Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Review of Lectures 16-24

- Lecture 16: Special methods, overloading
- Lecture 17: Class composition
- Lecture 18: Final programming assignment
- Lecture 19: Inheritance, base, derived class
- Lecture 20: Exception handling
- Lecture 21: String operations and methods
- Lecture 22: Regular expressions
- Lecture 23: File processing, I/O
- Lecture 24: File processing, scripting

Object-Oriented Programming

- Special methods for objects
  - Destructors
    - `__del__(self)`
  - Special method for object destruction
    - Called implicitly
      - whenever reference count to the object reaches zero
      - by the garbage collector
    - Typical situations when destructor is called:
      - the object is explicitly deleted by use of `del`
      - the object goes out of scope
      - a temporary object is no longer needed
      - the python interpreter exits
    - Allows for special clean-up procedures
    - Must not return any value (`None`)

Object-Oriented Programming

- Special methods for objects
  - String representation
    - built-in function `repr(object)`
      actually calls method `__repr__(self)`
      - formal representation from which the object can be replicated (if possible)
      - implicitly used in interactive mode to print results
      - Example:
        » `t1 = Time(); print repr(t1)`
    - string conversion function `str(object)`
      actually calls method `__str__(self)`
      - informal, human-readable string representation
      - implicitly used with `print`
      - Example:
        » `t1 = Time(); print str(t1)`

Operator Overloading

- What is operator overloading?
  - An operator performs different functions depending on the types of the arguments
- Some built-in operators are overloaded
  - Example: binary + operator
    - for integer operands, performs integer addition
    - for floating point operands, performs floating point addition
    - for string arguments, performs string concatenation
- Operator overloading for user-defined types
    - for class objects, most operators can be overloaded to perform a specific function defined by the class
    - special methods are defined for each operator that can be overloaded

Operator Overloading

- Mathematical operations
  - Unary operations
    - `+` `__pos__(self)` positive
    - `-` `__neg__(self)` negative
    - `~` `__invert__(self)` invert
    - `abs()` `__abs__(self)` absolute value
    - `int()` `__int__(self)` integer conversion
    - `float()` `__float__(self)` floating conversion
    - ...

Operator Overloading

- Mathematical operations
  - Binary operations
    - `+` `__add__(self, other)` addition
    - `-` `__sub__(self, other)` subtraction
    - `*` `__mul__(self, other)` multiplication
    - `/` `__div__(self, other)` division
    - `%` `__mod__(self, other)` modulo
    - `**` `__pow__(self, other)` power
    - `<<` `__lshift__(self, other)` left shift
    - `>>` `__rshift__(self, other)` right shift
    - `&` `__and__(self, other)` logical and
    - `|` `__or__(self, other)` logical or
    - `^` `__xor__(self, other)` logical xor
    - ...

Object-Oriented Programming

- Example: **`class Time`** (file **`time3.py`**)

```
class Time:
    """abstract data type for representation of time"""

    def __init__(self, hour=0, minute=0, second=0):
        """creates a time object and initializes it"""
        self.SetTime(hour, minute, second)

    def __del__(self):
        """cleans up and destroys a time object"""
        # nothing to do here, but let's print a message
        print "Time object being destroyed!"

    def __str__(self):
        """returns an informal string representation"""
        return "%02d:%02d:%02d" %     \
                (self.__hour,self.__minute,self.__second)

    def __repr__(self):
        """returns a formal string representation"""
        return "Time(%d,%d,%d)" %     \
                (self.__hour,self.__minute,self.__second)
...
```

Object-Oriented Programming

- Example: **`class Time`** (file **`time3.py`**)

```
...
    def __add__(self, other):
        """overloads the + operation for time objects"""
        return self.Add(other)
```

- Interactive Example:

```
% python
>>> from time3 import Time
>>> t1 = Time(10, 20)
>>> t2 = Time( 2, 11, 30)
>>> print t1, t2
10:20:00 02:11:30
>>> t3 = t1 + t2
>>> print t3
12:31:30
>>> del t1
Time object 10:20:00 being destroyed!
>>> ^d
Time object 12:31:30 being destroyed!
Time object 02:11:30 being destroyed!
%
```

Object-Oriented Programming

- Example:

```
from time3 import Time
t1 = Time(10, 20, 30)
t2 = Time( 2, 15)
t3 = t1 + t2
print t3
del t2
```

Global Namespace

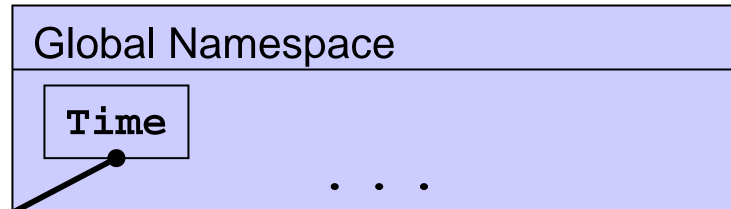
. . .

Data memory:

Object-Oriented Programming

- Example:

```
from time3 import Time
t1 = Time(10, 20, 30)
t2 = Time( 2, 15)
t3 = t1 + t2
print t3
del t2
```



1. compile file “time3.py”
2. create reference to class Time

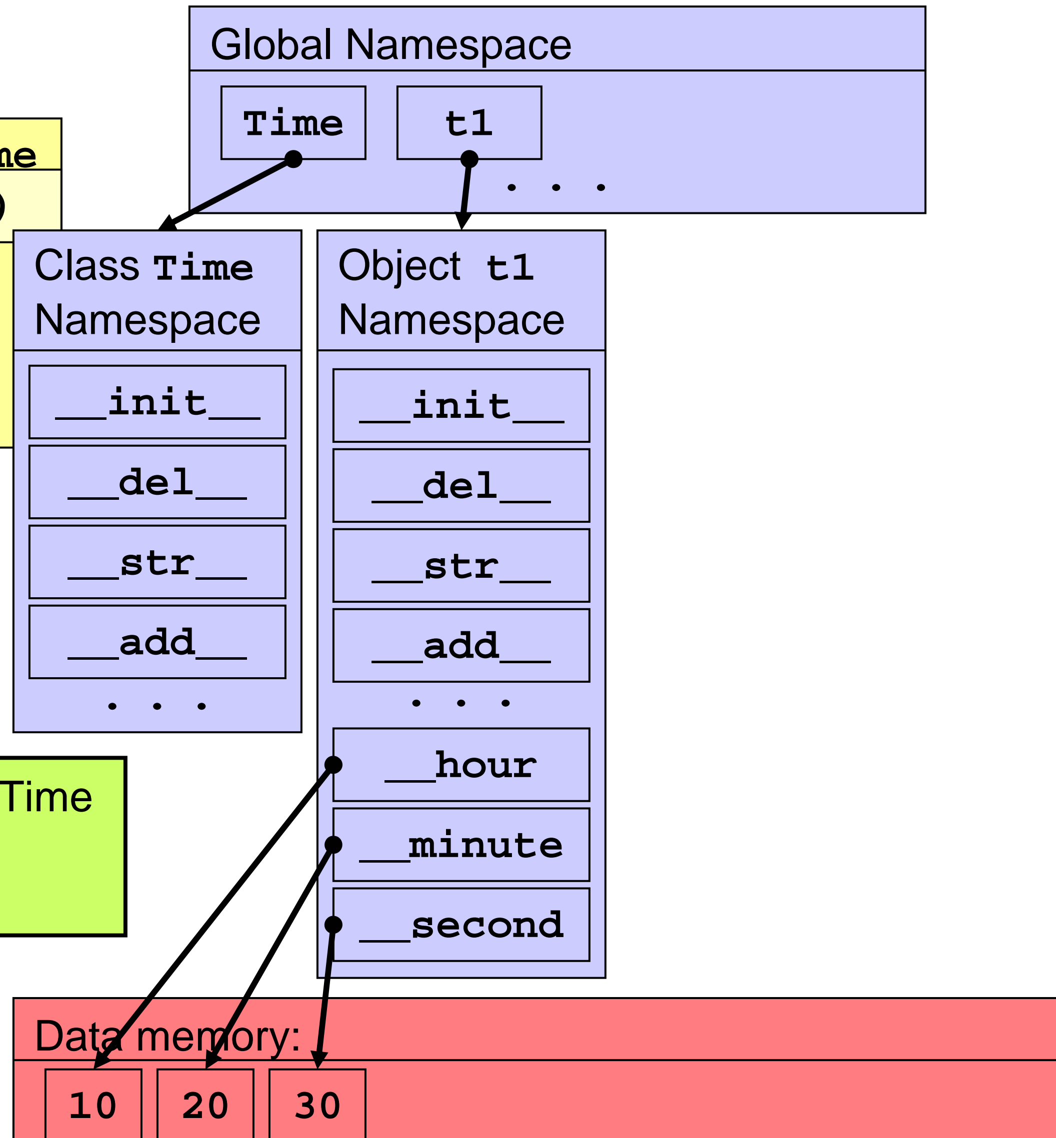
Data memory:

