10 Advanced Python Concepts To Level Up Your Python Skills

https://levelup.gitconnected.com/10-advance-python-concepts-to-level-up-your-python-skills-da3d6284ad53

[1] Exception Handling

```
import sys
try:
  f = open('myfile.txt')
  s = f.readline()
  i = int(s.strip())
except OSError as err:
  print("OS error: {0}".format(err))
except ValueError:
  print("Could not convert data to an integer.")
except:
  print("Unexpected error:", sys.exc_info()[0])
  raise
finally:
  print("Operation Successfully Done!!")(Example Taken From Official Python Docs)
```

[2] Collections

- **Counter**

```
from collections import Counter
data = [1,1,1,1,2,3,4,3,3,5,6,7,7]
count = Counter(data)
print(count)
--------------------------------------------------
Counter({1: 4, 3: 3, 7: 2, 2: 1, 4: 1, 5: 1, 6: 1})
```

list(count.elements( )) #returning to original list from Counter object
>> [1, 1, 1, 1, 2, 3, 3, 3, 4, 5, 6, 7, 7]

count.most_common( ) #list of tuples with item and its counts in most frequent to less frequent order
>> [(1, 4), (3, 3), (7, 2), (2, 1), (4, 1), (5, 1), (6, 1)]

- namedtuple

```
from collections import namedtuple
Direction = namedtuple('Direction','N,S,E,W')
dt = Direction(4,74,0,0)
print(dt)
-----------------------------------------------
Direction(N=4, S=74, E=0, W=0)
```

dt.N
>> 4

dt[1]
>> 74

getattr(dt, 'E')
>> 0

You can also define namedtuple like this:
pets=namedtuple('pets',['name','age'])
Fluffy=pets('Fluffy',8)
Fluffy
>> pets(name='Fluffy', age=8)

- **OrderedDict**

```
from collections import OrderedDict
dictt = OrderedDict()
dictt['a'] = 5
dictt['d'] = 2
dictt['c'] = 1
dictt['b'] = 3
print(dictt)
-----------------------------------------------------
OrderedDict([('a', 5), ('d', 2), ('c', 1), ('b', 3)])
```

- **defaultdict**

```
from collections import defaultdict
dictt = defaultdict(int)
dictt['a'] = 2
print(dictt['a']) ## return the value
print(dictt['b']) ## returns the default value
------------------------------
2
0
```

- **deque**

import collections
de = collections.deque([1,2,3])

append( ): add element at the right side
appendleft( ): add element at the left side
pop( ): remove element from the right side
popleft( ): remove element from the left side

[3] Itertools

- **product(iterable,iterable)**: cartesian product of two iterables
~~- **permutation(iterable)**: all possible ordering with no repeated elements → not in itertools~~
- **combinations(iterable,n)**: all possible combinations with specified length with no repetition. here **n** is the size of the combination tuple.
- **combinations_with_replacement(iterable,n)**: all possible combinations with specified length with repetition.
~~- **accumlate(iterable)** : returns accumulate the sum of elements of iterable. → not in itertools~~
- **groupby(iterable,key=FUNC)** : return an iterator with consecutive keys and groups from the iterable.

from itertools import combinations,product, combinations_with_replacement, groupby
a = [1,2,3]

print(list(combinations(a,2)))
>> [(1, 2), (1, 3), (2, 3)]

list(combinations_with_replacement(a,2))
>> [(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]

list(product(a,a))
>> [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]

[4] Lambda

```
even_or_odd = lambda a: a%2==0
numbers = [1,2,3,4,5]
even = list(map(even_or_odd,numbers))
print(even)
--------------------------------
[False, True, False, True, False]
```

[5] Decorators

A decorator is a feature in python that adds some new functionality to existing code without explicitly modifying it

Two Types of Decorators
- decorator function: has an @ before the function name.
- class decorators

e.g. adding division and multiplication features to the add ( ) function without explicitly modifying the add( )function

```
import functools
def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        a,b = args
        print(a*b)
        result = func(*args,**kwargs)
        print(a/b)
        return result
    return wrapper

@decorator
def add(x,y):
    return x+y
result = add(5,6)
print(result)
```

```
>>
30
0.8333333333333334
11
```

