algorithms and data structures

Algorithms and Data Structures

programming in Python revisited

sequences, dictionaries, lists

Persistent Data

storing information between executions

using DBM files

Object Serialization

defining data structures, for example: a set

using the Pickle module

an application to network programming

MCS 275 Lecture 36

Programming Tools and File Management

Jan Verschelde, 14 April 2008

algorithms and data structures

Algorithms and Data Structures

programming in Python revisited

Niklaus Wirth: programs = algorithms + data structures

Three basic control structures in any algorithm:

1. sequence of statements
2. conditional statement: if else
3. iteration: while and for loop

For every control structure,
we have a matching data structure:

	control structures	data structures
1	sequence	tuple
2	if else	dictionary
3	while / for	list

Algorithms and Data Structures

programming in Python revisited

Niklaus Wirth: programs = algorithms + data structures

Three basic control structures in any algorithm:

1. sequence of statements
2. conditional statement: if else
3. iteration: while and for loop

For every control structure,
we have a matching data structure:

	control structures	data structures
1	sequence	tuple
2	if else	dictionary
3	while / for	list

Algorithms and Data Structures

programming in Python revisited

Niklaus Wirth: programs = algorithms + data structures

Three basic control structures in any algorithm:

1. sequence of statements
2. conditional statement: if else
3. iteration: while and for loop

For every control structure,
we have a matching data structure:

	control structures	data structures
1	sequence	tuple
2	if else	dictionary
3	while / for	list

Algorithms and Data Structures

programming in Python revisited

Niklaus Wirth: programs = algorithms + data structures

Three basic control structures in any algorithm:

1. sequence of statements
2. conditional statement: if else
3. iteration: while and for loop

For every control structure,
we have a matching data structure:

	control structures	data structures
1	sequence	tuple
2	if else	dictionary
3	while / for	list

Algorithms and Data Structures

programming in Python revisited

Niklaus Wirth: programs = algorithms + data structures

Three basic control structures in any algorithm:

1. sequence of statements
2. conditional statement: if else
3. iteration: while and for loop

For every control structure,
we have a matching data structure:

	control structures	data structures
1	sequence	tuple
2	if else	dictionary
3	while / for	list

Algorithms and Data Structures

programming in Python revisited

Niklaus Wirth: programs = algorithms + data structures

Three basic control structures in any algorithm:

1. sequence of statements
2. conditional statement: if else
3. iteration: while and for loop

For every control structure,
we have a matching data structure:

	control structures	data structures
1	sequence	tuple
2	if else	dictionary
3	while / for	list

Algorithms and Data Structures

programming in Python revisited

Niklaus Wirth: programs = algorithms + data structures

Three basic control structures in any algorithm:

1. sequence of statements
2. conditional statement: if else
3. iteration: while and for loop

For every control structure,
we have a matching data structure:

	control structures	data structures
1	sequence	tuple
2	if else	dictionary
3	while / for	list

Algorithms and Data Structures

programming in Python revisited

Niklaus Wirth: programs = algorithms + data structures

Three basic control structures in any algorithm:

1. sequence of statements
2. conditional statement: if else
3. iteration: while and for loop

For every control structure,
we have a matching data structure:

	control structures	data structures
1	sequence	tuple
2	if else	dictionary
3	while / for	list

Algorithms and Data Structures

programming in Python revisited

Niklaus Wirth: programs = algorithms + data structures

Three basic control structures in any algorithm:

1. sequence of statements
2. conditional statement: if else
3. iteration: while and for loop

For every control structure,
we have a matching data structure:

	control structures	data structures
1	sequence	tuple
2	if else	dictionary
3	while / for	list

algorithms and data structures

Algorithms and Data Structures
- programming in Python revisited
- sequences, dictionaries, lists

Persistent Data
- storing information between executions
- using DBM files

Object Serialization
- defining data structures, for example: a set
- using the Pickle module
- an application to network programming

Programs are Data Transformations

tuples as sequences manipulated by functions

All data are sequences of bits, or bit tuples.

Swapping values:

```
>>> a = 1
>>> b = 2
>>> (b,a) = (a,b)
>>> b
1
>>> a
2
```

Functions take sequences of arguments on input and return sequences on output.

Programs are Data Transformations

tuples as sequences manipulated by functions

All data are sequences of bits, or bit tuples.

Swapping values:

```
>>> a = 1
>>> b = 2
>>> (b,a) = (a,b)
>>> b
1
>>> a
2
```

Functions take sequences of arguments on input and return sequences on output.

Programs are Data Transformations

tuples as sequences manipulated by functions

All data are sequences of bits, or bit tuples.

