Inteligența Artificială în Inginerie Energetică

NOȚIUNI DE PROGRAMAREA CALCULATOARELOR

Mihail - Bogdan CĂRUȚAȘIU, 2024
FACULTATEA DE ENERGETICĂ
UNIVERSITATEA NAȚIONALĂ DE ȘTIINȚĂ ȘI TEHNOLOGIE POLITEHNICA BUCUREȘTI

editura POLITEHNICA PRESS

ISBN: 978-606-9608-78-4

Mihail - Bogdan CĂRUȚAȘIU

INTELIGENȚA ARTIFICIALĂ ÎN INGINERIE ENERGETICĂ

NOȚIUNI DE PROGRAMAREA CALCULATOARELOR

Editura POLITEHNICA PRESS
BUCUREȘTI, 2024

Copyright © 2024, Politehnica Press
Toate drepturile asupra acestei ediții sunt rezervate editurii.

Adresă: Calea Griviţei, 132
10737, Sector 1, București
Telefon: 021.402.90.74

Referenți științifici:
prof.univ.dr.ing. **Horia NECULA**
conf.dr.ing. **Constantin IONESCU**

ISBN: 978-606-9608-78-4

Cuprins

1. INTRODUCERE 3
2. PYTHON – LIMBAJ DE PROGRAMARE PENTRU ȘTIINȚA DATELOR ȘI A.I. 4
2.1. MODALITĂȚI DE INSTALARE A LIMBAJULUI DE PROGRAMARE PYTHON 5
2.2. PRIMA APLICAȚIE SCRISĂ ÎN PYTHON 10
3. PARTICULARITĂȚI DE SINTAXĂ A LIMBAJULUI DE PROGRAMARE PYTHON 11
3.1. EXEMPLE DE UTILIZARE A CUVINTELOR CHEIE ÎN COD 17
3.2. EXEMPLE DE UTILIZARE A FUNCȚIILOR ÎNCORPORATE 49
4. TIPURI DE DATE PRINCIPALE ȘI METODELE ASOCIATE LOR 95
4.1. DATE NUMERICE: INT, FLOAT, COMPLEX 99
4.2. ȘIRURI DE CARACTERE: STR 104
4.3. LISTE: LIST 131
4.4. DICȚIONARE: DICT 147
4.5. TUPLURI: TUPLE 165
4.6. SETURI: SET 176
5. TIPURI DE DATE PROPRII – INTRODUCERE ÎN PROGRAMAREA OBIECTUALĂ (CLASE PYTHON) 188
5.1. FOLOSIREA CLASELOR 188
5.2. DEFINIREA UNEI CLASE – CUVÂNTUL CHEIE CLASS 189
5.3. MOȘTENIREA CLASELOR 197
SURSE SUPLIMENTARE 200
BIBLIOGRAFIE 201

Cuprins Figuri

Figura 1. Descărcarea limbajului de programare Python – site oficial 6
Figura 2. Pașii pentru instalarea limbajului de programare Python 6
Figura 3. Exemplu de linii de comanda în IDLE.. 7
Figura 4. Captură de ecran din PyCharm Comunity Edition IDE 7
Figura 5. Captură de ecran din Spyder IDE 8
Figura 6. Captură de ecran din Visual Studio Code IDE... 8
Figura 7. Inițierea unei file noi din interfața IDLE 10
Figura 8. Prima aplicație scrisă în Python..11
Figura 9. Generarea cuvintelor cheie din Python utilizând IDLE..15
Figura 10. Utilizarea celor două cuvinte cheie de tip boolean în cadrul unei expresii logice 18
Figura 11. Schema de execuție a blocului `try - except` 46
Figură 12. Injectarea datelor în șablon 64
Figură 13. Exemple de utilizare a funcției **`range()`** 85
Figură 14. Exemplu de scriere a unei funcții încorporate și fereastra ajutătoare generată............... 95
Figura 15. Atribuirea valorilor în memorie... 96
Figura 16. Exemplu de indexare listă imbricată..134
Figură 17. Indexarea pozitivă/negativă a unei liste - preluare https://www.geeksforgeeks.org/.......134
Figură 18. Diferența a trei seturi (a, b și c)..181

1. Introducere

Societatea modernă este din ce în ce mai mult bazată pe utilizarea, prelucrarea și interpretarea datelor. Colectarea acestora, sub o formă sau alta, a avut loc din cele mai vechi timpuri și reprezintă o modalitate prin care omul poate învăța lucruri noi, corelând patternuri (engl. model, șablon) și trăgând concluzii. Un exemplu concret este cel al unui copil căruia îi sunt necesare mai multe procese pentru a învăța un fapt banal: un corp cu o temperatură ridicată îl poate răni. O singură interacțiune cu obiectul fierbinte (acțiune similară cu achiziția unui singur set de date de intrare) poate să nu fie suficient pentru a trage o concluzie (acțiune similară cu găsirea unui pattern). Repetarea acțiunii oferă copilului mai multe date pe care le poate procesa și pe care le va folosi pentru a ajunge la concluzia evidentă pentru un adult: obiectele fierbinți îi pot provoca răni. Sau vizionarea primului film de mister, cu întorsătură de situație, în care vinovatul este persoana care nu ar fi putut fi niciodată anticipată. Cu cât numărul de filme de acest gen este mai mare și deznodământul mai surprinzător, cu atât creierul este mai pregătit pentru situații neașteptate, iar deznodământul probabil intuit cu o acuratețe crescută. Cele mai des utilizate sisteme fundamentate pe Inteligență Artificială (I.A.) funcționează similar: ele necesită un set de date pentru antrenare și validare, dezvoltându-și astfel capacitatea de generalizare, pentru ca apoi să extragă informații utile (patternuri) și să aplice cele „învățate“, în mod eficient, pe un nou set de date.

Industria energetică este similară în acest sens, mai cu seamă că acest sector este unul major implicat în reducerea dependenței Uniunii Europene (UE) de combustibilii fosili, și pentru care conceptul de digitalizare (procesul de transformare a informației din format fizic într-un format sub formă de digiți) este fundamental, vizând în mod direct corelarea sistemelor energetice existente cu algoritmi moderni care pot achiziționa, interpreta și acționa conform datelor primite. Acest lucru implică formarea interdisciplinară a viitorilor experți din domeniul ingineriei energetice, care, pe lângă cunoștințele tehnice specifice domeniului, pentru o mai bună adaptare la aceste cerințe trebuie să își dezvolte și competențe digitale ce nu trebuie să se limiteze doar la simpla utilizare a unor sisteme de calcul și/sau interconectarea acestora la sistemele energetice exploatate.

În această ordine de idei, lucrarea de față – constituită din trei părți complementare – dorește să răspundă acestei majore provocări, fiind gândită sub forma unei îmbinări armonioase între teorie și exemple, ce are ca inspirație paradigma celei mai utilizate metodologii moderne de predare: *hands-on* – învățare prin implicare activă și acumularea de cunoștințe prin practică efectivă în detrimentul învățării clasice. Pentru a realiza aceasta, cartea are prezentate o mulțime de exerciții rezolvate și exemple detaliate care vin să explice exhaustiv teoria prezentată în cel mai simplist mod. Mai mult, ținând cont că nu se adresează unor experți în programarea calculatoarelor sau a științei datelor și algoritmilor de inteligență artificială, lucrarea evidențiază algoritmii pe exemple din domeniul științei ingineriei energetice.

Abordând o tematică atât de complexă – prezentarea modalității de integrare a elementelor de Inteligență Artificială în Industria Energetică și exemplificarea comprehensivă a acestora – lucrarea este gândită în trei părți, de la simplu la complex, care vor contribui la formarea completă a experților în domeniul energetic. Cele trei părți sunt:

1. PARTEA I – **NOȚIUNI DE PROGRAMAREA CALCULATOARELOR**
2. PARTEA II – **NOȚIUNI DE ȘTIINȚA DATELOR**
3. PARTEA III – **IMPLEMENTAREA ALGORITMILOR DE INTELIGENȚĂ ARTIFICIALĂ**

În cadrul primei părți se va descrie funcționalitatea limbajului de programare Python, pentru care se vor explica toate funcționalitățile, funcțiile și metodele încorporate, cuvintele cheie și principalele tipuri de date încorporate. Formarea cunoștințelor minime de programarea calculatoarelor este esențială pentru înțelegerea algoritmilor complecși bazați pe inteligența artificială. Mai mult, în jurul Python s-a creat un ecosistem complex prin care se pot analiza datele – știința datelor (utilizând anumite librării specifice precum `numpy`, `pandas`, `matplotlib`, `seaborn` detaliate în `PARTEA a II-a`) și implementa algoritmii de inteligență artificială (utilizând librării precum `scikit learn`, `tensorflow`, `pytorch`, `keras` detaliate în `PARTEA a III-a`), propulsând acest limbaj de programare pe locurile fruntașe la utilizabilitate.

2. Python – limbaj de programare pentru știința datelor și A.I.

Prin acțiunea de programare a calculatoarelor se înțelege procesul de dezvoltare a unui program care funcționează pe un calculator și care deservește unei aplicații specifice. Tot ce rulează pe un calculator personal, telefon, server, TV, echipament electric de comandă etc. este rezultat în urma dezvoltării diverselor programe cu ajutorul unor așa-numite limbaje de programare. Limbajul de programare Python, denumit după un grup celebru de comedianți din Marea Britanie – Monty Python (și nu după șarpele piton), așa cum însuși dezvoltatorul (programatorul olandez Guido van Rossum) a afirmat, are un scop general, el fiind caracterizat de o curbă lină a procesului de învățare, ceea ce îl recomandă celor nu foarte familiarizați cu domeniul programării calculatoarelor, așa cum, de cele mai multe ori, este și cazul experților în domeniul ingineriei energetice. Acest limbaj de programare oferă posibilitatea atât a programării structurale (bazată pe crearea funcțiilor), cât și a celei orientate pe obiecte (POO). Se poate descărca gratuit de pe site-ul oficial disponibil la următoarea adresă: https://www.python.org/ – la momentul scrierii acestei lucrări varianta stabilă era Python 3.11.

Comparat cu alte limbaje de programare, codul Python se scrie relativ rapid, are sintaxa (modul de scriere a instrucțiunilor) gândită astfel încât să poată fi înțeleasă mai ușor (aproape de pseudocod) și beneficiază de o comunitate apreciabilă de utilizatori care au consolidat o serie de module și librării armonios integrate pentru dezvoltarea de proiecte în majoritatea ariilor: aplicații web, aplicații mobile, achiziția, stocarea (baze de date) și interpretarea datelor (știința datelor), machine learning, deep learning etc. Toate aceste aspecte recomandă utilizarea programului Python în lucrarea de față ca punct de plecare în rândul studenților Facultății de Energetică interesați de domeniul științei datelor și implementării inteligenței artificiale în diverse sisteme energetice. Mai mult, gradul mare de utilizare a acestui limbaj face ca online (majoritar gratuit) să existe o multitudine de materiale de învățare a bazelor programării, a limbajului în sine, ca și exemple de diverse aplicații dezvoltate din zero. Cea mai bună variantă, recomandată și de cei mai experimentați programatori, este analizarea documentației limbajului. Acesta, pe lângă sintaxă, conține tutoriale și exemple de bune practici și este disponibil gratuit accesând site-ul din referință: (docs.python.org, 2023).

Comparativ cu limbajele de programare compilate (mai aproape de limbajul mașină și de procesor), cum ar fi C, C++ sau Fortran, Python rulează relativ lent, fiind necesare diverse adaptări atunci când se impune o viteză mai mare de rulare – rularea codului pe plăci Raspberry Pi sau integrarea algoritmilor de inteligentă artificială (scriși în Python) în aplicații web. Principala cauză pentru o viteză mai scăzută de rulare este faptul că limbajul Python este un limbaj interpretat, adică executarea comenzilor se face linie cu linie. Însă rapiditatea cu care codul Python este scris, testat și livrat în producție face ca viteza mai redusă de rulare a aplicației să nu conteze, de aceea giganți IT, precum Google, YouTube, Netflix, Dropbox, BitTorrent, Maya, NSA, iRobot, îl folosesc exclusiv sau în combinație cu alte limbaje de programare pentru dezvoltarea și funcționarea aplicațiilor lor (Python Succes Stories, 2023).

Raportat la limbajele de programare scriptice, mai aproape de nivelul de pregătire al non-programatorilor, Python este gratis (spre deosebire de Matlab), are un ecosistem de librării pentru o multitudine de domenii (comparat cu Scilab, Octave, R etc.) și este un limbaj de programare cu scop general, nespecific pentru anume nișă, precum celelalte. În plus, Python permite scrierea sub formă de scripturi (foarte asemănătoare cu scrierea în Matlab sau Octave) prin intermediul interpretorului de cod IPython – *Interactive Python* (VanderPlas, 2017). Pentru dezvoltarea unor aplicații complexe, cum este și cazul celor de inteligență artificială, scrierea codului sub formă de scripturi nu este eficientă, ci se impune folosirea unor programe specifice, denumite *Integrated Development Environment*, IDE — Mediu Integrat de Dezvoltare. În această lucrare se va folosi un IDE de uz general, denumit *Visual Studio Code*, și care poate fi descărcat de pe site-ul oficial, disponibil la https://code.visualstudio.com/download/. Trebuie menționat faptul că toate exemplele vor fi scrise în acest IDE, putând fi copiate, testate, modificate și/sau îmbunătățite de cititori.

Deși scopul acestei lucrări nu este de a iniția cititorii în programarea calculatoarelor și nici de a realiza o prezentare exhaustivă a limbajului de programare Python, pentru a putea utiliza ecosistemul științific Python (format din nenumărate librării de calcul matricial, manipularea datelor tabulare, analiza statistică și reprezentarea grafică a datelor) este absolut necesară înțelegerea modului de funcționare a modulelor – instalarea, importarea, inițializarea și utilizarea funcțiilor/claselor constituente. Pentru acest lucru, utilizatorii trebuie să dobândească cunoștințe minime de programare și integrare a acestora utilizând cod scris în Python.

2.1. Modalități de instalare a limbajului de programare Python

Faima de care se bucură acest limbaj de programare și utilizarea sa pe scară largă au făcut ca mașinile de calcul de la Apple să aibă instalată cea mai nouă și stabilă versiune de Python în sistemul de operare macOS. Pentru utilizatorii de sisteme de operare Windows există posibilitatea de a instala versiunea dorită după ce se descarcă de pe site-ul oficial disponibil la următorul URL: www.python.org/donwloads/. Acest mod va instala versiunea aleasă de Python și va adăuga un IDE dedicat, denumit IDLE (*Integrated Development and Learning Environment* – Mediu Integrat de Dezvoltare și Învățare). În figura 1 este prezentat modul de descărcare și instalare al limbajului de programare pe un calculator personal ce funcționează cu un sistem de operare Windows.

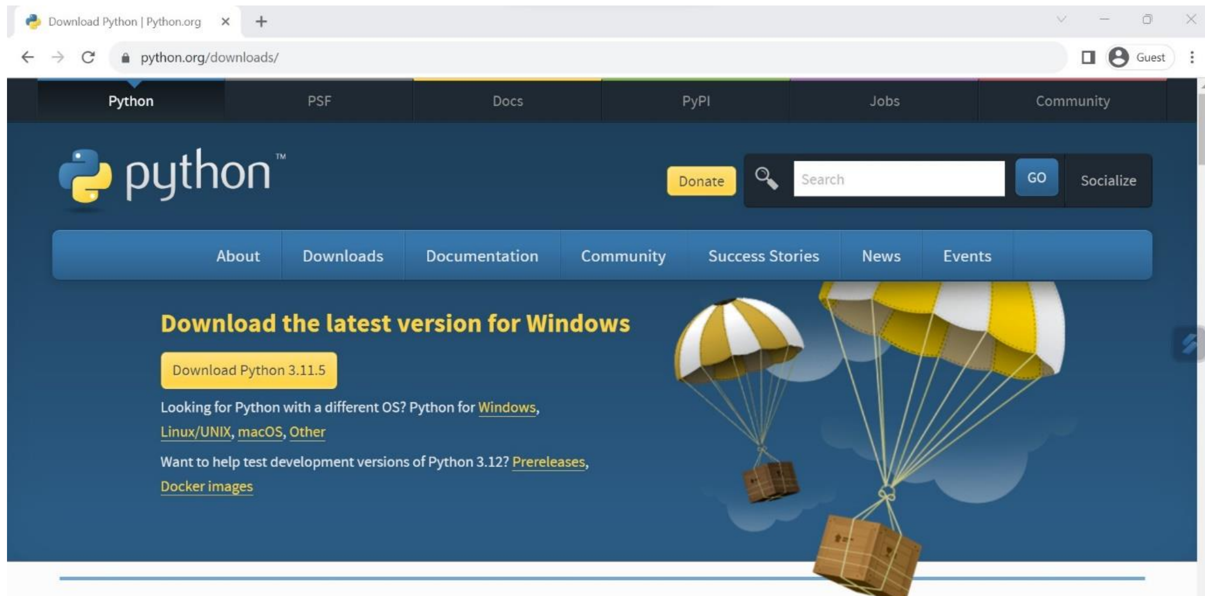


Figura 1. Descărcarea limbajului de programare Python – site oficial

După descărcarea kitului se poate instala limbajul de programare bifând opțiunea *Add python.exe to PATH* (pentru a putea folosi Python din orice terminal) și selectând *Customize installation – Choose location and features* pentru a putea personaliza modul de instalare, caracteristicile și locația. Aceste aspecte nu sunt atât de importante în funcționalitatea limbajului, dar oferă o flexibilitate mai mare experților. Mai mult, se oferă posibilitatea ca Python să fie instalat pentru toți utilizatorii bifând opțiunea *Install Python 3.11 for all users*. Pașii necesari instalării sunt prezentați în succesiunea de capturi de ecran din figura 2.

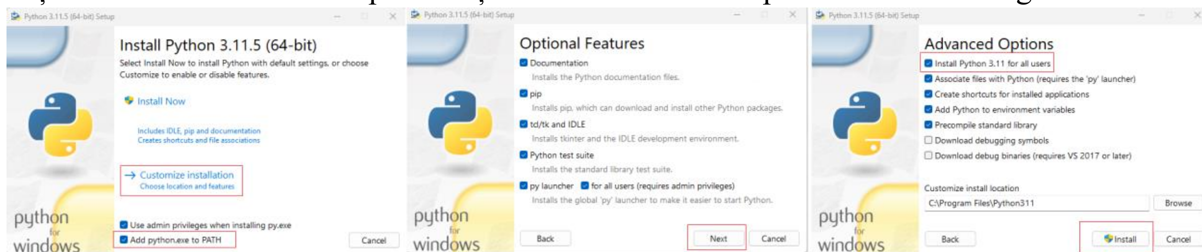


Figura 2. Pașii pentru instalarea limbajului de programare Python

Această variantă de instalare este cel mai des utilizată în rândul începătorilor și prezintă avantajul că instalează doar limbajul de programare Python și software-ul IDLE, care se poate deschide căutând în bara de start a sistemului de operare Windows. Interfața grafică a programului IDLE este foarte asemănătoare cu cea în care se pot crea scripturi sub forma liniilor de comandă în Matlab, mai exact *Matlab Command Window*. În plus, modul de lucru este similar. Acest mediu minimal de dezvoltare al programelor Python se bazează pe principiul REPL (*Read-Eval-Print-Loop*), adică: citește (*read*) fiecare instrucțiune (linie cu linie), o evaluează/execută (*eval*), afișează rezultatul evaluării (*print*) și repetă în buclă acest procedeu (*loop*) de câte ori este nevoie până parcurge întreg scriptul. În figura 3 este detaliată consola IDLE (*Interactive Python Shell*) și sunt scrise trei linii de comandă. De menționat faptul că sintaxa și funcțiile principale vor fi detaliate în capitolul următor.

- importul unei librării standard: `import sys`
- afișarea versiunii de Python folosită: `print(sys.version)`
- un calcul simplu și afișarea răspunsului: `10+3`

- poziția curentă a cursorului este indicată prin cele trei caractere ” >>>”
- executarea comenzii se face apăsând `Enter` de la tastatură

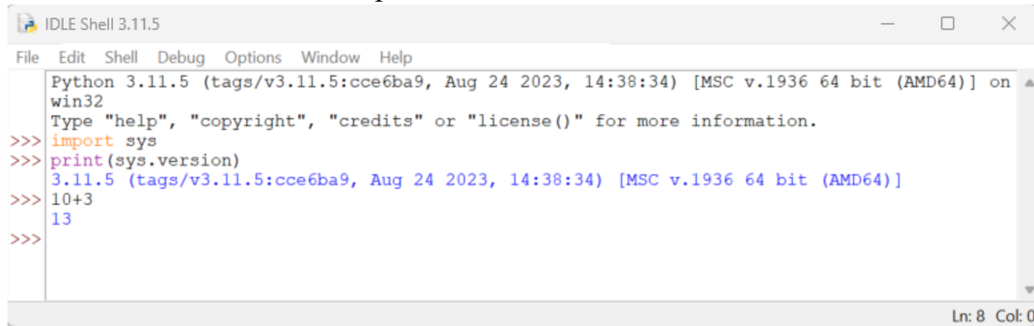


Figura 3. Exemplu de linii de comanda în IDLE

De cele mai multe ori, programele realizate (cum este și cazul algoritmilor augmentați de elemente de inteligență artificială) necesită medii de scriere a codului mai complexe, unde se pot crea mai ușor funcționalitățile necesare și algoritmi cu o complexitate crescută, compuși din cod reutilizat din diverse librării terțe. Pentru aceste funcționalități se pot folosi unul sau mai multe IDE-uri specifice limbajului Python sau chiar de uz general adaptate prin extinderea funcționalității lor. Există o multitudine de astfel de aplicații performante de editare de text: https://wiki.python.org/moin/IntegratedDevelopmentEnvironments. Cele mai utilizate sunt: **PyCharm** (considerat cel mai complex și complet editor de text pentru Python), **Visual Studio Code** (un editor de text gratuit de utilizare generală dar care, prin intermediul extensiilor și librăriilor, poate fi folosit și pentru crearea programelor Python), **Sublime Text** (mediu de editare scris chiar în Python), **Spyder** (IDE dedicat pentru dezvoltarea programelor din sfera științifică și care înglobează deja un număr mare de librării dedicate acestui domeniu – calcul numeric și matricial, analiză matematică, calcul simbolic etc.), **Jupyter-Lab** (un IDE web). Toate aceste editoare de text se folosesc împreună cu varianta de Python instalată anterior. Următoarele trei figuri reprezintă capturi de ecran de la utilizarea IDE-urilor PyCharm, Spider și Visual Studio Code.

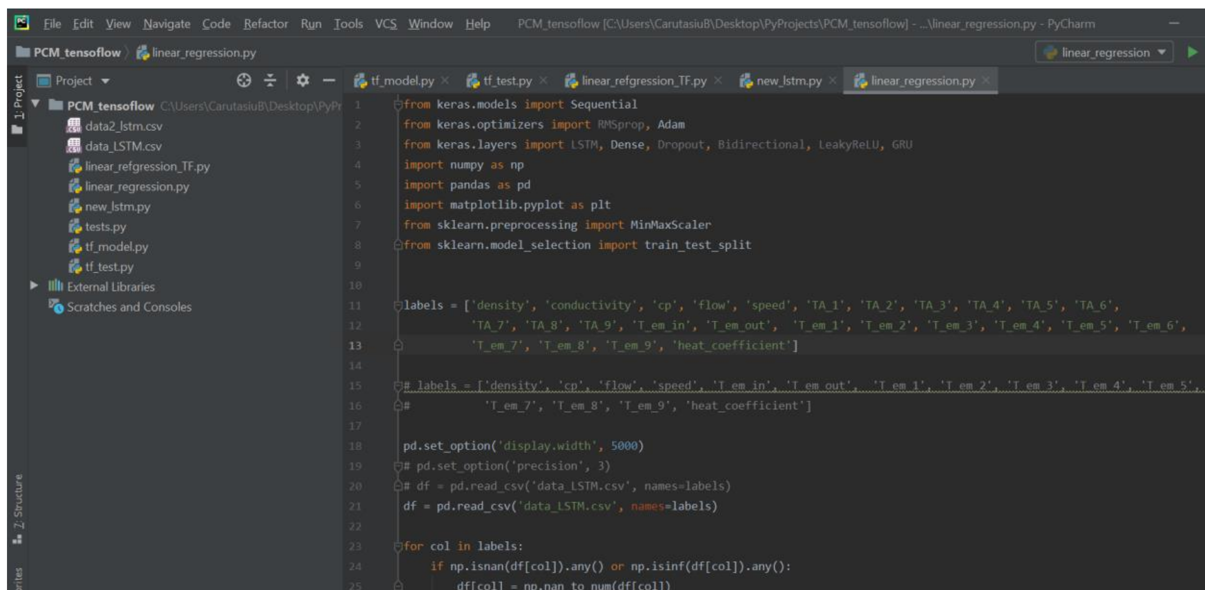


Figura 4. Captură de ecran din PyCharm Comunity Edition IDE

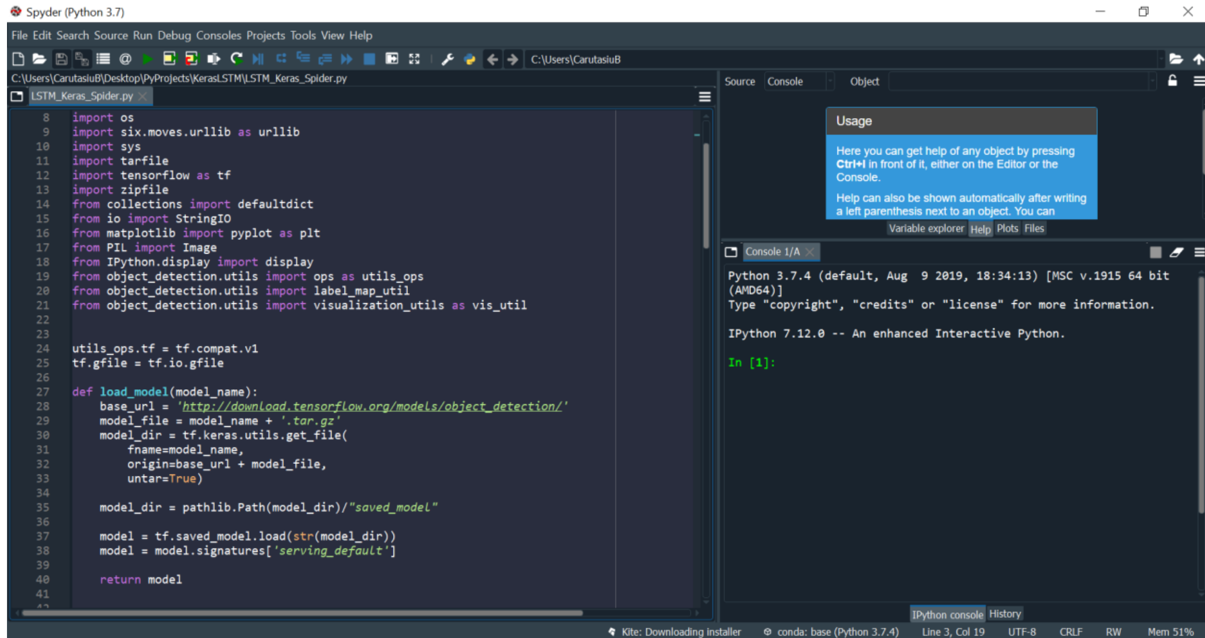


Figura 5. Captură de ecran din Spyder IDE

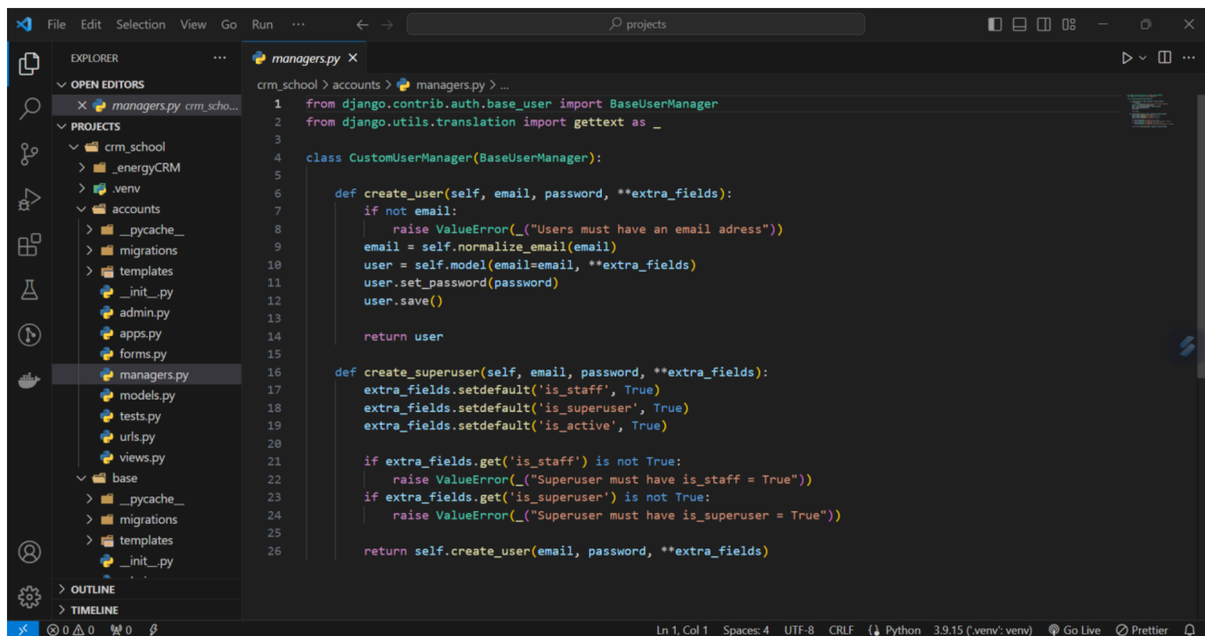


Figura 6. Captură de ecran din Visual Studio Code IDE

O altă variantă de instalare a limbajului Python și care este sugerată de majoritatea programatorilor (în special cei care se ocupă cu știința datelor) este de a descărca un întreg ecosistem Python sub forma unei platforme denumite Anaconda (sau Miniconda – varianta *categorie ușoară*) de pe site-ul dedicat: https://www.anaconda.com/. Aceasta este cea mai populară platformă dedicată dezvoltării aplicațiilor din domeniul științei datelor din lume datorită librăriilor dedicate preinstalate și programelor de scris cod gândite special pentru domeniul în discuție, incluzând IDE-uri speciale. Anaconda este, după cum se specifică pe site-ul oficial, o distribuție pentru dezvoltarea proiectelor data science gratuită în limbajele de programare Python și R. Această distribuție complexă este compusă din:

- **conda** – un manager pentru crearea și modificarea mediilor virtuale de programare;

- **terminal Anaconda** – pentru manipularea mediilor virtuale (asemănător cu terminalul CMD din Windows);
- **Anaconda Navigator** – o aplicație desktop ce permite lansarea mediilor virtuale și a IDE;
- 250 **pachete și librării** – optimizate pentru dezvoltarea proiectelor de data science;
- **IDE** – Spyder (*Scientific Python Development Environment*).

Varianta simplă (*light weight*), Miniconda, nu conține decât managerul conda și limbajul Python, dar oferă o mai mare flexibilitate în alegerea librăriilor și a versiunilor acestora, lucru care rezolvă în mare parte problema de incompatibilitate între diverse pachete și librării (de exemplu, librăria de machine learning, **TensorFlow**, nu funcționează decât cu Python versiune mai veche de 3.9!) și versiunea de Python existentă. Acest lucru se realizează prin crearea unor medii virtuale de programare (*virtual environments*), care vor fi populate cu librăriile necesare și variantele care au compatibilitate. Compatibilitatea și interdependențele funcționale între librării se identifică analizând documentațiile acestora.

Existând aceste două variante de distribuție Anaconda, apare întrebarea care să fie folosită în dezvoltarea aplicațiilor. Răspunsul nu este atât de evident și depinde foarte mult de nivelul de cunoștințe al utilizatorilor. În cazul începătorilor (cu experiență zero sau foarte puțină) se recomandă utilizarea distribuției Anaconda, deoarece conține deja o multitudine de librării care sunt configurate și interconectate astfel încât să asigure buna funcționalitate fără problema compatibilității între acestea. În schimb, un utilizator mai experimentat, care are cunoștințe în utilizarea liniilor de comandă într-un terminal, va opta pentru varianta Miniconda care este varianta minimală funcțional și care, neinstalând toate acele librării și IDE-ul, are o dimensiune considerabil mai mică decât versiunea integrală.

În continuare sunt prezentate câteva comenzi pentru a crea un mediu virtual utilizând managerul conda (lista integrală se regăsește pe https://conda.io/), cu mențiunea că terminalul se deschide (după instalarea distribuției Anaconda sau Miniconda) din bara de căutare a sistemului de operare Windows introducând cuvintele cheie „anaconda prompt“:

Tabel 1. *Instrucțiuni conda pentru medii virtuale*

Instrucțiunea	Ce face
`conda create --name mediuVirtual`	creează un mediu virtual denumit ”mediuVirtual”
`conda create --name mediuVirtual python==3.8`	creează un mediu virtual denumit ”mediuVirtual” cu versiunea de Python 3.8
`conda create --name mediuVirtual tensorflow`	creează un mediu virtual denumit ”mediuVirtual” cu versiunea curentă și cu o librărie terță: TensorFlow
`conda activate mediuVirtual`	activează mediul virtual creat
`conda info –envs`	afișează în terminal informații despre toate mediile virtuale existente
`conda install numpy`	instalează în mediul virtual activ librăria specificată (numpy în cazul acesta).
`conda list`	afișează toate librăriile instalate în mediul virtual activ

2.2. Prima aplicație scrisă în Python

Indiferent de opțiunea de instalare și de IDE-ul utilizat, sintaxa și modul de lucru în limbajul Python nu se modifică și exemplele prezentate în acest capitol vor funcționa la fel, generând aceleași rezultate. Acum se poate scrie o primă aplicație cu ajutorul căreia să putem afișa pe ecranul terminalului următorul mesaj: „Inteligența Artificială în Ingineria Energetică“. Pentru a scrie și rula un fișier Python simplu se poate opta pentru una din următoarele variante:

- se scrie codul Python într-un fișier text simplu (în Notepad sau alternativă), se salvează cu extensia .py și se rulează din Command Prompt specificând calea către fișier – aceasta este cea mai puțin recomandată alternativă;
- se deschide fereastra de comandă (Command Prompt) și în terminal se scrie `python`, comandă care va activa limbajul de programare direct în consolă – este recomandată pentru teste scurte care nu necesită multe linii de cod;
- se deschide IDE-ul preferat, se creează o nouă filă Python (cu extensia .py), se scrie codul și se rulează direct din IDE – este cea mai recomandată variantă.

Pentru acest exemplu simplu se va folosi IDLE care se instalează automat în momentul instalării limbajului de programare Python. Pe lângă modul de lucru în consolă (v. exemplul din Figura 3), acesta oferă și posibilitatea de a crea scripturi care pot conține mai multe linii de cod care vor fi executate integral la rularea fișierului. Pentru a se deschide un fișier nou, din fereastra IDLE se alege comanda **`New File`** din meniul **`File`** (sau mai rapid prin combinația de taste `Ctr+N`). Acest lucru deschide o nouă fereastră (inițial fără nume – *untitled*) în care se pot scrie liniile de cod, se va salva cu denumirea dorită (**`Save as`**) și se va rula apăsând comanda **`Run`** bara de comenzi sau tasta **`F5`**. Rezultatele rulării scriptului vor fi afișate în consola inițială.

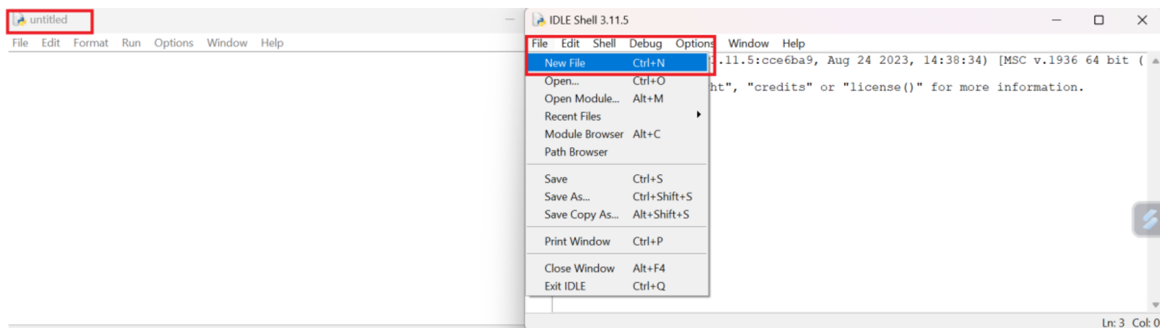


Figura 7. Inițierea unei file noi din interfața IDLE

În fereastra nou deschisă se introduc liniile de cod prezentate în Figura 8. Prima linie (cea mare începe cu #) este un comentariu. Aceasta va fi ignorată de interpretorul Python la rularea scriptului și are doar rolul de a oferi programatorilor informații despre cod în sine. Pe cea de-a doua linie este scrisă o simplă instrucțiune care va afișa pe ecranul consolei argumentul funcției `print()`. În acest caz, argumentul funcției este un șir de caractere (strig) care compun mesajul dorit. După scrierea liniilor de cod se salvează fișierul sub denumirea dorită (aici s-a optat pentru **`prima_app.py`**) și se rulează, rezultatul fiind afișat în aplicația IDLE Shell.

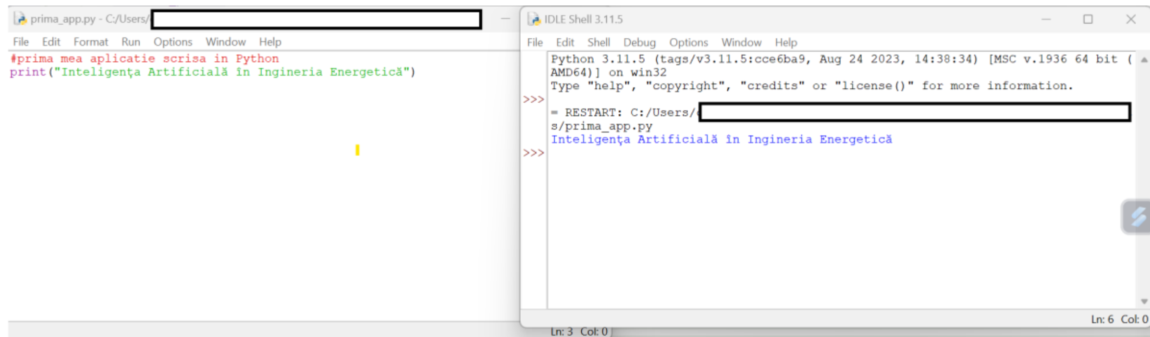


Figura 8. Prima aplicație scrisă în Python

3. Particularități de sintaxă a limbajului de programare Python

Încă de la început, Python a fost gândit ca o alternativă la limbajele de programare existente care să fie mai ușor de învățat pentru începători. Pentru a putea îndeplini acest aspect, dezvoltatorii se ghidează după un set de principii care se poate citi în limba engleză din orice IDE prin simplul import al modulului **this** (**import this**), modul care s-a instalat odată cu versiunea descărcată de pe site-ul oficial sau instalată prin distribuția Anaconda. Această instrucțiune afișează pe consola IDE cele 19 principii care stau la baza scrierii codului în limbajul Python și care sunt colectate sunt denumirea ***The Zen of Python, by Tim Peters.*** Dintre principii, cele mai interesante sunt primele patru care, în limba engleză, sunt prezentate în tabelul alăturat.

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Această colecție de principii alcătuiesc, de fapt, o sinteză a modului în care a fost gândită sintaxa limbajului de programare Python. Prin sintaxa unui limbaj de programare se înțelege **suma tuturor regulilor de scriere corectă** (în sensul acceptării codului la rularea sa – fără generarea erorilor de sintaxă). Un cod scris corect semantic va rula fără erori sau excepții, dar nu înseamnă neapărat că va genera rezultatul dorit. Sintaxa unui limbaj de programare se referă la totalitatea elementelor care asigură funcționalitatea acestuia. Printre acestea, se numără **modul de creare a identificatorilor** (posibilitățile valide de a denumi variabile, funcțiile, clasele etc.), **cuvintele cheie** și **funcțiile încorporate** (*built-in*).

IDENTIFICATORUL

Reprezintă numele ales pentru a folosi în cod diferite constante, variabile, funcții și clase create de programator cu scopul de a le putea diferenția de alte entități built-in (încorporate) sau create anterior. În sintaxa oricărui limbaj de programare există un set de reguli pentru scrierea acestor identificatori. În Python, aceștia se pot scrie ca o succesiune oricât de lungă de litere (minuscule sau majuscule), cifre sau caracterul special underline (_), **fără ca primul caracter să fie cifra** și **fără a se utiliza alte simboluri speciale**, precum !, @, #, $ etc. sau spații. O altă restricție este că un indicator **nu poate fi un cuvânt cheie existent în sintaxa Python**, detaliat în paragraful următor. Altfel spus, `variabila1`, `x`, `my_function`, `_my_ann`, `reteaNeuronalaArtificial`, `ReteaNeuronala` sunt exemple valide de identificatori, în timp ce `1variabila`, `functie$`, `numar!` vor genera o eroare de sintaxă invalidă.

Python face parte din limbajele de programare **case sensitive**, însemnând că face diferențierea între majuscule și litere mici în cadrul unui cuvânt scris. Cu alte cuvinte, identificatorii `inteligenta`, `Inteligenta`, `INTELIGENTA` vor descrie trei obiecte distincte și nu înseamnă același lucru. Ca o altă observație, nu este indicat ca funcțiile sau clasele create să aibă aceiași indicatori ca funcțiile/clasele deja existe. Spre exemplu, o funcție denumită `sum()` va suprascrie codul din funcția built-in și, chiar dacă programul nu va genera erori la compilare, este posibil ca funcționalitatea limbajului să fie alterată.

Ca o regulă nescrisă a programatorilor Python, variabilele și funcțiile care conțin mai mult de un cuvânt se scriu cu underline (denumit și snake_case), de exemplu `functia_suma`, în timp ce numele claselor se scriu cu majusculă la fiecare cuvânt mai puțin primul, de exemplu `clasaSuma`. Aceste reguli nu fac parte din sintaxa Python, nerespectarea lor negenerând nicio eroare de sintaxă.

