Introduction to Object-oriented Analysis and Design

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
Course information (Fall 2022)

This full-semester course introduces terminology of the object-oriented (OO) paradigm, such as classes, superclasses (or parent classes), subclasses (or child classes), class variables, objects, encapsulation, abstraction, simple inheritance, multiple inheritance, polymorphism, duck typing, exceptions, and abstract base classes.

You will learn the most useful tools in the object-oriented principles by practicing OO analysis, OO design and OO programming (using Python 3) for several existing software
applications.

The object-oriented programming paradigm may at first appear quite strange for people familiar with the procedural paradigm, but can turn out to be quite natural for many problems. This course aims to train students to view these problems and come up with solutions in an object-oriented fashion. Previous programming experience, though desirable, is not required. This is a rather practical course, in which concepts introduced in lectures will be soon applied in the labs as well as in the course project. Finally, we will analyze whether several claimed benefits of the OO paradigm, such as reliability, productivity, reuse, ease of modification, indeed exist.
**Paradigm**: a pattern or model.


A significant portion of the lecture notes is built on the above book.


**Grading policy**: 
<table>
<thead>
<tr>
<th>Component</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labs</td>
<td>30</td>
</tr>
<tr>
<td>Course Project</td>
<td>20</td>
</tr>
<tr>
<td>Final exam</td>
<td>50</td>
</tr>
</tbody>
</table>

**Software freely available for this course:**

- Python 3.8.5 interpreter:
  
  
  [scroll to bottom to find a installer for your operating system.]

• Flask: https://pypi.org/project/Flask/
Important things

• Class Diagram

• Sequence Diagram

• Use case Diagram

• Encapsulation

• Information Hiding
• Abstraction

• Interface

• Inheritance (simple and multiple)

• Mixin

• Attribute vs. Property

• Object Interaction

• The Decorator Pattern
• The Iterator/Generator Pattern

• The Factory Pattern

• The Template Pattern

• Abstract Base Classes

• Duck typing, Polymorphism

• Cohesion

• Coupling
• Composition and aggregation

• Responsibility-driven Design - Knows and Does

• The Single Responsibility Principle (SRP), Separation of Concerns

• The Don’t Repeat Yourself Principle (DRY), Code Reuse, Code Duplication

• Class Responsibility Collaborator (CRC) model

• Requirements Workshops
• Iterative and Incremental Development (IID)

• Timeboxed Sprints
Model

A model is an abstraction for the real-world stuff.

A model is a *simplification* of the real-world object.

A driverless car model.

A high-speed train model.

A space shuttle model.
Object

Object - a physical thing that we can sense, feel and manipulate (e.g., wallets, toys, babies, apples, oranges, etc).

Put simply, an object is a tangible thing.

It does not have to be physical, though.

• A point can be an object.

• A circle can be an object.
• A **file** can be an object.

• A **name** can be an object.

• A **title** can be an object.

• A **phone call** can be an object.

• A **job position** can be an object.

• An **operating system** can be an object.

An object is a *model* of something. A model is an abstraction.
An object has **data** (i.e., attributes) and **behaviors** (i.e., methods).

Exercise. Describe data and behaviors for each of the above objects.
Oriented

Orient’s dictionary definition. 1. align or position (something) relative to the points of a compass or other specified positions. Find one’s position in relation to new and strange surroundings. 2. adjust or tailor (something) to specified circumstances or needs.

Oriented - directed towards.

Object-oriented. Specify the style of software development: we are going to model objects and their interactions (if any). Functionally directed toward modeling objects.
The object-oriented umbrella

Object-oriented ≈ Modeling objects and their interactions

Object-oriented umbrella - OO analysis, OO design, and OO programming.

As we have learned from our Software Engineering course, design happens before programming, and analysis before design.

In reality, this order is not that strict.

In fact, iteration is common.
Object-oriented analysis

What?

DO THE RIGHT THING.

Analyze the requirements first.

Work products: use cases, user stories, interacting objects.

Understand the problem from the perspective of objects and classes.
Happen after or during the Software Requirements stage.

Understand what needs to be done (and what needs not to be done), by looking at a task and identifying objects and their interactions.

Object-oriented exploration - interview customers, study their processes, and eliminate possibilities (i.e., define scope).

For example, three requirements for an online food store:

- *review our history*
- *apply for jobs*
• browse, compare, and order products

Tasks:

• Write down requirements.

• Identify objects, e.g., Plane, Flight, Pilot.

• How do these objects interact?

A useful tutorial on OOAD

Textbook Example: inventory application for a fruit farm.
Behaviors?

**Research question**: Could AI automatically extract objects (data and behaviors) from user stories?
Object-oriented design

DO THE THING RIGHT.

How?

Figure out how things should be done.

Present a conceptual solution.

Work products: implementation specification.

More specifically, assign responsibilities to classes and collaborate objects.
Responsibility assignment is an important skill we need to master.

Responsibility-driven design. CRC cards. Class diagrams.

Convert analysis of requirements into implementation specification (classes, attributes and methods).

Interface examples:

- Attribute
- Method
- Abstract Base Class that provides only method head but
no method implementation. This kind of classes is called interface.

OO mentality:

• **Name the objects.** 3-5 main objects. The most prominent ones.
  
  – Define object attributes – data, e.g., tail_number in Plane. Primitives or classes?
  – Define object actions – behaviors, e.g., get_flight_history.
  – Levels of abstraction?

• Describe how the objects interact/collaborate. (Specify
which objects interact with other objects.

- Identify constraints.
Class Responsibility Collaborator (CRC) Models

**CRC** cards. Three components. What are they?

You should be able to draw CRC cards and move them around on a table.

A design technique.

An *effective* tool for

1. Conceptual modeling
2. Detailed design
• Class - class name (a singular noun)

• Responsibility - what the class knows or does.

• Collaborator - collaborate/interact with other classes. Need help from other classes. For example, Seminar could be Student’s collaborator for checking space availability and for enrollment.

Required readings:

• Class Responsibility Collaborator (CRC) Models: An Agile Introduction
• Scott W. Ambler. \textbf{CRC} modeling: bridging the communication gap between developers and users, an AmbySoft Inc. white paper.

I find it useful to consider how I am going to use the classes in the meantime.

Example 1 (recommending university majors):

\begin{verbatim}
a = Applicant('bookmark.xml', gaokao_score)
mr = MajorRecommender()
result = mr.recommend_for(a)
\end{verbatim}

Example 2 (the observer pattern):

\begin{verbatim}
i = Inventory()
\end{verbatim}
uk Observer = UnitedKingdomObserver(i)
i.attach(uk Observer)

us Observer = UnitedStatesObserver(i)
i.attach(us Observer)

i.product = 'E45'
i.quantity = 2
Object-oriented programming

Smalltalk, C++, Java, JavaScript, Python.

Convert design into a working program using an OO programming language.

Smalltalk (too ancient), C++ (too complex), Java (too verbose), JavaScript (everywhere), Python (will be faster).

Some classic designs are consolidated as design patterns.
Why has object-oriented technique grown slowly?

• David Parnas: The answer is simple. It is because [O-O] has been tied to a variety of complex languages. Instead of teaching people that O-O is a type of design, and giving them design principles, people have taught that O-O is the use of a particular tool. Unless we teach people how to design, the languages matter very little. The result is that people do bad designs with these languages and get very little value from them. If the value is small, it won’t catch
Les Hatton: To summarize, these two independent data sources suggest that corrective-maintenance costs are significantly larger in OO systems written in C++ when compared to conventional systems implemented in C or Pascal.
Test-driven development

It forces writing tests before writing code.

- It helps understand requirements better.

- It ensures that code continues working after we make changes.

Note that changes to one part of the code can inadvertently break other parts.

Useful frameworks:

- pytest
- selenium
- cypress
Object-oriented paradigm – tools and not rules

When not to use inheritance?

“Objects are just tools, and not rules.”

Example: the library catalog diagrams in pages 24-25.

Objects usually have both data and behavior.

If we work only with data, why not use built-in data structures (e.g., list, dictionary, set, etc)?

If we work only with behavior, why not use functions?
Group data and behavior when there is an *obvious* need.

Question: Is OO code always shorter than non-OO code? See pages 126-128 for one example.
The real world is murky

murky: dark and gloomy, especially due to thick mist. Not fully explained or understood.

No matter how hard we try to separate these stages (OOA, OOD and OOP), we will always find things that need further analysis while we are designing. When we are programming, we find features that need clarification in the design.

Remember the Agile Process we’ve talked in our Software Engineering course? A series of short development cycles.
Development is iterative. How iterative it is depends on the team’s experience, and the completeness of requirements.
Iterative and incremental development

IID - iterative and incremental development, dates back to late 1950s.

45% of the features in Waterfall requirements are never used.

A typical software project experienced a 25% change in requirements [Boehm and Papaccio].

Work on a series of mini-projects to get a series of partial systems, each being a growing subset of the final system.
1. Get some requirements.

2. Plan.

3. **Build** a partial system.

4. Ask for **feedback** (requirements clarification, marketplace change).

5. Analyze and **incorporate** feedback.

Repeat 2-5 for about 10-15 iterations.

See picture iterative-development.png

**UP** - Unified Process (Extreme Programming - Test-driven...
development and pair programming, Risk-driven development, Client-driven development, Scrum - war room, daily stand-up meeting, three special questions to be answered by each team member, Refactoring and CI)

Monday: one-hour meeting, review last iteration’s diagrams, whiteboards, pseudocode and design notes. No rush to code.

No overly-detailed design.

Short **timeboxed** iterations are preferred. Why?

**Agile methods**: Each iteration refines requirements, plans and design. *Whatever works.*
3 weeks better than 6 weeks. Why?

Too much work in the current iteration? De-scope iteration goals.

Object-oriented IID

- List as many relevant names as possible (i.e., brainstorming).
- Select names.
- Think about attributes and methods.
- Think about interactions between objects (names).
Waterfall thinking

Up-front analysis and modeling.

BUT software development is a high-change area.

Figure 2.3 Larman’s book PDF page 63.

“Let’s write all the use cases before starting to program.”

“Let’s do many detailed OO models in UML before starting to program.”

Consequences: 45% of the features in waterfall requirements
are never used, and early waterfall schedules and estimates vary up to 400% from the final actuals.
Evolutionary analysis and design

For example, a project needs 20 iterations, each timeboxed in 3 weeks.

The first 5 iterations include requirements workshops and critical prototypes, stabilizing 90% of the requirements, but only completing 10% of the whole software. That is, about 20% of the whole project time is devoted to requirements.

Figure 2.4 Larman’s book PDF page 69.

Starting coding near Day One of the project is also bad. We
need a middle way.

Iterative and evolutionary requirements analysis combined with early timeboxed iterative development and frequent stakeholder participation, evaluation, and feedback on partial results.
Agile modeling

The purpose of modeling (sketching UML, ...) is primarily to understand, not to document.

Treat it lightly.

Doing UML is not to create many detailed UML diagrams for programmers, but to quickly explore alternatives and paths to a good OO design.

