Introduction to Object-oriented Analysis and Design

Course information (Fall 2018)

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. The students will learn the most useful tools in the object-oriented principles by practicing OO analysis, OO design and OO programming (using python 3) for an open publishing service. 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.

Paradigm: a pattern or model.

Recommended textbook: Dusty Phillips. Python 3 Object Oriented Programming. Second Edition.

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

Grading policy:

Component	**Weight**
Class participation	10
Labs	10
Midterm test (November 13 in class)	15
Project (due on December 6)	15
Final exam	50

Software freely available for this course:

- Python 3.7.0 interpreter:

https://www.python.org/downloads/release/python-370/

[scroll to bottom to find a installer for your operating system.]

- Wing IDE 101: http://wingware.com/downloads/wing-101

Procedural versus object-oriented

A wallet for money deposit and withdrawal.

Procedural:

```
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 %.0f RMB.' % wallet['money']
```

Object-oriented:

```python
class Wallet:

    def __init__(self, money):
        self.money = money

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

    def withdraw(self, amount):
        self.money -= amount
```

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

```python
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

Inheritance

UML

The object-oriented umbrella

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

In reality, this order is not strict. In fact, iteration is common.

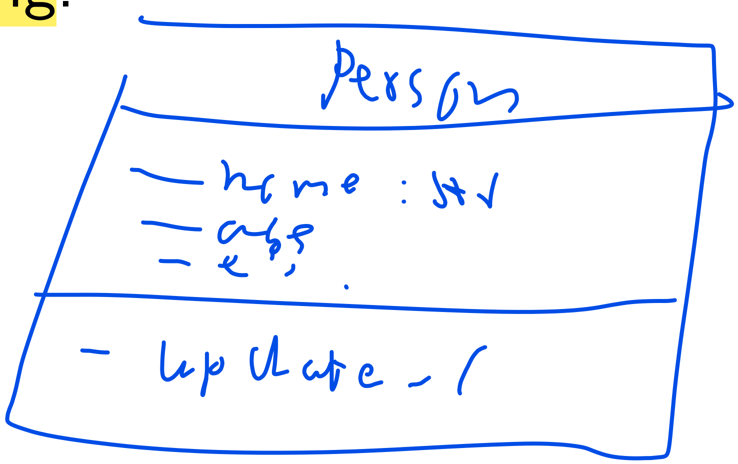
Object - a physical thing that we can sense, feel and manipulate (e.g., wallets, toys, babies, apples, oranges, etc).

In software development, an object is a collection of **data** and associated **behaviors**.

Oriented - directed toward.

Object-oriented - functionally directed toward modeling objects.

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



— data, attribute

behavior, method

Object-oriented analysis

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

The Software Requirements stage.

For example, an online food store:

- *review* our **history**
- *apply* for **jobs**

- *browse*, *compare*, and *order* **products**

Object-oriented design

Figure out *how* things should be done.

Convert requirements into implementation specification (classes and interfaces).

- Name the objects.
- Define the behaviors.
- Specify which objects interact with other objects.

Object-oriented programming

Convert design into a working program.

Murky real world

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.

Objects and classes

A class usually has **attributes** and **behaviors**.

Kind of objects is **class**.

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

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.

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__`.

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.

```
>>> 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.

Specifying attributes and behaviors

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.

Interacting objects

How do we make object interact? Pass objects as arguments to object methods.

```
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):
```

```
        return '%0.2f lbs orange from %s picked on %s' % \
               (self.weight, self.orchard, self.date)

    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
        self.oranges.append(item)

    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 = ''
```

```python
    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: 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 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

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

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

```python
class Man:
    def __init__(self, shoes):
        self.leg = Leg(120) # leg is instantiated inside class definit
        self.shoes = shoes # shoes is instantiated outside class defin


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, **object**.

Check the output of `help(object)` and `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 abstrac
```

```python
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 analyz
        print('Judiciously pick a peice 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.

```
class Contact:

    all_contacts = [] # class variable

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

```python
        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)
```

```python
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`.

```
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)
```

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

#for x in Contact.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`.

```
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
```

```python
                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

Polymorphism - many forms of (a function).

A fancy name.

Treat a class differently depending on which *subclass* is implemented.

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

Mostly talking about method overriding. A concept based on inheritance.

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

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



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

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


# Duck-typing
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

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

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 'f
```

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. 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
```

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):
```

```python
        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



b = B()
```

```
b.f() # Which version of f to use? L's f or R's f?
B.mro() # [<class '__main__.B'>, <class '__main__.L'>, <class '__main_
```

One potential consequence of the diamond problem is that the base class can be called twice. The following code demonstrates that.