INDENTAREA – SPAȚIEREA

Face parte chiar din sintaxa limbajului și este mai mult decât o modalitate de a face codul lizibil și cu un aspect plăcut. Spre deosebire de alte limbaje de programare care folosesc diferite caractere, pentru a indica unde se termină o instrucțiune sau un bloc de cod (begin și end sau { și }), în Python se folosește chiar caracterul *linie nouă* – „\n“. Un bloc de cod are toate liniile indentate cu același număr de spații – se recomandă 4 spații, echivalentul apăsării tastei `Tab`. În IDE-uri specifice scrierii codului Python, indentarea se face automat la apăsarea tastei Enter. În exemplul următor se pot analiza mai multe blocuri de cod indentate corespunzător. Detalii despre cuvintele cheie utilizate și funcționalitatea programului în sine sunt oferite în capitolele și subcapitolele următoare (textul este selectat intenționat pentru a se observa spațiile – sub formă de puncte în IDE Visual Studio Code).

```python
def functie_caracteristici(nume: str, varsta:int, ocupatie: str) -> str:
    
    caracteristici = {}
    if nume != "" and ocupatie != "":
        caracteristici["nume"] = nume
        caracteristici["ocupatie"] = ocupatie
    else:
        print("data necorespunzator")
        caracteristici['nume'] = "John Doe"
        caracteristici["ocupatie"] = 'No Data'
    if varsta >= 0:
        caracteristici["varsta"] = varsta
    else:
        print("Varsta necorespunzatoare")
        caracteristici["varsta"] = 0

        for number in range(13):
            for number2 in range(0, number):
                print(number - number2)

    return caracteristici


caracteristici_bogdan = functie_caracteristici("bogdan", 36, 'lector')
for k, v in caracteristici_bogdan.items():
    print(k, v)
    
print(f"Ma numesc {caracteristici_bogdan['nume']}, am {caracteristici_bogdan['varsta']} si lucrez ca {caracteristici_bogdan['ocupatie']}")
```

COMENTARIILE

Comentarea codului este o parte esențială a oricărui limbaj de programare și reprezintă texte care nu sunt luate în considerare de interpretor la compilarea codului sursă ci au rolul de a oferi detalii programatorilor despre funcționalitate. Mai mult, aceste comentarii sunt utilizate pentru crearea documentației unui program. În Python, comentariile linie se inițializează utilizând semnul „#“. Tot ce este după acest semn și până la o nouă linie (newline) este considerat comentariu și textul respectiv nu va fi considerat a fi component din codul sursă. În exemplul următor sunt scrise 3 comentarii linie, fiecare începând cu caracterul #. A se observa faptul că un comentariu linie poate fi scris chiar și în continuarea unei instrucțiuni Python – așa cum este cazul celui de-al doilea comentariu (`# idem`), dar și înainte de aceasta, pe același rând.

```python
# aceste este un comentariu pe o singura line
print("Cartea Inteligenta Artificiala in Inginerie Energetica") # idem
# print - functie incorporata in Python va afisa pe ecran o informatie
# argumentul este format din sirul de caractere "Cartea Inteligenta
Artificiala in Inginerie Energetica"
```

OUTPUT

```
Cartea Inteligenta Artificiala in Inginerie Energetica
```

În cazul în care informația din comentariu este mai lungă de un singur rând, se pot utiliza comentariile multi-linie. O primă metodă de a le genera este de a folosi semnul # pentru fiecare linie necesară. Acest lucru poate fi în schimb anevoios, lucru pentru care sintaxa limbajului de programare Python oferă o variantă alternativă: utilizarea ghilimelelor triplate – simple sau duble: `'''` sau `"""`. Trebuie menționat faptul că finalul comentariului trebuie, de data aceasta, indicat prin folosirea aceluiași tip de ghilimele triple folosit la început. Exemplu:

```
'''Acesta este un comentariu pe mai multe linii – se utilizeazaz ghilimele simple'''

"""Acesta este  un comentariu pe mai  multe linii – se utilizeaza ghilimele duble"""
```

Acest lucru are implicații directe în generarea automată a unui cod suplimentar denumit ”docstring” – prescurtare pentru *documentation string* (traducere: șir de caractere pentru documentarea codului). Docstring-urile se generează doar prin utilizarea ghilimelelor duble triplate (`"""acest sir de caractere va fi generat in docstring-ul aplicației"""`) și vor fi asociate unui obiect Python prin atributul `__doc__`.

OBSERVAȚIE! Utilizarea dublării caracterului ”_”: `__doc__` (CORECT) versus `_doc_` (INCORECT). Caracterul __ poartă denumirea de `dunder` (*double underline*).

În exemplul următor de cod este creată o funcție pentru care se scrie un docstring simplu care explică ce face acea funcție și ce returnează (mai multe detalii despre utilizarea funcțiilor în § 3 – FUNCȚII ÎNCORPORATE). Informațiile conținute în docstring se pot genera utilizând metoda `__doc__` sau funcția încorporată `help()` –, rezultatul fiind identic.

```
def functie_putere(tensiune:int, intensitate: int) -> int:
    """
    Aceasta functie va calcula puterea electrica
    parametri:
        tensiune - se introduce o valoarea numerica pentru tensiunea electrica in Volti
        intensitate - se introduce o valoare numerica pentru intensitatea curentului electric in Ampleri

    returneaza:
        valoarea puterii electrice calculata cu expresia: P = U x I
        valoarea se va masura in Watt
    """
    putere = tensiune * intensitate
    return putere

print("Utilizare metoda .__doc__")
print(functie_putere.__doc__)
```

OUTPUT

```
Utilizare metoda .__doc__

    Aceasta functie va calcula puterea electrica
    parametri:
        tensiune - se introduce o valoarea numerica pentru tensiunea electrica in Volti
        intensitate - se introduce o valoare numerica pentru intensitatea curentului electric in Ampleri

    returneaza:
        valoarea puterii electrice calculata cu expresia: P = U x I
        valoarea se va masura in Watt
```

Există mai multe stiluri de a scrie docstring-uri, dar detalierea lor nu face obiectul acestei lucrări. Se pot studia aici.

CUVINTELE CHEIE

Cuvintele cheie sunt cuvinte (scrise în limba engleză) care nu pot fi modificate și care nu pot fi atribuite ca nume de variabile/identificatori, funcții, clase etc., acestea fiind rezervate pentru ca interpretorul să poată transforma codul din limbaj Python în cod mașină. Acestea compun structura limbajului Python și numărul lor poate diferi puțin la variantele mai vechi ale limbajului. Pentru a vedea care sunt și numărul lor efectiv în varianta de Python folosită, se poate importa modulul `keyword` – folosind instrucțiunea `import keyword`. Pentru a vedea exact numărul cuvintelor cheie fără a le număra pe consolă după afișare, se poate utiliza funcția `len(keyword.kwlist)`, care va afișa pe ecranul terminalului IDLE numărul de elemente din lista cuvintelor cheie (generată prin atributul `kwlist`). Dacă se folosește alt IDE (de exemplu, Visual Studio Code), este necesară utilizarea unei funcții în plus pentru afișarea rezultatelor pe consolă, sintaxa devenind `print(len(keyword.kwlist))`. Funcțiile implicite Python vor fi discutate mai pe larg în capitolul următor. În fragmentul de cod din figura 9 sunt exemplificate procedura de afișare a cuvintelor cheie existente în limbajul de programare Python și numărul acestora. După importul librăriei `keyword`, se generează lista care conține toate cuvintele cheie (`keyword.kwlist`), pentru ca apoi să se afișeze numărul de elemente din lista generată (`len(keyword.kwlist)` – funcția `len()` (din engl., *length* – lungime) generează lungimea argumentului introdus, o listă în acest caz. Același rezultat îl va genera și utilizarea funcției `help()` cu argumentul `"keywords"`: `help("keywords")` – atenție: argumentul `"keywords"` se scrie între ghilimele, fiind un șir de caractere! În Figura 9 sunt exemplificate ambele metode pentru afișarea în terminal a tuturor cuvintelor cheie existente în sintaxa limbajului de programare Python.

Figura 9. Generarea cuvintelor cheie din Python utilizând IDLE

FUNCȚII ÎNCORPORATE

Sunt funcții (bucăți de cod – subrutine ce au o funcționalitate bine stabilită) care sunt instalate automat cu versiunea Python. Aceste funcții pot fi puțin alterate de la o versiune la alta, dar în linii mari ele își păstrează funcționalitatea și scopul pentru care au fost create. Este întotdeauna benefic a studia documentația oficială în momentul în care se descarcă o versiune anume. Funcțiile încorporate se pot analiza accesând următorul link.

Similar cu funcțiile matematice, funcțiile din programarea calculatoarelor primesc ca input date, le prelucrează și returnează un rezultat care mai apoi este utilizat în altă parte. Un mare avantaj în utilizarea funcțiilor este caracterul reutilizabil al acestora. Spre exemplu, unei funcții care are ca scop afișarea datelor pe consola terminalului, `print()` în Python, ar fi dificil și redundant de a-i fi scris corpul codului la fiecare reutilizare, încălcând totodată și un principiu important din programarea calculatoarelor: DRY – Don't Repeat Yourself (engl., nu te repeta) care are ca scop reducerea codului scris în dezvoltarea aplicațiilor.

Lista funcțiilor încorporate în Python versiunea 3.12 în ordine alfabetică se regăsește în Tabelul 2. Mai mult, fiecare funcție are denumirea scrisă sub forma unui hyperlink care deschide pagina de pe site-ul oficial Python unde există descrierea funcției respective. Detalierea acestora, precum și o serie de exemple de bună utilizare, este realizată în § 3.2. Exemple de utilizare a funcțiilor încorporate.

Tabelul 2. *Lista funcțiilor încorporate în Python*

Funcțiile incorporate în Python			
A	**E**	**L**	**R**
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
anext()			round()
any()	**F**	**M**	
ascii()	filter()	map()	**S**
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	**G**	**N**	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	**H**	object()	
callable()	hasattr()	oct()	**T**
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		**P**	**V**
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	**Z**
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		**_**
	iter()		__import__()

3.1. Exemple de utilizare a cuvintelor cheie în cod

Cuvintele cheie în Python se pot împărți în funcție de rolul pe care acestea îl au în sintaxa generală a programului în **10** categorii, prezentate în paragrafele următoare. Scurte descrieri ale cuvintelor cheie se regăsesc în Tabelul 3. Deși nu toate dintre ele vor deservi scopului acestei lucrări, e necesar a fi centralizate pentru cei ce doresc aprofundarea domeniului programării calculatoarelor utilizând limbajul Python.

Tabelul 3. *Scurtă descriere a cuvintelor cheie din Python*

Keyword	Descrierea funcționalității
`and`	Operator logic care returnează `True` dacă ambii operanți sunt adevărați și `False` altfel
`as`	Creează un alias la importul unei librării
`assert`	Verifică corectitudinea logicii codului scris – la depanarea codului
`async`	Creează corutine asincrone
`await`	Întrerupe codul unei funcții bazate pe corutină pe acea linie până când corutina execută
`break`	Termină iterația curentă
`class`	Definește o clasă
`continue`	Sare la următoarea iterație într-un calcul
`def`	Definește o funcție
`del`	Șterge referința curentă la un obiect
`elif`	Declarație condițională - împreună cu `if` și `else`
`else`	Declarație condițională - împreună cu `if` și `elif`
`except`	Utilizat în cazul excepțiilor - indică ce cod se execută dacă excepția a apărut
`finally`	Utilizat în cazul excepțiilor - indică ce cod se execută indiferent dacă excepția a apărut
`for`	Utilizat pentru a crea o buclă `for`
`from`	Utilizat pentru a importa o parte specifică a unui modul
`global`	Declară o variabilă globală
`if`	Declarație condițională - împreună cu `elif` și `else`
`import`	Utilizat pentru a importa un modul
`in`	Utilizat pentru a verifica dacă un obiect se regăsește într-un iterabil
`is`	Utilizat pentru a verifica dacă două variabile sunt identice sau nu
`lambda`	Definește o funcție lambda (anonimă) - fără denumire, cu utilizare unică
`None`	Definește o valoare nulă sau mulțimea vidă
`nonlocal`	Declară o variabilă ne-locală
`not`	Operator logic care returnează `True` dacă operantul este fals și `False` dacă este adevărat
`or`	Operator logic care returnează `True` dacă măcar un operant este adevărat și `False` altfel
`pass`	Utilizat pentru a specifica faptul că declarația nu face nimic
`raise`	Utilizat în cazul excepțiilor - ridică o excepție
`return`	Returnează o valoare și iese din funcție
`try`	Utilizat în cazul excepțiilor - compune o declarație `try-except`
`while`	Utilizat pentru a crea o buclă `while`
`with`	Utilizat pentru a simplifica excepțiile
`yield`	Termină executarea unei funcții și returnează un generator
`False`	Valoarea booleană care indică fals
`True`	Valoarea booleană care indică adevăr

`kw.1.` Cuvinte cheie de valoare booleană: `True, False, None`

Sunt singurele cuvinte cheie care se scriu cu majusculă. `False` și `True` sunt cele două cuvinte cheie boolean care descriu starea de adevăr a unei expresii, variabile etc., în timp ce `None` (echivalentul `null` din alte limbaje de programare) reprezintă valoarea nulă sau mulțimea vidă. Acest tip de cuvinte cheie se poate atribui unei variabile și, în general, se utilizează în diverse expresii care sunt evaluate boolean: *dacă afară plouă, voi lua o umbrelă*. În figura următoare sunt prezentate două exemple în care sunt utilizate cele două cuvinte cheie. Codul este scris în Visual Studio Code și fișierul salvat sub denumirea *afara_ploua.py*. Inițial s-a creat variabila denumită `afara_ploua` și s-a inițializat cu valoarea booleană `True`. Inițializarea variabilelor cu valori (indiferent de tipul datelor), în Python, se realizează într-un singur pas spre deosebire de alte limbaje de programare care necesită întâi declararea tipului variabilei și apoi inițializarea acesteia. Acest lucru garantează o mai mare rapiditate în scrierea codului.

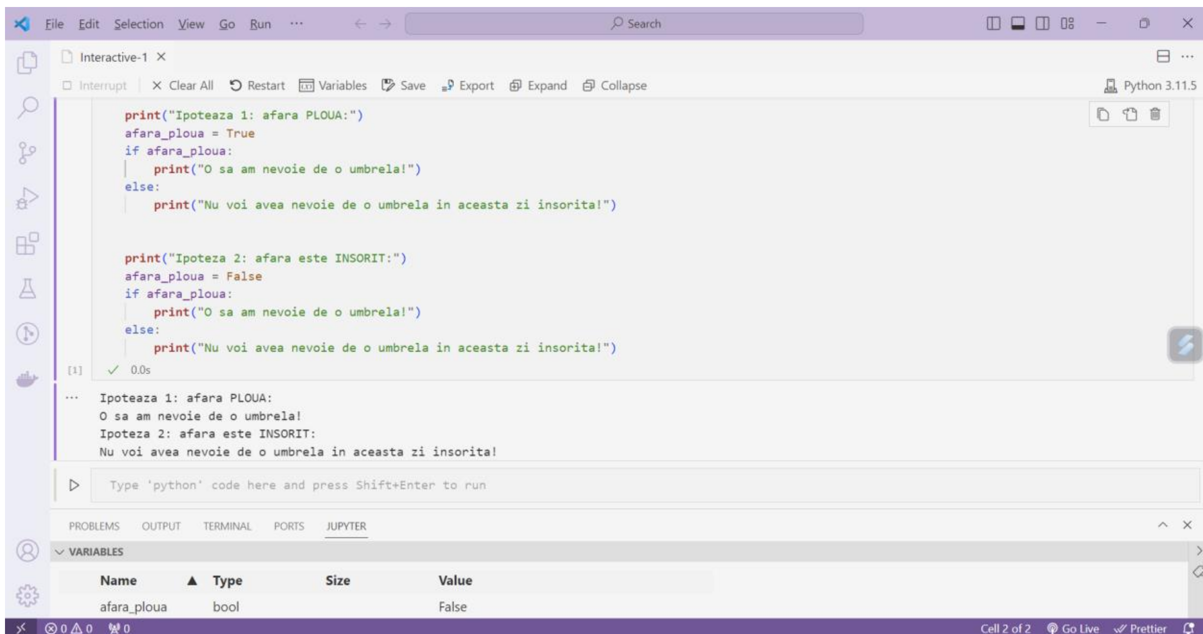


Figura 10. Utilizarea celor două cuvinte cheie de tip boolean în cadrul unei expresii logice

Pasul următor este de a folosi această variabilă într-o expresie logică: dacă afară plouă (`if afara_ploua`), textul `"O sa am nevoie de o umbrela"` va fi afișat pe consolă cu ajutorul funcției `print()`. În caz contrar (`else`), va fi afișat textul alternativ `"Nu voi avea nevoie de o umbrela in aceasta zi insorita!"`.

În primul exemplu, valoarea expresiei fiind adevărată, primul mesaj va fi afișat. În cel de-al doilea exemplu, valoarea atribuită aceleiași variabile este `False`, rezultând afișarea celui de-al doilea mesaj pe consolă. Adițional, în partea inferioară a capturii de ecran se pot analiza informații despre variabilă, precum nume, dimensiune, tip și ultima valoare care i-a fost atribuită (`False`, în acest caz).

Cuvântul cheie `None` simbolizează lipsa unei valori. Dacă o funcție nu returnează o valoare (nu se finalizează cu cuvântul cheie `return`), atunci ea va returna automat `None`.

`kw.2.` Cuvinte cheie operatori: `and`, `or`, `not`, `in`, `is`

Aceste cuvinte sunt operatori logici utilizați în diverse ramuri științifice și care vor fi evaluați cu o valoare booleană. `and` reprezintă **și logic**, `or` reprezintă **sau logic**, `not` reprezintă **negația logică**, `in` reprezintă apartenența – **în** și `is` reprezintă identitatea – **este**. În tabelul 4 sunt exemplificate rezultatele utilizării operatorilor logici `and`, `or` și `not`.

Tabel 4. Rezultatele operatorilor logici aplicați expresiilor în Python

`not` (negație)	expresie	`not` expresie	-
	False	True	
	True	False	
`and` (și logic)	expresie 1	expresie 2	expresie 1 `and` expresie 2
	False	False	False
	False	True	False
	True	False	False
	True	True	True

`or` (sau logic)	expresie 1	expresie 2	expresie 1 `or` expresie 2
	False	False	False
	False	True	True
	True	False	True
	True	True	True

Cuvântul cheie `in` verifică apartenența unui element la o mulțime. Date fiind un element de căutat și un element de tip container sau secvențial – care poate fi iterat și parcurs, cuvântul cheie `in` va returna `True` sau `False` în funcție de existența sau nu a elementului în container. Un exemplu bun este acela de a căuta o literă într-un cuvânt, o secvență de litere într-o propoziție sau un număr într-o listă. Sintaxa generală are forma generală:

```
<element> in <container>.
```

Cuvântul cheie `is` verifică identitatea dintre două obiecte. Diferă de operatorul `==` care verifică egalitatea între două tipuri de date Python. Acest lucru merită menționat pentru că, deși pot avea aceeași valoare (`<element 1> == <element 2>`), două variabile Python pot reprezenta două obiecte diferite în memoria internă (`<element 1> is not <element 2>`). `is` va returna `True` dacă `<element 1>` este exact același obiect în memorie ca `<element 2>`; va returna `False` altfel.

În continuare este prezentat, sub formă de cod, câte un exemplu de utilizare a fiecărui cuvânt cheie din această categorie. Exemplele se pot verifica într-un IDE la alegere.

```
# exemple de utilizare a cuvintelor cheie and, or, not, in, is
expresie_1 = True #creare variabila si initializata cu True
expresie_2 = False #creare variabila si initializata cu False
nume = "Inteligenta Artificiala in Ingineria Energetica" #creare variabila si initializata cu un sir de caractere

print(expresie_1 or expresie_2) #va returna True (sau logic)
print(expresie_1 and expresie_2) #va returna False (si logic)
print(not expresie_1) #va returna False (negatie logica)

print("Facultate" in nume) #va returna False
print("Inteligenta" in nume) #va returna True

nume_carte = nume # creare variabila si initializata cu alta variabila
print(nume is expresie_1) #va returna False
print(nume_carte is nume) #va returna True
```

kw.3. Cuvinte cheie de control al fluxului programului: `if`, `elif`, `else`

Aceste trei cuvinte cheie sunt utilizate pentru a controla modalitatea condiționată de a executa bucăți de cod. Ele urmează regula precedenței și interpretorul Python verifică starea de adevăr a fiecărei ramuri de cod și execută doar instrucțiunile ramurii care a rezultat în starea de `True`. Inițierea unui bloc de cod ce conține o declarație condiționată se realizează obligatoriu prin utilizarea cuvântului cheie `if`, celelalte două putând lipsi. Sintaxa unei declarații `if` începe utilizând acest cuvânt cheie urmat de o expresie care va fi evaluată de interpretorul Python (îi va analiza valoarea de adevăr) și restul instrucțiunilor se vor executa în funcție de rezultatul acestei evaluări. În pseudocod declarația `if` se poate scrie după cum urmează (de observat că după linia ce conține sintaxa `if`, a doua linie este indentată cu 4 spații sau Tab):

```
if <expresie1_adevarata>:
      <declarații de executat>
```

În cazul în care expresia din declarația `if` este evaluată ca fiind falsă, pentru a crea o alternativă, se poate folosi cuvântul cheie `else` care va comuta execuția blocului condiționat pe cea de-a doua ramură.

```
if <expresie1_adevarata>:
    <declarații de executat>
else:
    <declarații de executat>
```

Acest tip de declarație se poate prescurta folosind operatorul ternar din Python – o prescurtare a sintaxei care permite scrierea întregii declarații condiționate pe o singură linie și stă la baza unui mecanism specific Python cu care se pot crea diverse obiecte: **comprehensiune**.

```
<variabila> = <expresie 1> if <expresie 2> else <expresie 3>
```

Cuvântul cheie `elif` este specific limbajului de programare Python și permite crearea a oricât de multe ramuri de cod intermediare. Se folosește obligatoriu după linia ce conține `if` și înainte de linia ce conține `else`.

<table>
<tr>
<td>if <expresie 1>:
    <declarații de executat>
elif <expresie 2>:
    <declarații de executat>
elif <expresie 3>:
    <declarații de executat>
...
else:
    <declarații de executat></td>
<td>echivalent</td>
<td>if <a>:
    <execută cod 1>:
if (not a) and <b>:
   <execută cod 2>:
if (not <a>) and (not <b>) and <c>:
    <execută cod 3>:
...
if (not <a>) and (not <b>) and (not <c>)…:
   <execută cod 3></td>
</tr>
</table>

În continuare este prezentat un exemplu în care sunt folosite cele trei cuvinte cheie descrise teoretic anterior.

```
numar_1 = 10 #se creeaza o variabila si se initializeaza cu valoarea 10
numar_2 = 13 #se creeaza o variabila si se initializeaza cu valoarea 1

if numar_1 > numar_2:
    print("numarul 1 este mai mare decat numarul 2")
elif numar_1 == numar_2:
    print("numarul 1 este egal cu numarul 2")
else:
    print('numarul 1 este mai mic decat numarul 2')
```

OUTPUT

```
numarul 1 este mai mic decat numarul 2
```

kw.4. Cuvinte cheie iterative: `for`, `while`, `for`, `break`, `continue`, `else`

Buclele și iterațiile sunt concepte extrem de importante în programarea calculatoarelor, aceste proceduri făcând codul mai succint, mai rapid de scris și de rulat. Mai multe cuvinte cheie Python sunt folosite pentru a crea și a lucra cu bucle. Acestea, precum cuvintele cheie Python folosite pentru condiționalele anterioare, vor fi folosite și văzute în aproape fiecare program Python. Înțelegerea lor este fundamentală în orice domeniu de activitate care implică modelarea matematică, indiferent de complexitate.

`for` (pentru) este cea mai utilizată buclă iterativă din limbajele de programare și sintaxa completă implică și utilizarea cuvântului cheie `in` – detaliat anterior. Sintaxa iterativă utilizând bucla `for` parcurge fiecare element dintr-o mulțime și, de fiecare dată, execută o instrucțiune specificată. Astfel, se știe de la început numărul de instrucțiuni executate, bucla `for` fiind o buclă finită (numărul de elemente este cunoscut și este exclus riscul de apariție a iterațiilor infinite). În Python, un element care poate fi iterat (compus din mai multe elemente singulare) poartă denumirea re ***iterabil***. Iterabilele în Python sunt: șirurile de caracter (string-urile), listele, seturile, dicționarele și tuplurile. Toate acestea vor fi detaliate în subcapitolul următor, dedicat tipurilor de date Python. La modul general, sintaxa simplă a buclei `for` este următoarea:

```
for <element> in <container>:
    <instrucțiuni>
```

În continuare sunt prezentate câteva exemple de utilizare a buclei iterative `for`.

```
# se creeaza variabila lista_numere si este initilizata cu o lista formata din 3 elemente
lista_numere = [1, 2, 3]
# se itereaza prin lista de elemente utilizand bucla for
# se creeaza o variabila denumita numar si la fiecare iteratie se afizeaza pe ecran rezultatul incrementarii numarului cu 1
for numar in lista_numere:
    print(numar + 1)
```

OUTPUT

```
2
3
4
```

Merită menționat faptul că există posibilitatea ca variabila să nu mai fie inițializată anterior ci ca lista să fie direct declarată în corpul `for`: `for numar in [1, 2, 3]:`, rezultatul fiind identic. În exemplul următor se prezintă iterația utilizând bucla `for` printr-un șir de caractere pentru care se afișează pe ecranul consolei fiecare literă. Mai mult, în corpul buclei `for` se folosește și o expresie condiționată: dacă litera este „i“, o va afișa de 2 ori. Se folosește drept iterator variabila denumită `litera` (de observat dublarea semnului ”=„ sintaxa de verificare a unei egalități booleene în Python, spre deosebire de un singur semn ”=„ care are rolul de atribuire!).

```
for litera in "algoritmi":
    if litera == "i":
        print(2*litera, end="")
    else:
        print(litera, end="")
```

OUTPUT

```
algoriitmii
```

OBSERVAȚIE: Pentru a afișa rezultatul pe același rând (a se compara cu exemplul anterior!), în cadrul funcției `print()` trebuie adăugat un argument, `end`, căruia trebuie să i se dea o valoare – în cazul acesta i se atribuie valoarea de șir de caractere gol („”). Argumentul `end` se poate inițializa cu orice caracter sau șir de caractere recunoscute de sintaxa Python și valoarea implicită este caracterul linie nouă (newline – „`/n`”). În cazul în care s-ar opta pentru o altă valoare a argumentului `end`, spre exemplu caracterul ”-„ rezultatul ar fi de forma:

```
for litera in "algoritmi":
    if litera == "i":
        print(2*litera, end="-")
    else:
        print(litera, end="-")
```

OUTPUT

```
a-l-g-o-r-ii-t-m-ii-
```

Un alt exemplu este acela de a itera printr-o listă populată cu șiruri de caractere (de exemplu, nume de studenți) pentru care putem afișa pe ecran un mesaj cu funcția `print()`.

```
# se creeaza o lista populata cu 3 siruri de caractere - prenumele persoanelor
studenti = ["Bogdan", "Mihai", "Andra"]
# pentru fiecare student se afiseaza mesajul prenunume_student + un mesaj
for student in studenti:
    print(f"{student} este prezent la curs.")
```

OUTPUT

```
Bogdan este prezent la curs.
Mihai este prezent la curs.
Andra este prezent la curs.
```

`while` (în timp ce) este un alt cuvânt cheie în Python, cu ajutorul căruia se definesc operații iterative. Spre deosebire de bucla `for`, `while` se utilizează în cazul în care nu se știe exact numărul de iterații ce trebuie executate și va rula atât timp cât expresia de după cuvântul cheie execută în valoarea de adevăr `True`, apărând necesitatea folosirii unui mecanism de oprire a buclei (expresia să devină `False`); în caz contrar, aceasta rulând la infinit și producând o eroare. Sintaxa generară a buclei `while` este următoarea:

```
while <expresie_adevarata>:
    <instrucțiuni>
```

Un exemplu eronat de utilizare a buclei `while` este generarea unei iterații infinite, care, evident, va genera o eroare în terminal – asta în cazul în care execuția este oprită forțat.

```
# se creeaza o variabila care se initializeaza cu valoarea booleana True
sunt_la_facultate = True
# se itereaza utilizand bucla while
# atat timp cant variabila ramane True, instructiunile vor rula
while sunt_la_facultate:
    print("Frecventez cursurile")
# nu este specificata conditia de oprire a iteratiilor (sunt_la_facultate sa devina False)
```

OUTPUT

```
Frecventez cursurile
Frecventez cursurile
...
Frecventez cursurile
```

Pentru a evita apariția iterațiilor infinite, trebuie să existe un contor de control al execuției expresiei din cadrul sintaxei buclei `while`. Uzual, acest lucru se poate realiza cu ajutorul unei variabile suplimentare care are rolul de a contoriza de câte ori rulează bucla `while` și a opri iterațiile în cazul în care conturul indică numărul specificat sau o condiție terță este îndeplinită. Modificând exemplul de cod precedent, adăugăm o variabilă contor denumită `numar_cursuri` pe care o inițializăm cu valoarea numerică 4. La fiecare iterație din cadrul buclei `while`, variabila contor este modificată (se scade valoarea în acest caz) până când aceasta ajunge la valoarea dorită (în acest caz, până când nu mai sunt cursuri de frecventat `numar_cursuri = 0`). La fiecare iterație se verifică dacă este îndeplinită condiția booleană `numar_cursuri == 0` (de observat dublarea semnului ”=„ sintaxa de verificare a unei egalități booleene în Python, spre deosebire de un singur semn ”=„ care are rolul de atribuire valoare!) și în momentul în care

aceasta devine True, variabila sunt_la_facultate se reinițializează cu valoarea False și bucla while își termină execuția.

```
# se creeaza o variabila care se initializeaza cu valoarea booleana True
sunt_la_facultate = True
# se creeaza o variabila cu rolul de contor al buclei while si se initiali-
zeaza cu numarul 4
numar_cursuri = 4
# se itereaza utilizand bucla while
# atat timp cant variabila ramane True, instructiunile vor rula
while sunt_la_facultate:
    print("Frecventez cursul numarul", numar_cursuri, end=" - ")
    numar_cursuri -= 1
    print("Cursuri ramase: ", numar_cursuri)
    # daca numarul de cursuri a devenit 0, variabila sunt_la_facultate devine
Falsa si bucla while se opreste din executie
    if numar_cursuri == 0:
        sunt_la_facultate = False
        print("Pot pleca acasa!")
```

OUTPUT

```
Frecventez cursul numarul 4 - Cursuri ramase:  3
Frecventez cursul numarul 3 - Cursuri ramase:  2
Frecventez cursul numarul 2 - Cursuri ramase:  1
Frecventez cursul numarul 1 - Cursuri ramase:  0
Pot pleca acasa!
```

În exemplul anterior, funcția print()afișează atât un șir de caractere “Frecventez cursul numarul” cât și valoarea curentă a variabilei numar_cursuri. O a doua funcție print() este utilizată după de valoarea variabilei contor a fost modificată și va afișa acea valoare după șirul de caractere “Cursuri ramase: “. După ce expresia funcțională a buclei while devine False, se afișează pe ecranul terminalului mesajul “Pot pleca acasa!” .

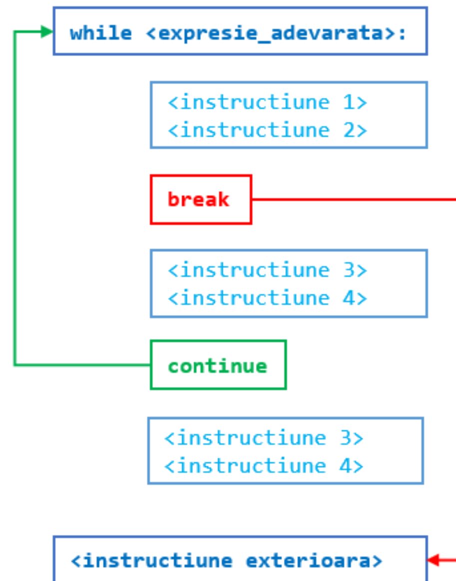
Există posibilitatea de a scrie setul de instrucțiuni care compun bucla while pe o singură linie, deși nu este nici indicat și nici nu se poate utiliza pentru cazuri cu complexitate crescută compuse din mai multe instrucțiuni condiționate, cum ar fi utilizarea instrucțiunii if.

```
# se creeaza variabila denumita numar si se instantiaza cu valoarea intreaga 3
numar = 3
# se scrie blocul iterativ while pe o singura linie
while numar >= 0: numar-=1; print(numar, end='. ')
```

OUTPUT

```
2. 1. 0. -1
```

break (întrerupe), împreună cu continue (continuă), este alternativa de a opri (sau a sări peste un set de instrucțiuni) execuția condiționată a codului, frecvent utilizată în programarea calculatoarelor. break este utilizată în cazul în care se dorește ieșirea din bucla iterativă înainte de îndeplinirea condiției de oprire, în timp ce continue se utilizează când se dorește omiterea iterației curent și executarea următoarei iterații. Ea precedă unei operații logice care are rolul de a verifica starea de adevăr a unei expresii. Merită menționat faptul că atât break cât și continue pot fi folosite în bucle for, dar și în bucle while. O exemplificare vizuală se poate vedea în imaginea alăturată.



Sintaxele generale ale celor două cuvinte cheie sunt:

for <element> in <container>:
    if <expresie_adevarata>:
        break
	for <element> in <container>:
	    if <expresie_adevarata>:
	        continue

Un exemplu de utilizare a cuvântului cheie break este acela de a crea un control asupra valorilor unor mărimi numerice – dacă valoarea trece de un prag impus de utilizator, blocul de cod iterativ ar trebui să își oprească execuția. În exemplul următor se creează o variabilă denumită lista_numere, pe care o inițializăm cu o listă ce conține numerele întregi de la 1 la 10, și o variabilă denumită suma_totala, inițializată cu valoarea 0 și care are rolul de a stoca suma numerelor până la iterația curentă (se actualizează la fiecare iterație). În momentul în care variabila suma_totala depășește valoarea 13 se execută break și se întrerupe bucla iterativă.

```
# se creeaza variabila lista_numere si se initializeaza cu o lista ce contine primele 10 numere naturale pozitive
lista_numere = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# variabila suma_total este initilizata cu valoarea 0 si are rolul de a stoca suma numerelor din list
suma_total = 0
# se itereaza cu o bucla for in lista cu numere
for numar in lista_numere:
    # de la o iteratie la alta variabila suma_total este updata, adaugand numarul de la iteratia curenta
    suma_total += numar
    # daca valoarea variabilei suma_total depaseste valoarea 13, iteratiile se termina
    if suma_total > 13:
        break
print(suma_total)
```

OUTPUT

```
15
```

În exemplul următor este prezentat un mod de a utiliza cuvântului cheie continue pentru a sări peste anumiți pași în blocul iterativ în funcție de valoarea de adevăr a expresiei din declarația if. Se creează o variabilă denumită nume pe care inițializăm cu șirul de caractere "Energeticieni". Se iterează prin șirul de caractere cu o buclă for și în cazul în care litera este "E"/"e" sau "i", iterația respectivă este omisă și se trece la următoarea. Instrucțiunea booleană if complexă returnează True dacă cel puțin una dintre cele două expresii ce o compun este True.

```
# se creeaza variabila nume si se initializeaza cu un sir de caractere
nume = "Energeticieni"
# se intereaza prin sirul de caractere folosid variabila litera
for litera in nume:
    # daca litera este E sau e sau daca litera este i, se omite iteratia
curenta
    if litera.lower() == "e" or litera == "i":
        continue
    # se afiseaza fiecare litera din cuvant, separate cu punct, mai putin cele
din iteratiile omiste
    print(litera, end='.')
```

OUTPUT

```
n.r.g.t.c.n.
```

OBSERVAȚIE: În exemplul anterior s-a folosit o metodă specifică tipului de date șir de caractere (sau string): lower(). Aceasta are rolul de a transforma orice literă majusculă în minusculă și este utilizată în **acest** caz pentru a asigura faptul că ambele cazuri posibile sunt luate în considerare și că atât majuscula E cât și minuscula e vor fi omise. Dacă se dorea ca doar minusculele să fie omise, declarația if se scria: if litera == „e" or litera == „i": (fără metoda lower() aplicată variabilei de iterare). Mai multe detalii sunt prezentate în capitolul următor, dedicat tipurilor de date din Python și funcțiilor/metodelor acestora.

În fragmentul următor de cod este exemplificată utilizarea ambelor cuvinte cheie detaliate anterior într-o buclă while. Se creează o variabilă cu valoarea numerică de tip număr întreg denumită numar_maxim și se inițializează cu valoarea 15. Bucla iterativă while va rula atât timp cât valoarea variabilei numar_maxim va fi mai mare ca 0. Pentru a evita apariția iterațiilor infinite, caz în care niciodată condiția de oprire a buclei while nu va fi îndeplinită – numar_maxim să fie mai mare sau egal cu 0 – la fiecare pas iterativ valoarea variabilei numar_maxim se va modifica (va descrește cu 1). În continuare se dorește a se afișa numere impare – restul împărțirii la 2 este diferit de 0. În limbajul de programare Python acest lucru se realizează prin utilizarea simbolului %. În cazul în care numărul de la iterația curentă este par se trece automat la următoarea iterație prin utilizarea cuvântului cheie continue. În momentul în care valoarea din iterația curentă a variabilei numar_maxim ajunge la valoarea 5, se întrerupe execuția blocului iterativ utilizând cuvântul cheie break și este afișat pe consolă mesajul din funcția print(). A se observa faptul că funcția finală print() este pe același nivel de indentare cu declararea varaibilei numar_maxim și începutul buclei while. Acest lucru înseamnă că, indiferent de modul de execuție a blocului iterativ, această linie va fi executată.

```
# se creeaza variabila si se initializeaza cu un numar int (15 in acest caz)
numar_maxim = 15
# se itereaza utilizand o bucla while care verifica la fiecare iteratie daca numar_maxim este mai mare ca 0
while numar_maxim > 0:
    # la fiecare iteratie, numar_maxim descreste cu 1
    # scrierea este echivalenta cu numar_maxim -= 1
    numar_maxim = numar_maxim -1
    # verifica daca restul impartirii valorii din variabila numar_maxim la 2 este egal cu 0
    # restul impartirii la 2 egal cu 0 inseamna un numar par
    if numar_maxim % 2 == 0:
        # daca numarul este par, sari peste acea iteratie - nu afisa numarul
        continue
    # daca numarul este impar, afiseaza numarul pe terminalul ecranului
    print(numar_maxim, end=', ')
    # daca numarul de la iteratia curenta, iesi din bucla while
    if numar_maxim == 5:
        break
# dupa finalizarea iteratiilor cu bucla while se afiseaza mesajul din sirul de caractere
print("Bucla if s-a terminat cu succes! Numerele 3 si 1 nu au fost afisate!")
```

OUTPUT

```
13, 11, 9, 7, 5, Bucla if s-a terminat cu succes! Numerele 3 si 1 nu au fost afisate!
```

`else` se poate folosi și în cazul sintaxelor iterative, și nu doar în expresii boolean. Această clauză opțională se poate utiliza la sfârșitul buclei `while`. Sintaxa generală este:

while <expresie_adevarata>: <instrucțiuni> **else**: <instrucțiuni_adiționale>	**while** <expresie_adevarata>: <instrucțiuni> <instrucțiuni_adiționale>

În acest caz, setul de `<instrucțiuni_adiționale>` va fi executat în momentul în care bucla iterativă `while` își va termina execuția. Acest mod de scriere a codului are același rezultat ca în cazul în care clauza `else` nu este utilizată și `<instrucțiuni_adiționale>` nu face parte din corpul buclei `while` (sintaxa de pe coloana din dreapta). Diferența între cele două modalități de scriere este însă fundamentală: `<instrucțiuni_adiționale>` din bucla `while` vor fi executate **doar** în momentul în care toate iterațiile s-au încheiat „prin epuizare“, adică bucla iterează până când condiția de control devine `True`. Dacă în schimb execuția buclei este oprită printr-o instrucțiune `break`, `<instrucțiuni_adiționale>` nu vor fi executate. În cel de-al doilea caz, fără utilizarea clauzei `else`, `<instrucțiuni_adiționale>` vor fi executate indiferent de modalitatea de oprire a blocului iterativ.

Cele două cazuri în antiteză sunt prezentate în exemplele următoare. Diferența este dată de valoarea pe care instrucțiunea condiționată `if` o folosește pentru a activa sau nu ieșirea din bucla iterativă utilizând `break` (valoarea pe care variabila `numar` o va lua la o anumită iterație).

În primul caz, instrucțiunea `if` nu va fi niciodată adevărată și, în consecință, bucla `while` își va finaliza execuția organic fiind afișat și mesajul din cadrul instrucțiunii `else`.

```
numar = 4
while numar > 0:
    numar -= 1
    print(numar, end=". ")
    if numar == 5:
        break
else:
    print('\nBucla s-a finalizat "prin epuizare"')
OUTPUT:
3. 2. 1. 0.
Bucla s-a finalizat "prin epuizare"
```

În cel de-al doilea caz, bucla `while` va fi întreruptă deoarece la a 4-a iterație variabila `numar` va lua valoarea 1 și instrucțiunea `break` va fi executată. Drept urmare, doar primele 3 numere vor fi afișate și instrucțiunea din cadrul clauzei `else` nu va fi executată.

```
numar = 4
while numar > 0:
    numar -= 1
    print(numar, end=". ")
    if numar == 1:
        break
else:
    print('\nBucla s-a finalizat "prin epuizare"')
OUTPUT:
3. 2. 1.
```

kw.5. Cuvinte cheie de structură: `def`, `class`, `with`, `as`, `pass`, `lambda`

În cazul programelor complexe în care apare necesitatea reutilizării anumitor bucăți de cod, acestea se pot scrie în interiorul unor funcții care vor fi executate ori de câte ori vor fi apelate.