**Misconception:** People translate UML diagrams mechanically to code.
Prefer sketching UML on whiteboards, and taking a picture for it.

Don’t do it alone.

“Good enough”, simple notations.
UML - Unified Modeling Language

Considered evil by many ...

Occasionally disparaged ... little worth

A waste of time ... how to keep UML diagrams up-to-date in IID?

Boxes and lines to intuitively illustrate classes and the association between them.

A useful communication tool during OO analysis and design.
A useful reference for myself in the future. *Why did I do that?*

**Caution:** best used only when needed. Don’t be lost among UML details. Hide uninteresting details.

**Reality:** the initial diagrams become outdated very soon. Because these diagrams are subject to change in subsequent iterations, some people think drawing UML class diagrams is a waste of time (if you spend too much time on it). Don’t make it too formal in the beginning.

Most useful diagrams: class diagrams, use case diagrams, activity diagrams, and sequence diagrams.
• **Class diagrams.** A box represents a class. A line between two boxes represents a relationship.

• **Sequence diagrams.** Model the interactions (step-by-step) among objects in a Use Case. Vertical lifeline (the dashed line hanging from each object), activation bars, horizontal arrows (messages, methods, return values).

• **Use case diagrams.**

• **Activity diagrams.**
Objects and classes

Classes describe objects. A class definition is like a **blueprint** for creating objects.

An object usually has **attributes** and **behaviors**.

Kind of objects is **class**.

In the class, we *abstract* common attributes and behaviors shared by these objects.

This orange of 50 grams belongs to Orange class, and that orange of 100 grams belongs to Orange class. The weight is
one of the *infinitely many* characteristics that are shared by all oranges. These infinitely many characteristics include farm, pick date, best before date, etc. But any application needs only a finite set of characteristics, we must *ignore irrelevant characteristics* while modeling. This selecting and ignoring process is called ABSTRACTION.

An object is called an *instance* of a class. This object instance has its own set of data and behaviors.

We can make arbitrary number of objects from a class. For example, I can create 10,000 Student objects from class Student. I can create 100,000 str objects using the Python’s
built-in \texttt{str} class. That is why class definition is useful.

An Orange class may have three attributes: weight, orchard and date picked. You can use it to describe/represent real oranges sold in the market, each having different weight, orchards and pick dates.

class Orange:

    ''' A blueprint for making many oranges. '''

    def \_\_init\_\_(self, weight, orchard, date\_picked):
        self.weight = weight
        self.orchard = orchard
        self.date = date\_picked

    def \_\_str\_\_(self):
        return 'Orange info: %.2f lbs picked on %s from %s.' %
            (self.weight, self.date, self.orchard)
```python
cheap_orange = Orange(0.08, 'Yiwu', '2018-12-01')
print(cheap_orange)
expensive_orange = Orange(0.12, 'Jinhua', '2018-11-29')
print(expensive_orange)

#Orange info: 0.08 lbs picked on 2018-12-01 from Yiwu.
#Orange info: 0.12 lbs picked on 2018-11-29 from Jinhua.
```

Use `cheap_orange.__dict__` to show attributes and their values. Try also `Orange.__dict__`.
Useful names and resources

Martin Fowler - UML Distilled, Refactoring.

Scott Ambler - Agile Modeling.

Procedural versus object-oriented

A wallet for money deposit and withdrawal.

Procedural:

```python
money = 0
def deposit(amount):
    global money
    money += amount
def withdraw(amount):
    global money
    money -= amount
```
def check():
    global money
    return 'The wallet has %.0f RMB.' % money

Problem: the above code only works for a single wallet. We can add a create function which returns a dictionary (e.g., {'money':0}) for each wallet created.

Procedural:

def create(amount):
    return {'money':amount}  # a dictionary

def deposit(wallet, amount):
    wallet['money'] += amount

def withdraw(wallet, amount):
```python
wallet['money'] -= amount

def check(wallet):
    return 'The wallet has %0.0f RMB.' % wallet['money']
```

Object-oriented:

**Think in Objects**

```python
class Wallet:
    def __init__(self, money):
        self.money = money

    def deposit(self, amount):
        self.money += amount

    def withdraw(self, amount):
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```
self.money -= amount

def check(self):
    return 'The wallet has %.0f RMB.' % self.money

w = Wallet(0)
w.check()
'The wallet has 0 RMB.'

w.deposit(1000)
w.check()
'The wallet has 1000 RMB.'

w.withdraw(500)
w.check()
'The wallet has 500 RMB.'
```
Chapter 1 Object-oriented design

Abstraction

Classes

Encapsulation: holding attributes and methods together in a capsule (i.e., a class).

Inheritance

UML
Example: Dice Game

Roll two dices and check whether their face values sum to 7.

*Use case:* Player rolls the dice. System returns results. If the dice face values totals 7, play wins; otherwise, player loses.

*Domain model, or conceptual object model*

*Interaction diagrams* We could use a sequence diagram: flow of message, invocation of methods.

*Class diagrams.*
Visual Modeling is a good thing. Use UML as a sketch. Don’t let too many uninteresting details get in the way.
Everything in Python is a class

We have seen a few examples of customary classes. Note that the internal python data types such as numbers, strings, modules, and even functions, are classes too.

```python
>>> import random
>>> type(random)
<class 'module'>

>>> x = 123
>>> type(x)
<class 'int'>

>>> x = '123'
```
>>> `type(x)`
    `<class 'str'>`

>>> `x = [1,2,3]`

>>> `type(x)`
    `<class 'list'>`

>>> `def f():`
    `pass`

>>> `type(f)`
    `<class 'function'>`

>>> `class Minimalism:`
    `pass`

>>> `x = Minimalism()`

>>> `type(x)`
    `<class '__main__.Minimalism'>`
I fail to see why not everything in the world cannot be described as a class.
Objects are *instances* of classes.

Each object of a class has its own set of data, and methods dealing with these data. With OOP, we in principle don’t access these class attributes directly, but *only* via class methods.

- Data describe objects.
  
  Data represents the individual characteristics of a certain object.
Attributes - values

All objects instantiated from a class have the same attributes but may have different values.

Attributes are sometimes called members or properties (usually read-only).

• Behaviors are actions.

Behaviors are actions (methods) that can occur on an object.

We can think of methods as functions which have access to all the data associated with this object.
Methods accept parameters and return values.

OOA and OOD are all about identifying objects and specifying their interactions.
Properties - attributes associated with behaviors

Common uses of properties: add some behaviors to the data.

Give a name property to Color.

class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self._name = name

    def _set_name(self, name):
        if not name:
            raise Exception("Invalid Name")
self._name = name

def _get_name(self):
    ''' This is get name method. '''
    return self._name

name = property(_get_name, _set_name)

if __name__ == '__main__':
c = Color("#0000ff", "bright red")
print(c.name)
c.name = 'BR'
print(c.name)

Another way to make a property.

class Color:

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```python
def __init__(self, rgb_value, name):
    self.rgb_value = rgb_value
    self._name = name

@property
def name(self):
    ''' This is get name method. '''
    return self._name

@name.setter
def name(self, name):
    if not name:
        raise Exception("Invalid Name")
    self._name = name
```
if __name__ == '__main__':
    c = Color("#0000ff", "bright red")
    print(c.name)
    c.name = 'br'
    print(c.name)

Use a property to cache content.

import time
from urllib.request import urlopen

class WebPage:
    def __init__(self, url):
        self.url = url
        self._content = None

@property
    def content(self):

if not self._content:
    print('get new page ...')
    self._content = urlopen(self.url).read()
return self._content

if __name__ == '__main__':
    webpage = WebPage('http://ccphillips.net/')
    print(webpage.content)
    print(webpage.content) # which print statement will take longer?

Exercise: define a property likes for class Article.

    a = Article(aid)
    print(a.likes)
    a.likes = a.likes + 1
Name mangling

**mangle** - damage by cutting, tearing, or crushing.

No real private attribute in a Python class. We do not emphasize that kind of enforced privacy.

Private by default vs. Public by default.

- Prefix a variable name with one underscore to denote privacy. For example, _money.

- Prefix a variable name with two underscores to denote strong
privacy. For example, __money.

Do we have a way to access __money? Yes. Trick: use __ClassName__money

Why do we need two leading underscores? To indicate that this is something that our class users really should not access directly without good reasons.

Example:

class SecretString:
    def __init__(self, secret, passphrase):
        self.__secret = secret
        self.__passphrase = passphrase
        self.__passphrase = passphrase
self.passphrase = passphrase

def decrypt(self, passphrase):
    '''Return secret iff the passphrase is correct.'''
    if passphrase == self.__passphrase:
        return self.__secret
    else:
        return 'Wrong passphrase'

if __name__ == '__main__':
    s = SecretString('iPhone 14 Pro', 'dynamicisland')
    print(s.decrypt('dynamicisland'))  # iPhone 14 Pro
    print(s.decrypt('A17'))  # Wrong passphrase
    print(s.passphrase)  # dynamicisland
    print(s.__passphrase)  # dynamicisland
    print(s.SecretString.__passphrase)  # dynamicisland (name mangling)
    print(s.SecretString.__secret)  # iPhone 14 Pro (name mangling)
print(s.__secret__) # AttributeError

References:

Python name mangling

Name Mangling in Python
How do we make object interact? Pass objects as arguments to object methods.

```python
class Orange:
    def __init__(self, weight, orchard, date_picked):
        self.weight = weight
        self.orchard = orchard
        self.date = date_picked

    def pick(self, basket):
        basket.accept(self)

    def __str__(self):
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
def squeeze(self):
    juice = self.weight * 0.7
    self.weight = self.weight - juice
    return juice

class Basket:

    ''' A basket dedicated to store oranges. '''

def __init__(self, location):
    self.location = location
    self.oranges = []

def accept(self, item):
```python
def sell(self, customer):
    while self.oranges:
        o = self.oranges.pop()
        customer.purchase(o)

def discard(self):
    self.oranges = []

class Customer:
    ''' A customer who keeps track of his purchases. '''

def __init__(self, name):
    self.name = name
    self.purchase_history = ''
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
def purchase(self, item):
    self.purchase_history += str(item) + '\n'

def get_purchase_history(self):
    return '%s has purchased:\n' % (self.name) \
    + self.purchase_history

# Make objects and make them interact
basket = Basket('Margate')
orange1 = Orange(0.5, 'Sutton', '2018-09-16')
orange2 = Orange(0.4, 'Holloway', '2018-09-17')
orange3 = Orange(0.3, 'Oldham', '2018-09-18')
orange3.squeeze()
customer1 = Customer('Pooter')
customer2 = Customer('Lupin')
orange1.pick(basket)
orange2.pick(basket)
orange3.pick(basket)

basket.sell(customer1)
basket.sell(customer2)

print(customer1.get_purchase_history())
print(customer2.get_purchase_history())

#Pooter has purchased:
#0.30 lbs orange from Oldham picked on 2018-09-18
#0.40 lbs orange from Holloway picked on 2018-09-17
#0.50 lbs orange from Sutton picked on 2018-09-16

#Lupin has purchased:
Composition and aggregation

Composition or aggregation: a class is made of several objects that are made from other classes.

**Composition**: collecting several objects to make a new one.

**Aggregation** is closely related to composition. Main difference: the aggregate objects may exist independently (they won’t be destroyed after the container object is gone).

Composite and aggregate objects have different lifespan.

The main difference is whether the child could exist...
independently from the parent.

A class has a list of students. Students could exist independently from Class. Aggregation.

A house has a number of rooms. Room could not exist independently from House. Composition.

The difference is not very important in practice.

- A car is composed of an engine, transmission, starter, headlights and windshield. The engine comprises many parts. We can decompose the parts further if needed. has a relationship.
• How about abstract components? For example, names, titles, accounts, appointments and payments.

Model chess game.

Two players - a player may be a human or a computer.

One chess set - a board with 64 positions, 32 pieces including pawns, rooks, bishops, knights, king and queen). Each piece has a shape and a unique move rule.

The pieces have an aggregate relationship with the chess set. If the board is destroyed, we can still use the pieces.

The positions have a composite relationship with the chess
set. If the board is destroyed, we cannot re-use positions anymore (because positions are part of the board).

**Man - Leg - Shoes**