Object-Oriented Programming

- Example:

```
from time3 import Time
t1 = Time(10, 20, 30)
t2 = Time( 2, 15)
t3 = t1 + t2
print t3
del t2
```

1. tmp = new object of type Time
2. __init__(tmp, 10, 20, 30)
3. t1 = tmp

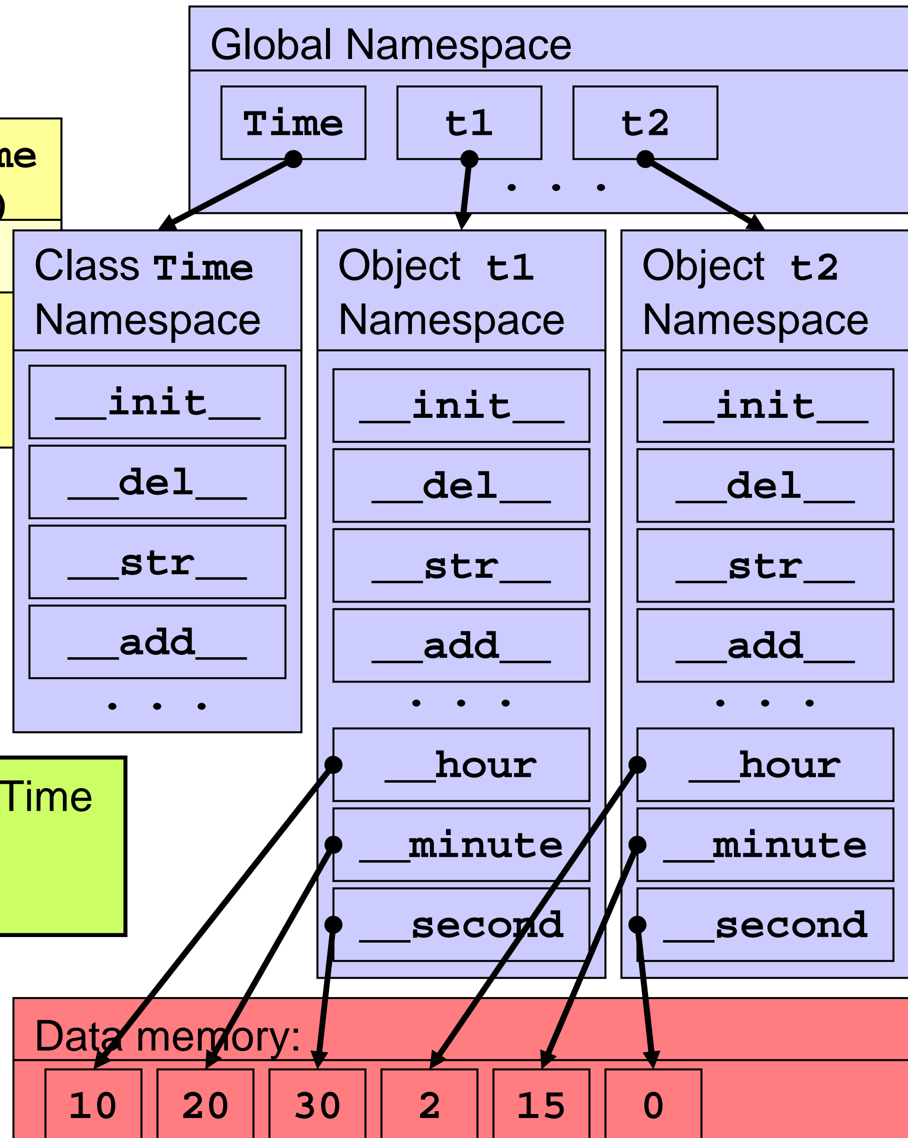


Object-Oriented Programming

- Example:

```
from time3 import Time
t1 = Time(10, 20, 30)
t2 = Time( 2, 15)
t3 = t1 + t2
print t3
del t2
```

1. tmp = new object of type Time
2. __init__(tmp, 2, 15, 0)
3. t2 = tmp

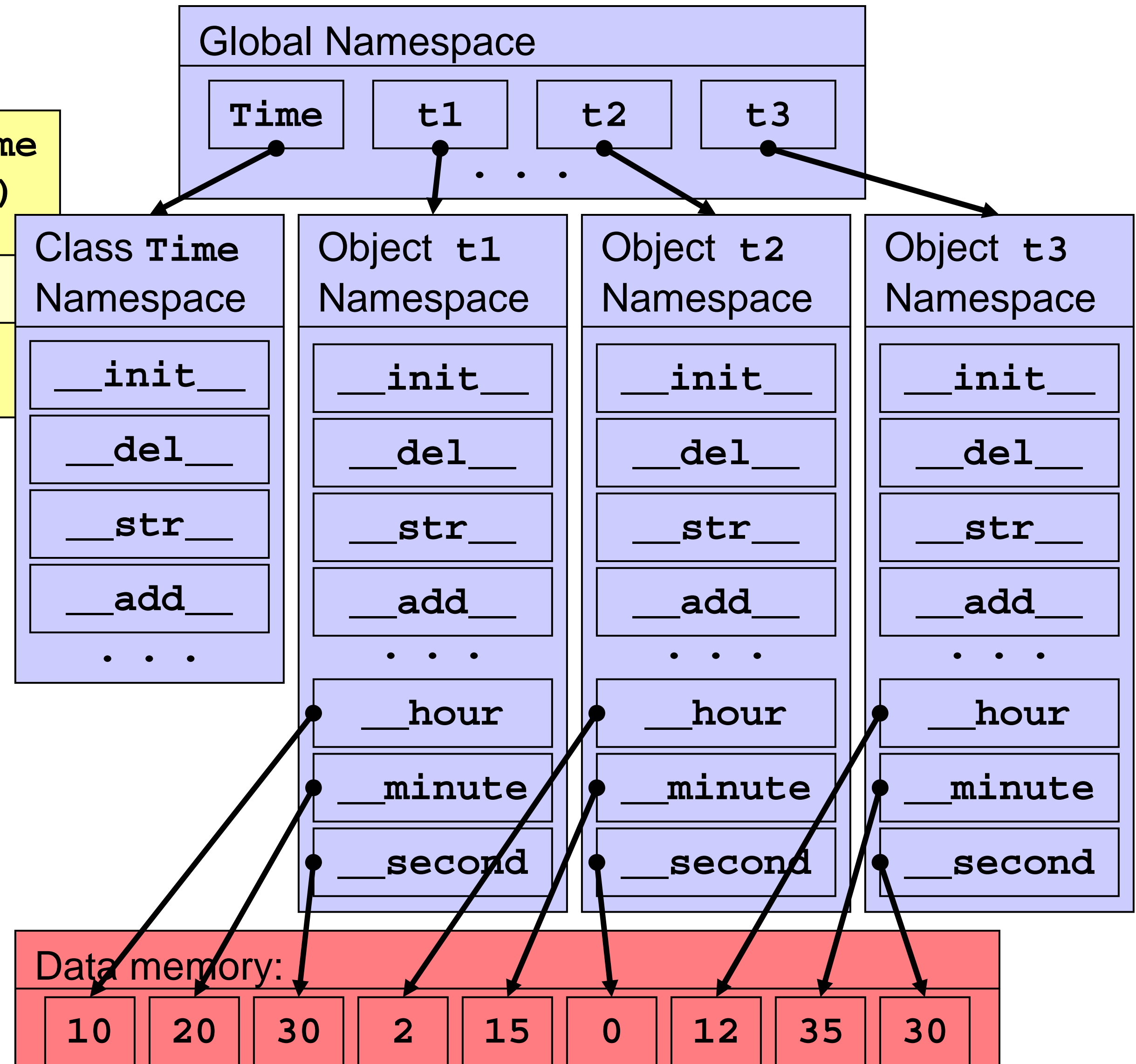


Object-Oriented Programming

- Example:

```
from time3 import Time
t1 = Time(10, 20, 30)
t2 = Time( 2, 15)
t3 = t1 + t2
print t3
del t2
```

1. tmp = __add__(t1, t2)
2. t3 = tmp

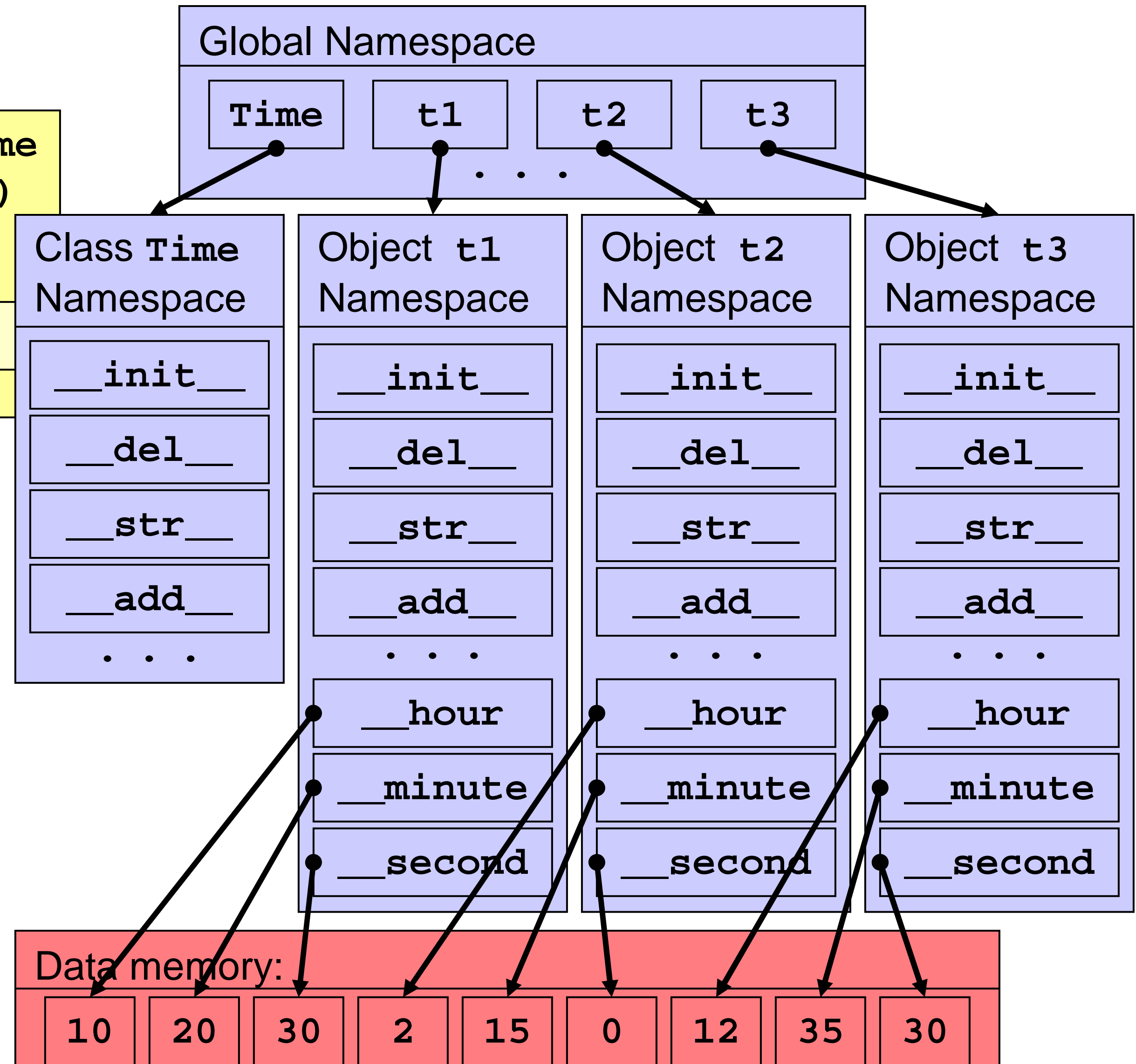


Object-Oriented Programming

- Example:

```
from time3 import Time
t1 = Time(10, 20, 30)
t2 = Time( 2, 15)
t3 = t1 + t2
print t3
del t2
```

1. tmp = __str__(t3)
2. print tmp

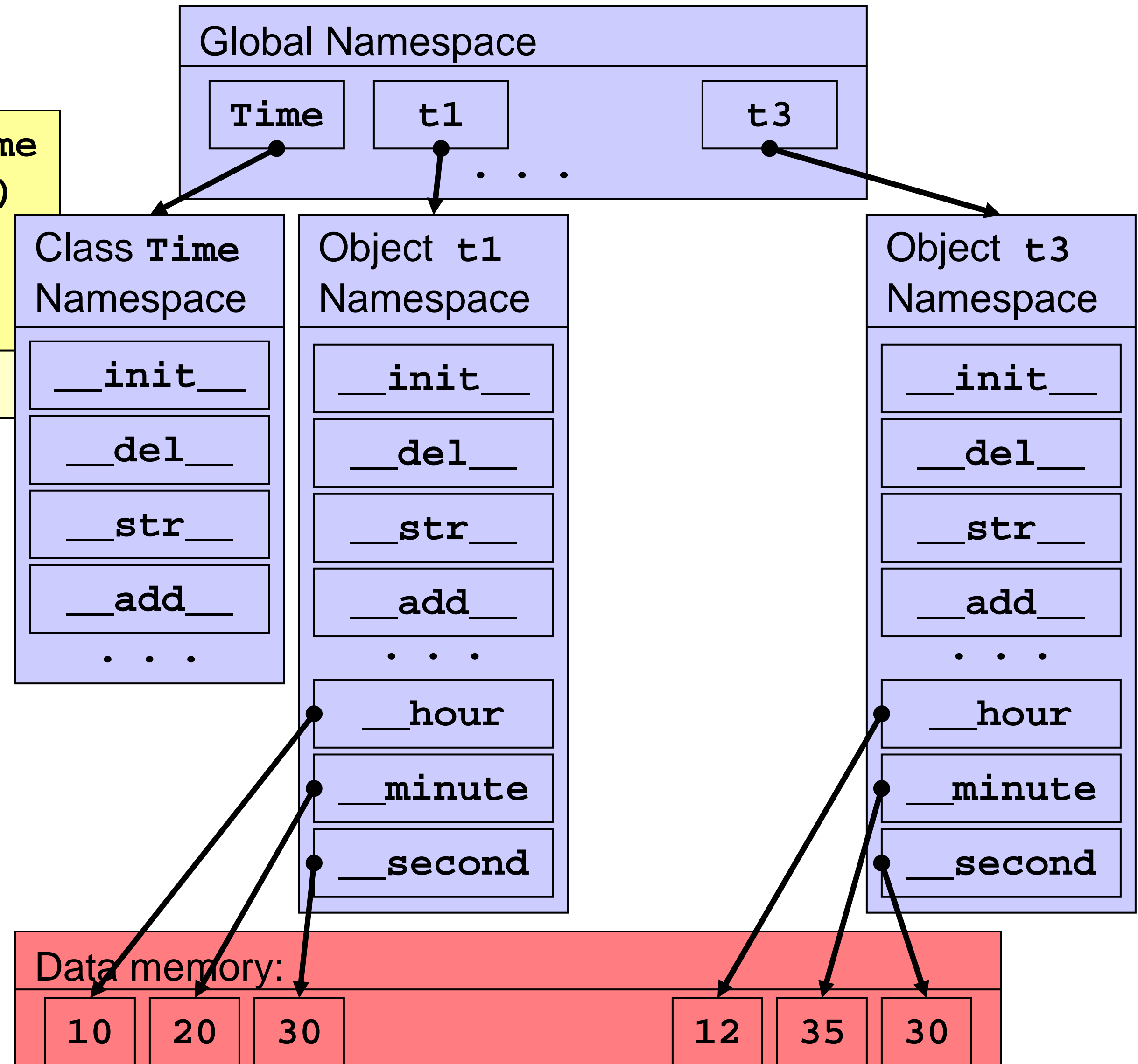


Object-Oriented Programming

- Example:

