Question1:

What is Python? What are the advantages of Python?

Python is a high level object-oriented programming language with objects, modules, threads, exceptions and automatic memory management.It can run equally on different platforms such as Windows, Linux, UNIX, Macintosh etc. Thus, Python is a portable language.

Benifits:

```
1. Dynamically Typed:
    dont need to state the type of a variables when you declare them.

    x = 111
    x = "Hello World"

2. Well suited to OOPs:
    allows defination of classes along with composition and inheritance
    it doesn't have access specifiers

3. Functions are first class objects:
    they can be assign to a variables, returned from other functions and passed int
o a functions.classes are also
    first class objects.

4. python Finds usages in:
    web application, automation, scientific modeling, big data applications and man
y more
```

Question2:

What is PEP8

[Python Enhancement Proposals]

Style guide for coding convention and suggestion. the main objectives of PEP8 is to make python code more readable. Coding conventions are about indentation, formatting, tabs, maximum line length, imports organization, line spacing etc. We use PEP 8 to bring consistency in our code. With this consistency, it is easier for other developers to read the code.

Question3:

What is Python zip() function?

This function transform multiple list to a single list of tuples by taking corresponding elements of the list that are passed as arguments

In [3]:
```
list1 = [1 ,2, 3, 4]
list2 = [7, 6, 2, 1]
list3= ['a', 'b', 'c', 'd']

for x,y,z in zip(list1, list2, list3):
    print(x, y, z)
```

```
1 7 a
2 6 b
3 2 c
4 1 d
```

Question4:

What is Python's parameter passing mechanism?

There are two parameter passing mechanism in Python:

Pass by references Pass by value By default, all the parameters (arguments) are passed "by reference" to the functions. Thus, if you change the value of the parameter within a function, the change is reflected in the calling function.

The pass by value is that whenever we pass the arguments to functions that are of type say numbers, strings, tuples. This is because of the immutable nature of them.

Question5:

How to overload constructors or methods in Python?

There are two ways you can overload constuctor or methods in python.

1. using @classmethod decorator
2. using metaclass = MultipleMeta

A much neater way to get 'alternate constructors' is to use classmethods. For instance:

In [2]:
```
# https://stackoverflow.com/questions/141545/how-to-overload-init-method-based-on-argument-type
class MyData:
     def __init__(self, data):
         "Initialize MyData from a sequence"
         self.data = data

     @classmethod
     def fromfilename(cls, filename):
         "Initialize MyData from a file"
         data = open(filename).readlines()
         return cls(data)

     @classmethod
     def fromdict(cls, datadict):
         "Initialize MyData from a dict's items"
         return cls(datadict.items())

#MyData([1, 2, 3]).data

#MyData.fromfilename("/tmp/foobar").data

MyData.fromdict({"spam": "ham"}).data
```

Out[2]:
```
dict_items([('spam', 'ham')])
```

#Available in python3 import time import MultipleMeta class Date(metaclass=MultipleMeta): def __init__(self, year:int, month:int, day:int): self.year = year self.month = month self.day = day def __init__(self): t = time.localtime() self.__init__(t.tm_year, t.tm_mon, t.tm_mday) Output: >>> d = Date(2012, 12, 21) >>> d.year 2012 >>> e = Date() >>> e.year 2018

Question6:

How memory is managed in Python?

https://www.youtube.com/watch?v=arxWaw-E8QQ (https://www.youtube.com/watch?v=arxWaw-E8QQ)

Algorithm used for this is reference counting.

Memory is managed in Python in following way:

1. Memory is managed in Python by private heap space. All Python objects and data structures are located in a private heap. The programmer does not have an access to this private heap and interpreter takes care of this Python private heap.
2. Python memory manager is responsible for allocating Python heap space for Python objects.
3. Python also have an inbuilt garbage collector, which recycle all the unused memory and frees the memory and makes it available to the heap space.

Question7:

In the context of design patterns, decorators dynamically alter the functionality of a function, method or class without having to directly use subclasses. This is ideal when you need to extend the functionality of functions that you don't want to modify. We can implement the decorator pattern anywhere, but Python facilitates the implementation by providing much more expressive features and syntax for that.

Decorators are callable objects which are used to modify functions or classes.

In [4]:
```
import time


def timing_function(some_function):

    """
    Outputs the time a function takes
    to execute.
    """

    def wrapper():
        t1 = time.time()
        some_function()
        t2 = time.time()
        return "Time it took to run the function: " + str((t2 - t1)) + "\n"
    return wrapper


@timing_function
def my_function():
    num_list = []
    for num in (range(0, 10000)):
        num_list.append(num)
    print("\nSum of all the numbers: " + str((sum(num_list))))


print(my_function())
```

```
Sum of all the numbers: 49995000
Time it took to run the function: 0.003988027572631836
```

In [ ]:
```
from functools import wraps

# Did you notice that the function gets passed to the functools.wraps() decora
tor?
# This simply preserves the metadata of the wrapped function.

def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function


@login_required
def upvote(request, product_id):
    if request.method == 'POST':
        product = get_object_or_404(Product, pk=product_id)
        product.votes_total += 1
        product.save()
        return redirect('/products/' + str(product.id))
```

Question8:

What are the rules for local, nonlocal and global variable in Python?

In Python, variables that are only referenced inside a function are called implicitly global. If a variable is assigned a new value anywhere within the function's body, it's assumed to be a local. If a variable is ever assigned a new value inside the function, the variable is implicitly local, and you need to explicitly declare it as 'global'.

In [ ]:
```
def f():
    s = "Only in spring, but London is great as well!"
    print(s)



f()
print(s)
```

In [4]:
```
#Global:

def f():
    global s
    print(s)
    s = "Only in spring, but London is great as well!"
    print(s)


s = "I am looking for a course in Paris!"
f()
print(s)

## once you call global after the "global s" if you assign anything to s, that
value will persist allover
```

```
I am looking for a course in Paris!
Only in spring, but London is great as well!
Only in spring, but London is great as well!
```

In [9]:
```
# Global inside nestead function:


def f():
    x = 42

    def g():
        global x # global x doesn't have impact inside f() but it will have im
pact outside f()
        x = 43
        print(x) # it will print 43 as it is
    print("Before calling g: " + str(x)) # this will be 42 because g() is not
 yet called
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))


f()
print("x in main: " + str(x)) # it will print 43
```

```
Before calling g: 42
Calling g now:
43
After calling g: 42
x in main: 43
```

Nonlocal:

Python3 introduced nonlocal variables as a new kind of variables. nonlocal variables have a lot in common with global variables. This means that nonlocal bindings can only be used inside of nested functions. A nonlocal variable has to be defined in the enclosing function scope. If the variable is not defined in the enclosing function scope, the variable cannot be defined in the nested scope. This is another difference to the "global" semantics.

In [10]:
```
def f():
    x = 42

    def g():
        nonlocal x ## it will have impact to enclosing function, here it is f
()

        x = 43 ## now we can change the value of x

    print("Before calling g: " + str(x))  # Before calling g: 42
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))  # After calling g: 43



f()
print("x in main: " + str(x))  # this will throw error ( NameError: name 'x' i
s not defined)
                               # this would have print 43 if we replace nonloc
al to global
```

```
Before calling g: 42
Calling g now:
After calling g: 43
x in main: 43
```

By example, a variable defined inside a function is local to that function. Another variable defined outside any other scope is global to the function. Suppose we have nested functions. We can read a variable in an enclosing scope from inside the inner function, but cannot make a change to it. For that, we must declare it nonlocal inside the function. First, let's see this without the nonlocal keyword.

In [47]:
```
def outer():
    a=7
    def inner():
        print(a)
    inner()
outer()
```

```
7
```

In [49]:
```
def outer():
    a=7
    def inner():
        print(a)
        a+=1
        print(a)
    inner()

outer()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-49-e87e4d51c287> in <module>()
      7     inner()
      8
----> 9 outer()

<ipython-input-49-e87e4d51c287> in outer()
      5         a+=1
      6         print(a)
----> 7     inner()
      8
      9 outer()

<ipython-input-49-e87e4d51c287> in inner()
      2     a=7
      3     def inner():
----> 4         print(a)
      5         a+=1
      6         print(a)

UnboundLocalError: local variable 'a' referenced before assignment
```

In [55]:
```
#So now, let's try doing this with the 'nonlocal' keyword:
def outer():
    a=7
    def inner():
        nonlocal a
        print(a)
        a+=1
        print(a)
    inner()

outer()
```

```
7
8
```

Question9:

What are iterators in Python

In Python, iterators are used to iterate a group of elements, containers like list. Iterator in python is any python type that can be used with a 'for in loop'. Python lists, tuples, dicts and sets are all examples of inbuilt iterators. These types are iterators because they implement following methods. In fact, any object that wants to be an iterator must implement following methods.

1. __iter__ method that is called on initialization of an iterator. This should return an object that has a next or **next** (in Python 3) method.
2. next (__next__ in Python 3) The iterator next method should return the next value for the iterable. When an iterator is used with a 'for in' loop, the for loop implicitly calls next() on the iterator object. This method should raise a StopIteration to signal the end of the iteration.

In [22]:
```
# you can implement iterator of your own. but you need to implement __iter__an
d __next__ method
class test(object):

    def __init__(self, steps):
        self.steps = steps

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):

        if self.n < self.steps:
            self.n = self.n + 1
            return self.n
        else:
            raise StopIteration


for i in test(10):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

Question10:

What is generator and difference between generator and function in Python?

function will have return but generator will have yield.

function will return 1st element of a list where generator will return object.

generator will print whole list

Function consumes too much memory wheres generator consumes almost same memory as before

In [11]:
```
def function(names):
    for name in names:
        return name

def generator(names):
    for name in names:
        yield name


students = function(['Abe', 'Bob', 'Christina', 'Derek', 'Eleanor'])
students1 = generator(['Abe', 'Bob', 'Christina', 'Derek', 'Eleanor'])

print(students1.__next__())
print(students1.__next__())
print(list(students1))
print(students) # this returns "Abe" bacause for name in names iterates only o
nce
```

```
Abe
Bob
['Christina', 'Derek', 'Eleanor']
Abe
```

In [10]:
```
import psutil
import random
import time
import os
import sys

def memory_usage_psutil():
    # return the memory usage in MB
    process = psutil.Process(os.getpid())
    mem = process.memory_info()[0] / float(2 ** 20)
    return mem

def memory_usage_resource():
    rusage_denom = 1024.
    if sys.platform == 'darwin':
        # ... it seems that in OSX the output is different units ...
        rusage_denom = rusage_denom * rusage_denom
    mem = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss / rusage_denom
    return mem

names = ['John', 'Corey', 'Adam', 'Steve', 'Rick', 'Thomas']
majors = ['Math', 'Engineering', 'CompSci', 'Arts', 'Business']

print('Memory (Before): {}Mb'.format(memory_usage_psutil()))

def people_list(num_people):
    result = []
    for i in range(num_people):
        person = {
                    'id': i,
                    'name': random.choice(names),
                    'major': random.choice(majors)
                }
        result.append(person)
    return result

def people_generator(num_people):
    for i in range(num_people):
        person = {
                    'id': i,
                    'name': random.choice(names),
                    'major': random.choice(majors)
                }
        yield person

# t1 = time.clock()
# people = people_list(1000000)
# t2 = time.clock()

t1 = time.clock()
people = people_generator(1000000)
t2 = time.clock()

print("Memory (After) : {} Mb".format(memory_usage_psutil()))
print("Took {} Seconds".format(t2-t1))
```

```
Memory (Before): 337.4296875Mb
Memory (After) : 68.046875 Mb
Took 0.13150728645410936 Seconds
```

```
In [38]: print("Memory (After) : {} Mb".format(10))
```

```
Memory (After) : 10 Mb
```

Question11:

What is slicing in Python?

Slicing is a mechanism used to select a range of items from sequence type like list, tuple, string etc.

```
+---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
   0   1   2   3   4   5
  -6  -5  -4  -3  -2  -1
```

https://stackoverflow.com/questions/509211/understanding-pythons-slice-notation (https://stackoverflow.com/questions/509211/understanding-pythons-slice-notation)

Question12:

What is Pass in Python?

Pass specifies a Python statement without operations. It is a place holder in a compound statement, where there should be a blank left and nothing has to be written there.

Question13:

Explain docstring in Python?

A Python documentation string is called docstring. It is used for documenting Python functions, modules and classes.

Question14:

What is negative index in Python?

Python sequences are indexed in positive and negative numbers. For example: 0 is the first positive index, 1 is the second positive index and so on. For negative indexes -1 is the last negative index, -2 is the second last negative index and so on.

Question15:

What is pickling and unpickling in Python?

Introduction:

Literally, the term pickle means storing something in a saline solution. Only here, instead of vegetables its objects. Not everything in life can be seen as 0s and 1s (gosh! philosophy), but pickling helps us achieve that since it converts any kind of complex data to 0s and 1s (byte streams). This process can be referred to as pickling, serialization, flattening or marshalling. The resulting byte stream can also be converted back into Python objects by a process known as Unpickling.

Why Pickle? Since we are dealing with binary, the data is not written but dumped and similarly, the data is not read, it is loaded. For example, when you play a game like 'Dave' and you reach a certain level, you would want to save it right? As you know there are various attributes to this game like, health, gems collected etc. So when you save your game, say at level 7 when you have one heart for health and thirty hundred points, an object is created from a class Dave with these values. When you click the 'Save' button, this object is serialized and saved or in other words pickled. Needless to say, when you restore a saved game, you will be loading data from its pickled state thus unpickling it.

The real world uses of Pickling and Unpickling are widespread as they allow you to easily send data from one server to another, and store it in a file or database.

WARNING: Never unpickle data received from an untrusted source as this may pose some serious security risks. The Pickle module is not capable of knowing or raising errors while pickling malicious data.

Pickling and Unpickling can be used only if the corresponding module Pickle is imported. You can do this by using the following command:

In [12]:
```
import pickle
```

In [ ]:
```
emp = {1:"A",2:"B",3:"C",4:"D",5:"E"}
pickling_on = open("Emp.pickle","wb")
pickle.dump(emp, pickling_on)
pickling_on.close()
```

Note the usage of “wb” instead of “w” as all the operations are done using bytes. At this point, you can go and open the Emp.pickle file in the current working directory using a Notepad and see how the pickled data looks.

So, now that the data has been pickled, let’s work on how to unpickle this dictionary.

In [ ]:
```
pickle_off = open("Emp.pickle","rb")
emp = pickle.load(pickle_off)
print(emp)
```

Now you will get the employees dictionary as we initialized earlier. Note the usage of “rb” instead of “r” as we are reading bytes. This is a very basic example, be sure to try more on your own.

If you want to get a byte string containing the pickled data instead of a pickled representation of obj, then you need to use dumps. Similarly to read pickled representation of objects from byte streams you should use loads.

Data stream format The data stream format is referred to as the protocol which specifies the output format of the pickled data. There are several protocol versions that are available. You must be aware of the protocol version to avoid compatibility issues.

Protocol version 0 - the original text-based format that is backwards compatible with earlier versions of Python. Protocol version 1 - an old binary format which is also compatible with earlier versions of Python. Protocol version 2 - introduced in Python 2.3 and provides efficient picking of classes and instances, Protocol version 3 - introduced in Python 3.0 but it is not backwards compatible. Protocol version 4 - added in Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data format optimizations. Note that the protocol version is saved as a part of the pickle data format. However, to unpickle data in a specific protocol, there are provisions to specify it while using the dump() command.

To know the protocol used, use the following command after importing the pickle library. This will return the highest protocol being used.

pickle.HIGHEST_PROTOCOL

Exceptions Some of the common exceptions to look out for:

Pickle.PicklingError: This exception is raised when you are trying to pickle an object that doesn’t support pickling.

Pickle.UnpicklingError: This exception is raised when a file contains corrupted data.

EOFError: This exception is raised when the end of file is detected.

Question16:

What is the usage of help() and dir() function in Python?

Help() and dir() both functions are accessible from the Python interpreter and used for viewing a consolidated dump of built-in functions.

Help() function: The help() function is used to display the documentation string and also facilitates you to see the help related to modules, keywords, attributes, etc.

Dir() function: The dir() function is used to display the defined attributes of the objects or modules.

In [13]:
```
help()
```

```
Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> modules

Please wait a moment while I gather a list of all available modules...


C:\Users\ABHISEK\Anaconda3\lib\site-packages\IPython\kernel\__init__.py:13: S
himWarning: The `IPython.kernel` package has been deprecated since IPython 4.
0.You should import from ipykernel or jupyter_client instead.
  "You should import from ipykernel or jupyter_client instead.", ShimWarning)
WARNING: AstropyDeprecationWarning: The astropy.vo.samp module has now been m
oved to astropy.samp [astropy.vo.samp]
C:\Users\ABHISEK\Anaconda3\lib\site-packages\h5py\__init__.py:36: FutureWarni
ng: Conversion of the second argument of issubdtype from `float` to `np.float
ing` is deprecated. In future, it will be treated as `np.float64 == np.dtype
(float).type`.
  from ._conv import register_converters as _register_converters
C:\Users\ABHISEK\Anaconda3\lib\site-packages\odo\backends\pandas.py:102: Futu
reWarning: pandas.tslib is deprecated and will be removed in a future versio
n.
You can access NaTType as type(pandas.NaT)
  @convert.register((pd.Timestamp, pd.Timedelta), (pd.tslib.NaTType, type(Non
e)))
C:\Users\ABHISEK\Anaconda3\lib\site-packages\nltk\twitter\__init__.py:20: Use
rWarning: The twython library has not been installed. Some functionality from
the twitter package will not be available.
  warnings.warn("The twython library has not been installed. "
C:\Users\ABHISEK\Anaconda3\lib\site-packages\skimage\viewer\utils\core.py:10:
UserWarning: Recommended matplotlib backend is `Agg` for full skimage.viewer
functionality.
  warn("Recommended matplotlib backend is `Agg` for full "
C:\Users\ABHISEK\Anaconda3\lib\site-packages\qtawesome\iconic_font.py:268: Us
erWarning: You need to have a running QApplication to use QtAwesome!
  warnings.warn("You need to have a running "
C:\Users\ABHISEK\Anaconda3\lib\site-packages\statsmodels\compat\pandas.py:56:
FutureWarning: The pandas.core.datetools module is deprecated and will be rem
oved in a future version. Please use the pandas.tseries module instead.
  from pandas.core import datetools
```

```
Crypto              builtins            mkl                 spyder_io_dcm
Cython              bz2                 mmap                spyder_io_hdf5
IPython             cProfile            mmapfile            spyder_profiler
OpenSSL             calendar            mmsystem            spyder_pylint
PIL                 certifi             modulefinder        sqlalchemy
PyQt5               cffi                mpmath              sqlite3
__future__          cgi                 msgpack             sre_compile
_ast                cgitb               msilib              sre_constants
_asyncio            chardet             msvcrt              sre_parse
_bisect             chunk               multipledispatch    ssl
_blake2             click               multiprocessing     sspi
_bootlocale         cloudpickle         navigator_updater   sspicon
_bz2                clyent              nbconvert           stat
_cffi_backend       cmath               nbformat            statistics
_codecs             cmd                 netbios             statsmodels
_codecs_cn          code                netrc               storemagic
_codecs_hk          codecs              networkx            string
_codecs_iso2022     codeop              nltk                stringprep
_codecs_jp          collections         nntplib             struct
_codecs_kr          colorama            nose                subprocess
_codecs_tw          colorsys            notebook            sunau
_collections        commctrl            nt                  symbol
_collections_abc    compileall          ntpath              sympy
_compat_pickle      comtypes            ntsecuritycon       sympyprinting
_compression        concurrent          nturl2path          symtable
_csv                conda               numba               sys
_ctypes             conda_build         numbers             sysconfig
_ctypes_test        conda_env           numexpr             tables
_datetime           conda_verify        numpy               tabnanny
_decimal            configparser        numpydoc            tarfile
_dummy_thread       contextlib          odbc                tblib
_elementtree        contextlib2         odo                 telnetlib
_findvs             copy                olefile             tempfile
_functools          copyreg             opcode              terminado
_hashlib            crypt               openpyxl            test
_heapq              cryptography        operator            test_path
_imp                csv                 optparse            test_pycosat
_io                 ctypes              os                  testpath
_json               curl                packaging           tests
_locale             curses              pandas              textwrap
_lsprof             cwp                 pandocfilters       this
_lzma               cycler              parser              threading
_markupbase         cython              parso               time
_md5                cythonmagic         partd               timeit
_msi                cytoolz             path                timer
_multibytecodec     dask                pathlib             tkinter
_multiprocessing    datashape           pathlib2            tlz
_nsis               datetime            patsy               token
_opcode             dateutil            pdb                 tokenize
_operator           dbi                 pep8                toolz
_osx_support        dbm                 perfmon             tornado
_overlapped         dde                 pickle              trace
_pickle             decimal             pickleshare         traceback
_pydecimal          decorator           pickletools         tracemalloc
_pyio               difflib             pip                 traitlets
_pytest             dis                 pipes               tty
_random             distributed         pkg_resources       turtle
```