Swapping values:

```
>>> a = 1
>>> b = 2
>>> (b,a) = (a,b)
>>> b
1
>>> a
2
```

Functions take sequences of arguments on input and return sequences on output.

Programs are Data Transformations

tuples as sequences manipulated by functions

All data are sequences of bits, or bit tuples.

Swapping values:

```
>>> a = 1
>>> b = 2
>>> (b,a) = (a,b)
>>> b
1
>>> a
2
```

Functions take sequences of arguments on input and return sequences on output.

Storing Conditions

dictionaries and if else statements

We can represent an `if` tt else statement

```
>>> import time
>>> hour = time.localtime()[3]
>>> if hour < 12:
...     print 'good morning'
... else:
...     print 'good afternoon'
...
good afternoon
```

via a dictionary:

```
>>> d = { True:'good morning',
... False : 'good afternoon'}
>>> d[hour<12]
'good afternoon'
```

Storing Conditions

dictionaries and if else statements

We can represent an `if` tt else statement

```
>>> import time
>>> hour = time.localtime()[3]
>>> if hour < 12:
...     print 'good morning'
... else:
...     print 'good afternoon'
...
good afternoon
```

via a dictionary:

```
>>> d = { True:'good morning',
... False : 'good afternoon'}
>>> d[hour<12]
'good afternoon'
```

Loops and Lists

storing the results of a for loop

Printing all lower case characters:

```
>>> for i in range(ord('a'),ord('z')):
...     print chr(i)
```

A list of all lower case characters:

```
>>> L = range(ord('a'),ord('z'))
>>> map(chr,L)
```

`map()` returns a list of the results of applying a function to a sequence of arguments.

The `while` statement combines `for` with `if else`: conditional iteration.

Loops and Lists

storing the results of a for loop

Printing all lower case characters:

```
>>> for i in range(ord('a'),ord('z')):
...     print chr(i)
```

A list of all lower case characters:

```
>>> L = range(ord('a'),ord('z'))
>>> map(chr,L)
```

`map()` returns a list of the results
of applying a function to a sequence of arguments.

The `while` statement combines `for` with `if else`:
conditional iteration.

Loops and Lists

storing the results of a for loop

Printing all lower case characters:

```
>>> for i in range(ord('a'),ord('z')):
...     print chr(i)
```

A list of all lower case characters:

```
>>> L = range(ord('a'),ord('z'))
>>> map(chr,L)
```

`map()` returns a list of the results
of applying a function to a sequence of arguments.

The `while` statement combines `for` with `if else`:
conditional iteration.

Loops and Lists

storing the results of a for loop

Printing all lower case characters:

```
>>> for i in range(ord('a'),ord('z')):
...     print chr(i)
```

A list of all lower case characters:

```
>>> L = range(ord('a'),ord('z'))
>>> map(chr,L)
```

`map()` returns a list of the results of applying a function to a sequence of arguments.

The `while` statement combines `for` with `if else`: conditional iteration.

Loops and Lists

storing the results of a for loop

Printing all lower case characters:

```
>>> for i in range(ord('a'),ord('z')):
...     print chr(i)
```

A list of all lower case characters:

```
>>> L = range(ord('a'),ord('z'))
>>> map(chr,L)
```

`map()` returns a list of the results
of applying a function to a sequence of arguments.

The `while` statement combines `for` with `if else`:
conditional iteration.

Loops and Lists

storing the results of a for loop

Printing all lower case characters:

```
>>> for i in range(ord('a'),ord('z')):
...     print chr(i)
```

A list of all lower case characters:

```
>>> L = range(ord('a'),ord('z'))
>>> map(chr,L)
```

`map()` returns a list of the results
of applying a function to a sequence of arguments.

The `while` statement combines `for` with `if else`:
conditional iteration.

List Comprehensions

defining lists in a short way

Instead of `map()`, `filter()`, etc... (eventually with `lambda` functions), *list comprehensions* provide a shorter way to create lists:

To sample integer points on the parabola $y = x^2$:

```
>>> [(x,x**2) for x in range(0,3)]
[(0, 0), (1, 1), (2, 4)]
```

Generating three random numbers:

```
>>> from random import uniform
>>> L = [uniform(0,1) for i in range(0,3)]
>>> [ '%.3f' % x for x in L]
['0.843', '0.308', '0.272']
```

List Comprehensions

defining lists in a short way

Instead of `map()`, `filter()`, etc... (eventually with `lambda` functions), *list comprehensions* provide a shorter way to create lists:

To sample integer points on the parabola $y = x^2$:

```
>>> [(x,x**2) for x in range(0,3)]
[(0, 0), (1, 1), (2, 4)]
```

Generating three random numbers:

```
>>> from random import uniform
>>> L = [uniform(0,1) for i in range(0,3)]
>>> [ '%.3f' % x for x in L]
['0.843', '0.308', '0.272']
```

List Comprehensions

defining lists in a short way

Instead of `map()`, `filter()`, etc... (eventually with `lambda` functions), *list comprehensions* provide a shorter way to create lists:

To sample integer points on the parabola $y = x^2$:

```
>>> [(x,x**2) for x in range(0,3)]
[(0, 0), (1, 1), (2, 4)]
```

Generating three random numbers:

```
>>> from random import uniform
>>> L = [uniform(0,1) for i in range(0,3)]
>>> [ '%.3f' % x for x in L]
['0.843', '0.308', '0.272']
```

List Comprehensions

defining lists in a short way

Instead of `map()`, `filter()`, etc... (eventually with `lambda` functions), *list comprehensions* provide a shorter way to create lists:

To sample integer points on the parabola $y = x^2$:

```
>>> [(x,x**2) for x in range(0,3)]
[(0, 0), (1, 1), (2, 4)]
```

Generating three random numbers:

```
>>> from random import uniform
>>> L = [uniform(0,1) for i in range(0,3)]
>>> [ '%.3f' % x for x in L]
['0.843', '0.308', '0.272']
```

algorithms and data structures

Algorithms and Data Structures
programming in Python revisited
sequences, dictionaries, lists
Persistent Data
storing information between executions
using DBM files
Object Serialization
defining data structures, for example: a set
using the Pickle module
an application to network programming

Persistent Data

storing information between executions

Data that is *persistent* outlives programs.

Objects constructed by a script are lost
as soon as the script ends.

Two extremes to make data persistent:

1. files: store string representations,
2. MySQL: store data in tables in a database.

Intermediate solution: DBM files.

Persistent Data

storing information between executions

Data that is *persistent* outlives programs.

Objects constructed by a script are lost as soon as the script ends.

Two extremes to make data persistent:

1. files: store string representations,
2. MySQL: store data in tables in a database.

Intermediate solution: DBM files.

Persistent Data

storing information between executions

Data that is *persistent* outlives programs.

Objects constructed by a script are lost as soon as the script ends.

Two extremes to make data persistent:

1. files: store string representations,
2. MySQL: store data in tables in a database.

Intermediate solution: DBM files.

Persistent Data

storing information between executions

Data that is *persistent* outlives programs.

Objects constructed by a script are lost
as soon as the script ends.

Two extremes to make data persistent:

1. files: store string representations,
2. MySQL: store data in tables in a database.

Intermediate solution: DBM files.

Persistent Data

storing information between executions

Data that is *persistent* outlives programs.

Objects constructed by a script are lost
as soon as the script ends.

Two extremes to make data persistent:

1. files: store string representations,
2. MySQL: store data in tables in a database.

Intermediate solution: DBM files.

algorithms and data structures

Algorithms and Data Structures

- programming in Python revisited
- sequences, dictionaries, lists

Persistent Data

- storing information between executions
- using DBM files

Object Serialization

- defining data structures, for example: a set
- using the Pickle module
- an application to network programming

Using DBM Files

DBM files are standard in the Python library.

```
$ python
>>> import anydbm
>>> libdb = anydbm.open('library','c')
```

opened a new dbm with read-write access (flag = `'c'`)

```
>>> libdb['0'] = str({'author':'Rashi Gupta',
... 'title':'Making Use of Python'})
```

keys and values must be of type string

```
>>> libdb.keys()
['0']
>>> libdb.values()
["{'title': 'Making Use of Python', 'author': 'Rashi 
$ ls
```

→ `library` is a file in current directory.

Using DBM Files

DBM files are standard in the Python library.

```
$ python
>>> import anydbm
>>> libdb = anydbm.open('library','c')
```

opened a new dbm with read-write access (flag = `'c'`)

```
>>> libdb['0'] = str({'author':'Rashi Gupta',
... 'title':'Making Use of Python'})
```

keys and values must be of type string

```
>>> libdb.keys()
['0']
>>> libdb.values()
["{'title': 'Making Use of Python', 'author': 'Rashi
$ ls
```

→ `library` is a file in current directory.

Using DBM Files

DBM files are standard in the Python library.

```
$ python
>>> import anydbm
>>> libdb = anydbm.open('library','c')
```

opened a new dbm with read-write access (flag = `'c'`)

```
>>> libdb['0'] = str({'author':'Rashi Gupta',
... 'title':'Making Use of Python'})
```

keys and values must be of type string

```
>>> libdb.keys()
['0']
>>> libdb.values()
["{'title': 'Making Use of Python', 'author': 'Rashi G
$ ls
```

→ `library` is a file in current directory.

