COMPSCI 107
Computer Science Fundamentals

Lecture 09 – Classes

Learning outcomes

- At the end of this lecture, students should be able to:
  - Define a new class
  - Store state information about instances of the class
  - Define new methods of the class
  - Override the default behaviour for standard operations

Classes

- Python has a number of classes built-in
  - lists, dictionaries, sets, int, float, boolean, strings

- We can define our own classes
  - creates a new type of object in Python

```
class name_of_the_class:

    definition of the class goes here
```

- Classes consist of:
  - state variables (sometimes called instance variables)
  - methods (functions that are linked to a particular instance of the class)

Example: Point class

- Defining and using a simple class

```
class Point:
    def __init__(self, loc_x, loc_y):
            self.x = loc_x
            self.y = loc_y
```

```
>>> origin = Point(0, 0)
>>> destination = Point(34, 65)
>>> destination.x
34
>>> destination.y
65
```

Classes

- A class provides the definition for the type of an object
  - Classes can store information in variables
  - Classes can provide methods that do something with the information

- Example: A square class

```
class Square:

    def __init__(self, s):
        self.size = s
```

```
from Geometry import Square

side = 10
s = Square(side)
```

Example

- Task: Add a method to the class to calculate the perimeter of the square. The following code shows how the method may be used.

```
from Geometry import Square

side = 10
s = Square(side)
p = s.perimeter()
```

```
class Square:
    def __init__(self, s):
        self.size = s

    def perimeter(self):
        return self.size * 4
```

Exercise

- Add a method to the class to return a square that is bigger by a scaling factor. For example, if you scale the square by a factor of 2, then the sides of the square will be twice as long. The following code shows how the method may be used.

```
from Geometry import Square

side = 10
s = Square(side)
big_s = s.scaled_square(2)
```

Exercise

- Write a function that compares the size of two squares given as parameters. This function should not be part of the Square class.

```
def is_bigger(a, b):
    #returns true if a is larger than b
    #add your code here
```

Exercise

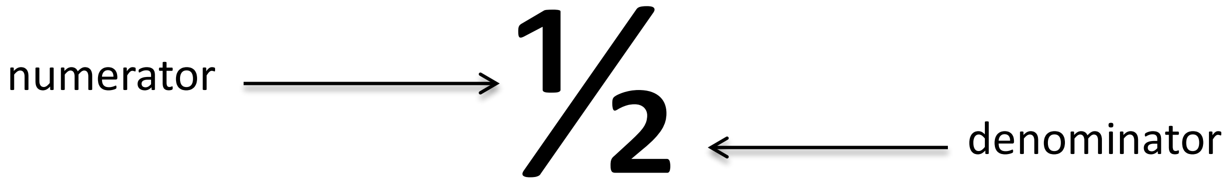
- Add a method to the Square class that compares the size of the square with the size of another square. The method should be called bigger_than() and should accept a square as a parameter

```
from Geometry import Square

s = Square(6)
t = Square(7)
if s.bigger_than(t):
    print(“The first square is bigger”)
```

Example: Fractions

- Write a class to represent fractions in Python
  - create a fraction
  - add
  - subtract
  - multiply
  - divide
  - text representation

