# Data structure and algorithm in Python

## Object-Oriented Programming

Seongjin Lee

Dept of Aerospace and Software Engineering,
Gyeongsang National University

# Table of contents

# Goals, Principles, and Patterns

The main "actors" in the object-oriented paradigm are called **objects**.

- Each object is an **instance** of a **class**.
- The class definition specifies **instance variables (data members)** that the object contains, as well as the **methods (member functions)**, that the object can execute.

# Goals, Principles, and Patterns

## Object-Oriented Design Goals

# Object-Oriented Design Goals

Software implementations should achieve robustness, adaptability, and reusability.

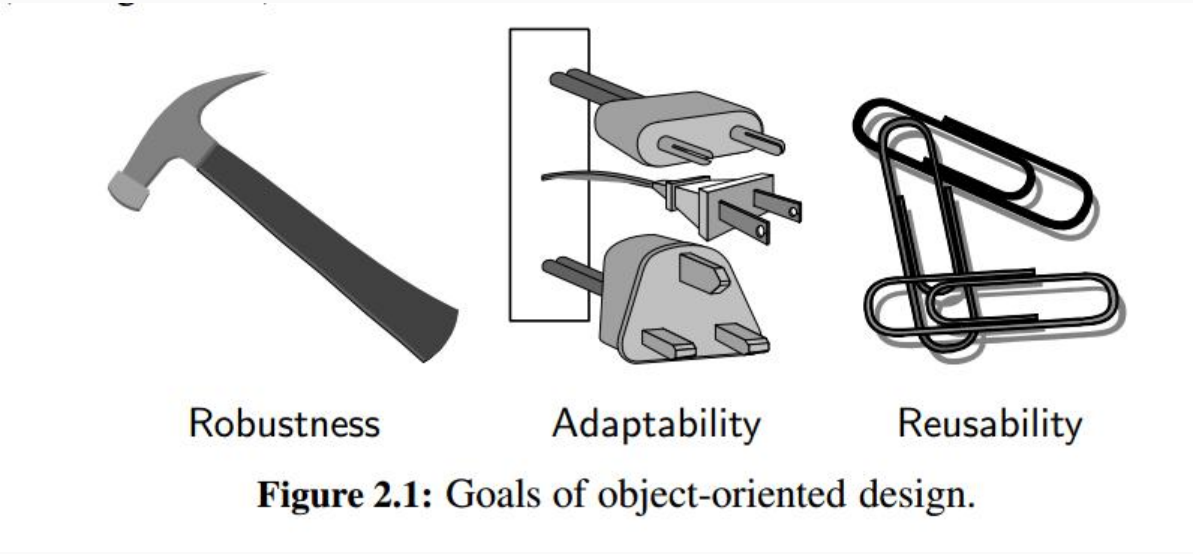



**Figure 2.1:** Goals of object-oriented design.

# Object-Oriented Design Goals: Robustness

**Definition 1.1: Robustness**

A software is capable of handling unexpected inputs that are not explicitly defined for its application.

**Example 1.1**

If a program is expecting a positive integer and instead is given a negative integer, then the program should be able to recover gracefully from this error.

# Object-Oriented Design Goals: Robustness

**Example 1.2**

In **life-critical** applications, where a software error can lead to injury or loss of life, software that is not robust could be deadly. This point was driven home in the late 1980s in accidents involving Therac-25, a radiation-therapy machine, which severely overdosed six patients between 1985 and 1987, some of whom died from complications resulting from their radiation overdose. All six accidents were traced to software errors.

# Object-Oriented Design Goals: Adaptability

**Definition 1.2: Adaptability**

A software needs to be able to evolve over time in response to changing conditions in its environment.

**Definition 1.3: Portability**

A software is able to run with minimal change on different hardware and operating platforms.

An advantage of writing software in Python is the portability provided by the language itself.

# Object-Oriented Design Goals: Reusability

**Definition 1.4: Reusability**

The same code should be usable as a component of different system in various applications.

**Example 1.3**

Developing quality software can be an expensive enterprise, and its cost can be offset somewhat if the software is designed in a way that makes it easily reusable in future applications.

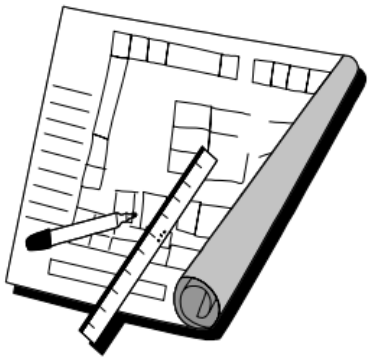
# Goals, Principles, and Patterns

## Object-Oriented Design Priciples

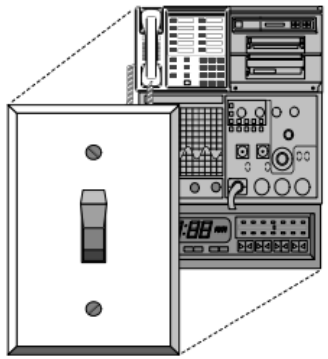
# Object-Oriented Design Priciples

- Modularity
- Abstraction
- Encapsulation

Modularity



Abstraction



Encapsulation

**Figure 2.2:** Principles of object-oriented design.

# Object-Oriented Design Priciples

**Definition 1.5: Modularity**

It refers to an organizing principle in which different components of a software system are divided into separate functional units.