Using DBM Files

DBM files are standard in the Python library.

```
$ python
>>> import anydbm
>>> libdb = anydbm.open('library','c')
```

opened a new dbm with read-write access (flag = `'c'`)

```
>>> libdb['0'] = str({'author':'Rashi Gupta',
... 'title':'Making Use of Python'})
```

keys and values must be of type string

```
>>> libdb.keys()
['0']
>>> libdb.values()
["{'title': 'Making Use of Python', 'author': 'Rashi G
$ ls
```

→ `library` is a file in current directory.

Adding Books to the Library

and selecting books using the key

```
>>> import anydbm
>>> mylib = anydbm.open('library','c')
>>> mylib.keys()
['0']
>>> mylib['1'] = str({'author':'S. Ceri et al.',
... 'title':'The Art and Craft of Computing'})
>>> mylib.values()
["{'title': 'Making Use of Python', 'author': 'Rashi Gupta
 "{'title': 'The Art and Craft of Computing', 'author': 'S
```

Selecting the author of book with key 1:

```
>>> V = mylib.values()
>>> d = V[int(mylib.keys()[1])]
>>> eval(d)['author']
'S. Ceri et al.'
```

Adding Books to the Library

and selecting books using the key

```
>>> import anydbm
>>> mylib = anydbm.open('library','c')
>>> mylib.keys()
['0']
>>> mylib['1'] = str({'author':'S. Ceri et al.',
... 'title':'The Art and Craft of Computing'})
>>> mylib.values()
["{'title': 'Making Use of Python', 'author': 'Rashi Gupta
 "{'title': 'The Art and Craft of Computing', 'author': 'S
```

Selecting the author of book with key 1:

```
>>> V = mylib.values()
>>> d = V[int(mylib.keys()[1])]
>>> eval(d)['author']
'S. Ceri et al.'
```

Adding Books to the Library

and selecting books using the key

```
>>> import anydbm
>>> mylib = anydbm.open('library','c')
>>> mylib.keys()
['0']
>>> mylib['1'] = str({'author':'S. Ceri et al.',
... 'title':'The Art and Craft of Computing'})
>>> mylib.values()
["{'title': 'Making Use of Python', 'author': 'Rashi Gupta
 "{'title': 'The Art and Craft of Computing', 'author': 'S
```

Selecting the author of book with key 1:

```
>>> V = mylib.values()
>>> d = V[int(mylib.keys()[1])]
>>> eval(d)['author']
'S. Ceri et al.'
```

Adding Books to the Library

and selecting books using the key

```
>>> import anydbm
>>> mylib = anydbm.open('library','c')
>>> mylib.keys()
['0']
>>> mylib['1'] = str({'author':'S. Ceri et al.',
... 'title':'The Art and Craft of Computing'})
>>> mylib.values()
["{'title': 'Making Use of Python', 'author': 'Rashi Gupta
 "{'title': 'The Art and Craft of Computing', 'author': 'S
```

Selecting the author of book with key 1:

```
>>> V = mylib.values()
>>> d = V[int(mylib.keys()[1])]
>>> eval(d)['author']
'S. Ceri et al.'
```

Adding Books to the Library

and selecting books using the key

```
>>> import anydbm
>>> mylib = anydbm.open('library','c')
>>> mylib.keys()
['0']
>>> mylib['1'] = str({'author':'S. Ceri et al.',
... 'title':'The Art and Craft of Computing'})
>>> mylib.values()
["{'title': 'Making Use of Python', 'author': 'Rashi Gupta
 "{'title': 'The Art and Craft of Computing', 'author': 'S
```

Selecting the author of book with key 1:

```
>>> V = mylib.values()
>>> d = V[int(mylib.keys()[1])]
>>> eval(d)['author']
'S. Ceri et al.'
```

DBM File Operations

an overview

Python code	description
`import anydbm`	load module anydbm
`f = anydbm.open('n','c')`	create or open dbm file with name n
`f['key'] = 'value'`	assign value for key
`value = f['key']`	load value for key
`count = len(f)`	number of entries stored
`found = f.has_key('key')`	see if entry for key
`del f['key']`	remove entry for key
`f.close()`	close dbm file

Typical use:

- every record in database has unique key
- values are dictionaries, stored as strings

DBM File Operations

an overview

Python code	description
`import anydbm`	load module anydbm
`f = anydbm.open('n','c')`	create or open dbm file with name n
`f['key'] = 'value'`	assign value for key
`value = f['key']`	load value for key
`count = len(f)`	number of entries stored
`found = f.has_key('key')`	see if entry for key
`del f['key']`	remove entry for key
`f.close()`	close dbm file

Typical use:

- every record in database has unique key
- values are dictionaries, stored as strings

DBM File Operations

an overview

Python code	description
`import anydbm`	load module anydbm
`f = anydbm.open('n','c')`	create or open dbm file with name n
`f['key'] = 'value'`	assign value for key
`value = f['key']`	load value for key
`count = len(f)`	number of entries stored
`found = f.has_key('key')`	see if entry for key
`del f['key']`	remove entry for key
`f.close()`	close dbm file

Typical use:

- every record in database has unique key
- values are dictionaries, stored as strings

DBM File Operations

an overview

Python code	description
`import anydbm`	load module anydbm
`f = anydbm.open('n','c')`	create or open dbm file with name n
`f['key'] = 'value'`	assign value for key
`value = f['key']`	load value for key
`count = len(f)`	number of entries stored
`found = f.has_key('key')`	see if entry for key
`del f['key']`	remove entry for key
`f.close()`	close dbm file

Typical use:

- every record in database has unique key
- values are dictionaries, stored as strings

DBM File Operations

an overview

Python code	description
`import anydbm`	load module `anydbm`
`f = anydbm.open('n','c')`	create or open dbm file with name n
`f['key'] = 'value'`	assign `value` for `key`
`value = f['key']`	load `value` for `key`
`count = len(f)`	number of entries stored
`found = f.has_key('key')`	see if entry for `key`
`del f['key']`	remove entry for `key`
`f.close()`	close dbm file

Typical use:

- every record in database has unique key
- values are dictionaries, stored as strings

DBM File Operations

an overview

Python code	description
`import anydbm`	load module anydbm
`f = anydbm.open('n','c')`	create or open dbm file with name n
`f['key'] = 'value'`	assign `value` for `key`
`value = f['key']`	load `value` for `key`
`count = len(f)`	number of entries stored
`found = f.has_key('key')`	see if entry for `key`
`del f['key']`	remove entry for `key`
`f.close()`	close dbm file

Typical use:

- every record in database has unique key
- values are dictionaries, stored as strings

DBM File Operations

an overview

Python code	description
`import anydbm`	load module `anydbm`
`f = anydbm.open('n','c')`	create or open dbm file with name `n`
`f['key'] = 'value'`	assign `value` for key
`value = f['key']`	load `value` for key
`count = len(f)`	number of entries stored
`found = f.has_key('key')`	see if entry for key
`del f['key']`	remove entry for key
`f.close()`	close dbm file

Typical use:

- every record in database has unique key
- values are dictionaries, stored as strings

DBM File Operations

an overview

Python code	description
`import anydbm`	load module anydbm
`f = anydbm.open('n','c')`	create or open dbm file with name n
`f['key'] = 'value'`	assign value for key
`value = f['key']`	load value for key
`count = len(f)`	number of entries stored
`found = f.has_key('key')`	see if entry for key
`del f['key']`	remove entry for key
`f.close()`	close dbm file

Typical use:

- every record in database has unique key
- values are dictionaries, stored as strings

DBM File Operations

an overview

Python code	description
`import anydbm`	load module `anydbm`
`f = anydbm.open('n','c')`	create or open dbm file with name n
`f['key'] = 'value'`	assign `value` for `key`
`value = f['key']`	load `value` for `key`
`count = len(f)`	number of entries stored
`found = f.has_key('key')`	see if entry for `key`
`del f['key']`	remove entry for `key`
`f.close()`	close dbm file

Typical use:

- every record in database has unique key
- values are dictionaries, stored as strings

algorithms and data structures

Algorithms and Data Structures
programming in Python revisited
sequences, dictionaries, lists
Persistent Data
storing information between executions
using DBM files
Object Serialization
defining data structures, for example: a set
using the Pickle module
an application to network programming

A Data Type Set

object oriented design of data structures

Builtin data types offer

- storage: object data attributes,
- methods: object functional attributes.

Example: define a class Set.
A set is a list without duplicates.

```
>>> from class_set import *
>>> E = Set()
>>> E
{}
>>> A = Set(2,5,2)
>>> A
{2,5}
```

A Data Type Set

object oriented design of data structures

Builtin data types offer

- storage: object data attributes,
- methods: object functional attributes.

Example: define a class Set.
A set is a list without duplicates.

```
>>> from class_set import *
>>> E = Set()
>>> E
{}
>>> A = Set(2,5,2)
>>> A
{2,5}
```

A Data Type Set

object oriented design of data structures

Builtin data types offer

- storage: object data attributes,
- methods: object functional attributes.

Example: define a class Set.
A set is a list without duplicates.

```
>>> from class_set import *
>>> E = Set()
>>> E
{}
>>> A = Set(2,5,2)
>>> A
{2,5}
```

A Data Type Set

object oriented design of data structures

Builtin data types offer

- storage: object data attributes,
- methods: object functional attributes.

