DM536
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

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

PROJECT PART 2

Organizational Details

- 2 possible projects, each consisting of 2 parts
- for 1st part, you have to pick one
- for 2nd part, you can STAY or you may SWITCH

- projects must be done individually, so no co-operation
- you may talk about the problem and ideas how to solve them

- deliverables:
  - written 4 page report as specified in project description
  - handed in electronically as a single PDF file!
  - deadline: October 31, 12:00

- ENOUGH - now for the CLASSY part …

Fractals and the Beauty of Nature

- geometric objects similar to themselves at different scales

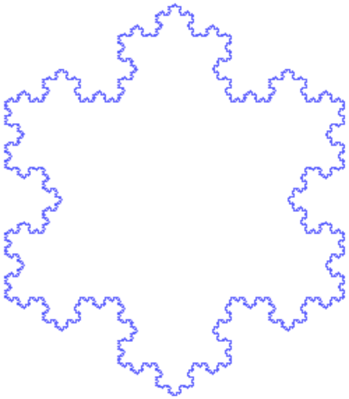
- many structures in nature are fractals:
  - snowflakes
  - lightning
  - ferns

- **Goal:** generate fractals from Fractal Description Language

- **Challenges:** Representation, Interpretation, File Handling

Fractals and the Beauty of Nature

- Task 0: Preparation
  - understanding descriptions given in .fdl files



F fd
L lt 60
R rt 120

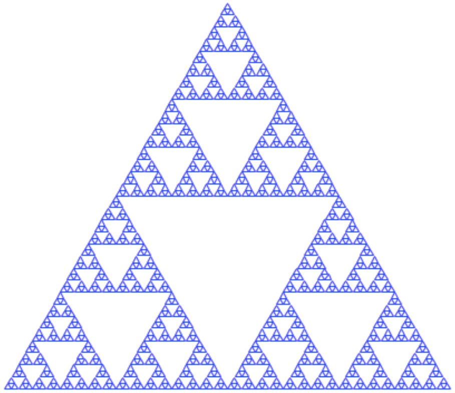
- Task 1: Rules
  - representing and applying rewriting rules

F -> F L F R F L F

F -> F L F L F L F F

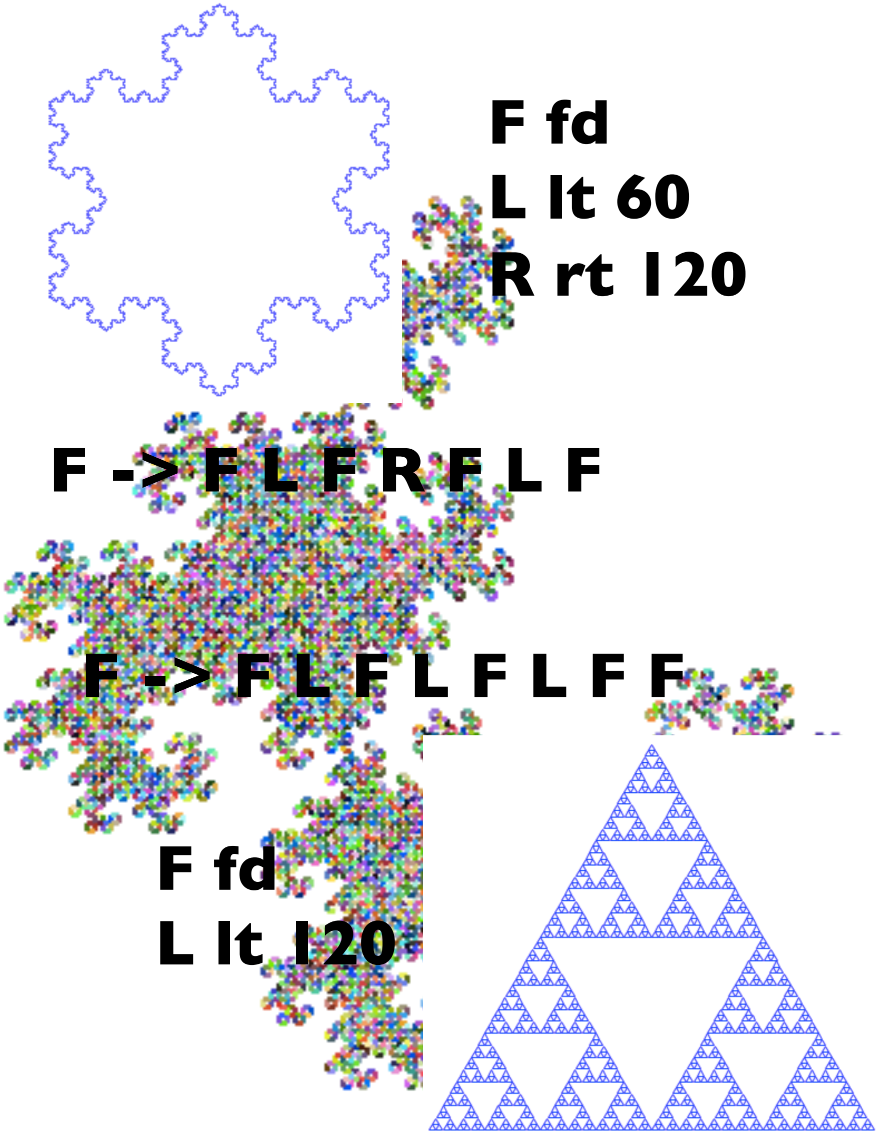
- Task 2: Commands
  - representing and executing turtle commands