`def` permite definirea **funcțiilor** în Python – denumite și **metode** dacă sunt create în interiorul unor clase și descriu funcționalitatea acestora. Pseudocodul sintaxelor de bază pentru definirea și apelarea funcțiilor este:

Definire funcție	**Apelare funcție**
def <nume_functie>(<parametri>): <corp_functie>	<nume_functie>(<argumente>)

În domeniul programării calculatoarelor, funcțiile reprezintă un fragment de cod autonom care încapsulează o sarcină bine definită și cu specificitate sporită. Odată definite, funcțiile se pot apela oriunde în corpul codului și execuția programului se va comuta automat către corpul funcției, interpretorul Python va executa funcția, va obține rezultatul și apoi va comuta din nou execuția pe linia de după apelarea funcției. Pentru a putea vizualiza acest lucru, este propus exemplul următor, în care se definește o funcție și se apelează după un anumit mesaj.

```
# se defineste o functie denumita functie_test()
def functie_test():
    # se creeaza o variabila si se initializeaza cu un sir de caractere
    mesaj = "Sunt in interiorul functiei denumte functie_test()"
    print(mesaj)

# inainte de executia functiei se afizeaza pe ecran un mesaj cu ajutorul
functiei print()
print("Inainte de apelarea functiei functie_test()")
#se apeleaza functia definita anterior
functie_test()
# dupa executia functiei se afizeaza pe ecran un mesaj cu ajutorul functiei
print()
print("Dupa apelarea functiei functie_test()")
OUTPUT:
Inainte de apelarea functiei functie_test()
Sunt in interiorul functiei denumte functie_test()
Dupa apelarea functiei functie_test()
```

Se definește o funcție denumită `functie_test()` pentru care nu se specifică niciun argument. Corpul acestei funcții conține o variabilă denumită `mesaj` inițializată cu șirul de caractere `"Sunt in interiorul functiei denumte functie_test()"` și afișează acest mesaj cu ajutorul funcției încorporate `print()`. Funcția nu returnează nimic. În afara corpului funcției se afișează șirul de caractere `"Inainte de apelarea functiei functie_test()"` - introdus ca argument al funcției `print()`, se apelează funcția `functie_test()` (se va executa partea de cod din funcție) care va afișa la rândul ei mesajul specificat, apoi se va reveni la linia de după apelare și se va executa funcția `print()` care va afișa un nou mesaj - `"Dupa apelarea functiei functie_test()"`.

Un exemplu de funcție cu parametri (modalitatea de a introduce date în corpul funcției) și returnarea unei valori este calculul fluxului termic unitar ce străbate un perete plan și omogen supus unei diferențe de temperatură. În acest exemplu, se propun următorii parametri: `temperatura_interioara`, `temperatura_exterioara`, `grosime`, `conductivitate`. Cu ajutorul acestor parametri, în corpul funcției se vor calcula următoarele variabile: `rezistenta_termica`, `diferenta_temperatura`, `flux` și funcția va returna valoarea ultimei variabile. Formulele utilizate au fost extrase din cartea *Bazele Transferului de Căldură și Masă* (Badea, Bazele Transferului de Căldură și Masă, 2005). Pentru calculul diferenței de temperatură s-a utilizat funcția înglobată `abs()` care returnează modulul unui număr. În continuare, funcția este apelată de două ori pentru două materiale diferite: un izolator și cărămidă uzuală, rezultatele fiind afișate pe ecran cu ajutorul funcției `print()`. Suplimentar, pentru a nu se crea confuzii în utilizarea funcției, se poate indica ce tip de dată reprezintă fiecare parametru. Acest lucru se realizează în momentul definirii acestora (cum este cazul parametrului `grosime: int`) din exemplul următor.

```
# se defineste funcia denumita flux_termic care are 4 parametri si returneaza
valoarea fluxului termic
def flux_termic(temperatura_interioara, temperatura_exterioara, grosime:
int,conductivitate):
    # se determina rezistenta termica - grosime impartia la conductivitate
    rezistenta_termica = grosime/conductivitate
    diferenta_temperatura = abs(temperatura_interioara -
temperatura_exterioara)
    # se calculeaza fluxul termic unitar de suprafata: diferenta de
temperatura impartita la rezistenta termica
    flux = diferenta_temperatura / rezistenta_termica
    # se foloseste cuvantul cheie return pentru a returna o valoare
    return flux

# exemplul 1 de calcul - materlial izolator
# rezultatul returnat de functie se va stoca intr-o variabila denumita
flux_termic_izolatie
flux_termic_izolatie = flux_termic(20, -15, 0.5, 0.15)
print(f"Valoarea flxului termic pentru materialul de izolatie este: {
flux_termic_izolatie} W/mp")

# exemplul 2 de calcul - material neizolator - caramida
# rezultatul returnat de functie se va stoca intr-o variabila denumita
flux_termic_caramida
flux_termic_caramida = flux_termic(20, -15, 0.5, 45)
print(f"Valoarea flxului termic pentru caramida este: { flux_termic_caramida}
W/mp")
OUTPUT:
Valoarea flxului termic pentru materialul de izolatie este: 10.5 W/mp
Valoarea flxului termic pentru caramida este: 3150.0 W/mp
```

OBSERVAȚIE. În Python rezultatul împărțirii a două numere (chiar dacă sunt numere întregi) este un număr zecimal (`float` – număr în virgulă mobilă).

La definirea funcției datele introduse se numesc **parametri** în timp ce la apelarea ei (în corpul programului principal) datele introduse se numesc **argumente**. Trebuie avut grijă ca numărul (și tipul!) de argumente introduse la apelare să coincidă cu numărul de parametri declarați la definirea funcției. În caz contrar, interpretorul Python va genera o eroare care va indica faptul că unul sau mai multe argumente poziționale lipsesc! Pentru a evita această posibilă sursă de erori în codul sursă există posibilitatea de a specifica numele argumentelor la apelarea funcției sub forma: `<argument>=<valoare>`, deși implică scrierea explicită a numelor parametrilor. În fragmentul de cod următor este prezentat un astfel de exemplu. Se introduc valori pentru argumente specificându-le numele la apelarea funcției.

```
flux_termic_bca = flux_termic(temperatura_exterioara=13,
                              temperatura_interioara=27,
                              grosime=0.75,
                              conductivitate=1.13)
```

Există inclusiv posibilitatea de a da valori parametrilor în momentul definirii funcției și aceștia se numesc **parametri impliciți**. În cazul optării pentru această variantă, toți parametrii trebuie să fie impliciți, apărând erori de sintaxă în caz contrar. Mai mult, argumentele de la apelarea funcției vor suprascrie aceste valori implicite, dar în cazul în care nu sunt specificate argumente, acestea vor lua valorile parametrilor impliciți.

```
def flux_termic(temperatura_interioara=20,
                temperatura_exterioara=-13,
                grosime=10 ,
                conductivitate=0.15):
    rezistenta_termica = grosime/conductivitate
    diferenta_temperatura = abs(temperatura_interioara-temperatura_exterioara)
    flux = diferenta_temperatura / rezistenta_termica
    return flux
```

`class` permite crearea claselor în Python, acesta fiind un limbaj de programare orientat spre obiecte (POO sau OOP – *Object Oriented Programming*), permițând crearea unor tipuri noi de date abstracte din care se pot crea obiecte specifice. O clasă în programare reprezintă o structură logică de date creată de utilizator cu ajutorul căruia se definește starea (atributele) și comportamentul (metodele) obiectelor create. Cu alte cuvinte, o clasă se poate viziona ca o matriță pentru crearea de obiecte, denumite și instanțe ale clasei. Fiecărui obiect creat dintr-o clasă îi sunt specificate particularitățile prin indicarea valorilor atributelor și metodelor generale. Un obiect Python încapsulează atât caracteristici ale acestor date cât și funcționalități care se pot realiza cu acestea. De fapt, în Python totul este creat sub forma unui obiect (chiar și tipurile de date de bază: numere, valorile boolean și șirurile de caractere).

Sintaxa generală pentru crearea unei clase este prezentată în continuare. Spre exemplu, clasa `Student` poate avea atribute precum `nume, an_studii, medie` etc. și metode precum `da_examen, frecventeaza_curs` etc. Altfel spus, cu ajutorul claselor se pot crea noi tipuri de date cu specificații și utilități complexe. Procesul de creare a instanțelor din clase (instanțele se mai numesc și obiecte – de aici și denumirea de POO) se numește **instanțiere.**

definire clasă	**creare instanță - instanțiere**
class <numeClasa>(<extinde>): <corpClasa>	nume_instanta = numeClasa()

Deși în această lucrare nu vor fi utilizate exhaustiv clasele, acestea prezentând un grad de abstracție ridicat, se va exemplifica modul de creare și instanțiere al unei clase, deoarece acestea fac parte din sintaxa Python și sunt folosite cu precădere în multe aplicații. În exemplul următor este realizată o clasă ce reprezintă șablonul general pentru crearea pereților unei clădiri și executarea unor operații simple din sfera transferului de căldură și masă. Ca atribute, această clasă va conține: `nume`, `grosime`, `conductivitate`, în timp ce metodele sunt `__init__()`, `calcul_rezistenta()`, `calcul_flux()`și `__str__()`. Mai multe detalii despre utilizarea avansată a claselor în Python se vor putea analiza în capitolul 5.

Pe prima linie se definește clasa utilizând cuvântul cheie `class` urmat de identificatorul dorit, astfel interpretorul Python știe că urmează a fi creat un nou tip de dată. Declararea atributelor

se realizează prin metoda specifică `__init__()` – initialize, care are rolul de **constructor** și inițializează de fapt instanțele nou create. Această metodă specială este apelată automat la inițializarea instanțelor. Nu există obligativitatea de a folosi această metodă, deoarece mecanismul intern Python are un constructor de instanțe implicit, apelat la simpla creare a clasei. Bunele practici indică faptul că totuși metoda constructor trebuie folosită și în mod obligatoriu aceasta are cel puțin un parametru `self`, cu rolul de a lega atributele de instanța curentă. În cazul prezentat, pe lângă parametrul `self` sunt adăugați alți trei: `grosime`, `conductivitate`, `nume` care vor trebui specificați la crearea obiectelor. Urmează declararea metodelor specifice pentru această clasă: `calcul_rezistenta()` și `calcul_flux()`, fiecare având cel puțin parametrul care le leagă de instanțele propriu-zise (`self`). Metodele realizează calcule simple utilizând în principal atributele obiectelor respective. După cum le sugerează denumirile, prima metodă va returna valoarea rezistenței termice a materialului respectiv și cea de-a doua va calcula, în funcție de alți doi parametri indicați doar în momentul utilizării metodei – `temp_int` și `temp_ext`, fluxul termic străbătut prin acest material. În final, ultima metodă utilizată `__str__()` (string – șir de caractere) este, la fel ca metoda `__init__()`, opțională și are rolul de a returna forma customizată a denumirii instanței. Metoda există în mod implicit și se creează în momentul definirii clasei, dar ea va returna denumirea standard a obiectelor sub forma: `<__main__.MatrialConstructie object at 0x000001AF3652C910>`. Dacă această metodă se customizează, la utilizarea funcției implicite `print(nume_obiect)` se va afișa textul dorit – spre exemplu: `Materialul denumit bca are conductivitatea termica 42 W/mp/K, grosimea 0.3 m. Rezistenta lui termica este 0.01 mp*K/W`. Diferența între cele două exemple este evidentă și accentuează utilitatea editării metodei `__str__()`.

După procesul de creare a clasei se pot genera instanțe ale acesteia prin apelarea numelui clasei pentru care se specifică atributele sub forma parametrilor: `bca = MatrialConstructie(0.3, 42, 'bca')`, unde `0.3` este valoarea pentru atributul `grosime`, `42` pentru `conductivitate` și șirul de caractere `'bca'` reprezintă valoarea pentru `nume`. Apelarea metodelor prin intermediul instanței se realizează utilizând sintaxa `nume_instanta.metoda(argumente)`, de exemplu: `rezistenta_termica = bca.calcul_rezistenta()`. În acest caz, valoarea returnată de metoda apelată este stocată într-o variabilă denumită `rezistenta_termica` pentru utilizări viitoare. Pentru utilizarea metodei `calcul_flux()` trebuie definiți doi parametri în prealabil, așa cum a fost ea declarată la definirea clasei.

```
# se defineste o clasa simpla denumite MaterialConstructie()
class MatrialConstructie():

    # se creeaza initializatorul clasei folosint metoda speciala __init__
    def __init__(self, grosime, conductivitate, nume):
        # se definesc cele 2 atribute necesare la instantierea obiectelor
        create din clasa
        # se utilizeaza cuvatul self inaitnea atributelor de clasa
        self.grosime = grosime
        self.conductivitate = conductivitate
        self.nume = nume

    # se defineste metoda calcul_rezistenta care va utiliza 2 atribute
```

```
    definite anterior si va returna o valoare
    def calcul_rezistenta(self):
        return round(self.grosime/self.conductivitate, 2)

    # se defineste metoda de calcul a fluxului termic unitar de suprafata
    # metoda aceasta foloseste doi parametri externi (temp_int, temp_ext) care
    trebuie definiti la folosirea metodei
    def calcul_flux(self, temp_int, temp_ext):
        rezistenta = self.calcul_rezistenta()
        return abs(temp_int - temp_ext)/rezistenta

    # metoda __str__ returneaza un text care poate contine orice informatie
    dorita a fi afisata in
    # momentul in care se foloseste functia print() pe o instanta creeata din
    clasa
    def __str__(self) -> str:
        name = f'Materialul denumit {self.nume} are conductivitatea termica
        {self.conductivitate} W/mp/K, grosimea {self.grosime} m.\n Rezistenta
        lui termica este {self.calcul_rezistenta()} mp*K/W'

        return name
# se instantiaza instanta denumita bca pe baza celor 3 atribute declarata in
metoda __init__
# grosime = 0.3, conductivitate = 0.42 si nume = 'bca'
bca = MatrialConstructie(0.3, 42, 'bca')
# se afiseaza mesajul din metoda __str__
print(bca)
# se apeleaza metoda calcul_rezistenta() si valoarea returnata se stocheaza in
variabila rezistenta_termica
rezistenta_termica = bca.calcul_rezistenta()
# se afiseaza valoarea rezistentei termice injectata intr-un mesaj
print(f'rezistenta termica are valoarea {rezistenta_termica} mp*K/W')
# se definesc si initializeaza doua variabile pentru a putea fi utilizate in
metoda calul_flux()
# valorile se pot introduce direct la apelarea metodei!
temperatura_exterioara = -15
temperatura_interioara = 20
# se apeleaza metoda flux_termic cu variabilele externe si rezultatul se
inmagazineaza in variabila flux_termic
flux_termic = bca.calcul_flux(temperatura_interioara, temperatura_exterioara)
# se afiseaza valoarea fluxului termic injectata intr-un mesaj
print(f'fluxul termic are valoarea {flux_termic} [W]')
OUTPUT:
Materialul denumit bca are conductivitatea termica 42 W/mp/K, grosimea 0.3 m.
Rezistenta lui termica este 0.01 mp*K/W
rezistenta termica are valoarea 0.01 mp*K/W
fluxul termic are valoarea 3500.0 [W]
```

with se folosește, împreună cu as, în cazul managerilor de context. Aceștia reprezintă bucăți de cod integrate în limbajul de programare Python care asigură buna funcționalitate a atribuirii și eliberării resurselor de memorie în anumite aplicații. O utilizare destul de des întâlnită a unui manager de context este reprezentată de lucrul cu fișiere externe (cum ar fi un fișier text sau Excel) care trebuie deschis, modificat și finalmente închis. Utilizarea managerului de context în acest caz asigură închiderea automată a fișierului după ieșirea din blocul de cod with. Sintaxa generală are forma, menționând faptul că utilizarea alias-ului as este opțională:

```
with expresie [as variabila]:
    <corp bloc with>
```

În exemplul următor se presupune că există în locația de lucru curentă de pe hard disk-ul calculatorului utilizat un fișier text denumit date.txt. Acesta se va deschide cu funcția implicită open() în managerul de context generat de utilizarea cuvântului cheie with și se va redenumi cu un alias (as fisier). Dacă se deschide în modul scriere ('w'), în cazul în care acesta nu există în locația curentă, va fi creat automat. După aceste instrucțiuni, se va scrie un text cu ajutorul metodei .write(), care va adăuga valoarea argumentului pe o nouă linie în fișierul folosit.

```
# se defineste o variabila si se initializeaza cu numele fisierului
file_name = 'date.txt'
# se utilizeaza managerul de context pentru a deschide fisierul si de a-i da
un alias
# fisierul se deschide in modul scriere - write (w)
with open(file_name, 'w') as fisier:
    # se scrie un text in fisierul deschis
    fisier.write("Inteligenta Artificiala in Inginerie Energetica")
# dupa iesirea din blocul with, fisierul va fi inchis
```

O alternativă la a folosi managementul de context al blocului with este prezentată în bucata de cod următoare, unde fiecare instrucțiune este executată separat, dezavantajul fiind în cazul în care apare o eroare la rulare: fișierele se pot corupe. Pentru a închide fișierul trebuie utilizată metoda implicită .close(), omiterea acesteia putând duce, de asemenea, la coruperea fișierului utilizat.

```
# se defineste o variabila si se initializeaza cu numele fisierului
file_name = 'date.txt'
# se deschide fisierul in modul scriere – write (w)
fisier = open(file_name,'w')
# se scrie textul in fisier
fisier.write("Inteligenta Artificiala in Inginerie Energetica")
# se inchide explicit fisierul utilizand metoda close()
fisier.close()
```

pass se utilizează în momentul în care un bloc de cod Python este lăsat intenționat necompletat, programatorul urmând să revină asupra lui. De exemplu, dacă se dorește a se declara o funcție, dar parametrii utilizați încă nu sunt total cunoscuți, se poate scrie header-ul funcției, conținând cuvântul cheie def, identificatorul (numele) și opțional parametrii, în timp ce corpul funcției va fi înlocuit de cuvântul cheie pass. Acest cuvânt cheie se poate folosi inclusiv în blocuri iterative și definirea claselor.

```
# se defineste o functie pentru care nu se specifica instructiunile
def functia_mea():
    pass
# se defineste o clasa
class ClasaMea():
    pass
# se utilizeaza o instructine conditionata if
# pass se poate folosi si pe aceeasi linie cu instructiunea principala
if 'student' == True: pass
```

`lambda` permite definirea unui tip special de funcție: funcții anonime (sau funcții lambda) – funcții care nu au un nume și care au o utilizare specifică și unică. Mai mult, aceste funcții nu au decât o singură instrucțiune. Sintaxa generală a unei funcții lambda este:

```
lambda <argument(e)>: instructiune
```

A se observa modul atipic de a scrie acest tip de funcție: fără o denumire concretă și fără un bloc de cod indentat – totul se scrie pe o singură linie. În fragmentul de cod următor este prezentat un exemplu de utilizare a funcției `lambda` pentru a genera pătratul unui număr și două posibilități de utilizare. Prima variantă este cea în care rezultatul funcției anonime este păstrat într-o variabilă care apoi va fi folosită precum o funcție – se apelează cu un argument. Cea de-a doua metodă de utilizare (cel mai des întâlnită în practică) este de a folosi rezultatul funcției `lambda` drept argument al unei alte funcții – funcția `print()` în acest exemplu.

```
# exemplul 1 de utilizare
# se defineste functia lamda care va ridica la patrat argumentul
# valoarea returnata va fi stocata intr-o variabila
valoare = lambda x: x**2valoare = lambda x: x**2
# se afiseaza pe ecran rezultatul functiei cu argumentul 3
print(valoare (3))
################################################################################
# exemplul 2 de utilizare
print((lambda x: x**2)(3))
OUTPUT:
9
9
```

`kw.6.` Cuvinte cheie pentru returnarea valorilor: `return`, `yield`

`return` este folosit în strânsă legătură cu declararea funcțiilor care returnează o valoare așa cum a fost descris în exemplele aferente cuvântului cheie `def` și în cazul declarării metodelor claselor. Acest lucru face ca utilizarea cuvântului cheie `return` să se facă doar în cazul definirii unei funcții și, în momentul în care acesta este executat de interpretorul Python, se iese din funcție și execuția revine din locul de după apelarea funcției. În cazul în care `return` este utilizat într-o instrucțiune condiționată (`if`, `elif`, `else`), pot exista mai multe astfel de cuvinte, cu mențiunea că doar condiția care va returna `True` va fi executată. În exemplul următor se verifică dacă un număr este par sau impar și va returna un mesaj în consecință.

```
# se defineste o functie cu un parametru numar intreg
# functia va returna un sir de caractere care va spune daca numarul este par
sau impar
def par_impar(numar:int)-> str:
    # daca rezultatul impartirii numarului la 2 este 0, rezultatul este par
    if numar % 2 == 0:
        # functia va returna faptul ca este par si va iesi din executie
        return f'{numar} este par'
    # daca nu este par, nu se executa return, deci se trece peste acea linie
de cod si se va executa automat urmatoarea
    return  f'{numar} este impar'
print(par_impar(2))
print(par_impar(13))
OUTPUT:
2 este par
13 este impar
```

Un exemplu asemănător este crearea unei funcții pentru generarea factorialului unui număr indicat ca parametru al funcției. În cazul în care numărul este 1, funcția returnează valoarea și iese din funcție. De asemenea, funcția va returna și valoarea numărului însuși înmulțit cu valoarea factorialului numărului anterior, devenind o funcție recursivă (se apelează pe ea însăși).

```
def factorial(numar):
    if numar == 1:
        return 1
    else:
        return numar * factorial(numar-1)

print(factorial(5))
OUTPUT:
120
```

`yield` este foarte similar cu `return` fiind de asemenea folosit pentru valorile returnate de o funcție. Cu toate acestea, când se folosește `yield`, aceasta va returna un **generator**, utilizabil în anumite funcții încorporate în sintaxa Python pentru a obține doar următoarea valoare. Astfel de funcții se numesc **funcții generator**. Utilitatea lor se regăsește în cazurile în care numărul de date cu care lucrează un program este foarte mare și parcurgerea întregului set devine ineficientă din punctul de vedere al resurselor computaționale – cum este cazul lucrului cu baze de date foarte mari. O altă diferență majoră este că, spre deosebire de `return`, execuția funcției nu se oprește după `yield`. În exemplul de cod următor se creează o secvență infinită de pătrate ale numerelor și se vor afișa pe rând cu funcția `next()`. De fiecare dată când se folosește această funcție, va genera doar valoarea următoare, dispărând problema iterațiilor infinite.

```
# se creeaza o functie generator fara niciun argument
# aceasta va genera o secventa infinita de patrate ale numerelor naturala
def patrate_infinite():
    # variabila numar se initializeaza cu valoarea 0
    numar = 0
    # se creaza o bucla infinita while
    while True:
        # se genereaza patratul numarului curent
        yield numar**2
        # se incrementeaza numarul cu 1
        numar += 1
```

```
# se apereaza functia generator si se salveaza intr-o variabila
numere = patrate_infinite()
# se genereaza prima valoare - 0^2 = 0
print("numarul 0 la patrat: ", next(numere))
# se genereaza URMATOARELE 5 patrate ale numerelor:
# cum deja s-a generat anterior o valoare, acum se vor genera urmatoarele 4
valori
for numar in range(5):
    print(f'numarul {numar+1} la patrat:', next(numere))
OUTPUT:
numarul 0 la patrat: 0
numarul 1 la patrat: 1
numarul 2 la patrat: 4
numarul 3 la patrat: 9
numarul 4 la patrat: 16
numarul 5 la patrat: 25
```

Mai mult, în exemplul anterior se folosește și o funcție încorporată în Python - `range()`, care este tot o funcție generator și care returnează un iterabil format din numere consecutive. Argumentul funcției `range()`, în acest caz, induce numărul de elemente ale secvenței de numere generate – 5! Totuși, trebuie reținut că în Python indexarea mărimilor secvențiale se face din 0 (și nu din 1 ca în alte limbaje de programare), drept urmare `range(5)` va genera următoarea secvență de numere: `0, 1, 2, 3, 4`! De aceea, pentru a afișa în șirul de caractere din funcția print valoarea curentă, se adaugă 1 la numărul generat: `'numarul {numar+1} la patrat:'`. Detalii se vor prezenta în capitolul în care se descriu tipurile de date primitive încorporate în sintaxa Python și funcțiile asociate lor – capitolul 4. Tipuri de date principale și metodele asociate lor

`kw.7.` Cuvinte cheie de import: `import`, `from`, `as`

În majoritatea cazurilor în care se dezvoltă o aplicație, mai ales cele bazate pe știința datelor sau pe elemente de inteligență artificială, există necesitatea de a utiliza librării externe care pot fi instalate prin intermediul ecosistemului Python. Acest lucru este extrem de benefic, deoarece permite reutilizarea codului scris de alte persoane sau entități, permițând totodată particularizarea acestuia în funcție de nevoile noilor aplicații. Librăriile (sau modulele) sunt în

esență colecții de funcții și/sau clase dezvoltate de programatori sau grupuri de programatori și care deservesc un scop specific – creare de site-uri web, știința datelor, AI etc.

Pentru a utiliza aceste funcționalități externe, orice limbaj de programare oferă prin intermediul propriei sintaxe posibilități de a realiza aceste importuri de cod din surse interne sau externe. În Python, importul unor astfel de librării se realizează utilizând cuvântul cheie `import`. Spre exemplu, pentru a putea lucra cu date tabulare sau cu date numerice complexe, trebuie utilizate librăriile numite **pandas** și **numpy**.

Există o multitudine de module externe ce se pot utiliza în aplicații specifice – unele instalate odată cu Python și altele externe, și care trebuie instalate în prealabil de a fi utilizate. Modulele existente se pot studia utilizând URL-ul, în timp ce pentru librăriile externe există Python Package Index (PyPI), un depozit de software pentru limbajul de programare Python, disponibil aici. Librăriile din a doua categorie trebuie inițial instalate în mediul virtual creat sau în ecosistemul deja existent al limbajului Python. Presupunând că modulul sau librăria este deja existent/ă sau a fost deja instalat/ă, sintaxa generală pentru a importa este:

```
import <modul>
```

Importul unei librării poate avea loc oriunde în codul sursă, dar este indicat ca acest lucru să se realizeze la început, din primul rând. În cazul în care sunt necesare mai multe librării, acestea vor fi importate separat, pe rânduri consecutive. Această abordare asigură evitarea unor erori la execuție în cazul în care se încearcă utilizarea librăriilor înaintea importării acestora. Un exemplu de import și utilizare al unei librării încorporate este prezentat în codul următor. Se importă două module denumite `cmath` și `math`, primul destinat calculului cu numere complexe și al doilea pentru calculul geometric și trigonometric, ambele foarte des utilizate în ingineria energetică. În cazul în care se doresc informații despre funcțiile și/sau clasele existente, se poate utiliza funcția încorporată `help()`, adăugând ca argument numele modulului. Importul celor două librării permite accesul la toate constantele și funcțiile încorporate, astfel că se pot utiliza valorile constantelor matematice `pi` și `e` deja integrate în `cmath` (pe lângă multe altele). `math` este un alt set de funcții, printre care și funcția `factorial()`, care se apelează ca o metodă: `math.factorial()`. În cazul în care se dorește utilizarea acestor elemente fără ca modulele să fie importate, interpretorul Python va genera o eroare, neputând recunoaște acele denumiri. Încercând, spre exemplu, utilizarea constantei `pi` pentru efectuarea unui calcul, va fi generată următoarea eroare: `NameError: name 'pi' is not defined`.

```
# se importa cele doua module.
# importul poate avea loc si pe o singura linie: import cmath, math
import cmath
import math
```

```

# se afiseaza pe ecran valoarea constantei matematice pi.
print('valoarea trunchiata a lui pi este: ', cmath.pi)
# se afiseaza pe ecran valoarea constantei matematice e.
print('valoarea trunchiata a lui e este: ', cmath.e)
# se afiseaza pe ecran valoarea lui 5!, folosind o funcitie din modulul math.
print('5! este: ',math.factorial(5))
```

```
OUTPUT:
valoarea trunchiata a lui pi este:  3.141592653589793
valoarea trunchiata a lui e este:  2.718281828459045
5! este:  120
```

În cazul în care se dorește doar o funcție sau clasă specifică dintr-o librărie (de cele mai multe ori din considerente de memorie utilizată în cazul librăriilor foarte mari – cum este cazul librăriilor de machine learning și inteligență artificială), se utilizează sintaxa compusă din:

```
from <modul> import <functie/clasa>
```

Spre exemplu, se dorește importul funcției `acos` (arccosinus) și constantei `pi` din modulul `math` și se dorește a se afla dacă este adevărată relația `acos(0)` este egal cu `pi/2`.

```
# se importa functia acos si constante pi din modulul math
from math import acos, pi
# se afiseaza valoarea de adevar a instructiei logice
# se verifica daca expresia este adevarata sau falsa
print(acos(0) == pi/2)
OUTPUT:
True
```

O ultimă abordare în importul librăriilor externe este acela de a le utiliza sub forma unui alias, schimbând denumirea librăriei specifice la utilizarea în programul principal. Deși nu aduce nicio schimbare în rezultate, este o bună practică folosită de programatorii Python și utilizarea unor librării cu ajutorul alias-urilor a devenit un standard. De exemplu, utilizarea unei librării speciale de machine learning, denumită tensorflow, se utilizează în cadrul programelor drept `tf`. Sintaxele generale de a importa o librărie sau o anumită funcție/clasă cu un alias sunt:

```
import <modul> as <alias>
from <modul> import <functie/clasa> as <alias>
```

```
# se importa libraria tensorlow cu alias-ul tf
import tensorflow as tf
# se importa libraria numpy cu alias-ul np
import numpy as np
# se importa libraria pandas cu alias-ul pd
import pandas as pd
```

Cu această abordare, metodele se vor apela utilizând alias-ul specificat, așa cum este prezentat în exemplul următor în care se folosește clasa `Fraction` din modulul `fractions` apelând alias-ul `Fr` (deși nu este un caz des întâlnit în practică). Librăria încorporată `fractions` oferă ustensilele necesare pentru a lucra cu numerele raționale și are o suită de clase pentru aceasta, cea mai utilizată fiind clasa de formare a numerelor raționale. A se observa modul în care este afișat rezultaul în acest caz: sub forma unei fracții și nu sub forma unui număr float!

```
# se importa clasa Fraction din modulul fractions cu alias-ul Fr
from fractions import Fraction as Fr
# se alias-ul clasei pentru a genera doua numere raionale
numar_rational_1 = Fr(1, -8)
numar_rational_2 = Fr(1, 3)
# se afiseaza rezultatul adunarii
print(numar_rational_1 + numar_rational_2)
OUTPUT:
5/24
```

`kw.8.` Cuvinte cheie pentru tratarea excepțiilor: `try`, `except`, `rise`, `finally`, `else`, `assert`

Unul dintre aspectele importante în programare este găsirea posibilelor cauze ale erorilor indiferent de natura acestora (de sintaxă sau de logică). Erorile care apar la rularea codului Python, după ce interpretorul a verificat sintaxa, se numesc erori de logică sau excepții. Se ia cazul în care se scrie o funcție simplă care returnează rezultatul inversului unui număr pe care utilizatorul îl introduce drept argument. Dacă acel număr este 0, atunci împărțirea nu are sens, rezultând o eroare de logică în programare – o excepție. Există o multitudine de excepții pentru care Python oferă indicații sub forma unor obiecte de excepții, spre exemplu:
- `FileNotFoundError` – încearcă să deschidă un fișier inexistent
- `ZeroDivisionError` – încearcă să împartă la 0
- `ImportError` – încearcă să importe o librărie care nu există sau nu e instalată

Toate excepțiile se pot afișa pe ecran utilizând funcția: `dir(locals()['__builtins__']`, ca argument al funcției încorporate `print()` sau citind documentația de pe site-ul oficial: URL.

`try - except`. Pentru a evita aceste situații, sintaxa Python conține cuvinte cheie pentru a crea blocuri de verificare a excepțiilor – blocurile `try - except`. Blocul de gestionare a excepțiilor începe cu `try`, și pe linia următoare, indentat, se scrie bucata de cod care ar putea ridica acea excepție (eroare). După aceasta, se folosesc alte cuvinte cheie asociate cu blocul de gestiune a excepțiilor și care definesc ce se întâmplă în cazul în care acestea apar. Aceste cuvinte sunt `except`, `else` și `finally`. Sintaxa generală a blocului de excepții este prezentată în fragmentul de cod următor:

```
try:
    <instructiuni care pot genera exceptii>
<except|else|finally>:
    <instructiuni pentru tratarea exceptiei>
```

Un astfel de bloc de cod este invalid dacă după instrucțiunile din segmentul `try` nu există cel puțin un alt segment format din unul din cuvintele cheie prezentate anterior, adică dacă nu există cel puțin un segment de cod `except`, `else` sau `finally`. În fragmentul de cod următor este prezentată, din nou, o funcție care calculează rezistența termică a unui perete format dintr-un singur strat și care are doi parametri: grosimea și conductivitatea termică, returnând raportul dintre cele două mărimi. Acest lucru poate genera o excepție în cazul în care se împarte la 0, drept urmare se utilizează un bloc de cod `try - except`.

```
# se defineteste o functie care primeste 2 parametri si returneaza rezultatul impartirii lor
def calcul_rezistenta_perete(grosime: int, conductivitate:int)->float:
    # se scrie in blocul try instructiunea care poate genera exceptia
    try:
        # se calculeaza rezistenta termica
        rezistenta = grosime/conductivitate
    # in blocul except se indica ce tip de exceptie ar putea aparea
    except ZeroDivisionError:
        # daca apare acea exceptie, rezistenta va fi None
        rezistenta = None
    # se va returna valoarea rezistentetei termice
    return rezistenta

# se apeleaza functia si ca argument pentru conductivitatea se trece valoarea 0
rezistenta_caramida = calcul_rezistenta_perete(0.5, 0)
# se afiseaza valoarea, rezultatul fiind None
print(rezistenta_caramida)
rezistnta_caramida2 = calcul_rezistenta_perete(0.5, 0.2)
print(rezistnta_caramida2)
# se afiseaza pe ecran exceptia ridicata prin impartirea la 0
print(0.5/0)
OUTPUT:
```

```
None
2.5
---------------------------------------------------------------------------
ZeroDivisionError                Traceback (most recent call last)
in line 393
   391 rezistnta_caramida2 = calcul_rezistenta_perete(0.5, 0.2)
   392 print(rezistnta_caramida2)
--> 393 print(0.5/0)

ZeroDivisionError: float division by zero
```

Nu este dificil de a identifica o altă posibilă eroare în codul funcției prezentate anterior: ce se întâmplă în cazul în care utilizatorul introduce alt tip de date în loc de date numerice, precum un text sub forma unui caracter sau a unui șir de caractere? În acest caz, calculul matematic nu se poate realiza și la execuția codului interpretorul Python va genera o eroare de tip `TypeError`, care specifică exact această speță: `unsupported operand type(s) for /: 'str' and 'str'` – nu se poate realiza împărțirea a două șiruri de caractere (`str` în Python). Așadar, merită menționat faptul că pot exista oricât de multe blocuri `except`, existând inclusiv opțiunea de a nu specifica obiectul excepției pe care vrem să o tratăm, caz în care vor fi tratate toate excepțiile existente în sintaxa Python.

```
try:
    <instructiuni care pot genera exceptii>
except Exceptia1:
    <instructiuni pentru tratarea Exceptia1>
except Exceptia2 as e:
    <instructiuni pentru tratarea Exceptia2 cu alias-ul e>
```

```
except (Exceptia3, Exceptia4):
    <instructiuni pentru tratarea Exceptia3 și Exceptia4>
except:
    <instructiuni pentru tratarea tuturor exceptiilor>
```

rise. Putem modifica funcția astfel încât să poată conține și această verificare de excepție, utilizând totodată și cuvântul cheie rise, folosit special pentru a ridica (genera) automat anumite posibile excepții. Important de reținut este faptul că utilizarea cuvântului cheie rise produce o terminare bruscă a execuției codului în momentul în care apare excepția, comparativ cu cazul nefolosirii lui. Acest lucru înseamnă că nu se vor mai executa instrucțiunile din programul principal – din afara blocului try - except.

```
# se defineteste o functie care primeste 2 parametri si returneaza rezultatul
impartirii lor
def calcul_rezistenta_perete(grosime: int, conductivitate:int)->float:
    # se scrie in blocul try instructiunea care poate genera exceptia
    try:
        # se calculeaza rezistenta termica
        rezistenta = grosime/conductivitate
    # in blocul except se indica ce tip de exceptie ar putea aparea
    except ZeroDivisionError:
        # daca apare acea exceptie, rezistenta va fi None
        rezistenta = None

    except TypeError as exceptie:
        print("Argumentele functiei trebuie sa fie numere nu sir de
caractere")
        rezistenta = None
        raise exceptie
    # se va returna valoarea rezistentetei termice
    return rezistenta

# se apeleaza functia; ca argument pentru conductivitatea se trece 0
rezistenta_caramida = calcul_rezistenta_perete(0.5, 0)
# se afiseaza valoarea, rezultatul fiind None
print(rezistenta_caramida)
# se apeleaza functia; ca argument pentru conductivitatea se va trece un text
rezistenta_caramida2 = calcul_rezistenta_perete(0.5, "0.3")
# apelarea functiei va genera exceptia TypeError
# executia se opreste si se ofera indicatii despre eroare
print(rezistenta_caramida2)
# codul nu va fi executat daca se foloseste rise in exceptie
rezistenta_caramida3 = calcul_rezistenta_perete(0.5, 0.3)
print(rezistenta_caramida3)
OUTPUT:
None
Argumentele functiei trebuie sa fie numere nu sir de caractere
---------------------------------------------------------------------------
[…]
TypeError: unsupported operand type(s) for /: 'float' and 'str'
```

Este indicat ca fiecare excepție să fie cuprinsă în propriul bloc `try - except`. Excepțiile se pot scrie ca alias în blocul de cod `except`. În fragmentul de cod următor este prezentat un bloc de excepții imbricat, în care fiecare posibilă excepție este tratată separat într-un bloc propriu. Trebuie menționat faptul că doar prima excepție apărută este afișată – dacă s-ar fi împărțit la 0, acea excepție era afișată, apoi se ieșea din codul de excepții fără a influența codul exterior blocului – ultima instrucțiune va fi executată indiferent de existența sau nu a excepției/excepțiilor. În cazul prezentat cea de-a doua excepție va fi generată, fișierul neexistând în directorul curent de lucru.

```
try:
    rezultat = 3/1
    try:
        fisier = open("fisier.txt", 'r')
    except FileNotFoundError as e:
        print(f'Fisierul nu exista, exceptia {e}')
except ArithmeticError as e:
    print(f'Impartirea la o este invalida, exceptia {e}')
except:
    print('A aparut o eroare necunoscuta')
print("Codul se executa indiferent daca a aparut sau nu o eroare")
OUTPUT:
Fisierul nu exista, exceptia [Errno 2] No such file or directory: 'fisier.txt'
Codul se executa indiferent daca a aparut sau nu o eroare
```

`finally` se folosește pentru a indica fragmentul de cod care se va executa indiferent dacă o excepție a fost sau nu ridicată în blocurile `try`, `except` sau `else`. Acest cuvânt cheie se utilizează ca ultim element într-un bloc de gestiune al excepțiilor și sintaxa generală este:

```
try:
    <instructiuni care pot genera exceptii>
except Exceptia1:
    <instructiuni pentru tratarea Exceptia1>
except Exceptia2 as e:
    <instructiuni pentru tratarea Exceptia2 cu alias-ul e>
finally:
    <instructiuni executate indiferent de try-except>
```

În fragmentul de cod următor, funcția de calcul pentru rezistențele termice ale pereților a fost îmbunătățită, folosindu-se și un bloc de cod `finally`, care va asigura execuția liniilor de cod constituente. Pentru a exemplifica acest lucru, se propun trei cazuri: primul caz care va ridica o excepție prin împărțirea la 0, al doilea caz în care se folosesc șiruri de caractere în locul valorilor numerice și un al treilea caz în care funcția este utilizată corespunzător. În toate cele trei cazuri funcția este executată fără a ieși din program și, mai mult, instrucțiunea din blocul `finally` este executată și mesajul este afișat în consolă. Se pot introduce orice fel de instrucțiuni în blocul `finally`, inclusiv tipul de date utilizate ca argumente ale funcției. Acest lucru se poate realiza foarte simplu prin folosirea funcției încorporate `type()`, care va returna tipul clasei din care obiectul variabilei trecute ca argument face parte. `type(grosime)` va căuta variabila `grosime`, va analiza ce tip de dată este și va returna răspunsul. În cazul în care variabila `grosime`

va fi de tipul `'str'`, funcția `type()` va returna: `<class 'str'>`. Pentru a vedea aceste lucruri ca output al funcției dezvoltate, în exemplul de mai jos se va face următoarea modificare în blocul `finally`: se va adăuga următoarea linie de cod (după linia funcției `print()`): `print(f'S-au introdus urmatoarele tipuri de date: {type(grosime)} pentru grosime si {type(conductivitate)} pentru conductivitate.')`. În funcție de tipurile de date folosite la apelarea funcției, unul dintre rezultate va fi: `S-au introdus urmatoarele tipuri de date: <class 'float'> pentru grosime si <class 'str'> pentru conductivitate.`

```
OUTPUT:
Aceast cod se executa indiferent de ce s-a petrecut anterior!
Valoarea rezistente este: None
-----------------------------------------------------------------
Argumentele functiei trebuie sa fie numere nu sir de caracteret
Aceast cod se executa indiferent de ce s-a petrecut anterior!
Valoarea resistentei este: None
-----------------------------------------------------------------
Aceast cod se executa indiferent de ce s-a petrecut anterior!
Valoarea rezistentei este 1.6666666666666667
-----------------------------------------------------------------
```

`else` a fost prezentat și în utilizarea instrucțiunilor condiționale (`if`), dar și în instrucțiunile recursive (`try` și `while`). În mod similar, se poate utiliza și într-un bloc de gestiune al excepțiilor, condiția este să existe în prealabil măcar un bloc `except`. Sintaxa generală este:

```
try:
    <instructiuni care pot genera exceptii>
except Exceptia1:
    <instructiuni pentru tratarea Exceptia1>
except Exceptia2 as e:
    <instructiuni pentru tratarea Exceptia2 cu alias-ul e>
else:
    <instructiuni>
```