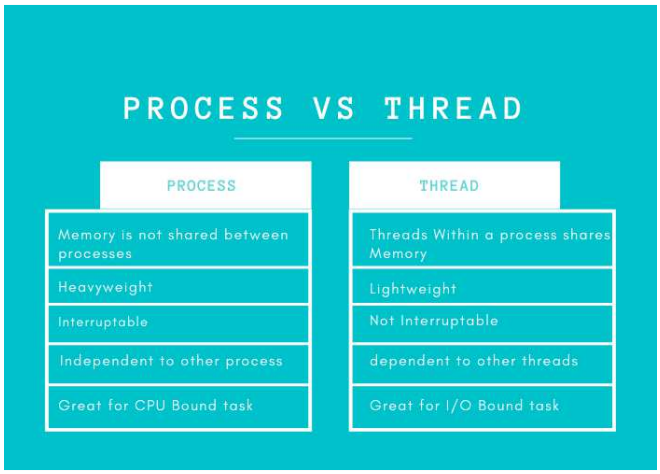
[6] Generators

Generators are a kind of function that returns an object that can be iterated over. It contains at least a yield statement. yield is a keyword in python that is used to return a value from a function without destroying its current state or reference to a local variable. A function with a yield keyword is called a generator. Memory efficient and takes less space in the memory.

```
-------------- Fibonacci Series Using Generators ------------
def fibon(limit):
  a,b = 0,1
  while a < limit:
     yield a
     a, b = b, a + bfor x in fibon(10):
   print (x)
```

[7] Threading and MultiProcessing

(In a separate doc in the same folder – get advice from Harry)



[8] Dunder Methods

Dunder methods or Magic methods are those that have two underscores __ before and after the method. These methods are invoked directly invoke from the class on a certain action. When you try to multiply two numbers using * a sign then the internal __mul__ method is called.

```
num = 5
num*6
>> 30num.__mul__(6)
>>30
```

[9] Logging

https://medium.com/pythoneers/master-logging-in-python-73cd2ff4a7cb

- Debug: Used for diagnosing the problem with detailed information. Severity level 10
- Info: Confirmation of success. Severity level 20
- Warning: when an unexpected situation occurs. Severity level 30
- Error: Due to a more serious problem than a warning. Severity level 40
- Critical: Critical error after which the program can't run itself. Severity level 50

```
import logging

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')

-------------------OUTPUT---------------------
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

➔ There are only three log messages printed (Under -----Output------). The reason behind this is the severity level of log messages and messages higher than or equal to the severity level of WARNING will only get printed.

We can Print the DEBUG and INFO message too by changing the basic configuration of the logger with the help of basicConfig(**kwargs)

There are some parameters that are commonly used in this —
- level: To change the root logger to a specified severity level.
- filename: Filename where the logs going to be stored.
- filemode: If a filename is given then this specifies the file mode in which the file will open. default is append (a)
- format: This is the format of the log message.
- datefmt : It specified the date and time format.

```
import
logging
        logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(name)s - %(levelname)s
        - %(message)s', datefmt='%d/%m/%Y %H:%M:%S')
        logging.debug('This is Debug Message')
        logging.info('This is an info message')
        -------------------------CONSOLE OUTPUT------------------
        12/05/2021 20:46:41 - root - DEBUG - This is Debug Message
        12/05/2021 20:46:41 - root - INFO - This is an info message
```

 # logging the result in a file you can add filename and file mode to basic Config.
`logging.basicConfig(level=logging.DEBUG,filename='data.log', filemode='w')`

 # Adding dynamic variable data to log

```
import
logging
        logging.basicConfig(level=logging.DEBUG,filename='data.log', filemode='a', format='%(asctime)s
        - %(name)s - %(levelname)s - %(message)s', datefmt='%d/%m/%Y %H:%M:%S')
        name = str(input("Enter Your Name:\n"))
        logging.info(f"{name} has logged in successfully !!")
        -----------------data.log file output-------------
        12/05/2021 21:01:37 - root - INFO - Abhay Parashar has logged in successfully !!
        12/05/2021 21:02:47 - root - INFO - Karl has logged in successfully !!
        12/05/2021 21:03:27 - root - INFO - Rahul has logged in successfully !!
```

➔ logging A Dynamic Variable Data onto a File, After executing it a few times the logs keeps appending on the log file `**data.log**`.

 # Logging Exeption

```
import
logging
        a = 10
        b = 0
        try:
          c = a / b
        except Exception as e:
          logging.error("Exception Occurred", exc_info=True)  ## At default it is True
         ----------------------------------CONSOLE OUTPUT-----------------------
        ERROR:root:Exception Occurred       ## If exc_info=False then only this message will print
        Traceback (most recent call last):
          File "C:\Users\abhay\Desktop\ds\python\advance_concepts\logging_code.py", line 5, in <module>
            c = a / b
        ZeroDivisionError: division by zero
```

Other logging concepts…
Handler
Configuration file
Rotating File Handler
TimeRotating File Handler
More in ➔ ➔ https://medium.com/pythoneers/master-logging-in-python-73cd2ff4a7cb

[10] Context managers

Context Managers are a great tool in python that help in resource management. They allow you to allocate and release resources when you want to. The most used and recognized example of context manager is with statement. with is mostly used to open and close a file.

```
file = open('data.txt','w')
try:
file.write("Hello")
except:
file.close()
```