F fd
L lt 120



Fractals and the Beauty of Nature

- Task 3: Loading Files
  - load and interpret fractal descripton language files
- Task 4: Generating Fractals
  - compute new states and draw the fractal
- Task 5 (optional): Colors / LW
  - add support for colors and line widths



F fd
L lt 60
R rt 120

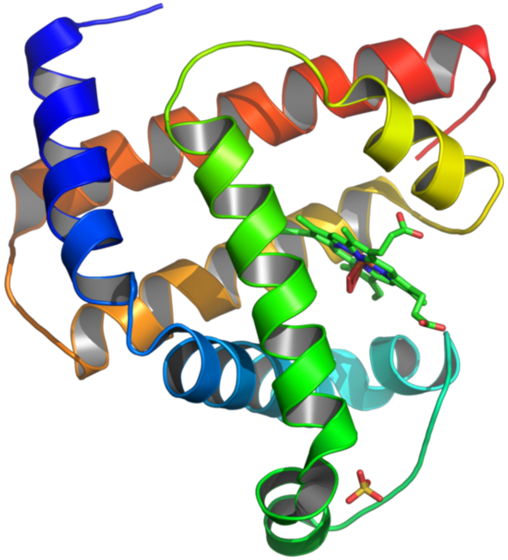
F -> F L F R F L F

F -> F L F L F L F F

F fd
L lt 120

From DNA to Proteins

- proteins encoded by DNA base sequence using A, C, G, and T
- Background:
  - proteins are sequences of amino acids
  - amino acids encoded using three bases
  - chromosomes given as base sequences



- **Goal:** build proteins from base sequences
- **Challenges:** Nested Data Structures, Representation

From DNA to Proteins

- Task 0: Preparation
  - output base sequences OR read them from file
- Task 1: Representing Amino Acids
  - create user-defined type and read instances from file
- Task 2: Setting up the Translation
  - create user-defined type Ribsome as translator
- Task 3: Creating Proteins
  - represent and assemble proteins as amino acid sequences
- Task 4 (optional): Representing Codons
  - replace strings of length 3 by a user-defined type

CLASSES & OBJECTS

User-Defined Types

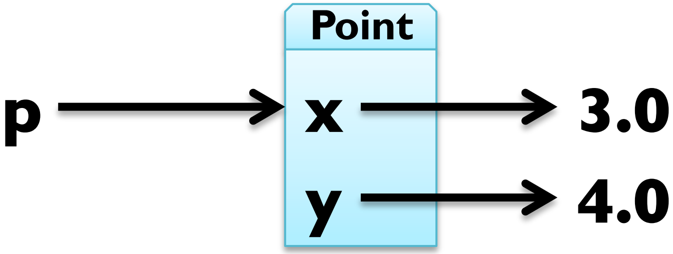
- we want to represent points (x,y) in 2-dimensional space
- which data structure to use?
  - use two variables x and y
  - store coordinates in a list or tuple of length 2
  - create user-defined type
- we can use Python’s classes to implement new types
- Example:

```
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
```



