* **Putem declara variabile fara a le atribui ceva prin global**

global a

**Input Output**

* print(‘ ‘) sau print(“ “)
* print(‘’’

text

pe randuri

‘’’)

* print(“””

text pe randuri

v2

“””)

* nu se pune tipul variabilei. La declarare, trebuie neaparat initializata
* a = input() – input string
* a = input(“text de afisaj cand se introduce ceva”)
* a = int(input()) transforma stringul din input in int
* print(“text”+string) – putem concantena stringuri, dar nu putem string cu int sau alte tipuri de date diferite
* print(”text”\*3) – afiseaza textul de 3 ori

**Assert Keyword**

* Assert este folosit pentru a decide, pe baza unei conditii, daca programul trebuie continuat sau oprit.
* Are forma: assert conditie, „mesaj de eroare”
* Daca conditia este True, programul continua mai departe executia, in caz contrar, daca e False, programul se opreste si afiseaza o AssertionError
* Poate fi utilizat si un bloc try except pentru a prinde o AssertionError

**Global Keyword**

* Cuvantul cheie global este folosit pentru a crea o variabila globala intr-o functie, metoda sau chiar in \_\_main\_\_ namespace sau
* E folosit daca dorim sa precizam intr-o functie ca vrem sa modificam anume o variabila gobala ci nu una locala cu acelasi nume din functie, asa incat sa evitam conflictul de variabile. Daca de exemplu variabila A este globala si in functie scriem A = A + 2, vom avea eroare, caci functia va incerca sa creeze o variabila A. Daca dorim sa ne referim la A global, trebuie sa scriem **global A** si apoi **A = A + 2**
* Nu putem crea o variabila cu cuvantul cheie global si sa-i si atribuim o valoare in acelasi timp!

**Operatii**

**\*\* -** putere

**/ -** impartirea returneaza mereu un float

**// -** returneaza un numar intreg

str() – trasnforma in string

**Control Flow**

* True/False – cu litera mare!
* Putem utiliza < > == si pentru a compara stringuri. Se compara primul element al primului cuvant cu primul al celuilalt.
* if(conditie): sau if conditie: else(conditie): else conditie:
* elif pentru elseif
* if conditie :

print() – se lasa loc liber neaparat!

* while i < 5: - aceeasi structura
* utilizam in loc de && || ! and or not

**Liste(vectori)**

* a = [] – nu se pun {}!!!
* a = [[1,2,3],[’c’,’d’],[”nr”,”nr2”]] – se accepta diferite tipuri de date in tablou
* a = [1,2,’c’,”d”]
* print(string[pozitie]]
* a~~[5]~~ = [1,2,3,4,5] – nu se pune dupa a []!!!
* Num = [1,2,3] print(Num+[4,5,6]) putem concantena si vectorii. Ca sa fie aratate toate elementele unui vector de variabile primitive, e de ajuns doar sa scriem vectorul fara for
* A = [1,2,3] print(A\*3) – afiseaza tabloul de 3 ori ca un tot intreg
* A = [”aa”,”bbb”,”cccc”] print(”aa” in A) – returneaza True sau False daca se gaseste acel element in tablou. Putem sa il utilizam si pentru a gasi un cuvant in string
* words = [”test1”,”test2”,”test3”]

for word in words:

print(word) - for each loop. E valabil si pentru string.

* a = range(nr) – creaza un iterator ce are valori de la 0 pana la nr-1.
* Ca sa afisam un iterator list cu print(), trebuie sa il convertim in list, asdica se face a = list(range(nr))
* range(final) sau range(inceput,final) sau range(inceput,final,interval intre ele). Un numar negativ la ordine inseamna ordinea din partea dreapta. Adica -1 este ultimul element, -2 penultimul etc.
* len(lista) – afiseaza nr de elemente
* print(lista[1:12:3)] – de la 1 la 12 din 3 in 3. Daca nu se pune nici un numar la primul, inseamna ca e incept si la ultimul e final, adica print(lista[::2]) de la inceput la final din 2 in 2
* a,b,c = [1,2,3] – le da valori variabilelor in ordinea lor de pozitionare
* .pop() – sterge ultimul element
* .pop(pozitie) – sterge elementul din pozitia data
* .remove(element) – sterge prima aparitie a elemetului
* .insert(pozitie,element)
* .append(element) – adauga un element la final
* .extend([element1,element2,…,]) – adauga mai multe elemente la final. Elementele trebuie puse intre []
* .clear() – sterge toate elementele
* .index(element) – returneaza indexul primului element gasit
* .count(element) – Numara de cate ori gaseste elemental in lista
* Print(‘hello’[0]) – asa ceva nu merge si in java

**IterTools**

Produce diferiti iteratori cu care putem lucra. Metodele din acest modul sunt foarte rapide si economisesc memorie.

* count(start,step) – creaza un iterator ce incepe de la start si merge din step in step
* cycle(obiect de iterat) – afiseaza fiecare element din obiectul de afisat si cand termina o face iar si iar
* repeat(valoare,nr de ori) – repeta o valoare de un nr infinit de ori sau de nr ori daca al doilea argument e prezent
* product(lista1,lista2,..,repeat = n) – creaza perechi a cate n elemente combinand toate elementele rand pe rand din fiecare lista sau din una daca doar una e
* permutation(lista,nr de elemente) – daca nu e oferit un nr de elemente, se ia automat lungimea listei si creaza permutari. Daca e oferit un nr de elemente,se fac aranjari cate nr de elemente
* combinations(lista,nr) – combinari fara repetare cate nr
* combination\_with\_replacement(lista,nr) – combinari cu repetare
* accumulate(lista,functie) – ia primul element si face operatia data din functie asupra la al doilea, apoi rezultatul face operatia asupra la al 3 si tot asa si returneaza fiecare operatie. Daca nu se ofera vreo func, implicit e adunarea
* chain(lista1,lista2,lista3) – creaza o lista comuna din mai multe liste
* chain.from\_iterable(lista din mai multe liste) – la fel ca chain, doar ca ia o lista din mai multe liste ca argument
* compress(lista simpla,lista de valori boolene) – afiseaza rezultatele in dependenta de valorile boolene din lista a 2
* dropwhile(func,lista) – incepe sa afiseze elemente din lista abea cand func returneaza pentru prima data un False
* takewhile(func,lista) – afiseaza valori pana la aparitia primului False
* filterfalse(func,lista) – afiseaza doar valorile pentru care func returneaza false
* islice(lista,inceput,final,pasi) – printeaza de la inceput pana la final din pasi in pasi
* starmap(func,lista de tuples) – afiseaza cate un element din fiecare tuple din lista in dependenta de ce returneaza func
* tee(lista,nr) – creaza nr de iteratori dupa lista

**Functii**

* print este o functie
* print(‘a’,’b’,’c’,….) – le va afisa pe toate automat cu spatiu
* print(var,sep=’’,end=’’) – sep si end se pun doar o singura data. Sep se pune odata si se refera la toate variabilele si obiectele din print
* len(lista) – arata nr de elemente din lista sau din string
* lista.append(nr) – adauga un element la finalu listei
* lista.insert(pozitie,item) – adauga un item la pozitia data. Elementul vechi va fi impins inainte
* lista.index(obiect) – returneaza pozitia primului obiect gasit in lista
* max(lista) min(lista) – returneaza minimul si maximul din lista
* lista.count(obiect) – Numara de cate ori se gaseste obiectul in lista
* lista.remove(obiect) – sterge un obiect din lista
* lista.reverse() – reverseaza lista
* format – adauga elemente in string la pozitiile indicate

a = [1,2,3,4]

x = "Numere: {0} {1} {2}".format(a[0],a[1],a[2])

pot fi si mai multe de {0}, si deci primul argument din functia format va aparea peste tot unde e {0} .Nu trebuie sa fie mai putine argumente ca pozitii in {}

Putem plasa si variabile

a = [10,20,30,40]

x = 'Numere = {x} {y}'.format(x=5,y=0)

print(x)

* x = ‘, ‘.join(‘element1’,’element2’,’element3’) – creaza un string din mai multe substringuri, fiecare substring din noul string fiind delimitat de o , sau de elemental dintre ‘’
* string.split(‘ ‘) – separa un string in substringuri la fiecare aparitie a semnului din ‘ ‘
* string.replace(‘vechi’,’nou’)
* string.lower() si string.upper() – inocuiesc literele mari cu mici si invers
* string.find(cuvant,start,end) – cauta un cuvant. Daca nu il gaseste, returneaza -1
* **def func(parametri):**
* putem returna sim ai multe valori sub forma de lista

return [x, y, ‘5’]

* id(obj) – returneaza id
* type(obj) – arata carui clasa corespunde obj
* any – returneaza True daca cel putin un element dintr-o lista de valori booliene este True
* all – returneaza True daca toate elementele dintr-o lista sunt True. In caz contrar, returneaza False.

**Comentarii**

* # comentariu pe o linie
* ”””

Un test - comentariu pe mai multe linii.

”””

**Dictionare**

Functioneaza pe baza de keye si valoare. La dictionare se folosesc {} in loc de [].

Dictionarele pot avea drept keye doar obiecte imutabile, adica care nu pot fi modificate. Doar listele si dictionarele sunt mutabile.

