Typical Python Questions

<font color = "red">I listed the necessary Python questions that people need to know for a job require Python. All of these appear many times when I am looking for a job, and people seldom ask other questions than these. There are some errors there in the code, I meant to do it to show you the possible errors. Some people ask pandas, numpy and scipy packages. I didn't cover these. But they just expect you have used them. So if you have time, just read some introduction and practice a little bit. Otherwise, this notebook is very helpful for Python interview.

1. List comprehension

It's not a question though, but people tend to know whether you are proficient about Python. And this is the special point about Python. They like to ask whether you have used this. I will give the example.

In [28]:

```
#Example
if __name__=='__main__':
    eg_list = [x**2 for x in range(10)] #This is list comprehension, Python guys
 like to use this to make code look niceer
    eg_dict = {x:y for x, y in zip(range(10), range(10, 20))} #This is for dicti
onary, but they are the same
    print(eg_list)
    print(eg_dict)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
{0: 10, 1: 11, 2: 12, 3: 13, 4: 14, 5: 15, 6: 16, 7: 17, 8: 18, 9: 1
9}
```

2.Python class implementaion

2.1 No private stuff

Python doesn't has private method or member, technically. People have convention to use prefix '_' to indicate this is private and prevent you to call it. But technically, you can do it.

2.2 Different constructor mechanism

Derived class constructors don't call base class constructors automatically. You need to use 'super' to call it

2.3 Always inherite class from 'object'

In python3, it doesn't make any difference at all whether inherite from 'object' or not. In python2, class inherited from object is new-style, otherwise, it's called old-style. They have difference in multi-inheritance. That's too deep. No one ever ask this. But they might be interested whether you know the story.

In [29]:

```
# Example
class A(object):
    def __init__(self, x0):
        self.x = x0

class B(A):
    def __init__(self, y0):
        self.y = y0

class C(A):
    def __init__(self, y0):
        self.y = y0

        #in Python 3, you just need to do 'super().__init__'
        super(C, self).__init__(10)

if __name__=='__main__':
    a = B(1)
    #this cause error, since B didn't use super
    print(a.x)
```

```
---------------------------------------------------------------------
-------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-29-92d7983a54e8> in <module>()
     18     a = B(1)
     19     #this cause error, since B didn't use super
---> 20     print(a.x)

AttributeError: 'B' object has no attribute 'x'
```

In [30]:

```
# this is fine, since C called super().__init__
if __name__=='__main__':
    b = C(20)
    print(b.x)
```

```
10
```

3. mutable and immutable types

This is a big part. And people from C++ cannot understand it initially.

In Python, there are two types of objects. One is immutable type like int, double , string and tuple. Dictionary and List are mutable types.

3.a Mutable doesn't not mean you can change parameter value through assignment inside a function

In [31]:

```
#exmaple
def foo(x):
    x = [2,3]
def bar(x):
    x.append(20)

if __name__=='__main__':
    y = [10]

    #y won't change
    foo(y)
    print(y)

    #append will make y change
    bar(y)
    print(y)
```

```
[10]
[10, 20]
```

This is very different from C++. Here is my understanding, mutable type only mean you can modify the object through object member function like 'append' and '[]'. But assignment is different, it's not value assignment. Assignment in Python means 'copy reference of right side'. in function foo(), x copies the reference of '[2,3]'. x nolonger point to outside y anymore. That's why no affection for y in foo()

3.b Requirement for hash table (dictionary) keys

You need to have immutable type for keys. The reason is when you construct a dictionary, you need a hash function to create a hash for the keys. If the keys can be modified, then the hash will be different.

In [32]:

```
if __name__=='__main__':
    x = {}
    y = [12]

    #This will cause error, list is not hashable
    x[y] = 1
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-32-efef5197ce2e> in <module>()
      4
      5     #This will cause error, list is not hashable
----> 6     x[y] = 1

TypeError: unhashable type: 'list'
```

3.c Shallow Copy Case Problem

It will become tricky when you do list multiplication like the following. Shallow copy will make you change the value for each item you copied. But remeber assignent won't work

In [33]:

```
if __name__=='__main__':

    #This is a shallow copy for [1,2 ]
    x =[[1,2]]*3
    print(x)

    #we only append 3 for the first one, but actually it append 3 for every list
    x[0].append(3)
    print(x)

    #It works for append, but doesn't work for assignment, since we are assignin
g a new reference not changing old reference
    x[0]=[100]
    print(x)

    #It also not works for int, since int cannot modify except for assigning a n
ew number reference
    y = [1]*4
    y[0] = 10
    print(y)
```

```
[[1, 2], [1, 2], [1, 2]]
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
[[100], [1, 2, 3], [1, 2, 3]]
[10, 1, 1, 1]
```

3.d Default value with mutable type for parameters will become problem

In Python, '[]', 1,2 are not values. They are objects. When you make x=1, you are not assign 1 to x, you just copy 1's reference to x. Read the following example.

In [34]:

```
def foo_params(y, x = []):
    #[] represent an object, everytime you call this function you are assgining
 the same object reference to x
    #So next time you run it, it keeps the old stuffs
    x.append(y)
    print(x)

if __name__=='__main__':
    for i in range(10):
        foo_params(i)
```

```
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4. Generator

Generator is built to save memory and time when you loop through a very long container. It will only read current item one time instead of reading the whole container.

There are several way to implement generator, take a look at the following example.

In [35]:

```
#Use function to declare generator
def foo_generator():
    for i in range(10):
        yield i

if __name__=='__main__':
    x = foo_generator()
    print(next(x))
    print(next(x))

    print('new start:')
    for y in foo_generator():
        print(y)
```

```
0
1
new start:
0
1
2
3
4
5
6
7
8
9
```

In [36]:

```
#use list comprehension
if __name__=='__main__':
    #use '()' to declare generator
    x = (i for i in range(10))
    print(type(x))
```

```
<type 'generator'>
```

In [37]:

```
#define a geneator class
class A(object):
    def __init__(self, n):
        self.i = 0
        self.x = list(range(n))

    def __iter__(self):
        return self

    #in python 3 this function is enough
    def __next__(self):
        if self.i < len(self.x):
            self.i += 1
            return self.x[self.i-1]
        else:
            raise StopIteration()

    #This is specific for python 2, in python 3, we don't need this
    def next(self):
        return self.__next__()

if __name__=='__main__':
    a = A(10)
    print(next(a))
    for i in A(20):
        print(i)
```

```
0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

5. Decorator

Decorator is a function a decorate another function. Decorator takes function object as parameters and makes modification about it and return new function object. Check the example.

In [38]:

```
def my_decorator(some_function):

    def wrapper():

        print("Something is happening before some_function() is called.")

        some_function()

        print("Something is happening after some_function() is called.")

    return wrapper

@my_decorator
def foo():
    print("Hello world!")

if __name__=='__main__':
    foo()
```

```
Something is happening before some_function() is called.
Hello world!
Something is happening after some_function() is called.
```

In [ ]: