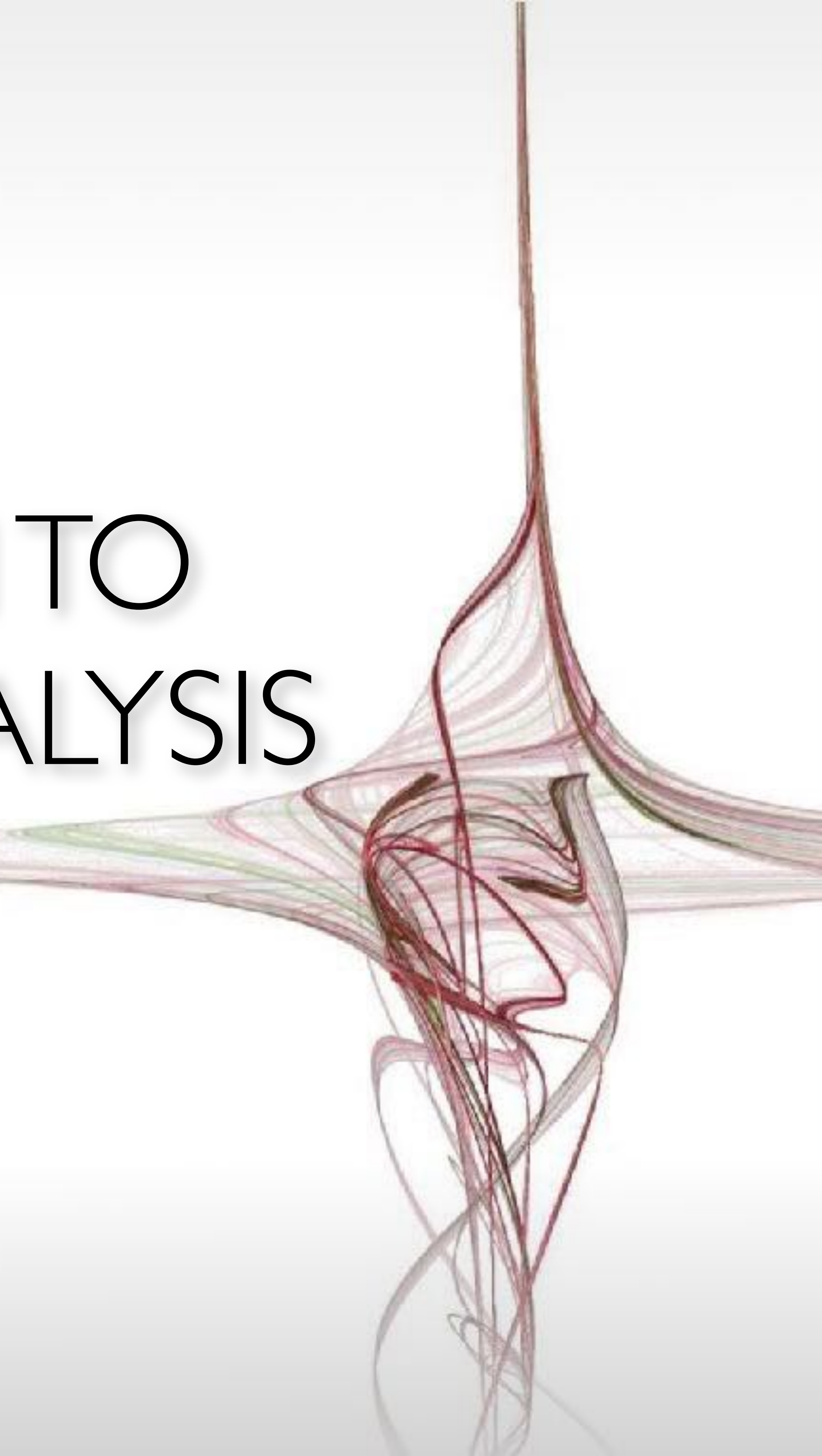
2020

INTRODUCTION TO NUMERICAL ANALYSIS



Lecture 1-5: I/O, exceptions and class

Kai-Feng Chen
National Taiwan University

INPUT & OUTPUT: SCREEN VERSION

- Printing to the screen — the simplest way to produce **output** is using the **print** statement. You are already doing that all the time!

```
>>> print('Hello World!')
Hello World!
```