```
_sha1               distutils           pkginfo             turtledemo
_sha256             doctest             pkgutil             types
_sha3               docutils            platform            typing
_sha512             dummy_threading     plistlib            unicodecsv
_signal             easy_install        pluggy              unicodedata
_sitebuiltins       email               ply                 unittest
_socket             encodings           poplib              urllib
_sqlite3            ensurepip           posixpath           urllib3
_sre                entrypoints         pprint              uu
_ssl                enum                profile             uuid
_stat               errno               prompt_toolkit      venv
_string             et_xmlfile          pstats              warnings
_strptime           fastcache           psutil              wave
_struct             faulthandler        pty                 wcwidth
_symtable           filecmp             py                  weakref
_system_path        fileinput           py_compile          webbrowser
_testbuffer         filelock            pyclbr              webencodings
_testcapi           flask               pycodestyle         werkzeug
_testconsole        flask_cors          pycosat             wheel
_testimportmultiple fnmatch             pycparser           widgetsnbextensio
n
_testmultiphase     formatter           pycurl              win2kras
_thread             fractions           pydoc               win32api
_threading_local    ftplib              pydoc_data          win32clipboard
_tkinter            functools           pyexpat             win32com
_tracemalloc        gc                  pyflakes            win32con
_warnings           genericpath         pygments            win32console
_weakref            getopt              pylab               win32cred
_weakrefset         getpass             pylint              win32crypt
_win32sysloader     gettext             pyodbc              win32cryptcon
_winapi             gevent              pyparsing           win32event
_winxptheme         glob                pytest              win32evtlog
_yaml               glob2               pythoncom           win32evtlogutil
abc                 greenlet            pytz                win32file
adodbapi            gzip                pywin               win32gui
afxres              h5py                pywin32_testutil    win32gui_struct
aifc                hashlib             pywintypes          win32help
alabaster           heapdict            pywt                win32inet
anaconda_navigator  heapq               pyximport           win32inetcon
anaconda_project    hmac                qtawesome           win32job
antigravity         html                qtconsole           win32lz
argparse            html5lib            qtpy                win32net
array               http                queue               win32netcon
asn1crypto          idlelib             quopri              win32pdh
ast                 idna                random              win32pdhquery
astroid             imageio             rasutil             win32pdhutil
astropy             imagesize           re                  win32pipe
asynchat            imaplib             regcheck            win32print
asyncio             imghdr              regutil             win32process
asyncore            imp                 reprlib             win32profile
atexit              importlib           requests            win32ras
attr                inspect             rlcompleter         win32rcparser
audioop             io                  rmagic              win32security
autoreload          ipaddress           rope                win32service
babel               ipykernel           ruamel_yaml         win32serviceutil
backports           ipykernel_launcher  run                 win32timezone
base64              ipython_genutils    runpy               win32trace
```

```
bdb                 ipywidgets          sched               win32traceutil
binascii            isapi               scipy               win32transaction
binhex              isort               scripts             win32ts
binstar_client      itertools           seaborn             win32ui
bisect              itsdangerous        secrets             win32uiole
bitarray            jdcal               select              win32verstamp
bkcharts            jedi                selectors           win32wnet
blaze               jinja2              send2trash          win_inet_pton
bleach              json                servicemanager      win_unicode_conso
le
bokeh               jsonschema          setuptools          wincertstore
boto                jupyter             shelve              winerror
bottleneck          jupyter_client      shlex               winioctlcon
brain_attrs         jupyter_console     shutil              winnt
brain_builtin_inference jupyter_core        signal              winperf
brain_collections   jupyterlab          simplegeneric       winpty
brain_curses        jupyterlab_launcher singledispatch      winreg
brain_dateutil      keyword             singledispatch_helpers winsound
brain_fstrings      lazy_object_proxy   sip                 winxpgui
brain_functools     lib2to3             sipconfig           winxptheme
brain_gi            linecache           sipdistutils        wrapt
brain_hashlib       llvmlite            site                wsgiref
brain_io            locale              six                 xdrlib
brain_mechanize     locket              skimage             xlrd
brain_multiprocessing logging             sklearn             xlsxwriter
brain_namedtuple_enum lxml                smtpd               xlwings
brain_nose          lzma                smtplib             xlwt
brain_numpy         macpath             sndhdr              xml
brain_pkg_resources macurl2path         snowballstemmer     xmlrpc
brain_pytest        mailbox             socket              xxsubtype
brain_qt            mailcap             socketserver        yaml
brain_re            markupsafe          socks               zict
brain_six           marshal             sockshandler        zipapp
brain_ssl           math                sortedcollections   zipfile
brain_subprocess    matplotlib          sortedcontainers    zipimport
brain_threading     mccabe              sphinx              zlib
brain_typing        menuinst            sphinxcontrib       zmq
brain_uuid          mimetypes           spyder
bs4                 mistune             spyder_breakpoints

Enter any module name to get more help.  Or, type "modules spam" to search
for modules whose name or summary contain the string "spam".

help> scipy

IOPub data rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_data_rate_limit`.

Current values:
NotebookApp.iopub_data_rate_limit=1000000.0 (bytes/sec)
NotebookApp.rate_limit_window=3.0 (secs)
```

```
help> q

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

In [14]:
```
dir()
```

Out[14]:
```
['In',
 'Out',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_i10',
 '_i11',
 '_i12',
 '_i13',
 '_i14',
 '_i2',
 '_i3',
 '_i4',
 '_i5',
 '_i6',
 '_i7',
 '_i8',
 '_i9',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 'exit',
 'function',
 'generator',
 'get_ipython',
 'majors',
 'memory_usage_psutil',
 'memory_usage_resource',
 'names',
 'os',
 'people',
 'people_generator',
 'people_list',
 'pickle',
 'psutil',
 'quit',
 'random',
 'students',
 'students1',
 'sys',
 't1',
 't2',
 'time']
```

In [15]:
```
dir(__builtins__)
```

```
Out[15]: ['ArithmeticError',
          'AssertionError',
          'AttributeError',
          'BaseException',
          'BlockingIOError',
          'BrokenPipeError',
          'BufferError',
          'BytesWarning',
          'ChildProcessError',
          'ConnectionAbortedError',
          'ConnectionError',
          'ConnectionRefusedError',
          'ConnectionResetError',
          'DeprecationWarning',
          'EOFError',
          'Ellipsis',
          'EnvironmentError',
          'Exception',
          'False',
          'FileExistsError',
          'FileNotFoundError',
          'FloatingPointError',
          'FutureWarning',
          'GeneratorExit',
          'IOError',
          'ImportError',
          'ImportWarning',
          'IndentationError',
          'IndexError',
          'InterruptedError',
          'IsADirectoryError',
          'KeyError',
          'KeyboardInterrupt',
          'LookupError',
          'MemoryError',
          'ModuleNotFoundError',
          'NameError',
          'None',
          'NotADirectoryError',
          'NotImplemented',
          'NotImplementedError',
          'OSError',
          'OverflowError',
          'PendingDeprecationWarning',
          'PermissionError',
          'ProcessLookupError',
          'RecursionError',
          'ReferenceError',
          'ResourceWarning',
          'RuntimeError',
          'RuntimeWarning',
          'StopAsyncIteration',
          'StopIteration',
          'SyntaxError',
          'SyntaxWarning',
          'SystemError',
          'SystemExit',
```

```
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'WindowsError',
'ZeroDivisionError',
'_',
'_ASTROPY_SETUP_',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
```

```
'hash',
'help',
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',
'print',
'property',
'range',
'repr',
'reversed',
'round',
'set',
'setattr',
'slice',
'sorted',
'staticmethod',
'str',
'sum',
'super',
'tuple',
'type',
'vars',
'zip']
```

Question17:

What are the differences between Python 2.x and Python 3.x?

Python 2.x is an older version of Python. It is a legacy now. Python 3.x is newer. It is the present and future of this language.

The most visible difference between them is in print statement. In Python 2 it is print "Hello" and in Python 3, it is print ("Hello").

Question18:

What is the shortest method to open a text file and display its content?

In [ ]:
```
with open("file-name", "r") as fp:
fileData = fp.read()
print(fileData)
```

Question19:

What is the usage of enumerate () function in Python?

The enemurate() function is used to iterate through the sequence and retrieve the index position and its corresponding value at the same time.

In [16]:
```
for i,v in enumerate(['Python','Java','C++']):
    print(i,v)
```

```
0 Python
1 Java
2 C++
```

Question20:

How will you specify source code encoding in a Python source file?

By default, every source code file in Python is in UTF-8 encoding. But we can also specify our own encoding for source files. This can be done by adding following line after #! line in the source file.

 # -- coding: encoding --

In the above line we can replace encoding with the encoding that we want to use.

Example1:

!/usr/bin/python

 # -- coding: latin-1 --

Example2:

!/usr/bin/python

 # -- coding: ascii --

Question21:

How does memory management work in Python?

Python memory management is been divided into two parts.

1. Stack memory
2. Heap memory

Methods and variables are created in Stack memory. Objects and instance variables values are created in Heap memory. In stack memory - a stack frame is created whenever methods and variables are created. These stacks frames are destroyed automaticaly whenever functions/methods returns. Python has mechanism of Garbage collector, as soon as variables and functions returns, Garbage collector clear the dead objects.

In CPython there is a memory manager responsible for managing the heap space.

There are different components in Python memory manager that handle segmentation, sharing,caching, memory pre-allocation etc.

Python memory manager also takes care of garbage collection by using Reference counting algorithm.

Question22:

How will you perform Static Analysis on a Python Script?

We can use Static Analysis tool called PyCheckerfor this purpose. PyChecker can detect errors in Python code. PyChecker also gives warnings for any style issues. Some other tools to find bugs in Python code are pylint,Flake8 and pyflakes.

Question23:

What is the difference between a Tuple and List in python?

In Python, Tuple and List are built-in data structures. Some of the differences between Tuple and List are as follows:

```
1. Syntax: A Tuple is enclosed in parentheses:  E.g. myTuple  = (10, 20, “apple”);
           A List is enclosed in brackets: E.g. myList = [10, 20, 30];
2. Mutable: Tuple is an immutable data structure. Whereas, a List is a mutable data
structure.
3. Size: A Tuple takes much lesser space than a List in Python.
4. Performance: Tuple is faster than a List in Python. So it gives us good performa
nce.
5. Use case: Since Tuple is immutable,we can use it in cases like Dictionary creati
on. Whereas, a List is preferred in
   the use case where data can alter.
```

Question24:

What is a Python Decorator?

A Python Decorator is a mechanism to wrap a Python function and modify its behavior by adding more functionality to it.