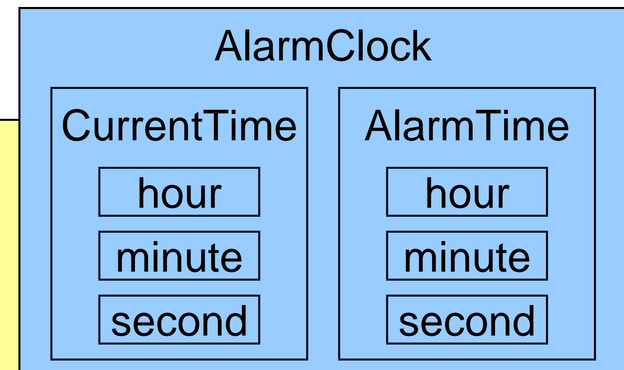
```
from time3 import Time
t1 = Time(10, 20, 30)
t2 = Time( 2, 15)
t3 = t1 + t2
print t3
del t2
```

1. __del__(t2)
2. garbage collection



Object-Oriented Programming

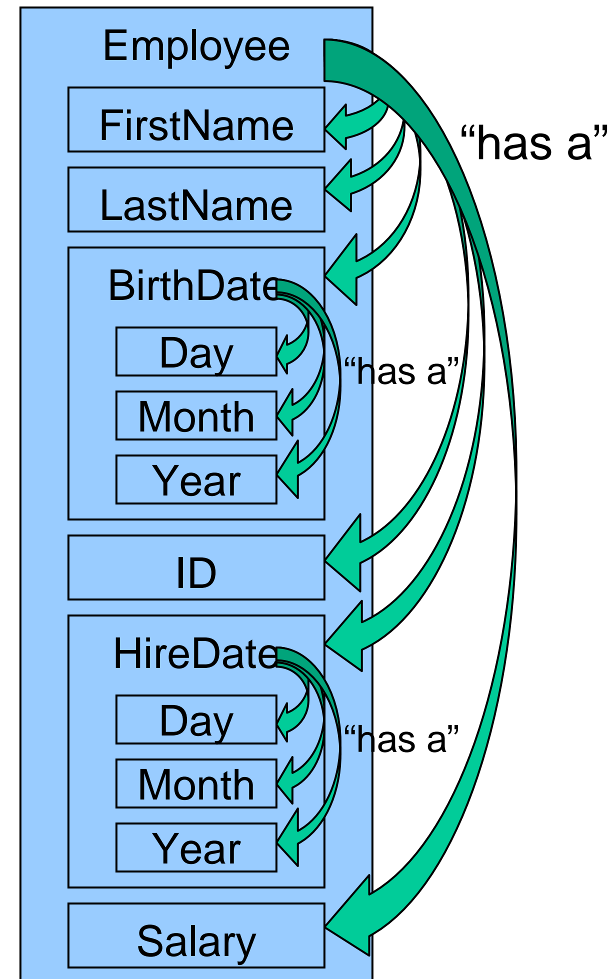
- Class Composition
  - Classes can be hierarchically organized in a containment relationship (membership)
  - Object references can be used as members
    - A class contains objects of other classes as members
  - Example: Alarm clock




```
class Time:
  def __init__(self,hour,minute,second)
    ...
class AlarmClock:
  def __init__(self,CurrentTime,AlarmTime)
    ...
```

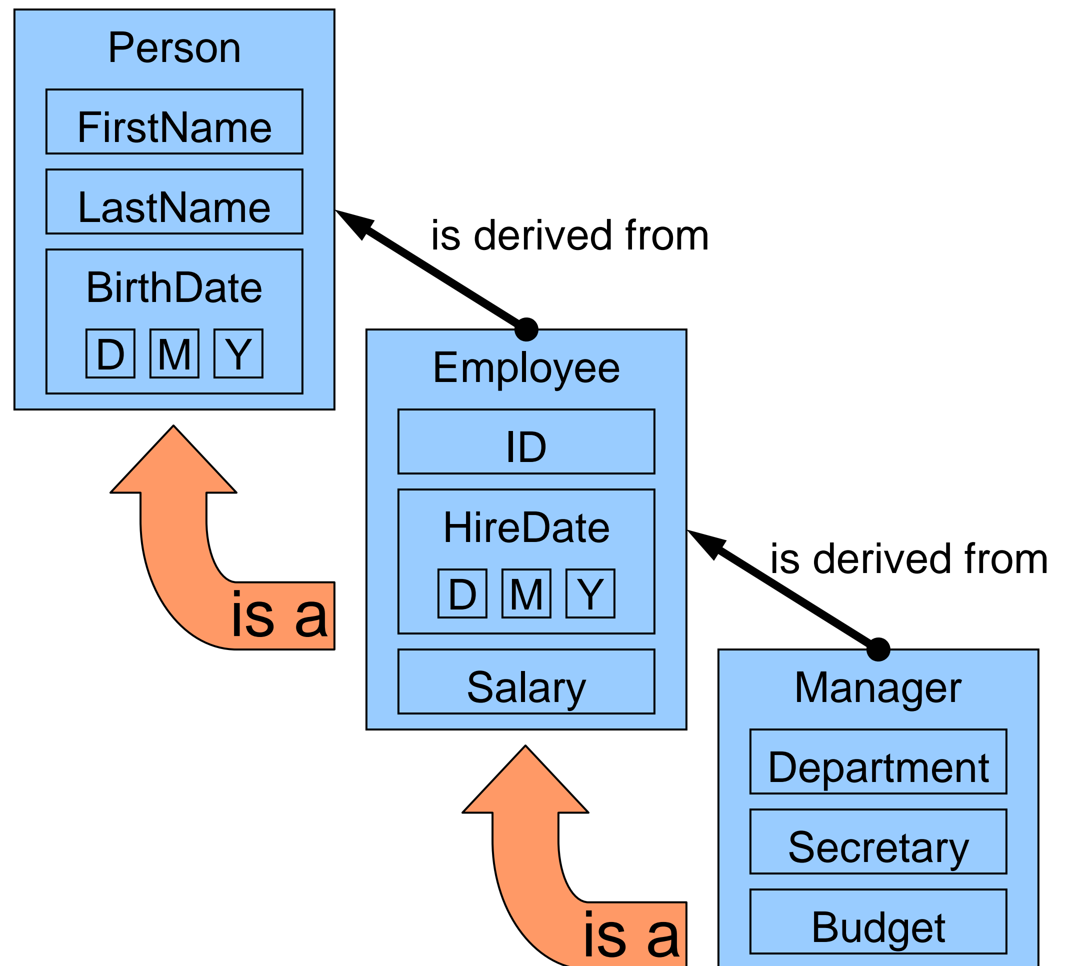
Object-Oriented Programming

- Membership composition
  - Example: Employee
    - class Employee:
      - String: FirstName
      - String: LastName
      - Date: BirthDate
        - » Integer: Year
        - » Integer: Month
        - » Integer: Day
      - Integer: ID
      - Date: HireDate
        - » Integer: Year
        - » Integer: Month
        - » Integer: Day
      - Floating point: Salary
  - "Has a" relationship