```python
class Shoes:
    def __init__(self, size):
        self.size = size  # US size

class Leg:
    def __init__(self, length):
        self.length = length  # in cm

class Man:
    def __init__(self, shoes):
        self.leg = Leg(120)  # leg is instantiated inside class definition
        self.shoes = shoes  # shoes is instantiated outside class definition
```
shoes = Shoes(9)
man = Man(shoes)
man.leg
man.shoes
del man
shoes
man.leg # the attribute leg is destroyed so NameError
Simple inheritance

This is the **most famous** object-oriented principle.

For creating *is a* relationship.

Abstract common logic into superclasses and manage specific details in the subclass.

Queen *is a* Piece.

So are Pawns, Bishops, Rooks, Knights and King.

This is inheritance. Inheritance is useful for sharing code
(and for avoiding duplicate code).

Everything in python is inherited (derived) from the most base class, \texttt{object}.

Check the output of \texttt{help(object)} and \texttt{dir(object)}.
Overriding methods

Re-defining a method of the superclass (the method name unchanged) in the subclass. We can override special methods (such as \_\_init\_\_, \_\_str\_\_) too.

class Piece:

    def \_\_init\_\_(self, color):
        self.color = color

    def move(self):
        raise NotImplementedError('Subclass must implement the abstract method')
class PuppetKing(Piece):
    ''' There is no move method in this class '''
    def __init__(self, color, shape):
        super().__init__(color)
        self.shape = shape

class King(Piece):

    def __init__(self, color, shape):
        super().__init__(color)
        self.shape = shape

    def move(self):
        print('King move')

class Player:
def __init__(self, chess_set):
    self.chess_set = chess_set

def calculate_move(self):
    print('Randomly pick a piece and make a legal move.')

class DeepBlue(Player):
    def __init__(self, chess_set):
        Player.__init__(self, chess_set)

def calculate_move(self):
    """Artificial intelligence decides the next move after analyzing self.chess_set""
    print('Judiciously pick a piece and make a smart move.')
super()

The `super()` function returns an object instantiated from the parent class, allowing us to call the parent methods directly.

The `super()` function can be called anywhere in any method in the subclass.

```python
class Contact:
    all_contacts = []  # class variable

    def __init__(self, name, email):
        self.name = name
        self.email = email
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
self.all_contacts.append(self)

def __str__(self):
    return '%s <%s>' % (self.name, self.email)

class Friend(Contact):

    def __init__(self, name, email, phone):
        print(id(super()))
        super().__init__(name, email)
        self.phone = phone

    def __str__(self):
        # print(id(super()))
        return super().__str__() + ' phone:%s' % (self.phone)
class Supplier(Contact):
    def order(self, order):
        print('Send %s to %s' % (order.upper(), self.name))

f1 = Friend('Bob', 'bob@wonderland.com', '(010) 8793180')
f2 = Friend('Nick', 'nick@starbucks.com', '(0579) 2865 2288')
print(f1)
print(f2)
print(id(f1))
print(id(f2))
print(id(f1.all_contacts))  # this and the following two line have the
print(id(f2.all_contacts))
print(id(Friend.all_contacts))
s = Supplier('Pizza Hut', 'order@pizzahut.com')
s.order('8 Chicken wings')
for p in s.all_contacts:
    print(p)

It makes sense to order something from a supplier (Supplier) but not from my friends (Friend). So Supplier has the method order() while Friend does not have this method, although both subclasses are derived from the same parent class, Contact.
Class variables

In class Contact, all_contacts is a class variable.

What is special about the class variable? It is shared by all instances of this class.

In the above example, whenever we create an object (from Contact, Friend, or Supplier), this object is appended to all_contacts.

We access the class variable via: Contact.all_contacts, Friend.all_contacts, or f.all_contacts.
Extending built-ins

• Add a search method to the built-in type list.

```python
class ContactList(list):

    def search(self, name):
        ''' Return all contacts that match name. '''
        matching_contacts = []

        for contact in self:
            if name in contact.name: # self is a list of objects
                matching_contacts.append(contact)

        return matching_contacts
```
```python
class Contact:

    all_contacts = ContactList()  # class variable

    def __init__(self, name):
        self.name = name
        self.all_contacts.append(self)

    c1 = Contact('John A')
    c2 = Contact('Robert B')
    c3 = Contact('John—Robert C')

    for x in c1.all_contacts.search('John'):
        print(x.name)
```
In fact, [] is **syntax sugar** for list().

- Add a longest_key method to the built-in type dict.

```python
class ContactList(list):
    def search(self, name):
        ''' Return all contacts that match name. '''
        matching_contacts = []

        for contact in self:
            if name in contact.name:  # self is a list of objects
                #print(contact.name)
```

matching_contacts.append(contact)
return matching_contacts

class Contact:
    all_contacts = ContactList()  # class variable

def __init__(self, name):
    self.name = name
    self.all_contacts.append(self)

c1 = Contact('John A')
c2 = Contact('Robert B')
c3 = Contact('John–Robert C')
for x in c1.all_contacts.search('John'):
    print(x.name)

#for x in c2.all_contacts.search('John'):
    #print(x.name)

#for x in Contact.all_contacts.search('John'):
    #print(x.name)
Polymorphism

A fancy name.

A cool thing in the object-oriented programming.

Used a lot in toy examples. Not often used in my practice.

Usually achieved by inheritance (e.g., in C++) and method overriding.

Polymorphism - many forms of a method. Used in the context of inheritance (class and subclass). For example,
several subclasses can have the same method called `move()` that behaves differently.

A circle has a method called `getArea()`, and a square has a method called `getArea()`. The function `getArea()` has two forms depending on which object it belongs to.

Treat a class differently *silently* depending on which *subclass* is used.

Different behaviors happen depending on which *subclass* is being used, without having to explicitly know what the subclass actually is.

Same method name, but different actions (method
definitions).

Example:

class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext): # self.ext not declared in class
            raise Exception('Not recognised file format. ')
        self.filename = filename

class MP3File(AudioFile):
    ext = 'mp3'
    def play(self):
        print('playing {} as mp3'.format(self.filename))

class WavFile(AudioFile):
ext = 'wav'
def play(self):
    print('playing {} as wav'.format(self.filename))

# Duck-typing. No inheritance! Works because Python is a dynamic language.
class FlacFile:

def __init__(self, filename):
    if not filename.endswith('.flac'):
        raise Exception('Invalid file format')
    self.filename = filename

def play(self):
    print('playing {} as flac'.format(self.filename))

a = MP3File('music.mp3')
a.play()

b = WavFile('music.wav')
b.play()

c = MP3File('music.wav')
c.play() # will raise an exception

d = FlacFile('music.flac')
d.play()
Duck typing

A *pythonic* form of polymorphism.

Do not need inheritance any more.

The only requirement: correct interface.

Example:

```python
# https://en.wikipedia.org/wiki/Duck_typing
class Duck:
    def fly(self):
        print("Duck flying")
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
class Airplane:
    def fly(self):
        print("Airplane flying")

class Whale:
    def swim(self):
        print("Whale swimming")

def lift_off(entity):
    entity.fly()

duck = Duck()
airplane = Airplane()
whale = Whale()

lift_off(duck) # prints 'Duck flying'
lift_off(airplane) # prints 'Airplane flying'
lift_off(whale) # Throws the error 'Whale object has no attribute 'fly''
This sort of polymorphism in Python is typically called **ducking typing**: “If it walks like a duck or swims like a duck, it is a duck”.

No inheritance is involved. Don’t care. We don’t really care if it really *is* a duck object, as long as it can fly (i.e., has the `fly()` method).
Multiple inheritance

A subclass inherits from more than one superclasses.

class RicohAficio(Copier, Printer, Scanner, Faxer):
  pass

OK if the parent classes have *distinct* interfaces. No conflicts.

Not used very often as it can accidentally create the **Diamond Problem** (or Diamond Inheritance): ambiguity in deciding which parent method (with the same name) to use.
class T:
    def f(self):
        print('Top')

class L(T):
    def f(self):
        print('Left')

class R(T):
    def f(self):
        print('Right')

class B(L, R):
    pass  # B does not override f
One potential consequence of the diamond problem is that the base class can be called twice. The following code demonstrates that.

```python
class T:
    def f(self):
        print('Top')

class L(T):
    def f(self):
        print('Left')
```

class R(T):
    def f(self):
        print('Right')
        T.f(self)

class B(L, R):
    def f(self):
        print('Bottom')
        L.f(self)
        R.f(self)

b = B()
b.f()

#The above statement produces the following:
Therefore, to avoid that, we need **Method Resolution Order** (MRO) (with `super()`).

