Python's New Features Batch:

Python 3.8 ===>Oct 14th 2019

Python 3.9 ==>October 5th 2020

More flexibility to the programmer.

1. The Walrus Operator:

:=

Python 3.8

This operator released as the part of PEP 572.

PEP-->Python Enhancement Proposals

To assign values to the variables as the part of expression itself.

Assignment Expressions

```
l = [10,20,30,40,50]
n = len(l)
if n > 3:
        print('List contains more than 3 elements')
        print('The length of the list is:',n)
```

```
l = [10,20,30,40,50]
```

```
if len(l) > 3:
        print('List contains more than 3 elements')
        print('The length of the list is:',len(l))
```

```
l = [10,20,30,40,50]
if (n := len(l)) > 3:
        print('List contains more than 3 elements')
        print('The length of the list is:',n)

print(n)
```

```
heroines=[]
heroine=input('Enter Your Favourite Heroine:')
while heroine != 'done':
        heroines.append(heroine)
        heroine=input('Enter Your Favourite Heroine:')
print(heroines)
```

```
heroines=[]
```

```
while (heroine:=input('Enter Your Favourite Heroine:')) != 'done':
        heroines.append(heroine)
print(heroines)
```

```
# read data line by line from abc.txt file and print to the console.

f=open('abc.txt')
line=f.readline()
while line != '':
        print(line,end='')
        line=f.readline()
f.close()
```

```
# read data line by line from abc.txt file and print to the console.

f=open('abc.txt')
while (line := f.readline()) != '':
        print(line,end='')
f.close()
```

The main advantage of the walrus operator:

It won't do any new thing.

It just reduces length of the code and readability will be improved.

Title: Python New Features- The Walrus Operator

Positional-Only Parameter:

Python 3.8 Version

PEP 570

Functions:

----

1. Positional arguments

2. keyword arguments

3. default arguments
4. variable length arguments

1. Positional arguments:
order and number both are important

2. keyword arguments:
by keyword(parameter name)
The number of arguments must be matched
order is not important.

3. default arguments:
argumens with default values
after default argument, we cannot take non-default arguemnts

4. variable length arguments
and all values will be converted into tuple

f1(*args) and f1(kwargs)**

f1(a,b,*args,kwargs):**

keyword only parameters:

After *, all paramters will become keyword only parameters.

At the time of calling we should pass values by keyword only.

```
def f1(*,a,b):
    print(a,b)

f1(a=10,b=20)     # valid
#f1(10,20) #TypeError: f1() takes 0 positional arguments but 2 were given
f1(10,b=20) #TypeError: f1() takes 0 positional arguments but 1 positional argument (and 1 keyword-only argument) were given
```

Python 3.0 version only.

what about def f1(a,*,b,c)

```
def f1(a,*,b,c):
    print(a,b,c)
```

f1(10,b=20,c=30) #valid
f1(10,20,c=30) #invalid

 # for a, we can pass value either by positional or keyword
 # But for b and c, compulsory we should use keyword only

def f1 (*,a=10,b=20)

def f1 (*,a=10,b=20):
**   print(a,b)**

f1()
f1(a=10)

def f1 (*,a=10,b,c)

keyword only arguments.

positional only arguments:

We should pass values by positional only arguments.

/-->forward slash

All parameters before /, will become positional only parameters.

```
def f1(a,b,/):
        print(a,b)

f1(10,20)
f1(a=10,b=20) #TypeError: f1() got some positional-only arguments passed as keyword arguments: 'a, b'
```

Python 3.8 version as the part PEP 570

```
# a and b are positional only parameters
# c and d are positional-or-keyword parameters
# e and f are keyword only parameters
def f1(a,b,/,c,d,*,e,f):
        print(a,b,c,d,e,f)

f1(10,20,30,d=40,e=50,f=60)
#f1(10,b=20,c=30,d=40,e=50,f=60) #TypeError: f1() got some positional-only arguments passed as keyword arguments: 'b'
f1(10,20,30,40,50,f=60) #TypeError: f1() takes 4 positional arguments but 5 positional arguments (and 1 keyword-only argument) were given
```

sir what happen is fun(*, a ,b , c,/)

```
D:\durgaclasses>py test.py
  File "D:\durgaclasses\test.py", line 1
    def f1(*, a ,b , c,/):
                   ^
SyntaxError: invalid syntax
```

f1(positional arguments, positional-or-keyword arguments,keyword-only arguments)

Problems without positional only arguments:

----

case-1:

without effecting client, we cannot change parameter names based on our requirement.

```
def display(fullname,age,rollno):
        print('Name:',fullname)
        print('Age:',age)
        print('RollNo:',rollno)
```

display('Ravi',14,101) # client/user
display(name='Ravi',age=14,rno=101) # client/user
display('Ravi',14,rno=101) # client/user

If arguments are positional only, then we can change variable names.

```
def display(fullname,age,rollno,/):
    print('Name:',fullname)
    print('Age:',age)
    print('RollNo:',rollno)
```

display('Ravi',14,101) # client/user

case-2:
Assume parameter names are secured, we don't want to expose to the outside world.
positional only parameters.

3. OOP

```
class Parent:
        def m1(self,a,b):
                print('Parent Method:',a+b)

class Child(Parent):
        pass

c=Child()
c.m1(10,20)
```

```
D:\durgaclasses>py test.py
Parent Method: 30
```

While overriding in the child class,
the arguments need not be same

```
class Parent:
        def m1(self,a,b,/):
                print('Parent Method:',a+b)

class Child(Parent):
        def m1(self,x,y,/):
                print('Child Method:',x+y)

c=Child()
```

c.m1(10,20)

4. performance improvement

5.

python inbuilt funciton xyz(a,b,c):

This python inbuilt function is implemented in C language
xyz(a,b,c):
calling C function
c(10,20,30)

xyz(10,20,30)
xyz(a=10,b=20,c=30)
xyz(10,20,c=30)
xyz(10,b=20,c=30)
xyz(c=30,a=10,b=20)

1. Without effecting callers, we can change function parameter names.

2. We are not required to expose our internal parameter names to the outside world.Hence we will get security
3. While overriding parent class method in child class, we are not required to use same parameter names.
4. Performance will be improved.
5. More compatibility between python function calls and internal c language calls.

positional only vs keyword only:
----
1. If the parameter names are not important and not having any meaning and there are only few arguments===>positional only arguments

2. If parameter names having meaning and function implementation is more understandable with these names===>keyword only arguments

/
***

Title: Python New Features- The Positional Only Parameter(/)

1. The Walrus Operator (:=) --->3.8Version
2. The Positional only parameter --->3.8 version
def f(a,b,/,c,d,,*,e,f):
Keyword only parameters--->3.0 version

3. f-strings or formatted strings or Literal String Interpolation (Python 3.6v)

(Python 3.6v)
PEP--->498

String formatting means inserting values and expressions in string literal.

3 types of techniques
1. %-formatting
2. str.format() method
3. f-strings

1. %-formatting:

It is the oldest way of string formatting.
It is available from beginning of the python

%i --->signed int value
%d --->signed int value
%f --->float value
%s --->string type

eg-1: Single variable

```
name='Durga'
s = 'Hello %s,Good Evening' %name
print(s) #Hello Durga,Good Evening
```

eg-2: Multiple Variables

```
name='Durga'
salary=10000
s = 'Hello %s,Your Salary is %i' %(name,salary)
print(s)
print('Hello %s,Your Salary is %i' %(name,salary))
```

o/p:

```
Hello Durga,Your Salary is 10000
Hello Durga,Your Salary is 10000
```

problems with %-formatting:

---

1. If more number of variables are there, then it is more verbose and more error prone.

```
name='Aaradhya'
father_name='Abhishek'
mother_name='Aiswarya'
gf_name='Big B'
subject='Python'

s='Hello %s,You are most luckiest girl as you have %s,%s and %s as family members,You can learn %s very easily'
%(name,father_name,mother_name,gf_name,subject)

print(s)
```

```
Hello Aaradhya,You are most luckiest girl as you have Abhishek,Aiswarya and Big B as family members,You can learn Python very easily
```

2. To process data from the dictionary is complex

```
student = {
        'name':'Aaradhya',
        'father_name':'Abhishek',
        'mother_name':'Aiswarya',
        'gf_name':'Big B',
```

```
        'subject':'Java'
}

s='Hello %s,You are most luckiest girl as you have %s,%s and %s as
family members,You can learn %s very easily'
%(student['name'],student['father_name'],student['mother_name'],st
udent['gf_name'],student['subject'])

print(s)
```

Hello Aaradhya,You are most luckiest girl as you have Abhishek,Aiswarya and Big B as family members,You can learn Java very easily

3. performance wise also not upto the mark.

To overcome some of these problems, to provide more options to the programmer we should go for str.format() method.

String formatting by using str.format() method:

It is introduced in Python 2.6 version
It is advanced version of %-foramatting technique.
It is very easy and provides several options than %-formatting technique.

eg1: Single Variable

```
name='Durga'
```

```
s='Hello {}, Good Morning'.format(name)
print(s)
print('Hello {}, Good Morning'.format(name))
```

```
Hello Durga, Good Morning
Hello Durga, Good Morning
```

eg2: Multiple variables

```
name='Durga'
salary=10000
gf='Sunny'
s='Hello {},Your Salary is {},Your Girl Friend:{} is
waiting'.format(name,salary,gf)
print(s)
```

```
Hello Durga,Your Salary is 10000,Your Girl Friend:Sunny is waiting
```

```
name='Durga'
salary=10000
gf='Sunny'
s1='Hello {},Your Salary is {},Your Girl Friend:{} is
waiting'.format(name,salary,gf)
s2='Hello {2},Your Salary is {1},Your Girl Friend:{0} is
waiting'.format(gf,salary,name)
s3='Hello {n},Your Salary is {s},Your Girl Friend:{g} is
waiting'.format(n=name,s=salary,g=gf)
```

print(s3)

multiple options to the programmer, only {} we have to use. order is not important.

