Intermediate

09 April 2025 11:51

- Intermediate course:
  - lists:
    - list.append("value")
    - list.clear() : clears all the value and returns empty list
    - list.insert(position, value)
    - list.reverse() : order gets reversed
    - list.sort()   : in ascending order
    - new_list = sorted(old_list) : doesn't change old list
    - list = [0] * 5 : [0,0,0,0,0]
    - list = list1 + list2
    - list = my_list[1:5] : 1 to 4 copied
    - [:5] : start from beginning
    - [::2] : step 2 value, [::-1] to reverse the list
    - list_cpy = list_org : will affect orginial list (same memory)
    - list_cpy = list_org.copy() : same as list(list_org), list_org[:]
    - list_sqr = [I * I for I in list]
    - index out of range error possible
    - remove() through error if not found

  - Tuples: ordered, immutable
    - mytuple = ("max" , 28) : same as mytuple = "max", 28
    - tuple = ("max",) : for one element
    - type(variable_name)
    - index out of range error possible
    - mytuple[0] = "madhu" : not possible
    - if "max" in tuple :
      ```
          print("Yes")
      else:
          print("no")
      //possible
      ```
    - print(len(my_tuple))
    - print(my_tuple.count('Madhu')) : shows the no. of Madhu present
    - my_tuple.index('l') :value error if not found
    - my_list = list(my_tuple) : changed to modify and use tuple() function
    - slicing is possible my_tuple[3:5] : default step 1
    - name, age, city = my_tuple : valueError if nos. doesn't match
    - *age used for unpacking and in excess converted to list
    - Large data sets are faster in tuples than list
      - import timeit
        print(timeit.timeit(list_name, number = '100000')) : will get the time required for making this list/array {list = tuple*16}

    - sys.getsizeof(array_name) : list>tuples

  - Dictionary: Key-value pair, mutable, unordered
    - mydict = { "name" : "Max" , "age" : 28 , "city" = "New York" }
    - dict(name="Max", city = "Boston", age = 38) : no quotes for keys
    - my_dict["last_name"] :will through error if not present
    - my_dict["email"] = "madhu.com" : updating values
    - del my_dict["key name"]
    - mydict.pop("key name")
    - mydict.popitem() : will delete the last inserted item
    - if "name" in mydict:
      ```
          print(mydict["name"]) // will produce error if not present
      ```
    - try:
      ```
          print(mydict["name"])
      except:
          print("error")
      ```
    - for key in mydict:
      ```
          print(key)
      for key in mydict.keys():
          print(key)
      //both are the same
      ```
    - for key, value in mydict.items():   : for getting both key and value
    - my_dict_cpy = my_dict : will affect the original
    - So : dict(my_dict) or my_dict.copy()

        - mydict.update(mydict2)

        - my_dict = { 3:9 , 6:36, 9:81 }
          value = my_dict[3] //produces value 9
        - keys can be tuple but not list as they are mutable
    - Sets: unordered , mutable, no duplicates inside
        - myset = set("Hello") : {'l', 'o', 'H', 'e'} and no 2 l's are present
        - myset.add(3) : will add to set
        - myset.discard(5) : no error if not present
        - myset.pop() : possible
        - Itteration:
              for I in myset:
                  print(i)
        - if 1 in myset:
              print("yes")
        - u = odds.union(evens) : all numbers
        - I = odds.intersection(evens) : empty set
        - diff = setA.difference(setB)
        - diff = setA.symetric_difference(setB) will give (A U B) - (A intersection B)
        - Last 4 returns values and no updation
        - setA.update(setB) will add A with unique elements
        - setA.intersection_update(setB)
        - setA.difference_update(setB)
        - setA.symmetric_difference_update(setB)
        - print(setA.issubset(setB)) : setA a subset of setB or not?
        - setA.issuperset(setA) :
        - setA.isdisjoint(setB) : checks for no element
        - setA = setB : will make a link
        - So : setA = setB.copy() or set(setB)
        - a = frozenset([1, 2, 3, 4]) : can't add or remove but union will work
    - String:
        - "Hello World" : 'string' : 'string\'s' : or """ multi line """" or : " ' " // will print single quotes inside
        - my_string[0] : indexing
        - my_string[0] = 'h' : error immutable
        - substring = string [1:5] : slicing(step value is also present)
        - name = "Tom " + greeting : concatination(space is required)
        - for I in greeting:
              print(I) //will print all the elements in string
        - if 'e' in greeting:
        - my_string = my_string.strip() : as strings are immutable
        - my_string.lower() : .upper() : .startswith("char_or_word") : endswith()
        - my_string.find('o') : first index of occurance
        - my_string.count('o')
        - my_string.replace('Old_word', "new_word") : does not change the original string
        - my_string = "How are you doing"
          my_list = my_string.split()
          //demeliter is space. : my_string.split(",") for csv files
          new_string = ''.join(my_list) // without any space if we need space for all elements use
          ' '.join(my_list)
        - from module import function as new_name_for_function
    - Formating:
        - %, .format() and F strings
        - my_string = " Hello %s" %var
              %d (decimal: truncates if floating), %.3f (floating: by default 6 point for floating values)
        - my_string = "Hello {:.2f} ".format(var)
            - □ var is a default value as float which :.2f is for 2 decimal places
            - □ "{} and {} ".format(var,var2) for multiples place holders
        - my_string = f"the variables is {var} and {var2}"
            - □ F strings are faster as they get evalutes at run time
    - Collections Module:
        - Counter: stored as dict
              from collections import Counter
              a = "aaaaabbbcccc"
              my_counter = Counter(a)
              print(my_counter) // will give all the key value pairs //my_counter.keys()
              print(my_counter.most_common(2)) //will give 2 most used letters
        - namedtuple:
              from collections import namedtuple
              Point_cal = namedtuple('Point','x,y')

```
pt = Point_cal(1,-4)
print(pt.x , pt.y)    //prints the coordinates
```