- Reading keyboard Input — python provides the built-in function **input** to read a line of text from standard input:

```
>>> var = input('Your input: ')
Your input: Hello World!  ⇐ var contains the inputted string.
>>> var
'Hello World!'
```

OPENING AND CLOSING FILES

- One of the simplest ways for programs to maintain their data is by **reading and writing text files** (in permanent storage, e.g. hard drives).
- The **open()** function returns a file object, and is most commonly used with two arguments: **filename** and **mode**. The **close()** method of a file object flushes any unwritten information and closes the file object:

```
>>> fout = open('foo.txt', 'w')
>>> fout
<_io.TextIOWrapper name='foo.txt' mode='w' encoding='UTF-8'>
>>> fout.close()
>>> fout.closed
True
```

OPENING AND CLOSING FILES (II)

- The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc.
  - **r** : opens a file for reading only. This is the default file access mode.
  - **w** : opens a file for writing only; overwrites the file if the file exists.
  - **a** : opens a file for appending. If the file does not exist, it creates a new file for writing.
- One can also add a **b** (e.g. 'rb', 'wb') to open a file for reading/ writing in binary format.

READING AND WRITING FILES

- The file object provides a set of access methods.
- The **write()** method writes any string to an open file. It does not add a newline character ('\n') to the end of the string.

```
fout = open('foo.txt', 'w')
fout.write('Imagination is more important than
knowledge.\n---Albert Einstein\n')
fout.close()
```

You should get the following content in foo.txt:

```
Imagination is more important than knowledge.
---Albert Einstein
```

READING AND WRITING FILES (II)

- The **read()** method reads a string from an open file. It is important to note that Python strings can have binary data and not just text.
- For example:

```
fin = open('foo.txt')
str = fin.read()
print 'foo.txt: ',str
fin.close()
```

This is the output:

```
foo.txt:  Imagination is more important than
knowledge.
---Albert Einstein
```

READING AND WRITING FILES (III)

- The **read()** method, accept a parameter which is the number of bytes to be read from the opened file. If count is missing, then it tries to read as much as possible (until the end of file).

```
>>> fin = open('foo.txt')
>>> fin.read(30)
'Imagination is more important '
>>> fin.read(30)
'than knowledge.\n---Albert Eins'
>>> fin.read(30)
'tein\n'
>>> fin.read(30)
''
```

LOOP WITH FILE OBJECT

- It is quite common to read the file line-by-line and process the content with a loop. This can be carried out as following:

```
fin = open('foo.txt')
for l in fin:
    print('line:',l)
fin.close()
```

```
line: Imagination is more important than knowledge.\n
\n  ⇐ add by print statement                    come with "l" ⇑
line: ---Albert Einstein\n
\n
```

You may notice the new line character ('\n') is in the content of each line which has been read in.

LOOP WITH FILE OBJECT (II)

- Alternatively the **readline()** method will simply read a “line” back. A similar method named **readlines()** will read everything and pack them into a list of strings.

```
>>> fin = open('foo.txt')
>>> fin.readline()
'Imagination is more important than knowledge.\n'
>>> fin.close()
```

```
>>> fin = open('foo.txt')
>>> fin.readlines()
['Imagination is more important than knowledge.\n',
'---Albert Einstein\n']
>>> fin.close()
```

SEEKING THROUGH A FILE

- The **tell()** method tells you the current position within the file. The **seek()** method changes the current position within the file.

```
>>> fin = open('foo.txt')
>>> fin.read(30)
'Imagination is more important '
>>> fin.tell()
30
>>> fin.seek(10, 0) ⇐ move to position 10, starting from beginning
>>> fin.read(30)
'n is more important than knowl'
>>> fin.seek(0, 2) ⇐ move to the end of file
>>> fin.tell()
65
```

Remark: the second parameter of seek() can be **0** (beginning of file), **1** (current position), or **2** (end of file), but **1, 2** are not always working for text mode.