By using str.format() technique, we can use dictionary data very easily.

```
student = {
        'name':'Aaradhya',
        'father_name':'Abhishek',
        'mother_name':'Aiswarya',
        'gf_name':'Big B',
        'subject':'Java'
}

s='Hello {name},You are most luckiest girl as you have {father_name},{mother_name} and {gf_name} as family members,You can learn {subject} very easily'.format(**student)
print(s)
```

Hello Aaradhya,You are most luckiest girl as you have Abhishek,Aiswarya and Big B as family members,You can learn Java very easily

problems with str.format() technique:

----

1. If number of variables are more still it is more verbose.

```
name='Aaradhya'
father_name='Abhishek'
mother_name='Aiswarya'
gf_name='Big B'
subject='Python'

s='Hello {n},You are most luckiest girl as you have {f},{m} and {g} as family members,You can learn {s} very easily'.format(n=name,f=father_name,m=mother_name,g=gf_name,s=subject)
print(s)
```

Hello Aaradhya,You are most luckiest girl as you have Abhishek,Aiswarya and Big B as family members,You can learn Python very easily

2. Performance wise also not upto the mark

To overcome these problems we have to go for f-strings:

It is introduced in Python 3.6 version.

As the part of PEP 498.

The syntax is similar to str.format()

It is more concise and more readable.

Performance is more when compared with other 2 techniques.

eg1: Single variable

```
name='Durga'
s1= 'Hello %s, Good Evening' %name
s2= 'Hello {}, Good Evening'.format(name)
s3= f'Hello {name}, Good Evening'
print(s3)
print(f'Hello {name}, Good Evening')
```

```
Hello Durga, Good Evening
Hello Durga, Good Evening
```

```
name='Durga'
salary=10000
gf='Sunny'

s1='Hello %s,Your Salary is %d and Your Girl Friend %s is waiting'
%(name,salary,gf)
s2='Hello {},Your Salary is {} and Your Girl Friend {} is
waiting'.format(name,salary,gf)
s3=f'Hello {name},Your Salary is {salary} and Your Girl Friend {gf} is
waiting'
print(s3)
```

```
Hello Durga,Your Salary is 10000 and Your Girl Friend Sunny is waiting
```

Note: we can use either f or F, we can use single quotes or double quotes or triple quotes also.

```
name='Durga'
salary=10000
gf='Sunny'

s1=f'Hello {name},Your Salary is {salary} and Your Girl Friend {gf} is waiting'
s2=F"Hello {name},Your Salary is {salary} and Your Girl Friend {gf} is waiting"
s3=F'''Hello {name},Your Salary is {salary} and Your Girl Friend {gf} is waiting'''
s4=F"""Hello {name},Your Salary is {salary} and Your Girl Friend {gf} is waiting"""
print(s1)
print(s2)
print(s3)
print(s4)
```

```
Hello Durga,Your Salary is 10000 and Your Girl Friend Sunny is waiting
Hello Durga,Your Salary is 10000 and Your Girl Friend Sunny is waiting
Hello Durga,Your Salary is 10000 and Your Girl Friend Sunny is waiting
Hello Durga,Your Salary is 10000 and Your Girl Friend Sunny is waiting
```

title: f-strings (formatted strings) 3.6 Version Enhancement Part-1

string formatting techniques

1. **%-formatting**
2. **str.format()**
3. **f-string**

concise code,readability,speed

```
name='Durga'
salary=10000
gf='Sunny'
print('Hello %s, Your Salary is %d,Your Girl Friend %s is waiting'
%(name,salary,gf))
print('Hello {}, Your Salary is {},Your Girl Friend {} is
waiting'.format(name,salary,gf))
print(f'Hello {name}, Your Salary is {salary},Your Girl Friend {gf} is
waiting')
```

timeit module:

By using timeit module, we can measure execution time of small coding snippets.

```
import timeit
```

```
t = timeit.timeit("print('Hello')",number=10000)
print(f'The Time Taken {t} seconds')
```

```
import timeit
t = timeit.timeit('''
name='Durga'
salary=10000
gf='Sunny'
s='Hello %s, Your Salary is %d,Your Girl Friend %s is waiting'
%(name,salary,gf)
''',number=100000)
print('The Time Taken:',t)

t = timeit.timeit('''
name='Durga'
salary=10000
gf='Sunny'
s='Hello {}, Your Salary is {},Your Girl Friend {} is
waiting'.format(name,salary,gf)
''',number=100000)
print('The Time Taken:',t)
```

```
t = timeit.timeit('''
name='Durga'
salary=10000
```

```
gf='Sunny'
s=f'Hello {name}, Your Salary is {salary},Your Girl Friend {gf} is waiting'
''',number=100000)
print('The Time Taken:',t)
```

output:

```
The Time Taken: 0.056381377999999996
The Time Taken: 0.07781707
The Time Taken: 0.037445747
```

Handling Quotes in f-string:

The symbol " is good

```
print(f'The symbol " is good')
print(f"The symbol ' is good")
print(f"The symbols \' and \" are good")
print(f'''The symbols ' and " are good''')
```

```
name='Durga'
subject='Python'
print(f'''The classes of '{subject}' by "{name}" are too good ''')
```

The classes of 'Python' by "Durga" are too good

Processing Dictionary data by f-strings:

```
student = {
        'name':'Aaradhya',
        'father_name':'Abhishek',
        'mother_name':'Aiswarya',
        'gf_name':'Big B',
        'subject':'Python'
}
s = f"Hello {student['name']}, You are the most luckiest girl as you have {student['father_name']},{student['mother_name']} and {student['gf_name']} as family members, You can learn {student['subject']} very easily"

print(s)
```

HelloAaradhya, You are the most luckiest girl as you have Abhishek,Aiswarya and Big B as family members, You can learn Python very easily

How to define multi line f-strings:

```
name='Durga'
```

```
age=60
subject='Python'
msg=f'''
Name:{name},
age:{age},
subject:{subject}
'''
print(msg)
Name:Durga,
age:60,
subject:Python
```

Python f-string calling a function:

We can call function directly from f-string.

```
name='Durga'
print(f'Faculty Name:{name.upper()}')

Faculty Name:DURGA
```

```
def mymax(a,b):
    max=a if a>b else b
    return max
```

```
a=int(input('Enter First Number:'))
b=int(input('Enter Second Number:'))
print(f'The Maximum of {a} and {b} is {mymax(a,b)}')
```

```
D:\durgaclasses>py test.py
Enter First Number:100
Enter Second Number:200
The Maximum of 100 and 200 is 200
```

```
def mymax(a,b,c):
    max=a if a>b and a>c else b if b>c else c
    return max

a=int(input('Enter First Number:'))
b=int(input('Enter Second Number:'))
c=int(input('Enter Third Number:'))
print(f'The Maximum of {a},{b} and {c} is {mymax(a,b,c)}')
```

```
Enter First Number:10
Enter Second Number:60
Enter Third Number:30
The Maximum of 10,60 and 30 is 60
```

Python f-strings for objects:

----

str.format() method will always call str() method only.
But in f-string, we can call either str() or repr() based on our requirement.

```
class Student:
    def __init__(self,name,rollno,marks):
        self.name=name
        self.rollno=rollno
        self.marks=marks
    def __str__(self):
        return
f'Name:{self.name},RollNo:{self.rollno},Marks:{self.marks}'
    def __repr__(self):
        return f'Student Name:{self.name},Student
RollNo:{self.rollno},Student Marks:{self.marks}'

s=Student('Ravi',101,90)
print('Information--->{}'.format(s)) #
print(f'Information --->{s}')
print(f'Information --->{s!r}')
```

```
D:\durgaclasses>py test.py
Information--->Name:Ravi,RollNo:101,Marks:90
Information --->Name:Ravi,RollNo:101,Marks:90
```

Information --->Student Name:Ravi,Student RollNo:101,Student Marks:90

Expressions inside f-strings:

We can pass expressions inside f-string and these expressions will be evaluated at runtime.

```
a=10
b=20
c=30
print(f'The Result:{10*20/3}')
print(f'The Result:{10*20/3:.2f}')
print(f'The Result:{a+b*c}')
```

```
D:\durgaclasses>py test.py
The Result:66.66666666666667
The Result:66.67
The Result:610
```

title: f-strings (formatted strings) 3.6 Version Enhancement Part-2

How to use curly brances inside f-strings:

```
print(f'{ is a special symbol')
```

SyntaxError: f-string: expecting '}'

```
print(f'{{ is a special symbol') #{ is a special symbol

print(f'{{{ is a special symbol') #SyntaxError: f-string: expecting '}'

print(f'{{{{ is a special symbol') #{{ is a special symbol
```

The same is applicable for } only.

