Outline

1. Modular Design
   programming in the large
   software engineering

2. Modules in Python
   importing modules
   stack of data

3. Summary + Assignments

MCS 260 Lecture 18
Introduction to Computer Science
Jan Verschelde, 6 October 2008
software engineering
modules in Python

1 Modular Design
programming in the large
software engineering

2 Modules in Python
importing modules
stack of data

3 Summary + Assignments
Modular Design
building large software systems

- we have practiced programming in the small
- programming in the large requires modular design
- characteristics of large programs:
  1. size: more than 100,000 lines of code
  2. effort: many teams of programmers
  3. time: program maintenance and evolution

- modular design of programs aims to control the complexity of a program by dividing it into *modules*
- a typical example of a module is a library of functions
Modular Design
building large software systems

- we have practiced programming in the small
- programming in the large requires modular design

- characteristics of large programs:
  1. size: more than 100,000 lines of code
  2. effort: many teams of programmers
  3. time: program maintenance and evolution

- modular design of programs aims to control the complexity of a program by dividing it into *modules*
- a typical example of a module is a library of functions
Modular Design
building large software systems

- we have practiced programming in the small
- programming in the large requires modular design
- characteristics of large programs:
  1. size: more than 100,000 lines of code
  2. effort: many teams of programmers
  3. time: program maintenance and evolution

- modular design of programs aims to control the complexity of a program by dividing it into *modules*
- a typical example of a module is a library of functions
Modular Design

building large software systems

- we have practiced programming in the small
- programming in the large requires modular design
- characteristics of large programs:
  1. size: more than 100,000 lines of code
  2. effort: many teams of programmers
  3. time: program maintenance and evolution

- modular design of programs aims to control the complexity of a program by dividing it into *modules*
- a typical example of a module is a library of functions
Modular Design
building large software systems

- we have practiced programming in the small
- programming in the large requires modular design
- characteristics of large programs:
  1. size: more than 100,000 lines of code
  2. effort: many teams of programmers
  3. time: program maintenance and evolution

- modular design of programs aims to control the complexity of a program by dividing it into *modules*
- a typical example of a module is a library of functions
Modular Design
building large software systems

- we have practiced programming in the small
- programming in the large requires modular design
- characteristics of large programs:
  1. size: more than 100,000 lines of code
  2. effort: many teams of programmers
  3. time: program maintenance and evolution

- modular design of programs aims to control the complexity of a program by dividing it into *modules*
- a typical example of a module is a library of functions
Design of Software Systems

Layers or Levels

Layers typical for almost any software system:

1. the *kernel* consists of basic functions
2. the *main operations* apply the kernel
3. the *user interface* defines how the user interacts with the software

The operating system is an example of a large software system. (See Lecture 5 for the 5 layers in an OS.)

For example, Sage consists of

1. components which focus on a particular area
2. ipython: modification of Python interpreter
3. notebook interface is GUI (Graphical User Interface)
Design of Software Systems

Layers or Levels

Layers typical for almost any software system:

1. the *kernel* consists of basic functions
2. the *main operations* apply the kernel
3. the *user interface* defines how the user interacts with the software

The operating system is an example of a large software system. (See Lecture 5 for the 5 layers in an OS.)

For example, Sage consists of

1. components which focus on a particular area
2. ipython: modification of Python interpreter
3. notebook interface is GUI (Graphical User Interface)
Design of Software Systems

Layers or Levels

Layers typical for almost any software system:

1. the *kernel* consists of basic functions
2. the *main operations* apply the kernel
3. the *user interface* defines how the user interacts with the software

The operating system is an example of a large software system. (See Lecture 5 for the 5 layers in an OS.)

For example, Sage consists of

1. components which focus on a particular area
2. ipython: modification of Python interpreter
3. notebook interface is GUI (Graphical User Interface)
Design of Software Systems

Layers or Levels

Layers typical for almost any software system:

1. the *kernel* consists of basic functions
2. the *main operations* apply the kernel
3. the *user interface* defines how the user interacts with the software

The operating system is an example of a large software system. (See Lecture 5 for the 5 layers in an OS.)

