` and `q`

In [11]:
```
p.odistance(), q.odistance()
```

Out[11]:
```
(5.0, 12.206555615733702)
```

Translate `p` and check the distance

In [12]:
```
p.translate(3,4)
p.odistance()
```

Out[12]:
```
10.0
```

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

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

```
<__main__.Point object at 0x7f5378ebe5d0>
```

In [14]:
```
print(p+q)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[14], line 1
----> 1 print(p+q)

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

- Use special functions to make these possible
- `print()` requires converting its argument to a string
  - Function `__str__()` specifies how to do this
- `+` implicitly calls `__add__()`
  - `p + q` gets translated as `p.__add__(q)`
  - `q + p` gets translated as `q.__add__(p)`
  - In either case, `__add__()` is executed in the context of one point and the other point is passed to it as an argument
  - We will define `__add__()` so that it returns a new `Point` without modifying its arguments

In [15]:
```
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))
  # Previous line is a concise way of saying
  #
  # newx = self.x + p.x
  # newy = self.y + p.y
  # newpt = Point(newx,newy)
  # return(newpt)
```

In [16]:
```
p = Point(3,4)
q = Point(7,10)
```

In [17]:
```
p.odistance(), q.odistance()
```

Out[17]: (5.0, 12.206555615733702)

In [18]:
```
p.translate(3,4)
p.odistance()
```

Out[18]: 10.0

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

```
(6,8)
```

In [20]:
```
str(p)
```

Out[20]: '(6,8)'

In [21]:
```
print(p+q)
```

```
(13,18)
```

In [22]:
```
print(p,q)
```

```
(6,8) (7,10)
```

- What if we want to compare two points?

In [23]:
```
p < q
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 1
----> 1 p < q

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

- `<` is mapped to a function `__lt__()`

In [24]:
```
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)
```

In [25]:
```
p = Point(3,4)
q = Point(7,10)
```

In [26]:
```
p < q, q < p
```

Out[26]:
```
(True, False)
```

Changing the implementation

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

In [27]:
```
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 [28]:
```
p = Point(3,4)
q = Point(7,10)
```

In [29]:
```
p.odistance(), q.odistance()
```

Out[29]: (5.0, 12.206555615733702)

In [30]:
```
p.translate(3,4)
p.odistance()
```

Out[30]: 10.0

In [31]:
```
print(p) # Note some lack of precision going from (x,y) to (r,theta) and back
```

```
(6.000000000000001,7.999999999999999)
```

In [32]:
```
str(p)
```

Out[32]: '(6.000000000000001,7.999999999999999)'

In [33]:
```
print(p+q)
```

```
(13.000000000000002,18.0)
```

In [34]:
```
print(p,q)
```

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

In [35]:
```
p < q, q < p
```

Out[35]: (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 [36]:
```
class Experiment:
    def __init__(self,a):
        x = a

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

In [37]:
```
z = Experiment(5)
```

In [38]:
```
str(z)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[38], line 1
----> 1 str(z)

Cell In[36], line 6, in Experiment.__str__(self)
      5 def __str__(self):
----> 6     return(str(x))

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

In [39]:
```
class Experiment2:
    def __init__(self,a):
        self.x = a

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

In [40]:
```
y = Experiment2(7)
str(y)
```

Out[40]: '7'

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

In [41]:
```
class Experiment3:
    def __init__(self,a):
        self.x = a
```

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

In [42]:
```
x = Experiment3(17)
print(x)
```

```
17
```