Outline

1. Modular Design
   - programming in the large
   - software engineering
2. Modules in Python
   - importing modules
   - stack of data
3. Good Design in Action
   - choosing between two designs
   - justifying the right choice
4. Summary + Assignments

MCS 260 Lecture 21
Introduction to Computer Science
Jan Verschelde, 29 February 2016

software engineering
modules in Python

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.

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)

software engineering modules in Python

1. Modular Design
   - programming in the large
   - software engineering
2. Modules in Python
   - importing modules
   - stack of data
3. Good Design in Action
   - choosing between two designs
   - justifying the right choice
4. Summary + Assignments

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.

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.

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.

software engineering
modules in Python

modules in Python – importing modules

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)`.

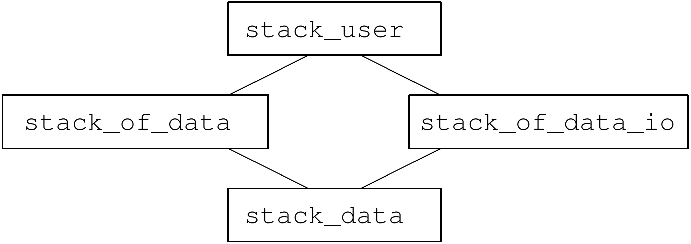
If `import math` is successful, then `help(math)` or `help(math.cos)` shows information about the module `math` or the function `math.cos`.

software engineering modules in Python

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:



The module `stack_data` is just one line:

```
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

```
from stack_data import STACK

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

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

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

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

interface of stack_of_data_io – exported resources

The information for `help(stack_of_data_io)`:

```
def push_all(elements):
    """
    pushes elements of a list to the 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 the stack does not have access to the list `STACK`.

body of stack_of_data_io

```
from stack_data import STACK

def push_all(elements):
    """
    pushes elements of a list to the stack"
    """
    for item in elements:
        STACK.insert(0, item)

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

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

testing the module

```
import stack_of_data
import stack_of_data_io

def pop_or_push(choice):
    """
    Calls pop and push functions.
    """
def test_input_output():
    """
    Tests input/output operations.
    """

while True:
    MENU = 'add, remove, or stop? (a/r/s) '
    CHOICE = input(MENU)
    if CHOICE == 's':
        break
    pop_or_push(CHOICE)
    test_input_output()
```

modifications: definition of pop_or_push(choice)

```
def pop_or_push(choice):
    """
    Calls pop and push functions.
    """
    if choice == 'a':
        item = input('Give an item : ')
        data = literal_eval(item)
        stack_of_data.push(data)
    elif choice == 'r':
        item = stack_of_data.pop()
        print('item popped : %d' % item)
    else:
        print('invalid choice, try again')
```

input and output: definition of test_input_output()

```
def test_input_output():
    """
    Tests input/output operations.
    """
    items = input('give a list : ')
    data = literal_eval(items)
    stack_of_data_io.push_all(data)
    lenstk = stack_of_data.length()
    print('length of stack : %d' % lenstk)
    print('printing top element')
    stack_of_data_io.show_top()
    print('printing the stack')
    stack_of_data_io.show_all()
```

software engineering modules in Python

making a good design

Consider the modular structure of a program to compose music for a band consisting of guitar, drum, and piano.
Each instrument comes with `in` and `out` functions.
The `in` function takes instructions and simulates the sound.
The `out` function prints instructions for the musician.

We can design the program in two ways:

1. There are two modules: `input` and `output`.
   The `input` module collects all `in` functions of the instrument.
   All `out` functions are in `output`.
2. There is a separate module for each instrument.
   Each module contains the `in` and `out` functions for the instrument.

Which design would be best?
Justify using the principles of good modular design.

software engineering
modules in Python

1. Modular Design
   - programming in the large
   - software engineering
2. Modules in Python
   - importing modules
   - stack of data
3. Good Design in Action
   - choosing between two designs
   - justifying the right choice
4. Summary + Assignments

justifying the right choice

The second design works best.

Justification along the three principles:

1. information hiding,
2. high cohesion and low coupling,
3. design for change.

Information Hiding

The programmer of a module in the first design needs to know the ins and outs of each instrument,
while in the second design, a programmer of a module can focus on one instrument.

In the second design, each module hides the details about its instrument to the other modules.

The Second Principle

high cohesion and low coupling

There is high cohesion in the second design *because* all functionality about one instrument remains in one module,

while the functionality of one instrument is spread out over several modules in the first design.

In the first design there is high coupling *because* the instructions for the input to simulate the sounds will be similar to what will be given to the musicians.

There is low coupling in the second design, *because* the programmer can share conventions in input and output routines, proper for each instrument.

The Third Principle

design for change

If another instrument is added to the band . . .

- In the second design,
  we have to add only another module,
  the existing modules remain the same.
- In the first design,
  we need to change all existing modules.

Summary + Assignments

Read §7.4 in *Computer Science, an overview.*

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. Define in the module `queue_of_data` the operations `enqueue` and `dequeue` to respectively add and remove elements. Define input/output and write a test program.
3. Design a program to search a phone directory. Users can enter a name (or a telephone number) and the program will then search for the corresponding telephone number (or name). Draw your modular design. For each module describe what functions are exported and what is imported. Justify your design, referring to the three key principles of good design.