In [38]:
```
def myfunc(fn):

    def wrap1():
        return "<h1>{}</h1>".format(fn())
    return wrap1


def wrap():
    return "Helloooo"

my_obj = myfunc(wrap) # this is normal way of calling decorator
print(my_obj())
```

```
<h1>Helloooo</h1>
```

In [39]:
```
# above instead of writting my_obj = myfunc(wrap) we can keep @myfunc above wr
ap function

def myfunc(fn):

    def wrap1():
        print("wrapper executed before {}".format(fn.__name__))
        print("<h1>{}</h1>".format(fn()))
        print("wrapper executed after {}".format(fn.__name__))

    return wrap1

@myfunc
def wrap():  # if you want to execute something with this function then you ca
n use myfunc as wrapper function
    return "Helloooo"


print(wrap())
```

```
wrapper executed before wrap
<h1>Helloooo</h1>
wrapper executed after wrap
None
```

In [42]:
```
# Decorator with multiple variables

def myfunc(fn):

    def wrap1(*args, **kwargs):
        print("wrapper executed before {}".format(fn.__name__))
        return fn(*args, **kwargs)

    return wrap1

@myfunc
def wrap():  # if you want to execute something with this function then you ca
n use myfunc as wrapper function
    return "Helloooo"

@myfunc
def wrap2(name, age):
    print("<h1>{}-{}</h1>".format(name, age))


print(wrap())
wrap2('abhi', 'g')
```

```
wrapper executed before wrap
Helloooo
wrapper executed before wrap2
<h1>abhi-g</h1>
```

In [ ]:
```
## Real World Example of Decorator:

from django.contrib.auth.decorators import login_required
@login_required
def upvote(request, product_id):
    if request.method == 'POST':
        product = get_object_or_404(Product, pk=product_id)
        product.votes_total += 1
        product.save()
        return redirect('/products/' + str(product.id))
```

http://devarea.com/python-closure-and-function-decorators/#.W_rsZOgzbIU (http://devarea.com/python-closure-and-function-decorators/#.W_rsZOgzbIU)

https://realpython.com/primer-on-python-decorators/ (https://realpython.com/primer-on-python-decorators/)

Question25:

How are arguments passed in a Python method? By value or by reference?

Every argument in a Python method is an Object. All the variables in Python have reference to an Object. Therefore arguments in Python methodare passed by Reference. Since some of the objects passed as reference are mutable, we can change those objects in a method. But for an Immutable object like String, any change done within a method is not reflected outside.

Correctly speaking, Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing". If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable. It's different, if we pass mutable arguments. They are also passed by object reference, but they can be changed in place in the function.

Question26:

What is the difference between List and Dictionary data types in Python?

Main differences between List and Dictionary data types in Python are as follows:

```
1. Syntax: In a List we store objects in a sequence. In a Dictionary we store objec
ts in key-value pairs.
2. Reference: In List we access objects by index number. It starts from 0 index. In
a Dictionary we access objects
              objects by key specified at the time of Dictionary creation.
3. Ordering: In a List objects are stored in an ordered sequence. In a Dictionary o
bjects are not stored in an
             ordered sequence.
4. Hashing: In a Dictionary, keys have to be hashable. In a List there is no need f
or hashing.
```

Question27:

What are the different built-in data types available in Python?

Some of the built-in data types available in Python are as follows:

Numeric types: These are the data types used to represent numbers in Python.

```
int: It is used for Integers
long: It is used for very large integers of non-limited length.
float: It is used for decimal numbers.
complex: This one is for representing complex numbers
```

In [46]:
```
a = 5
print(a, "is of type", type(a))

a = 2.0
print(a, "is of type", type(a))

a = 1+2j
print(a, "is complex number?", isinstance(1+2j,complex))
```

```
5 is of type <class 'int'>
2.0 is of type <class 'float'>
(1+2j) is complex number? True
```

Sequence types: These data types are used to represent sequence of characters or objects.

```
    str: This is similar to String in Java. It can represent a sequence of characte
rs.
    bytes: This is a sequence of integers in the range of 0-255.
    byte array: like bytes, but mutable (see below); only available in Python 3.x
    list: This is a sequence of objects.
    tuple: This is a sequence of immutable objects.
    Sets: These are unordered collections.
        set: This is a collection of unique objects.
        frozen set: This is a collection of unique immutable objects.
    Mappings: This is similar to a Map in Java.
    dict: This is also called hashmap. It has key value pair to store information b
y using hashing.
```

In [47]:
```
# Convert string to bytes
string = "Python is interesting."

# string with encoding 'utf-8'
arr = bytes(string, 'utf-8')
print(arr)

x = bytearray(string, "utf8")
print(x)

#Difference between bytes and bytearray object in Python
#bytearray objects are a mutable counterpart to bytes objects
x = bytearray("Python bytearray", "utf8")
print(x)

#can remove items from the bytes
del x[11:15]
print(x)

 #can add items from the bytes
x[11:15] = b" object"
print(x)

#can use the methods of mutable type iterable objects as the lists
x.append(45)
print(x)
```

```
b'Python is interesting.'
bytearray(b'Python is interesting.')
bytearray(b'Python bytearray')
bytearray(b'Python bytey')
bytearray(b'Python byte object')
bytearray(b'Python byte object-')
```

Question28:

What is a Namespace in Python?

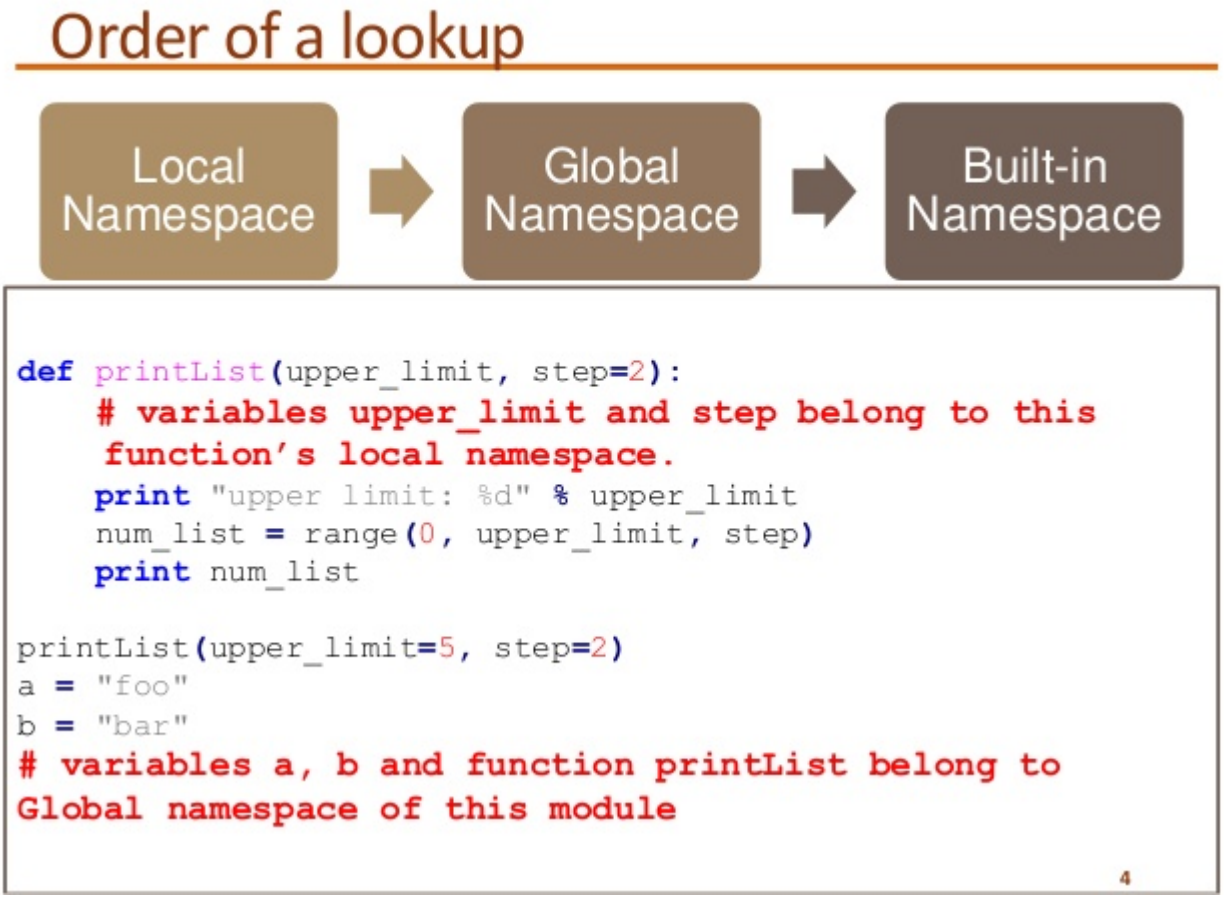
A namespace (sometimes also called a context) is a naming system for making names unique to avoid ambiguity. Namespaces in Python are implemented as Python dictionaries, this means it is a mapping from names (keys) to objects (values). The user doesn't have to know this to write a Python program and when using namespaces. suppose, a = 2, here we are assigning a name "a" to object 2

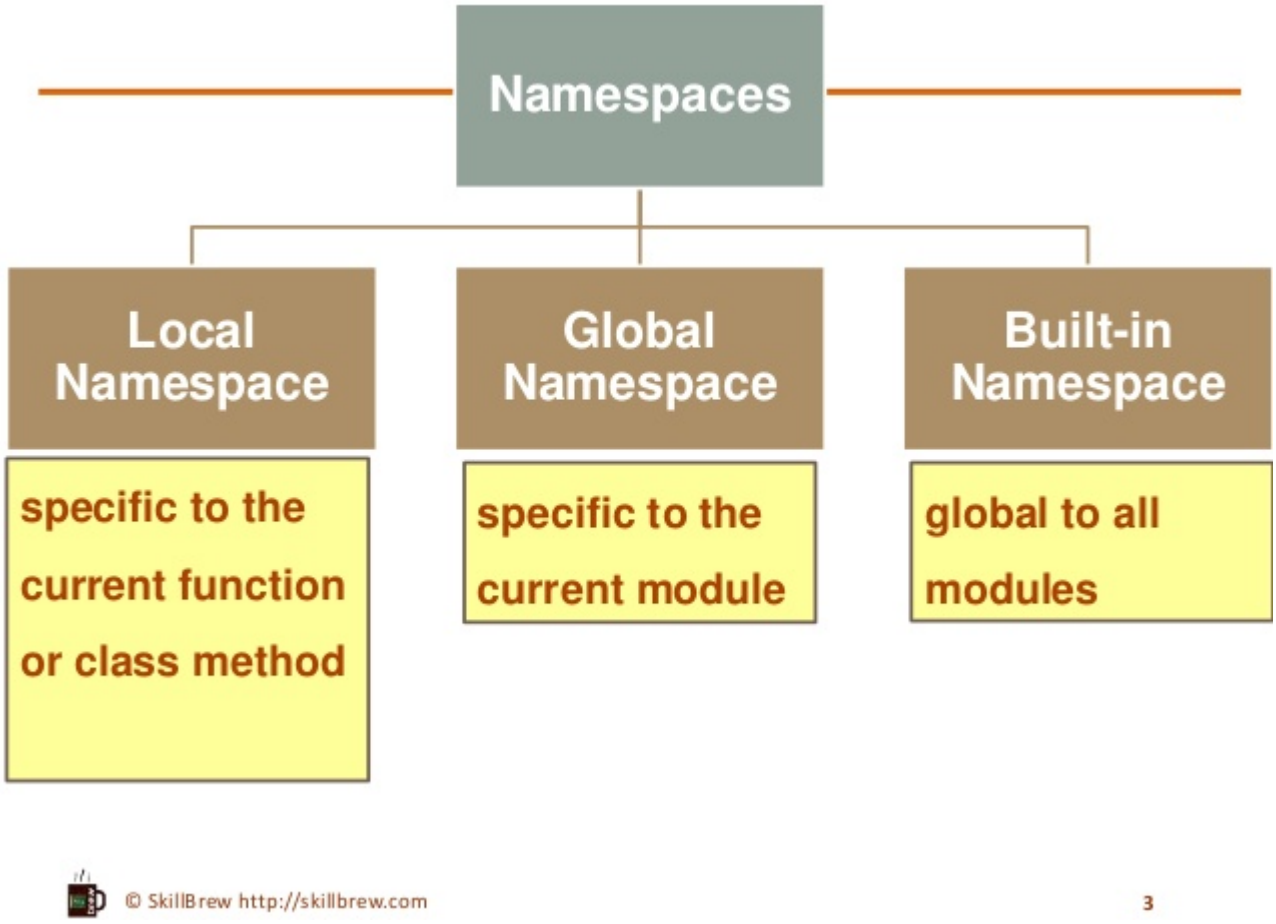
Some namespaces in Python:

global names of a module

local names in a function or method invocation

built-in names: this namespace contains built-in functions (e.g. abs(), cmp(), ...) and built-in exception names





Question29:

how will you concatenate multiple strings together in Python?

We can use following ways to concatenate multiple string together in Python:

```
use + operator:
    E.g. >>> fname="John" >>> lname="Ray" >>> print(fname+lname JohnRay)

use join function:
    E.g. >>> ''.join(['John','Ray']) 'JohnRay'
```

Question30:

What is the use of Pass statement in Python?

The use of Pass statement is to do nothing. It is just a placeholder for a statement that is required for syntax purpose. It does not execute any code or command. Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the pass statement to construct a body that does nothing.

In [50]:
```
# pass is just a placeholder for
# functionality to be added later.
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

We can do the same thing in an empty function or class as well.

In [51]:
```
def function(args):
    pass

class example:
    pass
```

Question31:

what is the use of slicing in Python?

We can use Slicing in Python to get a substring from a String. The syntax of Slicing is very convenient to use. E.g. In following example we are getting a substring out of the name John.

name="John"

name[1:3] 'oh'

In Slicing we can give two indices in the String to create a Substring. If we do not give first index, then it defaults to 0. E.g. >>> name="John" >>> name[:2] 'Jo' If we do not give second index, then it defaults to the size of the String. >>>

name="John"

name[3:] 'n'

Question32:

What is the difference between Docstring in Python and Javadoc in Java?

A Docstring in Python is a string used for adding comments or summarizing a piece of code in Python. The main difference between Javadoc and Docstring is that docstring is available during runtime as well. Whereas, Javadoc is removed from the Bytecode and it is not present in .class file. We can even use Docstring comments at run time as an interactive help manual. In Python, we have to specify docstring as the first statement of a code object, just after the def or class statement. The docstring for a code object can be accessed from the '__doc__' attribute of that object.

Note: Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code, which requires documentation in a predefined format.

```
    Following is a simple example where the lines inside /*….*/ are Java multi-line com
    ments.
    Similarly, the line which preceeds // is Java single-line comment.

    /**
```

- The HelloWorld program implements an application that
- simply displays "Hello World!" to the standard output. *
- @author Zara Ali
- @version 1.0
- @since 2014-03-31

*/

public class HelloWorld {

public static void main(String[] args) {

```
      /* Prints Hello, World! on standard output.

      System.out.println("Hello World!");
```

}}

In [53]:
```
def my_function():
    """Demonstrate docstrings and does nothing really."""

    return None

print("Using __doc__:")
print(my_function.__doc__)
```

```
Using __doc__:
Demonstrate docstrings and does nothing really.
```

Question33:

What is the difference between an Iterator and Iterable in Python?

An Iterable is an object that can be iterated by an Iterator. In Python, Iterator object provides *iter*() and next() methods. In Python, an Iterable object has *iter* function that returns an Iterator object. When we work on a map or a for loop in Python, we can use next() method to get an Iterable item from the Iterator. **Check Question 9**

In [57]:
```
#How to know if an object is iterable?
import collections
theElement = [1,2]
if isinstance(theElement, collections.Iterable):
    print("iterable")
else:
    print("not iterable")
```

```
iterable
```

Question34:

What is the use of Generator in Python?

We can use Generator to create Iterators in Python. A Generator is written like a regular function. It can make use yield statement to return data during the function call. In this way we can write complex logic that works as an Iterator. A Generator is more compact than an Iterator due to the fact that *iter*() and next() functions are automatically created in a Generator. Also within a Generator code, local variables and execution state are saved between multiple calls. Therefore, there is no need to add extra variables like self.index etc to keep track of iteration. Generator also increases the readability of the code written in Python. It is a very simple implementation of an Iterator.

Question35:

Give example of unit testing using pytest?

how to use pytest fixture

https://gist.github.com/pknowledge/1f91f96145189d08c61fb76dc99694ae (https://gist.github.com/pknowledge/1f91f96145189d08c61fb76dc99694ae)

In [ ]:
```
### test_unittest.py

"""What is unit Testing:"""
"""unit testing is a software testing method by which individual units of
   source code are tested to determine whether they are fit for use"""

from Functionstobetested import Queue,Product
import pytest
import sys
a_queue = Queue()
prod = Product(10, 10)


a_queue.enqueue(10)
a_queue.enqueue(12)
a_queue.enqueue(14)


def test_queue():
    assert a_queue.dequeue() == 10
    assert a_queue.dequeue() == 12
    assert a_queue.dequeue() == 14


"""You can skip a test case with reason"""
# @pytest.mark.skip(reason="don't execute this")
# def test_prod():
#     assert prod.mul() == 10


"""you can skip when satisfied some condition """
@pytest.mark.skipif(sys.version_info < (3, 6), reason="don't execute this")
def test_prod():
    assert prod.mul() == 100
    print("-----------Executed Successfully------------")


@pytest.mark.parametrize('num1, num2, result',
                         [
                             (7, 3, 21),
                             (10, 12, 120),
                             (11, 10, 110)
                         ]
                         )
def test_prod1(num1, num2, result):
    prod = Product(num1, num2)
    assert prod.mul() == result
```

In [ ]:
```
### Functionstobetested.py
class Queue:
    def __init__(self):
        self.inbox = Stack()
        self.outbox = Stack()

    def is_empty(self):
        return (self.inbox.is_empty() and self.outbox.is_empty())

    def enqueue(self, data):
        self.inbox.push(data)

    def dequeue(self):
        if self.outbox.is_empty():
            while not self.inbox.is_empty():
                popped = self.inbox.pop()
                self.outbox.push(popped)
        return self.outbox.pop()


class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, data):
        self.items.append(data)

    def pop(self):
        return self.items.pop()


class Product(object):

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

    def mul(self):
        return self.x * self.y
```

Question36:

Why are functions considered first class objects in Python???

https://www.quora.com/Why-are-functions-considered-first-class-objects-in-Python (https://www.quora.com/Why-are-functions-considered-first-class-objects-in-Python)

Question37:

What tools do you use for linting, debugging and profiling?

use of python debugger import pdb for i in range(10): pdb.set_trace() pass c ==> continue s ==> Debug step by step b ==> break/Add breakpoint r ==> return p ==> print multiple variables n ==> Move to next line

Question38:

Give an example of map, filter and reduce over an iterable object?

map example r = map(func, seq) # The first argument func is the name of a function and the second a sequence (e.g. a list) seq. map() # applies the function func to all the elements of the sequence seq.

In [4]:
```
# this example is using lambda function
C = [39.2, 36.5, 37.3, 38, 37.8]
F = list(map(lambda x: (float(9)/5)*x + 32, C))
F
```

Out[4]: [102.56, 97.7, 99.14, 100.4, 100.03999999999999]

In [ ]:
```
# map() can be applied to more than one list.
# map() will apply its lambda function to the elements of the argument lists,
 i.e. it first applies to the elements with the 0th index, then to the element
s with the 1st index until the n-th index is reached:
```

In [5]:
```
a = [1, 2, 3, 4]
b = [17, 12, 11, 10]
c = [-1, -4, 5, 9]
list(map(lambda x, y : x+y, a, b))
```

Out[5]: [18, 14, 14, 14]

In [6]:
```
list(map(lambda x, y, z : x+y+z, a, b, c))
```

Out[6]: [17, 10, 19, 23]

In [7]:
```
list(map(lambda x, y, z : 2.5*x + 2*y - z, a, b, c))
```

Out[7]: [37.5, 33.0, 24.5, 21.0]

In [8]:
```
## If one list has fewer elements than the others, map will stop when the shor
test list has been consumed:
```

In [9]:
```
a = [1, 2, 3]
b = [17, 12, 11, 10]
c = [-1, -4, 5, 9]
list(map(lambda x, y, z : 2.5*x + 2*y - z, a, b, c))
```

Out[9]: [37.5, 33.0, 24.5]

In [10]:
```
# Filtering example
#filter(function, sequence)
```

In [11]:
```
#offers an elegant way to filter out all the elements of a sequence "sequenc
e", for which the function function returns True.
#i.e. an item will be produced by the iterator result of filter(function, sequ
ence) if item is included in the sequence "sequence" and if function(item) ret
urns True.

# In other words: The function filter(f,l) needs a function f as its first arg
ument.
# f has to return a Boolean value, i.e. either True or False. This function wi
ll be applied to every element of the list l.
# Only if f returns True will the element be produced by the iterator, which i
s the return value of filter(function, sequence).
```

In [12]:
```
fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
list(filter(lambda x: x % 2, fibonacci))
```

Out[12]:
```
[1, 1, 3, 5, 13, 21, 55]
```

In [13]:
```
# Reducing a List example
#reduce(func, seq)

#continually applies the function func() to the sequence seq. It returns a sin
gle value.

