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):
    wallet['money'] -= amount

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

Object-oriented:

```
class Wallet:

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

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

    def withdraw(self, amount):
        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

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.

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)
```

```
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):
        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 = ''
    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 <u>composed of</u> 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 **player**s - a player may be a human or a computer.

One **chess set** - a **board** with 64 **positions**, 32 **piece**s 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

```

```
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
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
        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`.

```
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
```

```
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
```

```
                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))

```

```
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))
```

```python
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):
        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):
    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
        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).

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()
```

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

- import

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

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)
```

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
```

```
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:

```
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:

```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.

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

Public, protected and private attributes

```
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:
                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).

```
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
```

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

# 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:

```
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)
```

```
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)
```

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

Getters, setters and @property decorator

Data encapsulation.

No change to client code. Backward compatible.

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

```
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
```