**Example 1.4**

A **house or apartment** can be viewed as consisting of several interacting units:

- electrical
- heating and cooling
- plumbing
- structural

**Definition 1.6: Module**

A **module** is a collection of closely related functions and classes that are defined together in a single file of source code.

**Example 1.5**

Python's standard libraries include, for example,

- the **math** module, which provides definitions for key mathematical constants and functions,
- the **os** module, which provides support for interacting with the operating system.

**Definition 1.6: Module**

A **module** is a collection of closely related functions and classes that are defined together in a single file of source code.

**Example 1.5**

Python's standard libraries include, for example,

- the **math** module, which provides definitions for key mathematical constants and functions,
- the **os** module, which provides support for interacting with the operating system.

The use of modularity helps support the goals: **Robustness, Adaptability, Reusability**.

- Robustness is greatly increased because it is easier to test and debug separate components before they are integrated into a larger software system. Furthermore, bugs that persist in a complete system might be traced to a particular component, which can be fixed in relative isolation.
- The structure imposed by modularity also helps enable software reusability. If software modules are written in a general way, the modules can be reused when related need arises in other contexts. This is particularly relevant in a study of data structures, which can typically be designed with sufficient abstraction and generality to be reused in many applications.

# Object-Oriented Design Priciples

The notion of abstraction is to distill a complicated system down to its most fundamental parts. Typically, describing the parts of a system involves naming them and explaining their functionality.

**Definition 1.7: Abstract Data Types (ADT)**

An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations. An ADT specifies what each operation does, but not how it does it.

We will typically refer to the collective set of behaviors supported by an ADT as its **public interface**.

# Object-Oriented Design Priciples

Python has a tradition of treating abstractions implicitly using a mechanism known as **duck typing**.

As an interpreted and dynamically typed language, there is no "compile time" checking of data types in Python, and no formal requirement for declarations of abstract base classes.

**Example 1.6: Duck Typing**

```
class Duck():
    def walk(self):
        print('I walk like a duck')
    def swim(self):
        print('I swim like a duck')
class Person():
    def walk(self):
        print('This man walk like a duck')
    def swim(self):
        print('This man swim like a duck')
obj = Duck
obj().swim()
obj = Person
obj().swim()
```

**Example 1.6: Duck Typing**

```
class Duck():
    def walk(self):
        print('I walk like a duck')
    def swim(self):
        print('I swim like a duck')
class Person():
    def walk(self):
        print('This man walk like a duck')
    def swim(self):
        print('This man swim like a duck')
obj = Duck
obj().swim()
obj = Person
obj().swim()
```

The description of this as "duck typing" comes from an adage attributed to poet James Whitcomb Riley, stating that "when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

# Object-Oriented Design Priciples

Another important principle of object-oriented design is **encapsulation**.

- It gives one programmer freedom to implement the details of a component, without concern that other programmers will be writing code that intricately depends on those internal decisions.
- The programmer of a component only need to maintain the public interface for the component, as other programmers will be writing code that depends on that interface.
- It yields robustness and adaptability, for it allows the implementation details of parts of a program to change without adversely affecting other parts, thereby making it easier to fix bugs or add new functionality with relatively local changes to a component.

# Goals, Principles, and Patterns

## Design Patterns

# Design Patterns

A ***design pattern*** describes a solution to a "typical" software design problem.

A pattern

- provides a general template for a solution that can be applied in many different situations.
- describes the main elements of a solution in an abstract way that can be specialized for a specific problem at hand.
- consists of
  - a name: identifies the pattern
  - a context: describes the scenarios for which this pattern can be applied
  - a template: describes how the pattern is applied
  - a result: describes and analyzes what pattern produces

# Design Patterns

Several design patterns: falling into two groups

- patterns for solving algorithm design problems
- patterns for solving software engineering problems

# Design Patterns

The algorithm design patterns

- Recursion
- Amortization
- Divide-and-conquer
- Prune-and-search, also known as decrease-and-conquer
- Brute force
- Dynamic programming
- The greedy method

# Design Patterns

The software engineering design patterns

- Iterator
- Adapter
- Position
- Composition
- Template method
- Locator
- Factory method

# Software Development

# Software Development

Traditional software development involves several phases. Three major steps are:

1. Design
2. Implementation
3. Testing and Debugging

# Software Development

## Design

# Software Development

For object-oriented programming, the design step is perhaps the most important phase in the process of developing software.

- how to divide the workings of our program into classes,
- how these classes will interact,
- what data each will store,
- what actions each will perform

There are some rules of thumb that we can apply when determining how to design our classes:

- **Responsibilities**: Divide the work into different actors, each with a different responsibility.
- **Independence**: Define the work for each class to be as independent from other classes as possible.
- **Behaviors**: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.

There are some rules of thumb that we can apply when determining how to design our classes:

- **Responsibilities**: Divide the work into different actors, each with a different responsibility.
- **Independence**: Define the work for each class to be as independent from other classes as possible.
- **Behaviors**: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.

- **Responsibilities**: Divide the work into different actors, each with a different responsibility.

  Try to describe responsibilities using action verbs. These actors will form the classes for the program.
- **Independence**: Define the work for each class to be as independent from other classes as possible.

  Subdivide responsibilities between classes so that each class has autonomy over some aspect of the program. Give data to the class that has jurisdiction over the actions that require access to this data.
- **Behaviors**: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.

# Design

| Class: | CreditCard | |
|---|---|---|
| Fields: | `_customer` | `_balance` |
| | `_bank` | `_limit` |
| | `_account` | |
| Behaviors: | `get_customer()` | `get_balance()` |
| | `get_bank()` | `get_limit()` |
| | `get_account()` | `charge(price)` |
| | `make_payment(amount)` | |

# Software Development

## Coding Style and Documentation

# Coding Style and Documentation

Programs should be made easy to read and understand. Good programmers should therefore be mindful of their coding style, and develop a style that communicates the important aspects of a program's design for both humans and computers.

# Coding Style and Documentation

The main principles:

(1) Python code blocks are typically indented by 4 spaces.
   - avoid the use of tabs
   - Many Python-aware editors will automatically replace tabs with an appropriate number of spaces.

(2) Use meaningful names for identifiers. Try to choose names that can be read aloud, and choose names that reflect the action, responsibility, or data each identifier is naming.

# Coding Style and Documentation

- **Classes**:
  - have a name serves as a singular noun
  - should be capitalized
  - When multiple words are concatenated to form a class name, they should follow the so-called "CamelCase" convention in which the first letter of each word is capitalized (e.g., CreditCard).

# Coding Style and Documentation

- **Functions, including member functions**:
  - should be lowercase
  - If multiple words are combined, they should be separated by underscores (e.g., `make_payment`).
  - The name of a function should typically be a verb that describes its affect.

    if the only purpose of the function is to return a value, the function name may be a noun that describes the value (e.g., `sqrt` rather than `calculate_sqrt`).

# Coding Style and Documentation

- **Names that identify an dividual object** (e.g., a parameter, instance variable, or local variable):
  - should be a lowercase noun (e.g., `price`)

    Occasionally, we stray from this rule when using a single uppercase letter to designate the name of a data structures (such as tree `T`).
- **Identifiers that represent a constant value**
  - should use all captical letters and with underscores to seperate words (e.g., `MAX_SIZE`).
- **Identifiers in any context that begin with a single leading underscore (e.g., `_secret`)**
  - are intended to suggest that they are only for "internal" use to a class or module, and not part of a public interface.

# Coding Style and Documentation

(3) Use comments that add meaning to a program and explain ambiguous or confusing constructs.

- In-line comments are good for quick explanations;

```
if n % 2 == 1:   # n is odd
```

- Multiline block comments are good for explaining more complex code sections.

  In Python, these are technically multiline string literals, typically delimited with triple quotes (""""), which have no effect when executed.

# Documentation

Python provides integrated support for embedding formal documentation directly in source code using a mechanism known as a ***docstring***.

Formally, any string literal that appears as the first statement within the body of a module, class, or function (including a member function of a class) will be considered to be a docstring. By convention, those string literals should be delimited within triple quotes (""").

# Documentation

```
def scale(data, factor):
    """Multiply all entries of numeric data list
     by the given factor."""
    for j in range(len(data)):
        data[j] = factor
```

```
>>> import scale1
>>> help(scale1.scale)
Help on function scale in module scale1:

scale(data, factor)
    Multiply all entries of numeric data list by
     the given factor.
```

# Documentation

```
def scale(data, factor):
    """Multiply all entries of numeric data list by the given factor.

    data     an instance of any mutable sequence type (such as a list)
    containing numeric elements
    factor   a number that serves as the multiplicative factor for scaling
    """
    for j in range(len(data)):
        data[j] = factor
```

```
>>> from scale2 import scale
>>> help(scale)
Help on function scale in module scale2:

scale(data, factor)
    Multiply all entries of numeric data list by the given factor.

    data     an instance of any mutable sequence type (such as a list)
    containing numeric elements
    factor   a number that serves as the multiplicative factor for scaling
```

# Software Development

## Testing and Debugging

# Testing and Debugging

- **Testing**: the process of experimentally checking the correctness of a program
- **Debugging**: the process of tracking the execution of a program and discovering the errors in it.