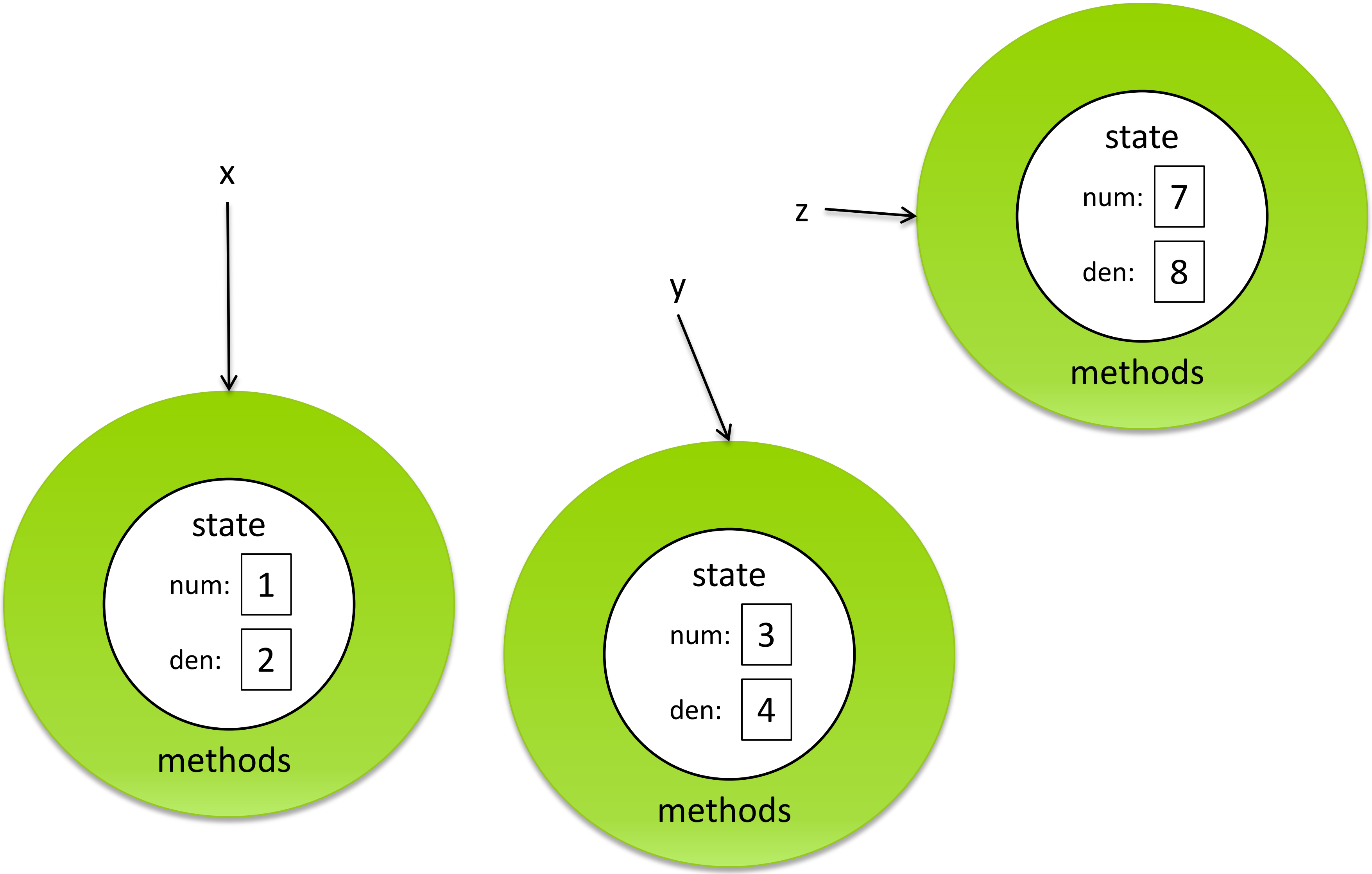


numerator

½

denominator

Model of objects in memory



Constructor

- All classes must have a constructor
  - The constructor for a Fraction should store the numerator and the denominator

```
class Fraction:
    def __init__(self, top, bottom):
        self.num = top           #numerator
        self.den = bottom        #denominator
```

Using the Fraction class

- So far, we can create a Fraction

```
>>> x = Fraction(3, 4)
```

- We can access the state variables directly
  - Although not generally good practice to do so

```
>>> x.num
3
>>> x.den
4
```

- What else can we do with Fractions?
  - Nothing yet. We need to write the functions first!

Overriding default behaviour

- All classes get a number of methods provided by default
  - Since default behaviour is not very useful, we should write our own versions of those methods

Aside: Use of string formatting syntax

- Often we want to use a string that combines literal text and information from variables

- Example:

```
name = 'Andrew'
greeting = 'Hello ' +  name + '. How are you?'
```

- We can use string formatting to perform this task
  - Use curly braces within the string to signify a variable to be replaced

```
my_name = 'Andrew'
greeting = 'Hello {name}.  How are you?'.format(name=my_name)
```

  - We can put the argument position in the curly braces

```
first = 'Andrew'
second = 'Luxton-Reilly'
greeting = 'Hello {0} {1}'.format(first, second)
```

Exercise

- What is the output from the following code:

```
sentence = 'Hello {2}.  It is {0} today and it is {1}.'.format('Andrew', 'Wednesday', 'Cold')
```

- Rewrite the code so that it uses explicit variable names in the string.