#If seq = [ s1, s2, s3, ... , sn ], calling reduce(func, seq) works like this:
#At first the first two elements of seq will be applied to func, i.e. func(s1,
s2)
#The list on which reduce() works looks now like this: [ func(s1, s2), s3, ...
, sn ]
#In the next step func will be applied on the previous result and the third el
ement of the list, i.e. func(func(s1, s2),s3)
#The list looks like this now: [ func(func(s1, s2),s3), ... , sn ]
#Continue like this until just one element is left and return this element as
 the result of reduce()
```

In [14]:
```
import functools
functools.reduce(lambda x,y: x+y, [47,11,42,13])
```

Out[14]:
```
113
```

In [15]:
```
f = lambda a,b: a if (a > b) else b
functools.reduce(f, [47,11,42,102,13])
```

Out[15]:
```
102
```

Question39:

What are list and dict comprehensions?

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

In [17]:
```
# without list comprehension
squares = []
for i in range(10):
    squares.append(i**2)   #list append
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [18]:
```
#using list comprehension
squares = [i**2 for i in range(10)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [19]:
```
#create a list of tuples like (number, square_of_number)
new_lst = [(i, i**2) for i in range(10)]
print(new_lst)
```

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64),
(9, 81)]
```

In [22]:
```
# Nested List Comprehensions
#let's suppose we have a matrix

matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]

#transpose of a matrix without list comprehension
transposed = []
for i in range(4):
    lst = []
    for row in matrix:
        print(i,row[i])
        lst.append(row[i])
    transposed.append(lst)

print(transposed)
```

```
0 1
0 5
0 9
1 2
1 6
1 10
2 3
2 7
2 11
3 4
3 8
3 12
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In [23]:
```
#with list comprehension
transposed = [[row[i] for row in matrix] for i in range(4)]
print(transposed)
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In [24]:
```
# Dict Comprehension
#Creating a new dictionary with only pairs where the value is larger than 2
d = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
new_dict = {k:v for k, v in d.items() if v > 2}
print(new_dict)
```

```
{'c': 3, 'd': 4}
```

In [25]:
```
#We can also perform operations on the key value pairs
d = {'a':1,'b':2,'c':3,'d':4,'e':5}
d = {k + 'c':v * 2 for k, v in d.items() if v > 2}
print(d)
```

```
{'cc': 6, 'dc': 8, 'ec': 10}
```

Question40:

What do we mean when we say that a certain Lambda expression forms a closure??

In [28]:
```
#Decorators are an example of closures. For example,

def decorate(f):
    def wrapped_function():
        print("Function is being called")
        f()
        print("Function call is finished")
    return wrapped_function

@decorate
def my_function():
    print("Hello world")

my_function()
```

```
Function is being called
Hello world
Function call is finished
```

In [ ]:
```
#The function wrapped_function is a closure, because it retains access to the
 variables in its scope--in particular,
#the parameter f, the original function. Closures are what allow you to access
it.
```

In [29]:
```
#Closures also allow you to retain state across calls of a function, without h
aving to resort to a class:

def make_counter():
    next_value = 0
    def return_next_value():
        nonlocal next_value
        val = next_value
        next_value += 1
        return val
    return return_next_value

my_first_counter = make_counter()
my_second_counter = make_counter()
print(my_first_counter())
print(my_second_counter())
print(my_first_counter())
print(my_second_counter())
print(my_first_counter())
print(my_second_counter())
```

```
0
0
1
1
2
2
```

In [30]:
```
#The criteria that must be met to create closure in Python are summarized in t
he following points.

#We must have a nested function (function inside a function).
#The nested function must refer to a value defined in the enclosing function.
#The enclosing function must return the nested function.
```

Question41:

What is the difference between list and tuple???

https://www.afternerd.com/blog/difference-between-list-tuple/ (https://www.afternerd.com/blog/difference-between-list-tuple/)

Question42:

What is the MRO in Python????

In [31]:
```
# method resolution order ==> applicable for multiple inheritance
# searching with depth first search order

class A(object):
    def dothis(self):
        print("doing this in A")


class B(A):
    pass


class C(object):
    def dothis(self):
        print("doing this for C")


class D(B,C):
    pass


d_instance = D()
d_instance.dothis()

print(D.mro())
```

```
doing this in A
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class '__
main__.C'>, <class 'object'>]
```

In [32]:
```
# diamond shape inheritance ==> ambiguous
# if we consider depth first search then order should be D->B->A->C->A but thi
s is ambiguous
# so it will remove occurance of first A

class A(object):
    def dothis(self):
        print("doing this in A")

class B(A):
    pass


class C(A):
    def dothis(self):
        print("doing this for C")


class D(B,C):
    pass


d_instance = D()
d_instance.dothis()

print(D.mro())
```

```
doing this for C
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__
main__.A'>, <class 'object'>]
```

Question43:

Is python a case-sensitive language???

Yes, Like most of the widely used programming languages like Java, C, C++, etc, Python is also a case sensitive language.

Languages like Pascal, Basic, Fortran, SQL, Lisp, etc are case in-sensitive.

Question44:

What is the difference between X range and range??

In Python 3, there is no xrange , but the range function behaves like xrange in Python 2.

range() – This returns a list of numbers created using range() function.

xrange() – This function returns the generator object that can be used to display numbers only by looping. Only particular range is displayed on demand and hence called “lazy evaluation“.

In [35]:
```
# initializing a with range()
a = range(1,6)

# initializing a with xrange()
# x = xrange(1,6)

# testing usage of slice operation on range()
# prints without error
print ("The list after slicing using range is : ")
print (a[2:5])

# testing usage of slice operation on xrange()
# raises error
# print ("The list after slicing using xrange is : ")
# print (x[2:5])
```

```
The list after slicing using range is :
range(3, 6)
```

Question45:

Explain how Python does Compile-time and Run-time code checking???

Runtime and compile time are programming terms that refer to different stages of software program development. Compile-time is the instance where the code you entered is converted to executable while Run-time is the instance where the executable is running. The terms "runtime" and "compile time" are often used by programmers to refer to different types of errors too. Compile-time checking occurs during the compile time. Compile time errors are error occurred due to typing mistake, if we do not follow the proper syntax and semantics of any programming language then compile time errors are thrown by the compiler. They wont let your program to execute a single line until you remove all the syntax errors or until you debug the compile time errors. The following are usual compile time errors: Syntax errors Typechecking errors Compiler crashes (Rarely) Run-time type checking happens during run time of programs. Runtime errors are the errors that are generated when the program is in running state. These types of errors will cause your program to behave unexpectedly or may even kill your program. They are often referred as Exceptions . The following are some usual runtime errors: Division by zero Dereferencing a null pointer Running out of memory

Question46:

What is monkeypatching? How can you do it in Python?

In Python, the term monkey patch refers to dynamic (or run-time) modifications of a class or module. In Python, we can actually change the behavior of code at run-time.

In [38]:
```
# monk.py
class A:
    def func(self):
        print("func() is being called")
```

In [42]:
```
#We use above module (monk) in below code and change behavior of func() at run
-time by assigning different value.

def monkey_f(self):
     print("monkey_f() is being called")

# replacing address of "func" with "monkey_f"
A.func = monkey_f
obj = A()

# calling function "func" whose address got replaced
# with function "monkey_f()"
obj.func()
```

```
monkey_f() is being called
```

In [43]:
```
#https://filippo.io/instance-monkey-patching-in-python/
```

Question47:

What is metaclasses in Python?

https://www.python-course.eu/python3_metaclasses.php (https://www.python-course.eu/python3_metaclasses.php)

Question48:

Whenever Python exists Why does all the memory is not de-allocated / freed when Python exits?

https://www.youtube.com/watch?v=arxWaw-E8QQ (https://www.youtube.com/watch?v=arxWaw-E8QQ)

Question49:

what is duck typing?

https://www.quora.com/What-is-Duck-typing-in-Python (https://www.quora.com/What-is-Duck-typing-in-Python)

https://www.youtube.com/watch?v=CuK0g8OFzwo (https://www.youtube.com/watch?v=CuK0g8OFzwo)

https://www.youtube.com/watch?v=x3v9zMX1s4s (https://www.youtube.com/watch?v=x3v9zMX1s4s)

Question50:

what is polymorphism in python?

https://www.youtube.com/watch?v=P1vH3Pfw6BI&list=PLsyeobzWxl7poL9JTVyndKe62ieoN-MZ3&index=56 we can achieve polymorphism using below 4 mechanism: ducktyping operator overloading method overloading method overridingMethod Overriding: two functions with same name and same number of parameters but in different class class Person: def __init__(self, first, last, age): self.firstname = first self.lastname = last self.age = age def __str__(self): return self.firstname + " " + self.lastname + ", " + str(self.age) class Employee(Person): def __init__(self, first, last, age, staffnum): super().__init__(first, last, age) self.staffnumber = staffnum def __str__(self): return super().__str__() + ", " + self.staffnumber x = Person("Marge", "Simpson", 36) y = Employee("Homer", "Simpson", 28, "1007") # Method Overloading is not possible in python but we can apply a trick to get the same functionality # method overloading is two method with same name but different parameters The second definition of f with two parameters redefines or overrides the first definition with one argument. Overriding means that the first definition is not available anymore. This explains the error message: def f(n): return n + 42 def f(n,m): return n + m + 42 f(3,4) 49 f(3) Traceback (most recent call last): File "", line 1, in TypeError: f() takes exactly 2 arguments (1 given) two ways we can write above problem def f(n, m=None): if m: return n + m +42 else: return n + 42 The * operator can be used as a more general approach for a family of functions with 1, 2, 3, or even more parameters: def f(*x): if len(x) == 1: return x[0] + 42 else: return x[0] + x[1] + 42

Question51:

What is JSON? How would convert JSON data into Python data??