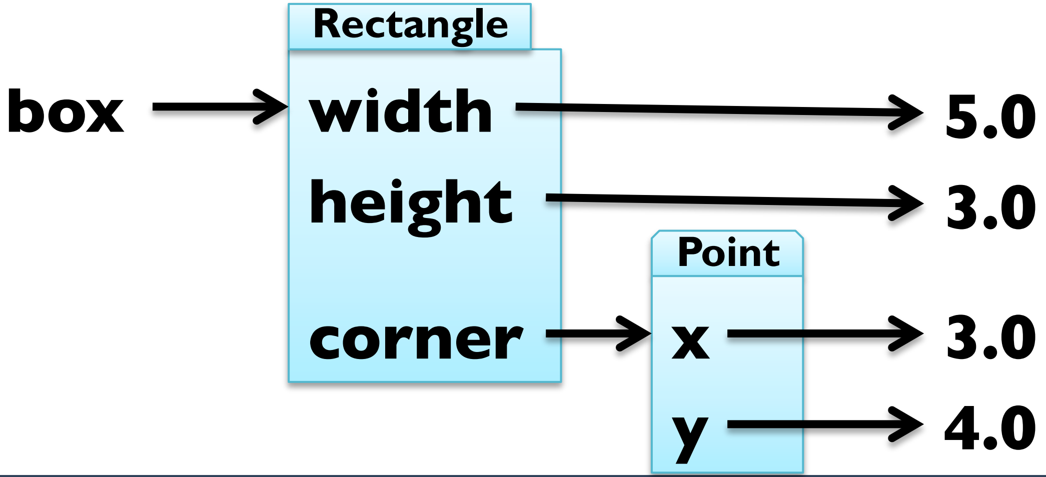
- instance variables are called *attributes*
- attributes can be assigned to and read like any variable
- Example:

```
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
```

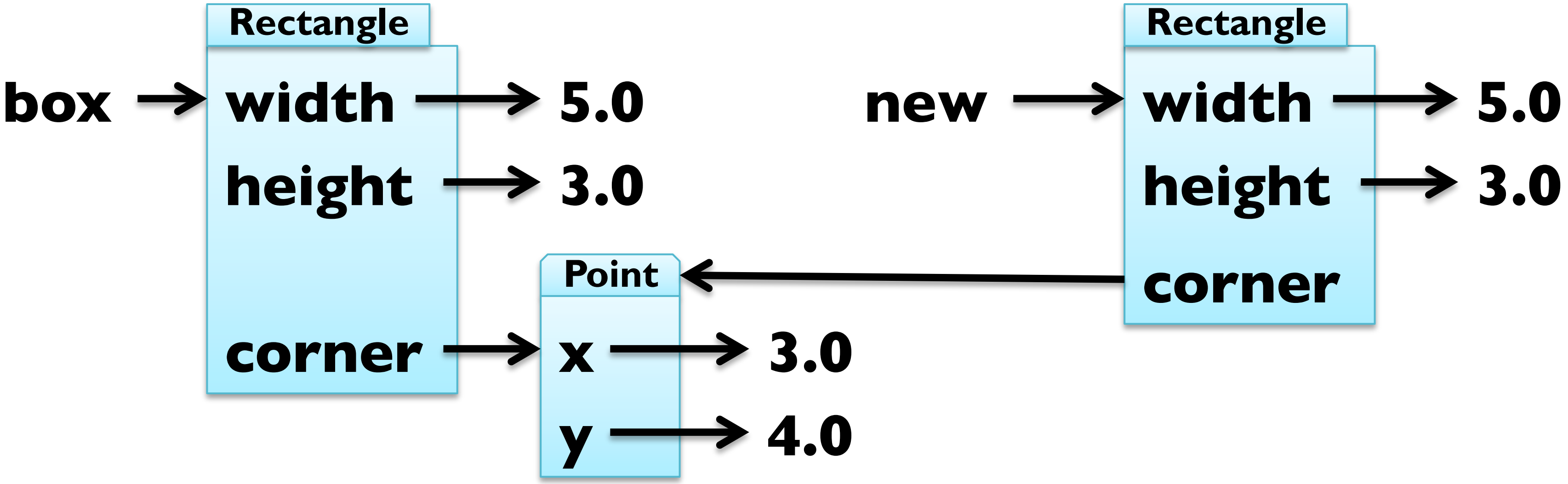
- consequently, also functions can change object arguments
- Example:

```
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)

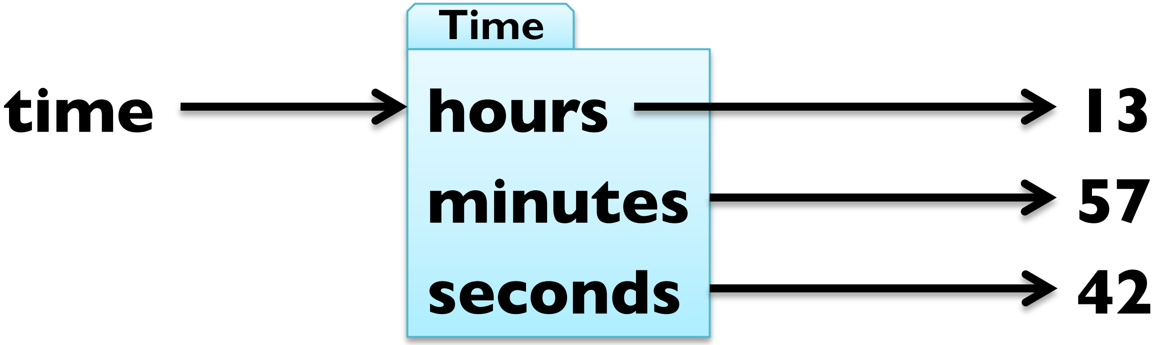
- print __doc__ and __module__ for more information!

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())
```