Q. Which of the following are valid?

print(f'} is my favourite symbol') --->Invalid SyntaxError: f-string: single '}' is not allowed
print(f'}} is my favourite symbol')-->valid
print(f'}}} is my favourite symbol')-->invalid
print(f'}}}} is my favourite symbol')-->valid

```
name='Durga'
print(f'Name:{name}') # Name:Durga
print(f'Name:{{name}}') #Name:{name}
print(f'Name:{{{name}}}') #Name:{Durga}
print(f'Name:{{{{name}}}}') #Name:{{name}}
```

```
Name:Durga
Name:{name}
Name:{Durga}
Name:{{name}}
```

3.6 version enhancements realted to f-string

3.8 version enhancements related to f-string:

We can use = symbol inside f-string for self documenting expressions and it is very useful for debugging purposes.

```
x=10
y=20
print(f'{x=}')
print(f'{y=}')


Name='Durga'
Salary=10000
Girl_Friend_Name='Sunny'
print(f'{Name=},{Salary=},{Girl_Friend_Name=}')

Name='Durga',Salary=10000,Girl_Friend_Name='Sunny'
```

We can also use Walrus Operator(:=) inside f-string:

```
import math
half_radius=10
print(f'The Area of Circle with radius {2*half_radius} is
{math.pi*2*half_radius*2*half_radius}')
```

```
print(f'The Area of Circle with radius {(r := 2*half_radius)} is {math.pi*r*r}')
print(f'The Area of Circle with radius {(r := 2*half_radius)} is {math.pi*r*r:.2f}')
```

```
The Area of Circle with radius 20 is 1256.6370614359173
The Area of Circle with radius 20 is 1256.6370614359173
The Area of Circle with radius 20 is 1256.64
```

f-string ===>Python 3.6
= and := operators are allowed inside f-string ===>Python 3.8

%-formatting
str.format()

performance
concise,less verbose,more readable

The walrus operator(3.8)
The positional only parameters(3.8)
f-strings concept(3.6 and 3.8)

6 months -->devops
6 months -->data science

durgasoftonlinetraining@gmail.com
8885252627
8096969696

durgasoftonline.com

one doubt, can people with Naga Dhosh can also learn Python sir? :)
durgasoftonline@gmail.com

title- Handling curly braces inside f-string and 3.8 version enhancements
title: f-strings (formatted strings) 3.6 & 3.8 Versions Enhancement Part-3

1. The Walrus operator(3.8 version)
2. The Positional only Parameter (3.8 version)
3. f-strings(3.6 version and 3.8 version)

4. Dictionary Related Enhancements(3.7,3.8 and 3.9 Versions):
----
dict is one of very commonly used types in python.

list,tuple,set ==>meant for individual elements
dict ===>a group of key-value pairs

```
d={100:'Sunny',200:'Bunny',300:'Chinny'}
d[400]='Vinny'
d[100]='Zinny'
print(d)
```

3.7 version enhancements:
----
list and tuple ===>insertion order will be preserved
set ===>order is not preserved

```
l=[10,20,30,40,50]
t=(10,20,30,40,50)
s={10,20,30,40,50}
print(l) #[10,20,30,40,50]
print(t) #(10,20,30,40,50)
print(s) #{50, 20, 40, 10, 30}
```

dict===>order won't be preserved.

```
d = {}
d[100]='Sunny'
d[200]='Bunny'
d[300]='Chinny'
d[400]='Vinny'
d[500]='Pinny'
```

print(d) # No guarantee for the order

Q. How to preserve insertion order in the dictionary???
By using OrderedDict we can preserve insertion order.
OrderedDict present in collections module.

```
from collections import OrderedDict

d = OrderedDict()
d[100]='Sunny'
d[200]='Bunny'
d[300]='Chinny'
d[400]='Vinny'
d[500]='Pinny'
for k,v in d.items():
        print(k,'---->',v)
```

```
100 ----> Sunny
200 ----> Bunny
300 ----> Chinny
400 ----> Vinny
500 ----> Pinny
```

This total teminology is applicable until 3.6 version only.
In 3.7 version normal dict functionality is replaced with OrderedDict functionality.

Hence from 3.7 version onwards, insertion order is guaranteed in dict.

```
d = {}
d[100]='Sunny'
d[200]='Bunny'
d[300]='Chinny'
d[400]='Vinny'
d[500]='Pinny'
print(d) #{100: 'Sunny', 200: 'Bunny', 300: 'Chinny', 400: 'Vinny', 500: 'Pinny'}
```

3.7 version
dict==>

3.8 Version Enhancements:

By using reversed() function we can do reversal of list and tuple also.

```
l1=[10,20,30,40]
r=reversed(l1)
l2=list(r)
print('Original Order:',l1)
print('Reversed Order:',l2)
```

```
t1=(10,20,30,40)
r=reversed(t1)
t2=tuple(r)
print('Original Order:',t1)
print('Reversed Order:',t2)
```

But reversed() function not aaplicable for set and dict also.

```
s1={10,20,30,40}
r=reversed(s1) #TypeError: 'set' object is not reversible
```

```
d = {100: 'Sunny', 200: 'Bunny', 300: 'Chinny', 400: 'Vinny', 500: 'Pinny'}
print(reversed(d)) #TypeError: 'dict' object is not reversible
print(reversed(d.keys())) #TypeError: 'dict_keys' object is not reversible
print(reversed(d.values())) #TypeError: 'dict_values' object is not reversible
print(reversed(d.items())) #TypeError: 'dict_items' object is not reversible
```

Until 3.7 version how to iterate items of dict in reverse order:

```
d = {100: 'Sunny', 200: 'Bunny', 300: 'Chinny', 400: 'Vinny', 500:
'Pinny'}
keys=d.keys()
l=list(keys) #[100,200,300,400,500]
r= reversed(l)
for k in r:
   print(k,'---->',d[k])
```

```
500 ----> Pinny
400 ----> Vinny
300 ----> Chinny
200 ----> Bunny
100 ----> Sunny
```

From 3.8 version onwards, we can apply reversed() for dict also.

```
d = {100: 'Sunny', 200: 'Bunny', 300: 'Chinny', 400: 'Vinny', 500:
'Pinny'}
print(reversed(d))
print(reversed(d.keys()))
print(reversed(d.values()))
print(reversed(d.items()))
```

This code is valid in 3.8 version but invalid in 3.7.

From 3.8 version how to iterate items of dict in reverse order:

```
d = {100: 'Sunny', 200: 'Bunny', 300: 'Chinny', 400: 'Vinny', 500:
'Pinny'}
r=reversed(d)
for k in r:
   print(k,'---->',d[k])
```

```
500 ----> Pinny
400 ----> Vinny
300 ----> Chinny
200 ----> Bunny
100 ----> Sunny
```

In 3.7 version, guarantee for the insertion order.

In 3.8 version, we can apply reversed() for dict also.

3.9 Version Enhancement:

How to merge 2 dictionaries into a third dictionary:

----

merge operation or union operation

```
d1 = {100: 'Sunny', 200: 'Bunny', 300: 'Chinny'}
d2 = {300:'Vinny',400:'Pinny',500:'Zinny'}

# merging operation
'''d3 = {**d1, **d2} #1st way
print(d1)
print(d2)
print(d3)

d3=d1.copy() #2nd way
for k,v in d2.items():
        d3[k]=v
print(d1)
print(d2)
print(d3) '''

#in 3.9 version as the part of PEP 584
d3 = d1 | d2
print(d1)
print(d2)
print(d3)
```

How to update an existing dictionary with items of another dictionary:

---

update --->inline merge

```
d1 = {100: 'Sunny', 200: 'Bunny', 300: 'Chinny'}
d2 = {300:'Vinny',400:'Pinny',500:'Zinny'}

'''#update operation
d1.update(d2) # 1st way
print(d1)
print(d2)'''

# from 3.9 version onwards
d1 |= d2
print(d1)
print(d2)
```

From Python 3.9 version onwards , we can use | and |= operator for dictionaries also.

1. In 3.7 version --->Guarantee for insertion order
2. In 3.8 version --->We can apply reversed() for dict also
3. In 3.9 version --->We can apply | and |= operators for dict also

Few Important Interview Questions:

Q1. How to merge two lists into a new list?

```
l1=[10,20,30]
l2=[40,50,60]
l3=l1+l2 #1st way
print(l1)
print(l2)
print(l3)
```

```
l1=[10,20,30]
l2=[40,50,60]
l3=[*l1,*l2] #2nd way
print(l1)
print(l2)
print(l3)
```

Q2. How to update existing list with elements of another list?

```
l1=[10,20,30]
l2=[40,50,60]
l1.extend(l2)
print(l1)
print(l2)
```

Q3. How to merge two tuples into a new tuple?

```
t1=(10,20,30)
```

t2=(40,50,60)
t3=t1+t2 #1st way
print(t3)

t4=(*t1,*t2) #2nd way
print(t4)

Q4. How to update existing tuple with elements of another tuple?
It is impossible.
Tuple is immutable. Once we creates a tuple, we cannot perform any changes.

Q5. How to merge two sets into a new set?
s1 = {10,20,30}
s2 = {40,50,60,30}

s3 ={*s1,*s2} # 1st way
print(s3)

s4=s1|s2 #2nd way -->it is old approach
print(s4)

Q6. How to update existing set with elements of another set?
s1 = {10,20,30}
s2 = {40,50,60,30}

#update s1 with elements of s2

```
#s1.update(s2) #1st way
s1|=s2 #2nd way
print(s1)
print(s2)
```

Title: Dictionary Related Enhancements(3.7,3.8 and 3.9 Versions)

New SyntaxWarnings in Python 3.8 Version:

Python has a SyntaxWarning that can warn you about dubious(doubtful) syntax, that is typically not a syntaxerror.

BaseException
|-Exception
|-Warning
|-SyntaxWarning

1. is and is not operators for numbers and strings
2. While creating large collections, we may miss ,

1. is and is not operators for numbers and string literals
----
is operator vs == operator

is operator meant for reference comparison.
== operator meant for content comparison

```
l1= [10,20,30,40]
l2= [10,20,30,40]
print(l1 is l2)
print(l1 == l2)
```

10 and 10

'durga'   and 'durga'

Warning (from warnings module):
** File "D:\durgaclasses\test.py", line 2**
**   if x is 1000:**
SyntaxWarning: "is" with a literal. Did you mean "=="?
>>>
========== RESTART: D:\durgaclasses\test.py
==========
x value is 1000

A new SyntaxWarning added extra in 3.8 version which will be raised if we use is and is not operators for number and string literals.
It is highly recommended to use == operator instead of is for literal comparison.