    - OrderedDict

      ```
      from collections import OrdereDict
      ordered_dict = OrderedDict()
      ordered_dict['a'] = 1
      ordered_dict['b'] = 2
      ordered_dict['c'] = 3
      print(ordered_dict) // will always maintain the order and does not change
      ```

    - defaultdict: sets a default value for key if not given

      ```
      from collections import defaultdict
      d = defaultdict(int)
      d['a'] = 1
      d['b'] = 2
      print(d['c']) // will print 0 as the default value is integer and does not produce
      error // float : 0.0 // list : []
      ```

    - deque:

      ```
      from collection import deque
      d = deque()
      d.append(1)
      d.append(2)
      d.appendleft(3) // will append elements to left side
      d.pop() //d.popleft()
      d.clear() // clears all element
      d.extend([4, 5 , 6]) //extendleft() and 6 would be the very left element in the list
      d.rotate(1) //shift all element 1 place to the left negative numbers also given
      ```

- Itertools: Used in for loop
    - product:

      ```
      from itertools import product
      a = [ 1, 2 ]
      b = [ 3, 4 ]
      prod = product(a, b, repeat = 2)
      print(list(prod))
      ```

        - when b = [3] alone as the repeat value is 2

          ```
          [(1, 3, 1, 3), (1, 3, 1, 4), (1, 3, 2, 3), (1, 3, 2, 4),
           (1, 4, 1, 3), (1, 4, 1, 4), (1, 4, 2, 3), (1, 4, 2, 4),
           (2, 3, 1, 3), (2, 3, 1, 4), (2, 3, 2, 3), (2, 3, 2, 4),
           (2, 4, 1, 3), (2, 4, 1, 4), (2, 4, 2, 3), (2, 4, 2, 4)]
          ```

    - permutations:

      ```
      from itertools import permutations
      a = [1, 2, 3]
      perm = permutations(a)
      print(list(perm))
      ```

        - permutations(a , 2) can be given for ordered sets of 2
        - [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]

    - combinations:

      ```
      from itertools import combinations
      a = [1, 2, 3, 4]
      comb = combinations(a,2)
      print(list(comb))
      ```

        - when used combination with replacement function (1,1) and the respective elements will be added

    - accumulates: sums

      ```
      from itertools import accumulate
      a = [1, 2, 3, 4]
      acc = accumulate(a)
      print(a)
      print(list(acc))
      ```

        - acc = accumulate(a, func = operator.mul)
        - acc = accumulate(a, func = max)

    - groupby:

      ```
      from itertools import groupby
      def greater_than_3(s):
          return s>3
      a = [1, 2, 3, 4]
      group_obj = groupby(a, key=greater_than_3)
      for key, value in group_obj:
          print( key, list(value))
      ```

    - count,cycle and repeat:

```
from itertools import count
for I in count(10):  // starting value of index
        print(i)
        if I == 15:
                break
```

- Output:

```
10
11
12
13
14
15
```

- cycle(list) is to loop any list infinitely until break conditions
- repeat(1) will infinetly get the value 1

- How to capture time for a step:
  - from timeit import default_timer as timer

```
start = timer()
<code>
stop = timer()
print(stop - start)
```

- Lambda :: one line function without name

```
add10 = lambda x:x+10
print(add10(5))

mult = lambda x,y : x*y
print(mult(2,7))
```

  - Using sorted functions:

```
points2D = [(1,2) , (15,1) , (5,-1)]
points2D = sorted(points2D, key = lambda x : x[1])
// will print the in ascending order with y coordinates( in key not mentioned with x coordinates)
a = [1, 2, 3, 4]
b = map(lambda x : x*2, a)    // b= [x*2 for x in a]
print(list(b))   //list(b) is important
c = filter(lambda x : x%2 == 0 , a) //only even number// c = [x for x in a if x%2 ==0]
```

  - reduce::

```
from functools import reduce
a = [1, 2, 3, 4]
product_a = reduce (lambda x,y : x*y , a)
print(product_a) //24
```

- Errors and exceptions:
  - TypeError: a= 5 + '10'
  - ModuleNotFoundError : if module is wrong
  - NameError : b = c // without defining c
  - FileNotFoundError : no file exists
  - a = [1, 2, 3, 4]
    a.remove(5) //ValueError
  - a[5] //IndexError
  - my_dict = { 'name' : 'Max' }
    my_dict['age'] //KeyError
  - Code:

```
x = -5
if  x<0:
        raise Exception('x should be positive')
//assert (x>0), 'x is not positive'     ////will through an error msg if condition not satisfied
try :
        a= 5/0
except :  //except Exception as error_msg: //// will capture the error for printing
        print('an error happened')
except ZeroDivisionError as e:
        print(e )
except TypeError as e:
        print ( e )
else :
        print( "everything is fine")
finally:
        print("cleaning up ...")
```