For example, Sage consists of

1. components which focus on a particular area
2. ipython: modification of Python interpreter
3. notebook interface is GUI (Graphical User Interface)
Design of Software Systems
Layers or Levels

Layers typical for almost any software system:

1. the *kernel* consists of basic functions
2. the *main operations* apply the kernel
3. the *user interface* defines how the user interacts with the software

The operating system is an example of a large software system. (See Lecture 5 for the 5 layers in an OS.)

For example, Sage consists of

1. components which focus on a particular area
2. ipython: modification of Python interpreter
3. notebook interface is GUI (Graphical User Interface)
Design of Software Systems

Layers or Levels

Layers typical for almost any software system:

1. the *kernel* consists of basic functions
2. the *main operations* apply the kernel
3. the *user interface* defines how the user interacts with the software

The operating system is an example of a large software system. (See Lecture 5 for the 5 layers in an OS.)

For example, Sage consists of

1. components which focus on a particular area
2. ipython: modification of Python interpreter
3. notebook interface is GUI (Graphical User Interface)
Design of Software Systems
Layers or Levels

Layers typical for almost any software system:
1. the *kernel* consists of basic functions
2. the *main operations* apply the kernel
3. the *user interface* defines how the user interacts with the software

The operating system is an example of a large software system. (See Lecture 5 for the 5 layers in an OS.)

For example, Sage consists of
1. components which focus on a particular area
2. ipython: modification of Python interpreter
3. notebook interface is GUI (Graphical User Interface)
Modular Design
programming in the large
software engineering

2 Modules in Python
importing modules
stack of data

3 Summary + Assignments
A software system consists of

1. a collection of modules; and
2. the relations between the modules.

Modular design defines the decomposition of a system into modules. A module can have modular components.

Each module has an **interface** and a **body**:  

- **interface** is the set of all elements in a module available to all users of the module, also called the module’s **exported resources**
- **body** is what realizes the functionalities of a module, also called the **implementation**.

A module *imports* resources from another module. A module *exports* resources via its interface.
A software system consists of

1. a collection of modules; and
2. the relations between the modules.

Modular design defines the decomposition of a system into modules. A module can have modular components.

Each module has an **interface** and a **body**:
- **interface** is the set of all elements in a module available to all users of the module, also called the module’s **exported resources**
- **body** is what realizes the functionalities of a module, also called the **implementation**.

A module *imports* resources from another module. A module *exports* resources via its interface.
A software system consists of

1. a collection of modules; and
2. the relations between the modules.

Modular design defines the decomposition of a system into modules. A module can have modular components.

Each module has an **interface** and a **body**:

- **interface** is the set of all elements in a module available to all users of the module, also called the module’s **exported resources**

- **body** is what realizes the functionalities of a module, also called the **implementation**.

A module **imports** resources from another module. A module **exports** resources via its interface.
Modular Design
programming in the large
software engineering

Modules in Python
importing modules
stack of data

Summary + Assignments

A software system consists of

1. a collection of modules; and
2. the relations between the modules.

Modular design defines the decomposition of a system into modules. A module can have modular components.

Each module has an **interface** and a **body**:

**interface** is the set of all elements in a module available to all users of the module, also called the module’s **exported resources**

**body** is what realizes the functionalities of a module, also called the **implementation**.

A module *imports* resources from another module. A module *exports* resources via its interface.
A software system consists of

1. a collection of modules; and
2. the relations between the modules.

Modular design defines the decomposition of a system into modules. A module can have modular components.

Each module has an **interface** and a **body**:

**interface** is the set of all elements in a module available to all users of the module, also called the module’s **exported resources**

**body** is what realizes the functionalities of a module, also called the **implementation**.

A module **imports** resources from another module. A module **exports** resources via its interface.
A software system consists of

1. a collection of modules; and
2. the relations between the modules.

Modular design defines the decomposition of a system into modules. A module can have modular components.