Object-Oriented Programming

- Inheritance
  - Example:
    - Person
    - Employee
    - Manager
  - "Is a" relationship
  - Base class
    - Person
  - Derived classes
    - Employee
    - Manager



Object-Oriented Programming

- Example: **`manager.py`** (part 1/5)

```
# manager.py: example for class inheritance
# author: Rainer Doemer
#
# modifications:
# 03/04/04 RD    initial version (based on employee.py)

# class definitions

class Date:
    """class for representation of a date"""
    def __init__(self, day, month, year):
        """creates a date object"""
        self.Day = day
        self.Month = month
        self.Year = year

    def __str__(self):
        """returns the date as printable string"""
        return "%d/%d/%d" % (self.Month, self.Day, self.Year)
...
```

Object-Oriented Programming

- Example: **`manager.py`** (part 2/5)

```
...

class Person:
    """class for representation of a person"""

    def __init__(self, FirstName, LastName, BirthDate):
        """creates a person object"""
        self.FirstName = FirstName
         self.LastName = LastName
         self.BirthDate = BirthDate

    def __str__(self):
        """returns the person data in printable format"""
        return "%s %s, born %s" % (self.FirstName, self.LastName,
                                   str(self.BirthDate))
...
```

Object-Oriented Programming

- Example: **`manager.py`** (part 3/5)

```
...

class Employee(Person):
    """class for representation of an employee"""

    def __init__(self, FirstName, LastName, BirthDate,
                          ID, HireDate, Salary):
        """creates an employee object"""
        Person.__init__(self, FirstName, LastName, BirthDate)
        self.ID = ID
        self.HireDate = HireDate
        self.Salary = Salary

    def __str__(self):
        """returns the employee data in printable format"""
        return "%s,\nID %d, hired %s, making $%d" %        \
                (Person.__str__(self), self.ID,
                        str(self.HireDate), self.Salary)

...
```

Object-Oriented Programming

- Example: **`manager.py`** (part 4/5)

```
...

class Manager(Employee):
    """class for representation of a manager"""

    def __init__(self, FirstName, LastName, BirthDate,
                      ID, HireDate, Salary,
                      Dept, Secretary, Budget):
        """creates a manager object"""
        Employee.__init__(self, FirstName, LastName, BirthDate,
                              ID, HireDate, Salary)
        self.Dept = Dept
        self.Secretary = Secretary
        self.Budget = Budget

    def __str__(self):
        """returns the manager data in printable format"""
        return "%s,\nDept. %s, Secretary %s, Budget $%d" % \
                    (Employee.__str__(self), self.Dept,
                            self.Secretary, self.Budget)
...
```

Object-Oriented Programming

- Example: **`manager.py`** (part 5/5)

```
...

# driver program

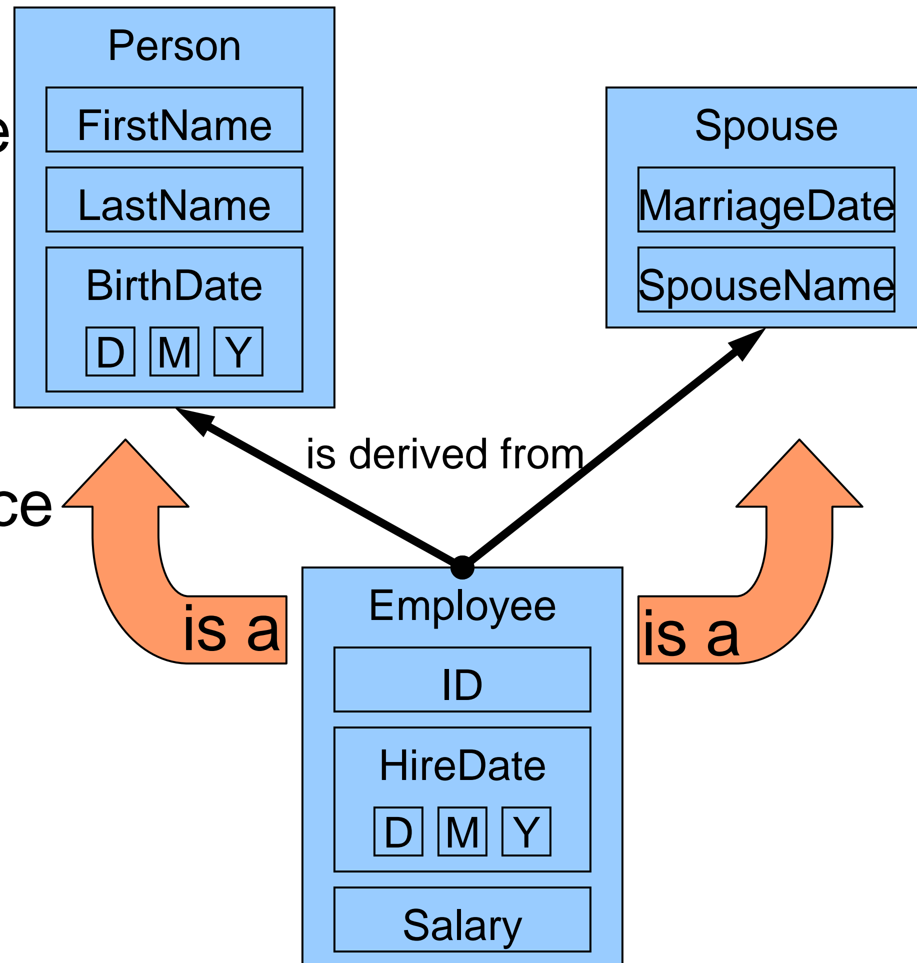
if __name__ == "__main__":
    e1 = Employee("John", "Smith", Date(23, 2, 1968),
                      1001, Date(1, 1, 2003), 40000)
    e2 = Employee("Jeff", "Somebody", Date(31, 12, 1969),
                      1002, Date(1, 2, 2003), 35000)
    p1 = Person("Jane", "Doe", Date(30, 6, 1971))
    p2 = Person("Joe", "Average", Date(30, 6, 1970))
    m1 = Manager("Jenny", "X", Date(15, 1, 1959),
                      1003, Date(1, 1, 2000), 55000,
                      "Imports", "Jeff", 1000000)
    m2 = Manager("Jonathan", "Y", Date(16, 2, 1958),
                      1004, Date(1, 1, 2000), 55000,
                      "Exports", "Jane", 1500000)
    for p in [e1, e2, p1, p2, m1, m2]:
        print p
```

Object-Oriented Programming

- Inheritance
  - Overriding methods
    - Method **`__init__`** of class **`Person`** is overwritten by method **`__init__`** of class **`Employee`**
    - Method **`__str__`** of class **`Person`** is overwritten by method **`__str__`** of class **`Employee`**
    - etc.
  - Polymorphism
    - An object behaves differently depending on its type
      - A person is printed with name and birthday
      - An employee is printed as person, plus ID, hire date, and salary
      - A manager is printed as an employee, plus department, secretary and budget

Object-Oriented Programming

- Inheritance
  - Single inheritance
    - Employee is a Person
  - Multiple inheritance
    - Employee is a Person and a Spouse



Exception Handling

- Introduction
  - Exception handling is required in situations where some code detects an error condition and is unable to handle it
    - an exception is *raised* or *thrown*
  - Exceptions may be caught and handled by exception handlers to take appropriate actions
    - an exception is *caught* and *handled*
  - Any exception that is not caught and handled will terminate the program execution with an error message
    - Abort with error message is the default exception handler
- Examples
  - Invalid arguments to arithmetic operations
    - Division by zero, square root of negative numbers, ...
  - Input/Output errors, file read/write error, disk full, ...
  - etc. etc.