Comparing with Methods

- Example: add is_after 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())
    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Initializing Objects

- special method __init__(self, …) to create new objects
- usually first method written for any new class!
- Example: initialize Time objects using __init__

```
class Time(object):
    """represents time of day using hours, minutes, seconds""”
    def __init__(self, hours, minutes, seconds):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
start = Time(12, 23, 42)
start = Time()
start.hours = 12;  start.minutes = 23; start.seconds = 42
```

String Representation of Objects

- special method __str__(self) to convert objects to strings
- Example: print Time objects using __str__

```
class Time(object):
    """represents time of day using hours, minutes, seconds""”
    def __init__(self, hours, minutes, seconds):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
    def __str__(self):
        t = (self.hours, self.minutes, self.seconds)
        return "%dh %dm %ds" % t
print Time(7, 42, 23)
```

Representation of Objects

- special method __repr__(self) to represent objects
- Example: make Time objects more usable in lists

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""
    def __str__(self):
        t = (self.hours, self.minutes, self.seconds)
        return "%dh %dm %ds" % t
    def __repr__(self):
        t = (self.hours, self.minutes, self.seconds)
        return "Time(%s, %s, %s)" % t
print [Time(7, 42, 23), Time(12, 23, 42)]
```

Representation of Objects

- special method __repr__(self) to represent objects
- Example: make Time objects more usable in lists

```
class Time(object):
    """represents time of day using hours, minutes, seconds""”
    def as_tuple(self):
        return (self.hours, self.minutes, self.seconds)
    def __str__(self):
        return ”%dh %dm %ds" % self.as_tuple()
    def __repr__(self):
        return "Time(%s, %s, %s)" % self.as_tuple()
print [Time(7, 42, 23), Time(12, 23, 42)]
```

Overloading Operators

- special method __add__(self, other) to overload “+” operator
- likewise, you can use __mul__(self, other) etc.
- Example: add Time objects using __add__

```
class Time(object):
    """represents time of day using hours, minutes, seconds""”
    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return self.int_to_time(seconds)
t1 = Time(2, 40, 19)
t2 = Time(10, 2, 23)
print t1 + t2
```

Type-Based Dispatch

- we want to add both Time objects and seconds
- use isinstance(object, class) to determine type of argument
- Example:

```
class Time(object):
    def __add__(self, other):
        if isinstance(other,Time):   return self.add_time(other)
        else:                        return self.add_seconds(other)
    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return self.int_to_time(seconds)
    def add_seconds(self, seconds):
        return self.int_to_time(seconds + self.time_to_int())
```

Polymorphism

- polymorphic = working on different argument types
- Examples:
  - histogram(s) can be used for lists & tuples of elements, that can be used as dictionary keys
  - sum(t) can be used for lists & tuples of elements, for which "+" works, i.e., also for Time
- to use e.g. Time as dictionary keys, implement __hash__(self)
- important that returned integer identical for identical objects

Debugging by Introspection

- hard to work with objects where attributes are added
- try to always use __init__(self, …) to create attributes
- do not create attributes (or methods) from “outside”

- you can use dir(object) to get list of attributes and methods

- special attribute __dict__ maps attributes to values
- Example: print all atributes and their values and types

```
for var, value in time.__dict__.items():
    print "%s -> %s (%s)" % (var, value, type(value))
```