- □ Defining exceptions:

```
        class ValueTooHighError(Exception) :
                def __init__(self, message, value):
                        self.message = message
                        self.value = value
        def test_value(x):
                if x > 100:
                        raise ValueTooHighError('value is too high', x)
                if x< 10 :
                        raise ValueTooLowError('value is too low', x)
        try:
                test_value(200)
        except ValueTooHighError as e:
                print(e )
        except ValueTooLowError as e:
                print(e.message, e.value)
```

- ▪ Logging:

```
import logging
logging.debug("This is a debug message")
//.info
.error
.warning
.critical
//only .error , .warning , .critical msg will be displayed
// can add time ,level name, format for date and time, and error msg : refer
logging.basicConfig()
```

  - □ IN main.py file:

```
import logging
logger = logging .getLogger(__name__)
//logger.propagate = False ////this won't allow the helper module to access this
file , default = True
logger.info('hello from helper')

IN helper.py file:
import logging
logging.basicConfig(level = logging.DEBUG)
import helper
//Output: helper - INFO - hello from helper

Example 2:
import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
//this will print all the msgs as the level is set to DEBUG
```

- ▪ Log handler:
  - □ import logging

```
logger = logging.getLogger(__name__)
#Creating handler
stream_h = logging.StreamHandler()
file_h = logging.Filehandler("file.log")
#level and the format
stream_h.setLevel(logging.WARNING)
file_h.setLevel(logging.ERROR)

formatter = logging.Formatter("%(name)s - %(levelname)s - %(message)s")
stream_h.setFormatter(formatter)
file_h.setFormatter(formatter)
```

```
logger.addHandler(stream_h)
logger.addHandler(file_h)

logger.warning('this is a warning')
logger.error('this is an error')

# this will create 'file.log' file to store error msg's and not warning as they were
not mentioned and both will be displayed in the terminal

#can be done using config files also
```

- logging using try and except:
  - □ import logging
    ```
    try:
            a = [1, 2, 3]
            val = a[4]
    except:
            logging.error(e, exc_info = True # will include the stack trace {for all details})
            # same can be done using traceback method
            # RotatingFileHandler used for different log files based on the memory
            constraints of each log file generated
            #time.sleep(5) will stop the code for 5 seconds
    ```

- ○ JSON :: (java script object notation)
  - Encoding:(converting dict to object)
    - □ import jsom
      ```
      person = { "name" : "John" and some other values}
      personJSON = json.dumps(person, # indent = 4 will set spaces between 2 key value
      pair)
      print(personJSON)

      with open ('person.json' , 'w' ) as file:
              json.dumps(person, file) # will create a person.json file and dump person
              dict as object inside it
              json.load(file) # in 'r' mode will get the data from json file to python
      ```

- ○ Random Numbers:
  - Pseudo Random numbers:
    ```
    import random
    a = random.random()#0 to 1
    print(a)
    #random.uniform(1,10): float value
    #random.randint(1,10) : also include 10
    #random.randrange(1,10) : doesn't include 10
    #random.normalvariate(0, 1) : mean 0, variance 1
    ```
  - Get a random character from a string:
    ```
    mylist = list("ABCDEFGH")
    a = random.choice(mylist)
    print(a)
    ```
  - Get a list of (length=3) random characters from string (without duplicate):
    ```
    mylist = list("ABCDEFGH")
    a = random.sample(mylist, 3)
    print (a)
    ```
  - Get a list of (length=3) random characters from string (with duplicate):
    ```
    mylist = list("ABCDEFGH")
    ```
    - □ a = random.choices(mylist, k=3)
    - □ print(a)
  - Shuffle a given list :
    - □ random.shuffle(mylist)
    - □ print(mylist)
  - How to get the random value multiple times?
    - □ random.seed(1)
    - □ print(random.random())
    - □ random.seed(1)
    - □ print(random.random()) // will get the same values if we same seed value. But is

not good in terms of security

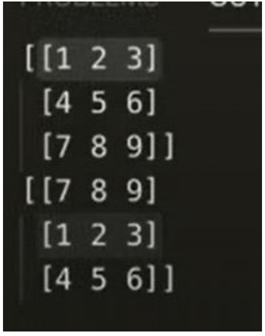
- How to make true random values:
  - □ Use secrets module : import secrets

- Make a random matrix with random values:
  - □ import numpy as np

```
a = np.random.randint(0,10,(3,5))
print(a) //will get a [3*5] matrix with values in the 1 to 10
```

- Shuffle just the x-axis components of list:

```
import numpy as np ##Also has seed values
arr = np.array([ [1,2,3], [4,5,6], [7,8,9]   ] )
np.random.shuffle(arr)
print(arr)
```



```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[7 8 9]
 [1 2 3]
 [4 5 6]]
```

- ○ Decorators:(Add new functionality to a already existing function)

  Examples: Add time of execution, debug, get more conditions satisfied

  Basic sytax:

  - •

```
def start_end_decorator(func):
    def wrapper():
        print("Start")
        func()
        print("End")
    return wrapper
def print_name():
    print("Madhu")
print_name_with_decorator = start_end_decorator(print_name)
print_name_with_decorator()

Output: Start
    Madhu
    End
```

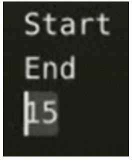
  - •

```
def start_end_decorator(func):
    def wrapper():
        print("Start")
        func()
        print("End")
    return wrapper

@start_end_decorator
def print_name():
    print("Madhu")

print_name()
// will aslo give the same result as above
```

- Add input values inside function decorator:

```
def start_end_decorator(func):
    def wrapper(*args, **kwargs):
        print("Start")
        result = func(*args, **kwargs)
        print("End")
        return result
    return wrapper

@start_end_decorator
def add5(x):
    return x+5
```

```
result = add5(10)
print(result)
```



```
Start
End
15
```

- Decorators can also get values:

```
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                func(*args, **kwargs)
        return wrapper
    return decorator_repeat

@repeat(num_times=3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
OUTPUT:
Hello, Alice!
Hello, Alice!
Hello, Alice!
```

- Using functools for preserving data :

```
import functools

def start_end_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("Start")
        result = func(*args, **kwargs)
        print("End")
        return result
    return wrapper

@start_end_decorator
def add5(x):
    """Adds 5 to the input."""
    return x + 5

print(add5.__name__)  # Output: add5
print(add5.__doc__)   # Output: Adds 5 to the input.
```

- import functools

```
def repeat(num_times):
        def decorator_repeat(func):
                @functools.wraps(func)
                def wrapper(*args,**kwargs):
                        for _ in range (num_times):
                                result = func(*args, **kwargs)
                        return result
                return wrapper
        return decorator_repeat

@repeat(num_times=3)
def greet(name):
        print(f'Hello {name}')

greet("Alex")
```

- Another Example:

```
import functools
def start_end_decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            print("Start")
            result = func(*args, **kwargs)
            print("End")
            return result
        return wrapper

    def debug(func):
        @functools.wrap(func)
        def wrapper(*args, **kwargs):
            args_repr = [repr(a) for a in args]
            kwargs_repr = [f"{k} = {v!r}" for k,v in kwargs.item()]
            signature = ", ".join(args_repr + kwargs_repr)
            print (f"Calling {func.__name__}({signature})")
            result = func(*args, **kwargs)
            print (f"{func.__name__!r} returned {result!r}")
            return result
        return wrapper

    @debug
    @start_end_decorator
    def say_hello(name):
        greeting = f'Hello {name}'
        print(greeting)
        return greeting

    greet('Alex')
    #repr(a) is a built-in function that returns a string representation of the object a.
    This representation is often useful for debugging because it includes information
    about the type and value of the object.
    #v!r applies the repr function to the value v, producing a string representation of
    v.
```

- Output:

```
Calling say_hello('Alex')
Start
Hello Alex
End
'say_hello' returned 'Hello Alex'
```

- Class Decorator:

```
    class CountCalls:
        def __init__(self,func):
            self.func = func
            self.num_calls = 0
        def __call__(self, *args, **kwargs):
            self.num_calls +=1
            print(f"This is executed {self.num+} times")
            return self.func(*args, **kwargs)

    @CountCalls
    def say_hello():
        print('Hello')

    say_hello()
    say_hello()
```

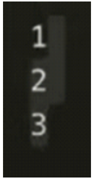
- Output:
  - ◇

```
This is executed 1 times
Hello
This is executed 2 times
Hello
```

- Generators:
  - return a object when asked for and can iterate inside and more efficient
  - Example:

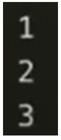
```
    def mygenerator():
        yield 1
        yield 2
        yield 3
    g = mygenerator()
```

```
for I in g :
    print(i)
```



```
1
2
3
```

```
value = next(g)
print(value)
value = next(g)
print(value)
value = next(g)
print(value)
```

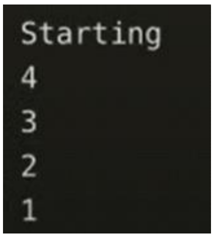


```
1
2
3
```

```
value = next(g)
print (value) # for the 4th time will produce error when no yield is found

print(sum(g)) # Output: 6
print(sorted(g)) # [1,2,3]
```

- Example:

```
def countdown(num):
    print("Starting")
    while num>0:
        yield num
        num -=1
cd = countdown(4)
print(next(cd))
print(next(cd))
print(next(cd))
print(next(cd))
```



```
Starting
4
3
2
1
```

- Example:

```
def firstn(n):
    nums = []
    num = 0
    while num < n:
        nums.append(num)
        num +=1
    return nums

mylist = firstn (10)## will take a lot of memory
def firstn_generator(n):
    num = 0
    while num<n:
        yield num
        num +=1

print (sum(firstn_generator(10)))## size for this memory is very less comparitively
print(sum(firstn(10)))
```

- Example for fibonaci:

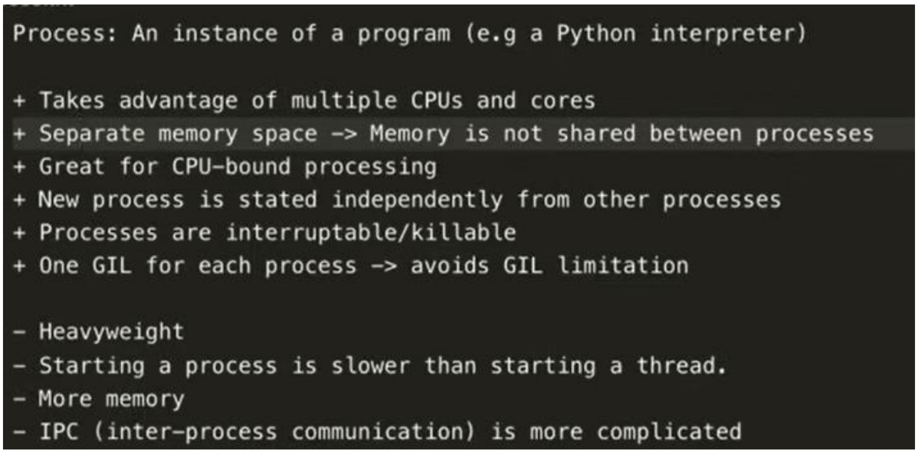
```
def fibonacci(limit):
    a, b = 0,1
    while a<limit:
        yield a
        a, b = b, b+a
fib = fibonacci(30)
```

```
for I in fib:
        print(i)
```

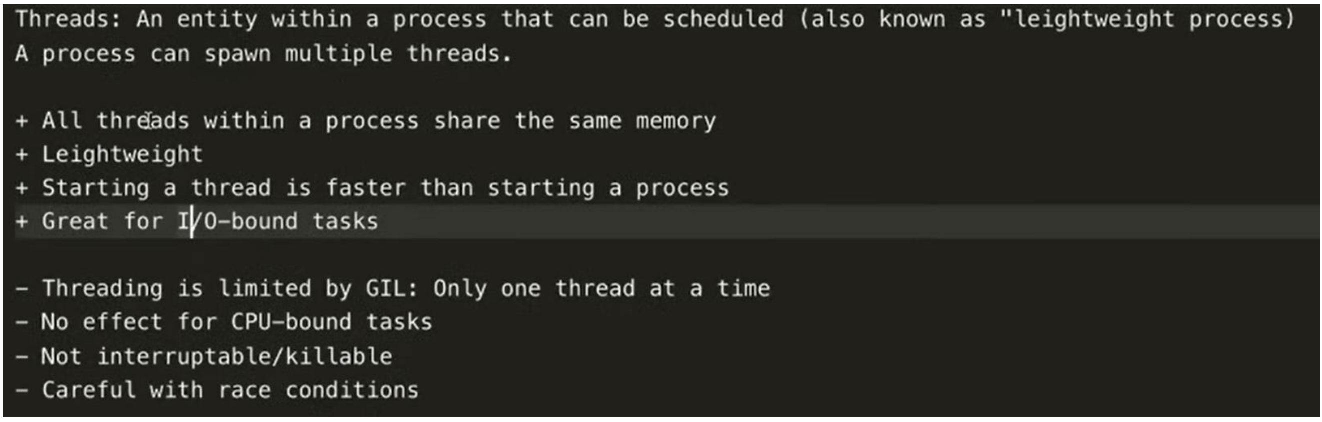
- Example:

```
mygenerator = (I for I in range(10) if i%2 ==0) ## saves a lot of memory
mylist = [I for I in range(10) if i%2 ==0 ]
```

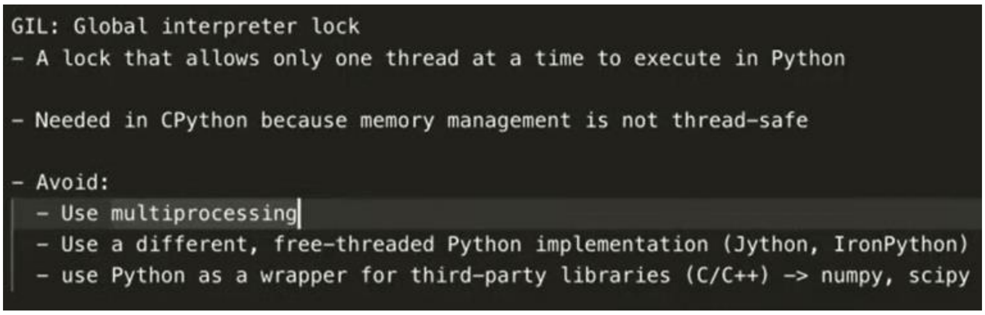
- Process and Threads:



```
Process: An instance of a program (e.g a Python interpreter)

+ Takes advantage of multiple CPUs and cores
+ Separate memory space -> Memory is not shared between processes
+ Great for CPU-bound processing
+ New process is stated independently from other processes
+ Processes are interruptable/killable
+ One GIL for each process -> avoids GIL limitation

- Heavyweight
- Starting a process is slower than starting a thread.
- More memory
- IPC (inter-process communication) is more complicated
```



```
Threads: An entity within a process that can be scheduled (also known as "leightweight process)
A process can spawn multiple threads.

+ All threads within a process share the same memory
+ Leightweight
+ Starting a thread is faster than starting a process
+ Great for I/O-bound tasks

- Threading is limited by GIL: Only one thread at a time
- No effect for CPU-bound tasks
- Not interruptable/killable
- Careful with race conditions
```



```
GIL: Global interpreter lock
- A lock that allows only one thread at a time to execute in Python

- Needed in CPython because memory management is not thread-safe

- Avoid:
  - Use multiprocessing
  - Use a different, free-threaded Python implementation (Jython, IronPython)
  - use Python as a wrapper for third-party libraries (C/C++) -> numpy, scipy
```