Exception Handling

- Raising Exceptions
  - Implicit exceptions raised by built-in functions
    - Built-in operators and functions can raise exceptions
  - Explicit exceptions raised by programmer
    - **`raise`** statement
      - Syntax:
        - » **`raise`** *ExceptionClass*, *ExceptionArgument*
        - » *ExceptionClass* is predefined or user-defined
        - » *ExceptionArgument* is optional
      - Example:

```
def SetMinute(self, minute=0):
    """sets the minute of a time object"""
    if (0 <= minute <= 59):
        self.minute = minute
    else:
        raise ValueError, "Minute value out of range 0-59"
```

Exception Handling

- Raising Exceptions
  - Assertions
    - **`assert`** statement
      - Syntax:
        - » **`assert`** *Test*, *ErrorMessage*
        - » *Test* is a condition that is expected to evaluate to true; if it evaluates to false, an assertion is raised
        - » *ErrorMessage* is optional
      - Example:

```
def CircleArea(radius):
    """computes the area of a circle"""
    assert radius >= 0, "Expected non-negative radius!"
    return 3.14159 * radius * radius
```

      - Assertions are debugging statements
        - » executed only if **`__debug__`** is true
        - » ignored if optimization (-o) is turned on

Exception Handling

- Python Exception Classes (part 1/2)
  - Exception — Exception base class
    - StandardError — Error base class
      - ArithmeticError — Arithmetic exceptions
        - » FloatingPointError — Failure of floating-point operation
        - » OverflowError — Arithmetic overflow
        - » ZeroDivisionError — Division or modulus by zero
      - AssertionError — `assert` statement failure
      - AttributeError — Invalid class/object attribute
      - EnvironmentError — External errors
        - » IOError — Input/output error
        - » OSError — Operating system error
      - EOFError — End-of-file reached
      - ImportError — `import` statement failure
      - KeyboardInterrupt — CTRL-C!
      - LookupError — Indexing and key errors
        - » IndexError — Index out of range
        - » KeyError — Non-existing dictionary key
      - ...

Exception Handling

- Python Exception Classes (part 2/2)
  - Exception
    - StandardError — Error base class
      - ...
      - MemoryError — Out of memory
      - NameError — Unknown identifier
      - RuntimeError — Generic error
      - SyntaxError — Parsing error
      - SystemError — Error in the interpreter
      - SystemExit — generated by `sys.exit()`
      - TypeError — Invalid type for operation
      - ValueError — Invalid value
    - Warning — Warning base class
      - UserWarning — Default warning
      - DeprecationWarning — Deprecated feature
      - SyntaxWarning — Dubious syntactic feature
      - FutureWarning — Language feature change
      - RuntimeWarning — Dubious runtime feature

Exception Handling

- Catching and Handling Exceptions
  - **`try`** - **`except`** statement
    - Syntax:
      - ```
        try:
            # do something
        except ExceptionClass, ExceptionArgument:
            # handle the exception
        except ...
            # handle more exceptions
        else:
            # do this if no exception occurred
        ```
      - multiple **`except`** clauses are allowed
      - one optional **`else`** clause is allowed
      - *ExceptionClass* is predefined or user-defined
      - *ExceptionArgument* is optional
        ```

Exception Handling

- Catching and Handling Exceptions
  - **try** - **except** statement
    - Example:

```
try:
    # input
    radius = int(raw_input("Enter the radius: "))
    # compute
    area = CircleArea(radius)
    # output
    print "The circle area is", area
except AssertionError, message:
    print message
except ValueError, message:
    print "Invalid value:", message
except KeyboardInterrupt:
    print "CTRL-C!"
except EOFError:
    print "End-of-file error! No input given!"
```

Exception Handling

- Creating User-defined Exceptions
  - Class Exception can be extended by inheritance
    - Example:

```
from exceptions import Exception

class RadiusError(Exception):
    def __init__(self, message):
        self.message = message

def CircleArea(radius):
    """computes the area of a circle"""
    if radius < 0:
         raise RadiusError, "Expected non-negative radius!"
    return 3.14159 * radius * radius

try:
    radius = int(raw_input("Please enter the radius: "))
    area = CircleArea(radius)
    print "The circle area is", area
except RadiusError, ErrorObject:
    print "Radius error:", ErrorObject.message
```

String Manipulation

- String Operations (revisited)
  - `len(s)`
    - returns the length of the string `s`
  - `s[index]`
    - returns a character of the string at `index`
  - `s[left:right]`
    - returns a slice of the string from index `left` to `right`
  - `s1 + s2`
    - returns the concatenation of `s1` and `s2`
  - `format % tuple`
    - returns the string `format` with `%`-sequences replaced by formatted arguments from `tuple`

String Manipulation

- String Methods
  - **`upper()`**
    - returns the string converted to upper-case
      - **`"Test String".upper()`** returns **`"TEST STRING"`**
  - **`lower()`**
    - returns the string converted to lower-case
      - **`"Test String".lower()`** returns **`"test string"`**
  - **`swapcase()`**
    - returns the string with upper- and lower-case exchanged
      - **`"Test String".swapcase()`** returns **`"tEST sTRING"`**

String Manipulation

- String Methods
  - **`capitalize()`**
    - returns the string with capitalized beginning
      - **`"test string".capitalize()`** returns **`"Test string"`**
  - **`title()`**
    - returns the string with every word capitalized
      - **`"test string".title()`** returns **`"Test String"`**

String Manipulation

- String Methods
  - **`ljust(width)`**
    - returns a string left-justified in **`width`** characters
      - `"test".ljust(10)` returns
        `"test      "`
  - **`rjust(width)`**
    - returns a string right-justified in **`width`** characters
      - `"test".rjust(10)` returns
        `"      test"`
  - **`center(width)`**
    - returns a string centered in **`width`** characters
      - `"test".center(10)` returns
        `"   test   "`

String Manipulation

- String Methods
  - **`lstrip()`**
    - returns the string with any leading white space removed
      - **`" \tTest string \t".lstrip()`** returns **`"Test string \t"`**
  - **`rstrip()`**
    - returns the string with any trailing white space removed
      - **`" \tTest string \t".rstrip()`** returns **`" \tTest string"`**
  - **`strip()`**
    - returns the string with any leading and trailing white space removed
      - **`" \tTest string \t".strip()`** returns **`"Test string"`**

String Manipulation

- String Methods
  - **isalpha()**
    - returns true if the string consists of only alphabetic characters
      - `"Test".isalpha()` returns `1`
      - `"test string".isalpha()` returns `0`
  - **isdigit()**
    - returns true if the string consists of only digits
      - `“123".isdigit()` returns `1`
      - `“1.23".isdigit()` returns `0`
  - **isalnum()**
    - returns true if the string consists of only alphanumeric characters
      - `"Test123".isalnum()` returns `1`
      - `"test string".isalnum()` returns `0`
  - **isspace()**
    - returns true if the string consists of only white space characters
      - `" \t \t".isspace()` returns `1`
      - `"Test string".isspace()` returns `0`

String Manipulation

- String Methods
  - **`islower()`**
    - returns true if the string consists of only lower-case characters
      - **`"test".islower()`** returns **`1`**
      - **`"Test".islower()`** returns **`0`**
  - **`isupper()`**
    - returns true if the string consists of only upper-case characters
      - **`"TEST".isupper()`** returns **`1`**
      - **`"Test".isupper()`** returns **`0`**
  - **`istitle()`**
    - returns true if each word in the string is capitalized
      - **`"Test String".istitle()`** returns **`1`**
      - **`"Test string".istitle()`** returns **`0`**

String Manipulation

- String Methods
  - **`find(substring, start, end)`**
    - returns position where **`substring`** is found in the string, or -1, if **`substring`** is not found
    - **`start`** and **`end`** are optional search indices
      - **`"test string".find("st")`** returns **`2`**
      - **`"test string".find("ST")`** returns **`-1`**
  - **`rfind(substring, start, end)`**
    - returns position from the end where **`substring`** is found in the string, or -1, if **`substring`** is not found
    - **`start`** and **`end`** are optional search indices
      - **`"test string".rfind("st")`** returns **`5`**
      - **`"test string".rfind("ST")`** returns **`-1`**