* A = {1 : “test1”,

2 : “test2” - se pune doar {} si intre key si value se pune :

}

* In si not in – arata daca o cheie se gaseste sau nu in dictionar

print(1 in a)

print(5 not in a)

* Dic[keyenoua] = valoare noua – adaugam o noua keye cu valoare
* len(a) – arata numarul de elemente
* .get(key,”mesaj in caz ca nu o gaseste”) – arata valoarea specifica keyi. Daca nu o gaseste, returneaza NONE sau valoarea indicate la apelarea functiei
* Print(dic[key]) in loc de get
* Dict = {} – dictionar gol
* Dict = dict({}) – functie de creare
* Dict = {‘1’ : ‘val1’, 4:{5:’val2’,6:’val3’}} – nested dictionar
* Print(Dict) – printam dictionarul
* Print(Dict[4][5]] – printam nested dictionar
* Dic[1][‘val10’] = – modifica valoarea pentru key 1(nested)
* Dic[5][‘test’] = – adauga un nou key cu value(nested)
* **clear() –**Remove all the elements from the dictionary
* **copy() –**Returns a copy of the dictionary
* **get() –**Returns the value of specified key
* **items() –**Returns a list containing a tuple for each key and value pair
* **keys() –**Returns a list containing dictionary’s keys
* **pop() –** Remove the element with specified key
* **popitem() –**Removes the last inserted key-value pair
* **update() –**Updates dictionary with specified key-value pairs(aduga mai multe perechi odata)
* **values() –** Returns a list of all the values of dictionary
* **key –** pot fi orice tip de date

**Tuples**

Sunt ca listele, dar sunt de nemodificat.

* Tuples = (1,2,3) sau Tuples = 1,2,3 – folosim () sau nu le folosim
* Pot fi nested
* Print(tuple[0:4:2]) - Putem folosi iteratorii
* tuple(list) – converteste o lista in tuple
* tuple = 1,2,3 a,b,c = tuple – atribuie elementele variabelor in ordinea lor
* x = {1,2,3,4,5,6,7,8}

a,b,\*c,d = x; - c va lua toate valorile incepand cu pozitia sa, asa incat se va asigura ca si variabilele care are urmeaza dupa ea vor avea variabile, dar cu pozitiile finale

* tuple(obj) – creaza un tuple

**Sets**

Nu pot contine valori duble si nu pot fi accesate cu []. Desi sunt create in ordine, elementele sunt neordonate. Se creaza cu {}, ca si dictionarele. Elementele duble sunt sterse deodata

* Mai pot fi si frozenSet – care nu pot fi modificate dupa crearea lor, adica nu mai putem adauga sau sterge elemente.
* Putem converti o lista in sets prin set(lista) sau in frozenset frozenset(lista)
* A = {1,2,3,4,5} – creare
* Set3 = set1.union(set2) sau Set3 = set1 | set2– reuniunea a 2 seturi
* Set3 = set1.intersection(set2) sau Set3 = set1 & set2– intersectia a 2 seturi
* Set3 = set1.difference(set2) sau Set3 = set1 - set2– arata ce se gaseste in set1 si nu e in set2
* Set3 = set1^set2 – afiseaza elementele ce se gasesc in set1 si nu in set2 si care se gasesc in set2 si nu in set1
* Set.clear() – sterge setul
* Print(5 in set) – verifica daca 5 se gaseste in set
* Print(set1>set2) sau print(set1.issuperset(set2))– set1 este superset, adica tot ce se afla in set2 se gaseste si in set1, dar nu tot ce e in set1 este in set2
* .add() – adauga un element
* .remove() – sterge un element

**Functii(2)**

* O functie poate avea ca parametru si o alta functie
* Parametrii unei functii, daca sunt obiecte, pot modifica obiectele din parametrii actuali.
* O functie poate fi

**Pura** – nu modifica valoarea unor variabile/obiecte din afara ei. F

**Impura** - functia poate modifica valorile unor obiecte din afara ei. De ex:

def func(a):

a.reverse() #a e string

totusi, daca vom incerca sa facem a = a + “hello”, functia va lua a ca un parametru local, si nu va modifica niciun obiect. Pentru a modifica valoarea unei variabile din afara functiei, punem in fata ei **global** si apoi mai jos efectuam operatiile

def func(b):

global a

a = b + 20

parametrul formal nu poate avea acelasi nume ca var globala!Compilatorul va crede ca facem var din parametri formali globala

Nepura – nu modifica variabile/obiecte din afara

**List Comprehension**

Este o lista care se creaza pe baza modificarii unei alte liste.

* *newList****=******[****expression(element)****for****element****in****oldList****if****condition****]***
* lista = [a\*10 for a in range(1,11) if(a%2==0)]

1. Se face iteratia din for
2. Se verifica conditia
3. Se face operatia a\*10 si se adauga in noua lista

Orice list comprehension e format din:

Expresie de iesire – a\*10

Secventa de intrare – range(1,11)

Variabila – a

Predicat – if

* Nu se pune si else
* Putem avea si nested list comprehension

lista = [[x for x in range(0,4)] for a in range(0,5)]

variabila a nu va fi nicaieri afisata, dar de fiecare data cand are loc o iteratie in al doilea for, se va face cate o iteratie complete a primului for, iata de ce lista va fi o lista de liste

* lista = [[10,20,30],[40,50,60],[70,80,90]]

lista2 = [[i[j] for i in lista] for j in range(len(lista))]

* daca avem o functie lambda,

se vor afisa tablouri create din fiecare element cu ordinea j din fiecare sublista din lista. Var i si j raman pe loc, nu dispar nicaieri. Totusi, elemetele din for nu sunt trimise automat catre lambda!de aceea e important sa egalam parametrii cu elementele din for

tabel = [lambda x=y : x\*x for y in range(1,5)]

for a in tabel:

print(a())

* Acest program afiseaza patratele intervalului [1,5).

Functia lambda este acea variabila care se adauga la lista, doar ca dupa ce se face o executie in for, variabila y nu se trimite auomat catre lambda. Aici tabelul va fi creat din elemente de tip functii lambda, deoarece el nu va efectua niciun calcul atat timp cat functia lambda nu e apelata. Deci, in [] lambda e doar pusa ca element care se va trimite, deci nu poate face niciun calcul automat, caci nu asa functioneaza list comprehension!

Deci, daca table e format din functii lambda, nu vom putea afisa valoarea decat cu o variabila ce ia ca valoare si ea o functie lambda in for.Asadar, a() din print inseamna ca se va accesa functia ce are ca parametru x=y, dar y e deja un numar aici, caci cand s-a facut iteratia, de exemplu nr 1, y a fost 1, si deci functia a fost trimisa ca

lambda x=1 : x\*x adica cu un parametru automat

tabel = [lambda x=x : x\*x for x in range(1,5)]

for a in tabel:

print(a())

acest program e identic cu cel de sus. x din lambda se va egala cu x din for! x din lambda e parametru formal, dar acel x dupa = e cel din for!

**Functii Anonymus(lambda)**

Sunt functii fara nume atribuite direct unei variabile sau obiect sau nimanui. In loc de def, pentru definirea lor se foloseste cuvantul lambda

* Var/ob = lambda arguments: expression
* Poate avea doar un singur rand cu declaratii care este executat si returnat
* a = lambda x : print(x)

print(a(10)) – afiseaza variabila ce are valoarea data de functia lamba cu parametrul 10

print(a) – afiseaza doar locatia functiei lamba in memorie

* putem apela functia lambda si fara ca ea sa apartina direct unei variabile:

a = ‘Hello’

(lambda x : print(x))(a)

Astfel, ei i se trimite ca parametru a

* var = lamba x : 2\*x if (x>10) else (x\*\*2) – else trebuie sa fie neaparat prezent daca este si if. Ambele trebuie neapart sa returneze ceva

**Import**

* from **modul** import **func/obiect1.2,3,4…**

from time import sleep

* sleep(time) – pauza la program
* asctime() returneaza timpul curent

**Map**

Returneaza un iterator dupa ce o functie data ca parametru a modificat toate elementele. Obiect = map(func,lista)

* lista = [1,2,3,4]

def multiple(x):

return x\*x

obj = map(multiple,lista)

print(list(obj))

afiseaza patratele lui lista

\*args si \*\*kwargs

* \*args – poate stoca un numar oricat de mare de valori sau de variabile. Il folosim daca nu stim cati parametri vom trece functiei

def func(\*args):

for i in args:

print(i)

func(‘hi’,’how’,’are’,’you’) sau func(a,b,c,d) – a,b,c,d – variabile

* In caz ca se declara si alte variabile impreuna cu \*args, ele vor lua primele valori si apoi \*args ce ramane
* \*\*kwargs – un dictionar ce ia variabile(key) cu valorile lor

def func(\*\*kwargs):

for i,k in kwargs.items():

print(i,k)

func(a=‘hi’,b=’how’,c=’are’,d=’you’)

* Putem sa apelam si o functie cu ele:

def func(a,b,c):

print(a,b,c)

a = ('hello','my','friend')

func(\*a)

b = {'a' : 'hello','b' : 'my', 'c' : 'friend'}

func(\*\*b)

insa,numarul de variabile stocate de \*a si \*\*b nu trebuie a depaseasca nr de parametron formali din functie si nici sa fie mai mic!