- Multi Processing:

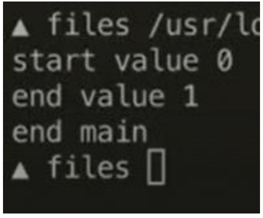
```
from multiprocessing import Process
import os
import time
def square_numbers():
        for I in range(100):
                i*i
                time.sleep(0.1)
processes = []
num_processes = os.cpu_count()
for I in range (num_processes):
        p = Process(target = square_numbers)
        processes.append(p)
for p in processes:
        p.start()

for p in processes:
        p.join()

print('End Main')
```

- Multi Threading:

```
from threading import Thread
import os
import time
def square_numbers():
    for I in range(100):
        i*i
        time.sleep(0.1)
threads = []
num_threads = 10
for I in range (num_threads):
    t = Thread(target = square_numbers)
    threads.append(p)
for t in threads:
    t.start()

for t in threads:
    t.join()

print('End Main')
```

- Threading

```
from threading import thread
import time

database_value = 0
def increase():
    global database_value
    local_copy = database_value

    #processing
    local_copy +=1
    time.sleep(0.1)# here the second thread is invoked when 1st thread waits
    database_value = local_copy

if __name__ == '__main__':

    print('start value:', database_value)
    thread1 = Thread(target = increase)
    thread2 = Thread(target = increase)

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

#Output:
```



```
▲ files /usr/l
start value 0
end value 1
end main
▲ files
```

and not 2 as there are 2 threads operating in race condition trying to alter the same variable

- Corrected Code:

```
from threading import thread
import time

database_value = 0
def increase(lock):
    global database_value
```

lock.acquire() #used as context manager
local_copy = database_value
 	
#processing
	with lock:
	  local_copy = database_value
	
	  #processing

local_copy +=1
time.sleep(0.1)
database_value = local_copy
lock.release()
	local_copy +=1
	time.sleep(0.1)
	database_value = local_copy

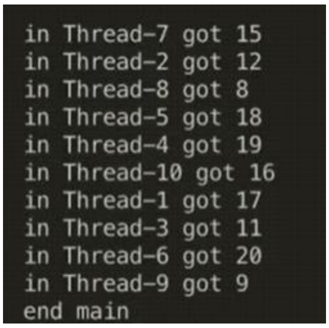
```
if __name__ == '__main__':

        lock = Lock()
        print('start value:', database_value)
        thread1 = Thread(target = increase, args = (lock,))
        thread2 = Thread(target = increase, args = (lock,))

        thread1.start()
        thread2.start()

        thread1.join()
        thread2.join()
```

- Queues: used for multi threading and processing applications

```
from threading import thread
from queue import Queue
import time

if __name__ == '__main__':
        q = Queue()
        q.put(1)
        q.put(2)
        q.put(3)

        #3 2 1 ---->
        first = q.get()
        print(first)  #1

        q.task_done() #mark the end of all tasks done with queue

        q.join()   #waits for all queue to get updated properly

        print('end main')
```

- Example:

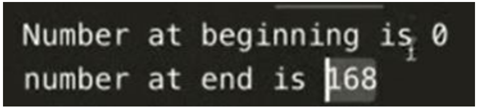
```
from threading import thread
from queue import Queue, Lock
import time

def worker(q , lock):
        while True:
                value = q.get()
                ##processing..
                with lock:
                        print(f'in {current_thread().name} got {value}')
                q.task_done()

if __name__ == '__main__':
        q = Queue()
        lock = Lock()
        num_threads = 10
        for I in range(num_threads):
                thread = Thread(target =worker, args = (q,lock) )
                thread.daemon = True
                thread.start()

        for I in range (1,21):
                q.put(i)
        q.join()

        print('end main')
```



```
in Thread-7 got 15
in Thread-2 got 12
in Thread-8 got 8
in Thread-5 got 18
in Thread-4 got 19
in Thread-10 got 16
in Thread-1 got 17
in Thread-3 got 11
in Thread-6 got 20
in Thread-9 got 9
end main
```

- Multi Processing:

```
from multiprocessing import Process, Value, Array
import os
import time

def add_100(number):
    for I in range(100):
        time.sleep(0.01)
        number.value+=1

if __name__ == '__main__':
    shared_number = Value('I' , 0)
    print('Number at beginning is ', shared_number.value)

    p1 = Process(target = add_100, args = (shared_number,))
    p2 = Process(target = add_100, args = (shared_number,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()

    print('number at end is' , shared_number.value)
```