În această configurație, blocul de cod din instrucțiunea `else` va fi executat **doar** dacă blocul `try` nu a generat (ridicat) nicio excepție. În caz contrar, se va executa blocul `except`. În funcția analizată anterior, să presupunem că dorim afișarea (și utilizarea pe mai departe) a unei valori trunchiate a rezistenței termice. În acest caz, doar dacă aceasta a fost calculată (adică nicio altă excepție nu a fost ridicată), se poate folosi un bloc de cod `else` în care variabilei de lucru să îi fie atribuită valoarea trunchiată (acesta fiind doar un exemplu de utilitate). În exemplul următor în blocul `else` se utilizează o instrucțiune `if` scrisă pe o singură linie (operator ternar – v. § `if - kw.3`) care verifică dacă valoarea înmagazinată în variabila `rezistenta` este diferită de `None`. Dacă instrucțiunea este `True`, valoarea rezultatului împărțirii este rotunjit superior la a doua zecimală. Astfel, rezultatul final va fi `0.71` și nu `0.7142857142857143`, așa cum este rezultatul împărțirii lui `0.5` la `0.7` – cazul exemplului.

```
# se defineteste o functie care primeste 2 parametri si returneaza rezultatul
impartirii lor
def calcul_rezistenta_perete(grosime: int, conductivitate:int)->float:
    # se scrie in blocul try instructiunea care poate genera exceptia
    try:
```

```
        # se calculeaza rezistenta termica
        rezistenta = grosime/conductivitate
    # in blocul except se indica ce tip de exceptie ar putea aparea
    except ZeroDivisionError:
        # daca apare acea exceptie, rezistenta va fi None
        rezistenta = None
    except TypeError:
        print("Argumentele functiei trebuie sa fie numere nu sir de 
caracteret")
        rezistenta = None
    # se va returna valoarea rezistentetei termice
    finally:
        print('Aceast cod se executa indiferent de ce s-a petrecut anterior!')
    return rezistenta

# se apeleaza functia si ca argument pentru conductivitatea se da valoarea 0
rezistenta_caramida = calcul_rezistenta_perete(0.5, 0)
# se afiseaza valoarea, rezultatul fiind None
print(f'Valoarea rezistente este: {rezistenta_caramida}')
print('-'*65)
# se apeleaza functia si ca argument pentru conductivitatea se va trece un sir 
de caractere
rezistenta_caramida2 = calcul_rezistenta_perete(0.5, "0.3")
# apelarea functiei va genera exceptia TypeError
# executia se opreste si se ofera indicatii despre eroare
print(f'Valoarea resistentei este: {rezistenta_caramida2}')
print('-'*65)
# codul nu va fi executat daca se foloseste rise in exceptie
rezistenta_caramida3 = calcul_rezistenta_perete(0.5, 0.3)
print(f'Valoarea rezistentei este {rezistenta_caramida3}')
print('-'*65)
# se defineste o functie care primeste 2 parametri si returneaza rezultatul 
impartirii lor
def calcul_rezistenta_perete(grosime: int, conductivitate:int)->float:
    # se scrie in blocul try instructiunea care poate genera exceptia
    try:
        # se calculeaza rezistenta termica
        rezistenta = grosime/conductivitate
    # in blocul except se indica ce tip de exceptie ar putea aparea
    except ZeroDivisionError:
        # daca apare acea exceptie, rezistenta va fi None
        rezistenta = None
    except TypeError:
        print("Argumentele functiei trebuie sa fie numere nu sir de 
caracteret")
        rezistenta = None
    # instructiunea else va fi executata doar daca try nu ridica nicio eroare
```

```
    else:
        rezistenta = round(rezistenta, 2) if rezistenta is not None else 
rezistenta
    return rezistenta
rezistenta_caramida = calcul_rezistenta_perete(0.5, 0.7)
print(f'Valoarea rezistentei este {rezistenta_caramida}')
print(0.5/0.3)
OUTPUT:
Valoarea rezistentei este 0.71
```

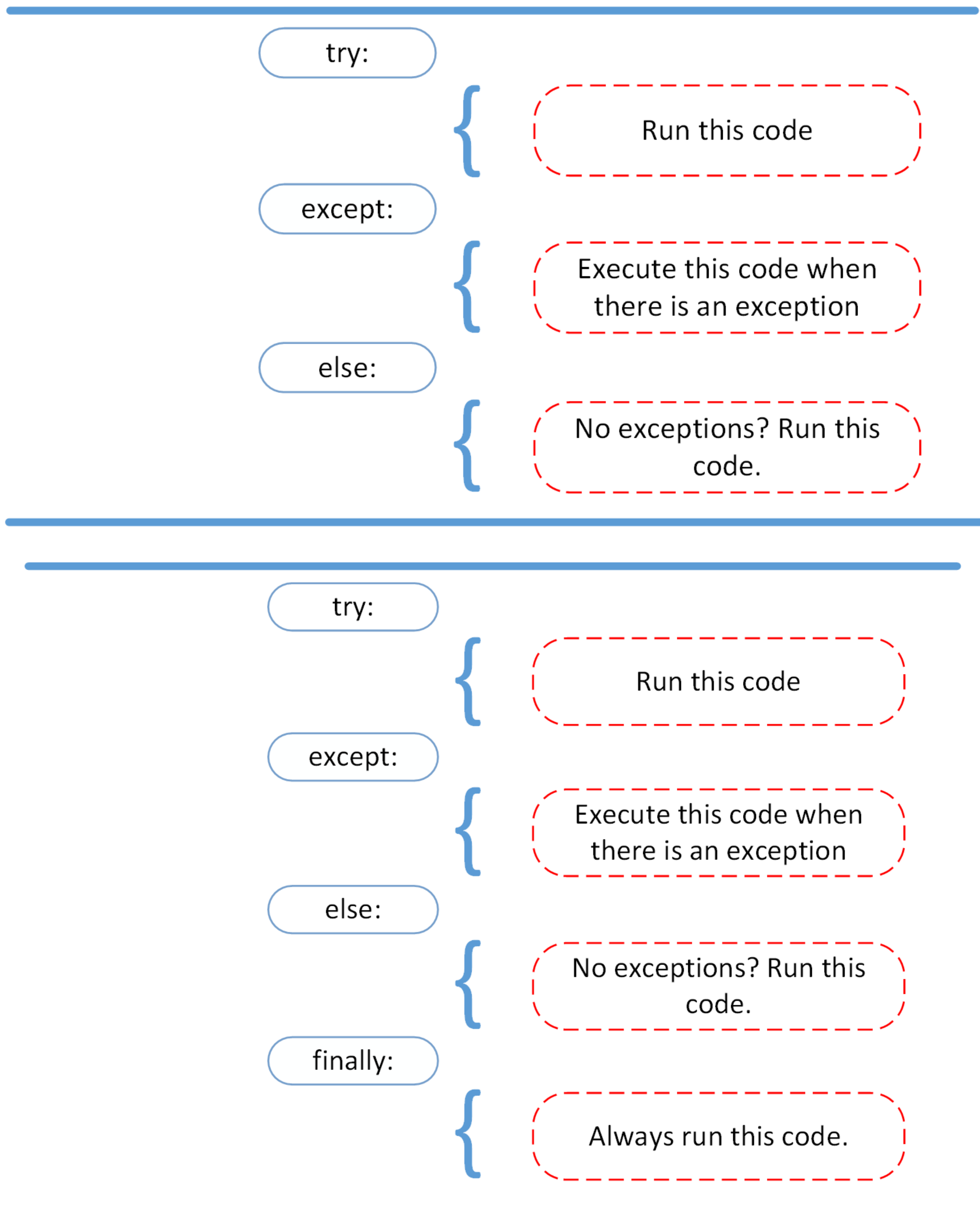


Figura 11. Schema de execuție a blocului `try` - `except`

`assert` este utilizat cu precădere în analiza aserțiunii (valorii de adevăr) unei expresii Python. Se folosește cu precădere în etapa de testare a codului scris și face parte din suita de ustensile pentru depanare (*debugging*) și testare în perioada de dezvoltare a proiectelor Python. O instrucțiune `assert` va permite testarea corectitudinii codului, prin verificarea stării de adevăr a anumitor condiții specifice. O instrucțiune `assert` va returna `True` dacă nu există o eroare de logică în programul scris. Sintaxa generală are forma:

```
assert <expresie>
```

Scopul acestei lucrări nu este de a intra adânc în metodele și uneltele depanării programelor scrise în Python, complexitatea algoritmilor scriși având scop didactic și de familiarizare a cititorilor cu sintaxa generală și, mai apoi, cu librăriile utilizate în domeniile științelor datelor și inteligenței artificiale. Drept urmare, `assert` nu va fi exemplificat. Pentru mai multe detalii, se poate analiza URL-ul sau documentația oficială.

SUMAR al acestor cuvinte cheie:

↔ `rise` permite programatorului să genereze o eroare oricând în cod
↔ `rise` permite verificarea îndeplinirii unei condiții și generarea unei excepții dacă nu este
↔ în clauza `try` toate instrucțiunile sunt executate până când este întâlnită o excepție
↔ `except` este utilizat pentru a prinde excepția/excepțiile care sunt ridicate în clauza `try`
↔ `else` delimitează secțiunea de cod care va fi executată doar dacă nu s-a întâlnit nicio excepție în clauza `try`
↔ `finally` delimitează secțiunea de cod care va fi executată indiferent dacă vreo excepție s-a generat anterior.

kw.9. Cuvinte cheie pentru manipularea datelor: `del`, `global`, `nonlocal`

`del` este utilizat în sintaxa Python pentru a anula (dealoca din memoria internă) o anumită variabilă sau un nume de variabilă, dar se poate utiliza și pentru tipuri de date încorporate mai complexe (cum ar fi listele, dicționarele) pentru a șterge datele dintr-un anumit index. Sintaxa generală este simplă, prezentată în continuare:

```
del <variabila>
```

```
# se definete o lista populata cu numere intregi
lista = [1, 2, 3, 4, 21, 7]
# se afiseaza lista initiala
print('Lista initiala:', lista)
# se itereaza prin lista, element cu element
for numar in lista:
    # se verifica daca numarul din iteratia curenta este par sau nu
    if numar % 2 == 0:
        # daca este par, se gaseste indexul(pozitia in lista) si se sterge
utilizand del
        del lista[lista.index(numar)]
# se afizeaza componenta listei dupa ce au fost sterse elementele
print("Lista dupa ce s-au sters numerele pare", lista)
OUTPUT:
Lista initiala: [1, 2, 3, 4, 21, 7]
Lista dupa ce s-au sters numerele pare: [1, 3, 21, 7]
```

Cel mai util exemplu este acela de a șterge anumite elemente dintr-o listă – chiar dacă există o metodă specială pentru a realiza acest lucru: metoda `.remove()`. Se creează inițial o listă pe care o populăm cu diverse numere întregi, atât pare cât și impare, după care se iterează prin această listă, analizând fiecare element în parte cu ajutorul unei bucle `for` – indicată pentru cazul în care se cunoaște dinainte numărul de iterații. Variabila de iterație `numar` va lua, pe rând, toate valorile din listă și se va analiza dacă restul împărțirii valorii curente la 2 este egal sau nu cu 0 (un număr par are restul împărțirii la 2 egal cu 0!) utilizând operatorul specific `%`. În cazul în care expresia condiționată returnează `True`, se va identifica indexul (poziția) elementului din lista inițială, prin utilizarea metodei specifice listelor `.index()`, al cărui argument va fi numărul din iterația curentă. Indexul va fi cel care va indica, de fapt, ce element va fi șters din lista inițială – utilizarea cuvântului cheie `del` – `del lista[lista.index(numar)]`.

`global` este utilizat pentru a defini variabile globale, care pot fi accesate în domeniul global al codului (de oriunde, inclusiv din interiorul funcțiilor și al claselor, fără a fi definite local, în interiorul acestora). În limbajul de programare Python există două domenii principale: domeniul `local`, care se referă la variabilele dintr-un domeniu bine specificat (spre exemplu, în corpul unei funcții, al unei clase sau variabilele iterative din sintaxele `for` și `while`) și domeniul `global` care reprezintă variabilele la nivel de modul/program principal. Totuși, interpretorul Python caută numele variabilelor utilizând regula LEGB: **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in, în această ordine. Sintaxa generală pentru utilizarea `global` este utilizarea:

```
global <variabila>
```

În fragmentul de cod următor se prezintă o modalitate simplistă de utilizare a acestui tip de variabilă globală. Se definește o variabilă denumită `variabila` și se inițializează cu valoarea 0. Se definește o funcție simplă care incrementează, la fiecare apelare, valoarea variabilei. Dacă aceasta nu se declară drept globală în domeniul funcției, nu se va putea utiliza, interpretorul Python generând o eroare: `UnboundLocalError: cannot access local variable 'variabila' where it is not associated with a value`. Acest lucru indică faptul că o variabilă declarată în afara domeniului de definiție al funcției nu poate fi alterată în interiorul funcției decât dacă aceasta este declarată drept variabilă globală.

```
# se defineste o variabila si de initializeaza cu valoarea 0
variabila = 0
# se afiseaza valoarea variabilei
print("Valoarea initiala: ", variabila)
# se defineste o funcie simpa
def increment():
    # se indica faptul ca variabila globala se utilizeaza in domeniul functiei
    global variabila
    # se incrementeaza valoarea variabilei globale
    variabila += 1
# se apeleaza functia care va incrementa variabila globala
increment()
print('Valoarea dupa incrementare: ', variabila)
OUTPUT:
Valoarea initiala:  0
Valoarea dupa incrementare:  1
```

nonlocal este utilizat pentru a defini variabile locale pentru a le putea altera din alt domeniu decât cel în care au fost declarate – similar cu global, având și sintaxa generală similară. Detalii despre domeniile Python se pot găsi aici.

kw.10. Cuvinte cheie pentru programarea asincronă: async, await

Programarea asincronă este paradigmă complexă și nu reprezintă scopul acestei lucrări, folosindu-se în general în programarea aplicațiilor care utilizează abundent baze de date complexe și care necesită executarea în paralel a mai multor taskuri. Un exemplu concret este reprezentat de aplicațiile de comunicare prin intermediul textului, al vocii și/sau al videoclipurilor. Utilizarea celor două cuvinte cheie nu reprezintă obiectul acestei cărți.

3.2. Exemple de utilizare a funcțiilor încorporate

Interpretorul din Python, versiunea 3.12, are încorporate nu mai puțin de **71** de funcții de utilitate generală, care pot fi folosite fără a importa niciun modul extern. Cele mai importante pentru scopul acestei lucrări vor fi descrise în acest subcapitol în ordine alfabetică.

f1. abs(argument) returnează valoarea absolută (valoarea în modul) a argumentului. Dacă argumentul este un număr complex, funcția returnează magnitudinea sa. Argumentul poate fi orice tip de dată numerică suportat de Python (număr întreg, număr flotant, număr complex – § 4. Tipuri de date principale și metodele). Exemple de utilizare a funcției se pot analiza în fragmentul următor de cod.

```
# se declara 3 variabile care se initializeaza cu 3 tipuri de date numerice:
int, float, complex
numar_int = -10
numar_float = -1.3
numar_complex = 3-4j
# se calculeaza modulul primului numar(tip int) si se atribuie valoarea unei
variabile
abs_numar_int = abs(numar_int)
# se afiseaza pe ecranul terminalului valoarea variabilei
print(f'|{numar_int}| = {abs(abs_numar_int)}')
# se afiseaza pe ecranul terminalului valoarea absoluta a celorlalte doua
tipuri de numere
# se foloseste functia abs() direct in corpul functiei print()
print(f'|{numar_float}| = {abs(numar_float)}')
print(f'|{numar_complex}| = {abs(numar_complex)}')
OUTPUT:
|-10| = 10
|-1.3| = 1.3
|(1.3-3j)| = 5.0
```

f2. all(iterabil) returnează True dacă toate elementele dintr-o dată iterabilă Python sunt adevărate și False în orice altă variantă. Un exemplu de utilizare a funcției este prezentat în fragmentul următor de cod, unde au fost declarate două date iterabile pentru care s-a folosit funcția all() pentru a verifica dacă absolut toate elementele au valoarea de adevăr True.

OBSERVAȚIE 1. Prin dată iterabilă sau iterabil se înțelege orice tip de dată din Python care se poate parcurge iterativ, element cu element. Exemple de iterabile: liste, dicționare, șiruri de caractere, tupluri, seturi etc. Toate aceste tipuri de date vor fi detaliate în § 4. Tipuri de date principale și metodele

OBSERVAȚIE 2. În Python, data numerică 0 reprezintă `False`, toate celelalte numere fiind asociate cu `True`. Același lucru se aplică și la alte tipuri de date care sunt formate doar din mulțimea vidă, cum ar fi: liste, tupluri, dicționare goale (0 elemente) și la cuvântul cheie `None`.

```
# se declara o variabila si se initializeaza cu un sir de caractere - iterabil
sir_caractere = 'Facultatea de Energetica'
# se verifica daca toate elementele iterabilului sunt True folosind functia
all()
print(all(sir_caractere))
# se declara o variabila de tip lista care contine mai multe tipuri de date,
printre care si None
lista = [1, 2, 3, 'Inginer', None]
# se verifica daca toate elementele iterabilului sunt True folosind functia
all()
print(all(lista))
OUTPUT:
True
False
```

Trebuie menționat faptul că iterabilul poate consta și într-o listă creată local utilizând o expresie condiționată. În exemplul următor se verifică dacă toate elementele unei liste sunt `True`, dar respectând o condiție impusă local de programator: toate numerele sunt pare. Rezultatul va fi `False` deoarece cel puțin un element din lista analizată este impară.

```
# se declara o variabila si se initializeaza cu o lista formata din numere
intregi si flotante
lista = [1, -3, 2, 4, 3.3, 1.2]
# se verifica daca toate elementele sunt pare folosind o expresie conditionate
print(all(element % 2 == 0 for element in lista))
OUTPUT:
False
```

`f3. any(iterabil)` returnează `True` dacă măcar un element dintr-o dată iterabilă Python este adevărat și `False` în orice altă variantă. Un exemplu de utilizare a funcției este prezentat în fragmentul de cod următor, în care este creată o variabilă inițializată cu o listă neomogenă compusă din mai multe tipuri de elemente Python: un număr întreg, un număr zecimal, un șir de caractere și două variabile de tip boolean. Lista este verificată utilizând funcția `any()`.

```
# se declara o variabila si se initializeaza cu o lista formata din numere intregi, flotante, șir de caractere și variabile booleane
lista = [True, 'Inginer', 1.5, 0, False]
# se verifica daca macar un element are valoarea True
print(any(lista))
OUTPUT:
True
```

În mod similar cu funcția `all()`, și în cazul acestei funcții se pot folosi argumente formate din expresii condiționate. În exemplul următor este prezentat un astfel de exemplu.

```
# initializarea unei liste formata din numere intregi
lista_test = [4, 5, 8, 9, 10, 17]
# se afiseaza in terminal lista originala
print("Lista originala : ", lista_test)
# Se verifica daca oricare element din lista indeplineste o conditie
rezultat = any(element > 10 for element in lista_test)
# se afiseaza in terminal rezultatul
print("Indeplineste vreun element conditia impusa(oricare element mai mare ca 10)?", rezultat)
OUTPUT:
Lista originala :  [4, 5, 8, 9, 10, 17]
Indeplineste vreun element conditia impusa (oricare element mai mare ca 10)?
True
```

`f4.` `ascii(obiect)` returnează o versiune lizibilă a oricărui obiect trecut ca argument (șir de caractere, tuple, liste etc.) trecând peste caracterele non-ascii, pe care le va înlocui cu unul dintre caracterele ” \x”, ” \u” sau ” \U”. Atât numele cât și funcționalitatea provin de la standardul ASCII care este reprezentarea numerică a literelor – pentru a putea fi procesate de procesorul calculatorului. Astfel, funcția, în cazul în care în codul Python apare un caracter care nu face parte din acest cod, funcția aceasta îl va înlocui, la afișarea acestuia, cu echivalentul său ASCII:

```
# initializarea unei variabile de tip str care contine un caracter nona-scii
text = 'Pythön is interesting'
# se inlocuieste ö cu valoarea sa in cod ascii
print(ascii(text))
print(ascii("¥"))
# initializarea unei variabile de tip str care contine un caracter nona-scii
test_text = "G ë ê k s f ? r G ? e k s"
print(ascii(test_text))
OUTPUT:
'Pyth\xf6n is interesting'
'\xa5'
'G \xeb \xea k s f ? r G ? e k s'
```

`f5.` `bin(int)` returnează versiunea un șir de caractere care conține versiunea binară a unui număr **întreg** – tip `int` introdus ca argument. Dacă se folosește altă dată numerică va genera o eroare de sintaxă de tip `TypeError`. Spre exemplu, versiunea binară a numărului 2 este 0b10.

Rezultatul va începe întotdeauna cu prefixul 0b – versiunea binară fiind, de fapt, numărul fără prefix – 10 pentru 2 și 1011010 pentru numărul 90, precum este exemplificat în continuare.

```
print("Versiunea binara a numarului 9 este:", bin(90))
OUTPUT:
Versiunea binara a numarului 9 este: 0b1011010
```

f6. `bool(obiect)` returnează valoarea booleană (de adevăr) a obiectului trecut ca argument. Obiectul argument poate fi orice fel de dată Python și funcția va returna `True` cu următoarele excepții – obiectul este gol, `False`, 0 sau `None`.

```
print("Valoarea de adevar a numarului 0 este: ", bool(0))
print("Valoarea de adevar a numarului 1 este: ", bool(1))
print("Valoarea de adevar a unei liste goale este: ", bool([]))
print("Valoarea de adevar a variabilei booleane False este", bool(False))
OUTPUT:
Valoarea de adevar a numarului 0 este:  False
Valoarea de adevar a numarului 1 este:  True
Valoarea de adevar a unei liste foale este:  False
Valoarea de adevar a variabilei booleane False este False
```

f7. `bytearray(sursa, encoding, error)` returnează un obiect `bytearray` – o secvență mutabilă (care se poate modifica) de octeți (bytes) – numere întregi în intervalul 0 și 256 (fără a include limita superioară). Parametrii pe care îi poate primi funcția sunt:
- `sursa` (opțional) sursă pe care o folosește pentru a crea obiectul `bytearray`. Poate fi un întreg (`int`) – caz în care se creează un `bytearray` gol cu dimensiunea egală cu numărul, sau un șir de caractere (`str`) – caz în care obligatoriu trebuie adăugat și parametrul `encoding`;
- `encoding` (doar în cazul în care primul parametru este de tip `str`) – algoritmul de codificare al șirului de caractere în octeți;
- `error` (doar dacă `encoding` este specificat) – specifică cum să abordeze ipoteza în care codificarea șirului de caractere în octeți nu reușește.

```
# se creeaza si se afiseaza un obiect bytearray gol de dimensiunea 3
print(bytearray(3))
# se creeaza o lista cu numere primele 6 numere prime
# va contine valorile 2, 3, 5, 7, 11, 13
numere_prime = [numar for numar in range(2,15) if not [nr for nr in range(2, numar) if not numar % nr]]
# se afiseaza pe consola terminalului sirul de octeti generat din listsa cu numere intregi
print(f'bytearray din lista: {numere_prime}: {bytearray(numere_prime)}')
# se creeaza un sir de caractere care contine un mesaj
mesaj = 'Inginerie Energetica - Inteligenta Artificiala'
# se genereaza obiectul bytearray utilizand codificarea utf-8
output = bytearray(mesaj, 'utf-8')
print(output)
OUTPUT:
bytearray(b'\x00\x00\x00')
```

```
bytearray din lista: [2, 3, 5, 7, 11, 13]: bytearay(b'\x02\x03\x05\x07\x0b\r')
bytearray(b'Inginerie Energetica - Inteligenta Artificiala')
```

f8. **bytes**(sursa, encoding, error) returnează un obiect bytes. Singura diferență între funcția bytearray și bytes este faptul că obiectul returnat de bytes este imutabil, însemnând că nu poate fi modificat după ce a fost creat. În schimb, obiectul bytearray este mutabil.

```
mesaj = 'Inginerie Energetica'
# se genereaza obiectul bytearray utilizand codificarea utf-16
output = bytes(mesaj, 'utf-16')
print(output)
OUTPUT:
b'\xff\xfeI\x00n\x00g\x00i\x00n\x00e\x00r\x00i\x00e\x00
\x00E\x00n\x00e\x00r\x00g\x00e\x00t\x00i\x00c\x00a\x00'
```

f9. **callable**(obiect) returnează True dacă obiectul este apelabil, însemnând un obiect Python care poate fi apelat utilizând paranteze și, opțional unul sau mai multe argumente (spre exemplu, funcțiile, metodele și clasele Python sunt obiecte apelabile).

```
# se verifica daca un obiect tip int este apelabil
print(callable(3))
# se verifica daca functia incorporata abs() este apelabila
print(callable(abs))
OUTPUT:
False
True
```

f10. **chr**(numar) returnează caracterul unicode corespunzător numărului de tip întreg furnizat ca argument. Numărul trebuie să reprezinte un caracter valid, adică să se regăsească în intervalul 0 – 1.114.111, altfel se generează o eroare de valoare: **ValueError**: chr() arg not in range(0x110000).

```
# se afizeaza litera unicode corespunzatoare valorii 98 - b
print("Valoarea ASCII a numarului 98 este", chr(98))
# se afizeaza litera unicode corespunzatoare valorii 8364 - €
print("Valoarea ASCII a numarului 8364 este", chr(8364))
OUTPUT:
Valoarea ASCII a numarului 98 este b
Valoarea ASCII a numarului 8364 este €
```

f11. **classmethod**(functie) returnează o metodă de clasă dintr-o funcție sau metodă. Funcția acceptă ca argument numele altei funcții pe care o convertește într-o metodă de clasă. Metoda de clasă este legată de clasa în sine și nu de obiectul/instanța creat/ă din acea clasă, însemnând că poate fi apelată prin intermediul atât al instanței cât și al clasei în sine. În versiunile noi de Python, în loc de funcția **classmethod**() se folosește decoratorul @classmethod înaintea definirii metodei pentru a indica faptul că aceasta este legată de clasă. Mai multe detalii aici.

În exemplul următor se creează o clasă denumită Student și care, la inițializare, necesită introducerea a două argumente, așa cum este definit constructorul (inițializatorul)

`__init__(self)`: `nume` și `varsta`. Parametrul `self` indică faptul că acea metodă este legată de instanța creată. Cea de-a doua metodă este o metodă de clasă utilizată pentru a calcula vârsta studentului în funcție de anul în care s-a născut. Aceasta se definește utilizând decoratorul `@classmethod` și, pe lângă parametrul implicit care o leagă de clasă în sine, `cls`, la apelare, aceasta va mai avea nevoie de două argumente: `nume` și `an_nastere`, calculează vârsta curentă a studentului și returnează o instanță clasei în sine: `return cls(nume, date.today().year - an_nastere)`. Ultima metodă este utilizată de instanța creată pentru a afișa datele specifice.

```
from datetime import date

# se creeaza o clasa denumita student
class Student:
    def __init__(self, nume, varsta):
        self.nume = nume
        self.varsta = varsta

    # se creeaza o metoda de clasa
    @classmethod
    def anNastere(cls, nume, an):
        return cls(nume, date.today().year - an)

    # se creeaza o metoda simpla
    def afisare_date(self):
        print(self.nume + " are " + str(self.varsta) + " ani")

# se instantiaza clasa Student cu argumentele necesare
student_iacob = Student('Iacob', 20)
student_iacob.afisare_date()

# apelare metoda de clasa prin intermediul clasei
stundet_zora = Student.anNastere('Zora',  2018)
stundet_zora.afisare_date()
OUTPUT:
```

Iacob are 20 ani
Zora are 5 ani

`f12. compile(source, filename, mode)` returnează argumentul `source` ca un obiect cod, gata să fie executat utilizând funcția `exec()`. De menționat că mai există încă trei parametri opționali (`flags, optimize, dont_inherit`), dar aceștia nu sunt utilizați frecvent. Parametrii utilizați:

- **`source`** – un șir de caractere normal sau un șir de caractere octeți care să fie compilat;
- **`filename`** – numele fișierului din care a fost citită sursa. Poate avea orice nume dacă informația nu provine dintr-un fișier;
- **`mode`** – pot fi trei moduri în care poate fi compilat codul sursă:
  - `'eval'` – dacă sursa este o expresie singulară;
  - `'exec'` – dacă sursa este un bloc de cod Python (funcții, clase, declarații etc.);
  - `'single'` – dacă sursa este o singură instrucțiune interactivă.

```
# se creeaza o variabila si se initializeaza cu un sir de caractere
# compus din cod Python
sursa = 'a = 7\nb=3\nmultiplicare=a*b\nprint("multiplicare =",multiplicare)'
# se compileaza obiectul utilizand argumente specifice
obiect = compile(sursa, 'numereInmultite', 'exec')
# se executa obiectul utilizat functia incorporata exec()
exec(obiect)
OUTPUT:
multiplicare = 21
```

```
# Exemplul2
numar_int = 50
obiect2 = compile('numar_int', 'test', 'single')
print(type(obiect2))
exec(obiect2)
OUTPUT:
<class 'code'>
50
```

`f13. complex(real, imaginary)` returnează numărul complex format din cele două argumente: `real` și `imaginary` sub forma: `real + imaginary*j`. Parametrii utilizați:

- `real` – poate fi orice dată numerică Python: `int`, `float`, `Decimal()`, `Fraction()` care va reprezenta partea reală a numărului complex (implicit este 0). Există posibilitatea să fie trecut un șir de caractere sub forma unui număr complex (spre exemplu`'3+5j'`), caz în care al doilea argument trebuie să lipsească;
- `imaginary` – (opțional) poate fi orice dată numerică Python: `int`, `float`, `Decimal()`, `Fraction()`. Implicit are valoarea 0.

```
# se importa libraria fractions si se creeaz un numar rational cu ajutorul
clasei Fraction()
import fractions
# se creeaza variabila complex1 si se initializeaza cu numarul complex 1+0j
complex1 = complex(1)
print(complex1)
# se creeaza variabila complex3 si se initializeaza cu un numar complex
rational
complex2 = complex(fractions.Fraction(1, -2), 2)
print(complex2)
# se creeaza variabila complex2 si se initializeaza cu un numar complex
complex3 = complex('3+7j')
print(complex3)
OUTPUT:
(1+0j)
(-0.5+2j)
(3+7j)
```

`f14. delattr(object, attribute)` șterge atributul specificat din obiectul specificat. Trebuie menționat faptul că funcția nu returnează nimic (de fapt, returnează obiectul mulțime vidă – `None`), deci rezultatul nu poate fi stocat într-o variabilă. În exemplul următor se creează o clasă

care, la instanțiere, primește 3 parametri de tip `int` și, cu ajutorul metodei `numere_pitagorice()`, determină dacă acestea sunt numere pitagorice sau nu, verificând dacă suma pătratelor primelor două este egală cu pătratul celei de-a treia mărimi. După instanțiere, se șterge metoda și se reîncearcă utilizarea ei într-un bloc de captare a excepțiilor `try - except`, generându-se, în mod evident o eroare care indică faptul că obiectul `Triplete` (numele clasei) nu are atributul/metoda denumit/ă `numere_pitagorice`.

```
class Triplete:
    def __init__(self, x, y, z) :
        self.x = x
        self.y = y
        self.z = z

    def numere_pitagorice(self):
        return self.x**2 + self.y**2 == self.z**2

caz = Triplete(12, 2, 1)
print('12, 2, 1 sunt numere pitagoreice?', caz.numere_pitagoreice())

delattr(Triplete, 'numere_pitagoreice')
try:
    caz.numere_pitagoreice()
except Exception as e:
    print(e)
OUTPUT:
12, 2, 1 sunt numere pitagoreice? False
'Triplete' object has no attribute 'numere_pitagoreice'
```

`f15. dict()` este o funcție constructor de dicționare Python. Tipurile de date standard utilizate de interpretorul Python vor fi detaliate în § 4. Tipuri de date principale și metodele . Pe scurt, un dicționar Python este un tip de dată utilizat pentru a stoca date sub forma unor perechi `key: value` (cheie: valoare). Funcția reprezintă echivalentul utilizării acoladelor (`{}`) în crearea dicționarelor. Fiind format din perechi de date, la crearea acestuia trebuie introduse atât o valoare pentru `key` cât și o valoare pentru `value`.

```
# se creeaza un dictionar ce contine date de identificare ale cartii
# se utilizeaza functia constructor dict()
info_dict = dict(titlu="I.A/I.E", an=2024, autor='Carutasiu Mihail-Bogdan')
# se afiseaza dictionarul in terminal
print(info_dict)
# se intializeaza acelasi dictionar dar folosind {} in locul constructorului
# se verifica daca cele doua sunt egale si rezultatul este atribuit variabilei
verificare
verificare = info_dict == {'titlu': "I.A/I.E", 'an': 2024, 'autor': 'Carutasiu
Mihail-Bogdan' }
print('Cele doua dictionare sunt egale?', end=' ')
print(verificare)
```

```
OUTPUT:
{'titlu': 'I.A/I.E', 'an': 2024, 'autor': 'Carutasiu Mihail-Bogdan'}
Cele doua dictionare sunt egale? True
```

OBSERVAȚIE. Dacă se utilizează constructorul `dict()`, `keys` sunt trecute fără ghilimele simple - `'`, sau duble - `"` însemnând că sunt, de fapt, numele unor variabile inițializate. În schimb, utilizând acoladele, `keys` sunt de timpul `str` – șir de caractere. Valorile pot fi orice tip de dată suportată de limbajul Python.

`f16. dir(obiect)` returnează o listă ce conține toate proprietățile și metodele unui obiect trecut ca argument. Obiectul poate fi orice tip de dată Python. Se poate utiliza funcția și pentru a returna informațiile specifice unei clase definite de programator. Generând o listă de elemente, prin rezultatul returnat de funcția `dir()` se va putea itera cu o buclă `for`. În exemplul următor a fost analizată clasa Triplete definită anterior, pentru care se știe sigur că a fost definită metoda `numere_pitagoreice()`. Funcția `dir()` va returna în schimb și metodele încorporate în orice clasă și activate la utilizarea constructorului de clase `class`, spre exemplu: `__class__`, `__delattr__`, `__dict__`, `__dir__` etc. Toate metodele sunt scrise utilizând două simboluri underscore "_". Pentru a nu le afișa pe ecranul consolei, în bucla iterativă `for` se folosește o instrucțiune condiționată și, dacă itemul de la iterația curentă se termină în __, iterația este sărită. Astfel, doar metoda/ele creată/e de programator vor fi afișate.

```
for item in dir(Triplete):
    if item.endswith('__'):
        continue
    print(item)
OUTPUT:
numere_pitagorice
```

```
for item in dir(2.2):
    if item.endswith('__'):
        continue
    print(f' "{item}" este o metoda specifica pentru unu numar de tip foat.')
OUTPUT:
"as_integer_ratio" este o metoda specifica pentru unu numar de tip foat.
 "conjugate" este o metoda specifica pentru unu numar de tip foat.
 "fromhex" este o metoda specifica pentru unu numar de tip foat.
 "hex" este o metoda specifica pentru unu numar de tip foat.
 "imag" este o metoda specifica pentru unu numar de tip foat.
 "is_integer" este o metoda specifica pentru unu numar de tip foat.
 "real" este o metoda specifica pentru unu numar de tip foat.
```

`f17. divmode(dividend, divizor)` returnează un tuplu (v. § 4. Tipuri de date principale și metodele) care conține câtul și restul împărțirii primului argument – `dividend` la cel de-al doilea argument – `divizor`. Cei doi parametri pot fi orice dată numerică suportată de Python, cu excepția numerelor complexe. De fapt, metoda `divmode()` folosește argumentele introduse pentru a realiza două operații specifice datelor numerice în Python: `floor division (dividend`

`// divizor`), care va genera partea întreagă a calculului și `mod` (`dividend % divizor`), care va genera câtul împărțirii și va returna ambele valori în același timp și cu același tip de dată: tuplu.

```
# se creeaza doua variabile si se initializeaza cu numere de tip int si foat
dividend = 13
divisor = 4.5
# se foloseste funcia divmod() care returneaza un tuplu cu catul si restul
rezultat = divmod(dividend, divisor)
# se afiseaza datele din tuplu prin indexarea lui.
print(f'Catul operatiei {dividend}/{divisor} este {rezultat[0]}')
print(f'Restul operatiei {dividend}/{divisor} este {rezultat[1]}')
# pentru comparatie, se utilizeaza divmod() si se printeaza tuplul
print('Rezultatul sub forma de tuplu al operatiei 35/4 este: ', divmod(35, 5))
OUTPUT:
Catul operatiei 13/4.5 este 2.0
Restul operatiei 13/4.5 este 4.0
Rezultatul sub forma de tuplu al operatiei 35/4 este:  (7, 0)
```

`f18. enumerate(iterabil)` returnează un obiect de tip `enumerate` din argumentul tip iterabil. Practic, va genera o listă care va conține perechi de indecși și datele din iterabil (prin indexul unui iterabil se înțelege poziția pe care acel element se află în succesiunea de elemente constituente ale iterabilului). Opțional, se poate indica cu ce număr să înceapă indexarea (`start` implicit are valoarea 0). În exemplul de cod următor se creează o listă care este populată cu cinci tipuri de date diferite; această listă este utilizată ca argument pentru funcția `enumerate` care va genera un obiect de tip enumerate: `<enumerate object at 0x00000153B6097CE0>`. Pentru a putea fi afișat pe ecranul consolei, acest tip de obiect trebuie să fie utilizat ca argument într-o altă funcție care va genera, utilizându-l, o listă – `list(enumerate()`. În cel de-al doilea exemplu, iterabilul utilizat este un șir de caractere și, în plus, este utilizat și parametrul `start`, căruia i se atribuie valoarea 3, indicând indexul de start al obiectului `enumerate`. De observat faptul că șirul de caractere `'energie'` este divizat pe litere și fiecărei litere i se atribuie un index increment cu valoarea 1, începând cu valoarea indicată prin argumentul `start`. Mai mult, obiectul `enumerate` nu mai este folosit în conjuncție cu funcția `list()`, ci este iterat cu ajutorul buclei `for`.

```
# se genereaza o lista - care este un obiect iterabil in Python
lista = [1, 2, 'I.A.', 14, 98, 'Inginerie Energetica']
# utilizeaza enumerate()
# rezultatul este folosit ca argument pentru functia list()
print(list(enumerate(lista)))
for item in enumerate('energie', start=3):
    print(item, end='; ')
OUTPUT:
[(0, 1), (1, 2), (2, 'I.A.'), (3, 14), (4, 98), (5, 'Inginerie Energetica')]
(3, 'e')/ (4, 'n')/ (5, 'e')/ (6, 'r')/ (7, 'g')/ (8, 'i')/ (9, 'e')/
```

OBSERVAȚIE. În Python, indexarea începe de la 0, însemnând că primului element dintr-un iterabil i se atribuie automat valoarea 0.

f19. eval(expresie, globals, locals) evaluează și execută o expresie, returnând o valoare numerică sau booleană. Trebuie menționat faptul că expresia trebuie să fie o instrucțiune Python legală, fără erori de sintaxă. Parametrii:
↔ expresie – expresia de evaluat introdusă sub forma unui șir de caractere – string;
↔ globals – opțional, un dicționar conținând parametrii globali;
↔ locals – opțional, un dicționar conținând parametrii locali.

```
numar = 13
numar_dublu = eval('numar * 2' )
print(numar_dublu)
print(eval("sum([10, 20, 30, 40])"))
OUTPUT:
26
100
```

Dacă nu se folosește funcția eval(), expresia va fi tratată ca un simplu șir de caractere și nu va fi evaluată de interpretorul Python, ci doar afișată ca un mesaj.

```
# se creeaza o variabila si se initializeaza cu un expresie matematica scrisa
sub forma de string
expresie = 'x * (x+10) + (x+20)'
# functia print va afisa informatia din interiorul sirului de caractere
print(expresie)
# se initializeaza si necunoscuta x cu o valoare
x = 13
print(eval(expresie))
OUTPUT:
x * (x+10) + (x+20)
332
```

f20. exec(expresie, globals, locals) execută un bloc de cod Python creat dinamic. A mai fost folosită și în conjuncție cu metoda compile(). Utilizată similar cu eval(), având aceiași parametri definitorii, există totuși o diferență majoră: exec() poate executa bucăți mai mari de cod și nu se rezumă la o expresie singulară, deși viteza de rulare scade. Parametrii:
↔ expresie – expresia de evaluat introdusă sub forma unui șir de caractere – string;
↔ globals – opțional, un dicționar conținând parametrii globali;
↔ locals – opțional, un dicționar conținând parametrii locali.

```
# se creeaza o variabila si se initializeaza cu un sir de caractere
# compus din cod Python
sursa = 'a = 10\nb=3\nmultiplicare=a+b\nprint("adunare =", adunare)'
# se compileaza obiectul utilizand argumente specifice
obiect = compile(sursa, 'numereAdunate', 'exec')
# se executa obiectul utilizat functia incorporata exec()
exec(obiect)
OUTPUT:
adunare = 13
```

```
program = "print('Suma dintre 10 si 30 este', (10 + 30))"
exec(program)
OUTPUT:
Suma dintre 10 si 30 este 40
```

`f21. filter(functie, iterabil)` returnează un obiect tip iterator rezultat în urma unei filtrări a unui obiect iterabil prin intermediul unei funcții test (filtru) care returnează `True` – elementul trece de filtru și este returnat sau `False` – elementul nu trece de filtru și nu este returnat. Funcția test este aplicată iterativ fiecărui element din obiectul filtrat. Este o funcție extrem de utilă în domeniul științei, prin care se pot crea foarte simplu noi seturi de date condiționând un set mai mare de date (de exemplu, de interes ar fi datele mai mari sau egale cu zero în momentul în care analizăm variațiile radiației solare despre care știm că nu poate avea valori negative). Funcția `filter()` returnând un obiect iterator, acesta va trebui să fie pasat drept argument într-o funcție constructor pentru un iterabil: `list()`, `set()` etc.

```
# se creeaza o variabila si se populeaza cu valori intregi
valori_radiație_solara = [-2, -1, 0, -4, 10, 15, 25, 79, 450, 900, 1300]
# se creeaza functia filtru - returneaza True doar daca elemetul este pozitiv
def doar_pozitiv(element):
    if element < 0:
        return False
    return True

print('Lista initiala', end=': ')
print(valori_radiație_solara)
# se filtreaza datele din lista initiala utilizand functia creata anterior
radiatie_solara_filtrata = tuple(filter(doar_pozitiv, valori_radiație_solara))
print('Tuplul dupa filtrare', end=': ')
print(radiatie_solara_filtrata)
OUTPUT:
Lista initiala: [-2, -1, 0, -4, 10, 15, 25, 79, 450, 900, 1300]
Tuplul dupa filtrare: (0, 10, 15, 25, 79, 450, 900, 1300)
```

OBSERVAȚIE 1. Funcția test, când se trece ca argument în funcția `filter()`, este trecută doar cu denumirea, fără paranteze după nume, care reprezintă apelarea acesteia. Drept urmare, funcția test NU se apelează în cadrul funcției `filter()`, ci doar se specifică denumirea sa.

OBSERVAȚIE 2. A se observa diferența de sintaxă între listă `[-2, -1, 0, -4, 10, 15, 25, 79, 450, 900, 1300]` – paranteze drepte și un tuplu: `(0, 10, 15, 25, 79, 450, 900, 1300)` – paranteze rotunde. Acestea sunt, pe lângă constructorii impliciți – `list()` și `tuple()`, metode legale de creare a obiectelor respective.

Dat fiind că funcțiile test au o utilitate specifică și, în multe cazuri, deservesc doar filtrării unor elemente, o metodă elegantă și eficientă este de a folosi funcții `lambda` – v. § 3.1. Exemple de utilizare a cuvintelor cheie în cod pentru reîmprospătarea cunoștințelor. Funcțiile `lambda` au funcționalitate și utilitate specifice, putând foarte ușor servi drept funcție filtru, așa cum este cazul prezentat prin intermediul fragmentului de cod următor. Funcția `lambda valoare: (valoare > 0)` va returna `True` doar dacă expresia dintre paranteze este adevărată – valoarea

este pozitivă. Tehnic, cele două exemple produc același rezultat, dar modalitățile de scriere sunt diferite. De preferat este utilizarea celei de-a doua metode.

```
# se creeaza o variabila si se populeaza cu valori intregi
valori_radiație_solara = [-2, -1, 0, -4, 10, 15, 25, 79, 450, 900, 1300]
# se creeaza functia filtru - returneaza True doar daca elemetul curent este pozitiv
print('Lista initiala', end=': ')
print(valori_radiație_solara)
# se filtreaza datele din lista initiala utilizand o functie lamdba
radiatie_solara_filtrata = list(filter(lambda valoare: (valoare>0), valori_radiație_solara))
print('Lista dupa filtrare', end=': ')
print(radiatie_solara_filtrata)
OUTPUT:
Lista initiala: [-2, -1, 0, -4, 10, 15, 25, 79, 450, 900, 1300]
Lista dupa filtrare: (0, 10, 15, 25, 79, 450, 900, 1300)
```

O a treia posibilitate de utilizare a funcției `filter()` este de a folosi cuvântul cheie `None` în loc de funcție test. În acest caz, doar elementele care prin natura lor generează `True` (toate obiectele mai puțin `0`, `False` și obiectele goale (liste, tupluri, dicționare etc.) vor fi păstrate după filtrare.

```
# se creeaza o variabila si se populeaza cu diverse obiecte
valori_radiație_solara = [0, -1, 0, -4, 10, 15, 900, 1300, False, list(), ()]
# se creeaza functia filtru - returneaza True doar daca elemetul curent este pozitiv
# se filtreaza datele din lista initiala utilizand functia creata anterior
print('Lista initiala', end=': ')
print(valori_radiație_solara)
radiatie_solara_filtrata = set(filter(None, valori_radiație_solara))
print('Set dupa filtrare', end=': ')
print(radiatie_solara_filtrata)
OUTPUT:
Lista initiala: [0, -1, 0, -4, 10, 15, 900, 1300, False, [], ()]
Set dupa filtrare: {900, 10, 15, 1300, -4, -1}
```

`f22. float(numar)` returnează un număr rațional (în virgulă flotantă) din argumentul indicat. Parametrul funcției poate fi o dată numerică sau un obiect tip șir de caractere care să fie versiunea literală a unui număr: `'23'`, spre exemplu. Practic, cu ajutorul acestei funcții se realizează conversia explicită a unui număr (întreg sau număr literal) în număr flotant. Dacă nu se indică niciun argument, funcția va returna numărul flotant `0.0`.

În cazul în care se încearcă rezolvarea unei expresii matematice formate doar din numere, trecerea de la număr întreg la număr flotant se face automat, dar încercarea de a executa calculul între un număr și un număr literal va genera o eroare de tip `TypeError: unsupported operand type(s) for +: 'int' and 'str'`. Acest lucru înseamnă că sintaxa Python nu permite astfel de operații. În schimb, utilizând conversia explicită a numărului literal, acesta va deveni număr flotant și operația este valabilă, executând fără probleme, așa cum este detaliat în codul următor.

```
print(12 + float('78'))
OUTPUT:
90.0
```

```
# se creeaza modifica implicit variabila tip str
numar_str = '23.12'
print(f'Numarul {numar_str} este de tip: ', type(numar_str))
str_float = float('23.12')
print(f'Numarul {str_float} este de tip: ', type(str_float))
# se creeaza modifica implicit variabila tip str
numar_int = 178
print(f'Numarul {numar_int} este de tip: ', type(numar_int))
int_float = float(numar_int)
print(f'Numarul {int_float} este de tip: ', type(int_float))
OUTPUT:
Numarul 23.12 este de tip:  <class 'str'>
Numarul 23.12 este de tip:  <class 'float'>
Numarul 178 este de tip:  <class 'int'>
Numarul 178.0 este de tip:  <class 'float'>
```

OBSERVAȚIE. Ca o ciudățenie, se poate folosi drept argument al funcției `float()` și obiectele `infinity`, `inf`, `NaN` – indiferent de modul de scriere (majuscule sau minuscule). Operațiunea este legală în Python și va genera obiecte de tip `float`, denumite `inf` și `nan`.

`f23. format(valoare, format)` formatează obiectul dorit în funcție de un format specific precizat și îl returnează sub forma unui text (șir de caractere). Parametrul `format` este în esență un caracter (`char`) care specifică modul în care să se realizeze formatarea respectivă și se pot adăuga oricâte pentru o obține formatul dorit – se adaugă toate sub forma unui șir de caractere. Următoarele caractere sunt disponibile și sunt exemplificate în bucățile de cod prezentate în continuare:

'<' (aliniază la stânga în limita spațiului disponibil); '>' (aliniază la dreapta în limita spațiului disponibil); '^' (centrează în limita spațiului alb disponibil); '=' (plasează semnul pe cea mai din stânga poziție); '+' (indică dacă rezultatul este pozitiv sau negativ); '-' (indică doar valorile negative); ' ' (se utilizează un spațiu pentru a indica numerele pozitive); ',' (se utilizează virgula ca separator de mii); '_' (se utilizează underscore ca separator de mii); 'b' (formatare binară); 'c' (convertește valoarea în caracterul unicode corespunzător); 'd' (formatare decimală); 'e' (format științific cu e minusculă); 'E' (format științific cu E majusculă); 'f' (formatare cu număr fix de zecimale); 'F' (formatare cu număr fix de zecimale cu majusculă): 'g' (formatare generală); 'G' (formatare generală cu majusculă); 'o' (formatare octală); 'x' (formatare hex – minusculă); 'X' (formatare hex – majusculă); 'n' (formatare numerică); '%' (formatare procentuală).

```
# argumente tip int
print("Numarul este: {:d}".format(871))
# argumente tip float
print("Numarul flotant este:{:f}".format(871.4567898))
# formatare octala, binara si hexazecimala
print("bin: {0:b}, oct: {0:o}, hex: {0:x}".format(13))
OUTPUT:
Numarul este: 871
Numarul flotant este:871.456790
bin: 1101, oct: 15, hex: d
```

```
# numar int cu o latime minima (7 caractere in cazul asta)
print("{:7d}".format(1))
# numar int cu o latime minima (10 caractere in cazul asta)
# precizie de 3 zecimale
print("{:10.3f}".format(12.2346))
# numar int cu o latime minima (3 caractere in cazul asta)
# umpluta cu 0 daca nu exista numere
print("{:03d}".format(12))
# numar int cu o latime minima (6 caractere in cazul asta)
# precizie de 2 zecimale
# umpluta cu 0 daca nu exista numere
print("{:06.2f}".format(12.2346))
OUTPUT:
      1
    12.235
012
012.23
```

```
# numere int cu aliniere la dreapta
print("{:5d}".format(10))
# numere float cu aliniere centrala si precizie de 3 zecimale
print("{:^10.4f}".format(13.131278))
# integer left alignment filled with zeros
# numere int cu aliniere la stanga si spatiu umplut cu 0
print("{:<05d}".format(10))
# numere foat cu aliniere centrala
print("{:=8.3f}".format(-1.2346))
OUTPUT:
   10
 13.1313
10000
-   1.23
```

Metoda `format()` este foarte importantă în modificarea modului de afișare al șirurilor de caractere folosind funcția `print()`, cele două putând fi utilizate în aceeași instrucțiune, sintaxa Python permițând integrarea armonioasă a diverselor variabile Python în text. Funcția oferă o modalitate versatilă de a construi output textual și permite manipularea șirurilor de caractere

complexe. Sintaxa generală este prezentată în continuare, menționând faptul că valoarea fiind injectată automat între cele două acolade, respectând modul de formatare indicat:

```
'{}'.format(valoare, formatare)
```

```
mesaj = "Conductivitatea {0} are valoarea {1:3.1f} W/mp*K".format("caramida", 45.576)
print(mesaj)
OUTPUT:
Conductivitatea caramida are valoarea 45.6 W/mp*K
```

Cum funcționează? Variabila `mesaj` este compusă din nouă părți: stringul șablon: (`"Conductivitatea {0} are valoarea {1:3.1f} W/mp*K"`) și funcția `format()` utilizată ca o metodă și care indică valorile – argumentele poziționale ale metodei – șirul de caractere `"caramida"` și numărul flotant `45.576`. Acestea sunt injectate în locul indecșilor indicați în șablon: `0` și `1`, așa cum este reprezentat în figura 12. Mai mult, codul de formatare al numărului flotant se indică în șablon: `3.1f` – minimum `3` spații va ocupa variabila care va fi injectată acolo (`45.576`), o precizie de `1` zecimală (`45.576` – rotunjit la o zecimală: `45.6`) specificându-se că numărul este un `f` (`float`). Pentru șirul de caractere nu este indicat niciun cod de formatare, acesta fiind inserat așa cum este indicat în metoda `format()`.

```
mesaj = "Conductivitatea {0} are valoarea {1:3.1f} W/mp*K".format("caramida", 45.576)
```

Figură 12. Injectarea datelor în șablon

Același rezultat se obține și folosind argumente `keyword`, care, în locul indicării indexului, necesită specificarea unui nume pentru variabila injectată, funcționând pe principiul dicționarelor Python – `key: value`, unde cheia este în șablon și valoarea în metoda `format()`.

```
mesaj = "Conductivitatea {material} are valoarea {valoare:3.1f} W/mp*K"
.format(material="caramida", valoare=45.576)
print(mesaj)
OUTPUT:
Conductivitatea caramida are valoarea 45.6 W/mp*K
```

În șablonul `mesaj` indecșii sunt înlocuiți cu numele variabilelor `material` și `valoare`, denumiri ce se regăsesc apoi specificate și în metoda `format()`. La rândul lor, în momentul afișării rezultatului, numele variabilelor sunt înlocuite cu valorile specificate în `format()`, și merită menționat faptul că acolo se atribuie valori variabilelor din șablon. Sintaxa Python permite posibilitatea de a se folosi o mixtură de argumente în aceeași declarație, așa cum este prezentat în exemplele următoare.

```
# argumente nespecificate - implicite
print('Titul cartii: {} in {}'.format('I.A.', 'I.E.'))
# argumente pozitionale
print('Titul cartii: {0} in {1}'.format('I.A.', 'I.E.'))
# argumente kwyword
print('Titul cartii: {nume1} in {nume2}'.format(nume1='I.A.', nume2='I.E.'))
# mixt de argumente
print('Titul cartii: {0} in {nume}'.format('I.A.', nume='I.E.'))
OUTPUT:
I.A. in I.E.
I.A. in I.E.
I.A. in I.E.
I.A. in I.E.
```

O alternativă la utilizarea metodei `format()`, mai recent implementată în sintaxa Python (introdusă în varianta 3.6), este utilizarea `f-string` – prescurtare de la șir de caractere formatat – și care are avantajul unei sintaxe mai simple cu ajutorul prefixului `f` înaintea stringului (în acest fel valorile variabilelor sunt plasate între acolade):

```
f'{variabila}'
```

`f-string` prezintă avantajul unei scrieri mai concise, unei lizibilități crescute a codului scris și, conform unor dezvoltatori, este mai rapid decât utilizarea funcției `format()`, dezavantajele constând în anumite limitări în formatarea șirului de caractere și în faptul că nu este disponibil în versiuni mai vechi de 3.6. Un alt avantaj este reprezentat de faptul că este posibil a interpola orice între acoladele unui `f-string` de la simple valori la expresii complexe, care vor fi evaluate în momentul execuției codului și rezultatul va fi returnat în locul lor.

În exemplul următor sunt definite două variabile (`grosime` și `conductivitate`) și sunt inițializate cu două valori de timp `int`. În interiorul `f-string` se introduc aceste variabile, dar, în același timp, se și execută două calcule simple: transformarea variabilei `grosime` din milimetri în metri și calculul efectiv al rezistenței termice, împărțind `grosime` la `conductivitate`. De observat că la execuția codului, în momentul în care se afișează valorile pe ecranul terminalului, doar rezultatele calculelor finale sunt returnate și interpolate în locul format din cele două acolade. Mai mult, se prezintă și modul de formatare a numărului flotant, model identic cu cel la utilizarea metodei `format()`: `grosime/1000/conductivitate:.4f` – numărul flotant se va afișa cu 4 zecimale după virgulă – în mod implicit s-ar afișa cu 2 zecimale.

```
grosime = 500 # grosimea in mm
conductivitate = 10 # conductivitatea in W/m/K
mesaj = f'Rezistenta termica este {grosime/1000} / {conductivitate} = 
{grosime/1000/conductivitate:.4f} W/mp/K'
print(mesaj)
OUTPUT:
Rezistenta termica este 0.5 / 10 = 0.0500 W/mp/K
```

Toate regulile de formatare discutate la funcția `format()` se aplică și la `f-string`!

```
string1 = 'a'
string2 = 'ab'
string3 = 'abc'
string4 = 'abcd'
print(f'{string1:>10}')
print(f'{string2:>10}')
print(f'{string3:>10}')
print(f'{string4:>10}')
OUTPUT:
         a
        ab
       abc
      abcd
```

f24. `frozenset(iterabil)` generează un obiect imutabil (care **nu** poate fi modificat după ce a fost creat) dintr-un iterabil mutabil (care poate fi modificat după ce a fost creat, cum ar fi un set sau o listă). Dacă se încearcă a modifica un obiect `frozenset`, interpretorul Python va genera o eroare tip `TypeError: 'frozenset' object does not support item assignment`.

```
# creez o variaibla si ii atribui o lista neomogena
lista_mea = [1, 2, 'Nume', True]
print(f'Lista initiala este: {lista_mea }')
# modific lista creeata anterior
lista_mea [2] = "IA in IE"
print(f'Lista modificata este: {lista_mea }')
my_frozenset = frozenset(lista_mea)
print(f'Frozenset initial este: {my_frozenset}')
my_frozenset[2] = 'Nume'
OUTPUT:
Lista initiala este: [1, 2, 'Nume', True]
Lista modificata este: [1, 2, 'IA in IE', True]
Frozenset initial este: frozenset({1, 2, 'IA in IE'})
---> 10 my_frozenset[2] = 'Nume'

TypeError: 'frozenset' object does not support item assignment
```

Obiectul creat, `frozenset`, se comportă ca o mulțime de elemente imutabilă, acceptând toate operațiile care se pot realiza pe mulțimi de date: copiere, diferență, intersecție, reuniune și diferență simetrică. Toate acestea se efectuează utilizând metode specifice pentru acest tip de dată: `copy()`, `union()`, `intersection()`, `difference()`, `symetric_different()`. Trebuie menționat că aceste metode se folosesc pe instanțele create din funcția `frozenset()` și nu pe funcție în sine. În fragmentul de cod următor sunt prezentate exemple cu aceste metode.

```
# se creeaza cele doua instante frozenset din liste cu elemente de tip integer
set_A = frozenset([1, 3, 5, 7])
set_B = frozenset([2, 4, 6, 8])
print(f'A: {set_A}')
print(f'B: {set_B}')
# se copiaza frozenset A si se atribuie variabilei set_C
set_C = set_A.copy()
print(f'C: {set_C}. Este identic cu A? {set_A is set_C}')
# uniunea celor doua frozenseturi initiale
set_D = set_A.union(set_B)
print(f'A reunit B = {set_D}')
# intersectia celor doua frozenseturi initiale
set_E = set_A.intersection(set_B)
print(f'A intersectat cu B = {set_E}')
# diferenta celor doua frozenseturi initiale
print(f'A - B = {set_A.difference(set_B)}')
# diferenta simetrica dinre doua frozenseturi initiale
print(f'A - B = {set_A.symmetric_difference(set_B)}')
OUTPUT:
A: frozenset({1, 3, 5, 7})
B: frozenset({8, 2, 4, 6})
C: frozenset({1, 3, 5, 7}). Este identic cu A? True
A reunit B = frozenset({1, 2, 3, 4, 5, 6, 7, 8})
A intersectat cu B = frozenset()
A - B = frozenset({1, 3, 5, 7})
A - B = frozenset({1, 2, 3, 4, 5, 6, 7, 8})
```

Adițional se pot utiliza metodele:

`A.isdisjoint(B)` – returnează `True` dacă niciun item din `A` nu se regăsește în `B`.

`A.issubset(B)` – returnează `True` dacă setul `A` este un subset al setului `B`.

`A.issuperset(B)` – returnează `True` dacă setul `A` este super-setul setului `B`.

`f25. getattr(obiect, atribut, default)` returnează valoarea atributului sau a metodei indicate din obiectul specificat. Face parte din aceeași categorie cu `delattr()` – descris anterior (f14), `hasattr()` – descrisă la (f27) `setattr()` – . Parametrul `default` se utilizează drept alternativă la un element care nu a fost găsit. Lipsa specificării lui va genera o eroare în cazul în care elementul nu există definit în clasa analizată. Metoda este echivalentă cu modul de obținere a datelor direct din instanță (`instanta.metoda()` și `instanta.atribut`), dar oferă în plus posibilitatea de a folosi parametrul `default`.

În exemplul următor se creează o clasă cu ajutorul căreia se poate calcula sarcina termică în funcție de 4 parametri specificați cu ajutorul metodei constructor `__init__()`. Suplimentar, sunt definite un atribut `tip_aparat` care ia valoarea `'schimbator de caldura tubular'` și o metodă `calcul_sarcina(self)` care va returna valoarea sarcinii termice a unui aparat specific. După instanțierea clasei se încearcă obținerea atributului, a metodei și a unui atribut specificat intenționat greșit, caz în care, specificându-se un mesaj prin intermediul argumentului `default`, acesta va fi afișat în locul unei erori (ultimul exemplu).

```
# se defineste o clasa care care:
# un atribut - tip aparat
# un constructor - __init__
# o metoda - calcul_sarcina()
class sarcinaTermica():
    tip_aparat = 'schimbator de caldura tubular'
    def __init__(self, debi_masic, caldura_specifica, t1, t2):
        self.debit_masic = debi_masic
        self.caldura_specifica = caldura_specifica
        self.t1 = t1
        self.t2 = t2

    def calcul_sarcina(self):
        return (self.debit_masic*self.caldura_specifica*(self.t1 -
self.t2))/1000
# se instantiaza clasa prin obiectul sarcina folosind date specificate in
constructor
sarcina = sarcinaTermica(45, 4780, 55, 35)
print(sarcina.calcul_sarcina())
# se obtine atat atributut cat si metoda si se afiseaza pe ecran
schimbator = getattr(sarcinaTermica, 'tip_aparat')
print(schimbator)
calcul = getattr(sarcinaTermica, 'calcul_sarcina')
print(calcul)
# se incearca obtinerea unui atribut care nu exista
# va fi afisat mesajul din default!!!
model = getattr(sarcinaTermica, 'model_aparat', 'Nu exista atribut/metoda cu
numele speciticat')
print(model)
model2 = getattr(sarcinaTermica, 'model_aparat')
OUTPUT:
4302.0
schimbator de caldura tubular
<function sarcinaTermica.calcul_sarcina at 0x0000020536A18540>
Nu exista atribut/metoda cu numele speciticat
28 model2 = getattr(sarcinaTermica, 'model_aparat')

AttributeError: type object 'sarcinaTermica' has no attribute 'model_aparat'
```

`f26.` `globals()` returnează un dicționar cu toate variabilele globale și simbolurile folosite în programul curent, incluzând numele variabilelor, numele metodelor, claselor, funcțiilor etc. Interpretorul Python conține un tabel cu simbolurile globale în care sunt conținute informațiile necesare despre programul care este curent scris, pe lângă tabelul cu simbolurile locale.

`f27.` `hasattr(obiect, atribut)` returnează `True` dacă `obiectul` sau instanța analizat/ă are `atributul` specificat și `False` altfel. E un mod de a interoga o clasă prin intermediul unui obiect pentru a vedea dacă aceasta are sau nu un atribut la un moment dat în dezvoltarea și/sau exploatarea programului. Această necesitate apare datorită caracterului dinamic al programării utilizând limbajul de programare Python și care permite modificarea obiectelor prin adăugarea

atributelor și metodelor necesare după ce acestea au fost instanțiate din clasa inițială. Metoda se poate utiliza și pe clasa în sine și este o alternativă mai lentă, dar mult mai lizibilă, la analiza cu blocul de excepții `try - except`. În exemplul următor se pleacă de la clasa `sarcinaTermica()` definită la f25 și se utilizează metoda atât pe clasă cât și pe metoda `sarcina`.

```
nume_atribut = 'model aparat'
print(f'Clasa {sarcinaTermica} are atributul "{nume_atribut}"?',
hasattr(sarcinaTermica, 'model_aparat'))
nume_metoda = 'calcul_caldura_specifica'
print(f'Instanta {sarcina} are metoda "{nume_metoda}"?', hasattr(sarcina,
nume_metoda))
OUTPUT:
Clasa <class '__main__.sarcinaTermica'> are atributul "model aparat"? False
Instanta sarcinaTermica are metoda "calcul_caldura_specifica"? False
```

`f28. hash(obiect)` returnează valoarea hexazecimală a obiectului introdus ca atribut – dacă acesta are una. Valoarea hash a unui obiect reprezintă o valoare numerică de lungime fixă, unică, folosită pentru a identifica date intern, reprezentând, de fapt, semnătura digitală a datelor. Doar obiectele imutabile în Python au asociată o valoare hash.

```
# se creeaza o lista si se atribuie variabilei denumite lista_mea
# listele sunt obiecte mutabile - pot fi modificate dupa creare
list_mea = [1, 23, 'lista mea', True]
# se creeaza un numar float si se atribuie variabilei numarul_meu
# numerele sunt obiecte imutabile - nu se pot modifica dupa createa
numarul_meu = 12.42
print(f'{numarul_meu} are valoaraea hash: {hash(numarul_meu)}')
print(f'{list_mea} are valoaera hash: {hash(list_mea)}')
OUTPUT:
12.42 are valoaraea hash: 968454063869751308
----> 9 print(f'{list_mea} are valoaera hash: {hash(list_mea)}')

TypeError: unhashable type: 'list'
```

`f29. help(obiect)` execută sistemul încorporat de help al limbajului de programare Python. Funcția a fost folosită și anterior când s-au analizat diverse module și posibilități de afișare a funcționalității acestora. În exemplul următor, clasa `sarcinaTermica()` a fost modificată puțin prin adăugarea unui comentariu tip documentație – v. § 3. Particularități de sintaxă a limbajului de programare Python. În urma rulării instrucțiunii `print(help(sarcinaTermica))`, rezultatul este afișat în fragmentul de cod următor. Merită menționat faptul că textul ce conține comentariul de tip documentație este următorul:

```
"""
clasa sarcinaTermica este utilizata pentru a instantierea schimbatoarelor de
caldura necesita la constructor 4 argumente: debit masic, caldura specifica
fluid si temperaturi are o metoda princ are calculeaza sarcina termica: Q = m
cp/1000 delta(T) [kW]
"""
print(help(sarcinaTermica))
```

```
OUTPUT:
Help on class sarcinaTermica in module __main__:

class sarcinaTermica(builtins.object)
 |  sarcinaTermica(debi_masic, caldura_specifica, t1, t2)
 |
 |  clasa sarcinaTermica este utilizata pentru a instantierea schimbatoarelor
de caldura
 |  necesita la constructor 4 argumente: debit masic, caldura specifica fluid
si temperaturi
 |  are o metoda princ are calculeaza sarcina termica: Q = m cp/1000 delta(T)
[kW]
 |
 |  Methods defined here:
 |
 |  __init__(self, debi_masic, caldura_specifica, t1, t2)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __str__(self) -> str
 |      Return str(self).
 |
 |  calcul_sarcina(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  tip_aparat = 'schimbator de caldura tubular'
```

OBSERVAȚIE. A se observa faptul că numele clasei/funcției este scris fără paranteze, indicându-se doar numele: `sarcinaTermica` (sau `print`). Acest lucru însemnă faptul că această clasă nu este apelată – adică nu se creează o instanță, caz în care ar fi necesară introducerea valorilor pentru cele 4 argumente specificate în constructorul `__init__()`.

Un alt exemplu de utilizare a funcției `help()` este redat în continuare, când se apelează această funcție pentru a afișa pe ecranul consolei informații despre funcția simplă `help()`. În textul generat se indică faptul că funcția face parte din categoria funcțiilor încorporate și se găsește în modulul `builtins`. Mai mult, se afișează sintaxa generală completă și sunt descriși parametrii funcționali (`sep`, `end`, `file` și `flush`). În mod similar, se pot analiza toate tipurile de funcții, clase, date, variabile existente în Python sau create de programator.

```
print(help(print))
OUTPUT:
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

f30. `hex(numar)` returnează valoarea hexazecimală a unui număr sub forma unui șir de caractere cu prefixul `0x`. Sistemul hexazecimal este un sistem de numerotare în baza 16 utilizat în mod frecvent în programarea calculatoarelor. Argumentul funcției trebuie neapărat să fie de timpul `int`, altfel funcția va genera o eroare te tip: `TypeError: 'float' object cannot be interpreted as an integer`. Dacă totuși apare necesitatea găsirii numărului hexazecimal pentru un număr de tip `float`, se poate utiliza o metodă specifică acestui tip de dată: `float.hex(numar_float)`. A se observa faptul că acum, metoda `hex` este o metodă specifică clasei `float` și nu mai face parte din modulul `builtins`. Utilizând funcția `help()` sunt generate inclusiv exemple de utilizare.

```
print("Help pentru hex")
print(help(hex))
print("Help pentru float.hex")
help(float.hex)
OUTPUT:
Help pentru hex
Help on built-in function hex in module builtins:

hex(number, /)
    Return the hexadecimal representation of an integer.

    >>> hex(12648430)
    '0xc0ffee'
Help pentru float.hex
Help on method_descriptor:

hex(self, /)
    Return a hexadecimal representation of a floating-point number.

    >>> (-0.1).hex()
    '-0x1.999999999999ap-4'
    >>> 3.14159.hex()
    '0x1.921f9f01b866ep+1'
```

f31. `id(obiect)` returnează id-ul (identification) unic al `obiectului` specificat ca argument. Absolut toate obiectele Python au un id unic care este atribuit automat la crearea obiectului

propriu-zis. De fapt, id-ul unui obiect este adresa din memoria internă a calculatorului pe care acel obiect o ocupă și este generată aleatoriu – însemnând că la fiecare rulare a programului această adresă va fi alta (excepție fac numerele de timp întreg care au o adresă constantă în memorie). Id-ul unui obiect este de tip număr întreg, așa cum reiese din exemplul următor.

```
# se creaza o lista si se initializeaza cu 5 date de tip sir de caractere
list_bibliografica = [
    'Davenport, T. (2012). Harvard Business Review.',
    'docs.python.org. (2023). Python tuturial.',
    'Parks, D. M. (2017). Defining Data Science and Data Scientist',
    'Python Succes Stories. (2023)',
    'VanderPlas, J. (2017). Python Data Science Handbook'
]
# se afiseaza id-ul de aceastei instante a clasei list
print(id(list_bibliografica))
print('Tip data: ', type(id(list_bibliografica)))
OUTPUT:
1653770788224
Tip data: <class 'int'>
```

OBSERVAȚIE. Cum în limbajul de programare Python numele variabilelor nu sunt altceva decât pointeri către zone de memorie în care sunt stocate diferite tipuri de date, există posibilitatea ca două sau mai multe variabile să indice către aceeași zonă de memorie! În exemplul următor este evidențiat un asemenea caz. Pentru lizibilitate, id-ul este transformat în număr hexazecimal.

```
# se creaza o variabila de tip int si se atribuie variabilei numar
numar = 21
# variaibla numar se atribuie si variabilei numar2, care va ocupa acelasi loc
in memorie
numar2 = numar # numar 2 va indica tot obiectul int 21
# variabila numar3 va indica tot spre obiectul int 21
numar3 = 21
print('locatia pentru variabila numar - ', hex(id(numar)))
print('locatia pentru variabila numar2 - ', hex(id(numar2)))
print('locatia pentru variabila numar3 - ', hex(id(numar3)))
print('Sunt locatiile din memorie identice?',
id(numar) == id(numar2) == id(numar3))
OUTPUT:
locatia pentru variabila numar -  0x7ffb61c3e5a8
locatia pentru variabila numar2 -  0x7ffb61c3e5a8
locatia pentru variabila numar3 -  0x7ffb61c3e5a8
Sunt locatiile din memorie identice? True
```

`f32. input(prompt)` permite utilizatorului să introducă date și să le trimită înapoi către programul Python, fiind o funcție extrem de importantă încorporată în sintaxă. Parametrul `prompt` (opțional) este un șir de caractere cu rolul de a furniza un mesaj înaintea inputului așteptat. Important de reținut este faptul că funcția returnează un șir de caractere indiferent de

tipul de dată introdus, făcând mereu conversia internă string – dată cu ajutorul constructorului implicit `str()` – detaliat la punctul ... Practic, `input()` preia inputul utilizatorului și îl returnează programului, creându-se astfel premisele interacțiunii program-utilizatori. În momentul utilizării funcției `input()`, rularea programului este oprită și reluată doar în momentul apăsării tastei `Enter`, moment în care interpretorul Python înțelege faptul că datele au fost introduse de la tastatură și va fi reluată execuția programului de pe linia următoare.

În exemplul de cod următor se creează o variabilă cu ajutorul căreia se stochează data introdusă de la tastatură. Deși este cerut un număr (fără a specifica tipul acestuia), la analiza tipului de dată returnat de funcția `input()` se poate observa faptul că aceasta returnează un obiect din clasa `<class 'str'>`, clasă specifică șirului de caractere. Acest lucru este foarte important, deoarece poate sta la baza erorilor de sintaxă. Spre exemplu, la încercarea de a folosi numărul introdus de la tastatură într-o expresie matematică se va genera o eroare de tip `TypeError: can only concatenate str (not "int") to str` – ce spune că nu este posibilă concatenarea a două tipuri de date diferite – `str` și `int` în acest caz!

```
numar = input("Introduceti numarul aici: ")
print("Numarul introdus este:  ", numar)
print(type(numar))
OUTPUT:
Introduceti numarul aici: 2 #se asteapta apasarea tastei Enter
Numarul introdus este:   2
<class 'str'>
```

Pentru a depăși această posibilă sursă de eroare, întotdeauna trebuie ca data introdusă de la tastatură să fie convertită explicit la tipul de dată dorit utilizând funcțiile constructor încorporate în sintaxa Python, cum ar fi `float()`, `int()`, `list()` etc. Acest lucru se poate realiza introducând rezultatul funcției `input()` drept argument la funcția constructor dorită:

```
numar = float(input("Introduceti numarul aici: "))
print("Numarul introdus este:  ", numar)
print(numar + 132)
OUTPUT:
Introduceti numarul aici: 2 #se asteapta apasarea tastei Enter
Numarul introdus este:   2.0
134.0
```

`f33. int(valoare, base)` returnează un număr de timp întreg – `int` din numărul trecut ca argument. Este, în esență, constructorul obiectelor de tip `int`. Opțional, mai există un parametru, denumit `base`, și care permite specificația formatului (sub forma unui int) în care se dorește transformarea – implicit este baza `10`, dar se pot introduce și următoarele variante de baze: 0, 2, 8, 10, 16. Se poate utiliza funcția constructor și pentru a genera numere de tip `int` din `string`, așa cum este exemplificat în blocul de cod următor, cu mențiunea că numărul literal din string trebuie să fie număr întreg, altfel va genera o eroare de tip `ValueError: invalid literal for int() with base 10: '14.56'` (exemplul pentru încercarea conversiei numărului literal de timp float `14.56`'). În schimb, sintaxa `int("15")`, unde argumentul este număr literal întreg este legală și va returna numărul de tip `int` și de valoare `15`.

```
numar_float = 12.995
print('Conversia float-int: ', int(numar_float))
print(int("14.56"))
OUTPUT:
Conversia float-int:  12
----> 4 print(int("14.56"))

ValueError: invalid literal for int() with base 10: '14.56'
```

OBSERVAȚIE. Merită observat faptul că funcția constructor `int()` NU rotunjește numărul la prima valoare întreagă ci TRUNCHIAZĂ partea întreagă de partea zecimală a numărului flotant. Practic șterge tot ce este după virgulă, indiferent de cât de aproape este de numărul superior. Suplimentar se poate specifica și baza în care se dorește să se realizeze conversia.

```
# conversia unui string (in format binar) in int
print("numarul binar 0b111 in int este:", int("0b111", 2))
# conversia unui string (in format octal) in int
print("Numarul octal 0o162 in int este:", int("0o162", 8))
# # conversia unui string (in format hexazecimal) in int
print("Numarul hexazecimal 0xA1 in int este:", int("0xA1", 16))
OUTPUT:
numarul binar 0b111 in int este: 7
Numarul octal 0o162 in int este: 114
Numarul hexazecimal 0xA1 in int este: 161
```

`f34. isinstance(obiect, tip_data)` returnează `True` doar dacă `obiectul` este de `tipul` specificat, altfel returnează `False`. Parametrul `tip_data` poate fi clasă, tip de dată Python sau tuplu cu clase sau tipuri de date. Dacă este introdus sub forma unui tuplu, atunci funcția returnează `True` dacă obiectul este cel puțin un element al tuplului. Se poate verifica astfel apartenența diverselor obiecte/instanțe la diverse clase. Spre exemplu, apartenența unui număr la clasa `int`.

```
# se primeste un numar de la tastatura si se atribuie variabilei
# se introduce 123 de la tastatura
numar_intreg = int(input("Introdu un numar intreg: "))
# se verifica daca numarul de la tastatura este de tip int
print(f"Numarul {numar_intreg} este de tip int?",
isinstance(numar_intreg, int))
OUTPUT:
Numarul 123 este de tip int? True
```

```
numar_test = 14.5
apartenenta = isinstance(numar_test, (int, float, complex))
print(f"Apartenenta este {apartenenta}")
str_test = "Sunt o instanta de tip str"
apartenenta = isinstance(str_test, (int, float, complex))
print(f"Apartenenta este {apartenenta}")
OUTPUT:
Apartenenta este True
Apartenenta este False
```

Deși aparent realizează același lucru cu funcția `type()`, există câteva diferențe fundamentale descrise succint în tabelul următor:

`isinstance()`	`type()`
Sintaxa: `isinstance(object, tip_data)`	Sintaxa: `type(object)`
Verifică dacă instanța aparține clasei specifice – returnează `True sau False`	Returnează clasa din care obiectul face parte
Poate verifica dacă obiectul face parte atât dintr-o clasă cât și dintr-o subclasă	Nu poate face asta
Mai rapidă comparat cu `type()`	`Mai înceată decât isinstance()`
Poate verifica mai multe clase simultan	Nu poate face asta
Exemplu: `isinstance(10, (int, str))`	**Exemplu:** `type(10)`

`f35.` **`issubclass`**`(clasa, subclasa)` returnează `True` dacă obiectul `clasa` este o subclasă a clasei specificate de parametrul `subclasa`. Aceasta funcție verifică, de fapt, dacă `obiectul` a moștenit elemente de la clasa părinte într-un concept denumit `inheritance` (moștenire) specific domeniului programării orientate pe obiecte. Cu ajutorul acestui concept funcționalitatea (reprezentată de atributele și metodele clase) unei clase – clasă părinte este transferată către o nouă clasă – clasă copil, fără a exista restricții asupra modificării funcționalității inițiale. Astfel se asigură reutilizarea codului și principiul DRY. Aceste concepte țin de programarea avansată și nu fac obiectul acestei lucrări.

`f36.` **`iter`**`(obiect, sentinel)` returnează un obiect tip iterator dintr-un `obiect` iterabil (care poate fi parcurs element cu element). Se folosește în conjuncție cu funcția **`next`**`(obiect)` – a se citi despre funcția f43 – pentru a genera următorul element din secvență. Parametrul `sentinel` este opțional și este o valoare specifică care va reprezenta finalul secvenței generate. În acest caz, se vor genera elemente până când parametrul `sentinel` este atins. Iteratorii au o proprietate foarte importantă, și anume că pot fi utilizați pentru a parcurge obiectul o singură dată, aceștia având implementat un contor intern care memorează iterația curentă și nu permite repornirea de la 0 în cadrul aceleiași rulări.

```
# se initializeaza o lista ce contine primele 5 numere pare
lista_pare = [2, 4, 6, 8, 10]
print("The list is of type : ", type(lista_pare))
# iterare prin lista
for numar_par in lista_pare:
    print("numarul curent este: ", numar_par)
# se converteste lista cu in iterabil
iter_pare = iter(lista_pare)
print("The iterator is of type : ", type(iter_pare))
# se itereaza prin iterabil
print(next(iter_pare))
for numar_par in iter_pare:
    print(f'numarul curent este: {numar_par}')
    print(next(iter_pare))
OUTPUT:
The list is of type :  <class 'list'>
numarul curent este:  2
```

```
numarul curent este:  4
numarul curent este:  6
numarul curent este:  8
numarul curent este:  10
The iterator is of type :  <class 'list_iterator'>
2
numarul curent este: 4
6
numarul curent este: 8
10
```

f37. `len(iterabil)` returnează numărul de elemente dintr-un iterabil (numărul de elemente dintr-o listă, dintr-un tuplu, numărul de perechi chei-valori dintr-un dicționar, numărul de caractere dintr-un șir de caractere etc.). Datele iterabile în Python pot fi de două tipuri: secvențiale (`string`, `byte`, `tuplu`, `list`, `range`) sau colecții (`dictionar`, `set`, `frozenset`). Orice obiect iterabil este caracterizat de numărul de elemente pe care îl conține și acesta se obține cu ajutorul funcției `len()`, care acceptă un singur parametru: iterabilul.

```
# se genereaza o lista cu numere impare mai mici sau egale cu 14
lista_impare = [item for item in range(14) if item%2 != 0]
print(lista_impare)
# se genereaza numarul de elemente si rezultatul se atribuie variabilei
numar_elemente
numar_elemente = len(lista_impare)
print(f'lista curenta are {numar_elemente} elemente')
OUTPUT:
[1, 3, 5, 7, 9, 11, 13]
lista curenta are 7 elemente
```

În exemplul următor se utilizează funcția `len()` pentru a genera numărul de elemente dintr-un obiect secvențial de tip `string`. Merită menționat faptul că funcția returnează toate elementele din obiectul `string`, incluzând aici caractere goale (spațiile albe) și semnele de punctuație – fără a contoriza și ghilimelele care formează șirul de caractere.

```
# se genereaza un text si se atribuie unei variabile
text_test = 'Text de test!'
# se afiseaza lungimea sirului de caractere
print(f'sirul de caractere "{text_test}" are {len(text_test)} caractere.')
OUTPUT:
sirul de caractere "Text de test" are 13 caractere.
```

f38. `list(iterabil)` este funcția constructor pentru crearea listelor în Python. Parametrul `iterabil` poate fi o dată secvențială, o colecție sau un interator și este opțional – funcția returnează o listă goală dacă nu îi este furnizat niciun argument. Funcția returnează un obiect tip listă de elemente. O alternativă la acest constructor este utilizarea parantezelor drepte `[]`.

În exemplul următor sunt generate mai multe liste: o listă goală utilizând funcția constructor `list()`, o listă goală utilizând parantezele drepte, o listă rezultată în urma utilizării unei funcții generatoare obiecte iterative `range()` și o listă generată dintr-un obiect tip șir de caractere. De menționat faptul că în ultimul caz, stringul va fi descompus pe caracterele constituente și acestea vor popula lista creată.

```
# se genereaza o lista goala
lista_goala1 = list()
print(lista_goala1)
lista_goala2 = []
print(lista_goala2)
# se genereaza o lista utilizand funcia iterator range()
lista_range = list(range(3))
print(lista_range)
# va imparti sirul de caractere pe caractere componente si va genera o lista
lista_str = list("I.A. in I.E.")
print(lista_str)
OUTPUT:
[]
[]
[0, 1, 2]
['I', '.', 'A', '.', ' ', 'i', 'n', ' ', 'I', '.', 'E', '.']
```

`f39.` `locals()` returnează un dicționar cu toate variabilele locale și simbolurile folosite în programul curent, incluzând numele variabilelor, numele metodelor, claselor, funcțiilor etc. Interpretorul Python conține un tabel cu simbolurile locale în care sunt conținute informațiile necesare despre programul care este curent scris, pe lângă tabelul cu simboluri globale (v. și `globals()` - `f26`).

`f40.` `map(funcție, iterabil)` returnează un obiect de tip `map` format din maparea `funcției` pe fiecare element din `iterabil`. Practic, o funcție definită de utilizator sau încorporată în sintaxa Python este aplicată fiecărui element din obiectul iterabil furnizat. Are același efect cu utilizarea unei bucle iterative `for`, dar este mult mai lizibilă, scurt de scris și oferă câteva funcționalități în plus. Parametrii sunt:

- `funcție` – necesar. O funcție care va fi executată pentru fiecare element din iterabil;
- `iterabil` – necesar. O dată secvențială, colecție sau iterator.

```
# se creeaza o lista initiala si se atribuie variabilei numere
numere = [1, 3, 5, 7, 9]
# se defineste o functie simpla care returneaza patratul unui numar
def numar_patrat(numar):
    return numar ** 2

# se foloseste functia map() pentru a mapa funcia definita
# fiecarui element din iterabil
numere_patrate = map(numar_patrat, numere)
# se afiseaza rezultatul maparii - va fi un obiect de tip map
print(numere_patrate)
# se afiseaza rezultul dupa ce a fost convertit intr-o lista
print(f'lista unitizand "map()": {list(numere_patrate)}')

# pentru comparatie, se utilizeaza bucla for pentru a realiza acelasi lucru
numere_patrate = []
for numar in numere:
    numere_patrate.append(numar_patrat(numar))
print(f'lista unitizand bucla "for": {list(numere_patrate)}')
OUTPUT:
<map object at 0x00000228064878B0>
lista unitizand "map()": [1, 9, 25, 49, 81]
lista unitizand bucla "for": [1, 9, 25, 49, 81]
```

Funcția utilizată poate fi inclusiv o funcție `lambda`, după cum este prezentat în continuare. Mai mult, se pot indica oricâți parametri de top `iterabil`, dar `funcția` trebuie să aibă același număr de parametri proprii. În al doilea exemplu sunt definite două liste care conțin date numerice și se utilizează funcția `map()` care aplică o funcție `lambda` pentru aceste liste. Trebuie observat faptul că funcția `lambda` necesită două argumente și returnează suma lor. Drept urmare, `map()` va aplica această logică și va aduna element cu element cele două liste.

```
# se creeaza un set de numere pare si se atribuie unei variaible
numere = (2, 4, 6, 8, 10)
# se utilizeaza o functie lambda
rezultat = map(lambda x: x+10, numere)

# conversia obiectul map intr-un obiect set utilizand constructorul set()
numere10 = set(rezultat)
print(numere10)
OUTPUT:
{12, 14, 16, 18, 20}
```

```
numere1 = [4, 5, 6]
numere2 = [7, 8, 9]

rezultat = map(lambda n1, n2: n1+n2, numere1, numere2)
print(list(rezultat))
OUTPUT:
[11, 13, 15]
```

f41. max(iterabil)/max(val1, val2, val3,...) returnează cea mai mare valoare dintr-un iterabil sau dintre două sau mai multe argumente separate de virgulă.

```
print("Maximul dintre 12 si 14 este: ", max(12, 14))

lista_numere = [9, 34, 11, -4, 27]
# gaseste numarul maxim din lista
numar_maxim = max(lista_numere)
print(f"Cel mai mare numar din lista {lista_numere} este: {numar_maxim}")
OUTPUT:
Maximul dintre 12 si 14 este: 14
Cel mai mare numar din lista [9, 34, 11, -4, 27] este: 34
```

În cazul în care iterabilul este compus din elemente de tip string, se va returna cel mai mare element din punctul de vedere al ordonării alfabetice a primei litere. În exemplul următor este definită o listă formată din trei elemente de timp string, pentru că aplică funcția max().

```
lista_str = ["Facultate", "Eneretica", 'Inteligenta', 'Z']
print(max(lista_str))
OUTPUT:
Z
```

Dacă se folosesc colecții sau iterabile neomogene (elemente de tipuri diferite) funcția va genera o eroare de tip TypeError: '>' not supported between instances of 'int' and 'str' , așa cum este cazul în care se dorește comparația între un int și un str. Suplimentar se poate utiliza un parametru opțional denumit key și cu ajutorul căruia se poate specifica o funcție încorporată (și nu numai) care să fie baza comparației când se dorește aflarea maximului. Luând exemplul precedent, maximul acelei liste a fost returnat în funcție de cea mai apropiată literă de sfârșitul alfabetului. În exemplul următor se va folosi pentru aceeași listă parametrul key specificând că se dorește ca lungimea elementului să fie comparată. Acum pentru fiecare element se va aplica funcția len() care va returna lungimea (v. f37) și apoi se vor compara toate aceste valori. În mod similar se pot folosi și alte funcții specifice.

```
lista_str = ["Facultate", "Eneretica", 'Inteligenta', 'Z']
print(max(lista_str, key=len))
OUTPUT:
Inteligenta
```

f42. min(iterabil)/min(val1, val2, val3,...) returnează cea mai mare valoare dintr-un iterabil sau dintre două ori mai multe argumente separate de virgulă. Analiza este analoagă cu funcția anterioară - min(), drept urmare nu va mai fi exemplificată.

f43. next(iterabil, default) returnează următorul element dintr-un iterabil. Funcția a fost folosită în conjuncție cu iter() – a se citi funcția f36. Parametrul iterabil este reprezentat de obiectul de tip iterabil pentru care se dorește a se genera succesiv valori, în timp ce default este un parametru opțional care specifică o valoare implicită ce este returnată de funcție în momentul în care iterația a ajuns la final. În exemplul următor se generează un iterabil utilizând

o listă ca argument al funcției `iter()` și se folosește funcția `iter()` pentru a genera valori succesive, urmând ca la final să fie generat un text de atenționare utilizând `default`.

```
# se creeaza o lista cu nume de persoane
lista_persoane = list(('Ana', "Andrei", 'Andreea'))
# se transforma lista intr-un obiect iterabil
iter_persoane = iter(lista_persoane)
# se creeaza o variabila de tip string ce va fi utilizata ca parametru default
parametru_default = 'Nu mai sunt elemente!'

# next() va genera prima valoare din iterabil
print(f"urmatoarea valoare din {lista_persoane} este {next(iter_persoane)}")
# next() va genera a doua valoare din iterabil
print(f"urmatoarea valoare din {lista_persoane} este {next(iter_persoane)}")
# next() va genera a doua valoare din iterabil
print(f"urmatoarea valoare din {lista_persoane} este {next(iter_persoane)}")
# next() va genera o eroare deoarece nu este indicat default
print(f"urmatoarea valoare din {lista_persoane} este {next(iter_persoane)}")
OUTPUT:
urmatoarea valoare din ['Ana', 'Andrei', 'Andreea'] este Ana
urmatoarea valoare din ['Ana', 'Andrei', 'Andreea'] este Andrei
urmatoarea valoare din ['Ana', 'Andrei', 'Andreea'] este Andreea
---> 16 print(f"urmatoarea valoare din {lista_persoane} este
{next(iter_persoane)}")

StopIteration:
```

În exemplul doi se inițializează argumentul `default`, inițializat cu variabila denumită `parametru_default` și care are ca valoare un obiect de timp string cu un mesaj de atenționare.

```
# next() va genera valoarea din parametrul default
print(f"urmatoarea valoare din {lista_persoane} este {next(iter_persoane, pa-
rametru_default)}")
# next() va genera valoarea din parametrul default
print(f"urmatoarea valoare din {lista_persoane} este {next(iter_persoane,
parametru_default)}")
# next() va genera valoarea din parametrul default
print(f"urmatoarea valoare din {lista_persoane} este {next(iter_persoane,
parametru_default)}")
OUTPUT:
urmatoarea valoare din ['Ana', 'Andrei', 'Andreea'] este Nu mai sunt elemente!
urmatoarea valoare din ['Ana', 'Andrei', 'Andreea'] este Nu mai sunt elemente!
urmatoarea valoare din ['Ana', 'Andrei', 'Andreea'] este Nu mai sunt elemente!
```

`f44. object()` returnează un obiect fără caracteristici, fiind constructorul explicit al obiectelor Python. Nu se pot adăuga proprietăți sau metode unui obiect creat cu această funcție constructor, dar acesta conține toate proprietățile și metodele încorporate care sunt utilizate de absolut toate clasele și obiectele Python. Nu are parametri și nu este în mod uzual folosită.

```
obiect_test = object()
print(type(obiect_test))
print(dir(obiect_test))
OUTPUT:
<class 'object'>
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
```

f45. `oct(int)` returnează un string ce reprezintă transformare octală a unui număr întreg (binar, zecimal sau hexazecimal) trecut ca argument. Toate numerele octale generate au prefixul `0o`. Dacă se încearcă utilizarea unui alt tip de dată în afară de număr int, funcția va genera o eroare de tip **TypeError**.

```
# int in octal
print('oct(13) este', oct(13))
# binar in octal
print('oct(0b101) is:', oct(0b101))
# hehazecimal in octal
print('oct(0XA) is:', oct(0XA))
print('Valoarea octala a lui "13" este', oct("13"))
OUTPUT:
oct(13) este 0o15
oct(0b101) is: 0o5
oct(0XA) is: 0o12
----> 7 print('Valoarea octala a lui "13" este', oct("13"))

TypeError: 'str' object cannot be interpreted as an integer
```

f46. `open(fișier, mod='r', buffering=1, encoding=None, errors=None, closefd=None, opener=None)` returnează un obiect tip `fișier` după ce l-a deschis. Există 4 metode (moduri) în care se por deschide fișierele în Python:

↔“r” (read) – deschide un fișier pentru a fi citit. Returnează eroare dacă fișierul nu există.
↔“a” (append) – deschide un fișier pentru a adăuga în el. Creează fișierul dacă nu există.
↔“w” (write) – deschide un fișier pentru a scrie în el. Creează fișierul dacă nu există.
↔“x” (create) – creează fișierul specificat. Returnează eroare dacă fișierul există deja.

Adițional se poate specifica modul în care va fi tratat fișierul:

↔“t” (text) - fișierul va ti tratat ca un text.
↔“b” (binary) - fișierul va fi tratat ca un obiect binar.

Cei mai utilizați parametri opționali sunt:

↔buffering (implicit = 1) - setarea politicii de buffering.
↔encoding - specificarea formatului de codificare a textului
↔error - specificarea modului de rezolvare a erorilor de codificare/decodificare a textului
↔newline - specificarea modului de abordarea a liniilor noi în text - se poate utiliza: None, ‘ ’, ‘\n’, ‘r’ și ‘\r\n’.

Sintaxa generală pentru a deschide un fișier în modul scriere (write) denumit "nume_fisier.txt" și care va fi tratat ca un text este:

```
f = open("nume_fisier.txt", 'wt')
```

f47. ord(caracter) returnează numărul care reprezintă codul unicode al caracterului ca argument, cu mențiunea că argumentul trebuie să fie trecut sub forma unui string. Funcția ord() este inversa funcției chr() – v. descrierea funcției f10.

```
print("Cifra specifica caracterului 'x' este: ", ord('x'))
print("Cifra specifica caracterului 'X' este: ", ord('X'))
OUTPUT:
Cifra specifica caracterului 'x' este:  120
Cifra specifica caracterului 'X' este:  88
```

f48. pow(x, y, z) returnează valoarea ridicării la pătrat a două numere specificate ca argument: x^y. Parametrii sunt: x – baza, y – exponentul, z (opțional) – mod. Spre exemplu, pow(2, 3, 3) este echivalent cu sintaxa: 23 % 3 – ridică 2 la cub, apoi calculează restul împărțirii rezultatului la 3, rezultatul final fiind 2. Pentru exemplificare, se propune următorul caz:

```
print(pow(2, 10) % 3)
print(pow(2, 10, 3))
print(pow(3, 12) % 5 == pow(3, 12, 5))
OUTPUT:
1
1
True
```

Funcția poate returna atât un număr de tip int cât și un float depinzând de semnul bazei și al exponentului, după următoarea logică și după cum este exemplificat în cele ce urmează:

- baza pozitivă / exponent pozitiv – obiect int
- baza pozitivă / exponent negativ – obiect float
- baza negativă / exponent negativ – obiect int
- baza negativă / exponent negativ – obiect float

```
print(f"baza + / exponent + este: { pow(2, 3)} - tip {type(pow(2, 3))}")
print(f"baza + / exponent - este: { pow(2, -3)} - tip {type(pow(2, -3))}")
print(f"baza - / exponent + este: { pow(-2, 3)} - tip {type(pow(-2, 3))}")
print(f"baza - / exponent - este: { pow(-2, -3)} - tip {type(pow(-2, -3))}")
OUTPUT:
baza + / exponent + este: 8 - tip <class 'int'>
baza + / exponent - este: 0.125 - tip <class 'float'>
baza - / exponent + este: -8 - tip <class 'int'>
baza - / exponent - este: -0.125 - tip <class 'float'>
```

f49. print(*obiects, separator, end, file, flush) afișează pe ecranul terminalului sau pe alt dispozitiv standard de afișare un mesaj specificat. Mesajul poate fi orice obiect Python, dar va fi convertit implicit la un string – șir de caractere de funcție. Aceasta a fost utilizată în

majoritatea exemplelor din această carte chiar dacă nu a fost detailată. Parametrii (și valorile lor implicite) sunt:
- `obiect` – orice obiect Python care va fi convertit la string și afișat. Se pot introduce oricât de multe obiecte separate prin virgulă;
- `sep=' '` (opțional) – specifică cum vor fi separate obiectele afișate;
- `end='\n'` (opțional) – specifică ce se va afișa la sfârșitul mesajului;
- `file=sys.stdout` (opțional) – specifică un obiect cu o metodă de scriere;
- `flush=False`.

OBSERVAȚIE. Argumentele opționale sunt de tipul keyword și, pentru a le putea modifica valorile implicite, numele lor trebuie să fie specificate în cadrul apelării funcției, altfel caracterul va fi considerat un obiect standard care va fi afișat pe ecranul consolei. În exemplul următor se modifică valoarea implicită a parametrului `sep`. În mod similar se pot modifica toți.

```
# se printeaza 3 obiecte de tipuri separate prin virgula
print("Inginerie", "Energetica", 2301)
# se printeaza 3 obiecte de tipuri diferite folosit ca separator
# caracterul “;” urmat de caracterul spatiu gol " "
print("Inginerie", "Energetica", 2301, sep='; ')
OUTPUT:
Inginerie Energetica 2301
Inginerie; Energetica; 2301
```

```
grupa = 2302
print("grupa =", grupa, sep='00000', end='\n\n')
print("grupa =", grupa, sep='0', end='')
OUTPUT:
grupa =000002302

grupa =02302
```

Obiectele se pot scrie și într-un fișier specificat prin intermediul argumentului `file`, dar pentru aceasta, inițial, trebuie să se creeze sau să se deschidă fișierul într-un mod în care este creat dacă nu există – v. `open()` detaliată la funcția f46. În exemplul următor se încearcă deschiderea unui fișier și, dacă nu există, se va crea în directorul curent de lucru (unde se află fișierul .py executat), după care se printează în el mesajul de tip string folosind funcția `print()` și specificând cu ajutorul argumentului `file` locul unde să fie afișat/scris obiectul. Merită observat faptul că în consola terminalului nu mai este afișat mesajul, dar în directorul de lucru va apărea un fișier cu extensia `.txt` care va conține mesajul. Dacă nu se folosește un manager de context (v. § „Cuvinte cheie de structură: `def`, `class`, `with`, `as`, `pass`, “ – paragraful specific cuvântului cheie `with`), fișierul va trebui să fie închis utilizând metoda `.close()`. Se exemplifică atât utilizarea `with` cât și cea fără.

```
# utilizare managerului de context with
with open('fisier_print.txt', 'w') as f:
        print(mesaj_print, file=f)
```

```
# se incearca deschierea unui fisier denumit fisier_print
# daca nu exista va fi creat - modul w (write)
fisier_sursa = open('fisier_print.txt', 'w')
# se creeaza un fisier tip string si se atribuie unei variabile
mesaj_print = 'Inteligenta Artificiala in Ingineria Energetica'
print(mesaj_print, file=fisier_sursa)
# pentru ca nu a fost deschis cu with, fisierul trebuie neaparat inchis!!!
fisier_sursa.close()
OUTPUT:
```

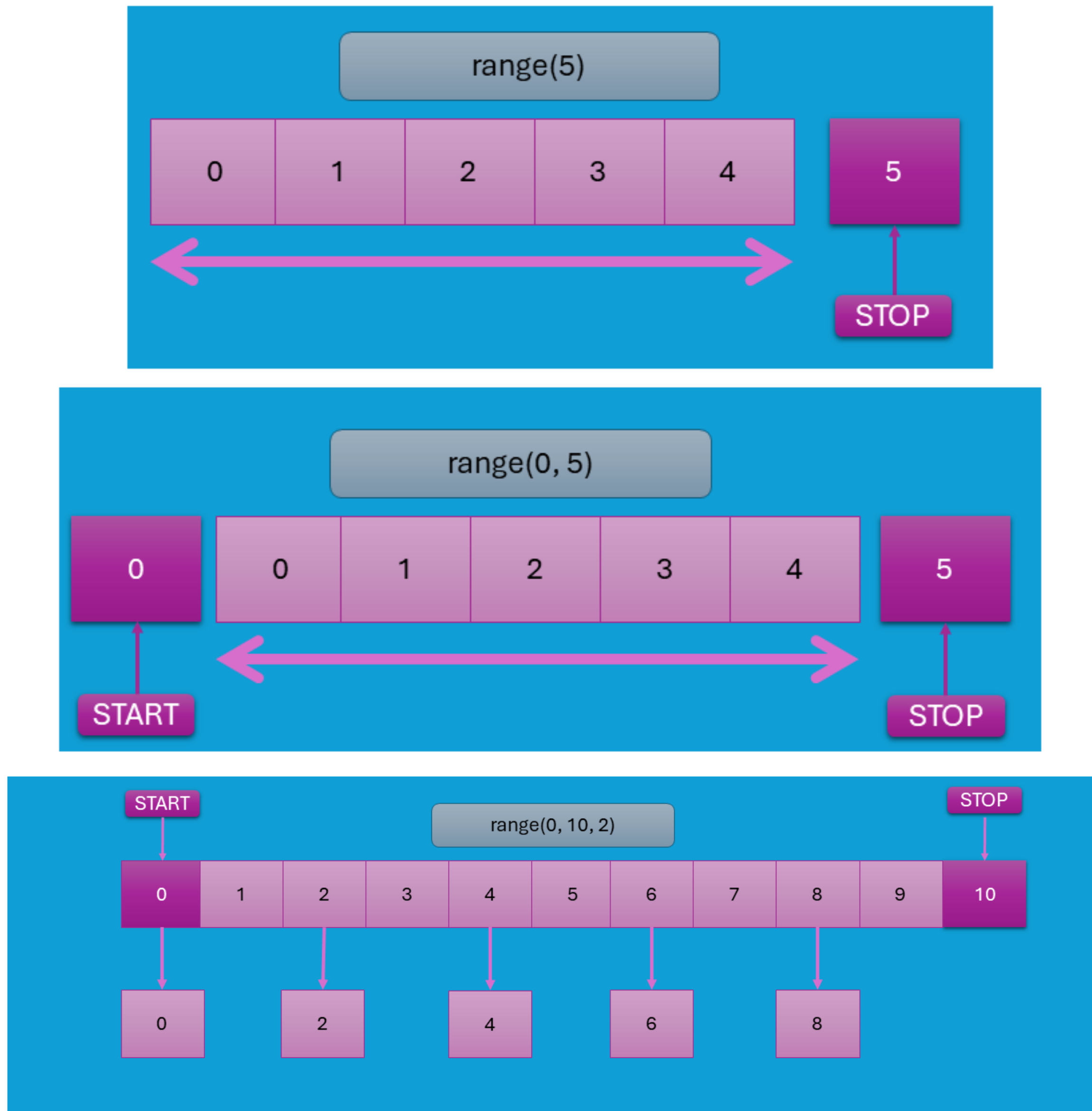
`f50. range(start, stop, pas)` returnează un obiect secvențial de numere începând de la valoarea `0` (implicit) sau de la o valoare indicată prin argumentul `start` prin incrementarea cu `1` (implicit) sau cu o valoare indicată prin argumentul `pas` până se atinge finalul indicat de argumentul `stop`. Singurul parametru obligatoriu este `stop`, funcția utilizând valorile implicite pentru ceilalți. Trebuie specificat faptul că, indexând din `0`, limita superioară indicată prin `stop` nu va fi atinsă. Spre exemplu, `range(3)` va returna un obiect secvențial compus din `3` numere: `0, 1, 2`. Dacă sunt furnizați doar doi parametri, aceștia vor fi interpretați drept `start` și `stop`.

```
# se creeaza o secventa de 10 numere incepand din 0 pana la 9
secventa_numere = range(10)
# se afiseaza tipul de obiect generat
print(type(secventa_numere))
# se afiseaza obiectul in sine
print(secventa_numere)
for numar in secventa_numere:
    print(numar, end=', ')
print()
# se creeaza o secventa de numere pare pana la 10
# se porneste din 0, se finalizeaza cu 12(neinclus) cu pas de 2
secventa_numere_pare = range(0, 12, 2)
for numar in secventa_numere_pare:
    print(numar, end=', ')
print()
# se creeaza o secventa de numere de la 5 pana la 8 (neinclus)
secventa_numere_ = range(5, 8)
for numar in secventa_numere_:
    print(numar, end=', ')
OUTPUT:
<class 'range'>
range(0, 10)
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 2, 4, 6, 8, 10,
5, 6, 7,
```

OBSERVAȚIE 1. Toate argumentele trebuie să fie numere de tip `int` (pozitive sau negative).

```
# incrementare cu -2
for i in range(25, 2, -2):
    print(i, end=" ")
OUTPUT:
25 23 21 19 17 15 13 11 9 7 5 3
```

OBSERVAȚIE 2. Fiind un obiect secvențial, trebuie iterat sau folosit ca argument al unei funcții constructor (de exemplu, `list()`) pentru a putea utiliza datele generate.



Figură 13. Exemple de utilizare a funcției `range()`

`f51. repr(obiect)` returnează o reprezentare imprimabilă a unui obiect trecut drept argument prin conversia obiectului într-un obiect tip string. Se folosește mai cu seamă în cazul în care se dorește afișarea rezultatului execuției unei expresii matematice sub forma unui șir de caractere fără a face conversia explicită `float` –`string`. În exemplul următor se inițializează o variabilă denumită `expresie_calcul` cu rezultatul împărțirii numărului `13` la `4` și se afișează inițial rezultatul și tipul acesteia, pe ecranul consolei fiind afișat numărul de tip `float` `3.25`. După utilizarea funcției `repr()` rezultatul este convertit într-un string.

```
# se initializeaza o variabila cu rezultatul unei espresii matematice
expresie_calcul = 13/4
# se afiseaza rezultatul si tipul datei returnate
print(f'{expresie_calcul} este de tipul {type(expresie_calcul)}')
# se analizeaza rezultaul functiei repr cu argumentul rezultatul expresiei anterioare
print(f'{repr(expresie_calcul)} este de tipul {type(repr(expresie_calcul))}')
OUTPUT:
3.25 este de tipul <class 'float'>
3.25 este de tipul <class 'str'>
```

`f52.` `reversed(obiect_secvențial)` returnează un obiect iterator inversat. De exemplu, dacă argumentul este o listă, funcția va returna acea listă, dar în ordine inversă (ultimul element devenind primul). Returnând un obiect tip iterator, pentru afișarea elementelor componente, acesta trebuie iterat utilizând o buclă iterativă – `for` sau `while` sau prin conversia explicită la o listă, un set etc. În exemplul următor se creează o listă și se populează cu trei obiecte de tip string, utilizându-se `reversed()` pentru a genera iteratorul inversat, printându-se inițial obiectul: `<list_reverseiterator object at 0x000001D9B35866E0>`. Obiectul este apoi convertit la o listă utilizând funcția constructor `list()` și rezultatele sunt afișate pe consolă în două moduri: prin afișare directă a listelor și folosind o buclă `while` împreună cu un element de oprire: variabila `index`.

```
lista_culori = ['galben', 'alb', 'mov', 'rosu']
lista_culori_inversa = reversed(lista_culori)
print(lista_culori_inversa)
lista_culori_inversa = list(lista_culori_inversa)

print(lista_culori)
print(lista_culori_inversa)

index = 0
while index < len(lista_culori_inversa):
    print(lista_culori[index],'-', lista_culori_inversa[index])
    index += 1
OUTPUT:
<list_reverseiterator object at 0x000001D9B35866E0>
['galben', 'alb', 'mov', 'rosu']
['rosu', 'mov', 'alb', 'galben']
galben - rosu
alb - mov
mov - alb
rosu - galben
```

Funcția `reversed()` se poate utiliza fără probleme și pe alt tip de dată secvențială, cum ar fi un șir de caractere, returnând textul de la cap la coadă în acest caz. Mai mult, se poate utiliza în conjuncție cu un alt iterator, cum ar fi funcția `range()` - `f50`. În fragmentul de cod următor sunt exemplificate și aceste cazuri.

```
# se creeaza o variabila si se initializeaza cu un obierct tip str
text = 'energie'
# se itereaaza prin iteratorul inversat
for litera in reversed(text):
    print(litera, end='')
print()
# se itereaza prin obiectul inversat al rezultaului range(5)
# range(5) genereaza secventa de numere 0, 1, 2, 3, 4.
# reversed(range(5)) returneaza 4, 3, 2, 1, 0
for numar in reversed(range(5)):
    print(numar, end='. ')
OUTPUT:
eigrene
4. 3. 2. 1. 0.
```

`f53.` **`round`**`(numar, digit)` returnează un număr de tip `float` care este versiunea rotunjită a numărului inițial, putându-se specifica, opțional, și numărul de zecimale – `digiti`. În exemplul următor se importă din modul implicit **`math`** constanta `pi` care se atribuie unei variabile – pentru care se folosește funcția **`round`**`()` cu și fără argumentul opțional `digit`. Se observă faptul că, dacă nu se folosește `digit`, funcția returnează un număr de tip `int` (cel mai apropiat număr întreg), și nu `float`.

```
# importam variabila pi din libraria math
from math import pi
numar_pi = pi
print(round(pi),'este de tipul', type(round(pi)))
print(round(pi, 1),'este de tipul', type(round(pi, 1)))
print(round(pi, 4))
print(round(pi, 13))
OUTPUT:
3 este de tipul <class 'int'>
3.1 este de tipul <class 'float'>
3.1416
3.1415926535898
```

`f54.` **`set`**`(iterabil)` este funcția constructor pentru obiectele de tip `set`, convertind orice obiect `secvențial`, `colecție` sau `iterator` trecut ca argument. Dacă argumentul nu este specificat, funcția returnează un obiect `set` gol. Detalii despre acest tip de obiect în capitolul următor. Este similară cu funcția **`frozenset`**`()` `- f24`. În exemplele din fragmentul de cod următor se creează un set gol și se convertesc atât o listă cât și un șir de caractere la seturi. Merită menționat faptul că un obiect de tip `set` este un obiect neordonat, însemnând că ordinea elementelor este aleatorie și afișarea lor poate diferi.

```
set_gol = set()
print(set_gol)
var_str = 'inginer-energetician'
var_set_str = set(var_str)
print(f'{var_set_str} este de tipul {type(var_set_str)}')
var_list = [1, 2, 'inginer', False, 12.3]
var_set_list = set(var_list)
print(var_set_list)
OUTPUT:
set()
{'t', 'n', 'c', 'e', 'r', '-', 'g', 'i', 'a'} este de tipul <class 'set'>
{False, 1, 2, 'inginer', 12.3}
```

f55. setattr(obiect, atribut, valoare) setează o valoare unui atribut specificat dintr-un obiect/clasă specificat/ă. Toți parametrii sunt obligatorii. Face parte din aceeași categorie cu funcțiile f14 - delattr(), f25 - getattr() și f27 - hasattr(). În exemplul următor se creează o clasă simplă care emulează un panou PV și care are trei atribute: tip, capacitate și eficienta. După instanțiere, două dintre acestea sunt modificate utilizând funcția setattr().

```
# se defineste o clasa simpla
class PanouPV():
    tip = 'poly_Si'
    capacitate = 220
    eficienta = 10.39

panoul_Si = PanouPV()
print('Cu valorile din clasa:')
print(f'Eficienta panoului meu este: {panoul_Si.eficienta}% si are capacitatea
de {panoul_Si.capacitate}W.')
# se atribuie alte valoari atributelor
setattr(panoul_Si, 'capacitate', 250)
setattr(panoul_Si, 'eficienta', 13.43)
print('Dupa setarea noilor valori:')
print(f'Eficienta panoului meu este: {panoul_Si.eficienta}% si are capacitatea
de {panoul_Si.capacitate}W.')
OUTPUT:
Cu valorile din clasa:
Eficienta panoului meu este: 10.39% si are capacitatea de 220W.
Dupa setarea noilor valori:
Eficienta panoului meu este: 13.43% si are capacitatea de 250W.
```

Funcția se poate utiliza și în cazul în care clasa nu conține inițial acel atribut, acesta fiind creat doar pentru obiectul respectiv, nu pentru întreaga clasă – exemplul următor.

```
setattr(panoul_Si, 'suprafata', 0.3)
print(f'panoul are o suprafata de: {panoul_Si.suprafata} mp')
OUTPUT:
panoul are o suprafata de: 0.3
```

f56. slice(start, stop, pas) returnează un obiect trunchiat/feliat; poate fi utilizat pentru a trunchia orice obiect de tip secvențial. Este asemănătoare cu funcția range() - f50. Parametrii sunt de tip int: start (opțional, implicit: None) – indică elementul de unde începe trunchierea; start (necesar) – indică elementul la care se oprește trunchierea; pas (opțional, implicit: None) – indică pasul de incrementare cu care se realizează trunchierea. Se utilizează pentru o obține obiecte secvențiale de dimensiuni mai mici din obiecte secvențiale inițiale. În exemplul următor se creează o variabilă de tip și de caractere și un obiect de tip slice care se utilizează pentru a trunchia șirul de caractere după până la al 3-lea caracter (0, 1 și 2), restul caracterelor fiind omise. În al doilea exemplu este creată o listă și un obiect tip slice cu toate argumentele.

```
text = "student"
trunchiator = slice(3)
print(f'{trunchiator} este de tipul {type(trunchiator)}')
print(text[trunchiator])

var_list = [1, 2, 'inginer', False, 12.3, 4.5, None, 'energetician']
trunchiator_lista = slice(1, 4, 2)
print(trunchiator_lista)
print(var_list[trunchiator_lista])
OUTPUT:
slice(None, 3, None) este de tipul <class 'slice'>
stu
slice(1, 4, 2)
[2, False]
```

OBSERVAȚIE. Ca și în cazul funcției range(), și aici se pot utiliza argumente negative.

f57. sorted(iterabil, key=key, reverse=reverse) returnează o listă cu elementele sortate din obiectul iterabil. Parametrii sunt: iterabil (necesar) – un obiect care va fi parcurs secvențial (secvență, colecție sau iterator) și se va sorta; key (opțional, implicit: None) – o funcție pe care să o execute pentru a stabili logica de sortare; reverse (opțional, implicit: False) – un boolean care indică direcția de sortare: crescător dacă este False și descrescător dacă este True. În exemplul următor o listă se sortează atât crescător cât și descrescător.

```
# se creeaza o lista cu numere
var_list = [1, 14, -12, 136,  12.3, 4.5, 457, -1789]
print(f'initial: {var_list}')
# se afiseaza pe ecranul consolei lista sortata crescator
print(f'creascator: {sorted(var_list)}')
print(f'descreascator: {sorted(var_list, reverse=True)}')
OUTPUT:
initial: [1, 14, -12, 136, 12.3, 4.5, 457, -1789]
creascator: [-1789, -12, 1, 4.5, 12.3, 14, 136, 457]
descreascator: [457, 136, 14, 12.3, 4.5, 1, -12, -1789]
```

În cazul în care se dorește o abordare specială a modului de sortare, funcția sorted() are argumentul key care permite specificarea unei funcții ce va fi utilizată drept etalon în sortare: spre exemplu, într-o listă ce conține cuvinte, să se sorteze în funcție de cel mai lung cuvânt –

utilizarea funcției încorporate `len()`- f37. În exemplul următor este creată o listă ce conține denumiri de materii specifice Facultății de Energetică. Lista va fi sortată în funcție de lungimea cuvintelor din componență, utilizând `key=len`. Atenție: funcția argument nu se apelează, ci doar se specifică!

```
lista_materii = ['Transfer de Caldura si Masa',
                 'Echipamentel si Instalatii Termice',
                 'Algebra',
                 'Analiza Matematica',
                 'Eficienta Energetica a Cladirilor']

print(sorted(lista_materii, key=len))
OUTPUT:
['Algebra', 'Analiza Matematica', 'Transfer de Caldura si Masa', 'Eficienta
Energetica a Cladirilor', 'Echipamentel si Instalatii Termice']
```

Funcționalitatea parametrului `key` nu se rezumă doar la a folosi funcții încorporate Python, ci se pot utiliza funcții proprii definite de programatori (inclusiv funcții `lambda`). În exemplul următor se creează o funcție care returnează restul împărțirii unui număr la 3 și se atribuie argumentului `key` la apelarea funcției. În cel de-al doilea exemplu, se folosește funcția `lambda` care generează ordonarea în funcție de cea mai mare valoare a mediilor de admitere. Lista de sortat este compusă din cinci dicționare care conțin perechi de chei și valori descriind o serie de studenți.

```
# Nested list of student's info in a Science Olympiad
# List elements: (Student's Name, Marks out of 100 , Age)
participant_list = [
    ('Alison', 50, 18),
    ('Terence', 75, 12),
    ('David', 75, 20),
    ('Jimmy', 90, 22),
    ('John', 45, 12)
]

def sorter(item):
    # Since highest marks first, least error = most marks
    error = 100 - item[1]
    age = item[2]
    return (error, age)

sorted_list = sorted(participant_list, key=sorter)
print(sorted_list)
OUTPUT:
[('Jimmy', 90, 22), ('Terence', 75, 12), ('David', 75, 20), ('Alison', 50,
18), ('John', 45, 12)]
```

```
# se defineste o functie care returneaza restul impartirii unui numar la 3
def par(numar):
    return numar % 3
# se defineste o lista de numere
var_list = [1, 14, -12, 136,  12.3, 4.5, 457, -1789]
# afiseaza ordinea in funcie de restul impartirii numerelor la 3 - crescator
print(sorted(var_list, key=par))
# afiseaza ordinea in funcie de restul impartirii numerelor la 3 -
descrescator
print(sorted(var_list, key=par, reverse=True))
print()
studenti = [
    {'cod': 'St00987', 'varsta': 20,'medie_admitere': 8.67},
    {'cod': 'St87665', 'varsta': 21, 'medie_admitere': 7.56},
    {'cod': 'St10990', 'varsta': 21, 'medie_admitere': 8.45},
    {'cod': 'St83665', 'varsta': 22, 'medie_admitere': 9.16},
    {'cod': 'St34990', 'varsta': 21, 'medie_admitere': 6.13},
]
studenti_sortati = sorted(studenti,key=lambda x: x['medie_admitere'])
for student in studenti_sortati:
    print(student)
```

OUTPUT:

```
[-12, 12.3, 1, 136, 457, 4.5, 14, -1789]
[14, -1789, 4.5, 1, 136, 457, 12.3, -12]
{'cod': 'St34990', 'varsta': 21, 'medie_admitere': 6.13}
{'cod': 'St87665', 'varsta': 21, 'medie_admitere': 7.56}
{'cod': 'St10990', 'varsta': 21, 'medie_admitere': 8.45}
{'cod': 'St00987', 'varsta': 20, 'medie_admitere': 8.67}
{'cod': 'St83665', 'varsta': 22, 'medie_admitere': 9.16}
```

f58. `str(obiect, encoding="utf-0", errors="strict")` returnează un obiect tip șir de caractere – `str`. În fapt, este funcția constructor pentru crearea șirurilor de caractere și convertorul explicit. Parametrul `obiect` specifică ce va fi convertit la string, `encoding` va indica modul în care se va face codificare (parametru opțional – valoarea implicită: UTF=8), `errors` indică ce trebuie să returneze dacă codificarea nu s-a putut realiza (parametru opțional, valoarea implicită: `"strict"`). Cele mai uzuale alternative pentru acest argument sunt: `ignore` – ignoră caracterul ne-codabil, `replace` – înlocuiește caracterul cu un caracter semnul întrebării. În exemplul următor se importă constanta universală `pi` din modulul `math` și se convertește explicit la tipul string utilizând funcția `str()`. Chiar dacă aparent are aceeași valoare, în cazul 2 obiectul este de tip `string`, și nu `float`, așa cum este inițial `pi`.

```
from math import pi
print(f'Initial pi are valoarea {pi} si tipul {type(pi)}')
print(f'Dupa conversrie pi este {str(pi)} si tipul {type(str(pi))}')
print(f'Adaug cateva zecimale random: pi este {str(pi)+ "9887"}')
```

OUTPUT:

```
Initial pi are valoarea 3.141592653589793 si tipul <class 'float'>
Dupa conversrie pi este 3.141592653589793 si tipul <class 'str'>
Adaug cateva zecimale random: pi este 3.1415926535897939887
```

f59. sum(iterabil, start) returnează un număr generat prin însumarea tuturor numerelor din parametrul iterabil. Parametrul start este opțional și reprezintă o valoare de la care pornește suma, implicit fiind 0. În exemplul următor se creează un obiect iterabil de tip tuplu, care conține numere întregi pentru care se dorește calculul sumei pornind din 0 și din 14. Numerele pot fi atât de timp int, cât și float, pozitive și negative.

```
# se creeaza un obiect tip tuplu si se atribuie unei variabile
valori = (1, 20, 13, 7)
print(f'suma numerelor {valori} este {sum(valori)}.')
print(f'daca pornim de la 14, suma numerelor este {sum(valori, 14)}.')
valori_neomogene = (-1, 2.0, 1345, -787, 23.67)
print(f'suma numerelor {valori_neomogene} este {sum(valori_neomogene)}.')
OUTPUT:
suma numerelor (1, 20, 13, 7) este 41.
daca pornim de la 14, suma numerelor este 55.
suma numerelor (-1, 2.0, 1345, -787, 23.67) este 582.67.
```

f60. tuple(iterabil) returnează un obiect tip tuplu din parametrul iterabil, fiind funcția constructor a acestor obiecte Python – detaliate în capitolul următor. Dacă argumentul iterabil nu este indicat, funcția va returna un tuplu gol. În fragmentul următor sunt prezentate mai multe obiecte iterabile convertite explicit la tupluri utilizând constructorul. A se observa modul de reprezentare a tuplurilor când sunt afișate pe terminalul consolei și a nu se confunda cu modalitatea de apelare a unui funcții.

```
# se creeaza un tuplu gol
tuplu_1 = tuple()
print('tuplu_1 =', tuplu_1)
# conversia lista - tuplu
tuplu_2 = tuple([1, 4, 6])
print('tuplu_2 =', tuplu_2)
# conversia string - tuplu
tuplu_3 = tuple('IA in IE')
print('tuplu_3 =',tuplu_3)
# conversia dictionar - tuplu
tuplu_4 = tuple({1: 'unu', 2: 'doi', 3: 'trei'})
print('tuplu_4 =',tuplu_4)
OUTPUT:
tuplu_1 = ()
tuplu_2 = (1, 4, 6)
tuplu_3 = ('I', 'A', ' ', 'i', 'n', ' ', 'I', 'E')
tuplu_4 = (1, 2, 3)
```

f61. type(obiect, bases, dict) returnează tipul parametrului obiect. Pe lângă parametrul obligatoriu, obiect, se pot indica, opțional (sub forma unui tuplu), clasele de bază din care obiectul derivă (parametrul bases) și un dicționar cu numele claselor (parametrul dict). A fost utilizată în nenumărate exemple până la acest paragraf.

```
print(type([]) is list)
print(type(()) is tuple)
print(type({}) is dict)
print(type({}) is not list)
OUTPUT:
True
True
True
True
```

f62. `zip(iterator1, iterator2,...)` returnează un obiect tip `zip` rezultat prin combinarea/concatenarea argumentelor `iterabile`. Concatenarea obiectelor se realizează element cu element. Astfel, primul element al argumentului `iterator1` va fi combinat cu primul element al argumentului `iterator2` sub forma unui tuplu. Dacă iterabilele au lungimi diferite, lungimea obiectului `zip` va fi dată de lungimea elementului mai scurt. Returnând un obiect `zip`, pentru a putea fi parcurs, acesta trebuie parcurs cu ajutorul unei declarații iterative. În exemplul următor se vor concatena trei liste conținând fiecare tehnice ale unor panouri fotovoltaice. La iterarea prin obiectul rezultat, se poate observa faptul că obiectul `zip` este alcătuit în esență dintr-un număr de tupluri generate prin combinarea elementelor din cele trei liste. Merită menționat faptul că se pot utiliza toate tipurile de date iterabile din sintaxa Python.

```
cod_panou = ['pv011', 'pv14', 'pt56', 'pv876', 'pt45']
putere_panou = [245, 245, 255, 345, 450]
randament_panou = [12.45, 11.45, 10.76, 12.98, 13.98]

descriere_panouri = zip(cod_panou, putere_panou, randament_panou)
print(descriere_panouri, type(descriere_panouri))

for panou in descriere_panouri:
    print(panou, type(panou))
OUTPUT:
<zip object at 0x000001D9B336D0C0> <class 'zip'>
('pv011', 245, 12.45) <class 'tuple'>
('pv14', 245, 11.45) <class 'tuple'>
('pt56', 255, 10.76) <class 'tuple'>
('pv876', 345, 12.98) <class 'tuple'>
('pt45', 450, 13.98) <class 'tuple'>
```

Există posibilitatea de a utiliza obiectul `zip` ca argument pentru o funcție constructor Python (`list()`, `tuple()`, `set()` etc) care poate și afișată pe ecranul consolei sau utilizată în cod. `list(zip(cod_panou, putere_panou, randament_panou))` va genera un obiect tip `list` utilizând funcția constructor `list()` și va fi atribuit variabilei `descriere_panouri` care va afișa: `[('pv011', 245, 12.45), ('pv14', 245, 11.45), ('pt56', 255, 10.76), ('pv876', 345, 12.98), ('pt45', 450, 13.98)] <class 'list'>`. A se observa modificarea tipului de dată generat cu ajutorul funcției `type()`. În exemplul următor se utilizează două liste și un tuplu care are o lungime mai mică decât cea a listelor. Aceste obiecte se concatenează utilizând `type()`, generându-se un obiect ce va conține un număr de elemente egal cu cel al tuplului (4 elemente), celelalte fiind eliminate. Adițional, se poate utiliza și funcția

`enumerate()` - f18 când se dorește procesarea în paralel a mai multor liste/tupluri și e nevoie de a accesa și indecșii acestora, oricare ar fi motivul.

