@ What Are **Closures** in Python**?**

They are **function objects** returned by another function. We use them to **eliminate code redundancy**.

In the example below, we've written a simple closure for multiplying numbers.

```
def multiply_number(num):
  def product(number):
    'product() here is a closure'
    return num * number
  return product

num_2 = multiply_number(2)
print(num_2(11))
print(num_2(24))

num_6 = multiply_number(6)
print(num_6(1))
```

The output is:

```
22
48
6
```

@ What Are **Decorators** In Python**?**

Python decorator can **add new behavior** to the given objects **dynamically**.
In the example below,

```
def decorator_sample(func):
  def decorator_hook(*args, **kwargs):
    print("Before the function call")
    result = func(*args, **kwargs)
    print("After the function call")
    return result
  return decorator_hook

@decorator_sample
def product(x, y):
  "Function to multiply two numbers."
  return x * y

print(product(3, 3))
```

The output is:

```
Before the function call
After the function call
9
```

@ JSON

-The **String** format is mainly used to pass the data into another program or load into a datastructure.

json loads -> returns an **object** from a **string** representing a json object.

```
import json

data = b'{"nonce":"820346305172674795","hash":"2E312E38343A353030302F746F67676C
655F646F6F722431"}'

decoded_data = data.decode('utf-8')
json_data = json.loads(decoded_data)
```

json dumps -> returns a **string** representing a json object from an object.

```
>>> a = {'foo': 3}
>>> json.dumps(a)
'{"foo": 3}'
```

load and **dump** -> read/write from/to **file** instead of **string**

@ Descriptor

Python **descriptors** are a way to create **managed attributes**. Among their many advantages, **managed attributes** are used to protect an attribute from changes or to automatically update the values of a dependant attribute.

It allows a programmer to easily and efficiently manage attribute access:
`get` , `set` , and `delete`
If any of these methods are defined for an object, it is said to be a descriptor.

In other languages, descriptors are referred to as *setter* and *getter*, where *public functions* are used to Get and Set a *private variable*.
Python doesn't have a private variables concept, and descriptor protocol can be considered as a Pythonic way to achieve something similar.

It is important to note that descriptors are assigned to a class, not an instance.

@ Generator (memory saving)

lazy processing, it's good for calculating **large sets** of data (in particular calculations involving loops themselves) where you don't know if you are going to need all results, or where you don't want to allocate the memory for all results at the same time.

Python's default file interface acts similar to a generator: it loads content lazily as chunks, and immediately throws it away again when getting the next chunk. The key is to reflect this with your application, and only work on a piece at a time.

Using a Generator, the memory required for processing the file is the maximum size of any **line**, plus whatever memory you use to process that line.

```
#  List version fibonacci-up-to-n function

def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result
```

=> rewritten with **Generator**

```
# Generator version

def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b
```

Generator function is clearer. you can call the **generator** like this:

```
for x in fibon(20):
    print x,
```

The **generator** is to treat results one by one(one at a time), **avoiding building huge lists** of results that you would process separated anyway.

in this example, if using the generator version, the whole 1000000 item list won't be created at all, just one value at a time. That would not be the case when using the **list** version, where a list would be created first.

@ Thread vs Process

1, Threads share data by default runing under one process; **processes** do not. They have their own memories. So, **sending** data between **processes** generally requires **pickling** and **unpickling it.**

2, The GIL is necessary because the Python interpreter is not *thread safe*.

@ For virtualenv to install all files in the requirements.txt file.

1. cd to the directory where **requirements.txt** is located.
2. activate your virtualenv.
3. **run**: pip install -r **requirements.txt** in your shell.

@ Combine two lists as a tuple in new list.

```
list_a = [1, 2, 3, 4]
list_b = [ "aaa" , "bbb" , "ccc" ]
mylist = list(zip(list_a, list_b))
>>> mylist
[(1, 'aaa'), (2, 'bbb'), (3, 'ccc')]
```

map

Example:

```
>>> map(lambda a, b: a+b, [1,2,3,4], (2,3,4,5))
[3, 5, 7, 9]

>>> map(lambda a, b: a + b if b else a + 10, [1,2,3,4,5], (2,3,4,5))   # the second iterable list is one item short
[3, 5, 7, 9, 15]

>>> map(None, [1,2,3,4,5], [1,2,3])

[(1, 1), (2, 2), (3, 3), (4, None), (5, None)]
```

reduce

reduce(function, sequence[, initial]) -> value

Example:

```
>>> reduce(lambda x, y: x+y, range(0,10))
45
>>> reduce(lambda x, y: x+y, range(0,10), 10)
55
```

Filter

filter(function or None, sequence) -> list, tuple, or string

Return those **items of sequence** for which function(item) is **true**. If function is None, return the items that are true.
If **sequence** is a **tuple** or **string**, return the same type, **else** return a **list**.