INTERMISSION

- We only tried to write a string into the file. How could we store a complex object like a list? e.g.

```
>>> l = [123, 2+5j, 3.14159, 'whatever']
```

What do you get if you do so?

```
>>> fout = open('foo.txt', 'w')
>>> fout.write(l)
```

Alternatively please try this:

```
>>> fout = open('foo.txt', 'w')
>>> fout.write(str(l))
```

INTERMISSION (II)

- You may find that you can finally get a file (foo.txt) with the following content:

```
[123, (2+5j), 3.14159, 'whatever']
```

- Try to read it back with the following commands:

```
>>> fin = open('foo.txt')
>>> tmp = fin.read()
>>> print(tmp)
```

Alternatively do this in addition to the lines above:

```
>>> l = eval(tmp)
>>> print(l)
```

WHEN SOMETHING GOES WRONG...

- If you write a code like this and run it:

```
fout = open('data.txt','w')
fout.write('ABCDEFG')
fout.close()

fin = open('date.txt') ⇐ an obvious typo here
tmp = fin.read()
fin.close()
```

iambuggy.py

```
FileNotFoundError: [Errno 2] No such file or
directory: 'date.txt'
```

You code will just stop and raise an **exception**.

WHEN SOMETHING GOES WRONG...(II)

- A lot of things can go wrong in your code, especially when you are accessing to files. For example:

```
>>> open('a_file_does_not_exist.txt')
FileNotFoundError: [Errno 2] No such file or
directory
>>> open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied
>>> open('/usr/bin')
IsADirectoryError: [Errno 21] Is a directory
```

Whenever you try to do something invalid, your program will stop immediately. Although this is not a critical issue (at least you will know what is wrong), but it makes your program "not-so-professional".

CATCHING EXCEPTIONS

- To avoid these errors, one can go ahead and try first — and deal with problems if they happen with a *special routine* — which is exactly what the **try** statement does.

```
try:
    fin = open('bad_file')
    for line in fin:
        print(line)
    fin.close()
except:
    print('Something went wrong.')
```

In this case, your program does not stop with the exception and continue to the **except** block.

CATCHING EXCEPTIONS (II)

- You can also attach the **else** block and **finally** block:
  - **else**: if there is no exception then execute this block.
  - **finally**: this would always be executed.

```
try:
    fin = open('foo.txt')
    for line in fin:
        print(line)
    fin.close()
except:
    print('Something went wrong.')
else:
    print('It is working well.')
finally:
    print('whatever, this block will be executed.')
```

CATCHING EXCEPTIONS (III)

- One can also separate different types of exceptions and execute different block of code:

```
try:
    print('Do something here...')
    # access to file, do some calculations, etc.
except OSError:
    print('There much be system-related error!')
except ValueError:
    print('The value must be wrong!')
else:
    print('Everything fine, move ahead!')
```

The list of standard exceptions can be found at https://docs.python.org/3/library/exceptions.html

RAISING AN EXCEPTION

- You can even raise an exception and send the code to run the exception block instead of the nominal path. This can be done by the **raise** statement.

```
n = int(input('Please enter an integer less than 10:'))

if n>=10:
    raise ValueError('invalid input!')
```

- In the above example, if you input a number greater than 10 the program will stop with a **ValueError** exception.
- Surely if you put the raise within the **try** statement, the code will jump to your predefined exception block.

INTERMISSION

- What will happen if the code generate an exception which is not in the exception block list, e.g.:

```
try:
    x = y = 1
    z = (x+y)/(x-y)
except OSError:
    print('There much be system-related error!')
else:
    print('Everything is fine!')
```

- Try to produce few different types of exceptions with some obvious "buggy" code.

CLASSES AND OBJECTS

- We have used many of Python's built-in types; now we are going to define a new type with the Python class extension.
- Defining a class is quite straightforward, for example:

```
class Point(object): ⇐ here object is the base class
    'Example point class for 2D space.' ⇐ doc string

    def __init__(self, x=0., y=0.): ⇐ constructor
        self.x,self.y = x,y
```

Now we get a new class named "Point".
By default it has two **attributes** x and y.