String Manipulation

- String Methods
  - **`index(substring, start, end)`**
    - returns position where **`substring`** is found in the string, or raises **`ValueError`** if **`substring`** is not found
    - **`start`** and **`end`** are optional search indices
      - **`"test string".index("st")`** returns **`2`**
      - **`"test string".index("st", 4,-2)`** returns **`5`**
      - **`"test string".index("x")`** raises **`ValueError`**
  - **`rindex(substring, start, end)`**
    - returns position from the end where **`substring`** is found in the string, or raises **`ValueError`** if **`substring`** is not found
    - **`start`** and **`end`** are optional search indices
      - **`"test string".rindex("st")`** returns **`5`**
      - **`"test string".rindex("st", 4,-2)`** returns **`5`**
      - **`"test string".rindex("x")`** raises **`ValueError`**

String Manipulation

- String Methods
  - **`count(substring, start, end)`**
    - returns number of times **`substring`** is found in a string
    - **`start`** and **`end`** are optional search indices
      - **`"test string".count("st")`** returns **2**
      - **`"test string".count("st", 4,-2)`** returns **1**
  - **`startswith(substring, start, end)`**
    - returns if string starts with **`substring`**
    - **`start`** and **`end`** are optional search indices
      - **`"test string".startswith("te")`** returns **1**
      - **`"test string".startswith("st")`** returns **0**
  - **`endswith(substring, start, end)`**
    - returns if string ends with **`substring`**
    - **`start`** and **`end`** are optional search indices
      - **`"test string".endswith("ing")`** returns **1**
      - **`"test string".endswith("st")`** returns **0**

String Manipulation

- String Methods
  - **`replace(old, new, max)`**
    - returns the string with all occurrences of **`old`** replaced by **`new`**
    - **`max`** optionally indicates maximum number of replacements
      - **`"test string".replace("st","X")`** returns **`"teX Xring"`**
      - **`"test string".replace("st","X",1)`** returns **`"teX string"`**
  - **`expandtabs(t)`**
    - returns a string where tabs are expanded to **`t`** spaces
    - tabulator size **`t`** is optional (defaults to **`8`**)
      - **`"\ttest string".expandtabs()`** returns **`"        test string"`**

String Manipulation

- String Methods
  - **`split(separator)`**
    - returns a list of strings created by splitting the string at **`separator`** strings found
    - **`separator`** is optional and defaults to white space
      - **`"a, b, c".split(", ")`** returns **`["a","b","c"]`**
      - **`"a, b, c".split()`** returns **`["a,","b,","c"]`**
  - **`splitlines()`**
    - returns a list of strings created by splitting the string at new line characters
      - **`"a\nb\nc".splitlines()`** returns **`["a","b","c"]`**
  - **`join(sequence)`**
    - returns a string concatenated of strings in the **`sequence`** using the original string as separator
      - **`", ".join(["a","b","c"])`** returns **`"a, b, c"`**

String Manipulation

- Example: **`name.py`**
  - Given a string with a full name, such as
    - **`"George W Bush"`**, or
    - **`"   george W      BUSH   "`**,
  - write the name in different common formats, as follows:

```
Please enter your full name:
  george W   BUSH
Your first name is          George
Your middle name is         W
Your last name is           Bush
Your full name is           George W Bush
Your initials are           GWB
Your email address is       george_bush@uci.edu
Suggested user IDs are      bush or gbush or georgeb
```

String Manipulation

- Example: **`name.py`** (part 1/3)

```
# name.py: example for string manipulation
# author: Rainer Doemer
# 03/10/04 RD  initial version

# input
name = raw_input("Please enter your full name:\n")

# compute
name = name.strip()
list = name.split()
if len(list) == 2:
    first = list[0].capitalize()
    middle = ""
    last = list[1].capitalize()
else:
    first = list[0].capitalize()
    middle = list[1].capitalize()
    last = list[-1].capitalize()
...
```

String Manipulation

- Example: **`name.py`** (part 2/3)

```
...

if middle:
    full = "%s %s %s" % (first,middle,last)
    initial = first[0] + middle[0] + last[0]
else:
    full = "%s %s" % (first,last)
    initial = first[0] + last[0]
email = first.lower() + "_" + last.lower() + "@uci.edu"
id1 = last.lower()
id2 = first.lower()[0] + last.lower()
id3 = first.lower() + last.lower()[0]
ids = [id1,id2,id3]
for i in range(len(ids)):
    if len(ids[i]) > 8:
        ids[i] = ids[i][:8]

...
```

String Manipulation

- Example: **`name.py`** (part 3/3)

```
...

# output
print "Your first name is".ljust(25), first
print "Your middle name is".ljust(25), middle
print "Your last name is".ljust(25), last
print "Your full name is".ljust(25), full
print "Your initials are".ljust(25), initial
print "Your email address is".ljust(25), email
print "Suggested user IDs are".ljust(25), " or ".join(ids)
```

String Manipulation

- String Methods to Search and Replace Text
  - Example
    - Text
      - **"The quick brown fox jumps over the lazy dog."**
    - Tasks
      - Find the **"dog"**
      - Find the **"cat"**
      - Replace the **"dog"** by a **"cat"**
    - Interactive Python program

```
% python
>>> text = "The quick brown fox jumps over the lazy dog."
>>> text.find("dog")
40
>>> text.find("cat")
-1
>>> text.replace("dog", "cat")
'The quick brown fox jumps over the lazy cat.'
```

String Manipulation

- String Methods to Search and Replace Text
  - Example
    - Text
      - **`"The quick brown fox jumps over the lazy dog."`**
    - Advanced Tasks
      - Find all lower-case words
      - Find all three-letter words
      - Find all three-letter words with an **`"o"`** in the middle
      - Find the first **`"dog"`**, **`"cat"`**, or **`"fox"`**
      - Find any **`"quick" "fox"`**
      - Replace any **`"dog"`**, **`"cat"`**, or **`"fox"`** by an **`"animal"`**
      - Replace any **`"the"`** by an **`"a"`**, ignoring the case
      - etc.

➢ Possible, but difficult and cumbersome with string methods!

String Manipulation

- Regular Expressions
  - Introduction
    - Regular expressions describe a set of strings by use of string patterns
    - Regular expressions are a powerful tool for searching and replacing text
    - The module **`re`** provides regular expressions in Python
  - Example
    - The pattern **`"ab*c"`** matches all strings that
      - start with **`"a"`**,
      - followed by zero or more **`"b"`**,
      - and end with **`"c"`**
    - such as
      - **`"abc"`**, **`"ac"`**, or **`"abbbbbc"`**
    - but not
      - **`"Abc"`**, **`"aabc"`**, nor **`"cba"`**

String Manipulation

- Regular Expressions
  - Composing Regular Expressions
    - Standard characters
      - **"abc"** matches only **"abc"**
    - Meta characters
      - **"."** matches any character except newline **"\n"**
      - **"x?"** matches zero or one occurrence of **"x"**
      - **"x+"** matches one or more occurrences of **"x"**
      - **"x*"** matches zero or more occurrences of **"x"**
      - **"^x"** matches **"x"** only at the beginning of the string
      - **"x$"** matches **"x"** only at the end of the string
      - **"x|y"** matches the occurrence of **"x"** or **"y"**
      - **"x{n}"** matches the occurrence of **"x" n** times
    - Character classes
      - **"[abc]"** matches the character **"a"**, **"b"**, or **"c"**
      - **"[a-d]"** matches the characters **"a"** through **"d"**
      - **"[^abc]"** matches any character except **"a"**, **"b"**, **"c"**

String Manipulation

- Regular Expressions
  - Composing Regular Expressions
    - Escape sequences (using *raw strings*)
      - **`r“\d”`** matches any digit (same as **`“[0-9]”`**)
      - **`r“\D”`** matches any non-digit (same as **`“[^0-9]”`**)
      - **`r“\s”`** matches any white space (same as **`“[ \n\f\r\t\v]”`**)
      - **`r“\S”`** matches any non-white space (same as **`“[^ \n\f\r\t\v]”`**)
      - **`r“\w”`** matches any alphanumeric word (same as **`“[a-zA-Z0-9_]”`**)
      - **`r“\w”`** matches any non-alphanumeric word (same as **`“[^a-zA-Z0-9_]”`**)
      - **`r“\\”`** matches the backslash character