Testing and debugging are often the most time-consuming activity in the development of a program.

# Testing

A careful testing plan is an essential part of writing a program.

**Example 2.1**

While verifying the correctness of a program over all possible inputs is usually infeasible, we should

- aim at executing the program on a **representative subset of inputs**;
- **method coverage**: at the very minimum, we should make sure that every method of a class is tested at least once;
- **statement coverage**: even better, each code statement in the program should be executed at least once.

# Testing

Programs often tend to fail on **special cases** of the input. Such cases need to be carefully identified and tested.

**Example 2.2**

When testing a method that **sorts** (that is, puts in order) a sequence of integers, we should consider the following inputs:

- The sequence has zero length (no elements).
- The sequence has one element.
- All the elements of the sequence are the same.
- The sequence is already sorted.
- The sequence is reverse sorted.

# Testing

In addition to special inputs to the program, we should also consider **special conditions** for the structures used by the program.

**Example 2.3**

if we use a Python list to store data, we should make sure that boundary cases, such as inserting or removing at the beginning or end of the list, are properly handled.

# Testing

While it is essential to use handcrafted test suites, it is also advantageous to run the program on a large collection of randomly generated inputs. The random module in Python provides several means for generating random numbers, or for randomizing the order of collections.

# Debugging

- Simplest way: using **print statements**.
- Better way: Using **Debugger**, such as **pdb** and **debugging enviroments** provided by most IDEs.

# Class Definitions

# Class Definitions

A class serves as the primary means for abstraction in object-oriented programming. In Python, every piece of data is represented as an instance of some class.

A class

- provides a set of behaviors in the form of **member functions** (also known as **methods**), with implementations that are common to all instances of that class.
- serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, or **data members**).

# Class Definitions

## Example: CreditCard Class

# Example: CreditCard Class I

```
class CreditCard:
  """A consumer credit card."""

  def __init__(self, customer, bank, acnt, limit):
    """Create a new credit card instance.

    The initial balance is zero.

    customer  the name of the customer
              (e.g., 'John Bowman')
    bank      the name of the bank
              (e.g., 'California Savings')
    acnt      the acount identifier
              (e.g., '5391 0375 9387 5309')
    limit     credit limit (measured in dollars)
    """
    self._customer = customer
    self._bank = bank
```

# Example: CreditCard Class II

```
    self._account = acnt
    self._limit = limit
    self._balance = 0

  def get_customer(self):
    """Return name of the customer."""
    return self._customer

  def get_bank(self):
    """Return the bank's name."""
    return self._bank

  def get_account(self):
    """Return the card identifying number (typically stored as a
    string)."""
    return self._account

  def get_limit(self):
    """Return current credit limit."""
```

# Example: CreditCard Class III

```
    return self._limit

  def get_balance(self):
    """Return current balance."""
    return self._balance

  def charge(self, price):
    """Charge given price to the card, assuming sufficient credit
    limit.

    Return True if charge was processed; False if charge was denied.
    """
    if price + self._balance > self._limit:  # if charge would
    exceed limit,
      return False                           # cannot accept charge
    else:
      self._balance += price
      return True
```

# Example: CreditCard Class IV

```
  def make_payment(self, amount):
    """Process customer payment that reduces balance."""
    self._balance -= amount

if __name__ == '__main__':
  wallet = []
  wallet.append(
    CreditCard('John Bowman', 'California Savings',
               '5391 0375 9387 5309', 2500) )
  wallet.append(
    CreditCard('John Bowman', 'California Federal',
               '3485 0399 3395 1954', 3500) )
  wallet.append(
    CreditCard('John Bowman', 'California Finance',
               '5391 0375 9387 5309', 5000) )

  for val in range(1, 17):
    wallet[0].charge(val)
    wallet[1].charge(2*val)
```

# Example: CreditCard Class V

```
    wallet[2].charge(3*val)

for c in range(3):
    print('Customer =', wallet[c].get_customer())
    print('Bank =', wallet[c].get_bank())
    print('Account =', wallet[c].get_account())
    print('Limit =', wallet[c].get_limit())
    print('Balance =', wallet[c].get_balance())
    while wallet[c].get_balance() > 100:
        wallet[c].make_payment(100)
        print('New balance =', wallet[c].get_balance())
    print()
```

Example: CreditCard Class V

```
    wallet[2].charge(3*val)

for c in range(3):
    print('Customer =', wallet[c].get_customer())
    print('Bank =', wallet[c].get_bank())
    print('Account =', wallet[c].get_account())
    print('Limit =', wallet[c].get_limit())
    print('Balance =', wallet[c].get_balance())
    while wallet[c].get_balance() > 100:
        wallet[c].make_payment(100)
        print('New balance =', wallet[c].get_balance())
    print()
```

The instances defined by the CreditCard class provide a simple model for traditional credit cards. They have identifying information about the customer, bank, account number, credit limit, and current balance. The class restricts charges that would cause a card's balance to go over its spending limit, but it does not charge interest or late payments.

