DATETIME

Python datetime contains below class for dealing with datetime.

1. **datetime.date**

An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: year, month, and day.

1. **datetime.time**

An idealized time, independent of any particular day, assuming that every day has exactly 24\*60\*60 seconds. (There is no notion of “leap seconds” here.) Attributes: hour, minute, second, microsecond, and tzinfo.

1. **datetime.datetime**

A combination of a date and a time. Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

1. **datetime.timedelta**

A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

1. **datetime.tzinfo**

An abstract base class for time zone information objects. These are used by the datetime and time classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

1. **datetime.timezone**

A class that implements the tzinfo abstract base class as a fixed offset from the UTC.

**Notes:**

The date, datetime, time, and timezone types share these common features:

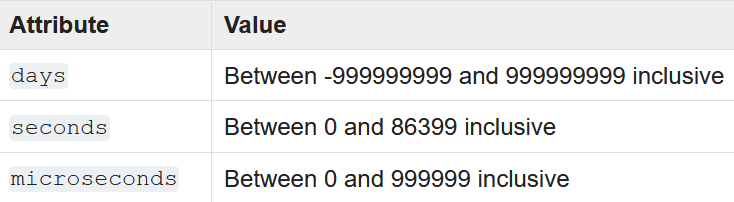
* Objects of these types are immutable.
* Objects of these types are hashable, meaning that they can be used as dictionary keys.
* Objects of these types support efficient pickling via the pickle module.

**timedelta**

datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)

all arguments are optional and default to 0.

On timedelta we have only below attributes accessible.



In case if we create any timedelta object using days, hours, .. but that will be converted into above three values and each can be accessed using by their attributed as listed above.

**Operation on timedelta object**

Timedelta object supports many operations few of them are-

* Addition
* Subtraction
* Multiplication by number(t1=t1\*mum)
* Division (division of two timedelta object i.e. t1/t2)
* Division (by number or float i.e. -- t1/f)
* abs(t) (create absolute value, something like mod in math)

delta = datetime.timedelta(days=50,seconds=27,microseconds=10,milliseconds=29000,minutes=5, hours=8,weeks=2)

print(delta.days) //64

print(delta.seconds) //29156

print(delta.hours) #this line will give error as hour is not a valid attribute

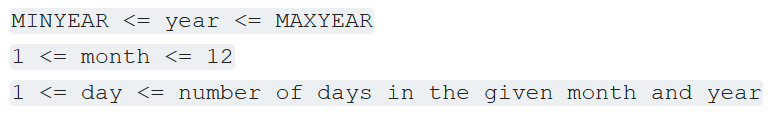
print(delta.microseconds) //10

**date**

A date object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions.

datetime.date(year, month, day) ---- constructor of creating date

---All arguments are required. Arguments must be integers and they must be in their range



January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on..

**Other constructor and class methods**

1. date.today()

Returns current local date

**Attributes of date class**

* date.year --- year between minyear and maxyear
* date.month ---- month between 1 and 12 inclusive
* date.day ---- day in integer

**Supported operations**

Date object have below supported operations

* addition of two date ---- date1+date2
* subtraction of two data ----- date1-date2
* comparison of two date --- date1 com\_operator date2

**Instance methods of date class**

It has many methods some commonly used are-

1. *date.replace(year=self.year, month=self.month, day=self.day)*

Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified

d = date(2002, 12, 31)

d.replace(day=26) #datetime.date(2002, 12, 26) day num. of d is replaced by give day num.

1. *date.weekday()* ---- returns the weekday, Monday as 0, Tues=1..
2. *date.ctime* ---- Returns string representations of date
3. *date.strftime(format)* --- Returns the string representation of date – in date, datetime class

**date.strftime(format) ----** this method is available indate, datetime class

This method is used to convert given datetime data into string as per specified format specifiers.

date=datetime.date.today()

str\_time=date.strftime('%Y-%m-%d') # gives data into YYY-MM-DD format

print(str\_time)

str\_time=date.strftime('%Y/%m/%d') # gives data into YYYY/MM/DD format

print(str\_time)

str\_time=date.strftime('%d-%m-%Y') # gives data into DD-MM-YYYY format

print(str\_time)

str\_time=date.strftime('%d/%m/%Y') # gives data into DD/MM/YYYY format

print(str\_time)

**strptime(date\_str, format\_of\_dat\_str)**

*This methods is available in datetime class only.*

This method is used to convert the give string into datetime object.

