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
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
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:

<table>
<thead>
<tr>
<th>control structures</th>
<th>data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sequence</td>
</tr>
<tr>
<td></td>
<td>tuple</td>
</tr>
<tr>
<td>2</td>
<td>if else</td>
</tr>
<tr>
<td></td>
<td>dictionary</td>
</tr>
<tr>
<td>3</td>
<td>while / for</td>
</tr>
<tr>
<td></td>
<td>list</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>control structures</th>
<th>data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sequence</td>
</tr>
<tr>
<td></td>
<td>tuple</td>
</tr>
<tr>
<td>2</td>
<td>if else</td>
</tr>
<tr>
<td></td>
<td>dictionary</td>
</tr>
<tr>
<td>3</td>
<td>while / for</td>
</tr>
<tr>
<td></td>
<td>list</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>control structures</th>
<th>data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>tuple</td>
</tr>
<tr>
<td>if else</td>
<td>dictionary</td>
</tr>
<tr>
<td>while / for</td>
<td>list</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>control structures</th>
<th>data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sequence</td>
</tr>
<tr>
<td></td>
<td>tuple</td>
</tr>
<tr>
<td>2</td>
<td>if else</td>
</tr>
<tr>
<td></td>
<td>dictionary</td>
</tr>
<tr>
<td>3</td>
<td>while / for</td>
</tr>
<tr>
<td></td>
<td>list</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>control structures</th>
<th>data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sequence</td>
</tr>
<tr>
<td></td>
<td>tuple</td>
</tr>
<tr>
<td>2</td>
<td>if else</td>
</tr>
<tr>
<td></td>
<td>dictionary</td>
</tr>
<tr>
<td>3</td>
<td>while / for</td>
</tr>
<tr>
<td></td>
<td>list</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>control structures</th>
<th>data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sequence</td>
</tr>
<tr>
<td></td>
<td>tuple</td>
</tr>
<tr>
<td>2</td>
<td>if else</td>
</tr>
<tr>
<td></td>
<td>dictionary</td>
</tr>
<tr>
<td>3</td>
<td>while / for</td>
</tr>
<tr>
<td></td>
<td>list</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>control structures</th>
<th>data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sequence</td>
</tr>
<tr>
<td>2</td>
<td>if else</td>
</tr>
<tr>
<td>3</td>
<td>while / for</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>control structures</th>
<th>data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. sequence</td>
<td>tuple</td>
</tr>
<tr>
<td>2. if else</td>
<td>dictionary</td>
</tr>
<tr>
<td>3. while / for</td>
<td>list</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>control structures</th>
<th>data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 sequence</td>
<td>tuple</td>
</tr>
<tr>
<td>2 if else</td>
<td>dictionary</td>
</tr>
<tr>
<td>3 while / for</td>
<td>list</td>
</tr>
</tbody>
</table>
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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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

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

via a dictionary:

```python
>>> 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:

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

A list of all lower case characters:

```python
>>> 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:

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

A list of all lower case characters:

```python
>>> 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:

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

A list of all lower case characters:

```python
>>> 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:

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

A list of all lower case characters:

```python
>>> 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:

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

A list of all lower case characters:

```python
>>> 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:

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

A list of all lower case characters:

```python
>>> 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 $y = x^2$:

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

Generating three random numbers:

```python
>>> 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 $y = x^2$:

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

Generating three random numbers:

```python
>>> 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 $y = x^2$:

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

Generating three random numbers:

```python
>>> 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 $y = x^2$:

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

Generating three random numbers:

```python
>>> 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,

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,

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,

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,

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,

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 Gupta"}']

$ 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 Gupta'}}"

$ 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 Gupta"}']
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 Gupta"}']

$ ls
   → library is a file in current directory.
Adding Books to the Library
and selecting books using the key

```python
>>> 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. Ceri et al.'}"

Selecting the author of book with key 1:

```python
>>> 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

```python
>>> 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'})
```  
Selecting the author of book with key 1:

```python
>>> 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

```python
>>> 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. Ceri et al.'}"

Selecting the author of book with key 1:

```python
>>> 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

```python
>>> 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'

Selecting the author of book with key 1:

```python
>>> 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

```python
>>> 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:

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

**an overview**

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

**Typical use:**

- every record in database has unique key
- values are dictionaries, stored as strings
### DBM File Operations

#### an overview

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

#### Typical use:
- every record in database has unique key
- values are dictionaries, stored as strings
DBM File Operations
an overview

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

Typical use:
- every record in database has unique key
- values are dictionaries, stored as strings
## DBM File Operations

### an overview

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

**Typical use:**

- every record in database has unique key
- values are dictionaries, stored as strings
# DBM File Operations

## an overview

<table>
<thead>
<tr>
<th>Python code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>python import anydbm f = anydbm.open('n','c') f['key'] = 'value' value = f['key'] count = len(f) found = f.has_key('key') del f['key'] f.close()</code></td>
<td><code>load module anydbm create or open dbm file with name n assign value for key load value for key number of entries stored see if entry for key remove entry for key close dbm file</code></td>
</tr>
</tbody>
</table>

**Typical use:**

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

an overview

<table>
<thead>
<tr>
<th>Python code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>import anydbm</td>
<td>load module anydbm</td>
</tr>
<tr>
<td>f = anydbm.open(‘n’,’c’)</td>
<td>create or open dbm file with name n</td>
</tr>
<tr>
<td>f[‘key’] = ‘value’</td>
<td>assign value for key</td>
</tr>
<tr>
<td>value = f[‘key’]</td>
<td>load value for key</td>
</tr>
<tr>
<td>count = len(f)</td>
<td>number of entries stored</td>
</tr>
<tr>
<td>found = f.has_key(‘key’)</td>
<td>see if entry for key</td>
</tr>
<tr>
<td>del f[‘key’]</td>
<td>remove entry for key</td>
</tr>
<tr>
<td>f.close()</td>
<td>close dbm file</td>
</tr>
</tbody>
</table>

Typical use:

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

an overview

<table>
<thead>
<tr>
<th>Python code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>import anydbm</td>
<td>load module anydbm</td>
</tr>
<tr>
<td>f = anydbm.open(’n’,’c’)</td>
<td>create or open dbm file with name n</td>
</tr>
<tr>
<td>f[’key’] = ’value’</td>
<td>assign value for key</td>
</tr>
<tr>
<td>value = f[’key’]</td>
<td>load value for key</td>
</tr>
<tr>
<td>count = len(f)</td>
<td>number of entries stored</td>
</tr>
<tr>
<td>found = f.has_key(’key’)</td>
<td>see if entry for key</td>
</tr>
<tr>
<td>del f[’key’]</td>
<td>remove entry for key</td>
</tr>
<tr>
<td>f.close()</td>
<td>close dbm file</td>
</tr>
</tbody>
</table>

Typical use:

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

an overview

<table>
<thead>
<tr>
<th>Python code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>import anydbm</td>
<td>load module anydbm</td>
</tr>
<tr>
<td>f = anydbm.open(’n’,’c’)</td>
<td>create or open dbm file with name n</td>
</tr>
<tr>
<td>f[’key’] = ’value’</td>
<td>assign value for key</td>
</tr>
<tr>
<td>value = f[’key’]</td>
<td>load value for key</td>
</tr>
<tr>
<td>count = len(f)</td>
<td>number of entries stored</td>
</tr>
<tr>
<td>found = f.has_key(’key’)</td>
<td>see if entry for key</td>
</tr>
<tr>
<td>del f[’key’]</td>
<td>remove entry for key</td>
</tr>
<tr>
<td>f.close()</td>
<td>close dbm file</td>
</tr>
</tbody>
</table>

Typical use:

- every record in database has unique key
- values are dictionaries, stored as strings
DBM File Operations
an overview

<table>
<thead>
<tr>
<th>Python code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>import anydbm</td>
<td>load module anydbm</td>
</tr>
<tr>
<td>f = anydbm.open(‘n’,‘c’)</td>
<td>create or open dbm file with name n</td>
</tr>
<tr>
<td>f[‘key’] = ‘value’</td>
<td>assign value for key</td>
</tr>
<tr>
<td>value = f[‘key’]</td>
<td>load value for key</td>
</tr>
<tr>
<td>count = len(f)</td>
<td>number of entries stored</td>
</tr>
<tr>
<td>found = f.has_key(‘key’)</td>
<td>see if entry for key</td>
</tr>
<tr>
<td>del f[‘key’]</td>
<td>remove entry for key</td>
</tr>
<tr>
<td>f.close()</td>
<td>close dbm file</td>
</tr>
</tbody>
</table>

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

Built-in data types offer
  ▶ storage: object data attributes,
  ▶ methods: object functional attributes.

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

```python
>>> 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

Built-in data types offer
- storage: object data attributes,
- methods: object functional attributes.

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

```python
>>> 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.

```python
>>> 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

Built-in data types offer
  - storage: object data attributes,
  - methods: object functional attributes.

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

```python
>>> 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

Built-in data types offer
- storage: object data attributes,
- methods: object functional attributes.

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

```python
>>> 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 }.
        ""
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 }.
            ""
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 \texttt{__init__}

```python
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)
```

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
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)
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\_

```python
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
```
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
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 conversion 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 \texttt{str} and \texttt{eval} works for dictionaries, but not for class instances. Recreating objects from standard string representations is in general not possible.

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

The \texttt{pickle} module is standard in Python. Its C implementation \texttt{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
$ 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
$ 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

(class_set
Set
p1
(dp2
S's'
(lp3
S'a'
aI3
a(class_set
Set
p4
(dp5
S's'
(lp6
I3
aS'five'
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:

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

2. unpickling
loading an object from file:

```python
< 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:

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

2. unpickling
loading an object from file:

   ```python
   < 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.

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

```python
>>> import StringIO
>>> output = StringIO.StringIO()
>>> cPickle.dump(A,output)
>>> output.getvalue()
"(iclass_set\nSet\np1\n(dp2\nS's'
(lp3\nS'a'
aI3
a(iclass_set
Set
p4\n(dp5\nS's'
(lp6
I3
aS'five'
p7\nasbasb."

```
Writing to Strings
using the module `StringIO`

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

```python
>>> 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
Set
p1
(dp2
S's'
(lp3
S'a'
aI3
a(iclass_set
Set
p4
(dp5
S's'
(lp6
I3
aS'five'
p7
asbasb."
```
Writing to Strings
using the module `StringIO`

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

```python
>>> 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'
aI3
a(iclass_set
Set
p4\n(dp5\nS's'\n(lp6\nI3
aS'five'
p7\nasbasb."
```
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.