# Example: CreditCard Class

- The construct begins with the keyword, **class**, followed by the name of the class, a colon, and then an indented block of code that serves as the body of the class.
- The body includes definitions for all methods of the class. These methods are defined as functions, yet with a special parameter, named **self**, that serves to identify the particular instance upon which a method is invoked.

# The self indetifier

In Python, the **self** identifier plays a key role.

Syntactically, **self** identifies the instance upon which a method is invoked.

**Example 3.1**

Assume that a user of our class has a variable, `my_card`, that identifies an instance of the `CreditCard` class.

- When the user calls `my_card.get_balance()`, identifier **self**, within the definition of the `get_balance` method, refers to the card known as `my_card` by the caller.
- The expression, `self._balance` refers to an instance variable, named `_balance`, stored as part of that particular credit card's state.

# The Constructor

A user can create an instance of the `CreditCard` using a syntax as

```
cc = CreditCard('John Doe', '1st Bank',
                '5391 0375 9387 5309', 1000)
```

- This results in a call to the specially named `__init__` method that serves as the **constructor** of the class.
- Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables.
  - In the case of the CreditCard class, each object maintains five instance variables, which we name:
    `_customer`, `_bank`, `_account`, `_limit`, and `_balance`.
  - The initial values for the first four of those five are provided as explicit parameters that are sent by the user when instantiating the credit card, and assigned within the body of the constructor.

# Ecapsulation

Recall that a single leading underscore in the name of a data member, such as `_balance`, implies that it is intended as **nonpublic**. Users of a class should not directly acess such members.

# Ecapsulation

Recall that a single leading underscore in the name of a data member, such as `_balance`, implies that it is intended as **nonpublic**. Users of a class should not directly acess such members.

As a general rule, we will treat all data members as nonpublic. This allows us to better enforce a consistent state for all instances.

# Ecapsulation

Recall that a single leading underscore in the name of a data member, such as `_balance`, implies that it is intended as **nonpublic**. Users of a class should not directly acess such members.

As a general rule, we will treat all data members as nonpublic. This allows us to better enforce a consistent state for all instances.

- We can provide accessors, such as `get_balance`, to provide a user of our class read-only access to a trait.
- If we wish to allow the user to change the state, we can provide appropriate update methods.

# Additional Methods

- The `charge` function typically adds the given price to the credit card balance, to reflect a purchase of said price by the customer.
- The `make_payment` charge reflects the customer sending payment to the bank for the given amount, thereby reducing the balance on the card.

# Error Checking

The implementation of the CreditCard class is not particularly robust.

- Did not explicitly check the types of the parameters to `charge` and `make_payment`, nor any of the parameters to the constructor.
  - The code will crash when attempting to add that parameter to the current balance, such as `visa.charge('candy')`.
  - If this class were to be widely used in a library, we might use more rigorous techniques to raise a `TypeError` when facing such misuse.
- May be susceptible to logical errors.
  - If a user were allowed to charge a negative price, such as `visa.charge(-300)`, that would serve to lower the customer's balance.

# Testing the class I

```
from credit_card import CreditCard
if __name__ == '__main__':
    wallet = [ ]
    wallet.append(CreditCard( 'John Bowman' ,
                              'California Savings' ,
                              '5391 0375 9387 5309' ,
                              2500) )
    wallet.append(CreditCard( 'John Bowman' ,
                              'California Federal' ,
                              '3485 0399 3395 1954' ,
                              3500) )
    wallet.append(CreditCard( 'John Bowman' ,
                              'California Finance' ,
                              '5391 0375 9387 5309' ,
                              5000) )

    for val in range(1, 17):
        wallet[0].charge(val)
```

# Testing the class II

```
        wallet[1].charge(2*val)
        wallet[2].charge(3*val)

    for c in range(3):
        print( 'Customer = ', wallet[c].get_customer( ))
        print( 'Bank = ', wallet[c].get_bank( ))
        print( 'Account = ', wallet[c].get_account( ))
        print( 'Limit = ', wallet[c].get_limit( ))
        print( 'Balance = ', wallet[c].get_balance( ))
        while wallet[c].get_balance( ) > 100:
            wallet[c].make_payment(100)
            print( 'New balance = ', wallet[c].get_balance( ))
        print( )
```

# Testing the class I

- These tests are enclosed within a conditional, if `__name__ == '__main__ '`:, so that they can be embedded in the source code with the class definition.
- Provide **method coverage**, as each of the methods is called at least once
- Does not provide **statement coverage**, as there is never a case in which a charge is rejected due to the credit limit.