```
# se definesc 2 liste si un tuplu
prieteni = ["Chandler", "Monica", "Ross", "Rachel", "Joey", "Phoebe",
"Joanna"]
sexe = ["M", "F", "M, "F", "M", "F", "Fe"]
varste = (35, 36, 38, 34)
# se concaneaza cele 3 elemente
ziper = zip(prieteni, sexe, varste)
print("Se itereaza prin obiectul zip:")
for element in ziper:
  print(element)
print()
print("Utilizand enumerate:")
for index, (priente, sex, varsta) in enumerate(zip(prieteni, sexe, varste)):
  print(index, "-", priente, sex, varsta)
```

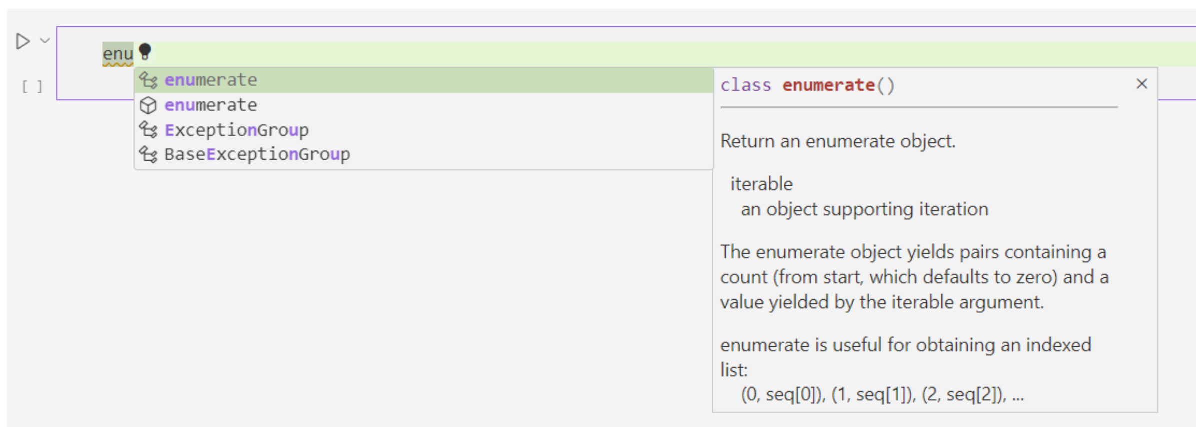
```
OUTPUT:
Se itereaza prin obiectul zip:
('Chandler', 'M', 35)
('Monica', 'F', 36)
('Ross', 'M', 38)
('Rachel', 'F', 34)

Utilizand enumerate:
0 - Chandler M 35
1 - Monica F 36
2 - Ross M 38
3 - Rachel F 34
```

CONCLUZII

În orice limbaj de programare există funcții încorporate care au rolul de a ușura procesul de creare a programelor, facilitând reutilizarea celor mai des utilizate funcționalități. Python are încorporate în sintaxa proprie aproximativ 70 de funcții, dintre care cele mai importante au fost prezentate și exemplificate în cadrul acestui capitol. Acest tip de funcții sunt utilizate pentru a efectua sarcini comune, cum ar fi: crearea tipurilor de obiecte încorporate, convertirea unor tipuri de date în alte tipuri de date, definirea funcțiilor și a claselor, afișarea datelor, manipularea datelor etc. În majoritatea cazurilor acestea nu sunt suficiente, fiind necesară extinderea funcționalității prin importarea modulelor sau a librăriilor specifice. Funcțiile încorporate nu pot fi create de programatorii uzuali și nici modificate, acest lucru însemnând în fond alterarea întregii sintaxe Python, lucru ce poate genera erori la compilare.

În cazul cel mai probabil în care este utilizat un mediu integrat de dezvoltare a aplicațiilor dedicat pentru limbajul Python (sau unul de uz comun cu funcționalitatea extinsă pentru a lucra cu Python), funcțiile încorporate vor fi presetate și acestea vor apărea în momentul în care se scriu primele litere din funcție. Mai mult de atât, în majoritatea IDE-urilor utilizate există și un help

încorporat care oferă date despre funcția în sine. Un astfel de caz este prezentat în Figură 14, unde au fost scrise primele litere ale funcției `enumerate()` și a fost generată automat o serie de sugestii de cuvinte cheie care se încep la fel și, în plus, o fereastră pop-up cu informații despre acea funcție. De menționat faptul că această funcționalitate nu se rezumă doar la funcții încorporate ci și la cuvinte cheie sau alte părți din sintaxa Python. De asemenea, atât această imagine cât și fragmentele de cod prezentate în acest capitol au fost generate utilizând Visual Studio Code IDE împreună cu o serie de extensii dedicate folosirii limbajului de programare Python.



Figură 14. Exemplu de scriere a unei funcții încorporate și fereastra ajutătoare generată

Toate aceste funcții sunt corelate tipurilor de date prezentate în capitolul următor.

4. Tipuri de date principale și metodele asociate lor

Orice limbaj de programare are încorporat în sintaxa proprie diferite tipuri de date esențiale utilizate pentru a se putea dezvolta diversele aplicații. Acestea fac parte din categoria tipurilor de **date primitive** și în sintaxa limbajului de programare Python există date: **numerice**: `integer` (număr întreg), `float` (număr zecimal – în virgulă flotantă), `complex`, **secvențiale**: `string` (șir de caractere), `list` (liste), `tuple` (tupluri), `range`, **mapare**: `dictionaire` (dicționare), **seturi**: `set`, `frozenset`, **clase**, **instanțe** și **excepții**.

În acest subcapitol vor fi detaliate aceste tipuri de date și modalitatea de a lucra cu ele (declarație, metode, transformări etc.). O particularitate interesantă a limbajului Python este că acesta a fost conceput să fie dinamic în scrierea codului (*dynamic typing*), însemnând că unei variabile i se alocă tipul de dată în momentul declarării și inițializării (aceste două acțiuni având loc în același timp), comparat cu alte limbaje de programare care necesită inițial specificarea tipului de dată pe care acea variabilă îl va referi. Drept urmare, este complet corect din punctul de vedere al sintaxei ca o variabilă să conțină inițial o dată de timp `int` ca mai apoi, acesteia să îi fie atribuit un șir de caractere – `str`. În fragmentul următor de cod unei variabile i se atribuie, pe rând, mai multe tipuri de date și, utilizând funcția încorporată `type()` se afișează pe consolă tipul respectiv de dată. Conform rezultatelor afișate, toate tipurile de date aparțin unei clase specifice, însemnând că, în Python, toate variabilele ce referă date sunt, de fapt, **obiecte** – instanțe ale **claselor** respective. Acest lucru a permis atribuirea unor funcționalități specifice prin intermediul metodelor ce acționează pe aceste tipuri de date.

```
# se defineste variabila si se initializeaza cu o data tip int
variabila = 1
print(type(variabila))
<class 'str'>
```

```
# variabilei definite si se atribuie acum o data tip float
variabila = 0.1
print(type(variabila))
# variabilei definite si se atribuie acum o data tip str
variabila = '1'
print(type(variabila))
OUTPUT:
<class 'int'>
<class 'float'
```

DATE MUTABILE ȘI DATE IMUTABILE

În limbajul de programare Python tipurile de date fundamentale se împart în două mari categorii: date mutabile și date imutabile. Această clasificare se referă la posibilitatea de a modifica o stare internă a unei instanțe după ce aceasta a fost creată sau nu. Obiectele **`mutabile`** sunt cele a căror stare internă poate fi alterată după ce au fost instanțiate, în comparație cu obiectele **`imutabile`** care nu pot fi modificate odată create. Modificarea datelor mutabile se realizează local, direct pe obiectul respectiv, fără a-i modifica acestuia identitatea. Caracteristica de mutabilitate a obiectelor este foarte importantă și înțelegerea acesteia asigură utilizarea tipului de dată potrivit în funcție de specificul subrutinei care se dorește a fi implementată. Ca o regulă nescrisă, obiectele imutabile sunt specifice tipului de programare funcțională și cele mutabile programării orientate pe obiecte. Cum Python permite ambele abordări, trebuie înțeleasă foarte bine diferențierea între cele două și utilizarea optimă a acestora.

Încă de la început trebuie specificat faptul că, în Python (spre deosebire de alte limbaje de programare similare), variabilele/IDENTIFICATORUL nu au asociate tipuri (de date) sau mărimi. Ele/ei nu sunt altceva decât pointeri (indicatori) etichetați către zone din memoria internă a sistemului de calcul în care sunt stocate obiectele create. În contrast cu acest lucru, obiectele Python reprezintă bucăți concrete de informații din memorie și către care pointerii indică. Acest lucru explică caracterul dinamic al sintaxei Python și faptul că nu trebuie specificat tipul variabilei la creare (spre deosebire de C++ sau Java).

Încă de la început trebuie specificat faptul că, în Python (spre deosebire de alte limbaje de programare similare), IDENTIFICATORUL nu au asociate tipuri (de date) sau mărimi. Ele/ei nu sunt altceva decât pointeri (indicatori) etichetați către zone din memoria internă a sistemului de calcul în care sunt stocate obiectele create. În contrast cu acest lucru, obiectele Python reprezintă bucăți concrete de informații din memorie și către care pointerii indică – v. fig. 13. Acest lucru explică caracterul dinamic al sintaxei Python și faptul că nu trebuie specificat tipul variabilei la creare (spre deosebire de C++ sau Java).

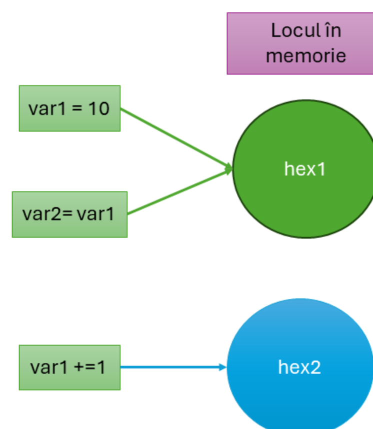


Figura 15. Atribuirea valorilor în memorie

Chiar dacă indicatorul și obiectul sunt două lucruri distincte, obiectul se instanțiază simultan cu crearea indicatorului, fără specificarea explicită a tipului de dată. În exemplul următor indicatorul creat este denumit `variabila` și este un pointer către o zonă de memorie unde este instanțiat un obiect tip `int` care are valoarea `1`.

```
variabila = 1
print(f'Indicatorul "variabila" are valoarea {variabila}.')
print(f'Indicatorul "variabila" este de tipul {type(variabila)}.')
print(f'Indicatorul "variabila" are id {id(variabila)}.')
OUTPUT:
Indicatorul "variabila" are valoarea 1.
Indicatorul "variabila" este de tipul <class 'int'>.
Indicatorul "variabila" are id 140713469535016.
```

În contrast cu alte limbaje de programare unde există date primitive și clase, în Python absolut orice tip de dată este gândit sub forma unei clase, incluzând aici și date numerice, boolean, mulțimea vidă, drept urmare toate datele utilizate în dezvoltarea programelor sunt instanțe ale acestor obiecte. Fiecare obiect este descris prin trei caracteristici: **valoare**, **identitate** și **tip**.

Valoarea este cea mai intuitivă caracteristică și reprezintă data (datele) efectivă pe care obiectul se fundamentează. În exemplul anterior valoarea indicatorului `variabila` este `1`. Valoarea este stocată pe o poziție din zona de memorie stabilită de procesorul calculatorului și această adresă reprezintă **identitatea** obiectului – un identificator unic care permite procesorului să facă diferențierea dintre diverse obiecte. În exemplul anterior, variabila denumită `variabila`, care indică spre obiectul tip `int` a fost salvată la adresa din memorie descrisă de identificatorul numărul `140713469535016`. Identitatea unui obiect este de tipul **read-only** și nu poate fi modificată odată ce obiectul a fost plasat la o anumită adresă.

Tipul este ultima caracteristică descriptivă a obiectelor Python și acesta determină dacă o dată este mutabilă sau imutabilă. În exemplul anterior tipul variabilei denumite `variabila` este de tip `int`, însemnând că este un obiect din clasa integer. Tipul obiectelor este fix, dar nu este **read-only**, acesta putând fi modificat prin modificarea atributului intern `.__class__`. Cu toate acestea, singura caracteristică ce se modifică în mod curent în programare este valoarea care este atribuită unui obiect.

Așadar, orice obiect care permite schimbarea valorii sale fără a-i schimba identitatea, este un obiect mutabil și operațiile care se realizează asupra sa se numesc *mutații* sau *alterări*. În contrast, obiectele imutabile nu permit niciun fel de operații care le pot altera valorile. Singura opțiune este aceea de a crea un nou obiect de același tip, dar cu o valoare diferită, și de a-l atribui aceluiași indicator/variabilă. În momentul în care se realizează acest lucru, variabila nu va mai indica către obiectul vechi, ci va face referința către obiectul nou creat.

```
# identificatorul va indica catre obiectul int 1
variabila = 1
print(f'Indicatorul "variabila" are id {id(variabila)}.')
# identificatorul acum va indica catre obiectul float 1.3
variabila = 1.3
print(f'Indicatorul "variabila" are id {id(variabila)}.')
OUTPUT:
Indicatorul "variabila" are id 140713469535016.
Indicatorul "variabila" are id 1869752188112.
```

În Python, tipurile de date încorporate în sintaxă sunt atât de tip imutabil (majoritatea), cât și mutabile. Tipurile de date singulare, precum date numerice și tip `boolean` sunt întotdeauna imutabile. De fapt, `boolean` este o subclasă a clasei `int` (`0` reprezintă `False` și orice alt număr reprezintă `False`), moștenind și imutabilitatea acesteia. Când vine vorba de colecții de date (sau containere) – string, liste, tupluri și seturi, lucrurile stau puțin diferit, deoarece Python, intern, ține cont de referințele fiecărui obiect care compune acel container, dar și de referința containerului în sine. Astfel, sunt create premisele realizării obiectelor mutabile – se poate schimba **valoarea** obiectului fără a-i modifica **identitatea** (conform definiției!). Cu alte cuvinte, dacă într-o listă se modifică un element de pe o anumită poziție, se modifică doar identitatea acelui element, identitatea obiectului listă rămânând nealterată. În fragmentul de cod următor este prezentat acest aspect prin analiza unei liste alterate prin modificarea obiectului de pe prima poziție.

```
# se intantiaza lista initiala
lista_mutabil = [1, 23, 'mutabil']
print(f"Lista initiala: {lista_mutabil}")
print(f'Indicatorul "lista_mutabil" are id {id(lista_mutabil)}.')
# se modifica lista:
lista_mutabil[0] = 0
print(f'Lista dupa alterare: {lista_mutabil}')
print(f'Indicatorul "lista_mutabil" are id {id(lista_mutabil)}.')
OUTPUT:
Lista initiala: [1, 23, 'mutabil']
Indicatorul "lista_mutabil" are id 1869759197696.
Lista dupa alterare: [0, 23, 'mutabil']
Indicatorul "lista_mutabil" are id 1869759197696.
```

Chiar dacă toate datele tip colecții/containere folosesc același mecanism intern de a stoca obiectele, în sintaxa Python acestea sunt atât de tip mutabil cât și imutabil: string și tupluri sunt imutabile, în timp de listele, dicționarele și seturile sunt mutabile.

Obiectele de tip string (șir de caractere) sunt, în esență, o colecție de obiecte de string de mărime `1`, reprezentând un singur caracter (în alte limbaje de programare există un tip de date specific denumit `char` – caracter). Cum acel string unitar este imutabil, rezultă că întregul container tip string este imutabil! Chiar dacă există o serie de metode specifice asociate obiectelor tip string, (care doar aparent le alterează), acestea nu fac altceva decât să returneze o copie modificată a acelui obiect, însemnând faptul că va crea intern un nou obiect și îl va stoca la o altă adresă sub o altă identitate. Un mare dezavantaj al acestui mecanism este faptul că necesită mai multă memorie pentru a lucra cu aceste tipuri de date, dar avantajul este că, neputând fi alterate, acestea sunt o sursă de erori în minus.

Concluzionând, în Python există încorporate atât obiecte mutabile cât și imutabile:

- imutabile:
  - `int`, `float`, `complex`, `string`, `tuple`, `bytes`, `bool`, `frozenset`
  - utilizate când este necesară stabilitatea și consistența datelor;
- mutabile:
  - `list`, `dict`, `set`, `bytearray`
  - utilizate când este necesară flexibilitatea și modificarea datelor.

Fiecare tip de dată va fi analizat în detaliu în secțiunile următoare.

4.1. Date numerice: `int`, `float`, `complex`

Clasele care definesc numerele în Python sunt: `int` – pentru numere întregi, `float` – pentru numere zecimale (cu virgulă flotantă) și `complex` – pentru numere complexe. Obiectele numerice sunt imutabile în Python. Pentru fiecare dintre aceste tipuri există și funcții constructor, așa cum a fost detaliat în § 3.2. *Exemple de utilizare a funcțiilor încorporate*: `int()` - `f33`, `float()` - `f22` și `complex()` - `f13`, utilizându-se cu precădere pentru conversia explicită de la un tip de dată la altul. Structurile de date bazate pe numere sunt fundamentul domeniilor științei datelor și inteligenței artificiale și importanța lor este evidențiată și de multitudinea de librării și module terțe utilizate în manipularea lor (spre exemplu, Numpy – Numeric Python). Fiind un limbaj de programare cu scriere dinamică, nu este neapărat nevoie de a indica tipul de dată care se atribuie unei variabile, Python realizând automat acest lucru în funcție de clasa specifică. Drept urmare, uzual este de a se inițializa direct variabila cu data numerică specifică (de exemplu, crearea unui număr întreg: `numar_integer = 10`), acest lucru denumindu-se crearea literară a unei `date` (`literal int`), deoarece **literalmente** se creează o variabilă conținând un obiect din clasa datei respective. Acest lucru este în contrast cu crearea explicită a datelor numerice încorporate folosind funcțiile constructor existente `int()` - `f33`, `float()` - `f22` și `complex()` - `f13`. Python, se poate utiliza inclusiv ca un calculator, acceptând toate tipurile de operații aritmetice existente, așa cum sunt exemplificate în tabelul 5.

Tabel 5. Principalele operații aritmetice în Python

definire variabile numerice	

exemplu de cod	`nr_int = 3`
	`nr_float = 1.3`
	`nr_complex = 1+3j`

	`print(nr_int, 'este de tipul', type(nr_int))` `output: 3 este de tipul <class 'int'>`
	`print(nr_float, 'este de tipul', type(nr_float))` `output: 1.3 este de tipul <class 'float'>`
	`print(nr_complex, 'este de tipul', type(nr_complex))` `output: (1+3j) este de tipul <class 'complex'>`
adunare	operator: `+`

exemplu de cod	`suma = nr_int + nr_float`
	`print(f'{nr_int} + {nr_float} = {suma} ({type(suma)})')`
	`output: 3 + 1.3 = 4.3 (<class 'float'>)`

	`suma = nr_int + nr_complex`
	`print(f'{nr_int} + {nr_complex} = {suma} ({type(suma)})')`
	`output: 3 + (1+3j) = (4+3j) (<class 'complex'>)`

	`suma = nr_int + 10`
	`print(f'{nr_int} + {10} = {suma} ({type(suma)})')`
	`output: 3 + 10 = 13 (<class 'int'>)`

scădere	operator: `-`

exemplu de cod	`diferenta = nr_int - nr_float`
	`print(f'{nr_int} - {nr_float} = {diferenta} ({type(diferenta)})')`
	`output: 3 + 1.3 = 4.3 (<class 'float'>)`

	`diferenta = nr_float - nr_complex` `print(f'{nr_float}-{nr_complex}={diferenta} ({type(diferenta)})')`

	output: 1.3 - (1+3j) = (0.3000000000000004-3j) (<class 'complex'>)
	diferenta = nr_int - 10 print(f'-{10} + {nr_int} = {diferenta} ({type(diferenta)})') output: -10 + 3 = -7 (<class 'int'>)
produs	operator: *
exemplu de cod	produs = nr_int * nr_float print(f'{nr_int} x {nr_float} = {produs} ({type(produs)})') output: 3 x 1.3 = 3.9000000000000004 (<class 'float'>)
	produs = nr_int * nr_complex print(f'{nr_int} x {nr_complex} = {produs} ({type(produs)})') output: 3 x (1+3j) = (3+9j) (<class 'complex'>)
	produs = nr_int * nr_complex print(f'{nr_int} x {nr_complex} = {produs} ({type(produs)})') output: 10 x 3 = 30 (<class 'int'>)
împărțire	operator: /
exemplu de cod	impartire = nr_int / nr_float print(f'{nr_int} ÷ {nr_float} = {impartire} ({type(impartire)})') output: 3 ÷ 1.3 = 2.3076923076923075 (<class 'float'>)
	impartire = nr_float / nr_complex print(f'{nr_float}÷{nr_complex}={impartire} ({type(impartire)})') output: 1.3 ÷ (1+3j) = (0.13-0.39j) (<class 'complex'>)
	impartire = nr_int / 3 print(f'{nr_int} ÷ {3} = {impartire} ({type(impartire)})') output: 3 ÷ 3 = 1.0 (<class 'float'>)
împărțire (partea întreagă)	operator: //

exemplu de cod	impartire = nr_int // nr_float
	print(f'{nr_int} // {nr_float} = {impartire} ({type(impartire)})')
	output: 3 // 1.3 = 2.0 (<class 'float'>)

	impartire = nr_int // 3 print(f'{nr_int} // {3} = {impartire} ({type(impartire)})') output: 3 // 3 = 1 (<class 'int'>)
împărțire (restul împărțirii)	operator: % (mod)
exemplu de cod	impartire = nr_int % nr_float print(f'{nr_int} % {nr_float} = {impartire} ({type(impartire)})') output: 3 % 1.3 = 0.3999999999999999 (<class 'float'>)
	impartire = nr_int % 2 print(f'{nr_int} % {2} = {impartire} ({type(impartire)})') output: 3 % 2 = 1 (<class 'int'>)

Python se poate folosi și la rezolvarea calculelor numerice complexe, bazându-se pe regulile matematice – PEMDAS (paranteze, exponenți, multiplication – înmulțire, division – împărțire, addition – adunare, substraction – scădere). Astfel, se păstrează și aici ordinea de precedență a operațiilor aritmetice. Prezentarea ordinii operațiilor se regăsește în tabelul 6, unde au fost trecute în ordine crescătoare din punctul de vedere al importanței: de la cel mai puțin important operand până la cel mai important. Trebuie menționat faptul că operatorii logici pe biți (SAU - |, OR - ^, AND - &) nu sunt luați în considerare.

Tabel 6. Precedența operațiilor în Python

Operand	Descriere	Exemplu cod
=	atribuire	nume = "Inteligenta Artificiala"
or	sau logic	print(1 or 10 + 10 or 8) """Operatorul or are o precedenta mai mica decat adunarea. Drept urmare se va calcua intai 10+10 si apoi ca calcula 1 or 10 or 8 – rezultatul va fi 1""" output: 1
and	și logic	print(19 or 20 + 10 and 8) """Operatorii or si and au o precedenta mai mica decat adunarea. Drept urmare se va calcua intai 20+10 si apoi se va calcula 19 or 30 and 8 – rezultatul va fi 19""" output: 19
not	negarea	print(not 10 == 10) """Operatorul not are o precedenta mai mare mica decat operatorul comparatie. Drept urmare se va analiza intai daca 10==10 si apoi rezultatul se va nega""" output: False
==;!=;>; >=;<;<=; is; is not; in; not in	comparații, identitate, apartenență	print(13 != 3 + 7) """Operatorul comparativ are o precedenta mai mare mica decat operatorul adunare.Drept urmare se va calcula intai 3+7 si apoi rezultatul se va compara cu 13""" output: True
+ -	adunare, scădere	print(10 + 3 - 500) """Operatorii adunare si scadere au aceeasi precendeta. Drept urmare se calculul se va realiza de la stanga la dreapta: adunare apoi scadere""" output: -487
* / // %	înmulțire, împărțire, întreg, rest, (operatori binari)	print(10 * 3 - 4 // 3 + 1 % 3) """Operatorii binari precendeta mai mare decat adunarea si scaderea. Se va calcula initial inmultirea (10*3), partea intreaga (4//3) si restul (1%3), stanga-dreapta, apoi scaderea si adunarea (stanga-dreapta)""" output: -487
** pow()	ridicare la putere	print(2**3 + 1 == pow(2, 3) + 2) """Operatorul de ridicare la putere are o precedenta mai mare decat adunarea si decat operatorul comparativ. Drept urmare, se va ridica la putere, se va aduna si apoi se vor compara rezultatele""" output: False
()	paranteze	print((1 - 3) * (2 + 7)) """Folosirea parantezelor area cea mai mare precedenta in aritmetica Python. Atentie – se folosesc doar partanteze rotunde, celelalte avand alte utilizari""" output: -18s

Cu ajutorul acestor operatori și respectând ordinea precedenței se pot realiza calcule complexe utilizând Python. În momentul în care expresiile cresc în complexitate, interpretorul Python grupează părțile componente ale acesteia în funcție de regulile precedenței, grupare care de fapt determină ordinea în care sub părțile componente vor fi evaluate. Precedența operațiilor poate fi modificată în funcție de nevoile specifice prin utilizarea parantezelor. Spre deosebire de matematică, în calculele Python se utilizează doar parantezele rotunde, restul având alte utilizări: definirea listelor – parantezele drepte (`[]`) și a dicționarelor – acolade (`{}`). Un astfel de exemplu complex este detaliat în fragmentul de cod următor, unde trebuie evaluată o expresie compusă din 5 operanzi reprezentând numere întregi și 5 operatori: `(), +, *, %` și `/`. Parantezele au cea mai mare precedență, și expresia din interiorul lor va fi evaluată prima dată, rezultând `12`. Precedența operatorilor `*, %` și `/` este mai mare decât `+` și vor fi executați înaintea adunării, ținând cont de regula asociativității – ordinea în care ele apar în expresie de la stânga la dreapta. `12 * 3` va rezulta numărul `36`. Urmează execuția `36 % 30` rezultând numărul întreg `6` și `2 / 4` rezultă numărul float `0.5`. Cele două rezultate intermediare vor fi, în final, adunate între ele ca ultimă operație a expresiei, rezultatul final fiind `6.5` (float).

```
# se definesc o variabila si se initializeaza cu o expresie
expresie = (2 + 10) * 3 % 30 + 2 / 4
# se afiseaza rezulatul evaluarii expresiei
print('(2 + 10) * 3 % 30 + 2 / 4 =', expresie)
OUTPUT:
(2 + 10) * 3 % 30 + 2 / 4 = 6.5
```

În fragmentul de cod următor sunt exemplificate operații de comparație între numere și rezultatele acestora. Expresiile comparative rezultă fie în valoarea booleană `True` fie în `False`.

```
# se definesc doua variabile si se initializeaza cu doua date numerice
# int
numar1 = 7
#float
numar2 = 3.5
# se utilizeaza operatorii logici pentru comparatii
```

```
print(f"{numar1} este mai mic decat {numar2}:", numar1 < numar2)
print(f"{numar1} este mai mare decat {numar2}:", numar1 > numar2)
print(f'{numar2} este jumatete din {numar1}:', numar2 == numar1/2)
print(f'{numar1} este de doua ori mai mare decat {numar2}:', numar1 ==
numar2*2 )
print(f'{numar1} este diferit de 7.0:', numar1 != 7.0)
OUTPUT:
7 este mai mic decat 3.5: False
7 este mai mare decat 3.5: True
3.5 este jumatete din 7: True
7 este de doua ori mai mare decat 3.5: True
7 este diferit de 7.0: False
```

Sintaxa Python permite și înlănțuirea operațiilor comparative; acești operatori au aceeași precedență și ei vor fi evaluați de la stânga la dreapta în cazul unei expresii complexe, precum este prezentat în fragmentul de cod următor. Expresia este divizată în subexpresii la nivelul

execuției codului, astfel că expresia complexă devine o înșiruire de și-uri logice: `(numar_1>numar_2) and (numar_2 == numar_3) and (numar_3 >= numar_4)`. Dacă una dintre operații rezultă în `False`, restul devin redundante deoarece expresiile `False and True` și `False and False` sunt evaluate mereu drept `False` și indiferent de valoarea booleană a următoarelor expresii, expresia finală va fi tot `False`!

```
# se definesc patru variabile si se initializea cu numere intregi
numar_1 = 100
numar_2 = 30
numar_3 = 30
numar_4 = 20
# se afiseaza pe ecran valoarea de adevar a expresiei comparative complexe
print(numar_1 > numar_2 == numar_3 >= numar_4)
OUTPUT:
True
```

OBSERVAȚIE. În Python nu există limită în mărimea ce o poate avea un număr de tip `int`, singura limitare fiind dată de memoria internă a sistemului de calcul folosit. Acest lucru vine în antiteză cu alte limbaje de programare care folosesc, pentru optimizarea memoriei, două tipuri distincte de date `int`: `short` (care poate ocupa maximum 16 biți) și `long` (care poate ocupa maximum 32 de biți). În schimb, numerele de tip `float` în Python au o mărime fixată, deși extrem de mare: 2×10^{400}. Dacă se atinge această limită, Python va returna `inf`.

În cazul operațiilor cu diferite tipuri de date, Python face automat conversia implicită la tipul de dată mai complexă. Acest lucru s-a putut observa în exemplele de cod prezentate în Tabel 5 când rezultatul unei operații între un număr de tip `int` și un număr de timp `float` este întotdeauna un număr de tip `float`, acesta fiind numărul mai complex dintre cele două. În mod similar, toate numerele sunt implicit convertite în numere complexe când acestea sunt folosite. Există și posibilitatea de a folosi conversia explicită a numerelor prin utilizarea funcțiilor constructor încorporate: `int()`, `float()` și `complex()`.

```
# se defineste o variabila si se initializea cu un numere zecimal - float
numar = 3.7
print(f'{numar} ca int este {int(numar)}')
print(f'3 ca float este {float(3)}')
print(f'{numar} ca numar complex este {complex(numar)}')
OUTPUT:
3.7 ca int este 3
3 ca float este 3.0
3.7 ca numar complex este (3.7+0j)
```

OBSERVAȚIE 1. Utilizarea conversiei explicite are câteva particularități foarte importante în rezultate: `int()` trunchiază numărul zecimal (nu rotunjește superior); `float()` adaugă partea zecimală, dar numărul zecimal este `0`; `complex()`adaugă partea imaginară, dar aceasta este `0j`.
OBSERVAȚIE 2. Conversia implicită funcționează doar pentru tipurile numerice.

În cazul în care există necesitatea de a lucra cu numere foarte mari, pentru o mai bună lizibilitate a acestora se poate folosi ”_„ pentru a separa unitățile între ele: `numar_mare = 100_000_000_000_000_000_000` este o expresie validă în sintaxa Python care atribuie numărul

de tip `int` variabilei denumite `numar_mare`. Adițional, când se lucrează cu numere de tip `float` foarte mari, Python utilizează notația exponențială, numărul `200_000_000_000_000_000.0` fiind afișat sub forma `2e+17`, însemnând 2 x 10^{17}. Numărul `2.2e-3` reprezintă 0.0022, însemnând, de fapt, 2.2 x 10^{-3}. Dacă se atinge numărul care reprezintă limita numerică maximă pe care un număr `float` o poate avea, Python va returna `±inf`:

```
print(1e400)
print(-1e400)
OUTPUT:
inf
-inf
```

Pentru detalii legate de funcțiile încorporate Python ce se pot utiliza pentru datele numerice se va analiza § *3.2. Exemple de utilizare a funcțiilor încorporate*. Obiectele numerice de tipul `int` și `float` au integrată o metodă de clasă cu ajutorul căruia se poate analiza dacă un număr este de timpul `int` sau nu, returnând `True` sau `False`. Metoda `is_integer()` este o metodă de clasă care nu acceptă niciun argument și care verifică dacă obiectul de care este legată face parte din clasa `int` sau nu. Adițional, se poate folosi metoda încorporată specifică pentru aceleași tipuri de date: `as_integer_ratio()` care returnează o pereche de numere (sub forma unui tuplu), al căror raport este exact egal cu valoarea inițială. În fragmentul de cod următor sunt exemplificate ambele metode specifice.

```
putere_consumata = 8.5
print(putere_consumata.is_integer())
print(putere_consumata.as_integer_ratio())
OUTPUT:
False
(17, 2)
```

Python este printre puținele limbaje de programare care dedică un tip separat numerelor complexe, fapt pentru care, această clasă este prevăzută cu două atribute (`real` și `imag`) și o clasă (`conjugate()`) specifice. După cum sugestiv au fost denumite, `real` și `imag` conțin partea reală și cea imaginară a numărului complex, în timp ce metoda `conjugate()` nu face altceva decât să returneze conjugatul numărului inițial – schimbarea semnului părții imaginare.

```
# se defineste o variabila ce contine un numar complex
numar_complex = complex(2, 5)
print(f'Partea reala a numarului {numar_complex} este {numar_complex.real}')
print(f'Partea imaginara a numarului {numar_complex}este{numar_complex.imag}')
print(f'Conjugatul numarului {numar_complex} este{numar_complex.conjugate()}')
OUTPUT:
Partea reala a numarului (2+5j) este 2.0
Partea imaginara a numarului (2+5j) este 5.0
Conjugatul numarului (2+5j) este (2-5j)
```

4.2. Șiruri de caractere: str

Un prim tip de dată **secvențială** analizată va fi șirul de caractere – tip `string`, definit total de clasa `str`. În esență, șirurile de caractere reprezintă o colecție de caractere (alte stringuri de lungime 1) ordonată utilizată pentru a stoca și reprezenta text și/sau informație de tip byte.

Funcțional, șirurile de caractere pot fi utilizate pentru a reprezenta orice obiect Python care poate fi codificat sub formă de text sau byți. Obiectele formate din clasa `str` sunt obiecte secvențiale **`imutabile`**, însemnând că odată create acestea nu mai pot fi alterate (v. § DATE MUTABILE ȘI DATE IMUTABILE). Cu toate acestea, sintaxa Python conține o multitudine de metode menite să manipuleze sub o formă sau alta acest tip de dată. În continuare se vor exemplifica cele mai uzuale de metode și funcții asociate clasei `str`.

Inițializarea unui obiect de tip string se poate face în două feluri:

1. Utilizând un literal string:
   a. Ghilimele simple: `'ghilimele simple'`
   b. Ghilimele duble: `"ghilimele duble"`
   c. Trei ghilimele simple: `'''trei ghilimele simple'''`
   d. Trei ghilimele duble: `"""trei ghilimele duble"""`
2. Utilizând funcția constructor pentru a converti orice alt tip de dată în string: `str(100)`.

Pentru procesarea acestui tip de date, sintaxa Python suportă operații cu expresii formate din șiruri de caractere, cum ar fi: **concatenarea** (alipirea a două sau mai multe șiruri de caractere – `+`), **multiplicarea** (repetarea unui string de un număr de ori – `*`), **indexarea** (returnarea unui caracter a cărui poziție este indicată `[index]`), **partiționarea** (returnarea un subșir din șirul inițial – `[start:stop:pas]`), verificarea apartenenței (utilizând cuvântul cheie `in`). De menționat faptul că aceste operații sunt valide pentru orice obiect secvențial.

Trebuie specificată modalitatea de indexare și respectiv partiționarea (felierea) unui obiect secvențial. În vederea indexării se utilizează sintaxa: `obiect_secvential[index]`, unde între paranteze drepte, se trece un număr corespunzător poziției de pe care se dorește a se genera valoarea. Dacă se introduce un număr mai mare decât numărul total de elemente minus 1, atunci se va genera o excepție de tipul **`IndexError:`** `index out of range`. Trebuie reamintit faptul că toate obiectele secvențiale Python sunt indexate de la `0`, însemnând că lungimea totală a elementului va fi `număr de elemente-1` (de exemplu, pentru un string de **`10`** caractere, primul caracter va fi pe poziția **`0`** și ultimul caracter va fi pe poziția **`9`**, adică `10-1`). De asemenea, Python suportă indexarea negativă, ultimul element din secvență fiindu-i alocat numărul `-1`, penultimului `-2`, și așa mai departe.

```
sir_caractere = "Inteligenta Artificiala in Ingineria Energetica."
print(sir_caractere)
# indexarea stringului
primul_caracter = sir_caractere[0] # va returna prima litera - I
print(primul_caracter)
ultimul_caracter = sir_caractere[-1] # va returna ultimul caracter - .
print(ultimul_caracter)
OUTPUT:
I
.
```

În cazul partiționării, lucrurile funcționează similar, doar că sintaxa este mai complexă și conține trei elemente – `start` (caracterul de început al partiționării), `stop` (caracterul de final al partiționării) și `pas` (opțional – din câte în cate elemente se realizează partiționarea). Toate

aceste elemente sunt introduse între paranteze drepte și despărțite de caracterul ”:” - `obiect_secvential[start:stop:pas]`. Din aceleași raționamente de indexare ca în exemplele anterioare, elementul `stop` nu este inclus în partiționare, subelementul obținut având ultimul element de pe poziția `stop-1`.

```
sir_caractere = "Inteligenta Artificiala in Ingineria Energetica."
print(sir_caractere[12:24]) # genereaza subsir de la elementul 12 la 23
print(sir_caractere[12:24:3]) # genereaza subsir de la elementul 12 la 28 din 3 in 3 ele
OUTPUT:
Artificiala
Aicl
```

De asemenea, se pot utiliza valori negative pentru partiționare:

```
sir_caractere = "Inteligenta Artificiala in Ingineria Energetica."
print(sir_caractere[-11:-1]) # genereaza subsir de la elementul -11 la -2
OUTPUT:
Energetica
```

Toate elementele din partiționare pot fi omise:

- dacă se omite `start` - `obiect_secvential[:stop]` – returnează un subelement care pornește de la primul element și se oprește la `stop-1`.

```
sir_caractere = "Inteligenta Artificiala in Ingineria Energetica."
print(sir_caractere[:24]) # genereaza subsir de la elementul 0 la 23
OUTPUT:
Inteligenta Artificiala
```

- dacă se omite `stop` - `obiect_secvential[start:]` – returnează un subelement care pornește de la `start` și se oprește la ultimul element.

```
sir_caractere = "Inteligenta Artificiala in Ingineria Energetica."
print(sir_caractere[27:]) # genereaza subsir de la elementul 27 la final
OUTPUT:
Ingineria Energetica.
```

- dacă se omit toate - `obiect_secvential[:]` – returnează elementul intact.

```
sir_caractere = "Inteligenta Artificiala in Ingineria Energetica."
print(sir_caractere[:]) # genereaza subsir de la elementul 27 la final
OUTPUT:
Inteligenta Artificiala in Ingineria Energetica.
```

OBSERVAȚIE. Indexarea și partiționarea utilizând indecși negativi oferă posibilitatea returnării unui șir de caractere inversat relativ simplu (fără a fi nevoie de o parcurgere secvențială cu o buclă iterativă): se parcurge tot șirul cu un pas de -1 elemente. Alternativa (deși nu este singulară) este prezentată în cel de-al doilea exemplu și utilizează atât o buclă `for` cât și funcțiile `range()` - `f50` și `len()` - `f37`.

```
string = "Inteligenta Artificiala"
print("String initial: ", string)
string_inversat = string[::-1]
print("String inversat: ", string_inversat)
OUTPUT:
String initial:  Inteligenta Artificiala
String inversat:  alaicifitrA atnegiletnI
```

```
# alternativa la returnarea inversului unui string
string = "Inteligenta Artificiala"
print("String initial: ", string)
string_inversat2 = ""
for index in range(len(string)-1,-1,-1):
    string_inversat2+=string[index]

print("String inversat - metoda 2:", string_inversat2)
# se verifica daca cele doua stringuri sunt identice
print("Sunt identice? ", string_inversat == string_inversat2)
OUTPUT:
String initial:  Inteligenta Artificiala
String inversat - metoda 2: alaicifitrA atnegiletnI
Sunt identice?  True
```

În exemplele din fragmentul următor de cod sunt exemplificate restul operațiilor acceptate pentru manipularea șirurilor de caractere.

```
sir_caractere = "Inteligenta Artificiala in Ingineria Energetica."
print(sir_caractere)
# concatenarea cu un alt string
sir_concatenat = sir_caractere + 'Partea 1 - Stiinta Datelor'
print(sir_concatenat)
# multiplicarea stringului
sir_multiplicat = sir_caractere * 2
print(sir_multiplicat)
OUTPUT:
Inteligenta Artificiala in Ingineria Energetica.
Inteligenta Artificiala in Ingineria Energetica.Partea 1 - Stiinta Datelor
Inteligenta Artificiala in Ingineria Energetica.Inteligenta Artificiala in
Ingineria Energetica.
```

În vederea analizei apartenenței unui subșir (sau subelement) în șirul (elementul) inițial, se utilizează cuvântul cheie de apartenență `in`, sintaxa generală fiind: `sub-element in element` și va returna `True` doar dacă subelementul s-a găsit în elementul părinte. În exemplul din fragmentul următor de cod se creează o variabilă denumită `cuvant_cautat` care va aștepta un input de la utilizator prin utilizarea funcției `input()` – `f32` pentru care s-a trecut ca argument un mesaj:`'Ce cuvant sa cautam in stringul initial?'`. Această funcție primește date de la tastatură și le returnează în program sub formă de string, valoarea fiind stocată în variabila creată. Aceasta din urmă va fi folosită într-o sintaxă specifică Python care va crea o listă – denumită `raspuns` – populată cu un element în funcție de valoarea de adevăr a expresiei

cuvant_cautat in string: dacă (if) expresia este adevărată, lista va conține stringul 'Adevarat' și 'False' dacă (if) expresia este falsă. Fiind o listă – tip de dată secvențial, aceasta se poate indexa și, conținând un singur element, acesta se va afla pe poziția 0, sintaxa raspuns[0], returnând valoarea din listă. Rezultatele a trei rulări sunt prezentate în continuare.

```
cuvant_cautat = input('Ce cuvant sa cautam in stringul initial?')
raspuns = ['Adevarat' if cuvant_cautat in string else 'Fals']
print(f'Cuvantul "{cuvant_cautat}" este in sirul initial?', raspuns[0])
OUTPUT:
Cuvantul "Inteligenta" este in sirul initial? Adevarat
Cuvantul "Energie" este in sirul initial? Fals
Cuvantul "inginerie" este in sirul initial? Fals
```

OBSERVAȚIE. Python face diferența foarte clar între cuvinte scrise cu minuscule și cuvinte scrise cu majuscule. Astfel, în exemplul anterior chiar dacă șirul de caractere inițial conține cuvântul "inginerie", acesta este scris cu majusculă: "Inginerie". Drept urmare, când interpretorul Python va căuta primul cuvânt în stringul inițial, nu îl va găsi și va genera False, lista fiind populată cu stringul 'Fals'.

CARACTERUL IMUTABIL AL OBIECTELOR DE TIP STRING

Caracterul imutabil al obiectelor string minimizează posibilitatea apariției erorilor în programare datorită faptului că acestea nu pot fi modificate odată create. Spre exemplu, încercând să modificăm o literă prin atribuirea altei valori pe un index specificat, interpretorul Python va genera o eroare tip TypeError: 'str' object does not support item assignment, însemnând că obiectelor din această categorie nu le pot fi atribuite itemi.