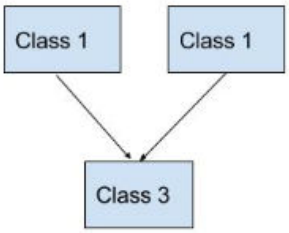
Example:

```
>>> filter(lambda d: d != 'a', 'abcd')       # filter out letter 'a'。
'bcd'
>>> def d(x):    # not using lambda function, instead using a predefined function
        return True if x != 'a' else False
>>> filter(d, 'abcd')
'bcd'
```

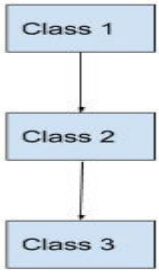
zip

```
weight = [0.5, 0.8, 0.7, 0.8, 1.0]
value = [10, 20, 10, 10, 30]
average = sum([weight[i] * value[i] for i in range(len(weight))])/sum(value)
zip_average = sum ([x * y for x,y in zip(weight,value)]) / sum(value)
> print (average, zip_average)
> 0.825 0.825
```

@ Class Inheritance



Multiple inheritance VS



Multilevel inheritance

Multiple Inheritance Example

This last part is crucial to understand. Let's consider the example again:

```
class First(object):
  def __init__(self):
    super(First, self).__init__()
    print ("first")

class Second(object):
  def __init__(self):
    super(Second, self).__init__()
    print ("second")

class Third(First, Second):
  def __init__(self):
    super(Third, self).__init__()
    print ("that's it")
```

According to this article about Method Resolution Order by Guido van Rossum, the order to resolve `__init__` is calculated (before Python 2.3) using a "**depth-first left-to-right** traversal" :
Third --> First --> object --> Second --> object
After removing all duplicates, except for the last one, we get :
Third --> First --> Second --> object

So, lets follow what happens when we instantiate an instance of the `Third` class,
e.g. `x = Third()`.

- According to MRO `__init__` of Third is called first.
- Next, according to the MRO, inside the `__init__` method `super(Third, self).__init__()` resolves to the `__init__` method of First, which gets called.
- Inside `__init__` of First `super(First, self).__init__()` calls the `__init__` of *Second, because that is what the MRO dictates!*

- Inside `__init__` of Second `super(Second, self).__init__()` calls the `__init__` of **object**, which amounts to nothing. After that **"second" is printed**.
- After `super(First, self).__init__()` completed, **"first" is printed**.
- After `super(Third, self).__init__()` completed, **"that's it" is printed**.

This details out why instantiating Third() results in to :

```
>>> x = Third()
second
first
that's it
```

Try Not To Blow Off Memory!

Don't use `+` for generating long strings
*if already you've contents available in the form of an iterable object, then use `''.join(iterable_object)` which is much **faster.***

@ Regular Expression

+ *means **1** or more.* , ***** *means **zero** or more.*
\d *means **digit*** , **\D** *means **non-digit***

```
>>> pattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$')
>>> pattern.search('415 867 5309 9999').groups()

>>> pattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')
>>> pattern.search('41586753099999').groups()

>>> pattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')
>>> pattern.search('(415)8675309 x9999').groups()
```

Good Luck with your preparation! ☺

Python Basic Level Interview Questions and Answers

1) A = 10, 20, 30
In the above assignment operation, what is the data type of 'A' that Python appreciates as?
Unlike other languages, Python appreciates 'A' as a tuple. When you print 'A', the output is (10,20,30). This type of assignment is called "**Tuple Packing**".

2) >>> A = 1,2,3,4
>>> a,b,c,d = A
In the above assignment operations, what is the value assigned to the variable 'd'?
4 is the value assigned to d. This type of assignment is called '**Tuple Unpacking**'.

3) >>> a = 10
>>> b = 20
Swap these two Variables without using the third temporary variable?
a, b = b, a

This kind of assignment is called a parallel assignment.

4) What is a Variable in Python?
When we say Name = 'john' in Python, the name is not storing the value 'john'. But, 'Name' acts like a **tag** to refer to the object 'john'. The object has types in Python but variables do not, all variables are just tags. All identifiers are variables in Python. Variables never store any data in Python.