Each module has an **interface** and a **body**:

- **interface** is the set of all elements in a module available to all users of the module, also called the module’s **exported resources**
- **body** is what realizes the functionalities of a module, also called the **implementation**.

A module *imports* resources from another module. A module *exports* resources via its interface.
Modules

components of software systems

A software system consists of

1. a collection of modules; and
2. the relations between the modules.

Modular design defines the decomposition of a system into modules. A module can have modular components.

Each module has an **interface** and a **body**:

- **interface** is the set of all elements in a module available to all users of the module, also called the module’s **exported resources**
- **body** is what realizes the functionalities of a module, also called the **implementation**.

A module **imports** resources from another module. A module **exports** resources via its interface.
Principles of Modular Design

criteria for good software engineering

The software architect designs the system architecture. The system architecture represents the decomposition of the system into modules and the intermodule relations.

A first recommendation to design modular software:

- Information Hiding
  The interface must be separated from the body. Programs that rely on the module via its interface do not have to be rewritten as the body changes.

This principle implies that the interface of a module contains the right kind of information.

Users who need to manipulate polynomials should not be required to take into account the internal data structures used to represent the polynomials.
Principles of Modular Design
criteria for good software engineering

The software architect designs the system architecture. The system architecture represents the decomposition of the system into modules and the intermodule relations.

A first recommendation to design modular software:

- Information Hiding
  The interface must be separated from the body. Programs that rely on the module via its interface do not have to be rewritten as the body changes.

This principle implies that the interface of a module contains the right kind of information.

Users who need to manipulate polynomials should not be required to take into account the internal data structures used to represent the polynomials.
Principles of Modular Design

criteria for good software engineering

The software architect designs the system architecture. The system architecture represents the decomposition of the system into modules and the intermodule relations.

A first recommendation to design modular software:

- Information Hiding
  The interface must be separated from the body. Programs that rely on the module via its interface do not have to be rewritten as the body changes.

This principle implies that the interface of a module contains the right kind of information.

Users who need to manipulate polynomials should not be required to take into account the internal data structures used to represent the polynomials.
Bottom-Up Design of Programs
principle of low coupling and high cohesion

Modules are implemented by different teams of programmers, often working over different time periods. The functionality of a module must be ready for testing and verification independently of the rest of the program.

A second recommendation to design modular software:

- Low Coupling and High Cohesion
  Low coupling means that modules are largely independent from each other.
  Functions often used together belong to the same module so each module has a high internal cohesion.

A module to manipulate polynomials should collect all the operations needed in the software system.
Bottom-Up Design of Programs
principle of low coupling and high cohesion

Modules are implemented by different teams of programmers, often working over different time periods. The functionality of a module must be ready for testing and verification independently of the rest of the program.

A second recommendation to design modular software:

- Low Coupling and High Cohesion
  Low coupling means that modules are largely independent from each other.

Functions often used together belong to the same module so each module has a high internal cohesion.

A module to manipulate polynomials should collect all the operations needed in the software system.
Bottom-Up Design of Programs
principle of low coupling and high cohesion

Modules are implemented by different teams of programmers, often working over different time periods. The functionality of a module must be ready for testing and verification independently of the rest of the program.

A second recommendation to design modular software:

- Low Coupling and High Cohesion
  Low coupling means that modules are largely independent from each other.

  Functions often used together belong to the same module so each module has a high internal cohesion.

A module to manipulate polynomials should collect all the operations needed in the software system.
Bottom-Up Design of Programs
principle of low coupling and high cohesion

Modules are implemented by different teams of programmers, often working over different time periods. The functionality of a module must be ready for testing and verification independently of the rest of the program.

A second recommendation to design modular software:

- **Low Coupling and High Cohesion**
  Low coupling means that modules are largely independent from each other.

Functions often used together belong to the same module so each module has a high internal cohesion.

A module to manipulate polynomials should collect all the operations needed in the software system.
Software development is an expensive process...