Pentru b, keyle trebuie sa fie identice cu numele parametrilor din functie! Atunci cand trimitem argumente prin func(\*\*b) – compilatorul va cauta variabilele din key si le va atribui acele valori

* \*args si \*kwargs mereu stau la final In lista de parametri formali

**Generatori**

Sunt functii care folosesc yield in loc de return. Spre deosebire de return, pot exista mai multe yield si odata ce unul a fost executat, se continua si urmatorul, daca functia e apelata din nou. Variabilele locale ale functiei nu sunt sterse. Aceste functii returneaza un obiect de tip generator ce poate fi iterat.De fiecare data cand se itereaza, se continua executia cate odata a functiei pentru a primi un nou element rand pe rand. Yield opreste executia functiei si trimite o valoare inapoi catre caller, dar memoreaza starea functiei pentru a permite mai apoi executia ei de unde a ramas. Odata ce se iese din for, functia se reseteaza. Spre deosebire de iteraotir, generatorii aloca memorie pentru fiecare nou element rand pe rand si nu pentru toate odata. O functie generator nu distruge variabilele sale locale nici dupa finisare. Odata ce se termina de executat, functia lasa obiectul ei, dare el continua sa existe tot. Functia genereaza cate un obiect separate pentru fiecare generator, adica a = func() si b = func() vor avea obiecte diferite!

def func():  
 for i in range(10):  
 i = int(input("i = "))  
 yield i  
  
def main():  
 for i in func():  
 print(i)  
main()

* Se utilizeaza bucla for pentru a itera
* x = func() – creaza un obiect generator x, dar nu poate fi afisat
* daca cream un obiect ce va lua obiectul generator al functiei, pentru a putea sa accesam ce returneaza yield, utilizam next(obj)

def func():

yield 1

yield 2

yield 3

x = func()

print(next(x))

print(next(x))

* Generator expression:

squres = (x\*x for x in range(5))

* Un obiect de tip generator poate fi convertit intr-o lista direct

def func(x):

for i in range(x+1):

yield i

a = func(10)

print(list(a))

**Pass**

Nu face nimic, pur si simplu e pus acolo unde trebuie sa fie un cod, dar noi nu vrem sa punem vreunul pentru moment, ci poate mai tarziu

De ex:

if(a>10):

pass

else:

a = 100

for i in range(10):

pass

In python dupa : mereu trebuie sa fie un cod din aliniat, de aceea exista pass

**Decoratori**

Decoratorii ne permit sa modificam o functie utilizand o alta functie. Acest lucru se face cand vrem sa extindem capacitatile unei functii fara a o modifica. Insusi decorator este numele dat acelei functii care va modifica functia ce dorim sa fie modificata indirect prin a o lua ca parametru formal si apoi a o introduce intr-o alta functie din decorator ce o va volosi dar va mai adauga ceva nou pe langa ea. Functiile decorator foloseste o functie noua in ea ce apeleaza functia de dorim sa o modificam, dar insasi functia decorator returneaza anume aceasta functie din ea pentru a o putea folosi in locul functiei ce dorim sa o modificam. Putem chiar sa modificam obiectul functiei pe care dorim sa o modificam.

def decorator(func,text):

def modifica\_functia():

print('-----------------------------------------------------')

func(text)

print('-----------------------------------------------------')

return modifica\_functia

def arata\_text(text):

print(text)

def main():

global arata\_text

arata\_text = decorator(arata\_text,"Salutare tuturor!")

arata\_text()

main()

* Am fi putut pune si variabila = decorator(arata\_text,"Salutare tuturor!") ,insa in acest caz noua variabila s-ar fi legat de blocul functiei din interiorul functiei decorator,adica de modifica\_functia() si nu chiar propriuz zis functia arata\_text
* Astfel, functia arata\_text va pierde legatura cu blocul sau initial si se va lega deja de un nou bloc, si anume de blocul functiei modifica\_functia, care va pastra si el legatura cu fostul obiect al functiei decorator dar si legatura cu fostul obiect al functiei arata\_text de care era legata. Dupa return functia decorator sterge namespaceul ei si pierde legatura cu vechiul obiect, dar oricum vechiul obiect functie de care ea a fost legat, continua sa ramana prin memorie, si deci orice apel nou al functie arata\_text deja va apela un alt bloc.

In loc de a scrie arata\_text = decorator(arata\_text,"Salutare tuturor!"), putem sa o omitem, dar sa scriem

@decorator

def arata\_text(text):

print(text)

Acest @ apeleaza functia decorator si trimite direct functia cu tot cu parametru in functia decorator,astfel incat primul parametru al functiei catre care s-a trimis(decorator) va fi chiar functia trimisa(arata\_text), chiar daca ea nu a fost apelata, si leaga variabila arata\_text de un nou obiect functie in dependenta de return din functia decorator. Ceilalti parametri din arata\_text nu se pot trimite celorlalti parametri din functia decorator, deci e necesar ca functia decorator sa aiba doar un argument, si anume functia ce o va decora.

* @decor1

@decor

def num():

return 10

print(num())

**echivalent cu decor1(decor(num))**

**Classes**

Clasa este o schita sau un model al unui obiect. Obiectul este o instantiere a unei clase sau o copie a acesteia cu diferite valori ale atributelor. Orice variabila dintr-o calasa este un atribut.

Obiectul e caracterizat de:

* Stare – e reprezentata de atributele sale si de proprietatile sale
* Comportament – e reprezentat de metodele sale
* Identitate – e reprezentat de numele sau care il diferentiaza de alte obiecte

Fiecare instanta de clasa(obiect) impartasesc aceeasi clasa, dar au valori diferite in memorie.

Ca si la C++ si Java, o clasa are declarate variabile si metode, insa diferenta e ca metodele nu acceseaza direct membrii clasei. De ex:

class test:  
 a = 100  
 b = "Hello"  
 def func(self):  
 print(a)  
 print(b)

a si b nu vor fi gasite, deoarece functia va incerca sa le caute in namespaceul ei si nu le va gasi. Orice functie are o referinta la namespaceul global, deci metodele din clasa au si ele referinta la namespaceul global, dar nu la cel al clasei. In C++

metodele din clasa au referinta la namespaceul din clasa, nu si in Python.

class test:  
 a = 100  
 b = "Hello"  
 def func(self):  
 global a,b  
 print(a)  
 print(b)

Un asa cod va face sa fie cautate a si b in namespace global, nu al clasei, deci pentru a accesa membrii clasei mereu se utilizeaza clasa.variabila sau obiect.\_\_class\_\_.variabila care nu va face referinta la var lui obiect, ci a cea a clasei.class sa si se lasa!. In C++ si Java utilizarea lui this pentru a accesa membrii din obiect nu e necesar, in Python DA

In C++ si Python, functiile au o referinta la namespace global si ce nu gasesc in namespace lor, vor cauta doar in cel global.

Totusi, in C++, metoda are referinta si la namespace la obiect,deci o metoda va cauta intai o variabila in namespace la ea,dupa in obiect si apoi in cel global!

In C++ si Java clasele nu au propriul namespace, de aceea chiar de au obiecte ca membri, fiecare obiect de tip clasa va avea aceste obiecte membri diferite ca locatii, spre deosebire de Python.

* Orice metoda dintr-o clasa trebuie sa aiba primul membru din parametrii formali self, sau un alt parameter care va fi ca self
* Constructorii se definesc ca o metoda cu denumirea \_\_init\_\_, deci nu ca si in C++ sau java!
* Variabilele se intializeaza la crearea constructorului sau in orice alta metoda, caci nu putem define variabile neinitializate in Python si trebuie cumva sa diferentiem namespace al obiectului de cel al clasei.

class test:  
 def \_\_init\_\_(self,a):  
 self.a = 100  
  
def main():  
 obj = test(5)  
 print(obj.a)  
  
main()

* Variabilele declarate in afara metodelor, adica in clasa, apartin tuturor obiectelor,caci ele sunt membri de clasa, ca variabilele statice(desi nu sunt statice) in C++ si Java, deci le putem accesa direct prin clasa.variabila.Totusi daca vom incerca sa modificam aceasta variabila la un obiect, ea deja se va declara in bloc la obiect si anume acea valaore atribuita nu va mai apartine tuturor obiectelor, ci doar aceului obiect.Cu alte cuvinte, cand se creaza un obiect, in blocul sau se adauga o referinta la blocul clasei, si daca incercam sa accesam prin el un obiect membru al clasei, acel membru se va cauta intai in namespace la obiect si deoarece nu va fi gasit, va utiliza referinta la clasa pentru a-l cauta in namespace la clasa . Putem adauga membri pentru un obiect si direct dintr-o metoda,ca main, prin obiect.variabila = obiect
* Din cauza ca in Python clasele au namespace, cand se creaza un nou obiect, el va avea referinta la blocul clasei, si deci va putea accesa toate metodele si membrii clasei. Deci metodele se gasesc propriu zis nu in obiect, ci in clasa, iata de ce e nevoie de trimis mereu self, caci clasa sa stie in ce obiect sa faca modificarile. In Python obiectele nu copie toti membrii clasei exact ca in C++ sau Java, ci doar au o referinta la clasa, si ce nu gasesc in blocul lor, cauta in blocul clasei.
* class test:  
   a = 'global'  
   def \_\_init\_\_(self,b):  
   self.b = 100  
    
  def main():  
   obj1 = test(5)  
   obj2 = test(100)  
   print(obj1.b)  
   print(obj2.b)  
    
   print(obj1.a)  
   print(obj2.a)  
    
   obj1.a = "Il modificam"  
   print(obj1.a)  
   print(obj2.a)  
   print(test.a)  
    
  main()
* Atributele sunt mereu publice
* Constructorii sunt cei care creaza si returneaza obiectele.
* Cand un obiect e creat, constructorul intai apeleaza metoda \_\_new\_\_ pentru a aloca memorie si abea dupa adauga sau modifica atributele din \_\_init\_\_

**Mostnenirea simpla**

* Python suporta mostenirea multipla
* Pentru a mosteni se tine cont de forma:

class copil(parinte):

* Clasa parinte = superclass
* Pentru a apela constructorul clasei parinte, se foloseste expresia clasaParinte.\_\_init\_\_(self,…)
* Apelarea constructorului parinte nu trebuie sa fie prima instructiune din constructorul clasei copil, spre deosebire de Java. Pote fi oriunde, doar ca membrii din constructorul clasei parinte nu vor exista pana nu va fi apelat constructorul.
* Mostenirea are loc cam ca si la Java si C++, desi are si o desoebire

1. In C++ si Java se creaza un namespace secundar,adica cel al obiectului unde se vor stoca membrii mosteniti.
2. In C++ si Java, se creaza copii ale obiectelor metodelor si variabelor mostenite si apoi membrii mosteniti din namespace secundar se leaga de aceste obiecte.
3. In C++ si Java, in namespace principal deja se adauga membrii clasei copil,se creaza obiecte functie sau obiecte caracteristice lor dupa modelul clasei si se leaga de noile obiecte.