```python
class T:
    def f(self):
        print('Top')

class L(T):
    def f(self):
        print('Left')
        super().f()
```

---

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
class R(T):
    def f(self):
        print('Right')
        super().f()

class B(L, R):
    def f(self):
        print('Bottom')
        super().f()

b = B()
b.f()  # use super() to make sure f in T is called only once!
#Bottom
#Left
“next” method versus “parent” method.

Many expert programmers recommend against using Multiple Inheritance because it will make our code messy and hard to debug.

Alternative: composition, instead of inheritance. Include an object from the superclass and use the methods in that object.

mixin. A mixin is a superclass that provides extra functionality.
class Contact:

    all_contacts = []  # class variable

    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.all_contacts.append(self)

    def __str__(self):
        return '%s <%s>' % (self.name, self.email)

class MailSender:

    def send_mail(self, message):
        print('Sending mail to %s with the following content:
%s' % (self.name, message))
class EmailableContact(Contact, MailSender):
    pass

e = EmailableContact('John Smith', 'jsmith@gitee.com')
e.send_email('Hello how are you doing')

In the above example, MailSender is a mixin superclass.

class AddressHolder:
    def __init__(self, street, city, province, code):
        self.street = street
        self.city = city
        self.province = province
self.code = code

class Contact:

    all_contacts = []  # class variable

    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.all_contacts.append(self)

    def __str__(self):
        return '%s <%s>' % (self.name, self.email)

class Friend(Contact, AddressHolder):
    def __init__(self, name, email, phone, street, city, province, code):
        Contact.__init__(self, name, email)  # cannot use super() here
        AddressHolder.__init__(self, street, city, province, code)
self.phone = phone
Hiding details and creating public interface

Design jobs:

- Derive objects by reading the requirements
- Make the objects interact by determining the public interface.

IMPORTANT.

API meetings.
Make it stable. Why?

Make it simple.

Make it easy to use.

Public Interface: the collection of attributes and methods that other objects can use to interact with that object.

As class users/clients, it is good enough to just know the interface (API documentation) without needing to worry about its internal workings. As class designers/programmers, they should keep the interface stable while making changes to its internals so that users’ code can still work (without modification).
The remote control is our interface to the Television. Each button is like a method that can be called on the TV.

We don’t care:

• Signal transmission from antenna/cable/satellite

• How the signals are converted to pictures and sound.

• Signal sent to adjust the volume

Vendor machines, cellphones, Microwaves, Cars, and Jets.

**Information hiding:** the process of hiding attributes and
functional details (internal workings). Sometimes loosely called **encapsulation**.

In Python, we don’t have or need *true* information hiding.

We should focus on the level of detail most appropriate to a given task and ignore irrelevant details while designing a class. This is called **abstraction**.

**Abstraction** is an object-oriented principle related to information hiding and encapsulation. Abstraction is the process of encapsulating information with separate public and private interfaces. The private information can be subject to information hiding.
A car driver has a different task domain from a car mechanic.

Driver needs access to brakes, gas pedal and should be able to steer, change gears and apply brake.

Mechanic needs access to disc brakes, fuel injected engine, automatic transmission and should be able to adjust brake and change oil.

So a car can have different abstraction levels, depending on who operates it.

Design tips:

• Keep the interface simple.
• When abstracting interfaces, model exactly what needs to be modeled and nothing more.

• Imagine that the object has a strong preference for privacy.
Packages, modules, classes and methods

Usually, methods are organized in a class, classes in a module (a file), and modules in a package.

A module is a file containing class/function definitions.

For small projects, just put all classes in one file. So we got only one module. For example, Lab3.py.

A package is a folder containing a few modules. We must create an empty `__init__` file under that folder to make the folder a package.
There are two options for importing modules: import and from-import.

Use simple imports if the imported modules are under the same folder as the importing file, or in system path.

```python
import database

db = database.Database()

from database import Database

db = Database()

from database import Database as DB

db = DB()
```
• import

```
import package.module  # use period operator to separate packages or modules.
c = package.module.UserClass()
```

For example,

```
import math

math.sqrt(4)
```

Can you do `import math.sqrt`? No. `sqrt` is not a module.

We can do `from math import sqrt`.

• from-import, or from-import-as.
from package.module import UserClass

c = UserClass()

For example,
from math import sqrt

sqrt(4)
Organizing modules to packages and properly importing them

```bash
proj/
main.py
ecommerce/
  __init__.py
database.py
products.py
payments/
  __init__.py
paypal.py
  creditcard.py
```

How to use the module paypal.py in main.py? Use `absolute`
imports, which specify the complete path.

Each module in the package can use absolute imports too. (But it won’t work if we want to run this module as main program. One solution is move the whole package to a system path called site-packages. We can get all system paths using sys.path.)

Module-level code will be executed immediately when the module is imported.

```python
import ecommerce.payments.paypal
ecommerce.payments.paypal.pay()

from ecommerce.payments.paypal import pay
pay()
```
from ecommerce.payments import paypal
paypal.pay()

main.py:
from ecommerce.database import Database
from ecommerce.products import Product

db = Database()
product = Product('Bordeaux Red Wine')
print(product)

from ecommerce.payments import paypal
paypal.pay()
print(__name__)

import sys
print(sys.path)

ecommerce/products.py:

from ecommerce.database import Database

class Product:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name

if __name__ == '__main__':
    p = Product('E45')
    print(p)

ecommerce/payments/paypal.py:

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
from ecommerce.products import Product

print('My __name__ is %s' % (__name__))

def pay():
    p = Product('E45 Hair Lotion')
    print('Pay %s using PayPal' % (p))

if __name__ == '__main__':
    print('Make a Product object')
    p = Product('E45 Body Lotion')
Each module has a special, hidden variable called __name__

Use print(__name__) to check its value.

If you run that module as a main program (note that each module is a file), then __name__ is equal to '__main__'.

We say running a module as a main program if we type in the command line like this: python module_name.py.

If you import that module, then that module’s __name__ contains its actual file name. For example, paypal.py’s
__name__ is “ecommerce.payments.paypal” when we import the module paypal.py using from ecommerce.payments import paypal.

When we import a module, the module’s code will be executed. But since that module’s __name__ is not ‘__main__’, then the code under that module’s if __name__ == '__main__': won’t be executed.

So it is a good idea to put if __name__ == '__main__': in the end of every module (main module or not) and put the test code for that module after it.
Organizing module contents

A typical order in a Python module:

```python
class UsefulClass:
    ''' This class might be useful to other modules. '''
    pass

def main():
    ''' Do something with it for our module. '''
    u = UsefulClass()
    print(u)

if __name__ == '__main__':
    main()
```
Inner classes and inner functions. Usually used as an *one-off* helper, not to be used by other methods or other modules.

```python
def format_string(s, formatter=None):
    ''' Format a string using the formatter object, which is expected
    to have a format() method that accepts a string. '''

    # AN INNER CLASS
    class DefaultFormatter:
        def format(self, s):
            ''' Return a string in title case '''
            return str(s).title()

        if not formatter:
            formatter = DefaultFormatter()

        return formatter.format(s)
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
def format_string2(s):
    # AN INNER FUNCTION
    def helper(w):
        '''Make the first letter uppercase'''
        return w[0].upper() + w[1:]
    result = ''

    for w in s.split():
        result += helper(w) + ' '  
    return result.strip()

if __name__ == '__main__':
    print(format_string('hello world'))
    print(format_string2('hello world'))
Public, protected and private attributes

class Wallet:
    ''' For demonstrating public, protected and private attributes. '''

    def __init__(self, rmb=0, cad=0, gbp=0):
        self.rmb = rmb
        self._cad = cad
        self.__gbp = gbp

    def deposit(self, amount, currency):
        if currency.lower() == 'rmb':
            self.rmb += amount
        if currency.lower() == 'cad':
            self._cad += amount
        if currency.lower() == 'gbp':
            self.__gbp += amount
def withdraw(self, amount, currency):
    if currency.lower() == 'rmb':
        self.rmb -= amount
    if currency.lower() == 'cad':
        self._cad -= amount
    if currency.lower() == 'gbp':
        self._gbp -= amount

def check(self):
    s = ''
    if self.rmb > 0:
        s += ' %.0f RMB ' % self.rmb
    if self._cad > 0:
        s += ' %.0f CAD ' % self._cad
    if self._gbp > 0:
        s += ' %.0f GBP ' % self._gbp
return s

Attribute names with two underscores are not visible.

```python
from wallet import Wallet

w = Wallet()
print(w.rmb)
print(w._cad)
print(w._gbp)

#0
#0
#Traceback (most recent call last):
  #File "C:/Users/Hui/Downloads/oop_prep/wallet_test.py", line 6, in <module>
    #print(w._gbp)
#builtins.AttributeError: 'Wallet' object has no attribute '__gbp'
```
# Wallet

```python
<wallet.Wallet object at 0x0000000002A9D390>

<wallet.Wallet'>
```

```python
<Wallet.
dict
```

```python
mappingproxy({'__module__': 'wallet', '__doc__': 'For demonstrating public, protected and private attributes.'})
```

```python
w.
dict*
```

```python
#{'rmb': 0, '_cad': 0, '_Wallet__gbp': 0}
```
Abstract methods and interfaces

No interface keyword in Python. We use Abstract Base Classes (ABCs) instead.

All subclasses derived from an abstract base class must implement the abstract methods (marked by @abstractmethod). This is forced. It is like a contract between the class user and the class implementer.

We cannot instantiate an abstract base class. We cannot instantiate a subclass of abstract class without defining all its abstract methods.
Specify method names in abstract class, and implement these methods in subclasses.

**Built-in ABCs:**

```python
from collections.abc import Container

# Using the existing built-in abstract base class Container
class EvenContainer:
    def __contains__(self, x):
        # implement the abstract method in Container
        if not isinstance(x, int) or x % 2 != 0:
            return False
        return True

c = EvenContainer()
print(issubclass(EvenContainer, Container))  # True
print(isinstance(c, Container))  # True
```
Zero inheritance used above! However, the *is-a* relationship is established.

You can’t instantiate an object from a subclass which inherits from an ABC if that subclass does not have all the required abstract methods specified in its parent ABC. Example:

```python
from collections.abc import Container

# Inheriting from the built-in abstract base class Container
class EvenNumber(Container):
    # missed 's', therefore missing the required method __contains__
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```python
def __contains__(self, x):
    if not isinstance(x, int) or x % 2 != 0:
        return False
    return True
```

```python
# TypeError: Can't instantiate abstract class EvenContainer
```

```python
c = EvenNumber()
```

**Customized ABCs:**

```python
# simplified from https://python-course.eu/python3_abstract_classes.php
from abc import ABC, abstractmethod
class A(ABC):
    @abstractmethod
def speak(self):
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
print("Un–gu")

@abstractmethod
def add(self, a, b):
    pass

class S(A):
    def speak(self):
        super().speak()
        print("Every Sha–la–la–la Every Wo–o–wo–o")

    def add(self, x, y):
        return x + y

a = S()
a.speak()
Duck-typing and isinstance.

```python
# https://en.wikipedia.org/wiki/Duck_typing
from abc import ABC, abstractmethod

class Bird(ABC):
    @abstractmethod
    def fly(self):
        pass

@classmethod
def __subclasshook__(cls, subclass):
    if cls is Bird:
        attrs = set(dir(subclass))
        print('Executing __subclasshook__')
        if set(cls.__abstractmethods__) <= attrs:
            return True
        return NotImplemented
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
class Duck(Bird):
    def fly(self):
        print("Duck flying")

class Airplane(Bird):
    def fly(self):
        print("Airplane flying")

class ParkerSolarProbe:
    def fly(self):
        print("Destination is sun.")

duck = Duck()
airplane = Airplane()
parker = ParkerSolarProbe()

#isinstance(duck, ABC)
#isinstance(duck, Bird)
#isinstance(airplane, ABC)
#isinstance(airplane, Bird)

isinstance(parker, Bird) # Surprise! parker is not an object derived from Bird

issubclass(ParkerSolarProbe, Bird)

⊙ signifies a decorator.
Expecting the Unexpected

An exception is not expected to happen often. Use exception handling for really exceptional cases.

