Python

PartA 645533

A1. Python vs cpp

`not`	instead of	`!`
`and`	instead of	`&&`
`or`	instead of	`||`
`**`	instead of	`pow`
`True`	instead of	`true`
`False`	instead of	`false`

A2. if/for

- Iterable can be `range(len(collection))`, `list`, `tuple`, `string`, `set` or `dict`.
- `for else` is invoked if all loops are done without breaking.
- no more `{}` no more `;`
- start with `:` share same indentation

```
if x in mylist : action0
elif condition : action1
else : action2

for x in iterable :
    if condition : action0
    else : break
else : action1
```

A3. Function

Argument can be initialized as positional argument, default argument or keyword argument, position argument is disallowed after keyword argument. [1][2][3]

```
def fct(x, y = default_y, z = default_z) :
    fct_content ...

fct(1,2,3)        # positional argument
fct(1,2)          # default argument for z
fct(1, z = 2)     # keyword argument for z
```

Variadic argument with or without keywords are indicated by single `*` or double `**`. [4][5]

```
def fct0(arg, *args) :
    do_something_with(arg)
    for argument in args : do_something_with(argument)

def fct1(arg, **args) :
    u,v,s = 0,1,0
    for keyword in args : # keywords are considered as strings here
        if   keyword == 'mean' : u = args[keyword]
        elif keyword == 'var'  : v = args[keyword]
        elif keyword == 'skew' : s = args[keyword]
        else : print('unknown keyword')

fct0(x, x0, x1, x2, x3)
fct1(x, mean=x0, var=x1, skew=x2, kurtosis=x3) # keywords are NOT strings here
```

A4. Lambda function and map/filter/reduce/partial

- In cpp terms, `lambda` is functor while `map` is `std:transform`

```
f = lambda x,y,z : x**2 + y**2 + z**2
f(1,2,3)
f(a,b,c)

sum(map(lambda x : not x%k, array)) # count number of element divisible by k
sum(matrix)      # sum all rows, becoming single row
map(sum, matrix) # sum all cols, becoming single col
```

- `map` takes a `T->U` functor, it converts a list of `T` to a list of `U`.
- `filter` takes a `T->bool` functor, it converts a list of `T` to a reduced list of `T`.
- `reduce` takes a `(T,T)->T` functor, it converts a list of `T` to a single `T`.
- `partial` works like `std::bind` in C++

```
def f(x,y,z) : return x+y+z
g = partial(f,1,2)
g(3)
```

A5. Scope concept

We can access global scope variable inside function body, it is hidden by local scope variable having same name.

```
def fct() :
    x.append(4)
    print(x)


x = [1,2,3]
print(x) # print [1,2,3]
fct()    # print [1,2,3,4]
print(x) # print [1,2,3,4]
```

```
def fct() :
    x=[7,6,5]
    x.append(4)
    print(x)


x = [1,2,3]
print(x) # print [1,2,3]
fct()    # print [7,6,5,4]
print(x) # print [1,2,3]
```

```
def fct() :
    global x # nonlocal for nested-fct
    x=[7,6,5]
    x.append(4)
    print(x)


x = [1,2,3]
print(x) # print [1,2,3]
fct()    # print [7,6,5,4]
print(x) # print [7,6,5,4]
```

A6. Share pointer concept

In C++ terms :

- variables in python are share pointers, they are all lvalue
- objects in python are constructed with factories, they are all rvalue
- objects in python are destructed using `del`

```
variable0 = class(x,y,z)
variable1 = fct_return_obj()
del(variable0)
```

B1. Collection

`range`	this is not a list	x	
`list`	constructed with `[]`	9/8/5	
`tuple`	constructed with `()`, it is a constant list	x	empty tuple `()`, tuple with 1 int `(123,)`, `(123)` is regarded as int
`string`	constructed with `''`, it is a character tuple	6	
`set`	constructed with `set()` or `{'a','b','c'}`	2/4	key must be hashable, no set of list, but set of tuple
`dict`	constructed with `{}` or `{'a':x, 'b':y, 'c':z}`	2/4	key must be hashable …