In Python, clasa ce mosteneste are pur si simplu o referinta la clasa parinte, si atat.Ce nu gaseste obiectul de tip copil in propria clasa, va cauta in clasa parinte.In Python constructorul poate crea noi membri in namespace, caci python daca nu gaseste o oarecare varianila in namespace, pur si simplu o adauga si o leaga.

* Python nu are cuvinte cheie ca private,protected sau public. Totusi, exista o conventie prin care membrii ce nu trebuie accesati din afara se incep cu \_.

Mai mult ca atat, \_ are influenta doar in cazul dat:

**from module\_name import \* -** nu se vor mosteni variabilele care incep cu \_

* Membrii care au \_\_ la inceput in denumire sunt setati ca **privati**. Ei nu pot fi accesati inafara clasei si nici nu se mostenesc, dar ii putem accesa printr-un obiect de tip clasa parinte: obiect.\_clasa\_\_variabila
* Membrii care au \_ la inceput sunt protected. Ei pot fi accesati de oriunde, doar ca vor arata un avertisment ca sunt protected, dar e doar un avertisment simplu, nu blocheaza accesarea lor. Totusi, daca sunt folositi de clasa copil, nu apare niciun avertisment, indiferent ca e un obiect de tip parinte sau nu, spre deosebire de C++ care nu permite ca obiectele de tip parinte sa poata accesa membrii privati chiar si in clasa copil
* In Python super() tot e disponibil ca in Java, doar ca apelarea lui se face cu (), spre deosebire de Java. Totusi, super() nu mai e folosit in Python 3, si in locul lui se pune direct numele clasei parinte
* Super() nu cere si argumentul self, caci el returneaza obiectul clasei direct, dar clasa.\_\_init\_\_(self) cere self ca argument
* Cu clasaNume putem adauga membri in clasa parinte din clasa copil ce vor apartine clasei chiar, dar cu super() nu putem.
* In Python, constructorul clasei copil nu va apela automat constructorul default din clasa parinte ca in C++ sau Java!Trebuie facut manual.
* Python nu face o copie a fiecarui membru din clasa parinte ca in C++ sau java, ci doar clasa copil are doar o referinta la clasa parinte.

**Mostenirea multipla**

* In Python ambiguitatea mostenirii claselor exista, dar ea e rezolvata automat.
* Daca avem class1, class2(class1), class3(class1) si class4(class2,class3), atunci Python se va conduce mereu dupa class2. Daca era class4(class3,class2) era sa se conduca dupa class3. Python are un asa sistem de parcurgere a ordinii a claselor mostenite.
* Totusi, daca avem class(class2,class3) si class2 nu suprascrie metodele din class1 dar class3 da, atunci el va executa metoda suprascrisa din class3, dar in rest se va conduce dupa class2.
* Daca toate clasele au o metoda identica si class2 cheama in metoda data metoda din class1 si class3 la fel o face, iar class4 cheama ambele metode, adica din class2 si class3, metoda din class1 s-ar executa de 2 ori. Aceasta dubla executie va fi impiedicata de utilizarea lui super().func() in loc de class1/2/3.func().
* class class1:  
   def func(self):  
   print("class1")  
    
  class class2(class1):  
   def func(self):  
   print("class2")  
   class1.func(self)  
    
  class class3(class1):  
   def func(self):  
   print("class3")  
   class1.func(self)  
    
  class class4(class2,class3):  
   def func(self):  
   print("class4")  
   class2.func(self)  
   class3.func(self)  
    
  def main():  
   obj = class4()  
   obj.func()  
    
  main()

Daca vom utiliza super, nu va mai trebui sa apelam de 2 ori func din class2 si class3.

class class1:  
 def func(self):  
 print("class1")  
  
class class2(class1):  
 def func(self):  
 print("class2")  
 super().func()  
  
class class3(class1):  
 def func(self):  
 print("class3")  
 super().func()  
  
class class4(class2,class3):  
 def func(self):  
 print("class4")  
 super().func()  
  
  
def main():  
 obj = class4()  
 obj.func()  
  
main()

In asa fel, super va asigura ca intai se executa func din class2, apoi din class3,unde va fi executata si cea din class1. Adica va lasa metoda din class1 pe mana lui class3. Class1 iese ca se executa la final.

Python, in caz de mostenire multipla, intai verifica cum se executa instructiunile, si abea apoi le executa. De aceea, rezultatele nu au loc mereu in ordine temporala.

In Python, daca o clasa mosteneste de la mai multe clase, compilatorul va verifica daca nu exista o ambiguitate. Daca exista,utilizarea lui super() pentru a apela o metoda ce exista in ambele clase parinte va verifica intai daca nu cumva in ambele metode se apeleaza aceeasi metoda, dar din clasa bunic. Daca da, atunci intai se va executa metoda data din prima clasa mostenita pana la acel super, apoi se va executa intreaga metoda din clasa a 2 mostenita, cu tot cu acel super ce face referinta la clasa bunic, si apoi se va executa restul din ce a ramas din metoda clasei a 2.

class class1:  
 def func(self):  
 print("class1")  
  
class class2(class1):  
 def func(self):  
 print("class2")  
 super().func()  
 print("Dupa in clasa2")  
  
class class3(class1):  
 def func(self):  
 print("class3")  
 super().func()  
 print("Dupa in clasa3")  
  
class class4(class2,class3):  
 def func(self):  
 print("class4")  
 super().func()  
 super().func()  
  
def main():  
 obj = class4()  
 obj.func()  
  
main()

**Magic Methods sau Dunders(double underscors)**

Metodele Magice sunt acele metode la care Numele lor incepe si se termina cu \_\_

Ele permit supraincarcarea operatorilor, doar ca nu ca in C++. Fiecare operator are o metoda proprie. Se respecta regula din C++ care spune ca obiectul din stanga operatorului se ocupa de gestionarea lui, adica in clasa sa se supraincarca.

**\_\_sub\_\_** for -  
**\_\_mul\_\_** for \*  
**\_\_truediv\_\_** for /  
**\_\_floordiv\_\_** for //  
**\_\_mod\_\_** for %  
**\_\_pow\_\_** for \*\*  
**\_\_and\_\_** for &  
**\_\_xor\_\_** for ^  
**\_\_or\_\_** for |

**\_\_lt\_\_** for <  
**\_\_le\_\_** for <=  
**\_\_eq\_\_** for ==  
**\_\_ne\_\_** for !=  
**\_\_gt\_\_** for >  
**\_\_ge\_\_** for >=

**\_\_len\_\_**for len()  
**\_\_getitem\_\_** for indexing  
**\_\_setitem\_\_** for assigning to indexed values (=)  
**\_\_delitem\_\_** for deleting indexed values  
**\_\_iter\_\_** for iteration over objects (e.g., in for loops)  
**\_\_contains\_\_** for in

\_\_repr\_\_,\_\_str\_\_ for print()

x + y ar fi **x.\_\_add\_\_(y)**

* Daca totusi x nu implementeaza metoda de supraincarcare si x si y sunt diferite, atunci se va apela metoda din y dar cu prefiul r. In C++ nu functioneaza asa. De exemplu, fie

X + Y

Daca x nu implementeaza metoda \_\_add\_\_(self,other), atunci compilatorul va incerca sa apeleze metoda \_\_radd\_\_(self,other) a lui y, daca exista. Acest r vine de la reverse, adica in caz invers, si self va fi Y iar other X. Alti operatori nu folosesc acest r, ca si

**\_\_eq\_\_** for == si **\_\_ne\_\_** for !=

* Putem supradefini si functii built in ca len() etc.
* Daca sunt mai multi operatori, obiectele se trimit in ordinea lor:

A[y] = x

A.\_\_setitem\_\_(y,x)

Class Methods

Sunt metodele care pot fi apelate si de clasa, nu doar de obiecte. In loc de self, se pune deobicei cls. Ele nu iau un obiect ca prim parametru, ci o clasa.

Pentru a o crea, se utilizeaza decoratorul @classmethod sau metoda classmethod, Astfel, se va crea o noua metoda ce va opera doar asupra membrilor din blocul clasei, nu al obiectelor.

1. O putem face prin a trimite metodei classmethod metoda

**Clasa.metoda = classmethod(clasa.metoda)** – astfel se va returna o referinta la o noua metoda de clasa ce va face toate modificarile doar asupra membrilor clasei, nu ai obiectelor.

1. Sau o putem face asa ca in cazul decoratorilor scriind metoda in clasa, dar punand inaintea ei decoratorul @classmethod.

@classmethod

def func(cls,….)

**Static Methods**

Este o metoda care nu modifica nici clasa, nici vreun obiect de al ei, si nici nu are acces la ele. E prezenta doar pentru a face anumite lucruri independente. Ea, deobicei, se foloseste intr-o alta functie.

* Se foloseste mereu decoratorul @staticmethod inaintea ei
* Nu are un prim parametru implicit si nu stie nimic despre clasa sau obiect

**Property**

Property este o clasa care contine in ea mai multe metode ce ofera informatii despre obiect, si care sunt executate prin supraincarcarea operatoilor print(), = si del, si informatiile se ofera prin:

* Functia ce returneaza valorile
* Functia de setare a unor noi valori
* Functia de stergere a unor valori(cand se utilizeaza del obiect.numeatribut)

Rolul ei este de a oferi informatii legate de obiect prin fuctii getter setter si delete cand se folosesc operatorii mentionati. Ea ia ca parametri in constructor metodele specifice si le aduaga in metodele de suprincarcare a operatorilor,ca si decoratorii ce modifica comportamentul, apoi respectiv atributul, ce va fi un membru nou al clasei ,va fi legat de un obiect de tip property ce supraincarca operatorii si utilizeaza metodele noastre pentru a face diferite operatii.

* Se creaza prin crearea unui atribut nou ca membru de clasa ce apeleaza constructorul clasei property cu parametrii ce sunt referinte la metoda de afisare a valorii ce o si returneaza,de modificare a valorii si de stergere si leaga deci atributul de un obiect de tip property ce le contine pe toate 3 si le foloseste la supraincarcarea operatorilor
* La metoda de afisare se poate pune deasupra @property apoi la metoda cu acelasi nume, dar de setare se pune @nume.setter si de stergere cu acelasi nume @nume.deleter.
* Operatiile de setare,stergere si obtinere se vor folosi prin apelarea obiectului atribut de tip property setat
* Doar functia de setare ia un parametru formal pe langa self, restul nu!

class persoana:  
 def \_\_init\_\_(self,nume,prenume,varsta):  
 self.nume = nume  
 self.varsta = varsta  
 self.prenume = prenume  
  
 def getter(self):  
 print("Getter")  
 return ("Nume: "+self.nume+"\nPrenume: "+self.prenume + "\nVarsta: "+str(self.varsta))  
  
 def setter(self,args):  
 print('Setter')  
 self.nume = args.get("nume")  
 self.varsta = args.get("varsta")  
 self.prenume = args.get("prenume")  
  
 def deleter(self):  
 print("Deleter")  
 del self.nume  
 del self.prenume  
 del self.varsta  
  
def main():  
 Eduard = persoana("Mititiuc","Eduard",20)  
 persoana.modifica = property(persoana.getter,persoana.setter,persoana.deleter)  
 print(Eduard.getter())  
 Eduard.modifica = {"nume":"Goncear","prenume":"Alexandru","varsta":"20"}  
 print(Eduard.modifica)  
 del Eduard.modifica  
  
  
main()

Aici am creat 3 metode difertie ca nume, caci noul atribut creat, adica modifica, va fi un obiect de tip property ce va avea toate cele 3 metode in el, doar ca vor fi si supraincarcati operatorii print() = si del

De asemenea, am fi putut crea metoda modifica si direct in clasa, ca membru.

class persoana:  
 def \_\_init\_\_(self,nume,prenume,varsta):  
 self.nume = nume  
 self.varsta = varsta  
 self.prenume = prenume  
  
 @property  
 def newfunc(self):  
 print("Getter")  
 return ("Nume: "+self.nume+"\nPrenume: "+self.prenume + "\nVarsta: "+str(self.varsta))  
  
 @newfunc.setter  
 def newfunc(self,args):  
 print('Setter')  
 self.nume = args.get("nume")  
 self.varsta = args.get("varsta")  
 self.prenume = args.get("prenume")  
  
 @newfunc.deleter  
 def newfunc(self):  
 print("Deleter")  
 del self.nume  
 del self.prenume  
 del self.varsta  
  
def main():  
 Eduard = persoana("Mititiuc","Eduard",20)  
 print(Eduard.newfunc)  
 Eduard.newfunc = {"nume":"Goncear","prenume":"Alexandru","varsta":"20"}  
 print(Eduard.newfunc)  
 del Eduard.newfunc  
   
main()

Acest program e identic cu cel precedent. Metoda newfunc este supraincarcata,dar totusi in Python se ia mereu ultima sa declaratie drept corp. Totusi, Cand se cheama decoratorii, primul @property va lua corpul metodei si o va seta in constructor ca functie getter,si va lega atributul newfunc de un nou obiect de tip property. Se pune @func.setter pentru a apela metoda setter() din obiectul de tip property ce va returna un nou obiect de tip property la newfunc ce va avea deja definite metode getter si setter si supradefiniti 2 operatori deci, iar @newfunc.deleter va apela metoda delete() si va returna un nou obiect cu toate cele 3 metode si cu toti cei 3 operatori supraincarcati. Numele trebuie sa fie identic din cauza ca nu s-ar putea accesa obiecte ale altor metode, asa cum metodele ce sunt supraincarcate au mai multe corpuri. Ideea e ca metoda newfunc are si ea 3 corpuri, dar il poate utiliza doar pe ultimul, si deci primul decorator va modifica primul obiect de care e legat, al doilea al doilea si al treilea il va modifica pe ultimul, cel care si poate fi accesat.

**Exceptii**

**ImportError**: an import fails;  
**IndexError**: a list is indexed with an out-of-range number;  
**NameError**: an unknown variable is used;  
**SyntaxError**: the code can't be parsed properly;  
**TypeError**: a function is called on a value of an inappropriate type;  
**ValueError**: a function is called on a value of the correct type, but with an inappropriate value.

* Orice eroare in Python poate fi de 2 tipuri:

1. Syntax Error
2. Exceptions

* In Python se folosesc try,except si finally
* Ca si in C++ si Java, variabilele membre din blocul try nu pot fi accesate de blocul global si caci dupa executia lui try, sunt sterse. Adica membrii declarati in try pot fi accesati doar in el.
* Pentru a prinde o eroare de orice tip, folosim doar except:
* Ca si in Java si C++, daca in try se gaseste o eroare, nu se executa ceea ce e mai departe de acea eroare.
* Pentru a prinde un anumit tip de exceptie folosim **except typeerror**:
* Ca si in C++ si Java, Python cauta primul bloc except pentru a trata exceptia.
* Ca si in Java si C++, un bloc try nu poate exista singur in Python
* Ca si in Java, un bloc try poate exista doar cu un finally
* In Python,un bloc try poate avea si un else la final de toate blocurile except. Acesta se executa daca blocul try nu intalneste vreo exceptie. Totusi, daca avem si finally, else trebuie pus inaintea lui finally
* In Python, **raise** este echivalentul lui throw din java si C++. Raise poate arunca un anumit tip de exceptie, sau o un obiect al carei clasa mosteneste clasa Exception. De exemplu:

**Raise NameException(„Test Exception”)**

Acest obiect va fi trimis blocului

**except NameException**

si anume parametrului NameException

* Daca dorim sa vedem textul ce este pastrat in NameException, doar scriem raise la final de except bloc:
* try:  
   print("Aruncam o eroare")  
   raise NameError("EROARE aruncata din try")  
  except IndexError:  
   print("Index ERROR")  
  except NameError:  
   print("Name Error")  
   raise

scrisul din obiectul NameError va aparea cu rosu si va fi primul.

* Putem de asemenea afisa textul din obiectul NameError prin modificarea liniei except:

except NameError as i:  
 print(i)

* Spre deosebire de C++, in Python un bloc except poate sa trateze mai multe tipuri de erori

**except (typerror1, typerror2, typerror3)**

except nu se comporta ca o functie. E de ajuns ca un parametru sa aiba valoarea ca ea sa se execute. In java ele se separa prin | in loc de ,. Parantezele sunt obligatorii

Mai mult ca atat, putem utiliza si

**except (typerror1, typerror2, typerror3) as *exceptie***

* Cand trimitem cu raise o exceptie, putem trimite si un mesaj cu exceptia:

raise NameError(„Un text”)

* Daca rise nu are cui trimite exceptia, compilatorul o va executa. In C++ asa ceva nu are sens, caci nu face nimic.
* with – se foloseste pentru a face codul mai usor de citit si a administra mai bine exceptiile si resursele. Evita bugurile si potentialele probleme

**User Defined Exception**

* Pentru a crea o clasa de exceptie, clasa trebuie sa mosteneasca Exception
* Putem sa trimitem textul erorii chiar creand o valaore in constructor si apoi sa o afisam prin print prin supraincarcarea lui print cu \_\_str\_\_(self) sau direct sa o afisam:
* class Error(Exception):  
   def \_\_init\_\_(self,text):  
   self.text = text  
    
    
  def main():  
   try:  
   a = 20  
   if a == 10:  
   raise Error("Eroare necunoscuta")  
   else:  
   raise Error("Hopa, ceva nu a mers bine.")  
   except Error as e:  
   print(e.text)  
    
  main()
* Putem pur si simplu sa cream o clasa goala si sa prindem o eroare de acel tip, iar textul il afisam direct in except:
* Putem face si ca in java sa trimitem in constructor prin super().\_\_init\_\_(text) la clasa Except care are deja supraincarcat operatorul print

**Lucrul cu fisierele**

* Python poate opera cu 2 tipuri de fisiere: text si in format binar
* Memoria in PC e de 2tipuri: primara(care sterge totul din ea cand calculatorul se stine) si secundara(care salveaza totul).
* Deschiderea fisierului se face cu open(’adresa’,’mod’)
* Pentru a evita conflictele cauzate de caracterele speciale, ca \n etc. folosim **r** in fara la string : **r**„\nun text”
* Moduri:

r – doar citire,nu creare fisier**(e default)**

w – doar scriere,creare fisier nou

a – append,creare fisier

a+ - append si citire si creare fisier

r+ - citire si scriere

w+ - scriere si citire si creare fisier

r/w/a(b)+-la fel ca cele de sus, doar ca in format binar

* Un obiect fisier este ca un iterator ce poate fi citit cu o bucla for ce citeste fiecare linie
* File.read() – returneaza un string cu intregul continut din fisier
* File.readline() – citeste o singura linie la apelare si o returneaza. Muta bufferul dupa cu o linie.
* File.readlines() – citeste toate liniile si le returneaza pe fiecare sub un tabel de stringuri
* File.read(nr) – returneaza un nr de caractere din fisier. O valoare negativa e identic cu nici-un argument. Odata ce bufferul ajunge la final de fisier, el va returna caractere goale.De aceea, conditia a == ‚’, unde a = file.read(1), e suficienta pentru a sti cand fisierul a ajuns la final.
* File.write(text) – pentru a scrie. Returneaza nr de biti scrisi.
* File.writelines(tabel de stringuri) – scrie cate un string dintr-un tabel de stringuri, dar nu il va pune in linie noua!E exact ca write, doar ca e pentru tablouri de stringuri.
* File.close() – inchide fisierul deschis si elibereaza memoria alocata pentru

deschiderea lui

* File.seek(pozitie) – seteaza bufferul la pozitia data

Daca vrem sa cream un fisier, e important sa alegem modul w sau w+, nu r+

* with – face ca fisierul sa se inchida fara comanda close() si face ca sa nu fie necesar de mai folosi blocuri try in caz ca ar aparea erori, evitand posibilele buguri.

with open('C:\\Users\\user\\Desktop\\c.txt','w+') as file:  
 file.write("Saaaaalut")

* Pentru a putea utiliza with cu obiecte declarate de noi, e nevoie sa adaugam metodele \_\_enter\_\_(self) – pentru a deschide un fisier si a returna obiectul lui variabilei dupa as si \_\_exit\_\_(self,....) pentru a-l inchide. Defapt, rolul lui with este de a fi siguri ca nu lasam vreo resursa deschisa.
* class fisier(object):  
   def \_\_init\_\_(self,path):  
   self.path = path  
    
   def \_\_enter\_\_(self):  
   self.file = open(self.path,'w+')  
   return self.file  
    
   def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
   self.file.close()  
    
  def main():  
   with fisier("C:\\Users\\user\\Desktop\\c.txt") as file:  
   file.write("Un mesaj de test")  
    
    
  main()

anume cuvantul cheie as face ca ma intai sa se creeze un obiect,dupa metoda \_\_enter\_\_ din obiect sa fie apelata si sa se returneze un obiect care va fi preluat de variabila dupa as. Dupa ce se executa instructiunile din bloc, se apeleaza \_\_exit\_\_

* Variabilele declarate in with sunt disponibile doar in blocul lui
* Metodele \_\_enter\_\_ si \_\_exit se numesc Context Manager
* Putem sa le inlocuim prin decoratorul @context manager, ce trebuie importat: from contextlib import contextmanager. Astfel, cream doar o metoda ce va fi modificata de decorator, dar care trebuie sa aiba un yield in loc de return, caci ceea ce se va returna (de continuat)

**MultiProcessing**

* **Proces – instanta a unui program ce este executata. Orice proces este caracterizat de:**
* **Program de executat**
* **Date associate(vaiabile,workspace)**
* **Stare**
* MultiProcessing – Capacitatea sistemului de operare de a avea mai multe procesoare
* MultiProcesor – calculator cu mai mult de un processor central

Multi-core-processor – calculator cu un processor ce are mai multe core

* Fiecare process ruleaza independent si are propriul spatiu de memorie.
* Procesele ruleaza in acelasi timp. Chiar daca ruleaza mai multe procese de cat cores, ele ruleaza concurrent, adica unul se opreste pe putin timp, altul ii ia locul,apoi cel din urma iar se porneste si tot asa.
* Trebuie sa importam multiprocessing
* Crearea unui process are loc astfel:

Proces = multiprocessing.Process(target = functie,args=(parametri actuali,))

* **Proces.start()**
* **Proces.join()** – opreste procesul curent pana cel ce utilizeaza join nu este terminat
* **Proces**.**close()**
* **Proces.kill()**
* Cand Python citeste un python fisier, el creaza niste variabile inainte de a executa codul. De ex. ,\_\_name\_\_ este o variabila care stocheaza numele modulului unde se executa codul. Cand se importa,de ex in \_\_main\_\_,un modul, intregul lui cod este intai executat si apoi se trece la urmatoarea instructiune din \_\_main\_\_. Orice modul ce este executat de Python direct va fi \_\_main\_\_ iar cele importate vor avea un \_\_name\_\_ ca si numele fisierului lor. Deci, daca un fisier va fi direct executat, \_\_name\_\_ va fi \_\_main\_\_, insa daca e importat de un alt fisier ce e executat, \_\_name\_\_ al sau va coincide cu cel al fisierului. Totusi, un modul e \_\_main\_\_ doar cand e executat de compilator. Daca el mai apoi e executat de alt modul a 2 oara, atunci a 2 oara el nu va mai fi \_\_main\_\_.
* Python are un registru, numit sys.modules care verifica daca un modul a fost sau nu importat deja intr-un alt modul. De aceea, imoportarea aceluiasi modul de 2 ori sau importarea a 2 module riciproc nu va genera errori sau importari infinite.
* Din cauza ca orice proces creat, la creara sa importa modulul de baza, s-ar crea un numar infinit de procese, deoarece la importarea lui main, iar se va crea un process si apoi iar si iar, de aceea se foloseste mereu inainte de a declara un process:

If \_\_name\_\_ == “\_\_main\_\_

* **Process.is\_alive()** – returneaza true sau false
* **Import os** ne permite sa vedem id al unui process cu atributul **proces.pid** – pid de la process id sau functia **os.getpid()** pentru procesul main
* Orice variabile intr-un process sunt independente de celelalte procese. Aici deja variabilele din process au propriile lor obiecte de care sunt legate. Chiar si daca o variabila va fi declarata globala in process, ea va fi globala doar pentru el, nu si pentru \_\_main\_\_ sau alte procese. Daca un process apeleaza si alte functii, oricum toate obictele modificate vor fi modificate doar in cadrul procesului, nu a intregului program.
* Putem crea un array si o valoare ce sa fie impartita de toate procesele:

Array = multiprocessing.Array(“tipul prescurtat”,nr de element)

Valoare = multiprocessing.value(“tip”,valoarea inititala[optional])

Apoi pentru a modifica valoarea, valoare.value = noua valoare . Pentru a vedea valorile lui array folosi array[:]

* Cand un program ruleaza, un process server process este mereu pornit.Mereu cand e nevoie sa se creeze un nou process, procesul parinte se duce la server si cere sa fie creat un nou process. Un server process pastreaza obiecte ce pot fi accesate de toate procesele in comun.
* Clasa Manager ne permite sa controlam server processul.

**ObjManager = multiprocessing.Manager()** – cream un manager pentru procese

**Namespace = objManager.Namespace()** – returneaza o referinta la NameSpace al proceselor

**Lista\_comuna = objManger.list([…])** – creaza o lista si intorce o referinta la ea. Putem folosi si metodele Queu() . Nu e neaparat sa cream un objManager, putem direct sa cream lista comuna prin multiprocessing.Manager().list(…). Coada adauga elemente prin put()

**Namespace.var = valoare** – adauga o valoare in namespace

* Pentru a crea o clasa proces, trebuie ca ea sa mosteneasca multiprocessing.Process, sa trimita constructorului superclasei constructorul prin multiprocessing.Process.\_\_init\_\_(self) si sa suprascrie metoda run(). In asa caz, nu mai trebuie target sau args

**Synchronized Process**

* Sincronizarea asigura ca nu cumva 2 sau mai multe procese sa execute simultan aceeasi parte de cod
* Cand mai multe procese executa acelasi cod, e greu de spus ce schimbari ar putea face codul in viitor.
* Pentru a sincroniza, folosim un obiect de tip lock care foloseste un obiect de semaphore, care verifica daca nu cumva o secventa de cod e executata in acelasi timp de mai multe procese. Ea face ca procesele sa astepte pana altele se executa.
* Pentru a crea un obiect de tip lock:

Lock = multiprocessing.lock()

* Codul ce trebuie sincronizat se pune intre:

Lock.acquire() si Lock.release()

Lock.acquire() – face ca nici un alt process sa nu mai poata acesa codul

Lock.release() – face ca codul sa fie disponibil pentru alt process

* Fiecare process ocupa un core, dar uneori ar putea fi posibil ca unui process sa nu-I ajunga un core, si daca alte core sunt disponibile, ar fi mai usor sa imparta procesul tuturor core. Acest lucru se poate face cu un obiect e tip Pool.
* Obiect = multiprocessing.Pool()
* Odata ce cream un obiect pol, nu mai trebuie sa cream procese. El singur o va face. Totusi, putem doar sa efectuam anumite functii asupra lui, ca obiect.map(target,obj) etc.

Threads

* Thread – entitate/submultime dintr-un process ce poate fi executata in cadrul acestuia. Orice thread poate exista doar intr-un process.
* Toate threadurile impartasesc aceleasi variabile globale, nu le creaza pe ale sale asa ca procesul.
* La crearea unui Thread, el tot va importa modulul actual, ca si procesul,de aceea e nevoie de a pune in main **if \_\_name\_\_==”\_\_main\_\_”:**
* Daca se executa multe threaduri, ele vor face ca procesele, adica vor face schimb de resurse intre ele, asa incat fiecare se va executa un timp, dupa se va opri si va da resursele altui thread, apoi acel thread se va opri putin si va da altuia, si asa ele se executa concurential si concomitant.
* Import threading
* Obiect = threading.Thread(target=,args=,name=)
* **Threading.current\_thread().name** – returneaza numele threadului executat
* Pentru a sincroniza o parte de cod, cream un obiect lock de tip semafor:

Lock = threading.Lock()

* Obiectul de tip semafor se asigura ca partea de cod sincronizata este executata de un singur thread, si celelalte nu vor putea sa o execute pana cel current nu va termina.
* **acquire**() – are ca parametru default False,ceea ce inseamna ca codul este disponibil sa fie executat de un thread.Cand un thread executa codul sincronizat,el va modifica parametrul in True, ceea ce inseamna ca secventa de cod este blocata si orice thread ce va incerca sa o execute, va intalni True si nu va incepe sa o execute. Threadurile verifica des daca codul dat nu devine cumva disponibil.
* **release**() – daca codul era pus pe True, il deblocheaza setand parametrul False. In caz contrar, daca parametrul deja era False, da eroare.
* Pentru a crea o clasa thread, trebuie ca ea sa mosteneasca threading.Thread, sa trimita constructorului superclasei constructorul prin threading.Thread.\_\_init\_\_(self) si sa suprascrie metoda run(). In asa caz, nu mai trebuie target sau args.In constructor putem seta si self.thread\_name si self.thread\_ID
* threading.active\_count() – returneaza nr de threaduri active
* threading.enumerate() – returneaza o lista cu threadurile active
* threading.main\_thread() – returneaza threadul de baza(cel cu care Python a inceput)

**Abstract Class**

* **Clasa abstracta – clasa ce contine una sau mai multe metode abstracte**
* Clasa abstracta este doar un model pentru viitorii copii. Orice metoda abstracta trebuie sa fie suprascrisa de clasele copii. Ea poate avea fie metode simple, fie abstracte.
* from abc import ABC, abstractmethod
* ABC – abstract base class
* Clasa ce va fi abstracta trebuie sa mosteneasca clasa ABC
* Pentru a seta o metoda ca abstracta, folosim decoratorul @abstractmethod
* Python va furniza un un obiect functie metodei date si va seta clasa ca abstracta.
* Metoda abstracta oricum trebuie sa aiba un corp cu pass. Poate avea si un alt corp.