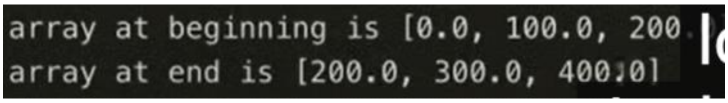


```
Number at beginning is 0
number at end is 168
```

# this happens as there is race condition (2 processes try to read and write into the object at same time). Use lock module to prevent this from happening

- Corrected code:

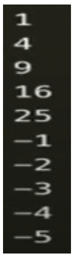
```
from multiprocessing import Process, Value, Array, Lock
import time

def add_100(numbers, lock):
    for I in range(100):
        time.sleep(0.01)
        for I in range(len(numbers)):
            with lock:
                numbers[i] +=1

if __name__ == '__main__':
    lock = Lock()
    shared_array = Array('d' , [0.0, 100.0, 200.0])


    print('Array at beginning is ', shared_array[:])

    p1 = Process(target = add_100, args = (shared_array,lock))
    p2 = Process(target = add_100, args = (shared_array,lock))

    p1.start()
    p2.start()

    p1.join()
    p2.join()

    print('Array at end is ', shared_array[:])
```



```
array at beginning is [0.0, 100.0, 200
array at end is [200.0, 300.0, 400.0]
```

- Example using queue:

```
from multiprocessing import Process, Value, Array, Lock
from multiprocessing import Queue
import time

def square(numbers, queue):
    for I in numbers:
        queue.put(i*i)
def make_negative(numbers, queue):
    for I in numbers:
        queue.put(-1*i)
if __name__ == '__main__':
    numbers = range(1,6)
    q = Queue()
    p1 = Process(target = square, args = (numbers, q))
    p2 = Process(target = make_negative, args = (numbers,q))
p1.start()
p2.start()

p1.join()
p2.join()

while not q.empty():
    print(q.get())
```



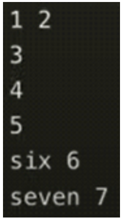
```
1
4
9
16
25
-1
-2
-3
-4
-5
```

- Process pool: break into smaller chunks for multi processing

```
from multiprocessing import Pool


def cube (number):
    return number*number*number

if __name__ == '__main__':
    numbers = range(10)
    pool = Pool()
    #map , apply, join , close
    result = pool.map(cube, numbers)
    # with one argument: pool.apply(cube, numbers[0])
    pool.close()
    pool.join()

    print(result)
```

○ Function Arguments:

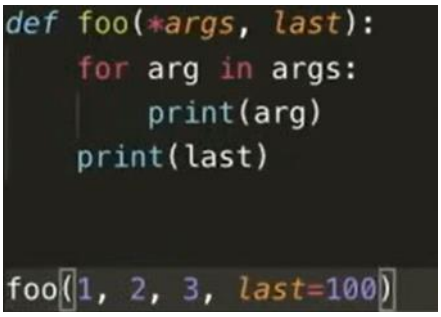
```
def foo(a,b,c):
    print(a, b, c)

foo(c =1, a = 2, b= 3)#key word arguments
foo(1, b=2, c=3) #will work
foo (1, b=2, 3) #Error
foo(1, b=2, a = 3) #Error
def foo(a,b,c,d =4) : # d is default argument foo(1,2,3) will assume a default value for 'd'.
def foo (a,b = 2,c,d =4) # Error

def foo(a, b, *args, **kwargs):
    print(a,b )
    for arg in args:
        print(arg)
        for key in kwargs:
                print(key, kwargs[key])

foo(1,2, 3, 4 , 5, six=6, seven = 7) #1,2 are positional arguments
```



```
1 2
3
4
5
six 6
seven 7
```

➢ def foo(a, b , * , c, d):   # '*' forces the value to be key word argument

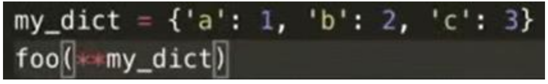
```
        print(a, b, c, d)
foo(1,2,c =3, d=4)
```



```
def foo(*args, last):
    for arg in args:
        print(arg)
    print(last)


foo(1, 2, 3, last=100)
```

○ Unpacking Arguments:

```
def foo(a, b, c):
        print(a, b, c)
my list = [0, 1, 2]# can also be a tuple .(1,2,3,4) will not work
foo(*my_list)
```



```
my_dict = {'a': 1, 'b': 2, 'c': 3}
foo(**my_dict)
```

## the above one also works(length and keys should match)

○ Local and Global variables:

```
def foo():
        global number
        x = number
        number = 3 # will throw an error if global variable is not mentioned
        print('number inside funtion:', x)


number = 0
foo()
print( number) # will get updated to 3
```

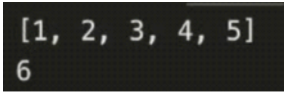
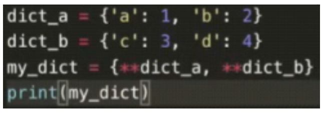
○ Parameter Parsing:

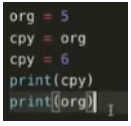
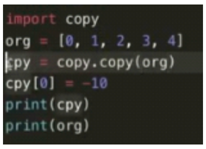
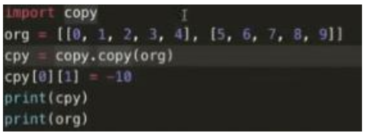
➢ Call by object and Call by object reference(immutable values inside mutable can be reassigned inside an object)

```
def foo(x):
        x = 4
var = 10
foo(var)
print(var)
# cant be changed
def foo(x_list):
        x.append(4)
        x.append[0] = 0
my_list = [1, 2, 3]
foo(my_list)
print(my_list)
## immutable data type is changed within a mutable data type
```

➢ ## rebinding will not work

```
def foo(x_list):
        x_list = [0, 1,3] # this will not work
        #x_list = x_list + [200, 300] will not work
        # x_list += [200, 300] will work
my_list = [1, 2, 3]
foo(my_list)
print(my_list)
```

- Astrik Operation:
  - ➢ result = 5*7
    print(result)
  - ➢ result = 5**7(power operation)
  - ➢ Create list , tuple or string with repeated elements:
    ```
    zero = [0, 1] * 5 # can be tuple or string
    ```
  - ➢ def foo(a, b , *args, **kwargs):
    ```
            print(a)
            for arg in args:
                    print(arg)
            for key in kwargs:
                    print (key, kwargs[key])
    foo(1,2,3,4,5,a=1, seven = 7)
    # all parameters after * will be key word arguments (a,b,* , c )
    ```
  - ➢ Unpacking elements:
    ```
    def foo(a,b,c):
            print(a, b, c)
    my_list = [0, 1, 2]  # will work for tuple
    foo(*my_list)
    my_dict = {'a' : 1, 'b':2 , 'c': 3}
    foo(**my_dict) # key and number of keys should match
    ```
  - ➢ Unpacking Containers:
    ```
    numbers = [1,2,3,4,5,6]
    *beginning, last = numbers
    print(beginning)
    print(last)
    ```
    
    ```
    [1, 2, 3, 4, 5]
    6
    ```
    #beginning will be always a list even if numbers was tuple
  - ➢ Merging(Unpacking):
    - my_tuple = (1,2,3)
      my_set = { 4, 5, 6 }
      my_list = [*my_tuple, *my_set]
    - 
      ```
      dict_a = {'a': 1, 'b': 2}
      dict_b = {'c': 3, 'd': 4}
      my_dict = {**dict_a, **dict_b}
      print(my_dict)
      ```

- Shallow vs Deep Copying:
  - ➢ 
    ```
    org = 5
    cpy = org
    cpy = 6
    print(cpy)
    print(org)
    ```
    Different values for cpy and org
    Mutable variables will get updated
  - ➢ 
    ```
    import copy
    org = [0, 1, 2, 3, 4]
    cpy = copy.copy(org)
    cpy[0] = -10
    print(cpy)
    print(org)
    ```
    This will create a duplicate list
    or list() function or org[:]
  - ➢ 
    ```
    import copy
    org = [[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
    cpy = copy.copy(org)
    cpy[0][1] = -10
    print(cpy)
    print(org)
    ```
    Will edit the first list as copy function is just one level deep
    To avoid this we use : copy.deepcopy(org) has to be used
    Also use custom object:

➢
```
import copy
class Person:
    def __init__(self , name, age):
        self.name = name
        self.age = age

class Company:
    def __init__(self, boss, employee):
        self.boss = boss
        self.employee = employee

p1 = Person('Alex', 27)
p2 = copy.copy(p1)
company  = Company(p1, p2)
company_clone = copy.deepcopy(company)
#otherwise age wont change as copy.copy is shallow copying
company_clone.boss.age = 56
print(company_clone.boss.age)
print(company.boss.age)
p2.age = 28
print(p2.age)
print(p1.age)
```

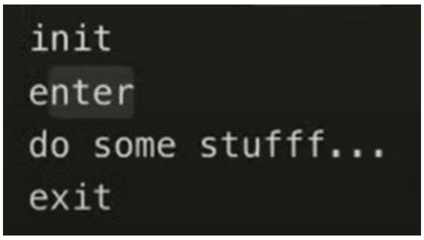
- Context Manager:(resource management)

```
with open ('notes.txt', 'w' ) as file:
    file.write('some todo.....')

file = open('notes.txt', 'w')
try:
    file.write('some to do')
finally:
    file.close()
```

➢
```
from threading import Lock
lock = Lock()
lock.acquire()
lock.release()
with lock:
    #....
```

➢ Context Manager for class:

```
class ManagedFile:
    def __init__(self, filename):
        print('__init__')
        self.filename = filename

    def __enter__(self):
        print('enter')
        self.file = open(self.filename, 'w')
        return self.file

    def __exit__(self, exc_type, exc_value, exc_traceback):
        if self.file:
            self.file.close()
        #add some other part to handle error if in case occured
        print('exit')

with ManagedFile('notes.txt') as file:
    print('do some stuff…')
    file.write('some to doo…')
```



```
init
enter
do some stufff...
exit
```

init : ManagedFile calls this class
enter: with calls this method

exit : method gets called when with loop is exited

- With context manager module:

```
import contextlib import contextmanager

@contextmanager
def open_manage_file (filename):
    f = open(filename, 'w')
    try :
        yeild f
    except:
        f.close()

with open_manage_file('notes.txt') as file:
    file.write('do something ….')
```