How json module stores in python object. Python | JSON dict | Object list | Array tuple | Array str | String int | Number float | Number True | true False | false None | null

In [1]:
```
# lets see to convert string to json object and vice-versa
json_data = """{ "office":
    {"medical": [
      { "room-number": 100,
        "use": "reception",
        "sq-ft": 50,
        "price": 75
      },
      { "room-number": 101,
        "use": "waiting",
        "sq-ft": 250,
        "price": 75
      },
      { "room-number": 102,
        "use": "examination",
        "sq-ft": 125,
        "price": 150
      },
      { "room-number": 103,
        "use": "examination",
        "sq-ft": 125,
        "price": 150
      },
      { "room-number": 104,
        "use": "office",
        "sq-ft": 150,
        "price": 100
      }
    ]},
    "parking": {
      "location": "premium",
      "style": "covered",
      "price": 750
    }
}"""

import json

# Not only can the json.dumps() function convert a Python datastructure to a J
SON string,
# but it can also dump a JSON string directly into a file. Here is an example
 of writing
# a structure above to a JSON file:

# json_string = json.dumps(json_data)
datastore = json.loads(json_data)
print(datastore)
print(type(datastore))  # python store json data as a dictionary

# now you can access elements of json data using below for loop
for i in datastore['office']['medical']:
    print(i['room-number'])

# lets delete sq-ft from json and store new json in a variable

for i in datastore['office']['medical']:
```

```
    del i['sq-ft']

new_string = json.dumps(datastore, indent=2, sort_keys=True)  # put some inden
tation to look better and sort_keys for
# sorting it with key
print(new_string)
```

```
{'office': {'medical': [{'room-number': 100, 'use': 'reception', 'sq-ft': 50,
'price': 75}, {'room-number': 101, 'use': 'waiting', 'sq-ft': 250, 'price': 7
5}, {'room-number': 102, 'use': 'examination', 'sq-ft': 125, 'price': 150},
{'room-number': 103, 'use': 'examination', 'sq-ft': 125, 'price': 150}, {'roo
m-number': 104, 'use': 'office', 'sq-ft': 150, 'price': 100}]}, 'parking':
{'location': 'premium', 'style': 'covered', 'price': 750}}
<class 'dict'>
100
101
102
103
104
{
  "office": {
    "medical": [
      {
        "price": 75,
        "room-number": 100,
        "use": "reception"
      },
      {
        "price": 75,
        "room-number": 101,
        "use": "waiting"
      },
      {
        "price": 150,
        "room-number": 102,
        "use": "examination"
      },
      {
        "price": 150,
        "room-number": 103,
        "use": "examination"
      },
      {
        "price": 100,
        "room-number": 104,
        "use": "office"
      }
    ]
  },
  "parking": {
    "location": "premium",
    "price": 750,
    "style": "covered"
  }
}
```

In [ ]:
```
# Load and store a json file

with open('example.json') as f:
    data = json.load(f)


for i in data['quiz']:
    print(i)


with open('storejson.json', 'w') as f:
    json.dump(data, f, indent=2)
```

In [3]:
```
# Example of realworld data
```

In [2]:
```
import json
from urllib.request import urlopen

with urlopen("http://api.zippopotam.us/us/ma/belmont") as response:
    source = response.read()

data = json.loads(source)
print(data)

print(data['state'])
print(data['places'][1]['post code'])
print(len(data['places']))
```

```
{'country abbreviation': 'US', 'places': [{'place name': 'Belmont', 'longitud
e': '-71.4594', 'post code': '02178', 'latitude': '42.4464'}, {'place name':
'Belmont', 'longitude': '-71.2044', 'post code': '02478', 'latitude': '42.412
8'}], 'country': 'United States', 'place name': 'Belmont', 'state': 'Massachu
setts', 'state abbreviation': 'MA'}
Massachusetts
02478
2
```

Question52:

What does the following code output?

In [13]:
```
def extendList(val, list=[]):
      list.append(val)
      return list
list1 = extendList(10)
list2 = extendList(123,[])
list3 = extendList('a')
list1,list2,list3
```

Out[13]:
```
([10, 'a'], [123], [10, 'a'])
```

You'd expect the output to be something like this: ([10],[123],['a']) Well, this is because the list argument does not initialize to its default value ([]) every time we make a call to the function. Once we define the function, it creates a new list. Then, whenever we call it again without a list argument, it uses the same list. This is because it calculates the expressions in the default arguments when we define the function, not when we call it.

Question53:

Write a regular expression that will accept an email id. Use the re module.?

In [14]:
```
import re
e=re.search(r'[0-9a-zA-Z.]+@[a-zA-Z]+\.(com|co\.in)$','ayushiwashere@gmail.com')
e.group()
```

Out[14]: 'ayushiwashere@gmail.com'

In [15]:
```
#https://data-flair.training/blogs/python-regex-tutorial/
```

In [18]:
```
# match() gives object or None
print(re.match('center','centre'))
```

```
None
```

In [17]:
```
print(re.match('...\w\we','centre'))
```

```
<_sre.SRE_Match object; span=(0, 6), match='centre'>
```

In [19]:
```
# search() is similar to match but return exact matched string or None. stops after first match
```

In [20]:
```
match=re.search('aa?yushi','ayushi')
match.group()
```

Out[20]: 'ayushi'

In [26]:
```
# gives error if no matching for None
match=re.search('^\w*end','Hey! What are your plans for the weekend?')
#match.group()
print(match)
```

```
None
```

In [27]:
```
# findall() : Python regex search() stops at the first match. But Python findall() returns a list of all matches found.
```

In [28]:
```
match=re.findall(r'advi[cs]e','I could advise you on your poem, but you would disparage my advice')
```

In [29]:
```
for i in match:
     print(i)
```

```
advise
advice
```

In [30]:
```
# findall in a file
```

In [ ]:
```
import os
os.chdir('C:\\Users\\lifei\\Desktop')
f=open('Today.txt')
match=re.findall(r'Java[\w]*',f.read())
for i in match:
      print(i)
```

In [31]:
```
# Options in regex

match=re.findall(r'hi','Hi, did you ship it, Hillary?',re.IGNORECASE)
for i in match:
    print(i)
```

```
Hi
hi
Hi
```

In [32]:
```
# Working with a string of multiple lines, this allows ^ and $ to match the st
art and end of each line, not just the whole string.
match=re.findall(r'^Hi','Hi, did you ship it, Hillary?\nNo, I didn\'t, but \
                 Hi',re.MULTILINE)
for i in match:
    print(i)
```

```
Hi
```

Question54:

How many arguments can the range() function take?

In [33]:
```
# The range() function in Python can take up to 3 arguments.
list(range(5))
```

Out[33]:
```
[0, 1, 2, 3, 4]
```

In [34]:
```
list(range(2,7))
```

Out[34]:
```
[2, 3, 4, 5, 6]
```

In [35]:
```
list(range(2,9,2))
```

Out[35]:
```
[2, 4, 6, 8]
```

Question55:

Does Python have a switch-case statement?

In [36]:
```
# Here, you may write a switch function to use. Else, you may use a set of if-elif-else statements.
# To implement a function for this, we may use a dictionary.
```

In [40]:
```
def switch_demo(argument):
    switcher = {
        1: "January",
        2: "February",
        3: "March",
        4: "April",
        5: "May",
        6: "June",
        7: "July",
        8: "August",
        9: "September",
        10: "October",
        11: "November",
        12: "December"
    }
    print(switcher.get(argument, "Invalid month"))
```

In [41]:
```
switch_demo (1)
```

```
January
```

In [42]:
```
# Dictionary mapping for functions
def one():
    return "January"

def two():
    return "February"

def three():
    return "March"

def four():
    return "April"

def five():
    return "May"

def six():
    return "June"

def seven():
    return "July"

def eight():
    return "August"

def nine():
    return "September"

def ten():
    return "October"

def eleven():
    return "November"

def twelve():
    return "December"


def numbers_to_months(argument):
    switcher = {
        1: one,
        2: two,
        3: three,
        4: four,
        5: five,
        6: six,
        7: seven,
        8: eight,
        9: nine,
        10: ten,
        11: eleven,
        12: twelve
    }
    # Get the function from switcher dictionary
    func = switcher.get(argument, lambda: "Invalid month")
```

```
    # Execute the function
    print(func())
```

In [43]:
```
numbers_to_months(1)
```

```
January
```

Question56:

What is a Counter in Python?

In [44]:
```
# The function Counter() from the module 'collections'. It counts the number o
f occurrences of the elements of a container.
from collections import Counter
Counter([1,3,2,1,4,2,1,3,1])
```

Out[44]:
```
Counter({1: 4, 2: 2, 3: 2, 4: 1})
```

Question57:

What is NumPy? Is it better than a list??

NumPy, a Python package, has made its place in the world of scientific computing. It can deal with large data sizes, and also has a powerful N-dimensional array object along with a set of advanced functions. Yes, a NumPy array is better than a Python list. This is in the following ways: It is more compact. It is more convenient. It Is more efficiently. It is easier to read and write items with NumPy.

Question58:

How would you create an empty NumPy array?

In [45]:
```
import numpy
numpy.array([])
```

Out[45]:
```
array([], dtype=float64)
```

In [46]:
```
numpy.empty(shape=(2,2))
```

Out[46]:
```
array([[1.05946444e-311, 1.69506143e+190],
       [1.75184137e+190, 9.48819320e+077]])
```

Question59:

How would you make a Python script executable on Unix?

For this to happen, two conditions must be met: The script file's mode must be executable The first line must begin with a hash(#). An example of this will be: #!/usr/local/bin/python

Question60:

HWhat functions or methods will you use to delete a file in Python?

In [ ]:
```
#For this, we may use remove() or unlink().

import os
os.chdir('C:\\Users\\lifei\\Desktop')
os.remove('try.py')
#or
os.unlink('try.py')
```

Question61:

What are accessors, mutators, and @property?

What we call getters and setters in languages like Java, we term accessors and mutators in Python. In Java, if we have a user-defined class with a property ‘x’, we have methods like getX() and setX(). In Python, we have @property, which is syntactic sugar for property().

In [59]:
```
class Employee(object):
    _emp_id = 0
    _first_name = ''
    _last_name = ''
    _salary = 0
    _age = 19

    def __init__(self, emp_id, first_name, last_name, salary, age):
        self._emp_id = emp_id
        self._first_name = first_name
        self._last_name = last_name
        self._salary = salary
        self._age = age

    @property
    def emp_id(self):
        return self._emp_id

    @emp_id.setter
    def emp_id(self, emp_id):
        self._emp_id = emp_id

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, first_name):
        self._first_name = first_name

    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, last_name):
        self._last_name = last_name

    @property
    def salary(self):
        return self._salary

    @salary.setter
    def salary(self, salary):
        self._salary = salary

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, age):
        if age > 18:
            self._age = age
        else:
            raise ValueError("Enter age greater than 18")
```

```
    def __str__(self):
        return '{}-{}-{}-{}-{}'.format(self._emp_id, self._first_name, self._l
ast_name, self._salary, self._age)


obj = Employee(1, 'abhisek', 'gantait', 10000, 28)
obj.__age = 30

print(obj) ## it will print 28 because you can't set the value outside
# output: 1-abhisek-gantait-10000-28


# MyClass.__private just becomes MyClass._MyClass__private


# A single leading underscore is simply a convention that means, "You probably
shouldn't use this."
# It doesn't do anything to stop someone from using the attribute.

# A double leading underscore actually changes the name of the attribute so th
at two classes in an inheritance
# hierarchy can use the same attribute name, and they will not collide.

# Two underlines in the beginning:
# This one causes a lot of confusion. It should not be used to mark a method a
s private,
# the goal here is to avoid your method to be overridden by a subclass. Let's
 see an example:

class A(object):
    def __method(self):
        print("I'm a method in A")

    def method(self):
        self.__method()


class B(A):
    def __method(self):
        print("I'm a method in B")


a = A()
a.method() #I'm a method in A
b = B()
b.method() #I'm a method in A

#https://hackernoon.com/understanding-the-underscore-of-python-309d1a029edc
```

```
1-abhisek-gantait-10000-28
I'm a method in A
I'm a method in A
```

Question62:

Differentiate between the append() and extend() methods of a list.