Example: define a class Set.
A set is a list without duplicates.

```
>>> from class_set import *
>>> E = Set()
>>> E
{}
>>> A = Set(2,5,2)
>>> A
{2,5}
```

A Data Type Set

object oriented design of data structures

Builtin data types offer

- storage: object data attributes,
- methods: object functional attributes.

Example: define a class Set.
A set is a list without duplicates.

```
>>> from class_set import *
>>> E = Set()
>>> E
{}
>>> A = Set(2,5,2)
>>> A
{2,5}
```

Incremental Design of a Class

first constructor and string representation

```
class Set:
    """
    A set is a list without duplicates.
    """
    def __init__(self,*elements):
        """
        Turns a sequence of arguments in
        elements into a set.
        """
    def __str__(self):
        """
        Returns a string representation
        of a set as { elements }.
        """
```

Incremental Design of a Class

first constructor and string representation

```
class Set:
    """
    A set is a list without duplicates.
    """
    def __init__(self,*elements):
        """
        Turns a sequence of arguments in
        elements into a set.
        """
    def __str__(self):
        """
        Returns a string representation
        of a set as { elements }.
        """
```

Incremental Design of a Class

first constructor and string representation

```
class Set:
   """
   A set is a list without duplicates.
   """
   def __init__(self,*elements):
      """
      Turns a sequence of arguments in
      elements into a set.
      """
   def __str__(self):
      """
      Returns a string representation
      of a set as { elements }.
      """
```

Code for the Constructor

in the function `__init__`

```
def __init__(self,*elements):
    """
    Turns a sequence of arguments in
    elements into a set.
    """
    self.s = []
    for e in elements:
        if not e in self.s:
            self.s.append(e)

def __repr__(self):
    """
    The string representation defines
    the set representation.
    """
    return str(self)
```

Code for the Constructor

in the function `__init__`

```
def __init__(self,*elements):
   """
   Turns a sequence of arguments in
   elements into a set.
   """
   self.s = []
   for e in elements:
      if not e in self.s:
         self.s.append(e)

def __repr__(self):
   """
   The string representation defines
   the set representation.
   """
   return str(self)
```

Code for the Constructor

in the function `__init__`

```
def __init__(self,*elements):
   """
   Turns a sequence of arguments in
   elements into a set.
   """
   self.s = []
   for e in elements:
      if not e in self.s:
         self.s.append(e)

def __repr__(self):
   """
   The string representation defines
   the set representation.
   """
   return str(self)
```

Defining the String Representation

in the function `__str__`

```
def __str__(self):
    """
    Returns a string representation
    of a set as { elements }.
    """
    n = len(self.s)-1
    if n < 0:
        r = '{}'
    else:
        r = '{'
        for i in range(0,n):
            r = r + str(self.s[i]) + ','
        r = r + str(self.s[n]) + '}'
    return r
```

Defining the String Representation

in the function `__str__`

```
def __str__(self):
    """
    Returns a string representation
    of a set as { elements }.
    """
    n = len(self.s)-1
    if n < 0:
        r = '{}'
    else:
        r = '{'
        for i in range(0,n):
            r = r + str(self.s[i]) + ','
        r = r + str(self.s[n]) + '}'
    return r
```

Defining the String Representation

in the function `__str__`

```
def __str__(self):
    """
    Returns a string representation
    of a set as { elements }.
    """
    n = len(self.s)-1
    if n < 0:
        r = '{}'
    else:
        r = '{'
        for i in range(0,n):
            r = r + str(self.s[i]) + ','
        r = r + str(self.s[n]) + '}'
    return r
```

Defining the String Representation

in the function `__str__`

```
def __str__(self):
    """
    Returns a string representation
    of a set as { elements }.
    """
    n = len(self.s)-1
    if n < 0:
        r = '{}'
    else:
        r = '{'
        for i in range(0,n):
            r = r + str(self.s[i]) + ','
        r = r + str(self.s[n]) + '}'
    return r
```

algorithms and data structures

Algorithms and Data Structures
programming in Python revisited
sequences, dictionaries, lists
Persistent Data
storing information between executions
using DBM files
Object Serialization
defining data structures, for example: a set
using the Pickle module
an application to network programming

Pickled Objects

mapping data structures to serial strings

Main limitation of DBM files:

- data stored under a key must be a string.

The `str` and `eval` works for dictionaries, but not for class instances. Recreating objects from standard string representations is in general not possible.

Serialization is the conversation of objects to strings. Arbitrarily data structures in memory are mapped to a serial string form.