A third recommendation to design modular software:

- Design for Change
  For example, use of parameters and constants for data that may later change.

For modules to manipulate polynomials, we foresee that different coefficient fields could be needed.

Object-oriented design is typically bottom up and leads to reusable software.

We will cover object-oriented programming in Python.
Software development is an expensive process...

A third recommendation to design modular software:

- Design for Change
  For example, use of parameters and constants for data that may later change.

For modules to manipulate polynomials, we foresee that different coefficient fields could be needed.

Object-oriented design is typically bottom up and leads to reusable software.

We will cover object-oriented programming in Python.
Reuse of Modules
standard libraries of components

Software development is an expensive process...

A third recommendation to design modular software:

- Design for Change
  For example, use of parameters and constants for data that may later change.

For modules to manipulate polynomials, we foresee that different coefficient fields could be needed.

Object-oriented design is typically bottom up and leads to reusable software.

We will cover object-oriented programming in Python.
Modular Design
programming in the large
software engineering

Modules in Python
importing modules
stack of data

Summary + Assignments
The syntax to import a module is

```python
import < module >
```

**For example:** `import math`.

Then we can compute $\sqrt{2}$ via `math.sqrt(2)`.

If we only need one element of a module:

```python
from < module > import < element >
```

**For example:** `from math import sqrt`.

Then we can compute $\sqrt{2}$ simply as `sqrt(2)`.

If `import math` is successful, then `help(math)` or `help(math.cos)` shows information about the module `math` or the function `math.cos`. 
The syntax to import a module is

```python
import < module >
```

For example: `import math`.

Then we can compute $\sqrt{2}$ via `math.sqrt(2)`.

If we only need one element of a module:

```python
from < module > import < element >
```

For example: `from math import sqrt`.

Then we can compute $\sqrt{2}$ simply as `sqrt(2)`.

If `import math` is successful, then `help(math)` or `help(math.cos)` shows information about the module `math` or the function `math.cos`. 
The syntax to import a module is

`import < module >`

For example: `import math`. Then we can compute \( \sqrt{2} \) via `math.sqrt(2)`.

If we only need one element of a module:

`from < module > import < element >`

For example: `from math import sqrt`. Then we can compute \( \sqrt{2} \) simply as `sqrt(2)`.

If `import math` is successful, then `help(math)` or `help(math.cos)` shows information about the module `math` or the function `math.cos`.
software engineering
modules in Python

1 Modular Design
programming in the large
software engineering

2 Modules in Python
importing modules
stack of data

3 Summary + Assignments
Suppose we want a stack of data as data structure.

The bottom up design hides the internal representation from the program `stack_user`. Four `.py` files:

```
stack_user

stack_of_data

stack_data

stack_of_data_io
```

The module `stack_data` is just one line:

```python
the_stack = []
```
A Stack of Data
bottom up design

Suppose we want a stack of data as data structure.

The bottom up design hides the internal representation from the program stack_user. Four .py files:

```
stack_user

stack_of_data

stack_of_data_io

stack_data
```

The module stack_data is just one line:
```
the_stack = []
```
Suppose we want a stack of data as data structure.

The bottom up design hides the internal representation from the program `stack_user`. Four `.py` files:

```
<table>
<thead>
<tr>
<th>stack_user</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack_of_data</td>
</tr>
<tr>
<td>stack_of_data_io</td>
</tr>
<tr>
<td>stack_data</td>
</tr>
</tbody>
</table>

The module `stack_data` is just one line:
```
the_stack = []
```
A Stack of Data
bottom up design

Suppose we want a stack of data as data structure.

The bottom up design hides the internal representation from
the program stack_user. Four .py files:

```
stack_user

stack_of_data

stack_of_data_io

stack_data
```

The module stack_data is just one line:

```
the_stack = []
```
Suppose we want a stack of data as data structure.

The bottom up design hides the internal representation from the program `stack_user`. Four `.py` files:

```
stack_user

stack_of_data

stack_data

stack_of_data_io
```

The module `stack_data` is just one line:
```
the_stack = []
```
Suppose we want a stack of data as data structure. The bottom up design hides the internal representation from the program `stack_user`. Four `.py` files:

The module `stack_data` is just one line:
```
the_stack = []
```
Interface of stack_of_data
exported resources

Function definitions in the file `stack_of_data.py`:

```python
def push(item):
    "pushes the item on the stack"

def length():
    "returns the length of the stack"

def pop():
    "returns an item off the stack"
```

This is the view offered to `stack_user`. The user does not know that a list stores the stack.
Interface of stack_of_data
exported resources

Function definitions in the file stack_of_data.py:

def push(item):
    "pushes the item on the stack"

def length():
    "returns the length of the stack"

def pop():
    "returns an item off the stack"

This is the view offered to stack_user. The user does not know that a list stores the stack.
Body of stack_of_data

an implementation of the module

from stack_data import the_stack

def push(item):
    "pushes the item on the stack"
    the_stack.insert(0, item)

def length():
    "returns the length of the stack"
    return len(the_stack)

def pop():
    "returns an item off the stack"
    return the_stack.pop(0)

The first import statement is *global*: it makes the_stack available to all functions.
For *local* use, move the import into a function definition.
Body of stack_of_data
an implementation of the module

from stack_data import the_stack

def push(item):
    "pushes the item on the stack"
    the_stack.insert(0,item)

def length():
    "returns the length of the stack"
    return len(the_stack)

def pop():
    "returns an item off the stack"
    return the_stack.pop(0)

The first import statement is global: it makes the_stack available to all functions.
For local use, move the import into a function definition.
importing modules

from stack_data import the_stack

def push(item):
    "pushes the item on the stack"
    the_stack.insert(0,item)

def length():
    "returns the length of the stack"
    return len(the_stack)

def pop():
    "returns an item off the stack"
    return the_stack.pop(0)

The first import statement is *global*: it makes *the_stack* available to all functions.
For *local* use, move the import into a function definition.
Body of stack_of_data
an implementation of the module

from stack_data import the_stack

def push(item):
    "pushes the item on the stack"
    the_stack.insert(0, item)

def length():
    "returns the length of the stack"
    return len(the_stack)

def pop():
    "returns an item off the stack"
    return the_stack.pop(0)

The first import statement is global: it makes the_stack available to all functions.
For local use, move the import into a function definition.
Body of stack_of_data
an implementation of the module

from stack_data import the_stack

def push(item):
    "pushes the item on the stack"
    the_stack.insert(0, item)

def length():
    "returns the length of the stack"
    return len(the_stack)

def pop():
    "returns an item off the stack"
    return the_stack.pop(0)

The first import statement is global: it makes the_stack available to all functions.
For local use, move the import into a function definition.
**Interface of stack_of_data_io**
exported resources for input and output

The information for `help(stack_of_data_io)`: 

```python
def input(L):
    "places elements of L on stack"

def show_all():
    "shows all elements in the stack"

def show_top():
    "shows the top element of the stack"
```

This input/output module is necessary, as the user of the stack does not have access to the list `the_stack`. 
Interface of stack_of_data_io
exported resources for input and output

The information for `help(stack_of_data_io)`: 

```python
def input(L):
    "places elements of L on stack"

def show_all():
    "shows all elements in the stack"