5) **>>> a = 10**
>>> b = a
>>> a = 20
>>> print b
What is the output?
In Python, a variable always points to an object and not the variable. So here 'b' points to the object 10, so the output is 10.

6) How do you find the type and identification number of an object in Python?
type(variableName) ,id(variablename)

7) A Dictionary
key-value pairs using a colon(":"). The **keys** should be of an **immutable** type, i.e., so we'll use the data-types which don't allow changes at runtime. We'll choose from an **int, string** or **tuple.**

`dict.items()` returns a list of 2-tuples (`[(key, value), (key, value), ...]`), whereas `dict.iteritems()` is a generator that yields 2-tuples. `items()` takes more space and time initially, but accessing each element is fast, whereas `iteritems()` takes less space and time initially, but a bit more time in generating each element.
One of Python 3's changes is that `items()` now return iterators, and a list is never fully built. The `iteritems()` method is also gone

use the "for" and "in" loop for traversing the dictionary object.

```
>>> for k, v in site_stats.items():
    print("The key is: %s" % k)
    print("The value is: %s" % v)
```

add elements by modifying the dictionary with a **fresh key** and then set the value to it.
If **key** was already exist it'll **replace it**.

```
>>> site_stats_new = {'traffic': 100, "type": "social media"}
>>> site_stats.update(site_stats_new)
```

delete a key in a dictionary by using the **del()**

```
>>> del site_stats["type"]
```

"**in**" operator to test the presence of a key inside a dict object.

```
>>> 'site' in site_stats
True
```

8) A Conditional Expression

```
>>> no_of_days = 366
>>> is_leap_year = "Yes" if no_of_days == 366 else "No"
>>> print(is_leap_year)
Yes
```

Ternary Operator is **alternative** for the conditional statements.
[onTrue] if [Condition] else [onFalse]

```
x, y = 35, 75
smaller = x if x < y else y
```

9) What are the Arithmetic Operators that Python supports?

'%' : Modulo division
'**' : Power Of
'//' : floor div

Python does not support unary operators like ++ or -- operators. On the other hand Python supports "Augmented Assignment Operators"; i.e.,

A += 10 means A = A+10
B -= 10 means B = B - 10

Other Operators supported by Python are **& , |, ~ , ^** which are and, or, bitwise compliment and bitwise xor operations respectively.

10) Enumerate?

The **enumerate()** function attaches **a counter** variable to **an iterable** and returns it as the "enumerated" object in **tuple**.

```
enumerate(iterable, to_begin=0)
```

```
# Example - enumerate function
alist = ["apple","mango", "orange"]
list(enumerate(alist))
[(0, 'apple'), (1, 'mango'), (2, 'orange')]
```

11) PDB Commands

Here are a few PDB commands to start debugging Python code.

python -m pdb python-script.py

- Add breakpoint **(b)**
- Resume execution **(c)**
- Step by step debugging **(s)**
- Move to the next line **(n)**
- List source code **(l)**
- Print an expression **(p)**

12) What is **for-else** and **while-else** in Python**?**

Example :

```
a = []
for i in a:
    print "in for loop"
else:
    print "in else block"
```

output:

in else block

*. The same is true with **while-else** too*

13)

14) How do you find the number of references pointing to a particular object?

15) How do you dispose a variable in Python?

'del' is the keyword statement used in Python to delete a reference variable to an object.

16

17) What is the difference between range() and xrange() functions in Python?

19) Write a generator function to generate a Fibonacci series?

A Generator is a kind of function which lets us specify a function that acts like an iterator and hence can get used in a "for" loop.
In a generator function, the **yield keyword** substitutes the **return statement**.

```
def fib(n):
    a, b = 0, 1
    print('fibonacci start with generator')

    for _ in range(n):
        yield a
        a, b = b, a + b
```

```
for i in fib(20):
    print (i)
```

20) What are the ideal naming conventions in Python?

All **variables** and **functions** follow lowercase and underscore naming convention. ex): is_prime(), test_var = 10 etc
Constants are all either uppercase. ex): MAX_VAL = 50, PI = 3.14
None, True, False are predefined constants follow camel case, etc.
Class names are also treated as constants and follow camel case.
ex): class UserNames():

21) What happens in the background when you run a Python file?
When we run a .py file, it undergoes two phases. In the first phase it checks the syntax and in the second phase it compiles to bytecode (.pyc file is generated) using Python virtual machine, loads the bytecode into memory and runs.

22)

23)

24)

25) What is Hashable? Immutable?

