Objects

• Python supports different type of data
  – 125 (int), 24.04 (float), “Hello” (string)
  – [3,7,8,10] (list)
  – {“India”:”New Delhi”, “Japan”:”Tokyo”}

• Each of above is an object
• Every object has a type, internal data representation and procedures for interaction.

• An object is an instance
  – 125 is an instance of int, Hello is an instance of string
Objects

• In Python everything is an object
• Can create an object
• Can manipulate objects
• Can destroy objects

explicitly using del or just “forget” about them python system will reclaim destroyed or inaccessible objects —called “garbage collection”
Objects

- objects are a data abstraction that capture:
  
  (1) an **internal representation**
  - through data attributes
  
  (2) an **interface** for interacting with object
  - through methods (procedures/functions)
  - defines behaviors but hides implementation
Example: List

- lists are internally represented as linked list
- \[1,2,3,4\]: internal representation should be private

- manipulation of lists
  - \(L[i], L[i:j], +\)
  - \(len(), min(), max(), del(L[i])\)
  - \(L.append(), L.extend(), L.remove(), L.reverse()\)...
Classes

- Classes make it easy to **reuse** code
- There is a distinction between **creating a class** and **using an instance** of the class
- **Creating the class involves**
  - defining the class name
  - defining class attributes
  - for example, someone wrote code to implement a list class
Classes

Abstract concept

Rectangle

Attributes
- width
- height
- colour

Functions
- area()

Multiple concrete instances

<table>
<thead>
<tr>
<th>Width</th>
<th>Height</th>
<th>Colour</th>
<th>Rect1</th>
<th>Rect2</th>
<th>Rect3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>width=5, height=3, colour=(200,0,0)</td>
<td>Rect1</td>
<td>Rect2</td>
<td>Rect3</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>width=7, height=5, colour=(0,200,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>width=3, height=5, colour=(255,200,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

area() -> 15
area() -> 35
area() -> 15
Classes

- **defining a class involves**

  ```python
class Coordinate (object):
    # define attributes here
  ```

  - definition
  - name of parent of class
  - class

  ```python
  # define attributes here
  ```

- **use a special method `__init__` to initialize some data attributes**
Classes

• defining a class involves

    class Coordinate (object):
        def __init__(self, x, y):
            self.x = x
            self.y = y

• creating an instance
  – c = Coordinate(3,4)
  – origin=Coordinate(0,0)
Classes: Methods

- procedural attribute, like a *function that works only with this class*

- Python always passes the object as the first argument
  - convention is to use `self` as the name of the first argument of all methods

- the “.” *operator* is used to access any attribute
  - a data attribute of an object
  - a method of an object
class Coordinate(object):
    
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq = (self.x - other.x)**2
        y_diff_sq = (self.y - other.y)**2
        return sqrt(x_diff_sq + y_diff_sq)
Classes: Methods

c = Coordinate(3,4)
z = Coordinate(0,0)

d = c.distance(z)
print(d)

d = Coordinate.distance(c, z)
print(d)

Equivalent
Classes: Methods

• **defining your own print method**

```python
class Coordinate (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"

c=Coordinate(3,4)
print(c) → <3,4>
```
Example: Fraction

- create a **new type** to represent a number as a fraction

- **internal representation** is two integers
  - numerator
  - denominator

- **interface** a.k.a. **methods** a.k.a **how to interact** with **Fraction objects**
  - add, subtract
  - print representation, convert to a float
  - invert the fraction
Example: Fraction

• fraction (rational number)

class Fraction (object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d
    def __str__(self):
        return str(self.num)+"/"+str(self.denom)
Example: Fraction

- fraction (rational number)

  add two rational numbers:
  ```python
def __add__(self, other):
    num = self.num*other.denom +
    self.denom*other.num
    denom = self.denom*other.denom
    return Fraction(num, denom)
  ```
Example: Fraction

• fraction (rational number)

subtract two rational numbers:

def __subtract__(self, other):
    num = self.num*other.denom - self.denom*other.num
    denom = self.denom*other.denom
return Fraction(num, denom)
Example: Fraction

- fraction (rational number)

    multiply two rational numbers:

def multiply(self, other):
    num = self.num*other.num
    denom = self.denom*other.denom
    return Fraction(num, denom)
Example: Fraction

- fraction (rational number)

  divide two rational numbers:

  ```python
def divide(self, other):
    num = self.num * other.denom
    denom = self.denom * other.num
    return Fraction(num, denom)
  ```
Example: Fraction

- **fraction (rational number)**

```python
def __float__(self):
    return self.num/self.denom

def inverse(self):
    return self.denom/self.num

def reduce(self):
    ....
```