def show_top():
    "shows the top element of the stack"
```

This input/output module is necessary, as the user of the stack does not have access to the list `the_stack`. 
Body of stack_of_data_io
an implementation of the module

from stack_data import the_stack

def input(L):
    "places elements of L on stack"
    for item in L:
        the_stack.insert(0,item)

def show_all():
    "shows all elements in the stack"
    print the_stack

def show_top():
    "shows the top element of the stack"
    if len(the_stack) > 0:
        print the_stack[0]
    else:
        print 'the stack is empty'
Body of stack_of_data_io

an implementation of the module

from stack_data import the_stack

def input(L):
    "places elements of L on stack"
    for item in L:
        the_stack.insert(0, item)

def show_all():
    "shows all elements in the stack"
    print the_stack

def show_top():
    "shows the top element of the stack"
    if len(the_stack) > 0:
        print the_stack[0]
    else:
        print 'the stack is empty'
Body of stack_of_data_io
an implementation of the module

from stack_data import the_stack

def input(L):
    "places elements of L on stack"
    for item in L:
        the_stack.insert(0,item)

def show_all():
    "shows all elements in the stack"
    print the_stack

def show_top():
    "shows the top element of the stack"
    if len(the_stack) > 0:
        print the_stack[0]
    else:
        print 'the stack is empty'
Body of stack_of_data_io
an implementation of the module

from stack_data import the_stack

def input(L):
    "places elements of L on stack"
    for item in L:
        the_stack.insert(0,item)

def show_all():
    "shows all elements in the stack"
    print the_stack

def show_top():
    "shows the top element of the stack"
    if len(the_stack) > 0:
        print the_stack[0]
    else:
        print 'the stack is empty'
The User Program

To test the module `stack_of_data`, we use an interactive program, with structure:

```python
import stack_of_data
import stack_of_data_io

def pop_or_push(choice):
    "calls pop and push functions"

def test_input_output():
    "test input/output operations"

while True:
    menu = 'add, remove, or stop? (a/r/s) '  
    choice = raw_input(menu)  
    if choice == 's': break  
    pop_or_push(choice)  
test_input_output()
```
The User Program

To test the module `stack_of_data`, we use an interactive program, with structure:

```python
import stack_of_data
import stack_of_data_io

def pop_or_push(choice):
    "calls pop and push functions"

def test_input_output():
    "test input/output operations"

while True:
    menu = 'add, remove, or stop? (a/r/s) '
    choice = raw_input(menu)
    if choice == 's': break
    pop_or_push(choice)
    test_input_output()
```
The User Program

To test the module `stack_of_data`, we use an interactive program, with structure:

```python
import stack_of_data
import stack_of_data_io

def pop_or_push(choice):
    "calls pop and push functions"

def test_input_output():
    "test input/output operations"

while True:
    menu = 'add, remove, or stop? (a/r/s) '
    choice = raw_input(menu)
    if choice == 's': break
    pop_or_push(choice)
    test_input_output()
```
The User Program

To test the module `stack_of_data`, we use an interactive program, with structure:

```python
import stack_of_data
import stack_of_data_io

def pop_or_push(choice):
    "calls pop and push functions"

def test_input_output():
    "test input/output operations"

while True:
    menu = 'add, remove, or stop? (a/r/s) '
    choice = raw_input(menu)
    if choice == 's': break
    pop_or_push(choice)
    test_input_output()
```
Modifications to the Stack

definition of pop_or_push(choice):

    "calls pop and push functions"
    if choice == 'a':
        item = input('Give an item : ')
        stack_of_data.push(item)
    elif choice == 'r':
        item = stack_of_data.pop()
        print 'item popped : %d' % item
    else:
        print 'invalid choice, try again'
def test_input_output():
    "test input/output operations"
    K = input('give a list : ')
    stack_of_data_io.input(K)
    L = stack_of_data.length()
    print 'length of stack : %d' % L
    print 'printing top element'
    stack_of_data_io.show_top()
    print 'printing the stack'
    stack_of_data_io.show_all()
Summary + Assignments

Background reading for this lecture:

- chapter 7 in *Python Power!*
- section 7.4 in *Computer Science, an overview*

Assignments:

1. Describe the cohesion and coupling for a novel and a textbook. Compare the differences in degrees of cohesion and coupling for both.

2. Modify the modular design of the stack into a module to represent a queue of data. Provide the operations `enqueue` and `dequeue` in the module `queue_of_data` to respectively add and remove elements. Define input and output and test the module interactively in a program.

Homework collection on Mon 13 Oct at 1PM: #2 of Lecture 10, #3 of Lecture 11, #1,2 of Lecture 12, #1 of Lecture 13.