Python

Python 2 Vs 3:

- Print
- __future__ module

  We have use this module if we want use python 3 futures in python 2
- Integer division

In 2	in 3
3/2 = 1	3/2 = 1.5

- Unicode: python 3 supports unicodes

In 2	in3
str, bytearray	str, bytes, bytearray

- xrange: no xrange() in python 3. range() in 3 is working as lazy evaluation
- Raising exception

raise IOError, “file error”	raise IOError(“file error”)

- Handling exception with ‘as’ in python 3

exception NameError, err	exception NameError as err:

- next() in Python 3

itr.next()	next(itr)

- Comparing unordered types not possible in python 3

[1,2] > ‘foo’	[1,2] > ‘foo’ → error

- input() only no raw_input() in 3
- Rounding to nearest even number

round(16.5) → 17	round(16.5) → 16

- Returning iterable object instead of list.

  Range, zip, map, filter, keys, values, items

→ Closure function:

Closure is nothing but inner function and returning inner function.

```
def multi(n):        #closure function.
    def inne(m):
        return n*m*2
    return inne      #should not give brackets

squa = multi(2)
tripl = multi(3)
squa(6)
tripl(3)
```

→ Decorator: @

We can change behavior of function / class without changing code changes.

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper
```

```
@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")
```

Decorators uses:

- Checking arguments with a decorator
- Counting function calls

→ Class as Decorator

```
class decorator2:

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

    def __call__(self):
        print("Decorating", self.f.__name__)
        self.f()

@decorator2
def foo():
    print("inside foo()")

foo()
```

→ A module is a file in python
→ A package is a directory with __init__.py
→ pdb
https://www.digitalocean.com/community/tutorials/how-to-use-the-python-debugger
→ BOTO3 for AWS
https://boto3.amazonaws.com/v1/documentation/api/latest/guide/quickstart.html#installation

→ Data structures
https://www.datacamp.com/community/tutorials/data-structures-python

IF condition:

value_true if <test> else value_false
e.g: a+b if a!=b else a*b

Another way can be:
[value_false, value_true][<test>]
e.g:
count = [0,N+1][count==N]

Python was created by Guido van Rossum during 1985- 1990.

Multiple Assignment

a = b = c = 1 #same memory location

a, b, c = 1, 2, "john"

XLS file handling

import xlrd

```
#----------------------------------------------------------------------
def open_file(path):
    """
    Open and read an Excel file
    """
    book = xlrd.open_workbook(path)

    # print number of sheets
    print book.nsheets

    # print sheet names
    print book.sheet_names()

    # get the first worksheet
    first_sheet = book.sheet_by_index(0)

    # read a row
    print first_sheet.row_values(0)

    # read a cell
    cell = first_sheet.cell(0,0)
    print cell
    print cell.value

    # read a row slice
    print first_sheet.row_slice(rowx=0,
                                start_colx=0,
                                end_colx=2)

#----------------------------------------------------------------------
if __name__ == "__main__":
    path = "test.xls"
    open_file(path)
```

Multiple inheritance:

`when looking up a method, base classes were searched using a simple depth-first left-to-right scheme.`

Tuples Vs Frozensets

Tuples are immutable lists, frozensets are **immutable** sets.
tuples are indeed an **ordered** collection of objects, but they can contain duplicates and unhashable objects, and have slice functionality
frozensets aren't indexed, but you have the functionality of sets - O(1) element lookups, and functionality such as unions and intersections. They also can't contain duplicates, like their mutable counterparts.

Example for class Vs object level variable:

```
class MyClass:
  static_elem = 123

  def __init__(self):
    self.object_elem = 456

c1 = MyClass()
c2 = MyClass()

# Initial values of both elements
>>> print c1.static_elem, c1.object_elem
123 456
>>> print c2.static_elem, c2.object_elem
123 456

# Nothing new so far ...

# Let's try changing the static element
MyClass.static_elem = 999

>>> print c1.static_elem, c1.object_elem
999 456
>>> print c2.static_elem, c2.object_elem
999 456