Cleaner code; more efficient.

**Look before you leap.** We can use the if–elif–else clause, why do we bother with exception?

**Ask forgiveness rather than permission.** Exception is not a bad thing, not something to avoid. It is a powerful way to communicate information (pass messages).
What does an exception class look like?

```python
class IndexError(LookupError):
    Sequence index out of `range`.
```

Method resolution order:
- IndexError
- LookupError
- Exception
- BaseException
- object

The exception hierarchy.

```
BaseException
    ^
SystemExit  KeyboardInterrupt  Exception
    ^
```
Most Other Exception

Other built-in errors: `ValueError`, `TypeError`, `KeyError`, `ZeroDivisionError` and `AttributeError`.

```python
class EvenOnly(list):
    def append(self, integer):
        if not isinstance(integer, int):
            raise TypeError('Only integer can be added. ')
        if integer % 2 != 0:
            raise ValueError('Only even numbers can be added. ')
        super().append(integer)

if __name__ == '__main__':
    L = EvenOnly()
    L.append(2)  # OK
    L.append('2')  # `builtins.TypeError: Only integer can be added.'
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
L.append(3) # builtins.ValueError: Only even numbers can be added.

try-except:
from EvenOnly import EvenOnly

if __name__ == '__main__':
    L = EvenOnly()
    try:
        L.append(2) # OK
        L.append('2') # raise TypeError and jump to except TypeError
        L.append(3) # won’t be reached
    except TypeError:
        print('Encountered a Type Error')
    except ValueError:
        print('Encountered a Value Error')

a = [1, 2, 3, 4]
b = 0
try:
    b = a[1] + a[4]
except Exception as e:
    print('Cannot add for some reason')
    print(type(e))  # we can do something on the object e
print('b is %d' % b)

We can omit “Exception as e” or use LookupError or IndexError instead. Using IndexError is best as we are explicit here which exception we want to catch (and then handle).

def no_return():
    print('1')
    raise Exception('Always raised.')
    print('2')  # Warning: this code will never be reached.
That is a surprise.

def f():
    print('3')
    no_return()
    print('4')

try:
    f()
except Exception as e:
    print('Exception handled. Arguments: %s' % e.args)

# 3
# 1
# Exception handled. Arguments: Always raised.
```python
Good floor:

def good_floor(n):
    if n == 4:
        raise Exception('Four is not a good number for Chinese')
    if n == 10:
        raise Exception('Four is not a good number for Chinese')
    if n == 13:
        raise Exception('Four is not a good number for Westerners')
    return n

import random
try:
    n = good_floor(random.randint(1,30))
except Exception as e:
    print('%s' % e.args)
else:
    print('%d' % n)
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
Stack exception clauses.

def funny_division(divider):
    try:
        return 100/divider
    except ZeroDivisionError:
        return 'Zero is bad as a divisor'

def funny_division2(divider):
    try:
        if divider == 13:
            raise ValueError('13 is an unlucky number. ')
        return 100/divider
    except (ZeroDivisionError, TypeError):
        return 'Zero is bad as a divisor. Non–zero values only.'

def funny_division3(divider):
try:
    if divider == 13:
        raise ValueError('13 is an unlucky number.')
    return 100/divider
except ZeroDivisionError:
    return 'Zero is bad as a divisor.'
except TypeError:
    return 'String is bad as a divisor.'
except ValueError:
    print('Cannot accept 13')
    raise  # raise the last exception ValueError

# test funny_division
print(funny_division(0))
print(funny_division(50.0))
# print(funny_division('0.0')) # this will raise builtins.TypeError: unsupported operand type(s) for /: 'str' and 'float'

# test funny_division2
```python
for v in [0, 'hello', 50.0, 13]:
    print('Testing {}:'.format(v), end=" ")
    #print(funny_division2(v))

# test funny_division3
for v in [0, 'hello', 50.0, 13]:
    print('Testing {}:'.format(v), end=" ")
    print(funny_division3(v))

try-except-else-finally:

a = [1, 2, 3, '4']
b = 0
try:
except IndexError:
    print('Cannot add due to Index Error')
except TypeError:
    print('Cannot add due to Type Error')
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```
print('Cannot add due to Type Error')
else:
    print('If there are no exceptions, I can be reached.')
finally:  # will be executed no matter what happens
    print('Whether or not there are exceptions, I can be reached.')
print('b is %d' % b)
```

```python
import random

exceptions = [ValueError, TypeError, IndexError, None]

try:
    choice = random.choice(exceptions)
    print('Raising {}' . format(choice))
    if choice:
        raise choice('An error')
except ValueError:
```
```python
print('Caught a ValueError.
except TypeError:
    print('Caught a TypeError.
except Exception as e:
    # a more general exception
    print('Caught some other error: %s.' % (e.__class__.__name__)))
else:
    print('No exception case.')
finally:
    print('Always reached.')
```

Things under `finally` will be executed no matter what happens (a good place to put clean-up statements). Extremely useful for

- Cleaning up an open database connection
• Closing an open file

We can use try-finally without the except clause.
Customized exceptions

Inherit from class Exception. Add information to the exception.

class InvalidWithdrawal(Exception):
    def __init__(self, balance, amount):
        super().__init__('account does not have ${}'.format(amount))
        self.amount = amount
        self.balance = balance

    def overdraft(self):
        return self.amount - self.balance

try:
    raise InvalidWithdrawal(25, 50)
except InvalidWithdrawal as e:
    print('Overdraft ${}'.format(e.overdraft()))
Getters, setters and @property decorator

Data encapsulation.

Make methods look like attributes.

“Blurring the distinction between behavior and data”.

No change needed in the client code, therefore achieving backward compatible.

See the Color example, starting from page 130 in Dusty’s book.
```python
class Man:
    def __init__(self, height):
        self.height = height

class Man2:
    ''' Later, we want to add some contraints to height ... '''
    def __init__(self, height):
        self._height = height  # in cm

@property
def height(self):
    print('xxx')
    return self._height

@height.setter
def height(self, h):
```

---

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```python
print('yyy')
if h < 100:
    raise Exception('Too short!')
elif h > 250:
    raise Exception('Too tall!')
self._height = h

m = Man2(170)
m.height
m.height = 213
m.height = 90  # too short exception
m.height = 321  # too tall exception
```
More on @property

property is in fact a special function that returns a property object.

```python
property(fget=None, fset=None, fdel=None, doc=None)
```

```python
p = property()
dir(p)  # attributes ['fdel', 'fget', 'fset', 'getter', 'setter']
help(p)
```

```python
Man2.height  # <property object at 0x0000000002A2FE58>
Man2.height.fget(m)
Man2.height.fset(m, 123)
Man2.height.__getattribute__('getter')
Man2.height.__getattribute__('setter')
```
class Score0:
    def __init__(self, value, date):
        self.value = value
        self.date = date

class Score:
    def __init__(self, value, date):
        self._score = value
        self._date = date

@property
def value(self):
    '''Return the value of the score.
    '''
    return self._score

@value.setter
```python
def value(self, v):
    if v > 100 or v < 0:
        raise Exception('Score value out of range.')
    self._score = v

## Note:
##
## @property
## def value(self):
##     return self._score
##
## is the same as
##
## def value(self):
##     return self._score
## value = property(value)
##
## More explanation at https://stackoverflow.com/questions/17330160/how
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
class Score2:
    def __init__(self, value, date):
        self._score = value
        self._date = date

    def getter(self):
        ''' Return the value of the score. '''
        return self._score

    def setter(self, v):
        if v > 100 or v < 0:
            raise Exception('Score value out of range. ')
        self._score = v

    value = property(fget=getter, fset=setter)
s0 = Score0(60, '20191216')
s0.value
s0.value = 101

s = Score(60, '20191216') # check validity for the value of score [0, 100]
s.value = 101

s2 = Score2(60, '20191216')
s2.value = -1

Another use case: to cache data.

from urllib.request import urlopen

class WebPage:
    def __init__(self, url):
        self.url = url
        self._content = None
@property
def content(self):
    '''Return the content of a web page.'''
    if self._content is None:
        self._content = urlopen(self.url).read()
    return self._content

page = WebPage('https://www.gutenberg.org/cache/epub/1228/pg1228.txt')
print('Access the free book for the first time')
content1 = page.content  # Need to wait for a while
print('Done')
print('Access the free book for the second time')
content2 = page.content  # Almost instantly done
print('Done')
More on decorators

A decorator is a function which adds some toppings to the function being decorated.

```python
def steamed_milk(func):  # Note: the argument is a function name!
    def decor():
        return 'Steamed milk * ' + func()  # add some extra flavour to
    return decor

def foamed_milk(func):
    def decor():
        return 'Foamed milk * ' + func()
    return decor

@steamed_milk
```
def coffee1():
    return 'Espresso'

@foamed_milk
def coffee2():
    return 'Espresso'

@steamed_milk
@foamed_milk
def coffee3():
    return 'Espresso'

def coffee():
    return 'Espresso'

coffee4 = steamed_milk(foamed_milk(coffee))
c = coffee1()
print(c)  # Steamed milk * Espresso


c = coffee2()
print(c)  # Foamed milk * Espresso


c = coffee3()
print(c)  # Steamed milk * Foamed milk * Espresso


c = coffee4()
print(c)  # Steamed milk * Foamed milk * Espresso

In the above example, @steamed_milk is a decorator. steamed_milk is a function that accepts one argument, the
name of the decorated function. steamed_milk returns its inner function, decor. The inner function adds some “toppings” (Steamed milk).

With overriding (single inheritance) we can add some toppings.

With mixin (multiple inheritance) we can also add some toppings.

We can stack decorators.
Design patterns

Bridges (stone arch, suspension, cantilever, etc).

We have proven bridge structures that work.

Standard solutions to design for frequently encountered problems.

**Pattern**: an example for others to follow.
Design patterns - the iterator pattern

An iterator object usually has a `next` method and a `done` method.

```python
while not iterator.done():
    item = iterator.next()
    # do sth with the item
```

In Python,

- We have a special method called `__next__`, accessible through `next(iterator)`. 
• There is no `done` method. Raise exception `StopIteration` instead.

```python
# from collections.abc import Iterator, Iterable

class CapitallIterator:
    def __init__(self, s):
        self.words = [w.title() for w in s.split()]
        self.index = 0

    def __next__(self):
        if self.index == len(self.words):
            # no done() method, raise StopIteration instead
            raise StopIteration()
        word = self.words[self.index]
        self.index += 1
        return word
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```python
def __iter__(self):
    return self
class CapitalIterable: # we can get an iterator from it
    def __init__(self, s):
        self.s = s
    def __iter__(self):
        return CapitalIterator(self.s)

# iterable then iterator
table = CapitalIterable('the brightest star in the night sky')

# Get an iterator using iter()
# The argument must supply its own iterator, we have CapitalIterator.
iterator = iter(iterator) # call __iter__
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```python
while True:
    try:
        print(next(iterator))  # same as iterator.__next__()
    except StopIteration:
        break

for w in iterable:
    print(w)

for w in iterable:
    print(w)

iterator = iter(iterable)
for w in iterator:
    print(w)

# iterator has been exhausted
```
for w in iterator:
    print(w)

Internally, a for loop is actually a while loop.

iter([1,2,3])
<list_iterator object at 0x000000002C40B00>

iter('123')
<str_iterator object at 0x000000002C45E10>

iter({'a':1, 'b':2})
<dict_keyiterator object at 0x000000002AD7098>

Two abstract base classes in collections.abc:

• Iterable must define __iter__. 
• Iterator must define `__next__` and `__iter__`, collectively called the iterator protocol.