2. While creating large collections, we may miss ,
To altert this, a new SyntaxWarning added to python 3.8

```
l = [
        [10,20,30]
        [40,50,60]
        [70,80,90]
]
print(l)
```

Traceback (most recent call last):

File "D:\durgaclasses\test.py", line 3, in <module>
[40,50,60]
TypeError: list indices must be integers or slices, not tuple

_ symbol in literal
Enum concept in python
continue inside finally

Type Hints

Title-New SyntaxWarnings in Python 3.8 version

1. B.Tech Student studying 3rd and then 4th year
He did a small part time job of daily 3 to 4 hours
public telephone booth -->operator 3k to 4k
72k

60% marks in B.Tech

2. 96.3

in some engineering college-->KLCE VJA
8K--->96K
82 percentile

correct thing at this correct time

Type Hints in Python/ Annotations

Type Hints in Python 3.5 version as the part of PEP 484.

Python is dynamically typed programming language, where we are not allowed to define the type.

From Python 3.5 version onwards, we can declare the type explicitly in python.
Static Type Checking is possible because of type hints(PEP 484).
We can declare the type for variables.
We can declare the type for function arguments.
We can declare the type for return values.

```
def add(x,y):
    print(x+y)
    #100 lines functionality
```

```
add(10,20)
add('durga','soft')
add({},{})

30
durgasoft
Traceback (most recent call last):
  File "D:\durgaclasses\test.py", line 7, in <module>
    add({},{})
  File "D:\durgaclasses\test.py", line 2, in add
    print(x+y)
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

Function Related Type Hints | Function Annotations:

---

static type checking can be done by using 3rd party tools.

mypy -->open source tool from DropBox

```
pip install mypy
D:\durgaclasses>pip install mypy
Collecting mypy
  Downloading
https://files.pythonhosted.org/packages/cd/76/65212259928df6bbf8
0e40a142c7990001f15ffb4ea5299bceca8c3b3183/mypy-0.790-cp37-
cp37m-win_amd64.whl (8.5MB)
```

|████████████████████████████████| 8.5MB 386kB/s

```
Collecting typing-extensions>=3.7.4
  Downloading https://files.pythonhosted.org/packages/60/7a/e881b5abb54db0e6e671ab088d079c57ce54e8a01a3ca443f561ccadb37e/typing_extensions-3.7.4.3-py3-none-any.whl
Collecting mypy-extensions<0.5.0,>=0.4.3
  Downloading https://files.pythonhosted.org/packages/5c/eb/975c7c080f3223a5cdaff09612f3a5221e4ba534f7039db34c35d95fa6a5/mypy_extensions-0.4.3-py2.py3-none-any.whl
Requirement already satisfied: typed-ast<1.5.0,>=1.4.0 in c:\users\lenovo\appdata\roaming\python\python37\site-packages (from mypy) (1.4.1)
Installing collected packages: typing-extensions, mypy-extensions, mypy
Successfully installed mypy-0.790 mypy-extensions-0.4.3 typing-extensions-3.7.4.3
WARNING: You are using pip version 19.3.1; however, version 20.2.4 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

How to perform static type checking by using mypy:

---

1. py -m mypy test.py

This command won't execute python program and it will perform just static type checking.

2. mypy test.py

This command won't execute python program and it will perform just static type checking.

```
def add(x: int,y: int) -> None:
        print(x+y)


add(10,20)
```

```
D:\durgaclasses>mypy test.py
Success: no issues found in 1 source file
```

```
def add(x: int,y: int) -> None:
        print(x+y)


add(10,20)
add('durga','soft')
```

```
D:\durgaclasses>mypy test.py
test.py:6: error: Argument 1 to "add" has incompatible type "str";
expected "int"
test.py:6: error: Argument 2 to "add" has incompatible type "str";
expected "int"
Found 2 errors in 1 file (checked 1 source file)
```

anyway client will execute by py test.py????

```
def add(x: int,y: int) -> None:
        print(x+y)


add(10,20)
add('durga','soft')
add(10,'soft')
```

```
def function_name(arg1: type, arg2: type) -> type_of_returned_value:
    body of the function


def function_name(arg1, arg2):
    body of the function
```

is not a contradiction to python is dynamically typed lang

```
def add(x: str,y: str) -> str:
        return x+y

print(add.__annotations__)
{'x': <class 'str'>, 'y': <class 'str'>, 'return': <class 'str'>}
```

Variable Related Type Hints | Variable Annotations:

```
name: str = 'Durga'
name=10
print(name)
```

```
test.py:2: error: Incompatible types in assignment (expression has
type "int", variable has type "str")
Found 1 error in 1 file (checked 1 source file)
```

syntax:

```
variablename: type = value
```

```
a: int = 10
b: int = 20
c: bool = True
'''
ksjdfsajldsjlak
ksjdfsajldsjlak
ksjdfsajldsjlak
ksjdfsajldsjlak
ksjdfsajldsjlak
ksjdfsajldsjlak
ksjdfsajldsjlak
ksjdfsajldsjlak'''
print(__annotations__)
D:\durgaclasses>py test.py
{'a': <class 'int'>, 'b': <class 'int'>, 'c': <class 'bool'>}
```

ay they given like a:int = 10 this sir, y they did not mention like int a = 10, int b = 20 like java? is there any reason?

```
int a =10;
int b = 20;
```

a: float

a='durga'
print(a)

test.py:3: error: Incompatible types in assignment (expression has type "str", variable has type "float")
Found 1 error in 1 file (checked 1 source file)

int,float,bool,str

list of int values
dict of int keys and string values
tuple of int values
set of float values

Typing Library:
----

x: int = 10

```
def f1(x: str,y: int ) -> int:
  pass
```

```
x: int =  10
x: float = 10.5
print(x) which one will come
```

```
D:\durgaclasses>py test.py
10.5

D:\durgaclasses>mypy test.py
test.py:2: error: Name 'x' already defined on line 1
Found 1 error in 1 file (checked 1 source file)
```

Title: Type Hints or Annotations (Python 3.5 Version New feature as the part of PEP 484)

Type Hints / Annotations:

Python 3.5 version as the part of PEP 484

1. Statically Typed Languages vs Dynamically Typed Languages

C,C++,Java

Python

Dynamically Typed Languages:

We are not required to declare the type explicitly.

The type will be considered automatically.

Easyness to the programmer.

Problems with Dynamically Typed Programming Languages:

```
class Test
{
	public static void main(String[] args)
	{
		int x=10;
		x="durga";
		System.out.println("Hello World!");
	}
}
```

The chance of failing java program with type related issues at runtime is very very less.
The chance of failing python program with type related issued at runtime is very common.
Dynamic Typing Bugs

```
def factorial(n):
    if n<0:
        return None
    elif n==0:
        return 1
    else:
        return n*factorial(n-1)

for i in range(11):
    print(f'The factorial of {i} is:{factorial(i)}')
```

```
def factorial(n):
    if n<0:
        return None
    elif n==0:
        return 1
    else:
```

```
        return n*factorial(n-1)

for i in range(11):
    print(f'The factorial of {i} is:{factorial(i)}')

print(factorial('durga'))
```

Type Hints in 3.5 version

Type hints can be used for static type checking before execution.

Type hints for functions:

```
def function_name(arg1,arg2):
  pass


def function_name(arg1: type_of_arg1,arg2: type_of_arg2) ->
return_type:
        pass
```

There is effect on runtime because of these type hints.
Python interpreter won't consider these type hints.
3rd party tools can understand these type hints.

mypy is 3rd party tool to perform static type checking for python program.

pip install mypy

py test.py

Execute Python program

mypy test.py

It won't execute python program and just perform static type checking

```
def factorial(n: int) -> int:
    if n<0:
        return None
    elif n==0:
        return 1
    else:
        return n*factorial(n-1)

for i in range(11):
    print(f'The factorial of {i} is:{factorial(i)}')

print(factorial('durga'))
```

```
D:\durgaclasses>mypy test.py
test.py:3: error: Incompatible return value type (got "None", expected
"int")
test.py:12: error: Argument 1 to "factorial" has incompatible type
"str"; expected "int"
Found 2 errors in 1 file (checked 1 source file)
```

```
def factorial(n: int):
    if n<0:
        return None
    elif n==0:
        return 1
    else:
        return n*factorial(n-1)

for i in range(11):
    print(f'The factorial of {i} is:{factorial(i)}')

print(factorial(6))
```

static vs Dynamic typed programming languages
Advantages of Dynamic typing
problems with Dynamic Typing
Dynamic Typing bugs

Type Hints Python 3.5
3rd party tool-->mypy

Type hints for functions:

```
def f1(x: int,y: int) -> int:
        return x*y
```

basic type hints:
----
int,float,bool,str

Type hints for variables:
----
variable_name = value
variable_name: type = value

i feel python is slowly converting to statically typed language

Q1. Write type hints for the function

1st argument should be of type int

2nd argument should be of type float

3rd argument should be of type bool

return value should be str

```
def f1(x,y,z):
    pass
```

```
def f1(x: int,y: float, z: bool) ->str:
    pass
```

Q2. Declare a variable pincode which is always int type?

```
pincode = 500038
pincode: int = 500038
```

Sir, correct me if i am wrong, pyhton interpreter does not perform type hints but python lang supports type hints. Is it not contradiction

So type hints is only for progammer but not useful to client

Complex Type Hints from typing library:

---

typing module

List,Tuple,Set,Dict,Sequence,Union,Optional etc

Typing Library List:

Q1. How to define list of string values by using type hints

without type hint:

```
# this code developed by dev1
def getnames():
    names = []

    names.append('durga')
    names.append('ravi')
    names.append('shiva')
    names.append(10.5)
    return names
```

```
# he is user
names = getnames()
for name in names:
        print(f'{name} contains {len(name)} characters')

D:\durgaclasses>py test.py
durga contains 5 characters
ravi contains 4 characters
shiva contains 5 characters
Traceback (most recent call last):
  File "D:\durgaclasses\test.py", line 13, in <module>
    print(f'{name} contains {len(name)} characters')
TypeError: object of type 'float' has no len()
```

```
# this code developed by dev1
from typing import List
def getnames() -> List[str]:
        names: List[str] = []

        names.append('durga')
        names.append('ravi')
        names.append('shiva')
        names.append(10)
        return names
```

```
# he is user
names = getnames()
for name in names:
        print(f'{name} contains {len(name)} characters')
```

```
D:\durgaclasses>mypy test.py
test.py:9: error: Argument 1 to "append" of "list" has incompatible
type "int"; expected "str"
Found 1 error in 1 file (checked 1 source file)
```

```
from typing import List

names: List[str] = ['A','B','C']
ages: List[int] = [10,20,30]
nestlist: List[List[int]] = [[10,20,30],[40,50,60],[70,80,90]]
```

```
from typing import List
```

names: List[str] = ['A','B','C',10]
ages: List[int] = [10,20,30,'A']
nestlist: List[List[int]] = [[10,20,30],[40,50,60],[70,80,90],100]

D:\durgaclasses>mypy test.py
test.py:3: error: List item 3 has incompatible type "int"; expected "str"
test.py:4: error: List item 3 has incompatible type "str"; expected "int"
test.py:5: error: List item 3 has incompatible type "int"; expected "List[int]"
Found 3 errors in 1 file (checked 1 source file)

sir try [10,[10,[10,'ram'],30],40]?

ages: List[int] = [10,20,30]

Typing Library for Tuple:
----
t=(10,20,30)
t: Tuple[int] = (10,20,30) ===>invalid

We should specify the type for all elements.
from typing import Tuple

```
t1: Tuple[int,int,int] = (10,20,30)
t2: Tuple[int,str,int] = (10,'A',30)
t3: Tuple[int,str,int] = (10,'A',30,40) #invalid
print(t1,t2,t3)
```

https://drive.google.com/drive/folders/1GgH7aFiCveMqc7S8P4c2RcpncultqULb?usp=sharing

Title: Overview of Type Hints and Type Hints for List and Tuple

statically Typed vs Dynamic Typed
Advantages of Dynamically Typed
Limitations of Dynamically Typed
Type Hints
Function related type hints
Variable related type hints
basic type hints
complex type hints: typing

List,Tuple

```
l = [10,20,30,40]
```

```
from typing import List
l: List[int] = []
l.append(10)
```

```
l.append(20)
l.append(30)
l.append(40)
l.append('A')
print(l)
```

```
test.py:7: error: Argument 1 to "append" of "list" has incompatible
type "str"; expected "int"
Found 1 error in 1 file (checked 1 source file)
```

```
from typing import Tuple

t: Tuple[int,int,int] = (10,20,'durga')
print(t)
```

```
test.py:3: error: Incompatible types in assignment (expression has
type "Tuple[int, int, str]", variable has type "Tuple[int, int, int]")
```

Type Hint for Set:

```
----------
from typing import Set

s: Set[int] = {10,20,30,40,'A'}
print(s)
```

test.py:3: error: Argument 5 to <set> has incompatible type "str"; expected "int"
Found 1 error in 1 file (checked 1 source file)

```
from typing import Set

s: Set[int] = set()
s.add(10)
s.add(20)
s.add(30)
s.add('durga')

print(s)
```

test.py:7: error: Argument 1 to "add" of "set" has incompatible type "str"; expected "int"
Found 1 error in 1 file (checked 1 source file)

Typing Library for dict:
----
dict means key-value pairs

```
from typing import Dict
students: Dict[int,str] = {}
```

```
students[100]='Durga'
students[200]='Ravi'
students[300]='Shiva'
students[400]='Pavan'
students['katrina']=500
students[500]=600
print(students)
```

```
test.py:7: error: Invalid index type "str" for "Dict[int, str]"; expected
type "int"
test.py:7: error: Incompatible types in assignment (expression has
type "int", target has type "str")
test.py:8: error: Incompatible types in assignment (expression has
type "int", target has type "str")
Found 3 errors in 1 file (checked 1 source file)
```

List,Tuple,Set,Dict

Sequence:

```
from typing import Sequence
s: Sequence
s='durga'
print(s)
```

```
D:\durgaclasses>mypy test.py
```

Success: no issues found in 1 source file

D:\durgaclasses>py test.py
durga

```
from typing import Sequence
s: Sequence
s=[10,20,30,40]
print(s)
```

D:\durgaclasses>mypy test.py
Success: no issues found in 1 source file

D:\durgaclasses>py test.py
[10, 20, 30, 40]

```
from typing import Sequence
s: Sequence
s=10
print(s)
```

D:\durgaclasses>mypy test.py
test.py:3: error: Incompatible types in assignment (expression has type "int", variable has type "Sequence[Any]")
Found 1 error in 1 file (checked 1 source file)

```
D:\durgaclasses>py test.py
10
```

Typing Hints : Callable:

Callable is the type hint for the function type.

f: Callable[[arg1Type,arg2Type],returntype]

```
from typing import Callable
def sum(a: int,b: int) -> None:
        print(a+b)

def f1(f: Callable[[int,int],None],x: int,y: int) ->None:
        f(x,y)

f1(sum,10,20)
```

```
D:\durgaclasses>py test.py
30

D:\durgaclasses>mypy test.py
Success: no issues found in 1 source file
```

```
from typing import Callable
def sum(a: str,b: str) -> None:
        print(a+b)

def f1(f: Callable[[int,int],None],x: str,y: str) ->None:
        f(x,y)

f1(sum,'durga','soft')
```

```
D:\durgaclasses>py test.py
durgasoft

D:\durgaclasses>mypy test.py
test.py:6: error: Argument 1 has incompatible type "str"; expected
"int"
test.py:6: error: Argument 2 has incompatible type "str"; expected
"int"
test.py:8: error: Argument 1 to "f1" has incompatible type
"Callable[[str, str], None]"; expected "Callable[[int, int], None]"
Found 3 errors in 1 file (checked 1 source file)
```

```
from typing import Callable
def sum(a,b):
        print(a+b)

def f1(f: Callable[[int,int],None],x: int,y: int) ->None:
        f(x,y)

f1(sum,10,20)
```

```
D:\durgaclasses>mypy test.py
Success: no issues found in 1 source file

D:\durgaclasses>py test.py
30
```

Callable acts as Type Hint for Functions

x=10 or 10.5

Type Hints : Union

Some times variable or argument can be of multiple types.

x can be either int or float
We can define such types by using Union

```
from typing import Union
def f1(x: Union[int,float]) -> None:
        print(x)

f1(10)
f1(10.5)
f1('durga')
```

```
D:\durgaclasses>py test.py
10
10.5
durga

D:\durgaclasses>mypy test.py
test.py:7: error: Argument 1 to "f1" has incompatible type "str";
expected "Union[int, float]"
Found 1 error in 1 file (checked 1 source file)
```

```
from typing import *
x: Union[List,int,Dict]

x=10
```

```
print(x)

x=[10,20,30,40]
print(x)

x={100:'durga',200:'shiva'}
print(x)

x=(10,20,30)
```

```
D:\durgaclasses>mypy test.py
test.py:13: error: Incompatible types in assignment (expression has type "Tuple[int, int, int]", variable has type "Union[List[Any], int, Dict[Any, Any]]")
Found 1 error in 1 file (checked 1 source file)

D:\durgaclasses>py test.py
10
[10, 20, 30, 40]
{100: 'durga', 200: 'shiva'}
```

```
def factorial(n: int) -> int:
    if n<0:
        return None
    elif n==0:
        return 1
    else:
        return n*factorial(n-1)

for i in range(11):
    print(f'The factorial of {i} is:{factorial(i)}')
```

```
test.py:3: error: Incompatible return value type (got "None", expected
"int")
Found 1 error in 1 file (checked 1 source file)
```

```
from typing import Union
def factorial(n: int) -> Union[int,None]:
    if n == None:
        return 1
    if n<0:
        return None
    elif n==0:
        return 1
    else:
```

```
            return n*factorial(n-1)

for i in range(11):
        print(f'The factorial of {i} is:{factorial(i)}')
```

Optional Type:

Union[int,None] --->Optional[int]--->Either int or None
Union[float,None] --->Optional[float]--->Either float or None

```
from typing import Optional
x: Optional[int]
x=10
print(x)
x=None
print(x)
```

```
D:\durgaclasses>mypy test.py
Success: no issues found in 1 source file

D:\durgaclasses>py test.py
10
None
```

```
from typing import Optional
def f1(x: int) -> int:
        if x>10:
                return None
        else:
                return x

f1(10)
```

```
test.py:4: error: Incompatible return value type (got "None", expected
"int")
Found 1 error in 1 file (checked 1 source file)
```

```
from typing import Union
def f1(x: int) -> Union[int,None]:
        if x>10:
                return None
        else:
                return x

f1(10)
```

```
D:\durgaclasses>mypy test.py
Success: no issues found in 1 source file
```

```
from typing import Optional
```

```
def f1(x: int) -> Optional[int]:
    if x>10:
        return None
    else:
        return x

f1(10)
```

```
D:\durgaclasses>mypy test.py
Success: no issues found in 1 source file
```

List,Tuple,Set,Dict,Sequence,Callable,Union,Optional

Type Hints In IDEs:

Pycharm
atom
vscode
eclipse

Most of the code will be generated by IDE itself because auto completion is available.
debugging will be very easy.

if we are doing some mistake immediately, IDE itself will alert us to correct that mistake.

