17-21 November 2016

Md Sorwar Alaam

PyCamp V1

Question Answer

esorwar.cse@gmail.com

**Question 1:** In Python, what does GIL mean? Why does it matter? Are there any Python implementations which are not affected by GIL? If Yes, please mention at least two. I0f No, please explain why is it so hard to get rid of GIL

**Answer:**

Python's GIL is intended to serialize access to interpreter internals from different threads. On multi-core systems, it means that multiple threads can't effectively make use of multiple cores. (If the GIL didn't lead to this problem, most people wouldn't care about the GIL - it's only being raised as an issue because of the increasing prevalence of multi-core systems

**Question 2:** What is the difference between xrange and range in Python 2.x? Why does the difference matter? How has it changed in Python 3? How to emulate the Python 2.x behaviour on Python 3?

**Answer:**

[**Python**](https://code-maven.com/python) has a built-in function called range that can easily generate a a range of whole numbers. There is another built-in function call xrange that provides the same result, but uses a lot less memory.

## **Memory Size**

The big difference is in the amount of memory they use:

1. #!/usr/bin/env python
2. from \_\_future\_\_ import print\_function
3. import sys
5. r = range(10000)
6. print(sys.getsizeof(r)) # 80072
8. x = xrange(10000)
9. print(sys.getsizeof(x)) # 40

The variable holding the range created by range uses 80072 bytes while the variable created by xrange only uses 40 bytes.

The reason is that range creates a list holding all the values while xrange creates an object that can iterate over the numbers on demand.

**Question 3:** How do you create a dictionary that maintains the order of the keys?

**Answer:**

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

mydict = {'carl':40, 'alan':2,'bob':1,'danny':3 }

**for** key **in** sorted(mydict.iterkeys()):

**print** "**%s**: **%s**" % (key, mydict[key])

**Output:**

alan: 2

bob: 1

carl: 40

danny: 3

**Question 3:** Please take a look at the following codes and explain what’s happening here:

>>> a = 256

>>> b = 256

>>> a is b

True

>>> c = 257

>>> d = 257

>>> c is d

False

>>> e = 258; f=258;

>>> e is f

True

>>>

**Answer:**

>>> a = 256

>>> b = 256

>>> id(a)

9987148

>>> id(b)

9987148

>>> a = 257

>>> b = 257

>>> id(a)

11662816

>>> id(b)

11662828

Show the code above when a=256 and b=256 then every id is same but a=257 and b=257 then cannot id same so it is print False. **is** meaning True if the operands are identical (refer to the same object)

The current implementation keeps an array of integer objects for all integers between -5 and 256, when you create an int in that range you actually just get back a reference to the existing object. So it should be possible to change the value of 1. I suspect the behavior of Python in this case is undefined.

**Question 4:** What is the difference between lists and tuples? Why should we use one over the other?

**Answer:**

Tuples are lists which can't be edited. Once you create a tuple, you cannot edit it, it is immutable. Lists on the other hand are mutable, you can edit them, they work like the array object in JavaScript or PHP. You can add items, delete items from a list; but you can't do that to a tuple, tuples have a fixed size.

**Here is an example demonstrating the mutable nature of lists in Python.**

animals = ['cat', 'dog']  
>>> animals  
['cat', 'dog']  
>>> animals.append('mat')  
>>> animals  
['cat', 'dog', 'mat']  
>>> animals[2] = 'bat'  
>>> animals  
['cat', 'dog', 'bat']

**An example showing the immutable nature of tuples in Python.**

>>> point = (3,7)  
>>> point  
(3, 7)  
>>> point[1]  
7  
>>> point[1] = 4  
Traceback (most recent call last):  
 File "", line 1, in   
TypeError: 'tuple' object does not support item assignment

For example, when we do something like this:

>>> x,y=1,2  
>>> x  
1  
>>>y  
2

that's all tuples in the background doing the work. What's happening above is called "tuple assignment" because Python first constructs the tuple (1, 2), and then pairwise assigns each value to its corresponding variable.

Python also uses tuples whenever we want to return multiple values from a function:

>>> def swap(a,b):  
>>> t=a

>>> a=b

>>> b=t

>>> return (a,b)  
>>>swap(20,10)

(10,20)

So tuples can be used by Python to quickly stuff multiple things into a "single" container.

**Question 5:** Write codes to demonstrate how decorators work in Python.

**Answer:**

A decorator is the name used for a software design pattern. Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated.

A decorator that accomplishes the same result and is also reusable and composable looks like this:

>>> def my\_decorator(func):

... def decorate():

... print("--------------")

... func()

... print("--------------")

... return decorate

...

