



DM502

Programming A

Peter Schneider-Kamp

petersk@imada.sdu.dk

<http://imada.sdu.dk/~petersk/DM502/>

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 BOTH electronically and as paper
 - deadline: October 28, 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:

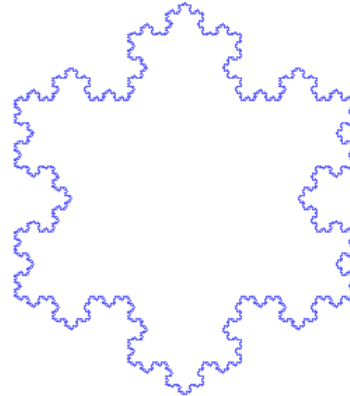
- snowflakes
- lightning
- ferns



- **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
- Task 1: Rules
 - representing and applying rewriting rules
- Task 2: Commands
 - representing and executing turtle commands

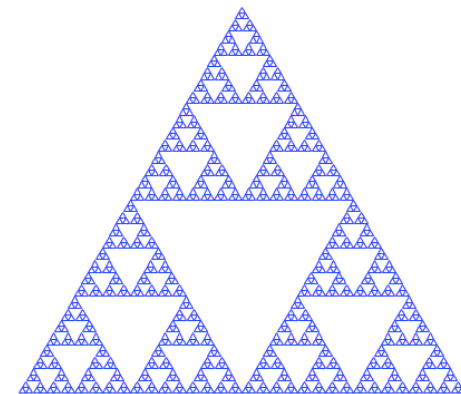


F fd
L lt 60
R rt 120

F -> F L F R F L F

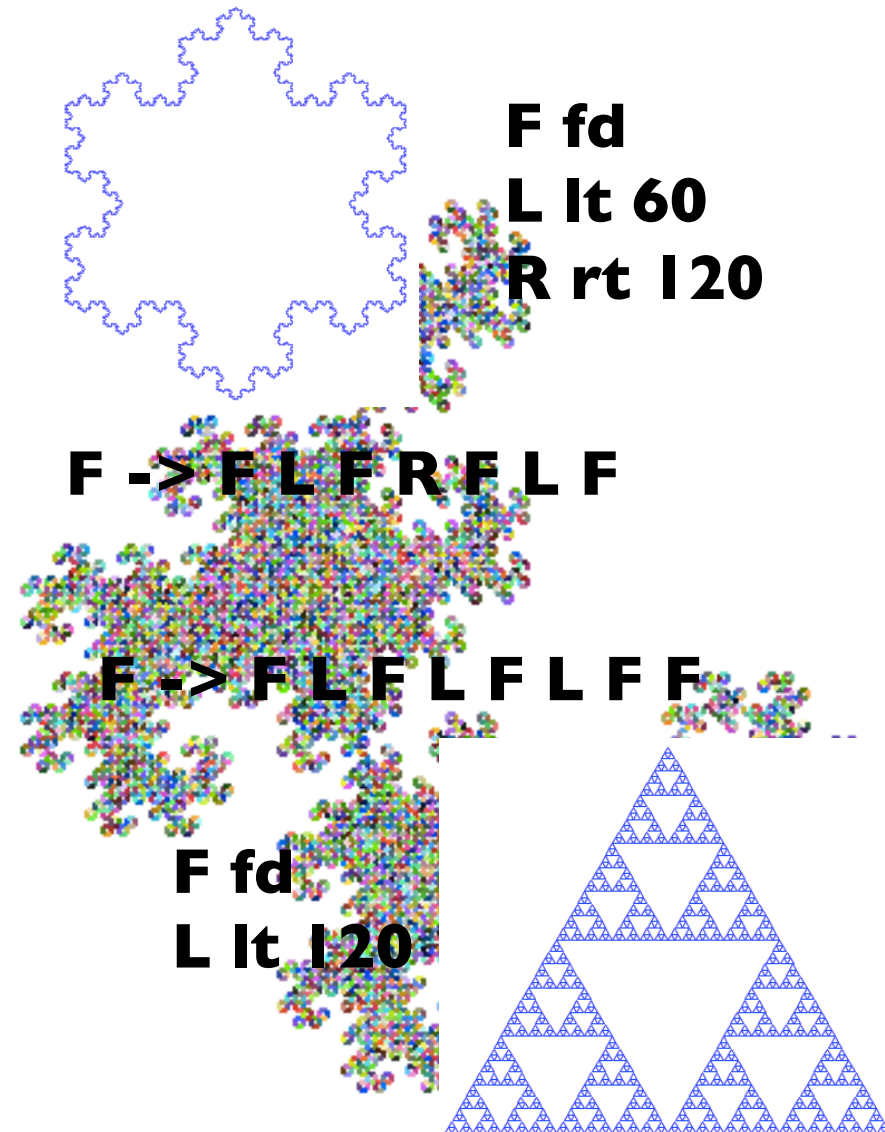
F -> F L F L F L F F

F fd
L lt 120



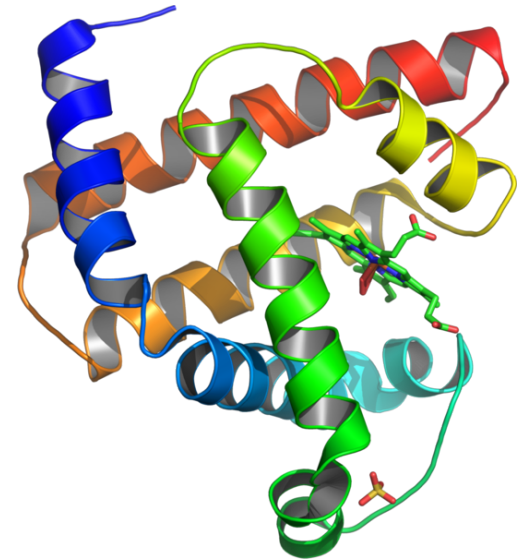
Fractals and the Beauty of Nature

- Task 3: Loading Files
 - load and interpret fractal description 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



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 **Ribosome** 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

FILE HANDLING

Persistence

- persistent = keeping (some) data stored during runs
- transient = beginning from input data each time over
- most programs so far have been transient
- examples of persistent programs:
 - operating systems
 - web servers
 - most app(lication)s on recent iOS and Mac OS X
- text files are easiest way to save some program state
- alternatively, program states can be saved in databases

Writing to a File

- we know how to read a file using `open(name)`
- we can specify read/write mode using `open(name, mode)`
- Example: `f1 = open("anna_karenina.txt", "r")`
`f2 = open("myfile.txt", "w")`
- use method `write(str)` of file object to append string to file
- Example: `f2.write("This is my first line!\n")`
`f2.write("This is my second line!\n")`
- each invocation of `write(str)` will append, not overwrite!
- when you are finished with a file, please `close()` it
- Example: `f1.close()`
`f2.close()`

Format Operator

- values need to be converted to a string for use in `write(str)`
- for single value, the `str(object)` function can be used
- Example: `f.write(str(42))`
- alternatively, use *format operator* “%”
- Example: `f.write("%d" % 42)`
`f.write("The answer is %d, my friend!" % 42)`
- first argument *format string*, second argument value
- format sequence `%d` for integer, `%g` for float, `%s` for string
- for multiple values, use tuple as value
- Example: `f.write("The %s is %g!" % ("answer", 42.0))`

Directories

- file are organized in *directories*
- every program has a *current directory*
- the current directory is used by default, e.g. for `open(name)`
- get current directory by importing `getcwd()` from `os` module
- Example:

```
import os  
print os.getcwd()
```
- change current working directory by using `chdir(path)`
- Example:

```
os.chdir("../")  
print os.getcwd()
```
- list contents of a given directory by using `os.listdir(path)`
- Example:

```
print os.listdir("dm502")
```

Filenames and Paths

- `path` = directory & file name
- *relative paths* start from current directory
- Example:

```
path1 = "dm502/tools/anna_karenina.txt"
```

- *absolute paths* are independent from current directory
- Example:

```
path2 = "/Users/petersk/sdu/dm502/tools/anna_karenina.py"
```

- can be obtained from relative path using `os.path.abspath(path)`
- Example:

```
path3 = os.path.abspath(path1)
```

Operations on Paths

- check whether a directory or file exists using `os.path.exists`
- Example: `os.path.exists(path I) == True`
`os.path.exists("no_name") == False`
- check whether a path is a directory using `os.path.isdir`
- Example: `os.path.isdir(path I) == False`
`os.path.isdir("..") == True`
- check whether a path is a file using `os.path.isfile`
- Example: `os.path.isfile(path I) == True`
`os.path.isfile("..") == False`

Traversing Directories

- build a path from directory and relative path using `os.path.join`
- Example: `path4 = os.path.join("../", "dm502")`
- Case: recursively find all files in a directory

```
def find_files(dir):
```

```
    for name in os.listdir(dir):
```

```
        path = os.path.join(dir, name)
```

```
        if os.path.isfile(path):    # print file name
```

```
            print path
```

```
        else:                        # recursively search subdirectory
```

```
            find_files(path)
```