# Now, let's try changing the object element
c1.object_elem = 888

>>> print c1.static_elem, c1.object_elem
999 888
>>> print c2.static_elem, c2.object_elem
999 456
```

Multiplay = lambda x : x*2

Multiplay(4)   >>>Output: 8

→ Serialization or pickle: converts into binary

pickle	json
Binary	UTF-8 (text)
Not human readable	Human readable
Python specific	Not related to python
Supports custom classes	Supports python built

```
print x,           # Trailing comma suppresses newline in Python 2
print(x, end=" ")  # Appends a space instead of a newline in Python 3
```

functools.wraps

When you use a decorator, you're replacing one function with another. In other words, if you have a decorator

```
def logged(func):
    def with_logging(*args, **kwargs):
        print func.__name__ + " was called"
        return func(*args, **kwargs)
    return with_logging
```

then when you say

```
@logged
def f(x):
   """does some math"""
   return x + x * x
```

it's exactly the same as saying

```
def f(x):
    """does some math"""
    return x + x * x
f = logged(f)
```

and your function f is replaced with the function with_logging. Unfortunately, this means that if you then say

```
print f.__name__
```

it will print `with_logging` because that's the name of your new function. In fact, if you look at the docstring for f, it will be blank because with_logging has no docstring, and so the docstring you wrote won't be there anymore. Also, if you look at the pydoc result for that function, it won't be listed as taking one argument `x`; instead it'll be listed as taking `*args` and `**kwargs` because that's what with_logging takes.

If using a decorator always meant losing this information about a function, it would be a serious problem. That's why we have `functools.wraps`. This takes a function used in a decorator and adds the functionality of copying over the function name, docstring, arguments list, etc. And since `wraps` is itself a decorator, the following code does the correct thing:

```
def logged(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print func.__name__ + " was called"
        return func(*args, **kwargs)
    return with_logging


@logged
def f(x):
   """does some math"""
   return x + x * x
```

```
print f.__name__  # prints 'f'

print f.__doc__   # prints 'does some math'
```

----

6) What are the tools that help to find bugs or perform static analysis?

PyChecker is a static analysis tool that detects the bugs in Python source code and warns about the style and complexity of the bug. Pylint is another tool that verifies whether the module meets the coding standard.

→ >>python -c print('hai')   # we can run commands in CMD
→ >>python -m <module_name> argv…   # we can give arg to module and run it
→ pyinstaller is a tool it is used to make python scripts executable
→ In interactive mode, the last printed expression is assigned to the variable `_`.

```
>>> 1+2
3
>>> 1+_
4
```

→ String:- Note that since -0 is the same as 0, negative indices start from -1.
→ want to change list items by looping with in list use [:]

```
for w in words[:]:  # Loop over a slice copy of the entire list.

for w in words:
```

→ list comprehension

```
>>> a=[1,2,3,1,3,2,1,1]
>>> [4 if x==1 else x for x in a]
[4, 2, 3, 4, 3, 2, 4, 4]
```

→ `range(0, 10, 3)`

```
  0, 3, 6, 9
```

→ Iterable: we can obtain successive items until supply is exhausted.
→ The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action
→ don't

```
def f(a, L=[]):
        L.append(a)
        return L

Do as
```

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

→
```
>>> list(range(3, 6))            # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))      # call with arguments unpacked from a list
[3, 4, 5]
```

→
```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

→ Annotations
```
def greet(name: str, age: int) -> str:
    print('Hello {0}, you are {1} years old'.format(name, age))
```

→ list.copy() #Return a shallow copy of the list. Equivalent to `a[:]`.
→ stack by using list means using pop()
→ Queues in list means use collections.deque

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
```

→ list comprehensions
A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses.
```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

```
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

→ Python does not check the cache in two circumstances. First, it always recompiles and does not store the result for the module that's loaded directly from the command line. Second, it does not check the cache if there is no source module. To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

→ Relative imports:

```
from . import echo
from .. import formats
```

→ `'!a'` (apply ascii()), `'!s'` (apply str()) and `'!r'` (apply repr()) can be used to convert the value before it is formatted:

```
>>> print('My hovercraft is full of {!r}.'.format(contents))
My hovercraft is full of 'eels'.
```

→ Ternary operator

V = `[on_true] if [expression] else [on_false]`

→ Polymorphism:

We can implement polymorphism by using abstract calls.