String Manipulation

- Regular Expressions
  - Examples
    - all lower-case word
      - **r" [a-z]+ "**
    - three letter word
      - **r" [a-zA-Z]{3} "**
    - three letter word with **"o"** in the middle
      - **r" [a-zA-Z]o[a-zA-Z] "**
    - **"dog"**, **"cat"** or **"fox"**
      - **r"dog|cat|fox"**
    - any quick fox
      - **r"quick.*fox"**
    - the word **"the"** in any case
      - **r"[tT][hH][eE]"**

String Manipulation

- Regular Expressions
  - Module **`re`** provides regular expression functions
    - **`search(regexp, string)`**
      - searches for an occurence of **`regexp`** in **`string`**
      - returns a match object, or **`None`** if no match is found
      - the match object provides methods
        - » **`group()`** the substring that matched the pattern
        - » **`start()`** the index where the match starts
        - » **`span()`** a tuple (start, stop) of the match
    - **`match(regexp, string)`**
      - matches **`regexp`** against the beginning of **`string`**
      - returns a match object, or **`None`** if no match is found
    - **`findall(regexp, string)`**
      - returns a list of substrings of **`string`** that match **`regexp`**

String Manipulation

- Regular Expressions
  - Module **`re`** provides regular expression functions
    - **`sub(regexp, replacement, string)`**
      - substitutes all occurrences of **`regexp`** in **`string`** by **`replacement`**
      - returns the new string
    - **`subn(regexp, replacement, string)`**
      - substitutes all occurrences of **`regexp`** in **`string`** by **`replacement`**
      - returns a tuple **`(s,n)`** with the new string **`s`** and the number of substitutions **`n`**
    - **`compile(regexp)`**
      - pre-compiles the **`regexp`** into a search pattern object for better performance
      - returns a pattern object that can be used as **`regexp`** in above functions

String Manipulation

- Regular Expressions
  - Interactive example

```
% python
>>> import re
>>> text = "The quick brown fox jumps over the lazy dog."
>>> re.findall(r" [a-z]+", text)
[' quick',' brown',' fox',' jumps',' over',' the',' lazy',' dog']
>>> re.findall(r" [a-z]{3}[ \.]", text)
[' fox ', ' the ', ' dog.']
>>> re.findall(r" [a-z]o[a-z][ \.]", text)
[' fox ', ' dog.']
>>> re.search(r"dog|cat|fox", text).group()
'fox'
>>> re.search(r"quick.*fox", text).group()
'quick brown fox'
>>> re.sub(r"dog|cat|fox", "animal", text)
'The quick brown animal jumps over the lazy animal.'
>>> re.sub(r"[Tt][Hh][Ee]", "a", text)
'a quick brown fox jumps over a lazy dog.'
```

File Processing

- Introduction
  - Persistent data
    - Up to now, all data processed was available only during program run time; when the program terminates, all data was lost
    - *Persistent data* is stored even after a program exits
    - Persistent data is stored in files
      - on the harddisk
      - on a removable disk (floppy disk, etc.)
      - on a tape, ...
  - File I/O
    - Input/output operations on *file streams*
      - Creating files
      - Writing data to files
      - Reading data from files
      - Modifying data in files

File Processing

- File I/O Functions
  - built-in function **`open`** creates a file object (stream)
    - **`open(filename, "r")`**
      - opens an existing file named **`filename`** for input (read)
      - returns a **`file`** object with file position at the beginning, or raises **`IOError`** (i.e. the file could not be found)
    - **`open(filename, "w")`**
      - creates a new file named **`filename`** for output (write)
      - returns a **`file`** object with file position at the beginning, or raises **`IOError`** (i.e. the file could not be created)
    - **`open(filename, "a")`**
      - opens an existing file named **`filename`** for extension (append)
      - returns a **`file`** object with file position at the end, or raises **`IOError`** (i.e. the file could not be found)

File Processing

- Methods provided by a file object (stream)
  - **`read(size)`**
    - reads data from the file and returns it as a string
    - optionally **`size`** specifies the maximum number of bytes to be read
    - returns an empty string at the end of the file
  - **`readline(size)`**
    - reads a line of text from the file and returns it as a string
    - optionally **`size`** specifies the maximum number of bytes to be read
    - returns an empty string at the end of the file
  - **`readlines(size)`**
    - reads all lines of text from the file and returns it as a list of strings
    - optionally **`size`** specifies the maximum number of bytes to be read
  - **`write(data)`**
    - writes the string **`data`** to the file
  - **`writelines(list)`**
    - writes each string in the **`list`** to the file

File Processing

- Methods provided by a file object (stream)
  - **`flush()`**
    - flushes the stream buffer
    - completes a read/write operation
  - **`close()`**
    - closes the stream (and releases the file resource)
  - **`fileno()`**
    - returns the file descriptor (an integer)
  - **`isatty()`**
    - returns `1` if the stream is an interactive terminal, otherwise `0` (standard disk files)
  - **`seek(offset)`**
    - moves the file position to the specified **`offset`** in bytes
  - **`tell()`**
    - returns the current file position in bytes
  - **`truncate(size)`**
    - truncates the file to the specified **`size`**
    - **`size`** is optional; if not specified, the file is truncated at position zero, so it will be empty

File Processing

- Module **`sys`** (system environment)
  - Standard I/O streams (opened by the environment)
    - **`sys.stdin`** standard input stream (read access)
    - **`sys.stdout`** standard output stream (write access)
    - **`sys.stderr`** standard error stream (write access)
  - Standard functions
    - **`sys.exit(value)`**
      - terminates the program execution
      - returns result **`value`** to the calling shell
        - » return **`value 0`** indicates normal program exit
        - » return **`value>0`** indicates exit due to an error condition
  - Command line arguments
    - **`sys.argv`** list of command line arguments (strings)
      - **`sys.argv[0]`** program name (always given)
      - **`sys.argv[1]`** first argument (if any)
      - **`sys.argv[2]`** second argument (if any), ...

File Processing

- Python as a scripting language
  - Any Python program can be seen as
    - a special Unix *shell* (the Python interpreter)
    - that executes a *shell script* (the Python program)
  - Example:
    - **`sh script.sh`** runs **`script.sh`** in **`sh`**
    - **`python program.py`** runs **`program.py`** in **`python`**
  - Automatic script execution
    - If the *first line* in the script is a valid *pseudo comment*, the Unix operating system automatically calls the specified shell for executing the script
    - Example:
      - **`#!/usr/bin/python`** calls **`python`** directly
      - **`#!/usr/bin/env python`** calls **`env`** which then calls **`python`**
    - The script essentially becomes a *command* !

File Processing

- Example: **`print.py`** (part 1/2)

```
#!/usr/bin/env python
#
# print.py: command to print the contents of a file
# author: Rainer Doemer
# 03/14/04 RD    initial version

import sys

# parse command line arguments
if len(sys.argv) != 2:
    sys.stderr.write("Usage: %s <file_name>\n"    \
                     % sys.argv[0])
    sys.exit(10)
filename = sys.argv[1]

# open the file for reading
try:
    file = open(filename, "r")
except IOError, message:
    print "Cannot open file %s for reading: %s"   \
                     % (filename,message)
    sys.exit(10)
...
```

File Processing

- Example: **`print.py`** (part 2/2)

```
...

# read the file
lines = file.readlines()

# close the file
file.close()

# output the contents page by page
lineno = 0
pageno = 0
pages = len(lines) / 10 + 1
for line in lines:
    if lineno % 10 == 0:
        pageno += 1
        print "\nFile \"%s\" Page %d/%d\n"      \
                 % (filename,pageno,pages)
    lineno += 1
    print "%4d" % lineno, line,
```

File Processing

- Example: **`print.py`** (output)

```
% print.py print.py

File "print.py" Page 1/5

   1 #!/usr/bin/env python
   2 #
   3 # print.py: command to print the contents of a file
   4 #
   5 # author: Rainer Doemer
   6 #
   7 # modifications:
   8 # 03/14/04 RD      initial version
   9
  10 # imports

File "print.py" Page 2/5

  11 import sys
...
```