File objects are iterators too. It has method `__next__` and `__iter__` (inherited from `IOBase`)

If there is no `StopIteration`, we have an infinite iterator.

```python
import random
class RandomIterator:
    def __init__(self, n):
        self.n = n
    def __iter__(self):
        return self
    def __next__(self):
        return random.randint(0, self.n)
```
```python
r = RandomIterator(56)
print(next(r))
print(next(r))
print(next(r))
print(next(r))
# we have infinitely many of them
```

Is a list object of type `Iterator`? No. Is a list object of type `Iterable`? Yes.

```python
from collections.abc import Iterator, Iterable
lst = [1, 2, 3]
print(isinstance(lst, Iterator))  # False: list is not Iterator
print(isinstance(iter(lst), Iterator))  # True: but can be converted to
print(isinstance(lst, Iterable))  # True: list is Iterable
```

Where can I find the definition of classes `Iterator` and
Exercise: verify that a file object is an Iterator object.
Comprehensions

Convert/map a list of items of this form to a list of items of that form. Each item from the original list can be passed into a function and the return value of that function becomes the new item in the converted list.

Usually done in **one** line of code, called one-liner.

Benefits: very concise expression.

We have *list* comprehensions, *set* comprehensions and *dictionary* comprehensions.
“List comprehensions are far faster than for loops when looping over a huge number of items.” - I don’t agree as of November 2018. In fact, they have similar performance.

```python
# For-loop and comprehension have similar performance.
slst = ['1', '12', '123', '1234']
int_lst = [int(s) for s in slst if len(s) > 2]

import random, time
N = 1000000
start = time.time()
big_slst = [''.join(random.choices('0123456789', k=random.randint(1, 5)) for i in range(N))]  # generate N strings, each string contains
end = time.time()
print('Time used [comprehension]: %4.2f' % (end-start))

# another way of generating big_slst
```
```
start = time.time()
big_slst2 = []
for i in range(N):
    big_slst2.append(''.join(random.choices('0123456789', k=random.randint(1, 5))))
end = time.time()
print('Time used [for]: %4.2f' % (end-start))

# Comprehension
start = time.time()
result = [int(s) for s in big_slst if len(s) > 2]
end = time.time()
print('Time used [comprehension]: %4.2f' % (end-start))

# The for loop
start = time.time()
result = []
```
for s in big_slist:
    if len(s) > 2:
        result.append(s)
end = time.time()
print('Time used [for]: %4.2f' % (end-start))

Time used [comprehension]: 5.93
Time used [for]: 6.19
Time used [comprehension]: 0.31
Time used [for]: 0.30

We can also make set comprehensions or dictionary comprehensions. To do that, we use braces instead of brackets.

from collections import namedtuple
Book = namedtuple('Book', 'author title genre')
books = [
    Book('Pratchett', 'Thief of Time', 'fantasy'),
    Book('Pratchett', 'Nightwatch', 'fantasy'),
    Book('Le Guin', 'The Dispossessed', 'scifi'),
    Book('Le Guin', 'A Wizard of Earthsea', 'fantasy'),
]

fantasy_authors = {b.author for b in books if b.genre == 'fantasy'}
print(fantasy_authors)  # print 'Pratchett' only once though he has 2 fantasy books

fantasy_titles = {b.title:b for b in books if b.genre == 'fantasy'}
print(fantasy_titles)

Lists, sets and dictionaries are called *containers*.

In JavaScript,

```javascript
const updatedPoints =
    points.map(pt => ({ x: pt.x + dx, y: pt.y + dy }));
```
let lengths = ["ooad", "ase", "softarch"].map(course => course.length)
Generators

Sometimes the data is too large, e.g., GB or TB, and we don’t need all of it at once, so we don’t need to load everything into computer memory.

Try this:

```python
lst = [2*x for x in range(1000000000)]  # list comprehension
```

Is your computer freezing now? Is your RAM fully occupied now?
We can also use a comprehension-like expression, called a generator.

\[
\text{lst} = (2 \times x \text{ for } x \text{ in range}(1000000000)) \quad \# \text{ generator expression}
\]

Another example of generator expression:

```python
import sys

ip = '194.151.73.43' \# sys.argv[1]
log_file = 'access_log' \# sys.argv[2]

with open(log_file) as f:
    interesting_lines = (line for line in f if ip in line) \# a generator
    for line in interesting_lines:
        print(line)
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
Use *yield* inside a *function* to make a generator.

```python
bad_ip = '194.151.73.43'
log_file = 'access_log'

def ip_filter(f, ip):
    for line in f:
        if ip in line:
            yield line.replace(ip, '')  # yield a line with ip removed

with open(log_file) as f:
    filter = ip_filter(f, bad_ip)
    for line in filter:
        print(line)
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
What is the type of \texttt{filter}? It is a generator, which can be used to generate many values (one-the-fly). A generator is an iterator (since it has methods \texttt{\_\_next\_\_} and \texttt{\_\_iter\_\_}), which will be exhausted after one pass.

The generator is created without executing the code in the function body. When we put the generator in a \texttt{for} loop (which internally call the method \texttt{\_\_next\_\_}), each iteration will stop after the \texttt{yield} statement.

Question: Is an iterator also a generator? No, because a generator needs methods \texttt{send()} and \texttt{throw()} too, which an iterator does not have.
What is going on inside ip_filter looks like:

```python
import sys
from collections.abc import Iterator, Generator

bad_ip = '194.151.73.43'
log_file = 'access_log'

class IPFilter:
    ''' Iterator, but not generator '''
    def __init__(self, f, ip):
        self.f = f
        self.ip = ip
    def __iter__(self):
        return self
    def __next__(self):
        line = self.f.readline()
        while line and not self.ip in line:
            line = self.f.readline()
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
if not line:
    raise StopIteration
return line.replace(self.ip, '')

with open(log_file) as f:
    filter = IPFilter(f, bad_ip)
    for line in filter:
        print(line)

# Iterator, but not generator
print(isinstance(filter, Iterator))
print(isinstance(filter, Generator))

# See https://anandology.com/python-practice-book/iterators.html

def foo():
    print("BEGIN")
    for i in range(3):
```python
print("before yield %d." % i)
yield i
print("after yield %d." % i)
print("END")

gen = foo()
x = next(gen)
y = next(gen)
z = next(gen)
a = next(gen)

```

```
BEGIN
before yield 0.
after yield 0.
before yield 1.
after yield 1.
```
before yield 2.
after yield 2.
END

Traceback (most recent call last):
  File "C:/Users/Hui/Downloads/ZJNU/OO/oop_prep/apache-samples/generator3.py", line 15, in <module>
a = next(gen)
builtins.StopIteration:

,,,

\[ x \text{ is } 0, \ y \text{ is } 1 \text{ and } z \text{ is } 2. \]

Yield data from another generator using \texttt{yield from}.

\begin{verbatim}
def foo():
    for i in range(3):
        yield i
\end{verbatim}
```python
def bar():
    generator = foo()
    yield from generator

b = bar()
print(next(b))
print(next(b))
print(next(b))
```

List all files and directories under a directory:

```python
class File:
    def __init__(self, name):
        self.name = name

class Dir(File):
    def __init__(self, name):
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui

205
super().__init__(name)
self.children = []

proj
- run.py
- templates
  - a.html
  - b.html
  - old
    - a0.html
    - b0.html

old = Dir('old')
old.children.append(File('a0.html'))
old.children.append(File('b0.html'))
templates = Dir('templates')
templates.children.append(File('a.html'))
templates.children.append(File('b.html'))
templates.children.append(old)

proj = Dir('proj')
proj.children.append(File('run.py'))
proj.children.append(templates)

def walk(d):
    if isinstance(d, Dir):
        yield d.name + '/'
        for f in d.children:
            yield from walk(f)
    else:
        yield d.name
for f in walk(proj):
    print(f)
Coroutines

"Generators with a bit extra syntax."

A coroutine is a special routine in which you can pass data in (using `send` in Python) and get data out (using `yield` in Python).

Distinguish a coroutine with a function.

Execution order:

- `yield` occurs and the generator pauses.
• `send()` occurs and the generator resumes.

• The value sent in is assigned (to the left side of the `yield` statement).

• The generator continues until the next `yield` statement.

def coroutine(y):
    for i in range(y):
        x = yield i
        print('i=%d, x=%d' % (i, x))

c = coroutine(4)
next(c) # generate 0
```python
c.send(10)  # print i=0, x=10 and generate 1
c.send(20)
c.send(30)

#Output:
#i=0, x=10
#i=1, x=20
#i=2, x=30

def echo():
    just_received = 'nothing'
    try:
        while True:
            received = yield just_received
            just_received = received
            print('I got {}'.format(just_received))
    except GeneratorExit:  # when closed with close()
        print('Coroutine closed!')
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```python
def tally():
    score = 0
    while True:
        incr = yield score  # incr captures the sent value
```

```python
# g = echo()
# next(g)
# 'nothing'
# g.send(1)
# I got 1.
# 1
# g.send(2)
# I got 2.
# 2
# g.close()
# Coroutine closed!
```
score += incr

bluejays = tally()
next(bluejays)
bluejays.send(1)  # return next yielded value or raise StopIteration.
bluejays.send(2)

whitesox = tally()
next(whitesox)
whitesox.send(2)
whitesox.send(1)

Linux kernel log parsing:

unrelated log messages
sd 0:0:0:0 Attached Disk Drive
unrelated log messages
sd 0:0:0:0 (SERIAL=ZZ12345)
unrelated log messages
sd 0:0:0:0 [sda] Options
unrelated log messages
XFS ERROR [sda]

unrelated log messages
sd 2:0:0:1 Attached Disk Drive
unrelated log messages
sd 2:0:0:1 (SERIAL=ZZ67890)
unrelated log messages
sd 2:0:0:1 [sdb] Options
unrelated log messages

sd 3:0:1:8 Attached Disk Drive
unrelated log messages
sd 3:0:1:8 (SERIAL=WW11111)
unrelated log messages
sd 3:0:1:8 [sdc] Options
unrelated log messages
XFS ERROR [sdc]
unrelated log messages

import re
```python
def match_regex(fname, regex):
    with open(fname) as f:
        lines = f.readlines()
    for line in reversed(lines):
        m = re.match(regex, line)
        if m:
            regex = yield m.groups()[0]

def get_serials(fname):
    ERROR_RE = 'XFS ERROR ([sd][a-z])'
    matcher = match_regex(fname, ERROR_RE)
    device = next(matcher)
    while True:
        bus = matcher.send(' (sd \S+){}*'.format(re.escape(device)))
        serial = matcher.send(' {} \(SERIAL=([^])\)'.format(bus))
        yield serial
        device = matcher.send(ERROR_RE)
```

for serial_number in get_serials('EXAMPLE_LOG.log'):
    print(serial_number)
Design patterns - the observer pattern

- One **core** object

- Several **observer** objects

  The observers are told (updated with) changes occurred to the **core** object.