```
class Car:
  def __init__(self, name):
    self.name = name

  def drive(self):
    raise NotImplementedError("Subclass must implement abstract method")
    def stop(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Sportscar(Car):
    def drive(self):
        return 'Sportscar driving!'

    def stop(self):
        return 'Sportscar braking!'

class Truck(Car):
    def drive(self):
        return 'Truck driving slowly because heavily loaded.'

    def stop(self):
        return 'Truck braking!'


cars = [Truck('Bananatruck'),
        Truck('Orangetruck'),
        Sportscar('Z3')]

for car in cars:
    print car.name + ': ' + car.drive()
```

→ String passing to print:

There are many ways to do this. To fix your current code using `%`-formatting, you need to pass in a tuple:

> Pass it as a tuple:
> print("Total score for %s is %s" % (name, score))
> A tuple with a single element looks like `('this',)`.

Here are some other common ways of doing it:

> Pass it as a dictionary:
> print("Total score for %(n)s is %(s)s" % {'n': name, 's': score})

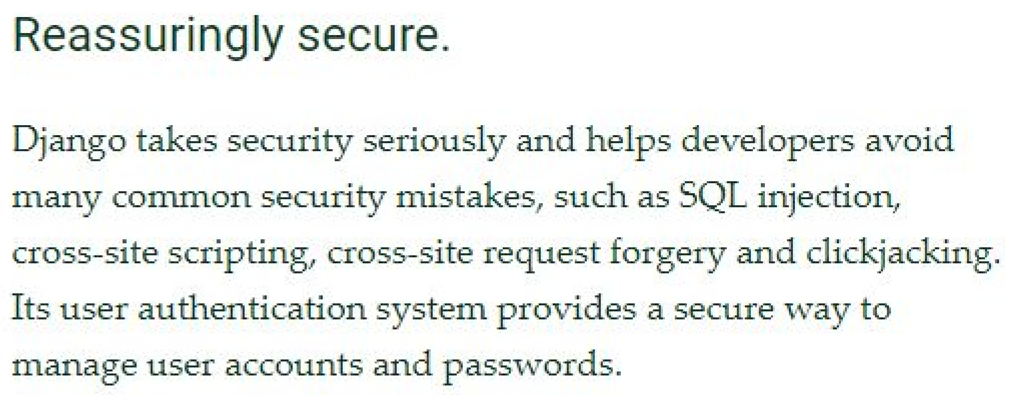
There's also new-style string formatting, which might be a little easier to read:

> -Use new-style string formatting:
> print("Total score for {} is {}".format(name, score))
> -Use new-style string formatting with numbers (useful for reordering or printing the same one multiple times):

```
print("Total score for {0} is {1}".format(name, score))
```
-Use new-style string formatting with xpliceit names:
```
print("Total score for {n} is {s}".format(n=name, s=score))
```
-Concatenate strings:
```
print("Total score for " + str(name) + " is " + str(score))
```
The clearest two, in my opinion:

-Just pass the values as parameters:
```
print("Total score for", name, "is", score)
```
-If you don't want spaces to be inserted automatically by print in the above example, change the sep parameter:
```
print("Total score for ", name, " is ", score, sep='')
```
-If you're using Python 2, won't be able to use the last two because print isn't a function in Python 2. You can, however, import this behavior from __future__:
```
from __future__ import print_function
```
-Use the new f-string formatting in Python **3.6**:
```
print(f'Total score for {name} is {score}')
```

Django



Reassuringly secure.

Django takes security seriously and helps developers avoid many common security mistakes, such as SQL injection, cross-site scripting, cross-site request forgery and clickjacking. Its user authentication system provides a secure way to manage user accounts and passwords.