The output data or datetime object depends on the date\_str and format specifier.

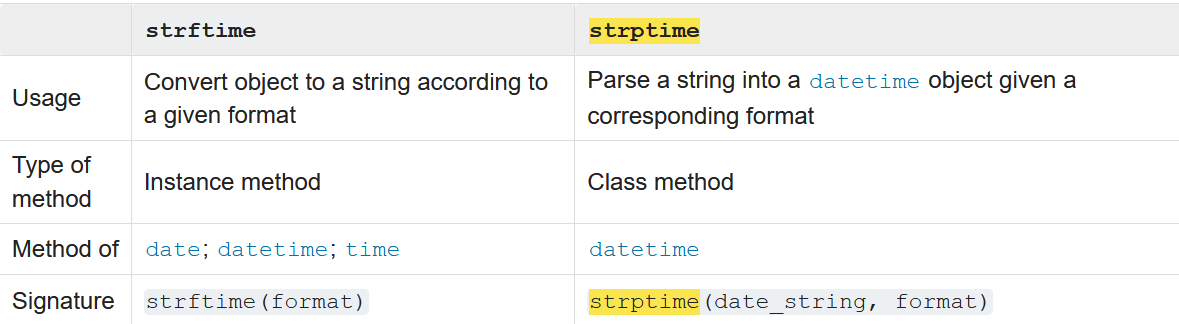
date\_str ----- string that we want to convert the string into date or datetime

format\_of\_date\_str ----- it specifies the in which format date\_str is given.

*This will always return date or datetime in YYYY-MM-DD or YYYY-MM-DD HH-MM-SS format*

*If date\_str doesn’t match with format\_specifier then it retuns ValueError.*

*In case if only datetime is needed then call date() on returned datetime object.*



datetime\_object = datetime.strptime('26032021', '%d%m%Y')

print(datetime\_object)

print(datetime\_object.date())

**Datetime**

A datetime object is a single object containing all the information from a date object and a time object.

datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, \*, fold=0)

* The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass
* The remaining arguments must be integers in the following ranges.

**Class methods of datetime**

It has below class methods.

* datetime.today()

Return the current local datetime, with tzinfo None

* datetime.now(tz=None)

Return the current local date and time\

* datetime.utcnow()

Return the current UTC date and time, with tzinfo None

* datetime.strptime(date\_string, format)

Return a datetime corresponding to date\_string, parsed according to format

**Attributes of datetime**

* datetime.year -------- year value from datetime object
* datetime.month -------- month value from datetime object
* datetime.day -------- day value from datetime object
* datetime.hour -------- hour value from datetime object
* datetime.minute -------- minute value from datetime object
* datetime.seconds -------- seconds value from datetime object
* datetime.microseconds -------- microseconds value from datetime object
* datetime.tzinfo -------- tzinfo value from datetime object

**instance methods of datetime**

datetime.date() ---------gives date value from datetime

datetime.time() ---------gives time value from datetime

datetime.timetz() ---------gives timetx value from datetime

datetime.replace() ---------same as previous

datetime.weekday() ---------gives weekday value from datetime

datetime.strftime() ---------check previous

**TIME**

A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a tzinfo object

datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None, \*, fold=0)

All arguments are optional. tzinfo may be None, or an instance of a tzinfo subclass.

**class attributes**

time.hour

time.second

time.microsecond

###############################################################################

# Multiprocessing #

###############################################################################

**Multithreading vs Multiprocessing**

In multithreading some unit of is done one by one for all threads. **It creates multiple threads.**

In multiprocessing each work is done separately. **It creates multiple process.**

**How multi-threading in Python works**

In python each process executes on a single core. when we create multiple threads of the same process each execute on the same core and thus share the resources and the memory space. To prevent one thread from altering the execution results of another thread since they share same resources Python has a concept of “Global Interpreter Lock”.

**How multi-processing in Python works**

In case of multi-processing, each process runs on different core depending upon the number of cores on the machine. Hence there’s no need of a GIL while doing multi-processing since all processes are independent.

Thus, multi-processing is actually providing us the actual parallelism in python.