Advantages:

----

1. Type Hints helps to catch errors related to type.

2. Type Hints helps to document our code.

3. Type Hints improve IDE functionality

Limitations:

----

1. Type hints take more developers time as we have to add type hints explicitly.

2. Readability will be reduced as the length of the code increases

3. Type Hints will work only in newer versions

4. The startup time may increases if we use typing library.

Title: Type Hints - Set,Dict,Sequence,Callable,Union,Optional and Advantages and Limitations of Type Hints

Dataclasses --->Python 3.7 version PEP-557:

Python-->All Rounder(POP,OOP,Scripting Language,Modular PL)

```
class Employee:
        def __init__(self,eno,ename,esal,eaddr):
                self.eno = eno
                self.ename = ename
                self.esal = esal
                self.eaddr = eaddr

e = Employee(100,'Sunny',10000,'Mumbai')
print(e.eno,e.ename,e.esal,e.eaddr)
print(e.__dict__)


D:\durgaclasses>py test.py
100 Sunny 10000 Mumbai
{'eno': 100, 'ename': 'Sunny', 'esal': 10000, 'eaddr': 'Mumbai'}



class Employee:
        def __init__(self,eno,ename,esal,eaddr):
                self.eno = eno
                self.ename = ename
                self.esal = esal
                self.eaddr = eaddr
        def __repr__(self):
                return
f'Employee({self.eno},{self.ename},{self.esal},{self.eaddr})'
```

```
e1 = Employee(100,'Sunny',10000,'Mumbai')
e2 = Employee(200,'Bunny',20000,'Hyderabad')
print(e1)
print(e2)
```

```
D:\durgaclasses>py test.py
Employee(100,Sunny,10000,Mumbai)
Employee(200,Bunny,20000,Hyderabad)
```

Difference between is operator and == operator:

---

is operator meant for reference comparison(address comparison)
== operator meant for content comparison

```
class Employee:
    def __init__(self,eno,ename,esal,eaddr):
        self.eno = eno
        self.ename = ename
        self.esal = esal
        self.eaddr = eaddr
    def __repr__(self):
        return
f'Employee({self.eno},{self.ename},{self.esal},{self.eaddr})'
```

```
e1 = Employee(100,'Sunny',10000,'Mumbai')
e2 = Employee(200,'Bunny',20000,'Hyderabad')
e3 = Employee(100,'Sunny',10000,'Mumbai')
print(e1 == e2) #False
print(e1 == e3) #True
```

o/p:
False
False

We didn't override == opertor for employee objects.
== operator simply calls is operator only which is meant for reference comparison.

+ ----> __add__()

== ---->__eq__()

```
class Employee:
    def __init__(self,eno,ename,esal,eaddr):
        self.eno = eno
        self.ename = ename
        self.esal = esal
```

```
            self.eaddr = eaddr
        def __repr__(self):
            return
f'Employee({self.eno},{self.ename},{self.esal},{self.eaddr})'

        def __eq__(self,other):
            if self.eno == other.eno and self.ename == other.ename
and self.esal == other.esal and self.eaddr == other.eaddr:
                return True
            else:
                return False


e1 = Employee(100,'Sunny',10000,'Mumbai')
e2 = Employee(200,'Bunny',20000,'Hyderabad')
e3 = Employee(100,'Sunny',10000,'Mumbai')
print(e1 == e2) #False
print(e1 == e3) #True

o/p:
D:\durgaclasses>py test.py
False
True
```

We can implement __eq__() method in shortcut way as follows:

```
def __eq__(self,other):
        return (self.eno,self.ename,self.esal,self.eaddr) ==
(other.eno,other.ename,other.esal,other.eaddr)
```

Boiler Plate code

Dataclasses --->Python 3.7 version PEP-557
We should use @dataclass decorator which is present in dataclasses module.
Dataclass will generate boiler plate code for our classes. It provides several methods like __init__(),__repr__(),__eq__() etc
The main advantage is length of the code will be reduced and readability will be improved.
The programmers life will become very simple.

```
from dataclasses import dataclass

@dataclass
class Employee:
    eno: int
    ename: str
    esal: int
    eaddr: str
```

```
e1 = Employee(100,'Sunny',10000,'Mumbai')
e2 = Employee(200,'Bunny',20000,'Hyderabad')
e3 = Employee(100,'Sunny',10000,'Mumbai')
print(e1.eno,e1.ename,e1.esal,e1.eaddr)
print(e1)
print(e2)
print(e1 == e2)
print(e1 == e3)
```

```
D:\durgaclasses>py test.py
100 Sunny 10000 Mumbai
Employee(eno=100, ename='Sunny', esal=10000, eaddr='Mumbai')
Employee(eno=200, ename='Bunny', esal=20000, eaddr='Hyderabad')
False
True
```

```
@dataclass
class Employee:
    eno: int
    ename: str
    esal: int
    eaddr: str
```

```
@dataclass()
class Employee:
        eno: int
        ename: str
        esal: int
        eaddr: str
```

```
@dataclass(init=True,repr=True,eq=True,order=False,unsafe_hash=Fal
se, frozen=False)
class Employee:
        eno: int
        ename: str
        esal: int
        eaddr: str
```

If we specify order=True then these magic methods will be generated.

< -----> __lt__()
<= -----> __le__()
> ---->__gt__()
>= ---->__ge__()

```
from dataclasses import dataclass

@dataclass(repr=False,eq=False)
class Employee:
        eno: int
        ename: str
        esal: int
        eaddr: str



e1 = Employee(100,'Sunny',10000,'Mumbai')
e2 = Employee(200,'Bunny',20000,'Hyderabad')
e3 = Employee(100,'Sunny',10000,'Mumbai')
print(e1.eno,e1.ename,e1.esal,e1.eaddr)
print(e1)
print(e2)
print(e1 == e2)
print(e1 == e3)

D:\durgaclasses>py test.py
100 Sunny 10000 Mumbai
<__main__.Employee object at 0x000002A1C08E8160>
<__main__.Employee object at 0x000002A1C08E8E50>
False
False
```

The dataclass with frozen parameter(To create immutable classes):

----

frozen means fixed and we cannot change.

The default value for frozen parameter is False.

If we set True, then we cannot change values of fields and become constants.

Bydefault Dataclass objects are mutable. But if we set frozen=True, then the object will become immutable.

```
from dataclasses import dataclass

@dataclass()
class Employee:
    eno: int
    ename: str
    esal: int
    eaddr: str



e1 = Employee(100,'Sunny',10000,'Mumbai')
e1.esal = 20000
print(e1)
D:\durgaclasses>py test.py
Employee(eno=100, ename='Sunny', esal=20000, eaddr='Mumbai')
```

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Employee:
        eno: int
        ename: str
        esal: int
        eaddr: str



e1 = Employee(100,'Sunny',10000,'Mumbai')
e1.esal = 20000
print(e1)

D:\durgaclasses>py test.py
Traceback (most recent call last):
  File "D:\durgaclasses\test.py", line 13, in <module>
    e1.esal = 20000
  File "<string>", line 4, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 'esal'
```

sir if we want to make only one constant out of 4 variables: field()

the boilerplate code will be generated automatically and we are not required to write explicitly.
The complexity of programming will be reduced.
The length of the code will be reduced and readability will be improved.

```
class Student:
    def __init__(self,rollno,name,s1marks,s2marks,s3marks):
        self.rollno=rollno
        self.name=name
        self.s1marks=s1marks
        self.s2marks=s2marks
        self.s3marks=s3marks

    def __repr__(self):
        return f'Student(rollno = {self.rollno},name =
{self.name},s1marks = {self.s1marks},s2marks =
{self.s2marks},s3marks = {self.s3marks})'

    def __eq__(self,other):
        return
(self.rollno,self.name,self.s1marks,self.s2marks,self.s3marks) ==
```

```
(other.rollno,other.name,other.s1marks,other.s2marks,other.s3marks
)

s1=Student(101,'Sunny',70,80,90)
s2=Student(101,'Sunny',70,80,90)
print(s1 == s2)
```

```
from dataclasses import dataclass

@dataclass
class Student:
    rollno: int
    name: str
    s1marks: int
    s2marks: int
    s3marks: int


s1=Student(101,'Sunny',70,80,90)
s2=Student(101,'Sunny',70,80,90)
print(s1)
print(s2)
print(s1 == s2)
```

```
D:\durgaclasses>py test.py
Student(rollno=101, name='Sunny', s1marks=70, s2marks=80,
s3marks=90)
Student(rollno=101, name='Sunny', s1marks=70, s2marks=80,
s3marks=90)
True
```

Dataclass Parameters:

@dataclass(init=True,repr=True,eq=True,order=False,unsafe_hash=False,frozen)

init : ___init__() method will be generated if True.
Default value is True.

repr : ___repr__() method will be generated if True.
Default value is True.

eq : __eq__() method will be generated if True.
Default value is True.

order: __lt__(),__le__(),__gt__(),__ge__() will be generated if True.
Default value: False

unsafe_hash: __hash__() method will be generate if True.
frozen: if True, we cannot change values(immutable object)
Default value is False.

Bydefault dataclass object is Mutable. To make as immutable we have to set frozen=True.
Note: to make order value as True, compulsory we should make eq as True.
ValueError: eq must be true if order is true

```
from dataclasses import dataclass

@dataclass(eq=False,order=True)
class Student:
    rollno: int
    name: str
    s1marks: int
    s2marks: int
    s3marks: int



s1=Student(101,'Sunny',70,80,90)
s2=Student(101,'Sunny',70,80,90)
print(s1)
print(s2)
print(s1 == s2)
print(s1 < s2)
```

```
D:\durgaclasses>py test.py
Traceback (most recent call last):
  File "D:\durgaclasses\test.py", line 4, in <module>
    class Student:
  File
"C:\Users\lenovo\AppData\Local\Programs\Python\Python39\lib\da
taclasses.py", line 1013, in wrap
    return _process_class(cls, init, repr, eq, order, unsafe_hash, frozen)
  File
"C:\Users\lenovo\AppData\Local\Programs\Python\Python39\lib\da
taclasses.py", line 917, in _process_class
    raise ValueError('eq must be true if order is true')
ValueError: eq must be true if order is true
```

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Student:
        rollno: int
        name: str
        s1marks: int
        s2marks: int
        s3marks: int
```

```
s1=Student(101,'Sunny',70,80,90)
s2=Student(101,'Sunny',70,80,90)
s1.s1marks=100
print(s1)

D:\durgaclasses>py test.py
Traceback (most recent call last):
  File "D:\durgaclasses\test.py", line 14, in <module>
    s1.s1marks=100
  File "<string>", line 4, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 's1marks'
```

Default values for Fields:

Based on our requirement,we can provide default values for our fields.