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

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

class R(T):
```

```python
    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:
#Bottom
#Left
#Top
```

```
#Right
#Top
```

Therefore, to avoid that, we need **Method Resolution Order** (MRO) (with super()).

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

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



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
#Right
#Top
B.mro() # [<class '__main__.B'>, <class '__main__.L'>, <class '__main_
```

"next" method versus "parent" method.

Many expert programmers recommend against using it 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
```

```python
        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:\n %s' % 
              (self.email, message)) # email here is not a class attri
        # smtplib stuff


class EmailableContact(Contact, MailSender):
    pass
```

```
e = EmailableContact('John Smith', 'jsmith@gitee.com')
e.send_mail('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, cod
        Contact.__init__(self, name, email)  # cannot use super() here
        AddressHolder.__init__(self, street, city, province, code)
        self.phone = phone
```

UML - Unified Modeling Language

The UML diagram is a useful communication tool in OO analysis and design.

Caution: best used only when needed.

Reality: the initial diagrams become outdated very soon.

Some people think drawing UML class diagrams is a waste of time (if you spend too much time on it).

Most useful diagrams: class 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.

Hiding details and creating public interface

Determine the **public interface**. Make it stable.

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 functional details. 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 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 organised 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.

```
import database

db = database.Database()


from database import Database
db = Database()
```

```python
from database import Database as DB
db = DB()
```

- import

```python
import package.module  # use period operator to separate packages o

c = package.module.UserClass()
```

For example,

```python
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)
```

Organising modules to packages and properly importing them

```
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.

```
import ecommerce.payments.paypal
```

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

from ecommerce.payments.paypal import pay
pay()

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

main.py:

```python
from ecommerce.database import Database
from ecommerce.products import Product

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

from ecommerce.payments import paypal
```

```python
paypal.pay()
print(__name__)

import sys
print(sys.path)
```

ecommerce/products.py:

```python
from ecommerce.database import Database

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



if __name__ == '__main__':
```

```python
    p = Product('E45')
    print(p)
```

ecommerce/payments/paypal.py:

```python
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.

Organising module contents

A typical order in a Python module:

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


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

```python
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
        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()
```

```python
    return formatter.format(s)


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'))
```

```python
    print(format_string2('hello world'))
```

Public, protected and private attributes

```python
class Wallet:
    ''' For demostrating 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.

```
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 <
    #print(w.__gbp)
#builtins.AttributeError: 'Wallet' object has no attribute '__gbp'
```

```
#w
#<wallet.Wallet object at 0x0000000002A9D390>
#Wallet
#<class 'wallet.Wallet'>
#Wallet.__dict__
#mappingproxy({'__module__': 'wallet', '__doc__': ' For demostrating p
#w.__dict__
#{'rmb': 0, '_cad': 0, '_Wallet__gbp': 0}
```

Abstract methods and interfaces

No `interface` keyword in Python. We use ABCs (Abstract Base Classes) 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 class users and class implementers.

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.

```
# simplified from https://python-course.eu/python3_abstract_classes.ph
from abc import ABC, abstractmethod

class A(ABC):

    @abstractmethod
    def speak(self):
        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.

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



class Bird(ABC):
```

```
    @abstractmethod
    def fly(self):
        pass

    #@classmethod
    #def __subclasshook__(cls, C):
        #if cls is Bird:
            #if any("fly" in B.__dict__ for B in C.__mro__):
                #return True
        #return NotImplemented

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Bird:
            attrs = set(dir(C))
            #print(attrs)
            #print(set(cls.__abstractmethods__))
            if set(cls.__abstractmethods__) <= attrs:
```

```python
                return True
        return NotImplemented


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 f
```

@ is called 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?

```
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.

```
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)
```

```python
if __name__ == '__main__':
    L = EvenOnly()
    L.append(2) # OK
    L.append('2') # builtins.TypeError: Only integer can be added.
    L.append(3) # builtins.ValueError: Only even numbers can be added.
```

try-except:

```python
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).

```python
def no_return():
    print('1')
    raise Exception('Always raised.')
    print('2') # Warning: this code will never be reacched.
    return '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.
```

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)
```

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
```

```python
# test funny_division
print(funny_division(0))
print(funny_division(50.0))
#print(funny_division('0.0')) # this will raise builtins.TypeError: un

# test funny_division2
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:

```python
a = [1, 2, 3, '4']
b = 0
try:
    b = a[1] + a[3] # what will happen if use a[4] instead?
except IndexError:
    print('Cannot add due to Index Error')
except TypeError:
    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]
```

```python
try:
    choice = random.choice(exceptions)
    print('Raising {}'.format(choice))
    if choice:
        raise choice('An error')
except ValueError:
    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 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)
```

```python
except InvalidWithdrawal as e:
    print('Overdraft ${}'.format(e.overdraft()))
```

Getters, setters and @property decorator

Data encapsulation.

No change to client code. Backward compatible.

```
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 mm
```

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

    @height.setter
    def height(self, h):
        print('yyy')
        if h < 100:
            raise Exception('Too short!')
        elif h > 250:
            raise Exception('Too tall!')
        self._height = h



m = Man2(123)
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.

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

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

```
Man2.height #<property object at 0x0000000002A2FE58>
Man2.height.fget(m)
Man2.height.fset(m, 123)
Man2.height.__getattribute__('getter')
Man2.height.__getattribute__('settter')
```

More on decorators

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

```
def steamed_milk(func):  # INTERESTING - the argument is a function na
    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
```

```python
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))
```

```python
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

Standard solutions to design for frequently encountered problems.

Design patterns - the iterator pattern

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

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

In Python,

- We have a special method called `__next__`, accessible by `next(iterator)`.

- There is no `done` method. Raise exception `StopIetration` instead.

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

class CapitalIterator:
    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):
            raise StopIteration() # no done() method, raise StopIterat
        word = self.words[self.index]
        self.index += 1
        return word

    def __iter__(self):
```

```
        return self

class CapitalIterable: # we can get a iterator from it
    def __init__(self, s):
        self.s = s

    def __iter__(self):
        return CapitalIterator(self.s)


# iterable then iterator
iterable = 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(iterable)  # call __iter__

while True:
```

```python
    try:
        print(next(iterator)) # same as iterator.__next__()
    except StopIteration:
        break


for w in iterable:
    print(w)


for w in iterator:
    print(w)
```

Internally, a `for` loop is actually a `while` loop.

```python
iter([1,2,3])
```

```
<list_iterator object at 0x0000000002C40B00>

iter('123')
<str_iterator object at 0x0000000002C45E10>

iter({'a':1, 'b':2})
<dict_keyiterator object at 0x0000000002AD7098>
```

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 StopIetration, we have an infinite iterator.

```
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)



r = RandomIterator(56)
print(next(r))
print(next(r))
```

```
print(next(r))
# we have infinitely many of them
```

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.

Benefits: very concise.

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.

```
# 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 conta
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.ran
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_slst:
    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 f
fantasy_titles = {b.title:b  for b in books if b.genre == 'fantasy'}
print(fantasy_titles)
```

Lists, sets and dictionaries are called *containers*.

Generators

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

We can use a `for` loop.

We can also use a comprehension-like expression, called a generator.

```
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 generat
    for line in interesting_lines:
        print(line)
```

Use `yield` inside a function to make a generator.

```
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
```

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

What is the type of 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 `__next__` and `__iter__`), which will be exhausted after one pass.

What is going on inside looks like:

```python
import sys

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

```python
class IPFilter:
    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()
        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)
```

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

```
# See https://anandology.com/python-practice-book/iterators.html
def foo():
    print("BEGIN")
    for i in range(3):
        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/generat
    a = next(gen)
```

```
builtins.StopIteration:

'''
```

x is 0, y is 1 and z is 2.

Yield data from another generator using `yield from`.

```
def foo():
    for i in range(3):
        yield i

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:

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

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

```python
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.”

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
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!')


#g = echo()
#next(g)
#'nothing'
#g.send(1)
#I got 1.
```

```
#1
#g.send(2)
#I got 2.
#2
#g.close()
#Coroutine closed!
```

```
def tally():
    score = 0
    while True:
        incr = yield score # incr captures the sent value
        score += incr




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

```python
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
```

```python
import re

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]
```

```python
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

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
```

```python
    @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()

    def _update_observers(self):
        for o in self.observers:
            o()
```

```python
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')
        print(self.inventory.product)
        print(self.inventory.quantity)
```



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


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

i.product = 'E45'
i.quantity = 2
```

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. We don't have to mix code handling *multiple behaviors* code. Instead, we put such code in individual observers, facilitating maintainability.

Design patterns - the strategy pattern

Provide different solutions to a single problem, each in a different object.

```
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.s
        # 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
                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
        return out_img # call out_img.save('result.jpg') to save the f

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_
        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


strategy = TiledStrategy() # an strategy object
new_img = strategy.make_background('trump.jpg', (2400, 1200))
new_img.save('trump2.jpg')
```

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('trump.jpg', (2400, 1200))`).

Design patterns - the state pattern

See page 314 in the textbook.

The XML file `book.xml` is shown below.

```
<book>
    <author>Phillips</author>
    <publisher>PACKT</publisher>
    <title>OOP</title>
    <content>
        <chapter>
            <number>1</number>
            <title>Design</title>
            <section>
```

```
                <number>1.1</number>
                <title>Introducing Object-oriented</title>
            </section>
        </chapter>

        <chapter>
            <number>2</number>
            <title>Objects</title>
            <section>
                <number>2.1</number>
                <title>Creating Python Classes</title>
            </section>
        </chapter>

    </content>
</book>
```

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__() + '\n'
        for n in root.children:
            s += level * '.' +  _tree_structure(n, level+1)
        return s
    return _tree_structure(self.root)
```

```python
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('</'):
```

```
            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:]
```

```python
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()

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

```python
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
```

```
#content
#.chapter
#..number:1
#..title:Design
#..section
#...number:1.1
#...title:Introducing Object-oriented
#.chapter
#..number:2
#..title:Objects
#..section
#...number:2.1
#...title:Creating Python Classes
```

I have to say the above design is quite clever.

Parse `book.xml` with a generator.

```python
# Parse XML with a generator. Copyright (C) 2018 Hui Lan

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
    i = s.find('<')
    j = s.find('>')
    return s[i+1:j], j

def eat_close_tag(s):
    i = s.find('<')
    j = s.find('>')
    return s[i+2: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__() + '\n'
    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)
```

```
                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
```

```
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>
```

```
Open <title>
In 'Introducing Object-oriented'
Close <title>
Close <section>
Close <chapter>
Open <chapter>
Open <number>
In '2'
Close <number>
Open <title>
In 'Objects'
Close <title>
Open <section>
Open <number>
In '2.1'
Close <number>
Open <title>
In 'Creating Python Classes'
```

```
Close <title>
Close <section>
Close <chapter>
Close <content>
Close <book>
book
author:Phillips
publisher:PACKT
title:OOP
content
.chapter
..number:1
..title:Design
..section
...number:1.1
...title:Introducing Object-oriented
.chapter
..number:2
```

```
..title:Objects
..section
...number:2.1
...title:Creating Python Classes
```

Design patterns - the decorator pattern

Purpose: wrap an object to make it look different. In other words, add extra things.

Decorate a string.

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

    def __str__(self):
        return self.s.upper()
```

```python
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
        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 c
print(Hard2Read('tsunami caused by collapse of volcano, experts confir
print(WarningMessage('private property no trespassing').__str__())
print(Hard2Read('private property no trespassing').__str__())
```

```
#TSUNAMI CAUSED BY COLLAPSE OF VOLCANO, EXPERTS CONFIRM
#TsunaMi cauSed By colLAPse oF volcanO, ExPERts coNFirM
#PRIVATE PROPERTY NO TRESPASSING
#PRiVATE pRoPeRTy NO TrEspassIng
```

We have produced *altered* strings by wrapping it in a class and rewriting the `__str__` method.

Gift wrapping (many options).

```
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):
```

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

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

The interface is not changed. Recall that we learned property decorators before.

```
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'.\
              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 1.0000569820404053 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.

Several tasks share a common subset of steps. For example, query a 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.

The picture in page 325.


```python
import sqlite3

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

    def construct_query(self): # to be overriden
        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 NotImplementedError()

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

```
    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

    def output_results(self):
        fname = 'gross_sales_{0}.txt'.format(datetime.date.today().str
        f = open(fname, 'w')
        f.write(self.formatted_results)
        f.close()


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 ,
    conn.execute('DELETE FROM Sales')
    conn.execute('INSERT OR REPLACE INTO Sales VALUES ("Tim", 16000, 2
    conn.execute('INSERT OR REPLACE INTO Sales VALUES ("Tim",
9000, 2006, "Ford Focus", "false")')
    conn.execute('INSERT OR REPLACE INTO Sales VALUES ("Gary", 8000, 2
    conn.execute('INSERT OR REPLACE INTO Sales VALUES ("Gary",28000, 2
    conn.execute('INSERT OR REPLACE INTO Sales VALUES ("Don", 20000, 2
    conn.commit()
    conn.close()
    # Create distinct query objects
    new_vehicle_query = NewVehicleQuery()
    new_vehicle_query.do_process()
    user_gross_query = UserGrossQuery()
    user_gross_query.do_process()
```

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.

Test-driven development

It forces writing tests before writing code.

It helps understand requirements better.

A good reading: Yamaura, Tsuneo. "How to Design Practical Test Cases." IEEE Software (November/December 1998): 30-36.

It ensures that code continues working after we make changes.

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