DM536 / DM550 Part I
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM536/

MIDWAY EVALUATION

Group Formation

- Real world problem:
  - divide the students evenly to exactly 7 groups
  - do not count the students beforehand
- Algorithm design:
  - repeatedly assign the numbers 1 to 7 to the students
  - sort according to the assigned numbers
- Pseudo Python:

```
sorted(zip(range(1,7+1)*9**9),students))
```

- Implementation!

Group Work

- Time frame: max 10 minutes
- Tasks:
  1. elect a speaker for the panel discussion
  2. find at least three items for each of the three columns:

What has been good during the course?	What has been less good?	Suggestions for improvements
1.		
2.		
3.		

Panel Discussion

- I will be the panel's secretary ☺
- Time frame: max 10 minutes
- Three Phases:
  1. Presentation of the results of your group work.
  2. Panel discussion regarding whether and to which degree you agree with the results of the other groups.
  3. Open discussion with the "secretary" and the whole "class".

CLASSES & OBJECTS

User-Defined Types

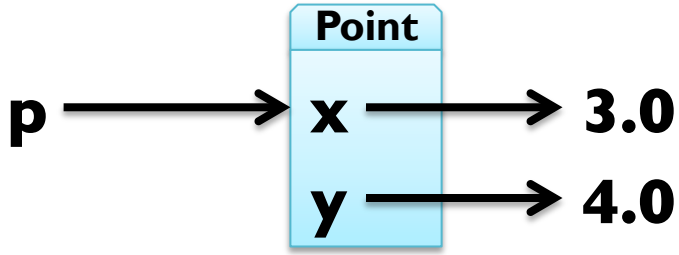
- we want to represent points (x,y) in 2-dimensional space
- which data structure to use?
  - use two variables x and y
  - store coordinates in a list or tuple of length 2
  - create user-defined type
- we can use Python's classes to implement new types
- Example:

```
class Point(object):
    """represents a point in 2-dimensional space"""
print Point      # class
p = Point()      # create new instance of class Point
print p          # instance
```

Attributes

- using *dot notation*, you can assign values to instance variables
- Example:

```
p.x = 3.0
p.y = 4.0
```



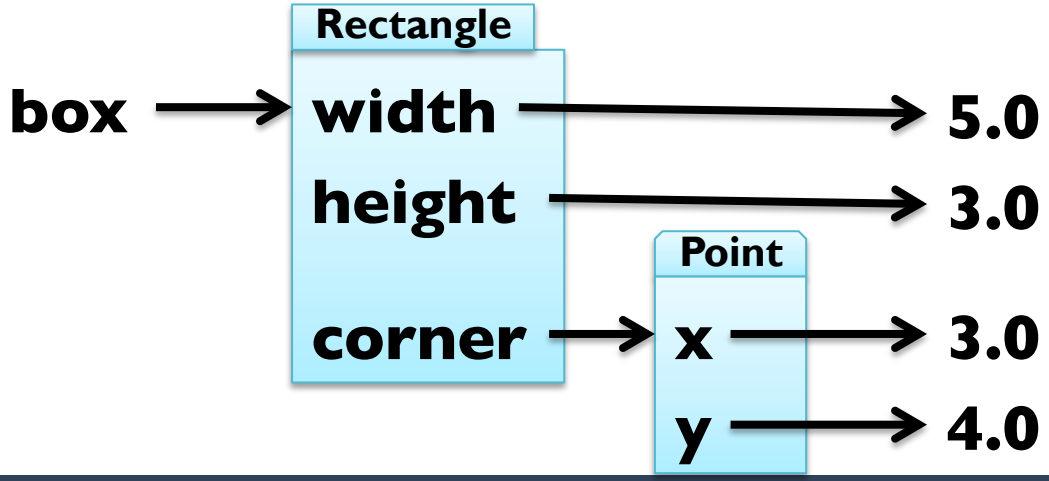
- instance variables are called *attributes*
- attributes can be assigned to and read like any variable
- Example:

```
print "(%g, %g)" % (p.x, p.y)
distance = math.sqrt(p.x**2 + p.y**2)
print distance, "units from the origin"
```

Representing a Rectangle

- rectangles can be represented in many ways, e.g.
  - width, height, and one corner or the center
  - two opposing corners
- here we choose width, breadth and the lower-left corner
- Example:

```
class Rectangle(object):
    "represents a rectangle using attributes width, height, corner”
box = Rectangle()
box.width = 5.0
box.height = 3.0
box.corner = p
```



Instances as Return Values

- functions can return instances
- Example: find the center point of a rectangle

```
def find_center(box):
    p = Point()
    p.x = box.corner.x + box.width / 2.0
    p.y = box.corner.y + box.height / 2.0
    return p
box = Rectangle()
box.width = 5.0;        box.height = 3.0
box.corner = Point()
box.corner.x = 3.0;     box.corner.y = 4.0
print find_center(box)
```

Objects are Mutable

- by assigning to attributes, an object is changed
- Example: update size of rectangle