The `pickle` module is standard in Python.
Its C implementation `cPickle` is more efficient.

Pickled Objects

mapping data structures to serial strings

Main limitation of DBM files:

- data stored under a key must be a string.

The `str` and `eval` works for dictionaries, but not for class instances. Recreating objects from standard string representations is in general not possible.

Serialization is the conversation of objects to strings. Arbitrarily data structures in memory are mapped to a serial string form.

The `pickle` module is standard in Python.
Its C implementation `cPickle` is more efficient.

Pickled Objects

mapping data structures to serial strings

Main limitation of DBM files:

- data stored under a key must be a string.

The `str` and `eval` works for dictionaries, but not for class instances. Recreating objects from standard string representations is in general not possible.

Serialization is the conversation of objects to strings. Arbitrarily data structures in memory are mapped to a serial string form.

The `pickle` module is standard in Python.
Its C implementation `cPickle` is more efficient.

Pickled Objects

mapping data structures to serial strings

Main limitation of DBM files:

- data stored under a key must be a string.

The `str` and `eval` works for dictionaries, but not for class instances. Recreating objects from standard string representations is in general not possible.

Serialization is the conversation of objects to strings. Arbitrarily data structures in memory are mapped to a serial string form.

The `pickle` module is standard in Python.
Its C implementation `cPickle` is more efficient.

Using cPickle

to store sets

```
$ python
>>> from class_set import *
>>> A = Set('a',3,Set(3,'five'))
>>> import cPickle
>>> setdb = open('oursets','w')
>>> cPickle.dump(A,setdb)
```

oursets is a file. A new Python session:

```
$ python
>>> from class_set import *
>>> import cPickle
>>> f = open('oursets','r')
>>> S = cPickle.load(f)
>>> S
{a,3,{3,five}}
```

Using cPickle

to store sets

```
$ python
>>> from class_set import *
>>> A = Set('a',3,Set(3,'five'))
>>> import cPickle
>>> setdb = open('oursets','w')
>>> cPickle.dump(A,setdb)
```

oursets is a file. A new Python session:

```
$ python
>>> from class_set import *
>>> import cPickle
>>> f = open('oursets','r')
>>> S = cPickle.load(f)
>>> S
{a,3,{3,five}}
```

Using cPickle

to store sets

```
$ python
>>> from class_set import *
>>> A = Set('a',3,Set(3,'five'))
>>> import cPickle
>>> setdb = open('oursets','w')
>>> cPickle.dump(A,setdb)
```

`oursets` is a file. A new Python session:

```
$ python
>>> from class_set import *
>>> import cPickle
>>> f = open('oursets','r')
>>> S = cPickle.load(f)
>>> S
{a,3,{3,five}}
```

Using cPickle

to store sets

```
$ python
>>> from class_set import *
>>> A = Set('a',3,Set(3,'five'))
>>> import cPickle
>>> setdb = open('oursets','w')
>>> cPickle.dump(A,setdb)
```

`oursets` is a file. A new Python session:

```
$ python
>>> from class_set import *
>>> import cPickle
>>> f = open('oursets','r')
>>> S = cPickle.load(f)
>>> S
{a,3,{3,five}}
```

Using `cPickle`

to store sets

```
$ python
>>> from class_set import *
>>> A = Set('a',3,Set(3,'five'))
>>> import cPickle
>>> setdb = open('oursets','w')
>>> cPickle.dump(A,setdb)
```

`oursets` is a file. A new Python session:

```
$ python
>>> from class_set import *
>>> import cPickle
>>> f = open('oursets','r')
>>> S = cPickle.load(f)
>>> S
{a,3,{3,five}}
```

the File `oursets`

```
(iclass_set
Set
p1
(dp2
S's'
(lp3
S'a'
aI3
a(iclass_set
Set
p4
(dp5
S's'
(lp6
I3
aS'five'
p7
asbasb.
```

Syntax for general Use

two methods: dump and load

General syntax to pickle and unpickle:

1. pickling
   dumping an object to a file:

   ```
   import cPickle
   < file > = open( < name > , 'w' )
   cPickle.dump( < object > , < file > )
   ```

2. unpickling
   loading an object from file:

   ```
   < file > = open( < name > , 'r' )
   < object > = cPickle.load( < name > )
   ```

Not all objects can be pickled, e.g.: files.
An exception `PickleError` is provided.

Syntax for general Use

two methods: dump and load

General syntax to pickle and unpickle:

1. pickling
   dumping an object to a file:

   ```
   import cPickle
   < file > = open( < name > , 'w' )
   cPickle.dump( < object > ,  < file > )
   ```

2. unpickling
   loading an object from file:

   ```
   < file > = open( < name > , 'r' )
   < object > = cPickle.load( < name > )
   ```

Not all objects can be pickled, e.g.: files.
An exception `PickleError` is provided.