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Student:
        rollno: int
        name: str
        s1marks: int =0
        s2marks: int =0
        s3marks: int =0
```

```
s1=Student(101,'Durga')
print(s1)
```

```
D:\durgaclasses>py test.py
Student(rollno=101, name='Durga', s1marks=0, s2marks=0,
s3marks=0)
```

Note: For any field, if we provide default value, then for all the reamining fields compulsory we should provide default values otherwise we will get TypeError.

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Student:
    rollno: int
    name: str
    s1marks: int =0
    s2marks: int
    s3marks: int =0
```

TypeError: non-default argument 's2marks' follows default argument

Customize Behaviour of fields by using field() function:

```
from dataclasses import dataclass,field

@dataclass
class Student:
    rollno: int = field(repr=False)
    name: str
    s1marks: int
    s2marks: int
    s3marks: int


s1=Student(101,'Sunny',70,80,90)
print(s1)

D:\durgaclasses>py test.py
Student(name='Sunny', s1marks=70, s2marks=80, s3marks=90)

from dataclasses import dataclass,field

@dataclass
class Student:
    rollno: int = field(repr=False)
    name: str = field(init=False)
    s1marks: int
    s2marks: int
    s3marks: int
```

s1=Student(101,'Sunny',70,80,90)

```
D:\durgaclasses>py test.py
Traceback (most recent call last):
  File "D:\durgaclasses\test.py", line 12, in <module>
    s1=Student(101,'Sunny',70,80,90)
TypeError: __init__() takes 5 positional arguments but 6 were given
```

Field Parameters:

----

init: This field will be included in the generated __init__() method if True.

Default value is True

repr: This field will be included in the generated __repr__() method if True.

Default value is True

compare: This field will be included in the generated __lt__(),__gt__(),__le__(),__ge__(),__eq__() methods if True.

Default value is True

hash: This field will be included in the generated __hash__() method if True.

Default value is True

default: To provide default value
default_factory: If provided, it must be zero argument callable that will be called to provide default value.
metadata: It is a dictionary to provide some extra information about this field.
eg: distance : float =field(default=0.0,metadata={'unit':'kilometers'})

```
@dataclass
class Student:
        marks: List[int] = field(default_factory=list,
metadata={'max_marks_per_subject':100})

from dataclasses import dataclass,field

@dataclass
class Student:
        rollno: int
        name: str
        s1marks: int = field(default=0)
        s2marks: int
        s3marks: int


s1=Student(101,'Sunny',70,80,90)
```

TypeError: non-default argument 's2marks' follows default argument

```
from dataclasses import dataclass,field
from typing import List

@dataclass
class Student:
    marks: List[int] = field(default_factory=list)


s1=Student()
print(s1)

s2=Student([10,20,30,40])
print(s2)
```

```
D:\durgaclasses>py test.py
Student(marks=[])
Student(marks=[10, 20, 30, 40])
```

All parameters of dataclass

What is the purpose of field() function?

To customize behaviour of the fields.

What parameters for field() function?

use __post__init__() to control python dataclass initialization
Inheritance
How to covert dataclass objects into dictionary and tuple

Title-All Parameters of Dataclass and importance and parameters of field() function

Python Logging:

Library
Lab
office

Computer Lab Log Book:
s.NO, Date, Name, Roolno, login time, desk no/pc no logout

1. tracking
2.

log file
weblogic admin
Roope--->w7ejb1--->log file
complete flow
Exceptions information

It is highly recommended to store complete application flow and exception information to a file. This process is called logging.

The main advantages of logging are:

1. **We can use log files while performing debugging**
2. **We can provide statistics like number of requests per day etc**

module: logging

logging levels:

----

depending on type of information, logging data is divided into the following levels in python

1. CRITICAL --->50

Represents a very serious problem that needs high attention.

2. ERROR ---> 40

Represents a serious error

3. WARNING --->30

Represents a warning message, some caution needed. It is alert to the programmer.

4. INFO --->20

Represents a message with some important information

5. DEBUG --->10

Represents a message which can be used for debugging

6. NOTSET --->0

Represents logging level not set

DEBUG(10) --->INFO(20) --->WARNING(30) --->ERROR(40) --->CRITICAL(50)

Default logging level: WARNING

How to implement logging:

----

create log file to store our log messages, we have to specify level

basicConfig() function of logging module.

```
logging.basicConfig(filename='log.txt',level=logging.WARNING)
logging.basicConfig(filename='log.txt',level=30)

logging.debug(message)
logging.info(message)
logging.warning(message)
logging.error(message)
logging.critical(message)
```

Q. Write a python program to create a log file and write WARNING and higher level messages?

```
import logging
logging.basicConfig(filename='log.txt',level=logging.WARNING)
logging.debug('Debug Message')
logging.info('Info Message')
logging.warning('Warning Message')
logging.error('Error Message')
logging.critical('Critical Message')
```

Default Level: WARNING
If we are not specifying file name: console

Bydefault in the log file the data will be appended. instead of appending if we want overwriting

filemode='a'
filemode='w'
Default value for file mode: a means append

Session - 15 Introduction to logging, logging levels and demo program
On 26-11-2020

impo

How to format log messages?

---

default format of log message:

level:name of logger: message

If we want to format, we should go for: format argument.

```
import logging
logging.basicConfig(format='%(levelname)s-%(message)s')
logging.critical('It is critical message')
logging.error('It is error message')
logging.warning('It is warning message')
logging.info('It is info message')
logging.debug('It is debug message')
```

How to add Timestamp in the log messages:

```
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s'
)
```

```
import logging
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s'
)
logging.critical('It is critical message')
logging.error('It is error message')
logging.warning('It is warning message')
logging.info('It is info message')
logging.debug('It is debug message')
```

```
2020-11-28 21:41:01,001:CRITICAL:It is critical message
2020-11-28 21:41:01,001:ERROR:It is error message
2020-11-28 21:41:01,001:WARNING:It is warning message
```

How to change date and time format:

```
----------------------------------
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s'
,
datefmt='%d-%m-%Y %I:%M:%S %p')
```

28/11/2020 09:41:01 PM

%I --->12 Hours time scale
%H --->24 Hours time scale

```
import logging
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s'
,
datefmt='%d-%m-%Y %I:%M:%S %p')
logging.critical('It is critical message')
logging.error('It is error message')
logging.warning('It is warning message')
logging.info('It is info message')
logging.debug('It is debug message')
```

```
28-11-2020 09:48:10 PM:CRITICAL:It is critical message
28-11-2020 09:48:10 PM:ERROR:It is error message
28-11-2020 09:48:10 PM:WARNING:It is warning message
```

```
import logging
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s'
,
datefmt='%d-%b-%Y,%A %I:%M %p')
logging.critical('It is critical message')
logging.error('It is error message')
logging.warning('It is warning message')
logging.info('It is info message')
logging.debug('It is debug message')
```

https://docs.python.org/3/library/time.html#time.strftime

https://docs.python.org/3/library/logging.html#logrecord-attributes

How to write Python program exceptions to the log file:

```
logging.exception(msg)

import logging
logging.basicConfig(
                filename='mylog.txt',
                level=logging.DEBUG,
                format='%(asctime)s:%(levelname)s:%(message)s',
                datefmt='%d-%m-%Y %I:%M %p'
          )
logging.info('A New Request Came')
try:
        x = int(input('Enter First Number:'))
        y = int(input('Enter Second Number:'))
        print('The Result:',x/y)

except ZeroDivisionError as msg:
        print('cannot divide with zero')
        logging.exception(msg)

except ValueError as msg:
        print('please provide int values only')
        logging.exception(msg)
```

logging.info('Request Processing Completed')

Session - 15 Introduction to logging, logging levels and demo program On 26-11-2020
Session - 16 How to format log messages with date and time and write exceptions to the log file On 28-11-2020

Problems with root logger:
----
root logger is the default logger.
Problems:

1. Once we set basic configuration then that configuration is final and we cannot change.

2. It will always work only for one handler either file or console but not both simultaneously.

3. It is not possible to configure logger with different configurations at differerent levels.

4. We cannot specify multiple log files for multiple modules /classes/methods

To overcome these problems we should go for our own customized logger.

Customized Logger creation and usage:

----

1. Creation of logger object and set loglevel

```
logger = logging.getLogger('demologger')
logger.setLevel(logging.DEBUG)
```

2. Creation of Handler object

There are multiple types of handlers like StreamHandler, FileHandler etc

```
consoleHandler=logging.StreamHandler()
consoleHandler.setLevel(logging.ERROR)
```

3. Creation of Formatter object

```
formatter=logging.Formatter('%(asctime)s-%(name)s-%(levelname)s-%(message)s',
                datefmt='%d-%m-%Y %I:%M:%S %p')
```

4. Add formatter to Handler

```
consoleHandler.setFormatter(formatter)
```

5. Add Handler to logger

```
logger.addHandler(consoleHandler)
```

6. Write messages by using logger object

```
logger.debug('debug message')
logger.info('info message')
logger.warning('warning message')
logger.error('error message')
logger.critical('critical message')
```

eg1:

```
import logging
logger = logging.getLogger('demologger')
logger.setLevel(logging.DEBUG)

consoleHandler=logging.StreamHandler()
consoleHandler.setLevel(logging.ERROR)


formatter=logging.Formatter('%(asctime)s-%(name)s-%(levelname)s-
%(message)s',
                datefmt='%d-%m-%Y %I:%M:%S %p')
consoleHandler.setFormatter(formatter)
logger.addHandler(consoleHandler)

logger.debug('debug message')
logger.info('info message')
logger.warning('warning message')
logger.error('error message')
logger.critical('critical message')
```