__repr__

- The __repr__ method produces an string that unambiguously describes the object
  - All classes should have a __repr__ function implemented
  - Ideally, the representation could be used to create the object
  - For example, a fraction created using Fraction(2, 3) should have a __repr__ method that returned 'Fraction(2, 3)'

```
>>> x = Fraction(2, 3)
>>> x
<__main__.Fraction object at 0x02762290>
```

```
def __repr__(self):
    return 'Fraction({0}, {1})'.format(self.num, self.den)
```

```
>>> x = Fraction(2, 3)
>>> x
Fraction(2, 3)
```

__str__

- The __str__ method returns a string representing the object
  - By default, it calls the __repr__ method
  - The __str__ method should focus on being human readable

```
>>> x = Fraction(3, 4)
>>> print(x)
<__main__.Fraction object at 0x02714290>
```

- We should implement a version with a natural representation:

```
def __str__(self):
    return str(self.num) + '/' + str(self.den)
```

- After we have implemented the method, we can use standard Python

```
>>> x = Fraction(3, 4)
>>> print(x)
3/4
```

Exercise

- Write the __repr__ method for the Square class created earlier.
- Would it be useful to implement a __str__ method?
- What would you choose to produce as output from a __str__ method?

__add__

- The __add__ method is called when the + operator is used
  - If we implement __add__ then we can use + to add the objects
  - f1 + f2 gets translated into f1.__add__(f2)

```
def __add__(self, other):
    new_num = self.num * other.den + self.den * other.num
    new_den = self.den * other.den
    return Fraction(new_num, new_den)
```

```
x = Fraction(1, 2)
y = Fraction(1, 4)
z = x + y
print(z)
6/8
```

Greatest Common Divisor

- Use Euclid's Algorithm
  - Given two numbers, n and m, find the number k, such that k is the largest number that evenly divides both n and m.

```
def gcd(m, n):
    while m % n != 0:
        old_m = m
        old_n = n
        m = old_n
        n = old_m % old_n
    return n
```

Improve the constructor

- We can improve the constructor so that it always represents a fraction using the "lowest terms" form.
    - What other things might we want to add to a Fraction?

```
class Fraction:
    def __init__(self, top, bottom):
        common = Fraction.gcd(top, bottom)       #get largest common term
        self.num = top // common           #numerator
        self.den = bottom // common      #denominator

    def gcd(m, n):
        while m % n != 0:
            old_m = m
            old_n = n
            m = old_n
            n = old_m % old_n
        return n
```

__eq__

- The __eq__ method checks equality of the objects
  - Default behaviour is to compare the references
  - We want to compare the contents

```
def __eq__(self, other):
      return self.num * other.den == other.num * self.den
```

Exercise

- What is the output of the following code?

```
x = Fraction(2, 3)
y = Fraction(1, 3)
z = y + y
print(x == z)
print(x is z)
w = x + y
print(w == 1)
```

Improving __eq__

- Check the type of the other operand
  - If the type is not a Fraction, then not equal?
  - What other decisions could we make for equality?

```
def __eq__(self, other):
    if not isinstance(other, Fraction):
        return False
    return self.num * other.den == other.num * self.den
```

Exercise

- Check the type of the other operand
  - If the type is an integer, then compare against our Fraction

```
def __eq__(self, other):
    # Add your code to compare the Fraction with an int

    if not isinstance(other, Fraction):
        return False
    return self.num * other.den == other.num * self.den
```

Other standard Python operators

- Many standard operators and funtions:

  https://docs.python.org/3.4/library/operator.html

- Common Arithmetic operators
  - object.__add__(*self, other*)
  - object.__sub__(*self*, *other*)
  - object.__mul__(*self*, *other*)
  - object.__truediv__(*self*, *other*)

Inplace arithmetic operators

- object.__iadd__(*self, other*)
- object.__isub__(*self*, *other*)
- object.__imul__(*self*, *other*)
- object.__itruediv__(*self*, *other*)

- Common Relational operators
  - object.__lt__(*self*, *other*)
  - object.__le__(*self*, *other*)
  - object.__eq__(*self*, *other*)
  - object.__ne__(*self*, *other*)
  - object.__gt__(*self*, *other*)
  - object.__ge__(*self*, *other*)

Summary

- All types in Python are defined in a class
  - All operators are translated into a method call
  - All "standard" Python functions are translated into method calls
  - When we write our own classes, we can define behaviour for standard operators