Syntax for general Use

two methods: dump and load

General syntax to pickle and unpickle:

1. pickling
   dumping an object to a file:
   ```
   import cPickle
   < file > = open( < name > , 'w' )
   cPickle.dump( < object > ,  < file > )
   ```
2. unpickling
   loading an object from file:
   ```
   < file > = open( < name > , 'r' )
   < object > = cPickle.load( < name > )
   ```

Not all objects can be pickled, e.g.: files.
An exception `PickleError` is provided.

algorithms and data structures

Algorithms and Data Structures
programming in Python revisited
sequences, dictionaries, lists
Persistent Data
storing information between executions
using DBM files
Object Serialization
defining data structures, for example: a set
using the Pickle module
an application to network programming

Sending Objects through sockets

an application of pickling to network programming

Suppose we want to send sets from server to client or from client to server.

Using pickling as follows:

1. sender dumps object to local file
2. sender reads file into one string
3. sender sends the string
4. receiver writes strings to file
5. receiver loads object from file

Sending Objects through sockets

an application of pickling to network programming

Suppose we want to send sets from server to client or from client to server.

Using pickling as follows:

1. sender dumps object to local file
2. sender reads file into one string
3. sender sends the string
4. receiver writes strings to file
5. receiver loads object from file

Sending Objects through sockets

an application of pickling to network programming

Suppose we want to send sets from server to client or from client to server.

Using pickling as follows:

1. sender dumps object to local file
2. sender reads file into one string
3. sender sends the string
4. receiver writes strings to file
5. receiver loads object from file

Sending Objects through sockets

an application of pickling to network programming

Suppose we want to send sets from server to client or from client to server.

Using pickling as follows:

1. sender dumps object to local file
2. sender reads file into one string
3. sender sends the string
4. receiver writes strings to file
5. receiver loads object from file

Sending Objects through sockets

an application of pickling to network programming

Suppose we want to send sets from server to client or from client to server.

Using pickling as follows:

1. sender dumps object to local file
2. sender reads file into one string
3. sender sends the string
4. receiver writes strings to file
5. receiver loads object from file

Sending Objects through sockets

an application of pickling to network programming

Suppose we want to send sets from server to client or from client to server.

Using pickling as follows:

1. sender dumps object to local file
2. sender reads file into one string
3. sender sends the string
4. receiver writes strings to file
5. receiver loads object from file

Writing to Strings

using the module StringIO

Instead of working with a temporary file, we can directly write to strings.

```
>>> from class_set import *
>>> import cPickle
>>> f = open('oursets','r')
>>> A = cPickle.load(f)
>>> A
{a,3,{3,five}}
```

```
>>> import StringIO
>>> output = StringIO.StringIO()
>>> cPickle.dump(A,output)
>>> output.getvalue()
"(iclass_set\nSet\np1\n(dp2\nS's'\n(lp3\nS'a'\naI3\na
```

Writing to Strings

using the module `StringIO`

Instead of working with a temporary file,
we can directly write to strings.

```
>>> from class_set import *
>>> import cPickle
>>> f = open('oursets','r')
>>> A = cPickle.load(f)
>>> A
{a,3,{3,five}}

>>> import StringIO
>>> output = StringIO.StringIO()
>>> cPickle.dump(A,output)
>>> output.getvalue()
"(iclass_set\nSet\np1\n(dp2\nS's'\n(lp3\nS'a'\naI3\na
```

Writing to Strings

using the module `StringIO`

Instead of working with a temporary file,
we can directly write to strings.

```
>>> from class_set import *
>>> import cPickle
>>> f = open('oursets','r')
>>> A = cPickle.load(f)
>>> A
{a,3,{3,five}}

>>> import StringIO
>>> output = StringIO.StringIO()
>>> cPickle.dump(A,output)
>>> output.getvalue()
"(iclass_set\nSet\np1\n(dp2\nS's'\n(lp3\nS'a'\naI3\na
```

Summary + Assignments

We restarted *Making Use of Python*...

Assignments:

1. Use list comprehensions to generate points (x, y) uniformly distributed on the circle: $x^2 + y^2 = 1$. (For some angle t: $x = \cos(t)$, $y = \sin(t)$.)
2. Extend the Class set with a method `member` – e.g.: `A.member(3)` – returning True or False accordingly.
3. Augment the Class set with the method `add`, to add a sequence of elements to a set. The number of elements varies, e.g.: `S.add(2)`, `S.add(2,3)`, etc. are all valid uses of `add`.
4. Define the union and intersect operations on sets.
5. Give code for client and server to interchange a set.