Note: Bydefault logger level will be available to handler. But we can define our own level at handler level which will be the final for that handler.

handler level should be supported by logger. ie logger log level should be lower than handler level.other wise only logger log level will be considered.

eg2:

```
import logging
logger = logging.getLogger('demologger')
logger.setLevel(logging.DEBUG)

fileHandler=logging.FileHandler('abc.log',mode='a')
fileHandler.setLevel(logging.ERROR)

formatter=logging.Formatter('%(asctime)s-%(name)s-%(levelname)s-
%(message)s',
                datefmt='%d-%m-%Y %I:%M:%S %p')

fileHandler.setFormatter(formatter)
logger.addHandler(fileHandler)

logger.debug('debug message')
logger.info('info message')
```

```
logger.warning('warning message')
logger.error('error message')
logger.critical('critical message')
```

eg3:

```
import logging
logger = logging.getLogger('demologger')
logger.setLevel(logging.DEBUG)

fileHandler=logging.FileHandler('abcd.log',mode='a')
fileHandler.setLevel(logging.ERROR)

consoleHandler=logging.StreamHandler()
consoleHandler.setLevel(logging.WARNING)


formatter1=logging.Formatter('%(asctime)s-%(name)s-%(levelname)s-
%(message)s',
            datefmt='%d-%m-%Y %I:%M:%S %p')

formatter2=logging.Formatter('%(asctime)s-%(message)s',
            datefmt='%d-%m-%Y %I:%M:%S %p')

fileHandler.setFormatter(formatter1)
consoleHandler.setFormatter(formatter2)
```

```
logger.addHandler(fileHandler)
logger.addHandler(consoleHandler)

logger.debug('debug message')
logger.info('info message')
logger.warning('warning message')
logger.error('error message')
logger.critical('critical message')
```

Session - 17 Need of Custom Logger, How to define and use Custom Logger On 30-11-2020

Use separate log file for every module:

---

test.py:

```
import logging
import student
logger = logging.getLogger('testlogger')
logger.setLevel(logging.DEBUG)

fileHandler=logging.FileHandler('test.log',mode='a')
fileHandler.setLevel(logging.DEBUG)

formatter=logging.Formatter('%(asctime)s-%(name)s-%(levelname)s-%(message)s',
                datefmt='%d-%m-%Y %I:%M:%S %p')
```

```
fileHandler.setFormatter(formatter)
logger.addHandler(fileHandler)

logger.debug('debug message from test module')
logger.info('info message from test module')
logger.warning('warning message from test module')
logger.error('error message from test module')
logger.critical('critical message from test module')
```

student.py:

```
------------
import logging
logger = logging.getLogger('studentlogger')
logger.setLevel(logging.DEBUG)

fileHandler=logging.FileHandler('student.log',mode='w')
fileHandler.setLevel(logging.ERROR)

formatter=logging.Formatter('%(asctime)s-%(name)s-%(levelname)s-
%(message)s',
                datefmt='%d-%m-%Y %I:%M:%S %p')

fileHandler.setFormatter(formatter)
logger.addHandler(fileHandler)

logger.debug('debug message from student module')
logger.info('info message from student module')
```

```
logger.warning('warning message from student module')
logger.error('error message from student module')
logger.critical('critical message from student module')
```

Generic Custom logger

Importance of inspect module:

----

inspect --->inspection

From which module/function call is coming ...

[

FrameInfo(frame=<frame at 0x00000248F5E7CC40, file 'D:\\durgaclasses\\demo.py', line 3, code getInfo>, filename='D:\\durgaclasses\\demo.py', lineno=3, function='getInfo', code_context=['\tprint(inspect.stack())\n'], index=0),

FrameInfo(frame=<frame at 0x00000248F5E74DD0, file 'test.py', line 3, code f1>, filename='test.py', lineno=3, function='f1', code_context=['\tgetInfo()\n'], index=0),

FrameInfo(frame=<frame at 0x00000248F5E7C440, file 'test.py', line 6, code <module>>, filename='test.py', lineno=6, function='<module>', code_context=['f1()\n'], index=0)

]

FrameInfo(

frame=<frame at 0x000002A956354DD0, file 'test.py', line 3, code f1>,
filename='test.py',
lineno=3,
function='f1',
code_context=['\tgetInfo()\n'],
index=0

)

test.py

```
--------
from demo import getInfo
def f1():
        getInfo()


f1()
```

demo.py:

```
---------
import inspect
def getInfo():
        #print(inspect.stack())
        #print(inspect.stack()[1])
        print('Caller Module Name:',inspect.stack()[1][1])
        print('Caller Function Name:',inspect.stack()[1][3])
```

Creation of Generic Logger & usage:

Generic Logger template

custlogger.py:

```
import logging
import inspect
def get_custom_logger(level):
        function_name = inspect.stack()[1][3]
        logger_name = function_name+' logger'
        logger = logging.getLogger(logger_name)
        logger.setLevel(level)
        fileHandler=logging.FileHandler('abc.log',mode='a')
        fileHandler.setLevel(level)
        formatter=logging.Formatter('%(asctime)s-%(name)s-
%(levelname)s-%(message)s',
                datefmt='%d-%m-%Y %I:%M:%S %p')
        fileHandler.setFormatter(formatter)
        logger.addHandler(fileHandler)
        return logger
```

test.py:

```
from custlogger import get_custom_logger
import logging
def logtest():
        logger = get_custom_logger(logging.DEBUG) #logtest logger'
        logger.debug('debug message from test module')
        logger.info('info message from test module')
        logger.warning('warning message from test module')
        logger.error('error message from test module')
        logger.critical('critical message from test module')
logtest()
```

student.py:

```
-----------
from custlogger import get_custom_logger
import logging
def logstudent():
        logger = get_custom_logger(logging.ERROR) #logstudent logger
        logger.debug('debug message from student module')
        logger.info('info message from student module')
        logger.warning('warning message from student module')
        logger.error('error message from student module')
        logger.critical('critical message from student module')
logstudent()
```

Same module but different loggers in different functions:

```
----------------------------------------------------------
```

custlogger.py:

```
--------------
import logging
import inspect
def get_custom_logger(level):
        function_name = inspect.stack()[1][3]
        logger_name = function_name+' logger'
        logger = logging.getLogger(logger_name)
        logger.setLevel(level)
        fileHandler=logging.FileHandler('abc.log',mode='a')
```

```
        fileHandler.setLevel(level)
        formatter=logging.Formatter('%(asctime)s-%(name)s-
%(levelname)s-%(message)s',
                datefmt='%d-%m-%Y %I:%M:%S %p')
        fileHandler.setFormatter(formatter)
        logger.addHandler(fileHandler)
        return logger
```

test.py:

```
--------
from custlogger import get_custom_logger
import logging
def f1():
        logger = get_custom_logger(logging.DEBUG)
        logger.debug('debug message from f1 function')
        logger.info('info message from f1 function')
        logger.warning('warning message from f1 function')
        logger.error('error message from f1 function')
        logger.critical('critical message from f1 function')

def f2():
        logger = get_custom_logger(logging.WARNING)
        logger.debug('debug message from f2 function')
        logger.info('info message from f2 function')
        logger.warning('warning message from f2 function')
        logger.error('error message from f2 function')
```

```
    logger.critical('critical message from f2 function')


def f3():
    logger = get_custom_logger(logging.ERROR)
    logger.debug('debug message from f3 function')
    logger.info('info message from f3 function')
    logger.warning('warning message from f3 function')
    logger.error('error message from f3 function')
    logger.critical('critical message from f3 function')

f1()
f2()
f3()
```

Creation of separate log file for every function:

---

where custom loggers are used in real applications

application

- **somewhere every activity we have to track**
- **somewhere only important activity we have to track**

project:
multiple modules
for every module maintain module specific log file

project:
For some modules File Handler is required
For some modules Console Handler is required

tkinter
mongodb
IDEs
numpy
pandas
matplotlib

Title: Creation of Generic Custom Logger and Usage

Need of separating logger configurations into a file or dict or json or yaml:
----

1. We can perform changes very easily
2. Reusability of configurations
3. length of the code will be reduced and readability will be improved.

logging_config.init:

```
--------------------
[loggers]
keys=root,demologger

[handlers]
keys=fileHandler

[formatters]
keys=sampleFormatter

[logger_root]
level=DEBUG
handlers=fileHandler

[logger_demologger]
level=DEBUG
handlers=fileHandler
qualname=demologger

[handler_fileHandler]
class=FileHandler
level=DEBUG
```

```
formatter=sampleFormatter
args=('test.log','w')

[formatter_sampleFormatter]
format=%(asctime)s:%(name)s:%(levelname)s:%(message)s
datefmt=%d-%m-%Y %I:%M:%S %p
```

```
import logging
import logging.config
logging.config.fileConfig("logging_config.init")
logger = logging.getLogger('demologger')
logger.critical('It is critical message')
logger.error('It is error message')
logger.warning('It is warning message')
logger.info('It is info message')
logger.debug('It is debug message')
```

```
Demo program for console handler:
---------------------------------
logging_config.init:
--------------------
[loggers]
keys=root,demologger

[handlers]
```

keys=consoleHandler

[formatters]
keys=sampleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_demologger]
level=DEBUG
handlers=consoleHandler
qualname=demologger

[handler_consoleHandler]
class=StreamHandler
level=ERROR
formatter=sampleFormatter
args=(sys.stdout,)

[formatter_sampleFormatter]
format=%(asctime)s:%(name)s:%(levelname)s:%(message)s
datefmt=%d-%m-%Y %I:%M:%S %p

test.py:
----
import logging
import logging.config

```
logging.config.fileConfig("logging_config.init")
logger = logging.getLogger('demologger')
logger.critical('It is critical message')
logger.error('It is error message')
logger.warning('It is warning message')
logger.info('It is info message')
logger.debug('It is debug message')
```

Title- Separating logger configurations into a config file