  (The observers can then take their own actions.)

  Useful for making redundant backup (in database, remote host, local file, etc).
Useful for broadcasting an announcement (using email, text messages, and other instant messaging systems).

```python
class Inventory:
    
    An inventory is going to be monitored by a number of observers. The observers will be notified with the changes made to an Inventory object’s attributes product and quantity. See page 307 in Dusty Phillips’ book.

    def __init__(self):
        self.observers = []
        self._product = None
        self._quantity = 0

    def attach(self, observer):
        self.observers.append(observer)
```
@property
def product(self):
    return self._product

@product.setter
def product(self, value):
    self._product = value
    self._update_observers()

@property
def quantity(self):
    return self._quantity

@quantity.setter
def quantity(self, value):
    self._quantity = value
    self._update_observers()
```python
def _update_observers(self):
    for o in self.observers:
        o()

class ConsoleObserver:
    def __init__(self, inventory):
        self.inventory = inventory

    def __call__(self):
        print(self.inventory.product)
        print(self.inventory.quantity)

class UnitedKingdomObserver(ConsoleObserver):
    def __call__(self):
        print('Observer from Britain')
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```python
print(self.inventory.product)
print(self.inventory.quantity)

class UnitedStatesObserver(ConsoleObserver):
    def __call__(self):
        print('Observer from America')
        print(self.inventory.product)
        print(self.inventory.quantity)

i = Inventory()
uk_observer = UnitedKingdomObserver(i)
i.attach(uk_observer)
us_observer = UnitedStatesObserver(i)
i.attach(us_observer)

i.product = 'E45'
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
In the above example, whenever we change the properties quantity and product, the observers get updated (because we called the method _update_observers_).

We can use o() because we have defined the special method __call__.

The main point of using the observer pattern is that we can add (attach) different observers (having different behaviors) upon any change in the core object.

Different behaviors:
• Back up the data in a file.
• Back up the data in a database.
• Back up the data in an Internet application.
• Back up the data in a tape.

Main benefit: **Code detachment**. Shorten the core class. Plug in as many observer classes as we want.

We don’t have to mix code handling *multiple behaviors* code in the observed object. Instead, we put such code in individual observers, facilitating maintainability.
Design patterns - the strategy pattern

Provide different solutions to a single problem (e.g., sorting), each in a different object.

• Quick sort

• Merge sort

• Heap sort

Use if statement?
from PIL import Image

class TiledStrategy:
    def make_background(self, img_file, desktop_size):
        in_img = Image.open(img_file)
        out_img = Image.new('RGB', desktop_size)
        num_tiles = [o // i + 1 for o, i in zip(out_img.size, in_img.size)]
        # num_tiles looks like [3,2], where 3 is number of rows, and 2
        # of columns
        for x in range(num_tiles[0]):
            for y in range(num_tiles[1]):
                # the second argument for paste is a 4-tuple defining
                # left, upper, right, and lower
                out_img.paste(in_img,
                (in_img.size[0]*x, in_img.size[1]*y,
                in_img.size[0]*(x+1), in_img.size[1]*(y+1)))

        return out_img  # call out_img.save('result.jpg') to save the file
```python
class CenteredStrategy:
    def make_background(self, img_file, desktop_size):
        in_img = Image.open(img_file)
        out_img = Image.new('RGB', desktop_size)
        left = (out_img.size[0] - in_img.size[0]) // 2
        top = (out_img.size[1] - in_img.size[1]) // 2
        out_img.paste(in_img, (left, top, left + in_img.size[0], top + in_img.size[1]))
        return out_img

class ScaledStrategy:
    def make_background(self, img_file, desktop_size):
        in_img = Image.open(img_file)
        out_img = in_img.resize(desktop_size)
        return out_img

if __name__ == '__main__':
```

Each class has a single method, and the method names are the same in all three classes.

Each class does nothing else except provide a single function.
We can replace `make_background` with `__call__` to make the object directly callable (for example, `strategy(deep-autumn.jpg’, (2400, 1200))`).
Design patterns - the state pattern

See page 314 in the textbook.

The XML file book.xml is shown below.

```xml
<book>
  <author>Phillips</author>
  <publisher>PACKT</publisher>
  <title>OOP</title>
  <content>
    <chapter>
      <number>1</number>
      <title>Design</title>
    </section>
  </content>
</book>
```
1.1 Introducing Object-oriented

2 Creating Python Classes
Each class represents a state (see page 316 for the state transition diagram).

Each state class has the same method called process, which takes the context object parser as an argument. process consumes the input string, edits the tree, and modifies parser's state.

Note that the context class Parser also has method process, which invokes state object’s process for consuming the remaining string, and recursively calls itself.

class Node:
    ''' A node in a parsing tree. '''
def __init__(self, tag, parent=None):
    self.parent = parent
    self.children = []
    self.tag = tag
    self.text = ''

def __str__(self):
    if self.text:
        return self.tag + ':' + self.text
    else:
        return self.tag

class Parser:
    def __init__(self, s):
        self.s = s
        self.root = None
        self.curr_node = None
        self.state = FirstTag()
```python
def process(self, rs):
    # rs is remaining_string
    rs = self.state.process(rs, self)
    if rs:
        self.process(rs)

def build_tree(self):
    self.process(self.s)

def __str__(self):
    ''' Display the tree structure '''

def tree_structure(root, level=0):
    s = root.__str__() + '
    for n in root.children:
        s += level * '.' + tree_structure(n, level+1)
    return s

return tree_structure(self.root)
```

```
Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```
class FirstTag:
    def process(self, rs, parser):
        # parser is the context
        i = rs.find( '<' )
        j = rs.find( '>' )
        tag = rs[i+1:j]
        root = Node(tag)
        parser.root = root
        parser.curr_node = root
        parser.state = ChildNode()
        return rs[j+1:]

class ChildNode:
    ''' A transition state. '''
    def process(self, rs, parser):
        rs = rs.strip()
        if rs.startswith( '</' ):
```python
parser.state = CloseTag()

elif rs.startswith( '<' ):
    parser.state = OpenTag()

else:
    parser.state = TextNode()

return rs

class OpenTag:
    def process( self, rs, parser ):
        i = rs.find( '<' )
        j = rs.find( '>' )
        tag = rs[ i+1: j ]
        node = Node( tag, parser.curr_node )
        parser.curr_node.children.append( node )  # add child
        parser.curr_node = node
        parser.state = ChildNode()

return rs[ j+1: ]
```
class CloseTag:
    def process(self, rs, parser):
        i = rs.find('(<')
        j = rs.find('>')
        tag = rs[i+2:j]  # skip '/'
        parser.curr_node = parser.curr_node.parent  # move back
        parser.state = ChildNode()
        return rs[j+1:].strip()

classTextNode:
    def process(self, rs, parser):
        i = rs.find('(<')
        text = rs[:i]
        parser.curr_node.text = text
        parser.state = ChildNode()
        return rs[i:]

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
if __name__ == '__main__':
    import sys
    with open('book.xml') as f:
        p = Parser(f.read())
        p.build_tree()
        print(p)

        #nodes = [p.root]
        #while nodes: # list not empty
            #node = nodes.pop(0)
            #print(node)
            #nodes = node.children + nodes # depth-first??

#Output:
#book
#author: Phillips
#publisher: PACKT
#title: OOP
I have to say the above design is quite clever.

Parse `book.xml` with a generator.
class Node:
    ''' The node has a parent and a number of children. '''
    def __init__(self, tag, parent=None):
        self.parent = parent
        self.children = []
        self.tag = tag
        self.text = ''

    def __str__(self):
        if self.text:
            return self.tag + ':' + self.text
        else:
            return self.tag

def eat_open_tag(s):
```python
def eat_close_tag(s):
    i = s.find('<')
    j = s.find('>')
    return s[i+1:j], j

def eat_text(s):
    j = s.find('<')
    return s[:j], j-1

def close_tag(s):
    return s.startswith('</')

def open_tag(s):
    return s.startswith('<') and not close_tag(s)
```
```python
def tree_structure(root, level=0):
    s = root.__str__() + '
    for n in root.children:
        s += level * '.' + tree_structure(n, level+1)
    return s

def parse_xml(s):
    ''''Build a parse tree.''''
    s = s.strip()  # remove empty spaces
    root = None
    curr = None
    first = True  # encountered the first tag?
    while s:  # there are more characters to consume
        if open_tag(s):
            tag, k = eat_open_tag(s)
            if first:
                root = curr = Node(tag)
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
```python
first = False
else:
    # set parent and children
    node = Node(tag, curr)
    curr.children.append(node)
    curr = node
    yield 'Open <\%s>' % tag
elif close_tag(s):
    tag, k = eat_close_tag(s)
    curr = curr.parent
    yield 'Close <\%s>' % tag
else:
    text, k = eat_text(s)
    curr.text = text
    yield 'In \'%s\'' % text
s = s[k+1:]  # consume characters
s = s.strip()
yield root
```

```python
f = open( 'book.xml' )
s = f.read()
p = parse_xml(s)
f.close()
for x in p:
    if isinstance(x, Node):
        print( tree_structure(x) )
    else:
        print(x)
```

Output

Open <book>
Open <author>
In 'Phillips'
Close <author>
Open <publisher>
In 'PACKT'
Close <publisher>
Open <title>
In 'OOP'
Close <title>
Open <content>
Open <chapter>
Open <number>
In '1'
Close <number>
Open <title>
In 'Design'
Close <title>
Open <section>
Open <number>
In '1.1'
Close <number>
Introducing Object-oriented

Creating Python Classes
Objects

section

number: 2.1

title: Creating Python Classes
Design patterns - the decorator pattern

Purpose: to add extra things on top of the existing one.

Method: wrapping an object to make it behave differently.

Example: decorate a string.

```python
class WarningMessage:
    def __init__(self, s):
        self.s = s

    def __str__(self):
        return self.s.upper()
```
import random
class Hard2Read:
    def __init__(self, s):
        self.s = s

    def __str__(self):
        s = ''
        for c in self.s:
            s += c.upper() if random.sample([True, False], 1)[0] else c
        return s

# the interface is the same, __str__()
print('tsunami caused by collapse of volcano, experts confirm'.__str__())
print(WarningMessage('tsunami caused by collapse of volcano, experts confirm').__str__())
print(Hard2Read('tsunami caused by collapse of volcano, experts confirm').__str__())
print(WarningMessage('private property no trespassing').__str__())
print(Hard2Read('private property no trespassing').__str__())
We have produced *altered* strings by wrapping it in a class and rewriting its `__str__` method.

Gift wrapping (many options).