Let’s take ax example :

def live\_tracker(x):

print('I am', x)

l = []

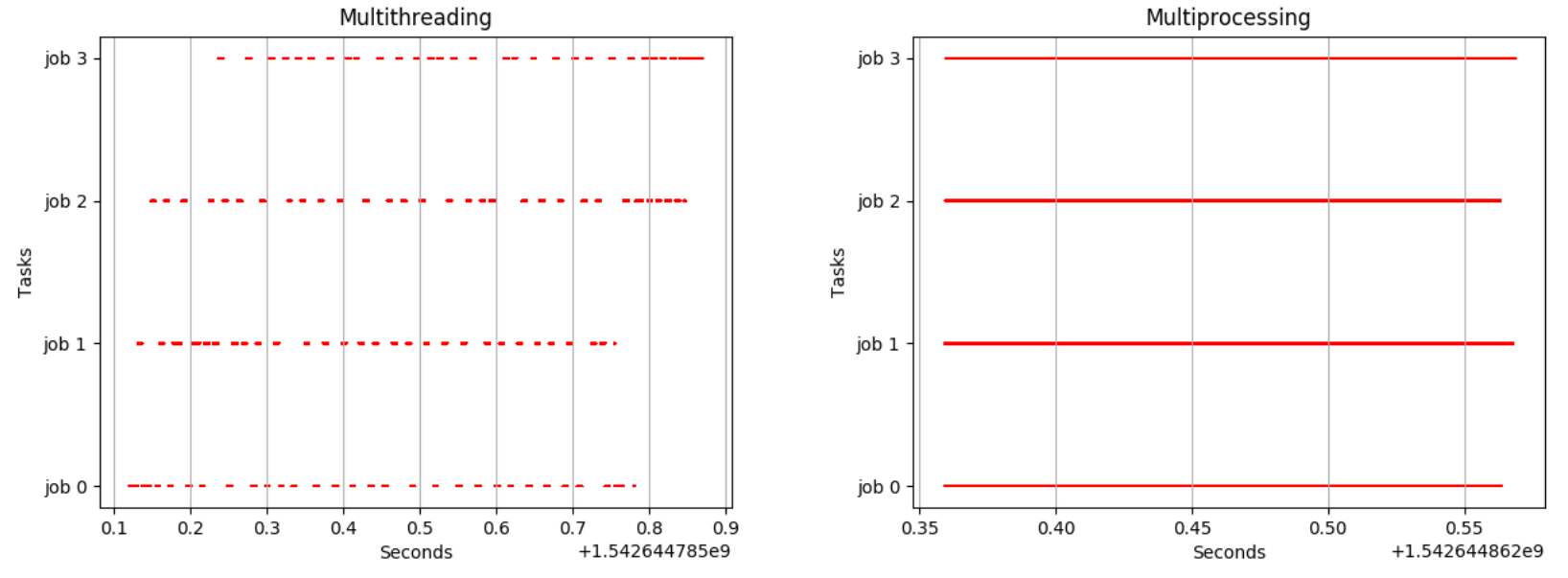
for i in range(10\*\*6):

l.append(time.time())

return l

visualize\_live\_runtimes(multithreading(live\_tracker, range(4), 4))

visualize\_live\_runtimes(multiprocessing(live\_tracker, range(4), 4))



**Difference between concurrency and parallelism**

**Concurrency** is essentially defined as handling a lot of work or different units of work of the same program at the same time.

**Parallelism** is Doing a lot of work of the same program at the same time to speed up the execution time.

There are 4 major class in multiprocessing-

1. Process
2. Pool
3. Lock
4. Queue

######################

# Process Clsss #

######################

class multiprocessing.Process(group=None, target=None, name=None, args=(), kwargs={}, \*, daemon=None)

**target** ---- > target is the callable object to be invoked by the run() method. It defaults to None, meaning nothing is called

**args** --- > args is the argument tuple for the target invocation, used to supply values to target.

**name** --- > name is the process name, if want to give any name.

**kwargs** ----- > kwargs is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only daemon argument sets the process daemon flag to True or False.

#######################

# Methods of Process class #

#######################

**run()**

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument.

Method representing the process’s activity.

**start()**

Start the process’s activity.

This must be called at most once per process object. It arranges for the object’s run() method to be invoked in a separate process.

**join([timeout])**

this method blocks until the process whose join() method is called terminates. If timeout is a positive number, it blocks at most timeout seconds.

**is\_alive()**

Return whether the process is alive.

**terminate()/kill()**

kill or terminate the process.