**Class parinte(ABC):**

@abstractmethod

def func():

pass

* O clasa abstracta nu poate fi instantiata, ca in C++ sau Java
* O clasa copil ce mosteneste de la o clasa abastracta daca nu suprascrie metodele abstracte, va fi si ea clasa abstracta.
* Putem apela si functia abstracta din clasa parinte cu super().metodaAbstracta(), desi asa ceva in C++ si Java nu ar putea fi posibil.

**Poliformism**

* Inseamna sa folosim aceeasi functie, dar in moduri diferite.
* Putem supascrie metodele exact ca in Java si C++, desi nu exista un @Override
* In Python putem supraincarca metode, dar, problema e ca spre deosebire de C++ sau Java, putem utiliza doar ultima metoda scrisa,nu si restul!
* Totusi, putem utiliza un modul.
* **from** multipledispatch **import** dispatch

**si deasupra metodei scriem**

@dispatch(int,int) de ex daca sunt 2 parametri formali,

* **dispatch creaza un obiect cu toate metodele si apoi o cauta pe cea potrivita**

**Serialization**

* Se foloseste modulul pickle
* **Import pickle**
* Pickling – transformarea obiectului ierarhie in intr-un stream de biti
* Unpickling – transformarea unui stream de biti intr-un obiect ierarhie
* **pickle.dump(obj,file,protocol=None,\*,fix\_imports = True)** – serializeaza obiectul ierarhie
* **pickle.load(file,\*,fix\_imports=True,encoding=”ASCII”,errors=”strict”)** – deserializeaza si returneaza obiectul ierarhie intreg
* File.Flush() – pentru a ne asigura ca s-a scris obiectul
* Ca si in java, cream un obiect fisier ce contine locatia, dar trebuie neaparat sa il deschidem si in byte mod!
* **pickle.dump*s*(obj) –** returneaza reprezentarea obiectului sub forma de biti
* **pickle.load() = pickle.**Unpickler(file).load()
* **pickle.loads() –** citeste o reprezentare de biti a unui obiect
* **exception pickle.PickleError** – orice eroare de tip pickle
* **PicklingError – eroare la scrierea unui obiect**
* **UnpicklingError – eroare la citirea obiectului din file**

import pickle  
  
class person:  
 university = "University of California"  
 def \_\_init\_\_(self,name,surname,age):  
 self.name = name  
 self.surname = surname  
 self.age = age  
 def \_\_str\_\_(self):  
 return ("University: "+self.university+"\nName: "+self.name+"\nSurname: "+self.surname+"\nAge: "+str(self.age))  
  
class file:  
 def \_\_init\_\_(self,path,mode):  
 self.path = path  
 self.mode = mode  
  
 def \_\_enter\_\_(self):  
 self.file\_object = open(self.path,self.mode)  
 return self.file\_object  
  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 self.file\_object.flush()  
 self.file\_object.close()  
  
def main():  
 # Write an object  
 try:  
 global Eduard  
 Eduard = person("Mititiuc","Eduard",20)  
 with file(r"C:\Users\user\Desktop\file.txt","wb") as my\_file:  
 pickle.dump(Eduard,my\_file)  
  
 except Exception:  
 print("ERROR to writing the object.")  
  
 else:  
 print("Object has written succesfully")  
 finally:  
 print("Object in bytes:",pickle.dumps(Eduard))  
  
  
 # read our written object  
 try:  
 global obj  
 with file(r"C:\Users\user\Desktop\file.txt","rb") as my\_file:  
 obj = pickle.load(my\_file)  
 print(obj)  
  
 except Exception:  
 print("ERROR to reading the object")  
  
 else:  
 print("Object has readed succesfully")  
  
 finally:  
 obiect = pickle.dumps(Eduard)  
 print("Object Transformed: ",pickle.loads(obiect))  
  
  
main()

* Putem si crea un obiect de tip pickle**r(doar scriere)** si unpickle**r(doar citire)** sa le trimitem direct fisierul.In asa fel vom face citirea si scrierea fara a include un fisier obiect. Nu putem sa folosim pickler pentru a si citi!
* import pickle  
    
  class person:  
   university = "University of California"  
   def \_\_init\_\_(self,name,surname,age):  
   self.name = name  
   self.surname = surname  
   self.age = age  
   def \_\_str\_\_(self):  
   return ("University: "+self.university+"\nName: "+self.name+"\nSurname: "+self.surname+"\nAge: "+str(self.age))  
    
  def main():  
   file = open(r"C:\Users\user\Desktop\file.txt","wb")  
   pickler\_object = pickle.Pickler(file)  
    
   Eduard = person("Mititiuc","Eduard",20)  
   pickler\_object.dump(Eduard)  
   file.flush()  
   file.close()  
    
   file = open(r"C:\Users\user\Desktop\file.txt","rb")  
   unpickler\_object = pickle.Unpickler(file)  
   print(unpickler\_object.load())  
    
    
  main()
* **Daca sunt mai multe obiecte scrise, Unpickler.load(file) le va returna pe toate intr-o lista!**

**Garbage Collection si Calcularea Referintelor**

* In Python, Alocarea si dezalocarea memoriei e automata, fiind facuta prin Garbage Collection(proces de adunare a obiectelor nefolosite si de stergere) . Acesta are 2 componente:

1. **Calcularea Referintelor**

Cand un obiect nu mai are nici-o referinta, el este sters. Un obiect poate avea si mai multe referinte. El mereu calculeaza cate referinte sunt la el, si daca un obiect pierde o referinta, numarul de referinte scade cu 1. Daca nr de referinte e 0, obiectul e sters. De ex:

A = 9

A = 5

Dupa ce A devine egal cu 9, referinta catre 9 e pierduta, deci 9 nu mai are nicio referinta si este sters. Aceasta strategie de stergere pe baza nr de referinte nu poate fi nici de cum oprita.O referinta se creaza cand o var e egala cu acel obiect, cand obiectul e adaugat intr-o structura de date sau e trecut ca parametru actual pentru o functie. Pentru a vedea nr de referinte a unui obiect:

**import sys**

**sys.getrefcount(var spre obiect)**

1. **Generational Garbage Collection**

class obj:  
 def \_\_str\_\_(self):  
 return "a"  
  
a = obj()  
a.add = a  
print(a.add)  
del a  
print(a.add)

Aici avem un obiect ce are un atribut cu o referinta la el insusi. Totusi, operatorul del desi va face ca obiectul sa nu mai poata fi accesat, el oricum continua sa existe, deoarece a nu are referinta la el, dar add are.del doar face ca atributul sa nu mai fie legat de obiect, nu sterge propriu zis obiectul, ci doar legatura. Mai apoi deja se decide de Calcularea Referintelor daca obiectul trebuie sters(daca are 0 referinte) ori nu. Aceasta problema este un ciclu de referinta ce nu poate fi rezolvata de Calcularea Referintelor de mai sus.

Mai mult ca atat, daca un obiect adauga un atribut ce are o referinta la alt obiect si cel din urma face la fel cu primul, iar se primeste ca fiecare obiect are cate 2 referinte. De ex:

class obj:  
 pass  
  
obj1 = obj()  
obj2 = obj()  
  
obj1.nr1 = obj2  
obj2.nr2 = obj1

Deci,stergerea lui obj1 si obj2 ar rupe cate o referinta asupra fiecarui obiect, dar una tot va mai exista, cele pe care le au ele una la alta.

Garbage Collector pastreaza o urma a tuturor obiectelor din program. Garbage Collection este doar procesul de verificare a obiectelor din Garbage Collector ce poate sterge si obiectele ce au referinta ciclica. Python executa Garbage Collection. Garbage Collection are 3 generatii.

**1**.Obiectele abea create fac mereu parte din prima generatie. Totodata, in prima generatie se calculeaza anume cate referinte are fiecare obiect pe parcursul executiei programului, chiar se verifica si fiecare obiect catre ce alte obiecte are referinte. Apoi, obiectele ce se depisteaza ca raman fara referinte, se sterg si cele ramase se trimit in generatia a 2.In prima generatie isi face treaba doar Calcularea Referintelor. Totusi, daca sunt obiecte ce nu se pot sterge, adica ce raman cu referinte, si isi pastreaza referinta mult timp, e clar ca numarul de obiecte create nu e egal cu cel sterse, si cand se ajunge la o oarecare limita permisa, se activeaza deja si Generational Garbage Collector ce se va ocupa de generatia a 2.

**2**.In generati a 2 ajung si obiectele cu referinta ciclica. Aici deja python le va verifica si le va elimina. De asta se ocupa Generational Garbage Collector

**3.**In generatia a 3 ajung doar obiectele ce le folosim de-a dreptul in program si nu au fost deci sterse in generatia a 2.