```python
def gift():
    return 'Coffee Mug'

def cushion(thickness):
    def decor(func):
```
def wrapper(*args):
    return thickness * 'Cushion + ' + func(*args)
return wrapper
return decor

def box(func):
    def wrapper(*args):
        return 'Box + ' + func(*args)
    return wrapper

def card(msg):
    def decor(func):
        def wrapper(*args):
            return 'Card [%s] + ' % msg + func(*args)
        return wrapper
    return decor

def ribbon(func):
def wrapper(*args):
    return 'Ribbon + ' + func(*args)
return wrapper

@ribbon
@card('Don’t worry. Be happy. — Bobby')
@box
@cushion(3)
def gift(name):
    return name

print(gift('Coffee mug'))
# Ribbon + Card [Don’t worry. Be happy. — Bobby]
# + Box + Cushion + Cushion + Cushion + Coffee mug
@ribbon
The interface is not changed. Recall that we learned property decorators before.

```python
def gift(name):
    return name

print(gift('Coffee mug'))  # Ribbon + Coffee mug
```

```python
import time

def log_calls(func):
    def wrapper(*args, **kwargs):
        now = time.time()
        print('DECOR Calling {0} with {1} and {2}'.
              format(func.__name__, args, kwargs))  # decor
        return_val = func(*args, **kwargs)
        print('DECOR Finished {0} in {1} ms\n'.
```
```python
format(func.__name__, time.time() - now)  # decor

return return_val
return wrapper

def test1(a, b, c):
    print('test1 called')

def test2(a, b):
    print('test2 called')

def test3(a, b):
    print('test3 called')
    time.sleep(1)

@log_calls
def test4(a, b):
    print('test4 called')
```
test1 = log_calls(test1)
test2 = log_calls(test2)
test3 = log_calls(test3)

test1(1,2,3)
test2(4,b=5)
test3(6,7)
test4(8,9)

#DECOR Calling test1 with (1, 2, 3) and {}
#test1 called
#DECOR Finished test1 in 0.0010001659393310547 ms

#DECOR Calling test2 with (4,) and {'b': 5}
#test2 called
#DECOR Finished test2 in 0.0 ms

#DECOR Calling test3 with (6, 7) and {}
#test3 called
#DECOR Finished test3 in 1.000569820404053 ms

#DECOR Calling test4 with (8, 9) and {}
#test4 called
#DECOR Finished test4 in 0.0 ms
Design patterns - the template pattern

The **Don’t Repeat Yourself** principle.

Useful in the situation where several tasks share a common subset of steps.

For example, query an SQLite database using different query constructions (statements).

Base class: a sequence of common steps.

Subclasses: overriding one or several of the above steps to provide customized behaviors.
Refer to the picture in Dusty’s book (page 325).

```python
import sqlite3

class QueryTemplate:
    def connect(self):
        self.conn = sqlite3.connect('sales.sqlite3')

    def construct_query(self):  # to be overridden
        raise NotImplementedError()

    def do_query(self):
        results = self.conn.execute(self.query)
        self.results = results.fetchall()

    def format_results(self):
        output = []
```
for row in self.results:
    row = [str(s) for s in row]
    output.append(','.join(row))
self.formatted_results = '\n'.join(output)

def output_results(self):  # to be overriden
    raise NotImplemented

def do_process(self):
    self.connect()  # shared among subclasses
    self.construct_query()
    self.do_query()  # shared
    self.format_results()  # shared
    self.output_results()

class NewVehicleQuery(QueryTemplate):
```python
def construct_query(self):
    self.query = 'select * from Sales where new="true"'  # a new attribute

def output_results(self):
    print(self.formatted_results)

import datetime
class UserGrossQuery(QueryTemplate):
    def construct_query(self):
        self.query = 'select salesperson, sum(amt) from Sales group by salesperson'

    def output_results(self):
        fname = 'gross_sales_{0}.txt'.format(datetime.date.today().strftime('%Y%m%d'))
        f = open(fname, 'w')
        f.write(self.formatted_results)
        f.close()
```

Copyright © 2018, 2019, 2020, 2021, 2022 Lan Hui
if __name__ == '__main__':
    # Create a database table and add a few records
    ## conn = sqlite3.connect('sales.sqlite3')
    ## conn.execute('CREATE TABLE IF NOT EXISTS Sales (salesperson text, amt currency, year integer, model text, new boolean)
    ## conn.execute('DELETE FROM Sales')
    ## conn.execute('INSERT OR REPLACE INTO Sales VALUES ("Tim", 16000, 2010, "Honda Fit", "true")')
    ## conn.execute('INSERT OR REPLACE INTO Sales VALUES ("Tim", 9000, 2006, "Ford Focus", "false")')
    ## conn.execute('INSERT OR REPLACE INTO Sales VALUES ("Gary", 8000, 2004, "Dodge Neon", "false")')
    ## conn.execute('INSERT OR REPLACE INTO Sales VALUES ("Gary", 28000, 2009, "Ford Mustang", "true")')
    ## conn.execute('INSERT OR REPLACE INTO Sales VALUES ("Don", 20000, 2010, "Toyota Prius", "false")')
    ## conn.commit()
    ## conn.close()

    # Create distinct query objects
    new_vehicle_query = NewVehicleQuery()
    new_vehicle_query.do_process()
    user_gross_query = UserGrossQuery()
In the two subclasses, we create `self.query`, which is not declared but assumed to be there in the superclass `QueryTemplate`.

We don’t have to declare all attributes in the `__init__` method. Instead, we can add attributes “on the fly” (in class methods or even after we’ve created an object from that class).

This design pattern can minimize change if we change to other SQL engines, such MySQL. We only need to change `QueryTemplate`, while leaving its subclasses not affected.
We can override `format_results` in `UserGrossQuery` if we want an HTML file instead of a text file.
Design patterns - the factory pattern

Make an object from the factory depending on a specific need.

Use cases:

- Country-specific date formatters (e.g., USA uses mm-dd-yyyy, while France uses dd/mm/yyyy)
class USAFormatterFactory:
    def create_date_formatter(self):
        return USADateFormatter()

class FranceFormatterFactory:
    def create_date_formatter(self):
        return FranceDateFormatter()

class USADateFormatter:
    def format_date(self, y, m, d):
        return '%02d-%02d-%04d' % (m, d, y)

class FranceDateFormatter:
    def format_date(self, y, m, d):
        return '%02d/%02d/%04d' % (d, m, y)
if __name__ == '__main__':
    factory = USAFormatterFactory()
    print(factory.create_date_formatter().format_date(2021, 11, 17))

    factory = FranceFormatterFactory()
    print(factory.create_date_formatter().format_date(2021, 11, 17))

    factory_map = {'US': USAFormatterFactory, 'France': FranceFormatterFactory}
    factory = factory_map.get('US')()
    print(factory.create_date_formatter().format_date(2021, 11, 17))

• Database backends (SQLite, MySQL, PostgreSQL, etc)

unit_of_work.py:
DEFAULT_SESSION_FACTORY = sessionmaker(
    bind=create_engine(
        config.get_postgres_uri(),
        isolation_level="REPEATABLE READ",
    )
)

class SqlAlchemyUnitOfWork(AbstractUnitOfWork):
    def __init__(self, session_factory=DEFAULT_SESSION_FACTORY):
        self.session_factory = session_factory

    def __enter__(self):
        self.session = self.session_factory()  # type:Session
        self.products = repository.SqlAlchemyRepository(self.session)
        return super().__enter__()

    def __exit__(self, *args):

super().__exit__(*args)
self.session.close()

def _commit(self):
    self.session.commit()

def rollback(self):
    self.session.rollback()

def session_factory(in_memory_db):
    return session_factory()

def postgres_session_factory(postgres_db):
    return postgres_session_factory()

def test_uow_can_retrieve_a_batch_and_allocate_to_it(self):
    session = session_factory()
test_uow.py:42:    uow = unit_of_work.SqlAlchemyUnitOfWork(session_factory)

test_uow.py:53:    def test_rolls_back_uncommitted_work_by_default(session_factory):

test_uow.py:54:    uow = unit_of_work.SqlAlchemyUnitOfWork(session_factory)

test_uow.py:58:    new_session = session_factory()

test_uow.py:63:    def test_rolls_back_on_error(session_factory):

test_uow.py:67:    uow = unit_of_work.SqlAlchemyUnitOfWork(session_factory)

test_uow.py:73:    new_session = session_factory()

test_uow.py:91:    def test_concurrent_updates_to_version_are_not_allowed(session_factory):

test_uow.py:93:        session = postgres_session_factory()
• cnblogs

• The Factory Method Pattern and Its Implementation in Python

• Card games
Responsibility-driven design

Nine principles in Craig Larman’s GRASP (General Responsibility Assignment Software Patterns).

These principles are a learning aid for object-oriented design with responsibilities.

Use methods to fulfill responsibilities.

• Information Expert.
PROBLEM: What is a basic principle by which to assign responsibilities to an object?
SOLUTION: Assign a responsibility to the class that has the information needed to respond to it.
Examples:

- Assign a *search* responsibility to a Catalog object, since the Catalog object has the information for all books in the library.
- Assign a *locate* responsibility to a LibraryItem object, since the LibraryItem object has the information for the DDS number for the book.
• Creator.

PROBLEM: Who creates an object A?
SOLUTION: Assign class B the responsibility to create an instance of class A if one of these is true:

– B “contains” or completely aggregates A
– B records A
– B closely uses A
– B has the initializing data for A

Example:
– Assign Catalog the responsibility to create LibraryItem objects, since Catalog “contains” library items.
– Assign Notebook the responsibility to create Note objects, since Notebook “contains” notes.

• Controller.

PROBLEM: What first object (beyond the UI layer) receives and coordinates a System Operation?
SOLUTION: Assign the responsibility to an object representing one of these choices:
– Represents the overall “system”, a root object.
– Represents a use case scenario within which the system operation occurs.

Example:

– Assign the responsibility to Notebook object. The UI layer is represented by class Menu (page 55 in Dusty Phillips book).

• Low Coupling

• High Cohesion

• Polymorphism.
PROBLEM: How to handle alternatives based on type?
SOLUTION: Giving the same name to similar or related services.
Example:

- LibraryItem is inherited by Book, CD, and DVD, each having the same service *locate*.

- Pure Fabrication.

PROBLEM: Sometimes assigning responsibilities only to domain layer software classes leads to problems like poor
cohesion or coupling, or low reuse potential.
SOLUTION: Assign a highly cohesive set of responsibilities to a convenience class that does not represent a domain concept.
Example:

– Make a fabricated class called Cup that holds the dice, roll them, and know their total. Cup can be reused in other applications.

• Indirection

• Protected Variations
Refactoring design

• Remember: objects can be conceptual. They do not have to be objects in real life.