The key of a dictionary should be unique. Internally, the key is transformed by a hash function and become an index to a memory where the value is stored. Suppose, if the key is changed, then it will be pointing somewhere else and we lose the original value the the old key was pointing.

the key should be **hashable**.

Hashable objects are **integers**, **floats**, **strings**, **tuples**, and **frozensets**. Note that all hashable objects are also **immutable** objects.

Mutalbe objects are **lists**, **dictoroaries**, and **sets**.

26)

27) How do you get the last value in a list or a tuple?
When we pass -1 to the index operator of the list or tuple, it returns the last value.

Example:
>>> a = [1,2,3,4]

```
>>> a[-1]
4
>>> a[-2]
3
>>>
>>> b = (1,2,3,4)
>>> b[-1]
4
>>> b[-2]
3
```

28)

29) What is slice notation in Python to access elements in an iterator?
In Python, to access more than one element from a list or a tuple we can use '**:**' operator. Here is the syntax. Say 'a' is list

```
a[startindex:endIndex:Step]
```

Example:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8]

>>> a[3:]      # Prints the values from index 3 till the end
[4, 5, 6, 7, 8]

>>> a[3:6]    #Prints the values from index 3 to index 6.
[4, 5, 6]

>>> a[2::2]    #Prints the values from index 2 till the end of the list with step count 2.
[3, 5, 7]
>>>
```

The above operations are valid for a tuple too.

30) How do you convert a list of integers to a comma separated string?
List elements can be turned into a string using join function.

Example:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8]
>>> numbers = ','.join(str(i) for i in a)
>>> print numbers
1,2,3,4,5,6,7,8
>>>
```

31) What is the difference between Python append () and extend () functions?
The **extend()** function takes an iterable (list or tuple or set) and adds each element of the iterable to the **list**. Whereas **append**() takes a value and adds to the list as a single object.

Example:

```
>>> b = [6,7,8]
>>> a = [1,2,3,4,5]
>>> b = [6,7,8]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6, 7, 8]

>>> c = ['a','b']
>>> a.append(c)
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, ['a', 'b']]
```

```
>>>
```

32) What is List Comprehension? Give an Example.
List comprehension is a way of elegantly constructing a list. It's a simplified way of constructing lists and yet powerful.

Example:
```
>>> a = [x*2 for x in range(10)]
>>> a
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In the above example, a list is created by multiplying every value of x by 2.
Here is another example-
```
>>> [x**2 for x in xrange(10) if x%2==0]
[0, 4, 16, 36, 64]
>>>
```
Here the numbers ranging from 0-9 and divisible by 2 are raised by power 2 and constructed into a list.

33) Tell me about a few string operations in Python?
Here are the most commonly used text processing methods.

```
>>> a = 'hello world'
>>> a.upper() , a.lower() , a.capitalize ,
    a.title()  #First character of the every word is capitalized.

>>> a.split()  #Splits based on default space and returns a list.
['hello', 'world']
>>> record = 'name:age:id:salary'  #Splitting based on ':'
>>> record.split(':')
['name', 'age', 'id', 'salary']

>>> a.strip()      #strips leading and trailing white spaces and newlines.
>>> a.lstrip() , a.rstrip()
>>> a.isalpha()  #returns true only if the string contains alphabets.
>>> a.isdigit()  #returns True only if the string contains digits.
>>> a.isalnum()  #returns true if the string contains alphabets and digits.
>>>
>>> a = 'Name: {name}, Age: {age}'
>>> a.format(name='john', age='18')
'Name: john, Age: 18'
>>>
```

34) How do you create a list which is a reverse version on another list in Python?
Python provides a function called reversed(), which will return a reversed iterator. Then, one can use a list constructor over it to get a list.

Example:
```
>>> a =[10,20,30,40,50]
>>> b = list(reversed(a))
>>> b
[50, 40, 30, 20, 10]
>>> a
[10, 20, 30, 40, 50]
```

35) What is a dictionary in Python?
In Python, dictionaries are kind of hash or maps in another language. Dictionary consists of a key and a value. Keys are unique, and values are accessed using keys. Here are a few examples of creating and accessing dictionaries.

Examples:
```
>>> a = dict()
>>> a['key1'] = 2
>>> a['key2'] = 3
```

```
>>> a
{'key2': 3, 'key1': 2}
>>> a.keys()       # keys() returns a list of keys in the dictionary
['key2', 'key1']
>>> a.values()       # values() returns a list of values in the dictionary
[3, 2]
>>> for i in a:       # Shows one way to iterate over a dictionary items.
   print i, a[i]
key2 3
key1 2
>>> print 'key1' in a    # Checking if a key exists
True
>>> del a['key1']    # Deleting a key value pair
>>> a
{'key2': 3}
```

36) How do you merge one dictionary with the other?

Python provides an update() method which can be used to merge one dictionary on another.

Example:

```
>>> a = {'a':1}
>>> b = {'b':2}
>>> a.update(b)
>>> a
{'a': 1, 'b': 2}
```

37) How to walk through a list in a sorted order without sorting the actual list?

In Python we have function called sorted(), which returns a sorted list without modifying the original list. Here is the code:

```
>>> a
[3, 6, 2, 1, 0, 8, 7, 4, 5, 9]
>>> for i in sorted(a):
   print i,
0 1 2 3 4 5 6 7 8 9
>>> a
[3, 6, 2, 1, 0, 8, 7, 4, 5, 9]
>>>
```

38) names = [‘john’, ‘fan’, ‘sam’, ‘megha’, ‘popoye’, ’tom’, ‘jane’, ‘james’,’tony’]
Write one line of code to get a list of names that start with character ‘j’?

Solution:

```
      #One line code to filter names that start with ‘j’
>>> jnames=[name for name in names if name[0] == 'j']
>>> jnames
['john', 'jane', 'james']
>>>
```

39) a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Write a generator expression to get the numbers that are divisible by 2?
Solution:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b = (i for i in a if i%2 == 0)    #Generator expression.

>>> for i in b: print (i)
0 2 4 6 8
>>>
```

40) What is a set?
A Set is an unordered collection of unique objects.

41) a = "this is a sample string with many characters"
Write a Python code to find how many different characters are present in this string?

Solution:

```
>>> a = "this is a sample string with many characters"
>>> len(set(a))
16
```

42) a = "I am Singh and I live in singh malay and I like Singh is King movie and here I am"

***Write a program that prints the output as follows:**

I :4
am : 2
Singh : 3
and so on..

i.e <word> : <number of its occurrence> **in the string 'a' .**

Solution:

```
>>> k = {}
>>> for item in set(a.split()):
   k[item] = a.split().count(item)
>>> for item in k: print item, k[item]
and 3
king 1
like 1
I 4
movie 1
is 1
am 2
malay 1
here 1
live 1
in 1
singh 3
```

43) What is *args and **kwargs?
*args is used when the programmer is not sure about how many arguments are going to be passed to a function, or if the programmer is expecting a list or a tuple as argument to the function.

**kwargs is used when a dictionary (keyword arguments) is expected as an argument to the function.

44) >>> def welcome(name='guest',city): print 'Hello', name, 'Welcome to', city
What happens with the following function definition?

Here the issue is with function definition, it is a syntax error and the code will not run.

Non-default arguments should be placed first and then comes the default(keyward) arguments in the function definition.
Here is the right way of defining:

```
def welcome(city, name='guest'): print 'Hello', name, 'Welcome to', city
```

The order of passing values to a function is, first one has to pass non-default arguments, default arguments, variable arguments, and keyword arguments.

45) Name some standard Python errors you know?

TypeError: Occurs when the expected type doesn't match with the given type of a variable.
ValueError: When an expected value is not given- if you are expecting 4 elements in a list and you gave 2.
NameError: When trying to access a variable or a function that is not defined.
IOError: When trying to access a file that does not exist.
IndexError: Accessing an invalid index of a sequence will throw an IndexError.
KeyError: When an invalid key is used to access a value in the dictionary.
We can use **dir(__builtin__)** will list all the errors in Python.

46) How Python supports encapsulation with respect to functions?

Python supports inner functions. A function defined inside a function is called an inner function, whose behavior is not hidden. This is how Python supports encapsulation with respect to functions.

47) What is a decorator?
In simple terms, decorators are wrapper functions that takes **a callable** as an argument and extends **its behavior** and returns a callable. Decorators are one kind of design patterns where the behavior of a function can be changed without modifying the functionality of the original function definition.

48) What is PEP8?
PEP 8 is a coding convention about how to write Python code for more readability.

49) How do you open an already existing file and add content to it?

In Python, **open(<filename>,<mode>)** is used to open a file in different modes. The open function returns a **handle** to the file, using which one can perform read, write and modify operations.

Example:

```
F = open("simplefile.txt","a+")   # Opens the file in append mode
F.write("some content")           # Appends content to the file.
F.close()                         # closes the file.
```

50) What mode is used for both writing and reading in binary format in file.?
"**wb+**" is used to open a binary file in both read and write format. It overwrites if the file exists. If the file does not exist it creates a new file for reading and writing.