CLASSES AND OBJECTS (II)

- In the previous slide, a class named **Point** has been defined:

```
>>> class Point(object):
...     def __init__(self, x=0., y=0.):
...         self.x,self.y = x,y
...
>>> Point
<class '__main__.Point'> ⇐
```

Point is defined at the top level, its "full name" is **__main__.Point**.

- To create a **Point** object, you call Point() as if it were a function:

```
>>> p = Point()
>>> p
<__main__.Point object at 0x1005cfd10>
```

Creating a new object is called instantiation, and the object is an **instance** of the class.

ATTRIBUTES

- In the example point class, it has two default attributes of x,y.
- Values can be assigned to an **instance** using the **dot** notation:

```
>>> p.x = 2.0
>>> p.x, p.y
(2.0, 0.0)
```

- The assigned values are only valid within the assigned instance:

```
>>> q = Point()
>>> q.x, q.y
(0.0, 0.0)
```

ATTRIBUTES (II)

- Unlike C/C++, the Python class attributes can be actually added on-the-fly to the specific instance. One can start with an empty class and insert your data and form a “structure-like” object:

```
>>> class placeholder:
...     pass
...
>>> obj = placeholder()
>>> obj.pi = 3.14159
>>> obj.list = [1,2,3]
>>> obj.str = 'hello world!'
>>> print(obj.pi,obj.list,obj.str)
3.14159 [1, 2, 3] hello world!
```

INSTANTIATION

- Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named **__init__()**:

```
    def __init__(self, x=0., y=0.):
        self.x,self.y = x,y
```

Class instantiation automatically invokes __init__() for the newly created class instance. In the example this is valid:

```
>>> p = Point(3.,4.)
>>> p.x, p.y
(3.0, 4.0)
```

METHODS

- Usually a method (object) can be defined under the class block:

```
class Point(object):
    def rho(self):
        return (self.x**2+self.y**2)**0.5
```

A method can be called right after it is bound:

```
>>> p = Point(3.,4.)
>>> p.rho()
5.0
```

The first argument of a method is called <u>**self**</u>. The attributes stored in the instance can be accessed through it.

BASE METHODS OVERRIDING

- You can always override some of the default base (magic) methods, which can be very useful to build up your own object and interact with some other standard Python operations.
- All of these base methods are named similar to **__init__** (which is indeed one of the base methods in fact). A couple of examples:
  - **`__del__`**`(self)`: destructor; called when it is deleted.
  - **`__str__`**`(self)`: define how the object to string conversion.
  - **`__cmp__`**`(self,other)`: define how to compare two objects: returning negative value if self<other; returning positive value if self>other; zero if self == other.
  - **`__add__`**`(self,other)`: define how to add two objects, ie. the "+" operator.

BASE METHODS OVERRIDING (II)

- So if you define the class like this:

```
class Point(object):
    'Example point class for 2D space.'
    def __init__(self, x=0., y=0.):
        self.x,self.y = x,y
    def rho(self):
        return (self.x**2+self.y**2)**0.5
    def __str__(self):
        return '(x=%g,y=%g)' % (self.x,self.y)
    def __lt__(self,other):
       return self.rho()<other.rho()
    def __add__(self,other):
        return Point(self.x+other.x, self.y+other.y)
    def __mul__(self,other):
        return self.x*other.x+self.y*other.y
```

⇐ just compare the distance toward origin.