- These tests are enclosed within a conditional, if `__name__ == '__main__'`:, so that they can be embedded in the source code with the class definition.
- Provide **method coverage**, as each of the methods is called at least once
- Does not provide **statement coverage**, as there is never a case in which a charge is rejected due to the credit limit.

How to correctly understand `if __name__ == '__main__'` ?

`__name__` is the name of the current module. When the module is directly running, the module name is `__main__`. The meaning of this sentence is,

- When the module is directly executed, the following code blocks will be executed
- When the module is imported, code block is not going to be executed.

The details can be found in

https://stackoverflow.com/questions/4042905/what-is-main-py

# Class Definitions

## Operator Overloading and Python's Special Methods

# Operator Overloading and Python's Special Methods

Python's built-in classes provide natural semantics for many operators.

**Example 3.2**

`a + b` invokes addition for numerical types, yet concatenation for sequence types.

When defining a new class, we must consider whether a syntax like `a + b` should be defined when a or b is an instance of that class.

# Operator Overloading and Python's Special Methods

By default, the + operator is undefined for a new class. However, the author of a class may provide a definition using a technique known as operator overloading.

This is done by implementing a specially named method. In particular, the + operator is overloaded by implementing a method named `__add__` , which takes the right-hand operand as a parameter and which returns the result of the expression.

**Example 3.3**

The syntax `a + b` is converted to a method call on object of the form `a.__add__(b)`.

# Operator Overloading and Python's Special Methods

| | |
|---|---|
| `a + b` | `a.__add__(b)` or `b.__radd__(a)` |
| `a - b` | `a.__sub__(b)` or `b.__rsub__(a)` |
| `a * b` | `a.__mul__(b)` or `b.__rmul__(a)` |
| `a / b` | `a.__truediv__(b)` or `b.__rtruediv__(a)` |
| `a // b` | `a.__floordiv__(b)` or `b.__rfloordiv__(a)` |
| `a % b` | `a.__mod__(b)` or `b.__rmod__(a)` |
| `a ** b` | `a.__pow__(b)` or `b.__rpow__(a)` |
| `a << b` | `a.__lshift__(b)` or `b.__rlshift__(a)` |
| `a >> b` | `a.__rshift__(b)` or `b.__rrshift__(a)` |
| `a & b` | `a.__and__(b)` or `b.__rand__(a)` |
| `a ^ b` | `a.__xor__(b)` or `b.__rxor__(a)` |
| `a \| b` | `a.__or__(b)` or `b.__ror__(a)` |

# Operator Overloading and Python's Special Methods

| | |
|---|---|
| `a += b` | `a.__iadd__(b)` |
| `a -= b` | `a.__isub__(b)` |
| `a *= b` | `a.__imul__(b)` |
| ... | ... |
| `+a` | `a.__pos__(b)` |
| `-a` | `a.__neg__(b)` |
| `~a` | `a.__invert__(b)` |
| `abs(a)` | `a.__abs__(b)` |
| `a < b` | `a.__lt__(b)` |
| `a <= b` | `a.__le__(b)` |
| `a > b` | `a.__gt__(b)` |
| `a >= b` | `a.__ge__(b)` |
| `a == b` | `a.__eq__(b)` |
| `a != b` | `a.__ne__(b)` |

# Operator Overloading and Python's Special Methods

| | |
|---|---|
| `v in a` | `a.__contains__(v)` |
| `a[k]` | `a.__getitem__(k)` |
| `a[k] = v` | `a.__setitem__(k, v)` |
| `del a[k]` | `a.__delitem__(k)` |
| `a(arg1, arg2, ...)` | `a.__call__(arg1, arg2, ...)` |
| `len(a)` | `a.__len__()` |
| `hash(a)` | `a.__hash__()` |
| `iter(a)` | `a.__iter__()` |
| `next(a)` | `a.__next__()` |

# Operator Overloading and Python's Special Methods

| | |
|---|---|
| `bool(a)` | `a._bool__()` |
| `float(a)` | `a.__float__()` |
| `int(a)` | `a.__int__()` |
| `repr(a)` | `a.__repr__()` |
| `reversed(a)` | `a.__reversed__()` |
| `str(a)` | `a.__str__()` |

# Non-Operator Overloads

Python relies on specially named methods to control the behavior of various other functionality, when applied to user-defined classes.