The methods append() and extend() work on lists. While append() adds an element to the end of the list, extend adds another list to the end of a list. Let's take two lists.

```
In [67]: list1,list2=[1,2,3],[5,6,7,8]

In [61]: list1.append(4)

In [62]: list1
Out[62]: [1, 2, 3, 4]

In [63]: list1.append(list2)

In [64]: list1
Out[64]: [1, 2, 3, 4, [5, 6, 7, 8]]

In [71]: list1.extend(list2)

In [72]: list1
Out[72]: [1, 2, 3, 4, 5, 6, 7, 8]
```

Question63:

Which methods/functions do we use to determine the type of instance and inheritance?

Here, we talk about three methods/functions- type(), isinstance(), and issubclass().a. type() This tells us the type of object we're working with.

```
In [73]: type(3)
Out[73]: int

In [74]: type(False)
Out[74]: bool

In [75]: type(lambda :print("Hi"))
Out[75]: function
```

b. isinstance() This takes in two arguments- a value and a type. If the value is of the kind of the specified type, it returns True. Else, it returns False.

```
In [76]: isinstance(3,int)
```

```
Out[76]: True
```

```
In [77]: isinstance((1),tuple)
```

```
Out[77]: False
```

c. issubclass() This takes two classes as arguments. If the first one inherits from the second, it returns True. Else, it returns False.

```
In [78]: class A: pass
         class B(A): pass
         issubclass(B,A)
```

```
Out[78]: True
```

Question64:

What is the PYTHONPATH variable?

PYTHONPATH is the variable that tells the interpreter where to locate the module files imported into a program. Hence, it must include the Python source library directory and the directories containing Python source code. You can manually set PYTHONPATH, but usually, the Python installer will preset it.

Question65:

What is a namedtuple?

A namedtuple will let us access a tuple's elements using a name/label. We use the function namedtuple() for this, and import it from collections.

```
In [79]: from collections import namedtuple
         result=namedtuple('result','Physics Chemistry Maths') #format
         Ayushi=result(Physics=86,Chemistry=95,Maths=86) #declaring the tuple
         Ayushi.Chemistry
```

```
Out[79]: 95
```

As you can see, it let us access the marks in Chemistry using the Chemistry attribute of object Ayushi.

Question66:

How do you take input in Python?

For taking input from user, we have the function input(). In Python 2, we had another function raw_input(). The input() function takes, as an argument, the text to be displayed for the task:

```
In [81]: a=input('Enter a number: ')
```

```
         Enter a number: 12
```

In [84]:
```
a = int(input('Enter a number: '))
```

```
Enter a number: 12
```

Question67:

What is a frozen set in Python?

First, we focus on Python sets. A set in Python holds a sequence of values. It is sequenced but does not support indexing.

https://data-flair.training/blogs/python-set-and-booleans-with-examples/ (https://data-flair.training/blogs/python-set-and-booleans-with-examples/)

A frozen set is in-effect an immutable set. You cannot change its values. Also, a set can't be used a key for a dictionary, but a frozenset can.However, a set is mutable. A frozen set is immutable. This means we cannot change its values. This also makes it eligible to be used as a key for a dictionary.

Question68:

How would you generate a random number in Python?

In [87]:
```
#To generate a random number, we import the function random() from the module
 random.
from random import random
random()
```

Out[87]:
```
0.270158566465622
```

In [88]:
```
help(random)
```

```
Help on built-in function random:

random(...) method of random.Random instance
    random() -> x in the interval [0, 1).
```

This means that it will return a random number equal to or greater than 0, and less than 1. We can also use the function randint(). It takes two arguments to indicate a range from which to return a random integer.

In [89]:
```
from random import randint
randint(2,7)
```

Out[89]:
```
2
```

Question69:

How will you capitalize the first letter of a string?

In [91]:
```
#Simply using the method capitalize().
'abhisek'.capitalize()
```

Out[91]: 'Abhisek'

Question70:

How will you check if all characters in a string are alphanumeric?

In [92]:
```
#For this, we use the method isalnum().
'Abhisek123'.isalnum()
```

Out[92]: True

In [93]:
```
'Abhisek123!'.isalnum()
```

Out[93]: False

Question71:

What is recursion?

When a function makes a call to itself, it is termed recursion. But then, in order for it to avoid forming an infinite loop, we must have a base condition. Let's take an example.

In [94]:
```
def facto(n):
    if n==1: return 1
    return n*facto(n-1)
facto(4)
```

Out[94]: 24

Question72:

does generator and iterators are same?

They do, but there are subtle differences: For a generator, we create a function. For an iterator, we use in-built functions iter() and next(). For a generator, we use the keyword 'yield' to yield/return an object at a time. A generator may have as many 'yield' statements as you want. A generator will save the states of the local variables every time 'yield' will pause the loop. An iterator does not use local variables; it only needs an iterable to iterate on. Using a class, you can implement your own iterator, but not a generator. Generators are fast, compact, and simpler. Iterators are more memory-efficient.

Question73:

With Python, how do you find out which directory you are currently in?

In [96]:
```
#To find this, we use the function/method getcwd(). We import it from the modu
le os.
import os
os.getcwd()
```

Out[96]: 'G:\\My Learning Series\\Python Interview Questions'

Question74:

How do you create your own package in Python?

https://data-flair.training/blogs/python-packages/ (https://data-flair.training/blogs/python-packages/)

Question75:

what is deep copy and shallow copy in Python?

Deep copy is a process in which the copying process occurs recursively. It means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original. In case of deep copy, a copy of object is copied in other object. It means that any changes made to a copy of object do not reflect in the original object. In python, this is implemented using “deepcopy()” function.

In [2]:
```
# Python code to demonstrate copy operations

# importing "copy" for copy operations
import copy

# initializing list 1
li1 = [1, 2, [3,5], 4]

# using deepcopy to deep copy
li2 = copy.deepcopy(li1)

# original elements of list
print ("The original elements before deep copying")
for i in range(0,len(li1)):
    print (li1[i],end=" ")
    print("\r")

# adding and element to new list
li2[2][0] = 7

# Change is reflected in l2
print ("The new list of elements after deep copying ")
for i in range(0,len( li1)):
    print (li2[i],end=" ")
    print("\r")

# Change is NOT reflected in original list
# as it is a deep copy
print ("The original elements after deep copying")
for i in range(0,len( li1)):
    print (li1[i],end=" ")
```

```
The original elements before deep copying
1
2
[3, 5]
4
The new list of elements after deep copying
1
2
[7, 5]
4
The original elements after deep copying
1 2 [3, 5] 4
```

A shallow copy means constructing a new collection object and then populating it with references to the child objects found in the original. The copying process does not recurse and therefore won’t create copies of the child objects themselves. In case of shallow copy, a reference of object is copied in other object. It means that any changes made to a copy of object do reflect in the original object. In python, this is implemented using “copy()” function.

In [3]:
```
# Python code to demonstrate copy operations

# importing "copy" for copy operations
import copy

# initializing list 1
li1 = [1, 2, [3,5], 4]

# using copy to shallow copy
li2 = copy.copy(li1)

# original elements of list
print ("The original elements before shallow copying")
for i in range(0,len(li1)):
    print (li1[i],end=" ")

print("\r")

# adding and element to new list
li2[2][0] = 7

# checking if change is reflected
print ("The original elements after shallow copying")
for i in range(0,len( li1)):
    print (li1[i],end=" ")
```

```
The original elements before shallow copying
1 2 [3, 5] 4
The original elements after shallow copying
1 2 [7, 5] 4
```

Question76:

What is the difference between Python Arrays and lists?

Arrays and lists, in Python, have the same way of storing data. But, arrays can hold only a single data type elements whereas lists can hold any data type elements.

In [4]:
```
import array as arr
My_Array=arr.array('i',[1,2,3,4])
My_list=[1,'abc',1.20]
print(My_Array)
print(My_list)
```

```
array('i', [1, 2, 3, 4])
[1, 'abc', 1.2]
```

Question77:

What is __init__?

__init__ is a method or constructor in Python. This method is automatically called to allocate memory when a new object/ instance of a class is created. All classes have the __init__ method.

In [5]:
```
class Employee:
    def __init__(self, name, age,salary):
        self.name = name
        self.age = age
        self.salary = 20000
E1 = Employee("XYZ", 23, 20000)
# E1 is the instance of class Employee.
#__init__ allocates memory for E1.
print(E1.name)
print(E1.age)
print(E1.salary)
```

```
XYZ
23
20000
```

Question78:

Define user define exception?

In [ ]:
```
https://www.programiz.com/python-programming/user-defined-exception
```

In [1]:
```
# define Python user-defined exceptions
class Error(Exception):
   """Base class for other exceptions"""
   pass

class ValueTooSmallError(Error):
   """Raised when the input value is too small"""
   pass

class ValueTooLargeError(Error):
   """Raised when the input value is too large"""
   pass

# our main program
# user guesses a number until he/she gets it right

# you need to guess this number
number = 10

while True:
   try:
       i_num = int(input("Enter a number: "))
       if i_num < number:
           raise ValueTooSmallError
       elif i_num > number:
           raise ValueTooLargeError
       break
   except ValueTooSmallError:
       print("This value is too small, try again!")
       print()
   except ValueTooLargeError:
       print("This value is too large, try again!")
       print()

print("Congratulations! You guessed it correctly.")
```

```
Enter a number: 11
This value is too large, try again!

Enter a number: 17
This value is too large, try again!

Enter a number: 10
Congratulations! You guessed it correctly.
```

Question79:

How to re-raise an exception in nested try/except blocks?

In [2]:
```
#In Python 3 the traceback is stored in the exception, so raise e will do the
 (mostly) right thing:

try:
    something()
except SomeError as e:
    try:
        plan_B()
    except AlsoFailsError:
        raise e
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-8cc3ba3d0bbb> in <module>()
      3 try:
----> 4     something()
      5 except SomeError as e:

NameError: name 'something' is not defined

During handling of the above exception, another exception occurred:

NameError                                 Traceback (most recent call last)
<ipython-input-2-8cc3ba3d0bbb> in <module>()
      3 try:
      4     something()
----> 5 except SomeError as e:
      6     try:
      7         plan_B()

NameError: name 'SomeError' is not defined
```

In [3]:
```
# https://stackoverflow.com/questions/18188563/how-to-re-raise-an-exception-in
-nested-try-except-blocks
```

Question80:

Define define exception in python?

In [ ]:
```
http://tutors.ics.uci.edu/index.php/79-python-resources/104-try-except
```