BASE METHODS OVERRIDING (III)

- Let’s try it out:

```
>>> p = Point(1.,2.)
>>> q = Point(2.,3.)
>>> print('p = '+str(p)+', q = '+str(q))
p = (x=1,y=2), q = (x=2,y=3)
>>> print('Is p closer to the origin than q?',p<q)
Is p closer to the origin than q? True
>>> str(p+q)
'(x=3,y=5)'
>>> p*q
8.0
```

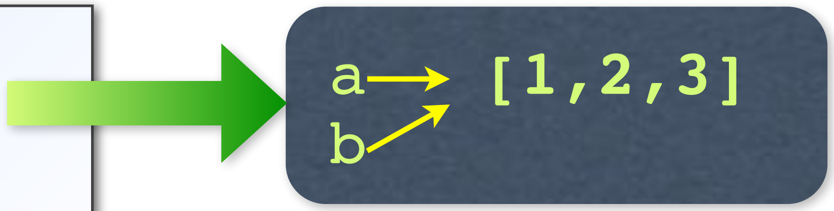
Surely you can add more and more “magic” methods to the class, and it will become very much similar to a regular built-in python type.

More information: https://www.python-course.eu/python3_magic_methods.php

ALIASING AND COPYING

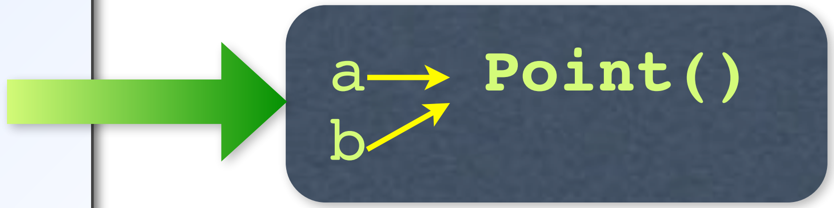
- Remember if you assign **b = a**, where **a** is a list. Both variables refer to the same object (aliasing):

```
>>> a = [1,2,3]
>>> b = a
>>> a is b
True
```



- This is also the behavior for your defined class, e.g.

```
>>> a = Point(3,4)
>>> b = a
>>> a is b
True
>>> a.x = 3.5
>>> b.x
3.5
```

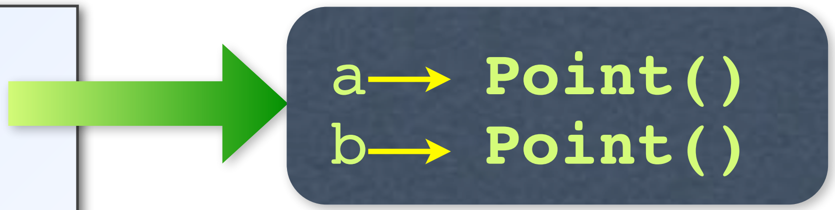


ALIASING AND COPYING (II)

- Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. Copying an object is often an alternative to aliasing.
- A quick solution is the **copy** module.

```
>>> import copy
>>> a = Point(3,4)
>>> b = copy.copy(a)
>>> a is b
False
>>> a.x = 3.5
>>> b.x
3
```

⇐ the **copy.copy()** function



ALIASING AND COPYING (III)

- However, there could be such a case — an object (such as list) is an attribute of your class. The **copy.copy()** function will not copy the underneath object but a reference. This is called a **shallow copy**.

```
>>> import copy
>>> class mydata:
...     def __init__(self,save):
...         self.data = save
...
>>> a = mydata([1,2,3])
>>> b = copy.copy(a)
>>> b is a   ⇐ although b is not an alias of a
False
>>> b.data is a.data   ⇐ but b.data is still a reference of a.data
True
```

ALIASING AND COPYING (IV)

- The solution is the **deep copy** with the **copy.deepcopy()** function. It copies not only the object but also the objects it refers to, and the objects they refer to, and so on.

```
>>> a = mydata([1,2,3])
>>> c = copy.deepcopy(a)
>>> c is a
False
>>> c.data is a.data   ⇐ now c.data is a full copy of a.data, not
False                     a reference anymore.
```

Remark: the copy module works well with regular type like list!

CLASS INHERITANCE

- Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.
- The child class **inherits** the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also **override** data members and methods from the parent.

```
class Point(object):
    'Example point class for 2D space.'
```

In the previous example, **object** is the base (parent) class.
Class object is the most base type in Python.