**Using/Implementing multiprocess**

We can do it in below way

1. Multiprocessing without any class
2. By extending process class

###############################

**#Multiprocessing without any class#**

###############################

if \_\_name\_\_ == '\_\_main\_\_':

p=proc(target=target\_name,args=(arguments))

p.start()

Except target all arguments are optional.

def func(a,b):

    print('func is called')

    print('sum of a and b is: ',a+b)

if \_\_name\_\_ == '\_\_main\_\_':

    p=proc(target=func,args=(2,3))

    p.start()

    p.join()

    print(p.is\_alive())

###############################

**# By extending process class #**

###############################

Inherit Process class in own class and override run(), create instance of own class and call start()

Class Myclass(Process):

def run(self)

p=MyClass()

p.start()

Note:

Better don’t use it b/c process is not stopping or getting killed even after execution , I don’t know why.

from multiprocessing import Process as proc

class MyProcess(proc):

    def run(self):

        print('run method is called')

if \_\_name\_\_ == '\_\_main\_\_':

    p=MyProcess()

    p.start()

##################

# Lock class #

##################

This is used for locking or unlocking resources, used typically where we are creating many processes and each process is using one particular resource.

e.g – Let we have two process ( for addition and subtraction), and each process using one shared variable, if we don’t use lock then we will not get correct result b/c by the time one process is running another process can modify the value.

Example:-

Here we are trying add and subtract operation on one object(Value) value. Here we have not used lock then at end value of shared variable will changed.

from multiprocessing import Process as proc

from multiprocessing import Lock, Value

def add\_num(num):

    for i in range(100):

        num.value+=5

def sub\_num(num):

    for i in range(100):

        num.value-=5

if \_\_name\_\_ == '\_\_main\_\_':

    number=Value('i',500)

    p1=proc(target=add\_num,args=(number,)) #first call add operstion

    p2=proc(target=sub\_num,args=(number,)) #Now call subtract operation

    p1.start()

    p2.start()

    p1.join()

    p2.join()

**print(number.value) # here we will get any result but expecting 500**

**Using lock**

from multiprocessing import Process as proc

from multiprocessing import Lock, Value

def add\_num(num,lock):

    for i in range(100):

        lock.acquire() # lock variable

        num.value+=5 # add 5

        lock.release() # release the lock for next addition

def sub\_num(num,lock):

    for i in range(100):

        lock.acquire() # lock variable

        num.value-=5 # add 5

        lock.release() # release the lock for next addition

if \_\_name\_\_ == '\_\_main\_\_':

    number=Value('i',500)

    lock=Lock()

    p1=proc(target=add\_num,args=(number,lock)) # Add some number

    p2=proc(target=sub\_num,args=(number,lock)) #Subtract same number

    p1.start()

    p2.start()

    p1.join()

    p2.join()

    print(number.value) # here at end we will get 500 always

class multiprocessing.Lock

A non-recursive lock object

Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released

Methods of lock class

1. **acquire(blok=True,timeout=None)**

Acquire a lock, blocking or non-blocking.

With the block argument set to True (the default), the method call will block until the lock is in an unlocked state.

When invoked with a positive, floating-point value for timeout, block for at most the number of seconds specified by timeout as long as the lock can not be acquired.

1. **release()**

Release a lock. This can be called from any process or thread, not only the process or thread which originally acquired the lock.

**class multiprocessing.RLock**

A recursive lock object: a close analog of threading.RLock. A recursive lock must be released by the process or thread that acquired it. Once a process or thread has acquired a recursive lock, the same process or thread may acquire it again without blocking; that process or thread must release it once for each time it has been acquired

It had also acquire() and release().

################################

# Pool class #

################################

One can create a pool of processes which will carry out tasks submitted to it with the help of Pool class.

class multiprocessing.pool.Pool([processes[, initializer[, initargs[, maxtasksperchild[, context]]]]])

**processes** is the number of worker processes or core/cpu to use. If processes is None then the number returned by os.cpu\_count() is used.

**Methods of Pool**

**apply** Call func with arguments args. It blocks until the result is ready.

apply(func[, args[, kwds]])

**apply\_async** It is better suited for performing work in parallel. It returns a AsyncResult object

apply\_async(func[, args[, kwds[, callback[, error\_callback]]]])

**map** A parallel equivalent of the map() built-in function (it supports only one iterable argument though, for multiple iterables). But it blocks. **Actually it created separate process foreach values.**

map(func, iterable[, chunksize])

**map\_async** It is better suited for performing work as map in parallel. it returns a AsyncResult object

map\_async(func, iterable[, chunksize[, callback[, error\_callback]]])

**Whenever we call above function on any pool object it created all process and runs**

**close()**

Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

**terminate()**

Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected terminate() will be called immediately.