Catching Exceptions

- file operations are error-prone
- Example: `open("no_name")` # raises IOError
- good idea to avoid errors using `os.path.exists` etc.
- not possible to check all possible situations
- use try-except statement to handle error situations
- Example:

```
try:  
    f = open(name)  
    lines = f.readlines()  
except:  
    lines = ["ERROR"]
```

Databases

- import module `anydbm` to open (& possibly create) database
- Example:

```
import anydbm
db = anydbm.open("phonebook.db", "c")
db["Schneider-Kamp, Peter"] = "65502327"
print db["Schneider-Kamp, Peter"]
```
- persistent, i.e., mapping still available after closing program
- Example:

```
import anydbm
db = anydbm.open("phonebook.db", "c")
print db["Schneider-Kamp, Peter"]
```
- in principle works exactly like a dictionary
- **BUT** can only map strings to strings!

Pickling

- import module `pickle` to translate objects into strings
- function `dumps(obj)` translates any object into a string
- Example: `blocked = [6550, 555]`
`db["blocked"] = pickle.dumps(blocked)`
- function `loads(str)` translates such a string into an object
- Example: `my_blocked = pickle.loads(db["blocked"])`
- `dumps` + `loads` results in a copy of the object
- Example: `blocked == my_blocked`
`blocked is my_blocked == False`

Shells and Pipes

- import module `os` for access to shells and pipes
- you can execute arbitrary shell commands using `os.system`
- Example: `os.system("ls -l")` `# print current directory`

- you can grab the output of commands using pipes
- Example: `f = os.popen("ls -l")`
`print f.read()`

- useful e.g. for reading a (g-)zipped files line by line
- Example: `f = os.popen("gunzip -c test.gz")`
`for line in f.readlines():` `print line`

Writing Modules

- any file containing Python code can be imported as module

- Example:

```
open("test.py", "w").write("def f(): return 42\nprint f()")  
import test
```

- any code in module will be executed
- to avoid that, it is common to test whether a program is run
- Example: better test.py

```
def f():  
    return 42  
  
if __name__ == "__main__":  
    print f()
```

Debugging File Operations

- when working with files, whitespace can be hard to debug
- printing a string containing whitespace makes it invisible
- use built-in function `repr(object)` instead
- Example:

```
s = "Hello\n\r\tWorld \t \t!"  
print s  
print repr(s)
```
- different operating systems use different line ends
- Linux & Mac OS X use `"\n"`, Windows uses `"\r\n"`
- use a tool (e.g. `dos2unix`, `unix2dos`) to convert
- alternatively, write your own Python program 😊