Fiecare generatie are un nr maxim de obiecte ce le poate pastra(default e 700,10,10).Prima generatie e cel mai des verificata, a 2 mai rar.Programul verifica astfel mereu fiecare din primele 2 generatii. Daca acel nr maxim e atins, se face o curatare a obictelor ca nr sa fie mai mic.Spre deosebire de Calcularea Referintelor, putem modifica comportamentul fiecarei generatii administrare de Generational Garbage Collector si nr maxim de obiecte .

>>> import gc  
>>> gc.get\_threshold()

get\_threshold() – returneaza limita de obiecte din fiecare generatie. In mod mplicit, prima are 700, a 2 si a 3 au cate 10

>>> gc.get\_count()

Arata nr de obiecte existent in fiecare generatie.

gc.collect()

porneste procesul Garbage Collection

>>> gc.set\_threshold(1000, 15, 15)

Modifica nr maxim din fiecare generatie

Nu trebuie sa modificam Garbage Collection deoarece el si asa este foarte bine optimizat!

* Putem suprascrie si del prin \_\_del\_\_

**Regular expressions**

* Are rolul de gasi un string sau seturi de stringuri.
* import re
* obj = re.search(text,variabila) – returneaza un obiect ce contine prima pozitie a textului gasit si ultima, ce pot fi accesate prin obj.start() si obj.end(). In caz contrar, returneaza None
* findall(text,var) – returneaza o lista ce contine de mai multe ori acelasi string text, in dependenta de cate ori s-a gasit in variabila string.
* split(“ ”,string) – transforma un text string in substringuri, iar fiecare nou substring e creat cand se gasesc caractere/caracter in dependenta de conditia din “ “
* sub(string,nou,var) – inlocuieste un string cu altul
* subn(string,nou,var) – ca si sub, doar ca arata si cate inlocuiri a facut
* compile(‘\d’) – creaza un obiect ce are direct metacaractere setate si orice metoda nu am folosi asupra lui, nu va mai trebui sa punem ‘\d’
* escape(text) – returneaza textul cu \ in loc de space
* **Metacaractere: caractere speciale, ce sunt ca o conditie ce trebuie respectata. Se pun mereu intre “”**
* **[]** – set de caractere

1. [abcd]– returneaza fiecare aparitie a lui a,b,c si d
2. [a-b] - gaseste toate caracterele de la a la b si le returneaza intr-o lista in ordinea aparitiei lor
3. [^abc] – returneaza totul inafara de caracterele a,b si c
4. [0-5][5-9] – returneaza orice numere ce au prima cifra intre 0 si 5 si a doua intre 5 si 9

* ***\*** secventa speciala:

1. **“\AText”** – returneaza Text daca Text e la inceput de string
2. **“\bText”** sau **“Text\b”** – returneaza Text daca el e present la inceput sau final de cuvant**(cuvantul e separate de space anume!!!)**
3. **“\BText”** sau **“Text\B”** - returneaza Text daca el nu e present anume la inceput sau final de cuvant,ci prin mijloc undeva
4. **“\d”** – returneaza cifrele ce se contin in text
5. **“\D”** – returneaza numai nu litere ce se contin in text
6. **“\s”** – returneaza fiecare caracter gol(space) ce se gaseste in text
7. **“\S”** – returneaza totul inafara de caracterele goale
8. **“\w”** – returneaza litere,cifre si \_
9. **“\W”** – nu returneaza litere,cifre si \_
10. **“text\Z”** – returneaza text daca el e la final de **string**
11. **“\**.” – face ca . sa fie ca simplu caracter, nu meta

? – unul sau nici un caracter va fi afisat

+ - unul sau mai multe caractere

\*- 0 sau 1 sau mai multe caractere

Pentru ? si \* unde se ia si varianta cu 0, caracterele ce nu corespund, adica au 0, sunt puse ca “” in obiectul returnat. Daca avem \* si + si se gasesc deci mai multe caractere ce corespund conditiei, se returneaza pana la primul ce nu corespunde.

* **“t..t”** – returneaza orice cuvinte ce incep cu t si se termina cu t, dar au orice in loc de puncte. Putem pune ce litere vrem si unde vrem punctele
* **“^text”** – returneaza text daca stringul incepe cu el
* **“text$”** – returneaza text daca stringul se termina cu el
* **“teX\*t”** – returneaza caracterele ce incep cu te, se termina cu X si au in mijloc oricati de X sau niciunul. Daca in loc de X am pune ., nu ar conta ce caractere sau cate sunt intre te si t. Adica daca mai avem cuvinte prin care se gaseste X, ele tot vor fi luate in considerare!Nu se va verifica doar un cuvant.
* **“teX+t”** – returneaza caracterele ce incep cu te, se termina cu X si au in mijloc cel putin un X. Daca in loc de X am pune . , nu ar conta ce caractere sau cate sunt intre te si t, dar nu 0!
* **“teX?t”** – returneaza caracterele ce incep cu te, se termina cu X si au in mijloc un X sau niciunul. Daca in loc de X am pune ., nu ar conta ce caracter este intre te si t
* **“teX{nr}t”** - returneaza caracterele ce incep cu te, se termina cu X si au in mijloc nr de X. Daca in loc de X am pun ., nu ar conta ce caracter este intre te si t, ci doar cate
* **“a{2,4}”** – returneaza orice cuvant ce are 2,3 sau 4 de a
* **“text1|text2”** – cauta ambele texte si le returneaza de fiecare data cand le gaseste
* **Daca X e un string mai lung de 1 caracter, il luam intre (), de ex**:

**“t(ex){2}t”. Ceea ce e intre () se numeste grup**

* Putem adauga inca un parametru **flags=re.IGNORECASE** pentru a face literele mari sa fie tratate ca si cele mari.
* Putem crea si grupuri. Orice grup este scris intre () si pot fi mai multe intr-o singura conditie. De ex:

Result = re.search(r"(\b[A-Z]+\b).+(\b\d+)

aici avem 3 grupuri. Primul este cel general, si apoi vin cate 2 in el. Al 2 grup va verifica ca cuvantul sa inceapa si sa se termine cu mai multe litere mari, apoi al 3 grup va verifica sa inceapa cu cifre. Intre ele pot fi orice alte caractere. Astfel, se vor crea grupuri. Primul grup va contine o fraza mare, al doilea grup cuvinte cu litere mari, iar al treilea cifre. Se vor returna stringuri pentru fiecare grup, inclusive cel general.

* Pentru a accesa lista de grupuri, utilizam **Result.groups()**
* Pentru a accesa fiecare grup, utilizam **Result.group(nr)**
* **Pentru a putea accesa groups sau group, folosim doar search sau finditer, doar ca search va cauta doar prima aparitie, pe cand finditer e un iterator. Findall la fel va returna in print stringurile fiecarui grup, doar ca nu poate fi accesata comanda group.**

import re  
  
string = "El s-a nascut pe 21.07.1955 si a decedat pe 17.01.2020"  
  
for i in re.finditer(r"((\d+)\.(\d+)\.(\d+))",string):  
 print(i.group(0))

* 0 va fi mereu grupul initial, 1 primul,2 al doilea si tot asa
* Daca vrem sa stocam mai multe grupuri generale, putem utiliza **finditer(tinta)** care va crea un iterator de grupe pentru fiecare cuvant/cuvinte gasit.
* Daca sunt mai multe grupe, putem sa le accesam pe tote prin group(nr1,nrx) adica de la nr1 pana la nr2
* Grupurile automat ca nume numarul lor de ordine. Insa nu e prea comod sa utilizam cifre pentru a le accesa. De aceea, putem sa le oferim nume. Trebuie doar de pus ?P<nume grup> inaintea la conditie si apoi o accesam cu group(“nume grup”) nu cu group(nr)

string = "El s-a nascut pe 21.07.1955 si a decedat pe 17.01.2020"  
  
for i in re.finditer(r"((?P<data>\d+)\.(?P<luna>\d+)\.(?P<an>\d+))",string):  
 print(i.group("data","luna","an"))

import re  
  
string = "Adresa edikutsu2002@mail.ru este blocata de pe data de 21.07.2022.Va rugam sa ne contanctati pe adresa help.facebook@gmail.com pana pe 14.08.2022"  
  
for i in re.finditer(r"(?P<data>(\d+)\.(\d+)\.(\d+))",string):  
 print(i.group("data"))  
  
for z in re.finditer(r"(?P<m>(\b( )[^ ]+(mail.ru|gmail.com)( )\b))",string):  
 print(z.group("m"))

* Se poate utiliza si **\nr grupa** pentru a verifica un cuvant ce se repeta iar. De ex: **w(…)[0-5] w\1[0-5],** acest \1 nu e egal cu (…)! El va fi egal cu ceea ce va gasi (…), adica cu acea secventa de text

import re  
  
word = "word1 word1"  
  
print(re.match(r"w(...)[0-5] w\1[0-5]",word))

**Lucruri suplimentare despre Python**

* In Python la fel exista un operator ternar, dar nu are aceeasi sintaxa ca in C++ sau Java:

a = "like"  
b = 1 if a=='like' else 0  
print(b)

* O bucla for sau while, ca si un try, pot avea si ele un else. Acest else se executa doar daca bucla nu este brusc intrerupta de un break.

for i in range(10):  
 if i == 5:  
 break  
else:  
 print("Nu se va afisa")  
  
for i in range(10):  
 pass  
else:  
 print("se va afisa")

* Cand un modul este importat, Python mereu va executa intai acel modul pentru a crea namespaceul sau. Totusi, daca e vorba de modulul main, de exemplu, am putea sa ne confruntam cu problema ca importarea lui in alt modul ar crea reexecutarea lui. Sau in general, unele module pot avea instructiuni ce nu trebuie executate cand sunt importate. Fiecare modul are o variabila \_\_name\_\_ care poarta numele modulului. Pentru a exclude executia lui la import folosim un if:

**If \_\_name\_\_ == “\_\_main\_\_”:**