`[](){}` means capture list, argument list and function body in C++ lambda
`[](){}` means character set, higher priority and number of occurrence in regex
`[](){}` means list, tuple and dict (or set) in Python

	modify current list	construct new list	iteration
`list`	`len(list)`	`x = []`	`if x in list0 :`
	`clear()`	`x = y[:]`	`for x in list0 :`
	`append(value)`	`x = y[m:]`	`for n in range(len(list0)) :`
	`extend(list_rhs)`	`x = y[:n]`	`for n,x in enumerate(list0) :`
	`insert(index, value)`	`x = y[m:n]`	`for x,y in zip(list0, list1) :`
	`remove(value)`	`x = [1,2,3]*10`	
	`pop(index)`	`x = [1,2,3]+[4,5,6]`	
	`sort()`	`x,y = y,x`	
	`reverse()`		
`string`		`s = 'a'*10`	
		`s = s0[0:3]+'_'+s0[4:]`	
		`s = '_'.join(words)`	
		`words = s.split()`	
		`s.upper()`	
		`s.replace('+','-')`	
`set`	`s.add(value)`	`s = s0 | s1`	
	`s.remove(value)`	`s = s0 & s1`	
		`s = s0 ^ s1` XOR	
		`s = s0 - s1`	
`map`	`m[key] = value`		`m.keys()`
	`m.pop(key)`		`m.values()`
			`for key in map : m[key] ...`
			`for key,value in map.items() :`

B2. Comprehension

Given a list, count the number of element equals to targer :

```
count = len([x for x in list0 if x == target])
```

Given a list, generate a list consisted of all possible pair-sum :

```
x = [array[i]+array[j] for i in range(len(array)) for j in range(i+1, len(array))]
```

Given a distance matrix, generate a submatrix if the points are clustered into two complement sets, one of which is `combo`

```
sub_dist = [[dist[y][x] for x in set(range(len(dist)))-set(combo)] for y in combo]
```

C1. Class

Here is the basic of a class. In fact `variable` is a shared pointer, pointing to anonymous object created by constructor or factory.

	define	invoked inside class	invoked outside class
constructor	`def __init__(self,x,y,z)`		`variable = classname(x,y,z)`
destructor	`def __del__(self)`		`del(variable)`
instance method	`def fct(self,x,y,z)`	`self.fct(x,y,z)`	`variable.fct(x,y,z)`
instance variable	init in instance method	`self.mem = ...`	`variable.mem = ...`
class method	`def fct(x,y,z)`	`classname.fct(x,y,z)`	`classname.fct(x,y,z)`
class variable	init in class body	`classname.mem = ...`	`classname.mem = ...`

- attributes are default to be public and virtual
- attributes can be made private by adding double underscore in their names
- class attributes can be listed by `classname.__dict__`
- instance attributes can be listed by `variable.__dict__`
- attribute pointer can be used like this :

```
mem_ptr = variable.fct
mem_ptr(x,y,z)
```

C2. Iterable class / Generator / Coroutine

We can make a class

- iterable by providing instance function `__iter__(self)` which returns an iterator object
- iterator is object providing instance function `__next__(self)` which jumps to next position and return current value
- in cpp, we usually offer friendship to iterator to access container, in python, we directly combine them into one class

```
class iterable_fibonacci :
    def __init__(self) :
        self.index = 0
        self.array = [None] * 50
        self.array[0] = 1
        self.array[1] = 1
        for n in range(2, 50) : self.array[n] = self.array[n-1] + self.array[n-2]

    def __iter__(self) :
        return self

    def __next__(self) :
        if self.index < len(self.array) :
            out = self.array[self.index]
            self.index = self.index + 1
            return out;
        else : raise StopIteration

array = iterable_fibonacci();
for x in array : print(x)
```

C3. Inheritance

Multi inheritance is common :

```
class my_class(base0, base1, base2) :
```

D1. Importing modules
There are several ways :

	latter invocations
`import numpy`	`numpy.fct(x,y,z)`
`import numpy as np`	`np.fct(x,y,z)`
`from numpy import fct0,fct1`	`fct0(x,y,z)`
`from numpy import fct as f`	`f(x,y,z)`

Main function and getting arguments

```
import sys
if __name__ == “main” :

    x = int(sys.argv[1])
    y = int(sys.argv[2])
    z = int(sys.argv[3])
```

D2. Debugger
Python debugger is also a module. It can be started by `python -m`, where option `m` means module.

```
>> python -m pdb my_code.py
(pdb) s     // start program
(pdb) n     // next step
(pdb) n     // next step
(pdb) s     // step inside
(pdb) p x   // print variable x
```

When running python code in linux with error `"bad interpreter /usr/bin/python^M"`, the solution is to run the following command in `vim`, which change current python code into unix format.

```
:set fileformat=unix
```

Intermediate level topics

- object introspection
- mutation
- decorators
- lambdas
- map, filter and reduce
- generators
- coroutines
- collections
- comprehensions
- enumerate
- zip and unzip