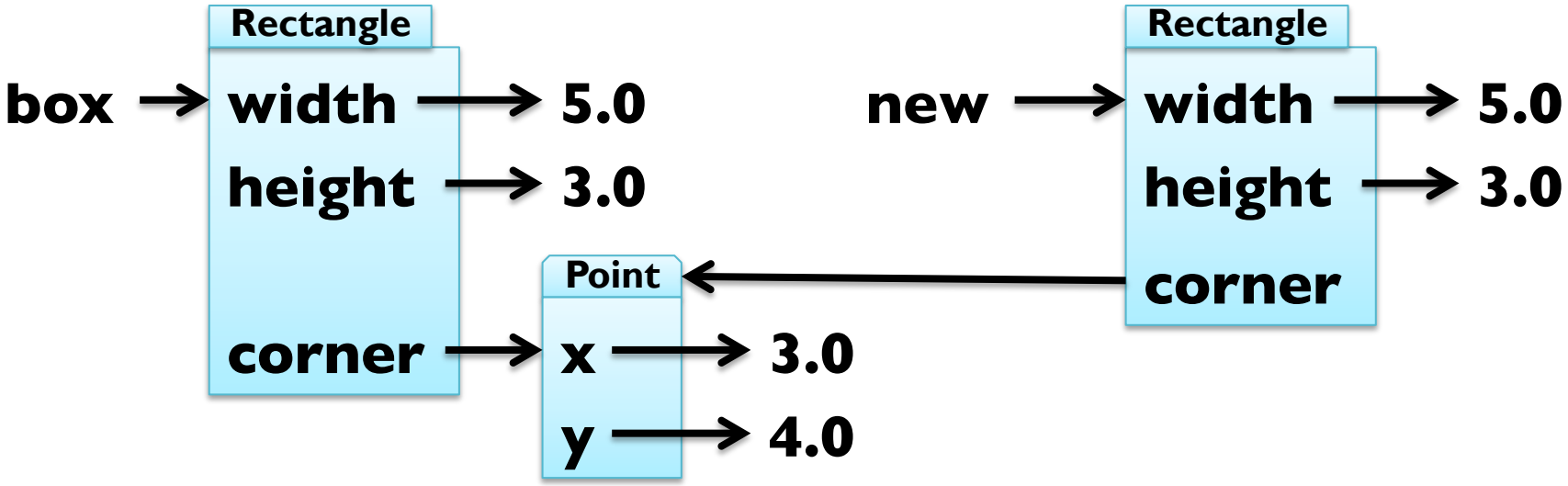
```
box.width = box.width + 5.0
box.height = box.height + 3.0
```

- consequently, also functions can change object arguments
- Example:

```
def double_rectangle(box):
    box.width *= 2
    box.height *= 2
double_rectangle(box)
```

Copying Objects

- import module copy to make copies of objects
- Example: import copy
  new = copy.copy(box)



- shallow copy, use copy.deepcopy(object) to also copy Point

Debugging User-Defined Types

- you can obtain type of an instance by using type(object)
- Example: print type(box)

- you can check if an object has an attribute using hasattr
- Example: hasattr(box, "corner") == True

- you can get a list of all attributes using dir(object)
- Example: dir(box)

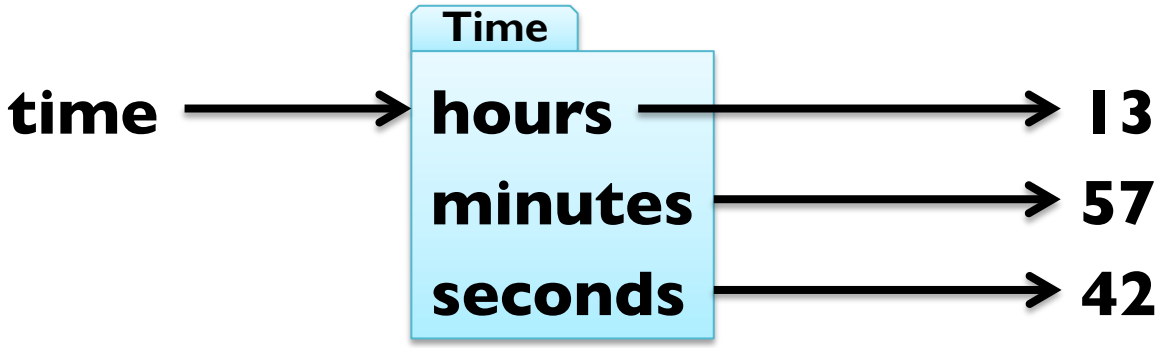
- print __doc__ and __module__ for more information!