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
```

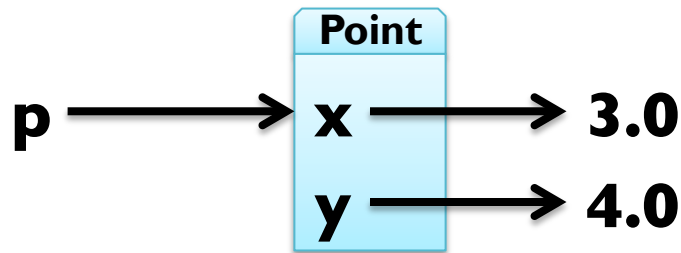
```
print p         # instance
```


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):

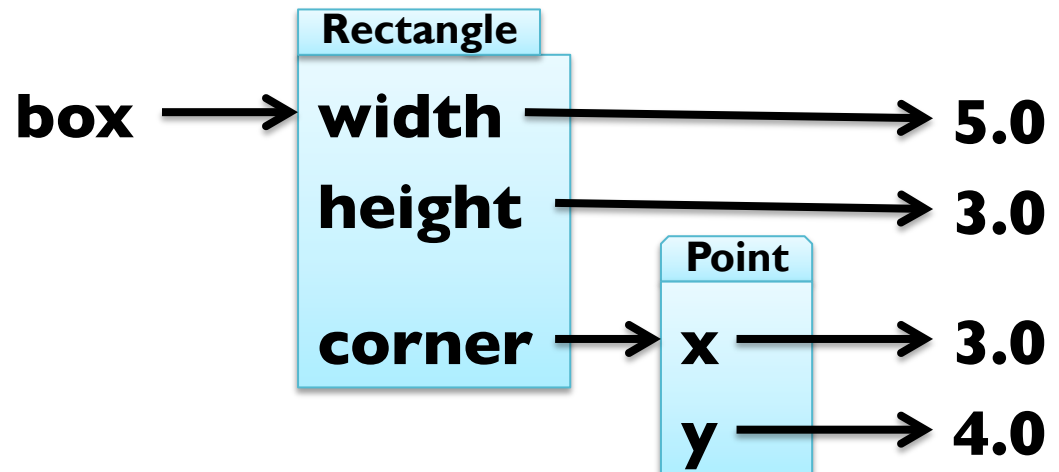
"represents a rectangle using attributes width, height, corner"

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

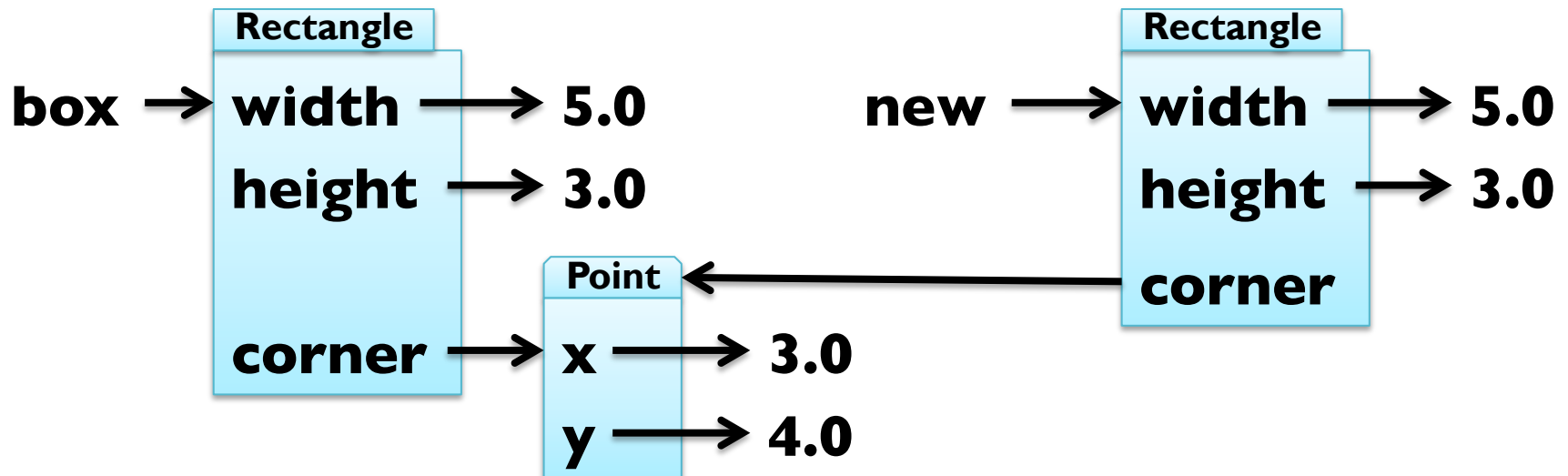
- by assigning to attributes, an object is changed
- Example: update size of rectangle

```
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!

CLASSES & FUNCTIONS

Representing Time

- Example: user-defined type for representing time

```
class Time(object):
```

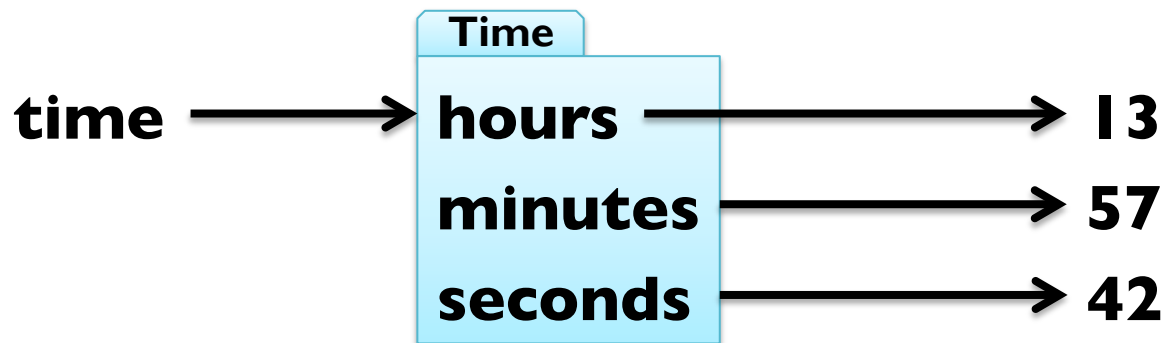
```
    """represents time of day using hours, minutes, seconds"""
```

```
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 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 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 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 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)]
```