CLASS INHERITANCE (II)

- An example of inheritance:

```
class counting:
    def __init__(self, init_value=0):
        self.counter = init_value
    def add_a_count(self):
        self.counter +=1
        print('counter+1:',self.counter)
class double_counting(counting):
    def add_two_counts(self):
        self.counter +=2
        print('counter+2:',self.counter)
dc = double_counting(10)
dc.add_a_count()
dc.add_two_counts()
```

```
counter+1:  11 ⇐ method in counting
counter+2:  13 ⇐ method in double_counting
```

screen printout

INTERMISSION

- Try to "observe" how many base (magic) methods have to be overwritten for a full-functioning type. You can try this to see how a Python complex number class working:

```
>>> help(complex)
```

- The copy module works with built-in object as well. Try to make a copy of a list of list, and see if sub-lists are real copies or just a reference.

```
>>> l = [[1,2,3],[4,5,6]]
```

HANDS-ON SESSION

- **Practice 1 (a):**
  Write a small program with **input()** function to fill a small form like this:

```
first name = Kai-Feng
last name = Chen
phone number = 33665153
address = R529, Department of Physics, NTU
```

Store those information into 4 variables:
`first_name, last_name, phone_number, address`
and write them to a file with the same format shown above.

HANDS-ON SESSION

- **Practice 1 (b):**
  Instead of **input()**, read the data back from the file and parse the file line-by-line. Store those information back and store them as a list of list, e.g.

```
[['last name', 'Chen'], ['first name', 'Kai-
Feng'], ['phone number', '33665153'],
['address', 'R529, Department of Physics,
NTU']]
```

HANDS-ON SESSION

- **Practice 2 (a):**
  A “cash” class is implemented as below, finish the method **convert()** to handle the correct TWD ⇔ EUR converting.

```
class cash(object):
    'An example class to handle cash in different currency'

    def __init__(self, amount = 0., currency = 'TWD'):
        self.amount = amount
        self.currency = currency

    def __str__(self):
        return str(self.amount)+' '+self.currency

    def convert(self, target_currency = 'EUR'):
        # converting from self.currency to target_currency
        pass

my_bill = cash(1000.0,'TWD')
print('>>> My bill shows',my_bill)

my_bill.convert('EUR')
print('>>> After converting to EUR, my bill shows',my_bill)

my_bill.convert('TWD')
print('>>> After converting to TWD, my bill shows',my_bill)
```

implement it!
⇓ Just take 1 EUR = 36.4 TWD

HANDS-ON SESSION

- **Practice 2 (b):**
  Implement a magic method **__add__()** that allows you to ADD two "cash" classes if their currencies are the same:

```
class cash(object):
    'An example class to handle cash in different currency'

    def __init__(self, amount = 0., currency = 'TWD'):
        self.amount = amount
        self.currency = currency
 . . . . . .                                          ⇓ implement it!
    def __add__(self,other):
    # add two cash class and return the sum
        return cash(0.)

my_bill_1 = cash(100.0,'TWD')
my_bill_2 = cash(500.0,'TWD')
print('>>> My bills (1+2) in total:',my_bill_1+my_bill_2)

my_bill_3 = cash(50.0,'EUR')
my_bill_4 = cash(20.0,'EUR')
print('>>> My bills (3+4) in total:',my_bill_3+my_bill_4)
```

HANDS-ON SESSION

Remark: you can get the template code for this practice 2 from CEIBA

- **Practice 2 (c):**
  Improve your magic method **__add__()** that allows you to ADD two "cash" classes even if their currencies are *different*:

```
class cash(object):
    'An example class to handle cash in different currency'

    def __init__(self, amount = 0., currency = 'TWD'):
        self.amount = amount
        self.currency = currency
 . . . . . .
    def __add__(self,other):
    # add two cash class and return the sum
        return cash(0.)

my_bill_1 = cash(100.0,'TWD')
my_bill_2 = cash(500.0,'TWD')

my_bill_3 = cash(50.0,'EUR')
my_bill_4 = cash(20.0,'EUR')

print('>>> My bills (1+4) in total:',my_bill_1+my_bill_4)
print('>>> My bills (3+2) in total:',my_bill_3+my_bill_2)
```

improve it! take the target ⇓ currency from 'self'