CLASSSES & FUNCTIONS

Representing Time

- Example: user-defined type for representing time

```
class Time(object):
  """represents time of day using hours, minutes, seconds"""
time = Time()
time.hours = 13
time.minutes = 57
time.seconds = 42
```



Pure Functions

- pure function = does not modify mutable arguments
- Example: add two times

```
def add_time(t1, t2):
  sum = Time()
  sum.hours = t1.hours + t2.hours
  sum.minutes = t1.minutes + t2.minutes
  sum.seconds = t1.seconds + t2.seconds
  return sum
time = add_time(time, time)
print "%dh %dm %ds" % (time.hours, time.minutes, time.seconds)
```

Modifiers

- modifiers = functions that modify mutable arguments
- Example: incrementing time

```
def increment(time, seconds):
    time.seconds += seconds
```

```
increment(time, 86400)
print "%dh %dm %ds" % (time.hours, time.minutes, time.seconds)
```

Modifiers

- modifiers = functions that modify mutable arguments
- Example: incrementing time

```
def increment(time, seconds):
    time.seconds += seconds
    minutes, time.seconds = divmod(time.seconds, 60)
    time.minutes += minutes
    time.hours, time.minutes = divmod(time.minutes, 60)
increment(time, 86400)
print "%dh %dm %ds" % (time.hours, time.minutes, time.seconds)
```

- this was *prototype and patch* (or *trial and error*)

Prototyping vs Planning

- alternative to prototyping is *planned development*
- high-level observation: time representable by just seconds
- Example: refactoring function working with time

```
def time_to_int(time):
    return time.seconds + 60 * (time.minutes + 60 * time.hours)
def int_to_time(seconds):
    time = Time();  minutes, time.seconds = divmod(seconds, 60)
    time.hours, time.minutes = divmod(minutes, 60);   return time
def add_time(t1, t2):
    return int_to_time(time_to_int(t1) + time_to_int(t2))
```

Prototyping vs Planning

- alternative to protyping is *planned development*
- high-level observation: time representable by just seconds
- Example: refactoring function working with time

```
def time_to_int(time):
    return time.seconds + 60 * (time.minutes + 60 * time.hours)
def int_to_time(seconds):
    time = Time();  minutes, time.seconds = divmod(seconds, 60)
    time.hours, time.minutes = divmod(minutes, 60);   return time
def increment(time, seconds):
    t = int_to_time(seconds + time_to_int(time))
    time.seconds = t.seconds;  time.minutes = t.minutes
    time.hours = t.hours
```

Prototyping vs Planning

- alternative to protyping is *planned development*
- high-level observation: time representable by just seconds
- Example: refactoring function working with time

```
def time_to_int(time):
    return time.seconds + 60 * (time.minutes + 60 * time.hours)
def int_to_time(seconds):
    time = Time();  minutes, time.seconds = divmod(seconds, 60)
    time.hours, time.minutes = divmod(minutes, 60);   return time
def increment(time, seconds):
    return int_to_time(seconds + time_to_int(time))
```

Debugging using Invariants

- invariant = requirement that is always true
- assertion = statement of an invariant using assert
- Example: check that time is valid

```
def valid_time(time):
    if time.hours < 0 or time.minutes < 0 or time.seconds < 0:
        return False
    return time.minutes < 60 and time.seconds < 60
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    return int_to_time(time_to_int(t1) + time_to_int(t2))
```

- also useful to check before return value

CLASSES & METHODS

Object-Oriented Features

- object-oriented programming in a nutshell:
  - programs consists of class definitions and functions
  - classes describe real or imagined objects
  - most functions and computations work on objects
- so far we have only used classes to store attributes
- i.e., functions were not linked to objects

- methods = functions defined inside a class definition
  - first argument is always the object the method belongs to
  - calling by using dot notation
  - Example: "Slartibartfast".count("a")

Printing Objects

- printing can be done by a normal function
- better done with a method
- Example:

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""'
    def print_time(time):
        t = (time.hours, time.minutes, time.seconds)
        print "%02dh %02dm %02ds" % t

def print_time(time):
    t = (time.hours, time.minutes, time.seconds)
    print "%02dh %02dm %02ds" % t
```

Printing Objects

- printing can be done by a normal function
- better done with a method
- Example:

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""'
    def print_time(self):
        t = (self.hours, self.minutes, self.seconds)
        print "%02dh %02dm %02ds" % t

def print_time(time):
    t = (time.hours, time.minutes, time.seconds)
    print "%02dh %02dm %02ds" % t
```

Printing Objects

- printing can be done by a normal function
- better done with a method
- Example:

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""
    def print_time(self):
        t = (self.hours, self.minutes, self.seconds)
        print "%02dh %02dm %02ds" % t
end = Time()
end.hours = 12;  end.minutes = 15;  end.seconds = 37
Time.print_time(end)          # what really happens
end.print_time()              # how to write it!
```

Incrementing as a Method

- Example: add increment as a method

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""
    def time_to_int(self):
        return self.seconds + 60 * (self.minutes + 60 * self.hours)
    def int_to_time(self, seconds):
        minutes, self.seconds = divmod(seconds, 60)
        self.hours, self.minutes = divmod(minutes, 60)
    def increment(self, seconds):
        return self.int_to_time(seconds + self.time_to_int())
```