**join()**

Wait for the worker processes to exit. One must call close() or terminate() before using join().

Example

def find\_square(num):

   return num\*num

if \_\_name\_\_ == '\_\_main\_\_':

    numbers=[1,2,3]

    p=Pool()

    result=p.map(find\_square,numbers) #run the find\_swuare method

    print(result)

####################

# Queue class #

###################

This is mainly used to share the data b/w process.

It is based on FIFO data structure.

from multiprocessing import Process as proc, process

from multiprocessing import Pool, Queue

from multiprocessing.context import Process, ProcessError

def find\_square(num,q):

    for i in num:

        q.put(i\*i) # append the result in queue instance

if \_\_name\_\_ == '\_\_main\_\_':

    q=Queue()

    numbers=[1,2,3]

    p=Process(target=find\_square,args=(numbers,q))

    p.start()

    p.join()

    while q.empty() is False:

        print(q.get()) #get data one by one from queue object

################################################################################

# Enumeration/enum in python #

################################################################################

Python have module 'enum' for enumeration.

This module defines four enumeration classes that can be used to define unique sets of names and values: Enum, IntEnum, Flag, and IntFlag

**What is enum class or enumerations.**

An enum is a user-defined type consisting of a set of named constants called enumerators. The idea is that instead of using an int to represent a set of values, a type with a restricted set of values is used instead.

Using enums increase the level of abstraction and lets the programmer think about what the values mean rather than worry about how they are stored and accessed. This reduces the occurrence of bugs.

**Creating an Enum**

Creating enum have same syntax as how to create class by inheriting 'enum'.

Example of enum

from enum import Enum

class Color(Enum):

RED = 1

GREEN = 2

BLUE = 3

NAME=’Rakesh’

**Note: Enum member values**

Member values can be anything: int, str, etc.

If the exact value is unimportant, you may use auto instances and an appropriate value will be chosen for you. Care must be taken if you mix auto with other values.

Enum class can’t be instantiated though it’s of no use if can be.

**Note Nomenclature**

* The class Color is an enumeration (or enum)
* The attributes Color.RED, Color.GREEN, etc., are enumeration members (or enum members) and are functionally constants.
* The enum members have names and values (the name of Color.RED is RED, the value of Color.BLUE is 3, etc.)

**Property of enumeration**

* If you have an enum member and need its name or value, then we can access using attribute 'name' and 'value'
* Enum members should be always unique, value can be duplicated.

Color.RED.name --> RED # accessing enum variable name

Color.RED.value -----> 1 # access value

* Two or more member of a enum class can have same value (if unique decorator not impl.)
* Enumeration members are hashable, so they can be used in dictionaries and sets.
* Enumerations support iteration, in declaration order ( i.e member will iterate in FIFO)

**Iteration of enum**

We can iterate over enums using for loop or any other looping statement.

class Color(Enum):

    RED = 1

    GREEN = 2

    BLUE = 3

for each in Color:

    print(each,'=',each.value)

**Note:**

While iterating more than one variable with same value comes only once.

class Color(Enum):

    RED = 1

    GREEN = 2

    BLUE = 3

    YELLOW=3

print(list(Color)) # [<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 3>]

We can see here YELLOW is missing b/c have duplicate values.

**Output:**

Color.RED = 1 # getting RED first as RED is defined as first member in Color

Color.GREEN = 2 # getting GREEN after RED b/c GREEN is defined after RED

Color.BLUE = 3

**Ensuring unique enumeration values**

By default, enumerations allow multiple variable names as aliases for the same value. When we don’t want this behavior then we can use 'unique' decorator from enum.

@unique

class Color(Enum):

    RED = 1

    GREEN = 2

    BLUE = 3

    YELLOW=3

**Note:**

Here we will get error as value 3 is being assigned to two variables. We will get below error-

ValueError: duplicate values found in <enum 'Color'>: YELLOW -> BLUE

**Using automatic values**

If the exact value is unimportant you can use auto.