>>> def print\_raw():

... print("Clear Text")

...

>>> decorated\_function = my\_decorator(print\_raw)

>>> decorated\_function()

--------------

Clear Text

--------------

>>> print\_raw = my\_decorator(print\_raw)

>>> print\_raw()

--------------

Clear Text

--------------

## **@decorator**

when define decorator function then we just call the function then output function

@my\_decorator

def print\_text():

print("Hello World!")

--------------

Hello World!

--------------

@my\_decorator

def print\_number():

for i in range(5):

print i

--------------

0

1

2

3

4

--------------

**Question 6:** Explain what is the function of generators. Write codes to show us an example.

**Ans:**

There was a lot of overhead in building an iterator in Python; we had to implement a class with**\_\_iter\_\_()** and **\_\_next\_\_()** method, keep track of internal states, raise **StopIteration** when there was no values to be returned etc. This is both lengthy and counter intuitive. Generator comes into rescue in such situations.

It is fairly simple to create a generator in Python. It is as easy as defining a normal function with**yield** statement instead of a **return** statement. If a function contains at least one **yield**statement (it may contain other **yield** or **return** statements), it becomes a generator function. Both **yield** and **return** will return some value from a function. The difference is that, while a**return** statement terminates a function entirely, **yield** statement pauses the function saving all its states and later continues from there on successive calls. Here is how a generator function differs from a normal function.

def my\_gen():

n=1

print(“This is printed first”)

#Generator function contains yield statements

yield n

**Python Generator Expression**

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy

my\_list = [1, 3, 6, 10]

>>> # square each term using list comprehension

>>> [x\*\*2 for x in my\_list]

[1, 9, 36, 100]

>>> # same thing can be done using generator expression

>>> (x\*\*2 for x in my\_list)

<generator object <genexpr> at 0x0000000002EBDAF8>

We can see above that the generator expression did not produce the required result immediately. Instead, it returned a generator object with produces items on demand.

>>> a = (x\*\*2 for x in my\_list)

>>> next(a)

1

>>> next(a)

9

>>> next(a)

36

**Question 7:** What is asyncio? How can it be compared to Node.JS?

**Answer:**

The [asyncio](https://pymotw.com/3/asyncio/" \l "module-asyncio" \o "asyncio: Asynchronous I/O, event loop, and concurrency tools) module provides tools for building concurrent applications using coroutines. While the [threading](https://pymotw.com/3/threading/index.html#module-threading) module implements concurrency through application threads and [multiprocessing](https://pymotw.com/3/multiprocessing/index.html#module-multiprocessing) implements concurrency using system processes, [asyncio](https://pymotw.com/3/asyncio/" \l "module-asyncio" \o "asyncio: Asynchronous I/O, event loop, and concurrency tools) uses a single-threaded, single-process approach in which parts of an application cooperate to switch tasks explicitly at optimal times. Most often this context switching occurs when the program would otherwise block waiting to read or write data, but [asyncio](https://pymotw.com/3/asyncio/" \l "module-asyncio" \o "asyncio: Asynchronous I/O, event loop, and concurrency tools) also includes support for scheduling code to run at a specific future time, to enable one coroutine to wait for another to complete, for handling system signals, and for recognizing other events that may be reasons for an application to change what it is working on.

It's built to handle asynchronous I/O from the ground up. Other environments have async. I/O features, but Node's the first environment where it's really pervasive. In most environments you'll find only limited pieces available in async. flavors, but in Node everything (or nearly everything) is async.-only. It's actually hard to write non-async. code in Node!  
  
Now, there's some debate over whether async. programming is really the silver bullet some claim it is, but in my mind there's little doubt that it's a really good match to a lot of common web- and network-development problems.

It's "just JavaScript." Every time I  context switch between Python on the backend and JavaScript on the frontend I waste stupid amounts of time making silly syntax errors — semicolons in my Python, missing braces in my JavaScript, etc. Some days I might switch a dozen or more times, and it really feels like I'm wasting brain cycles swapping in and out my language knowledge. Staying in a single language feels faster.

**Question 8:** Where does Python suck? Why should we care? Why do we still use Python

**Answer:**

* Auto code blocks using indentation. No need for brackets to organize functions, loops, or classes.
* Short constructs for for loop, and double for loop.
* Usage of lists, and dictionaries as built-in data types. Built-in functions operate on lists, and dictionaries without hassles.
* In-built map, reduce, slice, etc. function which makes operating on lists, or dictionaries quick and concise.
* Huge community of contributors. You may find a python module for almost anything.
* Duck-typing. It's both good and bad.