# Non-Operator Overloads

**Example 3.4**

The sytax `str(foo)` is formally a call to the constructor for the string class.

```
>>> str(123)
'123'
```

For user-defined class:

```
>>> class Test():
>>>     def __str__(self):
>>>         return "This is a test"
>>> t = Test()
>>> str(t)
This is a test
```

# Non-Operator Overloads

**Example 3.5**

The standard way to determine the size of a container type is by calling the top-level `len` function. Note that `len(foo)` is not the traditional method-calling syntax with the dot operator. However, in the case of a user-defined class, the top-level `len` function relies on a call to a specially named `__len__` method of that class. That is, the call `len(foo)` is evaluated through a method call, `foo.__len__( )`.

# Implied Methods

As a general rule, if a particular special method is not implemented in a user-defined class, the standard syntax that relies upon that method will raise an exception.

**Example 3.6**

Evaluating the expression, `a + b`, for instances of a user-defined class without `__add__` or `__radd__` will raise an error.

# Implied Methods

However, there are some operators that have default definitions provided by Python, in the absence of special methods, and there are some operators whose definitions are derived from others.

**Example 3.7**

- The `__bool__` method, which supports the syntax `if foo:`, has default semantics so that every object other than **None** is evaluated as **True**.
- For container types, the `__len__` method is typically defined to return the size of the container. If such a method exists, then the evaluation of `bool(foo)` is interpreted by default to be **True** for instances with **nonzero** length, and **False** for instances with **zero** length.

# Implied Methods

**Example 3.8**

- Python provides **iterators** for collections via the special method, `__iter__`. With that said, if a container class provides implementations for both `__len__` and `__getitem__`, a default iteration is provided automatically. Furthermore, once an iterator is defined, default functionality of `__contains__` is provided.

# Implied Methods

**Example 3.9**

- The distinction between expression `a is b` and expression `a == b` is that, the former evaluating whether identifiers a and b are aliases for the same object, and the latter testing a notion of whether the two idenifiers reference *equivalent* values. The notion of "equivalance" depends upon the context of the class, and semantics is defined with the `__eq__` method. However, if no implementation is given for `__eq__`, the syntax `a == b` is legal with semantics of `a is b`.

# Class Definitions

## Example: Multidimensional Vector Class

# Example: Multidimensional Vector Class

To demonstrate the use of operator overloading via special methods, we provide an implementation of a **Vector** class, representing the coordinates of a vector in a multidimensional space.

**Example 3.10**

In a three-dimensional space, we might wish to represent a vector with coordinates <5, 2, 3>. Although it might be tempting to directly use a Python list to represent those coordinates, a list does not provide an appropriate abstraction for a geometric vector. In particular, if using lists, the expression `[5, 2, 3] + [1, 4, 2]` results in the list `[5, 2, 3, 1, 4, 2]`. When working with vectors, if `u = <5, 2, 3>` and `<v = 1, 4, 2>`, one would expect the expression, `u + v`, to return a three-dimensional vector with coordinates <6, 2, 5>.

# Example: Multidimensional Vector Class I

```
import collections

class Vector:
  """Represent a vector in a multidimensional space."""

  def __init__(self, d):
    if isinstance(d, int):
      self._coords = [0] * d
    else:
      try:          # we test if param is iterable
        self._coords = [val for val in d]
      except TypeError:
        raise TypeError('invalid parameter type')

  def __len__(self):
    """Return the dimension of the vector."""
    return len(self._coords)
```

# Example: Multidimensional Vector Class II

```
def __getitem__(self, j):
  """Return jth coordinate of vector."""
  return self._coords[j]

def __setitem__(self, j, val):
  """Set jth coordinate of vector to given value."""
  self._coords[j] = val

def __add__(self, other):
  """Return sum of two vectors."""
  if len(self) != len(other):    # relies on __len__ method
    raise ValueError('dimensions must agree')
  result = Vector(len(self))     # start with vector of zeros
  for j in range(len(self)):
    result[j] = self[j] + other[j]
  return result

def __eq__(self, other):
  """Return True if vector has same coordinates as other."""
```

# Example: Multidimensional Vector Class III

```
    return self._coords == other._coords

  def __ne__(self, other):
    """Return True if vector differs from other."""
    return not self == other      # rely on existing __eq__
    definition

  def __str__(self):
    """Produce string representation of vector."""
    return '<' + str(self._coords)[1:-1] + '>'  # adapt list
    representation

  def __neg__(self):
    """Return copy of vector with all coordinates negated."""
    result = Vector(len(self))    # start with vector of zeros
    for j in range(len(self)):
      result[j] = -self[j]
    return result
```

# Example: Multidimensional Vector Class IV

```
  def __lt__(self, other):
    """Compare vectors based on lexicographical order."""
    if len(self) != len(other):
      raise ValueError('dimensions must agree')
    return self._coords < other._coords

  def __le__(self, other):
    """Compare vectors based on lexicographical order."""
    if len(self) != len(other):
      raise ValueError('dimensions must agree')
    return self._coords <= other._coords

if __name__ == '__main__':
  # the following demonstrates usage of a few methods
  v = Vector(5)     # construct five-dimensional <0, 0, 0, 0, 0>
  v[1] = 23         # <0, 23, 0, 0, 0> (based on use of __setitem__)
  v[-1] = 45        # <0, 23, 0, 0, 45> (also via __setitem__)
  print(v[4])       # print 45 (via __getitem__)
  u = v + v         # <0, 46, 0, 0, 90> (via __add__)
```

# Example: Multidimensional Vector Class V

```
print(u)          # print <0, 46, 0, 0, 90>
total = 0
for entry in v:   # implicit iteration via __len__ and __getitem__
  total += entry
```

# Class Definitions

## Iterators

# Iterators

Iteration is an important concept in the design of data structures. An **iterator** for a collection provides one key behavior:

- It supports a special method named next that returns the next element of the collection, if any, or raises a **StopIteration** exception to indicate that there are no further elements.

Iteration is an important concept in the design of data structures. An **iterator** for a collection provides one key behavior:

- It supports a special method named next that returns the next element of the collection, if any, or raises a **StopIteration** exception to indicate that there are no further elements.

Fortunately, it is rare to have to directly implement an iterator class. Our preferred approach is the use of the **generator** syntax which automatically produces an iterator of yielded values.

# Iterators

Python also helps by providing an automatic iterator implementation for any class that defines both `__len__` and `__getitem__`.

# Iterators I

```
class SequenceIterator:
  """An iterator for any of Python's sequence types."""

  def __init__(self, sequence):
    """Create an iterator for the given sequence."""
    self._seq = sequence          # keep a reference to the
    underlying data
    self._k = -1                  # will increment to 0 on first
    call to next

  def __next__(self):
    """Return the next element, or else raise StopIteration error.
    """
    self._k += 1                  # advance to next index
    if self._k < len(self._seq):
      return(self._seq[self._k])  # return the data element
    else:
      raise StopIteration()       # there are no more elements
```

# Iterators II

```
def __iter__(self):
  """By convention, an iterator must return itself as an iterator.
  """
  return self
```

# Class Definitions

## Example: Range Class

# Example: Range Class

Here we develop our own implementation of a class that mimics Python's built-in range class.

- Prior to Python 3 being released, **range** was implemented as a function, and it returned a list instance with elements in the specified range.

```
>>> a = range(2, 10, 2)  # Python2
>>> a
[2, 4, 6, 8]
```

Here we develop our own implementation of a class that mimics Python's built-in range class.

- Prior to Python 3 being released, **range** was implemented as a function, and it returned a list instance with elements in the specified range.

```
>>> a = range(2, 10, 2)  # Python2
>>> a
[2, 4, 6, 8]
```

A typical use of the function was to support a for-loop syntax, such as `for k in range(10000000)`. Unfortunately, this caused the instantiation and initialization of a list with the range of numbers. That was an unnecessarily expensive step, in terms of both time and memory usage.

# Example: Range Class

The mechanism used to support ranges in Python 3 is entirely different. It uses a strategy known as **lazy evaluation**. Rather than creating a new list instance, **range** is a class that can effectively represent the desired range of elements without ever storing them explicitly in memory.

```
>>> a = range(2, 10, 2)  # Python3
>>> a
range(2, 10, 2)
>>> a.__len__()          # equiv to len(a)
4
>>> a.__getitem__(2)     # equiv to a[2]
6
```

Because the class supports both `__len__` and `__getitem__`, it inherits automatic support for iteration, which is why it is possible to execute a for loop over a range.

# Example: Range Class I

```
class Range:
  """A class that mimic's the built-in range class."""

  def __init__(self, start, stop=None, step=1):
    """Initialize a Range instance.

    Semantics is similar to built-in range class.
    """
    if step == 0:
      raise ValueError('step cannot be 0')

    if stop is None:                    # special case of range(n)
      start, stop = 0, start            # should be treated as if
      range(0,n)

    # calculate the effective length once
    self._length = max(0, (stop - start + step - 1) // step)

    # need knowledge of start and step (but not stop) to support
     __getitem__
    self._start = start
```

# Example: Range Class II

```
    self._step = step

  def __len__(self):
    """Return number of entries in the range."""
    return self._length

  def __getitem__(self, k):
    """Return entry at index k (using standard interpretation if
     negative)."""
    if k < 0:
      k += len(self)                    # attempt to convert
      negative index

    if not 0 <= k < self._length:
      raise IndexError('index out of range')

    return self._start + k * self._step
```