@unique

class Color(Enum):

    RED = 1

    GREEN = 2

    BLUE = 3

    YELLOW=auto() #assigning automatic value to YELLOW member

for each in Color:

    print(each,'=',each.value)

**Comparison of enums**

1. Enumeration members are compared by identity operator (is, is not)
2. Ordered comparisons between enumeration values are not supported.
3. Equality comparisons are defined though ( of enum member not for values)

class Color(Enum):

    RED = 1

    GREEN = 2

    BLUE = 3

    YELLOW=3

print(Color.RED is Color.RED) #True , Rule 1

print(Color.RED is Color.BLUE) #False, Rule is 1

print(Color.BLUE is Color.YELLOW) #True , Rule 1

print(Color.RED.value == 1) #True , Rule 2

print(Color.BLUE == Color.RED) # Rule 3

print(Color.BLUE == Color.YELLOW) #Rule 3

**Allowed members and attributes of enumerations**

1. Enumerations are like Python classes, and can have methods and special methods as usual

from enum import Enum, auto, unique

class Color(Enum):

    RED = 1

    GREEN = 2

    BLUE = 3

    YELLOW=3

    def describe(self):

        print('i am describing about',self.value)

    @classmethod

    def class\_method(cls):

        print('i am in class method')

        print(cls.RED.value)

print(Color.RED.value)

print(Color.RED.describe)

Color.class\_method()

1. A new Enum class must have one base Enum class, up to one concrete data type, and as many object-based mixin classes as needed. The order of these base classes is as below-

class EnumName([mix-in, ...,] [data-type,] base-enum):

pass

1. A inherited enum class can be inherited in another class if enumeration (parent class) does not define any members

###################################

# Derived Enumerations --- IntEnum #

###################################

The first variation of Enum that is provided is also a subclass of int. Members of an IntEnum can be compared to integers; by extension, integer enumerations of different types can also be compared to each other.

class Color(Enum):

    RED = 1

class Shape(IntEnum):

    RED = 1

print(Color.RED==1) # False, b/c not inherited IntEnum

print(Shape.RED==1) #True, b/c inheriting IntEnum

**Note:**

* Member must be python number (integer, float, double float etc) data type.

###################################

# Derived Enumerations --- IntFlag #

###################################

The next variation of Enum provided, IntFlag, is also based on int. The difference being IntFlag members can be combined using the bitwise operators (&, |, ^, ~) and the result is still an IntFlag member.

However, as the name implies, IntFlag members also subclass int and can be used wherever an int is used.

Any operation on an IntFlag member besides the bit-wise operations will lose the IntFlag membership.

###################################

# Python enumerate() #

###################################

The enumerate() method adds counter to an iterable and returns it (the enumerate object).

enumerate(iterable, start=0) -------------- **Synatax**

* **iterable** - a sequence, an iterator, or objects that supports iteration.
* **start** (optional) - enumerate() starts counting from this number. If start is omitted, 0 is taken as start. Adds 1 on each iteration

grocery = ['bread', 'milk', 'butter']

enumerateGrocery = enumerate(grocery)

print(type(enumerateGrocery))

# converting to list

print(list(enumerateGrocery)) #[(0, 'bread'), (1, 'milk'), (2, 'butter')]

# changing the default counter

enumerateGrocery = enumerate(grocery, 10)

print(list(enumerateGrocery)) #[(10, 'bread'), (11, 'milk'), (12, 'butter')]

###########################

# Garbage collection #########

###########################

Garbage collection (GC) is a process of identifying and reclaiming the memory used by objects that are no longer reachable or referenced by the program. In other words, GC cleans up the memory that is not being used anymore, and then makes it available for new objects

An Object is suitable to be destroyed by garbage collector if it is not referenced by any variable.

Python uses a hybrid approach of reference counting and generational garbage collection to manage the memory.

Reference counting works well for most cases, but it has a limitation: it cannot detect circular references.

Example:

In below code we have circular reference hence reference counting will not work here to destry the object by garbage collector, it will be done by generational garbage.

# create two objects  
a = [1, 2, 3]  
b = [4, 5, 6]  
# create a circular reference  
a.append(b)  
b.append(a)  
# delete the references  
del a  
del b

**Generational garbage collection**

Generational garbage collection is a type of trace-based garbage collection. It can break cyclic references and delete the unused objects even if they refer to themselves.

**Destructor**

In python destructor is the method which is called by garbage collector before any object is destroyed. The purpose of destructor is to do cleanup(resource deallocation) activity if any, before destroying the object.

destructor method syntax:

def \_\_del\_\_(self):

#

#

**Note: --- very impt.**

* When execution is done of python code then garbage collector will destroy the all objects (if any object still exists) at end.
* Every time any object is deleted/destroyed destructor will be called by garbage collector.