```
$ django-admin startproject mysite
$ python manage.py runserver
```

→ Pyodbc : http://www.sqlrelease.com/connecting-python-3-to-sql-server-2017-using-pyodbc

→ CSV: http://www.pythonforbeginners.com/systems-programming/using-the-csv-module-in-python/
https://code.tutsplus.com/tutorials/how-to-read-and-write-csv-files-in-python--cms-29907
→ Json:
https://www.safaribooksonline.com/library/view/python-cookbook-3rd/9781449357337/ch06s02.html

→ Web Server Gateway Interface (WSGI)
Pymongo:
http://api.mongodb.com/python/current/tutorial.html

MongoDB stores data in BSON format.

```
>>>Import pymongo
>>>From pymongo import MongoClient  #making connection with mongoclient
client = MongoClient('localhost', 27017)
>>>db = client.test_database                    #accessing database
        OR
>>>db = client['test-database']
>>>collection = db.test_collection  OR  db['test_collection']
# data stores as documents(Records) and those are like dictionaries format
>>> post = {"author":"mike", "text":"My first blog",....}
>>>posts = db.posts
>>> Post_id = posts.insert_one(post).insert_id
>>> post_id
objectId('...')
#List of collections in database
>>> db.collection_names(include_system_collections=False)
[u'posts']
# find one record
>>> import pprint
>>> pprint.pprint(posts.find_one())
{u'_id': ObjectId('...'),
 u'author': u'Mike',
 u'date': datetime.datetime(...),
 u'tags': [u'mongodb', u'python', u'pymongo'],
 u'text': u'My first blog post!'}
>>> pprint.pprint(posts.find_one({"author": "Mike"}))
{u'_id': ObjectId('...'),
>>> pprint.pprint(posts.find_one({"_id": post_id}))
```

Note that an ObjectId is not the same as its string representation:

```
>>> post_id_as_str = str(post_id)
>>> posts.find_one({"_id": post_id_as_str}) # No result
```
convert the ObjectId from a string before passing it to `find_one`:

```
from bson.objectid import ObjectId

# The web framework gets post_id from the URL and passes it as a string
def get(post_id):
    # Convert from string to ObjectId:
    document = client.db.collection.find_one({'_id': ObjectId(post_id)})
```

Bulk Inserts

```
>>> new_posts = [{"author": "Mike",
...               "text": "Another post!",
...               "tags": ["bulk", "insert"],
...               "date": datetime.datetime(2009, 11, 12, 11, 14)},
...              {"author": "Eliot",
...               "title": "MongoDB is fun",
...               "text": "and pretty easy too!",
...               "date": datetime.datetime(2009, 11, 10, 10, 45)}]
>>> result = posts.insert_many(new_posts)
>>> result.inserted_ids
[ObjectId('...'), ObjectId('...')]
```

Querying for more than one document:

```
>>> for post in posts.find():
...   pprint.pprint(post)
```

Counting:

```
>>> posts.count()
3
```

or just of those documents that match a specific query:

```
>>> posts.find({"author": "Mike"}).count()
2
```
Indexing:

Adding indexes can help accelerate certain queries and can also add additional functionality to querying and storing documents.

```
>>> result = db.profiles.create_index([('user_id', pymongo.ASCENDING)],
...                                   unique=True)
>>> sorted(list(db.profiles.index_information()))
[u'_id_', u'user_id_1']
```

https://intellipaat.com/interview-question/python-interview-questions/
https://career.guru99.com/top-25-python-interview-questions/
https://www.tutorialspoint.com/python/python_interview_questions.htm
https://www.edureka.co/blog/interview-questions/python-interview-questions/
https://www.mytectra.com/interview-question/python-real-time-interview-questions-and-answers/
https://mindmajix.com/python-interview-questions