```
text = 'test'
text[2] = 'x'
OUTPUT:
TypeError: 'str' object does not support item assignment
```

Printre avantajele folosirii unor obiecte imutabile putem enumera:

1. pot fi folosite drept chei în dicționare, asigurând faptul că maparea între chei și valori (seturile keys-value ce compun dicționarele Python) sunt unice, îndeplinind scopul acestora;

2. sunt mai eficiente din punct de vedere al ocupării memoriei în unitatea de procesare. Datorită faptului că nu li se poate schimba valoarea, Python poate optimiza utilizarea memoriei interne și rapiditatea procesării codului;

3. oferă predictibilitate în procesul de depanare al programului datorită faptului că valorile nu se pot modifica odată create.

Totuși, există câteva variante pentru modificarea unui obiect de tip string. Va trebui creat un string nou care să fie atribuit unei variabile care are același nume cu variabila inițială. Acest lucru este posibil deoarece în Python numele variabilelor nu sunt altceva decât pointeri către zone de memorie care conțin date. Cu alte cuvinte, este perfect legal în sintaxa Python să existe două variabile denumite identic, dar care conțin obiecte diferite. Intern, acestea se vor diferenția prin valoarea hash – valoarea numerică de dimensiune fixă care identifică în mod unic datele.

Totuși, această metodă este cel mai puțin de dorit deoarece nu face altceva decât să utilizeze aceeași denumire dată unei variabile care intern indică un cu totul alt obiect.

```
nume = 'Zora'
print(hash(nume))
nume = 'Iacob'
print(hash(nume))
OUTPUT:
-8032144648990746385
1615944485227428344
```

Există și alte metode de abordare a caracterului imutabil al obiectelor de tip `str` în Python, printre care putem aminti: partiționarea și reasamblarea, concatenarea, utilizarea metodei `replace()`, formatarea obiectelor de tip `string` și convertirea explicită la obiecte mutabile – cum ar fi un obiect tip `listă`. În continuare toate metodele vor și exemplificate.

```
# se creeaza un obiect de tip str si se atribuie unei variabile
text_initial = 'Retea Neuronala Artificiala'
print(text_initial)
print('ID string: ', hex(id(text_initial)))
# Metoda 1 - concatenarea
print('\nMetoda 1 - concatenarea')
text_modificat = text_initial + '- ANN'
print(text_modificat)
print('ID string: ', hex(id(text_modificat)))
# Metoda 2 - partitionarea si reasamblarea
print('\nMetoda 2 - partitionarea si reasamblarea')
text_modificat = text_initial[:5] + text_initial[15:]
print(text_modificat)
print('ID string: ', hex(id(text_modificat)))
# Metoda 3 utilizarea metodei replace()
print('\nMetoda 3 - utilizarea metodei replace()')
text_modificat = text_initial.replace(text_initial, 'ANN')
print(text_modificat)
print('ID string: ', hex(id(text_modificat)))
# Metoda 4 - convertirea la un obiect mutabil
print('\nMetoda 4 - convertirea la obiect mutabil si utilizarea metodei
join()')
text_modificat = list(text_initial)
text_modificat[:5] = 'ANN'
text_modificat = ''.join(text_modificat)
print(text_modificat)
print('ID string: ', hex(id(text_modificat)))
OUTPUT:

Retea Neuronala Artificiala
ID string:  0x22a7e98f730

Metoda 1 - concatenarea
```

```
Retea Neuronala Artificiala- ANN
ID string:  0x22a7e547db0

Metoda 2 - partitionarea si reasamblarea
Retea Artificiala
ID string:  0x22a7e5b3910

Metoda 3 - utilizarea metodei replace()
ANN
ID string:  0x22a7e92fbf0

Metoda 4 - convertirea la obiect mutabil
ANN Neuronala Artificiala
ID string:  0x22a7e936e70
```

METODE SPECIFICE STRINGURILOR

Fiind una dintre cele mai importante tipuri de date înglobate în sintaxa Python, pentru clasa `str` sunt definite o multitudine de metode care ușurează crearea programelor. O listă completă se regăsește în documentația oficială. În acest subcapitol vor fi exemplificate cele mai uzuale metode ale acestui tip de dată imutabilă. Merită menționat faptul că obiectele de tip string suportă toate operațiile specifice datelor secvențiale pe lângă metodele specifice.

`met_str.1.` `obiect_str.capitalize()` – returnează o copie a obiectului string cu primul caracter scris cu majuscule și restul cu minuscule. Asta înseamnă că dacă mai există vreun caracter scris cu majusculă în interiorul șirului de caractere, inclusiv acesta va fi transformat în minusculă.

```
obiect_str = 'inteligenta Artificiala'
print(obiect_str.capitalize())
OUTPUT:
Inteligenta artificiala
```

`met_str.2.` `obiect_str.casefold()` – returnează o copie fără majuscule a șirului de caractere original. Metoda este extrem de utilă dacă se dorește compararea a două sau mai multe șiruri de caractere scrise cu cazuri mixte de litere.

```
obiect_str = 'inteligenta ArtificIalA'
print(obiect_str.casefold())
OUTPUT:
inteligenta artificiala
```

`met_str.3.` `obiect_str.center(length, character)` – returnează o copie cu textul centrat a șirului de caractere original după ce umple spațiul suplimentar cu un caracter specificat. Metoda acceptă doi parametri: `length` – necesar și care descrie lungimea totală a șirului nou creat care conține centrat textul și `character` – opțional și specifică ce caracter este utilizat pentru a umple spațiul din jurul textului centrat; implicit este caracterul spațiu gol. În exemplul următor, un text cu o lungime de 5 caractere va fi centrat într-un șir nou de 10 caractere (`length` = 10) și spațiul gol umplut cu caractere ”-„. În cazul în care lungimea specificată este mai mică decât lungimea textului inițial stringul inițial este returnat.

```
obiect_str = 'teste'
print(obiect_str.center(10, '-'))
OUTPUT:
--teste---
```

met_str.4. obiect_str.count(value, start, end) – returnează un număr reprezentând numărul de apariții ale unei valori specificate în string-ul original sau într-un substring din acesta generat prin specificarea celor două argumente opționale: start – implicit este 0 și end – implicit este finalul string-ului. Trebuie avut grijă la faptul că metoda este case-sensitive, adică face diferența clară între majusculă și minusculă (A este diferit față de a).

```
obiect_str = 'Inteligenta Artificiala'
print('Literat "A" apare de', obiect_str.count('A'), 'ori.')
print('Litera  "i" apare de', obiect_str.count('i'), 'ori.')
print('Litera  "n" apare de', obiect_str.count('n', 3, 10), f'ori in subtrin-
gul [3:10] ({obiect_str[3:10]}).')
OUTPUT:
Literat "A" apare de 1 ori.
Litera  "i" apare de 4 ori.
Litera  "n" apare de 1 ori in subtringul [3:10] (eligent).
```

OBSERVAȚIE. Argumentul value poate avea inclusiv valoarea unui substring, nefiind limitat la specificarea unui caracter singular.

```
obiect_str = 'Inteligenta Artificiala'
print('Sub-stringul "ia" apare de', obiect_str.count('ia'), 'ori.')
OUTPUT:
Sub-stringul  "ia" apare de 1 ori.
```

met_str.5. obiect_str.endswith(sufix, start, end) – returnează True dacă obiectul string se termină cu un sufix specificat și False altfel. Funcția primește trei parametri: sufix (obligatoriu) – reprezintă valoarea sufixului șirului de caractere, start (opțional) – o valoare numerică întreagă specificând poziția de început al căutării, stop (opțional) – o valoare numerică întreagă specificând poziția de sfârșit al căutării. Utilizarea celor doi parametri opționali va genera căutarea într-un sub-string. Se utilizează similar cu metoda count().

```
# se creaza un string si se atribuie unei variabile
text_endswith = 'Inteligenta Artificiala!'
# se verifica daca textul se termina cu "!"
# printeaza Da sau Nu in consecinta
print('Da' if text_endswith.endswith('!') else 'Nu')
# se verifica daca un sub-text (caracterele de la 3 la 10) se termina cu "!"
# printeaza Da sau Nu in consecinta
print('Da' if text_endswith.endswith('!', 3, 10) else 'Nu')
OUTPUT:
Da
Nu
```

OBSERVAȚIE. Este complet legal să se folosească o expresie condiționată în sintaxa funcției `print()`. În exemplul anterior, se afișează pe ecranul consolei textul care este asociat cu valoarea de adevăr a expresiei `if - else`.

Există posibilitatea ca argumentul `sufix` să fie compus din mai multe valori, transmise sub forma unui tuplu de valori. Astfel, funcția va returna `True` dacă șirul de caractere are sufixul oricare valoare din tuplul respectiv. Această abordare este echivalentă cu: `obiect_str.endswith(suffix1) or obiect_str.endswith(suffix2)`.

```
# se creaza un string si se atribuie unei variabile
text_endswith = 'Inteligenta Artificiala!'
# se verifica daca sufixul este identic cu cel putin un element din tuplu
print('Da' if text_endswith.endswith(('!', 'eng', 'ent'), 3, 10) else 'Nu')
print(text_endswith[3:10]) # acest subsir cotinine literere eligent
OUTPUT:
Da
eligent
```

Această metodă reprezintă o variantă foarte simplă și rapidă de verificare a veridicității unei adrese de email sau a unui website pentru care știm cu siguranță sufixul. În exemplul de cod următor s-a detaliat crearea unui mic script ce acceptă input de la utilizator, verifică dacă are terminația corectă și transmite un mesaj în consecință. În exemplul următor sunt utilizate alte două metode specifice șirurilor de caractere despre care vom discuta în acest capitol: `lower()` și `rstrip ()`.

```
adresa_email = input('Introduceti adresa de email: ').lower()
if adresa_email.endswith('upb.ro'):
    print(f'{adresa_email.rstrip("@upb.ro")}, bun venit!')
else:
    print('Adresa de email incorecta.')
# 1. Se introduce carutasiu@upb.ro
# 2. Se introduce carutasiu@gmail.com
OUTPUT:
bogdan, bun venit!
Adresa de email incorecta.
```

`met_str.6.` `obiect_str.expandtabs(value)` – returnează o copie a stringului inițial cu valoarea caracterului specific pentru Tab (termenul folosit pentru alinierea textului într-un procesor de text prin deplasarea cursorului într-o poziție predefinită) setată la o valoare indicată. Tab face parte din însăși sintaxa Python prin intermediul indentării spațiului (Tab = 4 spații). Caracterul specific pentru tab este `'\t'`. De fiecare dată când interpretorul Python identifică acest caracter special îl înlocuiește cu 4 spații goale, exceptând cazul în care se utilizează metoda `expandtabs()` și se introduce o valoare pentru argumentul opțional `value` (implicit = `8`).

```
text_expandtabs = 'E\tn\te\tr\tg\ti\te'
print('Original:', text_expandtabs)
print('4 spatii:', text_expandtabs.expandtabs(4))
print('10 spatii:', text_expandtabs.expandtabs(10))
OUTPUT:
Original:  E   n    e   r    g   i    e
4 spatii: E   n   e   r   g   i   e
10 spatii: E         n         e         r         g         i         e
```

met_str.7. obiect_str.find(value, start, stop) – returnează indexul primei apariții a unei valori specificate. În cazul în care nu este găsită valoarea, metoda returnează valoarea -1. Parametrii opționali start și stop sunt utilizați pentru a genera un substring în care se caută valoarea. În cazul în care nu sunt specificate aceste argumente au valorile: start – 0 și stop – sfârșitul șirului/subșirului de caractere. În continuare sunt prezentate câteva exemple de utilizare a metodei, cu și fără specificarea argumentelor opționale. Merită menționat faptul că atât start cât și stop pot fi și valori negative – parcurgerea inversă a string-ului.

```
text_find = 'inteligenta artificiala!'
print('Primul "e" din text are indexul:', text_find.find('e'))
print("Primul 'i' din subtextul [10:] are indexul", text_find.find('i', 10))
print("Primul 'inteli' din subtextul [5:15] are indexul", text_find.find('in-
teli', 5, 15))
OUTPUT:
Primul "e" din text are indexul: 3
Primul 'i' din subtextul [10:] are indexul 15
Primul 'inteli' din subtextul [5:15] are indexul -1
```

Chiar dacă metoda find() returnează indexul primei apariții în text a subtextului specificat, în fragmentul de cod următor este prezentată un algoritm pentru determinarea numărului total de apariții ale valorii specificate într-un text. Se creează un text care conține mai multe instanțe ale cuvântului 'inteligent' – va fi folosit pentru căutare în textul inițial. Se inițializează un contor care va conține numărul total de apariții ale cuvântului căutat și un index care va specifica de unde să se înceapă căutarea. Într-o buclă for care va itera prin tot textul inițial, se caută indexul primei apariții a subtextului (index_gasit) și dacă expresia condiționată if va fi True, noul index de căutare va fi index_gasit + 1. De asemenea, contorul va fi incrementat cu o unitate, însemnând că acel cuvânt a fost găsit. În cazul în care subtextul nu se mai găsește, if devine False și contorul nu va mai fi incrementat.

```
# se creaza un text si se atribuie unei variabile
text_find = """Inteligenta artificiala reprezinta "inteligenta" cu are este
inzestrata o masinarie.
Prin "inteligenta" intelegem capacitatea de a rationaliza.
Un agent artificial 'inteligent' poate lua decizii de unul singur."""
# se indica subtextul de cautat
sub_text  = 'inteligent'
# se initiaza o variabila care va contoriza de cate ori apare in text subtex-
tul specificat
contor = 0
# se specifica indexul de la care se incepe cautarea (se porneste initial de
la primul caracter)
index_start = 0
for i in range(len(text_find)):
    index_gasit = text_find.find(sub_text, index_start)
    if index_gasit!=-1: # inseamna ca subtextul a fost gaist
        index_start = index_gasit+1 # se updateaza indexul de start sa caute
din locul de unde a gasit deja
        contor += 1 # daca a gasit un element, va incrementa contorul cu o
unitate

print(f'"{sub_text}" apare de {contor} ori in textul initial.')
OUTPUT:
"inteligent" apare de 3 ori in textul initial.
```

`met_str.8.` `obiect_str.format(value1, value2,...)` – returnează o variantă formatată a șirului de caractere. Metoda a fost detaliată în § *3.2. Exemple de utilizare a funcțiilor încorporate* prin funcția cu același nume `format()` - `f23`. În esență, metoda introduce valorile într-un substituent de text reprezentat în `string`-ul inițial prin acolade `{}`. Este o alternativă la utilizarea `f-string`. Există mai multe moduri de a injecta valorile în substituenții de text: prin specificarea unui nume al variabilelor, nume ce va trebui să se regăsească cu aceeași denumire ca argumente în metoda `format()`, prin specificația unor indecși – numere întregi care reprezintă ordinea argumentelor în metoda `format()`, fără a specifica nimic în substituent. În fragmentul de text propus sunt detaliate toate aceste posibilități. Modalitățile de formatare se pot analiza aici (site oficial) sau aici.

```
text_format1 = "Cartea se intituleaza {denumire}, autor
{nume_autor}.".format(denumire='Inteligenta Artificiala', nume_autor='Bogdan
Carutasiu')
text_format2 = "Cartea se intituleaza {0}, autor {1}.".format('Inginerie
Energetica', 'Bogdan Carutasiu')
text_format3 = "Cartea se intituleaza {}, autor {}.".format('Stiinta Datelor',
'Bogdan Carutasiu')
print(text_format1)
print(text_format2)
print(text_format3)
OUTPUT:
Cartea se intituleaza Inteligenta Artificiala, autor Bogdan Carutasiu.
Cartea se intituleaza Inginerie Energetica, autor Bogdan Carutasiu.
Cartea se intituleaza Stiinta Datelor, autor Bogdan Carutasiu.
```

met_str.9. obiect_str.format_map(dictioar_input) – returnează o copie formatată a șirului de caractere utilizând seturile cheie-valoare ale dicționarului folosit ca argument. Aceasta este singura diferențiere între format_map() și format(), cu precizarea faptului că și cea de-a doua poate fi folosită cu argument un dicționar, diferența fiind că format(**dictionar_input) copiază efectiv datele din dicționar din obiectul de mapare (utilizând **), în timp de format_map(dictionar_input) creează un nou dicționar la apelare.

```
fmap_dic = {'coordonata_x': 3, 'coordonata_y': -15}
text_fmap = "coordonatele {coordonata_x}, {coordonata_y}".format_map(fmap_dic)
print(text_fmap)
OUTPUT:
coordonatele 3, -15
```

met_str.10. obiect_str.index(value, start, stop) – returnează indexul pe care s-a găsit prima dată valoarea indicată în argument la apelarea metodei. Funcționează identic cu metoda find() - met_str.7, diferența constând în faptul că dacă valoarea nu este găsită în textul sau subtextul analizat, metoda index() va genera o eroare de tip ValueError: substring not found (metoda find() returnează -1). În multe cazuri, în practică este preferată utilizarea metodei index() datorită erorii generate și faptului că programul încetează execuția, eliminând alte probleme de logică în programare.

```
text_index = 'inteligenta artificiala!'
print('Primul "e" din text are indexul:', text_index.index('e'))
print("Primul 'i' din subtextul [10:] are indexul", text_index.index('i', 10))
print("Primul 'inteli' din subtextul [5:15] are indexul", text_index.in-
dex('inteli', 5, 15))
OUTPUT:
Primul "e" din text are indexul: 3
Primul 'i' din subtextul [10:] are indexul 15
----> 4 print("Primul 'inteli' din subtextul [5:15] are indexul", text_in-
dex.index('inteli', 5, 15))

ValueError: substring not found
```

met_str.11. obiect_str.isalnum() – returnează True doar dacă toate elementele ce constituie obiectul string sunt alfanumerice. Metoda nu acceptă niciun parametru. În primul exemplu din fragmentul următor de cod metoda returnează False deoarece, pe lângă caracterele alfanumerice, există și un spațiu gol – caracter non-alfanumeric. Același rezultat obținându-se în cazul apariției oricărui astfel de element (@, !, $, # etc). În cel de-al doilea caz, metoda returnează False, variabila isalnum_text conținând doar valori alfanumerice.

```
isalnum_text = 'Inteligenta Artificiala'
print('Textul contine doar caractere alfanumerice?', isalnum_text.isalnum())
isalnum_text = 'InteligentaArtificiala13'
print('Textul contine doar caractere alfanumerice?', isalnum_text.isalnum())
OUTPUT:
Textul contine doar caractere alfanumerice? False
Textul contine doar caractere alfanumerice? True
```

O foarte bună utilizare a acestei metode este în analiza parolei introduse de un utilizator în crearea sau editarea unui cont pe un site web sau într-o aplicație. Dată fiind necesitatea creării unor parole complexe, acestea se pot verifica dacă sunt compuse doar din elemente alfanumerice (lucru ce poate reduce gradul de siguranță oferit de parolă). Acest exemplu este detaliat în fragmentul următor de text. Se creează o variabilă denumită `parola` care acceptă input de la utilizator. Această variabilă este utilizată mai apoi într-o buclă while care va rula atât timp cât `parola.isalnum()` returnează `True` – utilizatorul introduce **doar** o parolă compusă exclusiv din caractere alfanumerice. În momentul în care metoda va returna `False`, valoarea este salvată în variabila `parola`. Trebuie menționat faptul că scripturile de verificare ale parolelor sunt mult mai complexe și înglobează mai multe straturi de siguranță.

```
parola = input("Introdu parola - trebuie sa contina cel putin un element non-alfanumeric: ")
while parola.isalnum():
    parola = input("Parola contine doar elemente alfanumerice - Incearca din nou: ")

print('Parola a fost setata', parola)
OUTPUT:
inteligentaA#13
```

`met_str.12. obiect_str.isalpha()` – returnează `True` doar dacă toate elementele ce constituie obiectul string sunt alfabetice (a-z). Metoda nu acceptă niciun parametru și se analizează analog cu `isalnum()`.

```
isalpha_text = 'InteligentaArtificiala13'
print('Textul contine doar caractere alfanumerice?', isalpha_text.isalpha())
isalpha_text = 'Inteligenta Artificiala'
print('Textul contine doar caractere alfanumerice?', isalpha_text.isalpha())
isalpha_text = 'Inteligenta'
print('Textul contine doar caractere alfanumerice?', isalpha_text.isalpha())
OUTPUT:
Textul contine doar caractere alfanumerice? False
Textul contine doar caractere alfanumerice? False
Textul contine doar caractere alfanumerice? True
```

O aplicație des întâlnită este aceea de verificare dacă un obiect de tip string este gol sau nu – metoda `isalpha()` aplicată unui string gol va returna `False` întotdeauna! O altă posibilă aplicație este de a curăța un text dat de toate caracterele non-alfabetice. Trebuie precizat că vor fi înlăturate inclusiv spațiile dintre cuvinte, făcând textul nou generat nelizibil.

```
# se creaza un text si se atribuie unei variabile
text_find = """Prin "inteligenta 2.0" intelegem capacitatea de a rationaliza!
Un agent artificial #inteligent poate lua decizii de unul singur."""
text_alfabetic = ''
for caracacter in text_find:
    if caracacter.isalpha():
        text_alfabetic += caracacter

print(text_alfabetic)
OUTPUT:
PrininteligentaintelegemcapacitateadearationalizaUnagentartifi-
cialinteligentpoateluadeciziideunulsingur
```

`met_str.13. obiect_str.isascii()` – returnează `True` doar dacă toate elementele ce constituie obiectul string sunt caractere ascii (a-z). Metoda nu acceptă niciun parametru și se analizează analog cu `isalnum()-met_str.11` și `isalpha()- met_str.12`.

`met_str.14. obiect_str.isdecimal()` – returnează `True` doar dacă toate elementele ce constituie obiectul string sunt caractere zecimale (0-9). Metoda nu acceptă niciun parametru și se analizează analog cu `isalnum()-met_str.11`, `isalpha()-met_str.12` și `isascii()-met_str.13`. Numerele utilizate drept super-script (a^3) și sub-script (a_2) sunt cifre, dar nu sunt considerate zecimale!

```
decimal_text = '119813'
print('Textul contine doar caractere zecimale?', decimal_text.isdecimal())
decimal_text = '119813 DC'
print('Textul contine doar caractere zecimale?', decimal_text.isdecimal())
decimal_text = '17 23'
print('Textul contine doar caractere zecimale?', decimal_text.isdecimal())
# caracterul ascii pentru '½'
text = '\u00BD'
print(f'Textul {text} contine doar caractere zecimale?', text.isdecimal())
OUTPUT:
Textul contine doar caractere zecimale? True
Textul contine doar caractere zecimale? False
Textul contine doar caractere zecimale? False
Textul ½ contine doar caractere zecimale? False
```

O întrebuințare des întâlnită a acestei metode este aceea de a verifica dacă un șir de caractere este un număr, caz în care acesta este convertit implicit la o dată numerică Python (int, float sau complex) – trebuie amintit faptul că dacă se încearcă o convertire a unui string alfanumeric într-o dată numerică, interpretorul Python va genera o eroare de tip. În exemplul următor se creează o funcție care primește drept argument un obiect de tip `string`. În interiorul funcției acest obiect este analizat cu ajutorul metodei `isdecimal()` și în cazul în care aceasta returnează `True` – toate elementele constituente sunt numere (0-9), atunci stringul poate fi convertit fără să genereze erori, iar funcția va returna un obiect de tip `int`. În caz contrar, funcția va returna `None`. Pentru verificare sunt oferite două exemple – un string zecimal și unul alfanumeric.

```
def convertor_numeric(obiect_string):
    if obiect_string.isdecimal():
        return int(obiect_string)
    else:
        return None

string_test = '1478'
print(f'valoarea int a stringului numeric "{string_test}" este {convertor_nu-
meric(string_test)}')

string_test = '1478$'
print(f'valoarea int a stringului numeric "{string_test}" este {convertor_nu-
meric(string_test)}')
OUTPUT:
valoarea int a stringului numeric "1478" este 1478
valoarea int a stringului numeric "1478$" este None
```

`met_str.15. obiect_str.isdigit()` – returnează `True` doar dacă toate elementele ce constituie obiectul string sunt digiți (inclusiv exponenți). Metoda nu acceptă niciun parametru și se analizează analog cu `isalnum()-met_str.11`, `isalpha()-met_str.12`, `isascii()-met_str.13` și `isdecimal()-met_str.14`. Numerele utilizate drept super-script (a^3) și sub-script (a_2) sunt cifre – tratate ca digiți în sintaxa Python!

```
digit_text = '119813'
print('Textul contine doar caractere zecimale?', digit_text.isdigit())
digit_text = '119813 DC'
print('Textul contine doar caractere zecimale?', digit_text.isdigit())
digit_text = '17.23'
print('Textul contine doar caractere zecimale?', digit_text.isdigit())
# caracterul ascii pentru '²3455'
text = '\u00B23455'
print(f'Textul {text} contine doar caractere zecimale?', text.isdigit())
OUTPUT:
Textul contine doar caractere zecimale? True
Textul contine doar caractere zecimale? False
Textul contine doar caractere zecimale? False
Textul ²3455 contine doar caractere zecimale? True
```

`met_str.16. obiect_str.isidentifier()`– returnează `True` obiectul este un identificator valid în sintaxa Python – v.§ *3. Particularități de sintaxă a limbajului de programare Python*. Un identificator Python reprezintă numele valid pe care îl pot lua variabilele, funcțiile și clasele definite. Acestea trebuie să conțină doar caractere alfanumerice (a-z și 0-9) și underscore (_). Totodată, un identificator valid nu poate începe cu un număr și nici conține alt caracter special în afară de underscore sau spații.

```
identificator = '#inteligenta_artificiala'
print(f'{identificator} poate fi folosit ca nume de variabila?
{identificator.isidentifier()}')
identificator = 'inteligenta_artificiala'
print(f'{identificator} poate fi folosit ca nume de variabila?
{identificator.isidentifier()}')
OUTPUT:
#inteligenta_artificiala poate fi folosit ca nume de variabila? False
inteligenta_artificiala poate fi folosit ca nume de variabila? True
```

`met_str.17. obiect_str.islower()`– returnează `True` dacă toate caracterele alfabetice dintr-un string sunt minuscule (litere mici). Trebuie menționat faptul că alte tipuri de caractere nu sunt procesate chiar dacă sunt constituent al șirului de caractere analizat.

```
lower_text = 'inteligenta artificiala in ingineria energetica - i.a. in i.e.'
print(f'textul este compus doar din minuscule? {lower_text.islower()}')
lower_text = 'inteligenta artificiala in ingineria energetica - I.A. in I.E.'
print(f'textul este compus doar din minuscule? {lower_text.islower()}')
OUTPUT:
textul este compus doar din minuscule? True
textul este compus doar din minuscule? False
```

`met_str.18. obiect_str.isnumeric()`– returnează `True` dacă toate caracterele sunt numerice – incluzând sub-scriptul (a_2), super-scriptul (a^2) și numerele fracționare (`2/3`). Cu toate acestea, numerele negative (`-10`) și cele flotante (`1.2`) nu sunt considerate numerice deoarece conțin caracterele ” - „ , respectiv ”. „.

```
exemplu1 = "\u0030" #unicode pentru 0
exemplu2 = "\u00B2" #unicode super-script 2;
exemplu3 = "10W"
exemplu4 = "-10"
exemplu5 = "1.2"
exemplu6 = '\u00BD' # caracterul ascii pentru '½'

print(f'"{exemplu1}" este numeric? {exemplu1.isnumeric()}')
print(f'"{exemplu2}" este numeric? {exemplu2.isnumeric()}')
print(f'"{exemplu3}" este numeric? {exemplu3.isnumeric()}')
print(f'"{exemplu4}" este numeric? {exemplu4.isnumeric()}')
print(f'"{exemplu5}" este numeric? {exemplu5.isnumeric()}')
print(f'"{exemplu6}" este numeric? {exemplu6.isnumeric()}')
OUTPUT:
"0" este numeric? True
"²" este numeric? True
"10W" este numeric? False
"-10" este numeric? False
"1.2" este numeric? False
"½" este numeric? True
```

OBSERVAȚIE. Diferența între `isnumeric()`, `isdigit()` și `isdigital()`. În esență, toate aceste metode sunt folosite pentru a determina dacă un string conține doar date numerice sau nu, returnând un obiect boolean – `True` în cazul în care caracterele sunt numerice și `False` altfel. Cu toate acestea, fiecare metodă are particularitățile ei, care au fost mai mult sau mai puțin evidențiate în momentul descrierii lor.

- `isdigital()` – `True` pentru: `0` – `9`
  – `False` pentru: numere sub-script (2_2), super-script (2^2) și fracții ($^2/_2$).
- `isdigit()` – `True` pentru: `0` – `9`, sub-script (2_2), super-script (2^2)
  – `False` pentru: numere fracționare ($^2/_2$).
- `isnumeric()` – `True` pentru: `0` – `9`, sub-script (2_2), super-script (2^2), numere fracționare ($^2/_2$).

```
expresie1 = "4²"
expresie2 = '½'
print(f"expresia {expresie1} isdigit()?", expresie1.isdigit())
print(f"expresia {expresie1} isdecimal()?", expresie1.isdecimal())
print(f"expresia {expresie1} isnumeric()?", expresie1.isnumeric())
print()
print(f"expresia {expresie2} isdigit()?", expresie2.isdigit())
print(f"expresia {expresie2} isdecimal()?", expresie2.isdecimal())
print(f"expresia {expresie2} isnumeric()?", expresie2.isnumeric())
OUTPUT:
expresia 4² isdigit()? True
expresia 4² isdecimal()? False
expresia 4² isnumeric()? True

expresia ½ isdigit()? False
expresia ½ isdecimal()? False
expresia ½ isnumeric()? True
```

`met_str.19. obiect_str.isprintable()`– returnează `True` dacă toate caracterele care compun șirul de caractere se pot afișa pe ecranul consolei și `False`, altfel. În Python, prin caracter imprimabil se înțelege un element din următoarele categorii: digiți (0 – 9), litere mari (A – Z), litere mici (a – z), caractere de punctuație (!”#$%&'()*+, -./:;?@[\]^_`{ | }~) și spațiul gol, în timp ce un caracter neimprimabil este reprezentat de caracterele care nu ocupă spațiu la afișare și care sunt utilizate doar pentru formatarea textului – spre exemplu: linie nouă (`\n`), tab (`\t`) etc.

```
printable_text = 'Inteligenta Artificiala \n - I.E.'
print(printable_text)
print('Textul este printabil?', printable_text.isprintable())

printable_text = 'Inteligenta Artificiala - I.E.'
print(printable_text)
print('Textul este printabil?', printable_text.isprintable())
OUTPUT:
Inteligenta Artificiala
 - I.E.
Textul este printabil? False
Inteligenta Artificiala - I.E.
Textul este printabil? True
```

Această metodă se poate utiliza pentru a identifica și înlocui tipul de caracter neimprimabil cu alt tip de caracter (spațiu gol, spre exemplu). În fragmentul de cod următor este prezentată modalitatea de creare a unei funcții care realizează acest lucru.

```
def inlucuire_neimprimabil(obiect_string, caracter_inlocuire):
    # se verifica daca caracterul folosit pentru inlocuire este imprimabil sau nu
    if caracter_inlocuire.isprintable() == False:
        caracter_inlocuire = ' '
    # se creaza un string gol care va fi alcatuit din elementele care sunt printabile
    string_modificat=''
    # contorul care va contoriza numarul de elemente neimprimabile
    contor = 0
    # se parcurge textul element cu element
    for caracter in obiect_string:
        # daca s-a gasit un caracter neimprimabil
        if caracter.isprintable() == False:
            # contorul se incrementeaza cu 1
            contor+=1
            # stringul gol va fi populat cu caracterul de inlocuire in loc de caracterul neimprimabil
            string_modificat +=caracter_inlocuire
        else:
            # in cazul in care caracterul este imprimabil, se alipeste stringului gol
            string_modificat += caracter
    # functia imi va returna un tuplu de valori:
    # index 0 - valoarea finala a variabilei contor
    # index 1 - stringul care are caracterele neimprimabile inlocuite
    return (contor, string_modificat)
# se genereaza un string cu 4 caractere neimprimabile (\n și \t)
printable_text = 'Inteligenta\tArtificiala\n\t- I.E.\t'
# se foloseste
caractere_neimprimabile =  inlucuire_neimprimabil(printable_text, ' ')[0]
print("Cate caratere neimprimabile are textul?", caractere_neimprimabile)
text_modificat = print(inlucuire_neimprimabil(printable_text, '*')[1])
OUTPUT:
Cate caratere neimprimabile are textul? 4
Inteligenta*Artificiala**- I.E.*
```

Funcția denumită `inlucuire_neimprimabil()` are definiți doi parametri: `obiect_string` și `caracter_inlocuire`, care reprezintă textul pe care dorim să îl analizăm și caracterul string cu care să fie înlocuit caracterul neimprimabil. Inițial se folosește o expresie condiționată pentru a ne asigura că noul caracter este imprimabil și poate fi folosit în scopul cerut – înlocuirea unui caracter neimprimabil. Se creează un string care inițial este gol `string_modificat` și care are rolul de container populat cu caracterele imprimabile din stringul original și cu cele de înlocuire și o variabilă denumită `contor` pentru a contoriza numărul de elemente neimprimabile. Șirul

original este apoi parcurs caracter cu caracter cu ajutorul unei bucle `for` și la fiecare iterație se verifică valoarea de adevăr a metodei **`isprintable()`** cu ajutorul unei expresii booleene `if`: dacă expresia este `True`, atunci caracterul din iterația curentă este alipit la noul string; dacă expresia este `False`, atunci în locul caracterului curent este alipit `caracter_inlocuire` la noul string și, în plus, contorul este incrementat cu o unitate. Finalmente, funcția va returna un tuplu format din două elemente: pe poziția indexului 0 – contorul și pe poziția indexului 1 – șirul de caractere cu elementele neimprimabile înlocuite. Pentru a testa funcția se utilizează un text cu 4 caractere neimprimabile și cu `'*'` pentru înlocuirea acestora. După cum se poate vedea în exemplul precedent, funcția returnează numărul 4 pentru numerele de caractere neimprimabile și textul cu `'*'` în locul lor.

`met_str.20. obiect_str.isspace()` – returnează `True` dacă toate caracterele care compun șirul de caractere sunt spații albe sau caractere neimprimabile (`' '`, `'\t'`, `'\n'`, `'\v'`, `'\f'`, `'\r'`) și `False`, altfel. Metoda este folosită pentru a analiza dacă anumite fișiere de text sunt goale.

```
text_gol = ' \n  '
print('Este textul gol?', text_gol.isspace())
text_plin = '  caracter    '
print('Este textul gol?', text_plin.isspace())
OUTPUT:
Este textul gol? True
Este textul gol? False
```

`met_str.21. obiect_str.istitle()` – returnează `True` dacă toate cuvintele din string încep cu majusculă urmată de minuscule (de exemplu: `Inteligența Artificială`). În primul exemplu din fragmentul următor de cod rezultatul va fi `False`, deoarece nu toate cuvintele încep cu majusculă, în timp ce în al doilea exemplu această condiție este îndeplinită. De asemenea, metoda ignoră orice alt tip de caracter, dar va returna `False` în cazul în care stringul este gol.

```
titlu_carte = 'Inteligenta Artificiala in Inginerie Energetica'
print('Este titlul cartii in regula?', titlu_carte.istitle())
titlu_carte = 'Inteligenta Artificiala - Inginerie Energetica'
print('Este titlul cartii in regula?', titlu_carte.istitle())
OUTPUT:
Este titlul cartii in regula? False
Este titlul cartii in regula? True
```

`met_str.22. obiect_str.isupper()` – returnează `True` dacă toate caracterele unui string sunt majuscule și `False`, în orice altă situație. De asemenea, metoda neglijează orice caracter non-alfabetic, așa cum reiese din primul exemplu din fragmentul de cod următor. Merită observat faptul că metoda a fost aplicată direct pe un string în cadrul funcției încorporate **`print()`**. Întâi se evaluează argumentul (metoda **`isupper()`** aplicată șirului de caractere și apoi este evaluată funcția cu acel argument.

```
print('I.A.'.isupper())
print('Inteligenta articiaiala'.isupper())
OUTPUT:
True
False
```

met_str.23. obiect_str.join(iterabil) – returnează un string format cu toate elementele unui obiect iterabil (listă, tuplu, string, dicționare, seturi). Obiectul string este folosit în acest caz drept separator între componentele string-ului format. Parametrul iterabil este necesar și poate fi orice obiect cu acest caracter. Trebuie menționat că toate elementele iterabilului trebuie să fie obiecte string, metoda neconvertind la string alte tipuri de date. În exemplul următor s-a încercat îmbinarea elementelor de tip int dintr-o listă, folosind caracterul ”.„ drept separator și interpretorul Python a generat o eroare de tip TypeError, indicând faptul că se așteaptă instanțe ale clasei str, dar au fost găsite obiecte int.

```
iterabil = [1, 2, 3, 4]
join_text = '.'.join(iterabil)
print(join_text)
OUTPUT:
TypeError: sequence item 0: expected str instance, int found
```

Dacă în schimb numerele sunt convertite la obiecte string, metoda va funcționa. În exemplul următor s-a utilizat o abordare specifică limbajului de programare Python pentru a crea liste cu o singură linie de cod (**list comprehension**) și toate elementele din lista inițială au fost convertite explicit la string cu funcția constructor str() și utilizată în conjuncție cu metoda join(). Obiectul string pe care se aplică metoda – și care servește drept element de legătură – poate fi orice șir de caractere (inclusiv caracterul spațiu gol), așa cum este exemplificat în exemplul următor.

```
iterabil = [1, 2, 3, 4]
str_interabil = [str(item) for item in iterabil]
join_text1 = '.'.join(str_interabil)
join_text2 = '-'.join(str_interabil)
join_text3 = 'energie'.join(str_interabil)
print(join_text1)
print(join_text2)
print(join_text3)
OUTPUT:
1.2.3.4
1-2-3-4
1energie2energie3energie 4
```

met_str.24. obiect_str.ljust(lungime, caracter) – returnează o copie a obiect_str aliniată la stânga (*left justified*). Parametrul lungime – obligatoriu – reprezintă cât de lung să fie stringul nou, în tip ce parametrul caracter – opțional – reprezintă caracterul ce va fi utilizat pentru a umple spațiul gol rezultat în urma alinierii (implicit, caracter este un spațiu gol). Trebuie precizat faptul că parametrul lungime descrie lungimea TOTALĂ a noului șir de caractere, incluzând lungimea obiect_str. Dacă obiect_str conține 4 caractere, și lungime are valoarea 10, rezultă că stringul aliniat la stânga returnat va avea 6 caractere specificate. Acest lucru este exemplificat în fragmentul următor de cod, unde se analizează inclusiv lungimea celor două stringuri. Metoda se utilizează specificând caracter == '*'. Merită precizat faptul că dacă parametrul lungime are o valoare mai mică decât string-ul original (obiect_str), atunci metoda va returna chiar string-ul original.

```
text_original = 'Energetica'
print("Lungimea textului original:", len(text_original))
text_aliniatstanga = text_original.ljust(15, '*')
print('Lungimea textului aliniat la stanga:', len(text_aliniatstanga))
print(text_aliniatstanga)
OUTPUT:
Lungimea textului original: 10
Lungimea textului aliniat la stanga: 15
Energetica*****
```

met_str.25. `obiect_str.lower()` – returnează o copie a stringului original, în care toate caracterele sunt minuscule. Orice alt tip de dată în afară de literele alfabetului este ignorat.

```
text_original = 'Inteligenta Artificiala-Ingineria Energetica - I.A. IN I.E.'
text_minuscule = text_original.lower()
print(text_original)
print(text_minuscule)
OUTPUT:
Inteligenta Artificiala-Ingineria Energetica - I.A. IN I.E.
inteligenta artificiala-ingineria energetica - i.a. in i.e.
```

met_str.26. `obiect_str.lstrip(caracter(e))` – returnează o copie a `string`-ului original în care toate caracterele introduse prin parametrul `caracter(e)` din stânga `string`-ului sunt îndepărtate. Implicit, parametrul `caracter` este spațiu gol. În fragmentul următor de cod sunt prezentate două exemple: unul prin care se îndepărtează toate caracterele spațiu gol din stânga cuvântului `'energie'` – echivalentul a două tab-uri și un exemplu prin care se îndepărtează sufixul unui site-web – `'https://'`, afișându-se pe ecranul consolei atât textul original cât și rezultatul aplicării metodei `lstrip()`.

```
text_original = '            energie'
print(text_original)
text_lstrip = text_original.lstrip()
print(text_lstrip)

denumire_website = 'https://www.upb.ro'
print(denumire_website)
denumire_website_lstrip = denumire_website.lstrip('https://')
print(denumire_website_lstrip)
OUTPUT:
            energie
energie
https://www.upb.ro
www.upb.ro
```

met_str.27. `obiect_str.partitions(separator)` – returnează un tuplu compus din trei elemente: 1 – partea din `obiect_str` de **până** în parametrul `separator`, 2 – `string`-ul `separator` și 3 – partea din `obiect_str` de **după** parametrul `separator`. Metoda aplică această logică pentru prima apariție a parametrului `separator` în text. Dacă metoda nu găsește parametrul `separator` în text, atunci va returna două `string`-uri goale și parametrul în sine.

Ambele situații sunt detaliate în fragmentul de cod următor, unde, în primul exemplu metoda este utilizată cu argumentul `separator` egal cu `' - '`, care este diferit față de `'-'` prin existența spațiilor de dinainte și de după `-`. Drept urmare, prima apariție a caracterului `-` nu va fi folosită și partiționarea obiectului string se va realiza în funcție de a doua apariție a sa. Mai mult, trebuie avut în vedere faptul că Python este un limbaj de programare case sensitive, făcând foarte clar distincția între litere mici și majuscule atunci când se realizează partiționarea.

```
text_original = 'Inteligenta Artificiala-Ingineria Energetica - I.A. IN I.E.'
print(text_original)
partitionare = text_original.partition(' - ')
print(partitionare)

partitionare = text_original.partition('10')
print(partitionare)
OUTPUT:
Inteligenta Artificiala in Ingineria Energetica-I.A. IN I.E.
('Inteligenta Artificiala-Ingineria Energetica', ' - ', 'I.A. IN I.E.')
('Inteligenta Artificiala-Ingineria Energetica - I.A. IN I.E.', '', '')
```

`met_str.28. obiect_str.replace(substring_vechi, substring_nou, numar)` – returnează o copie a stringului original în care parametrul obligatoriu `substring_vechi` este înlocuit cu parametrul obligatoriu `substring_nou`. Parametrul opțional `numar` indică pentru câte apariții ale stringului `substring_vechi` se realizează înlocuirea cu `substring_nou` (implicit se realizează pentru toate aparițiile stringului `substring_vechi` în textul din `obiect_str`). În exemplul următor stringul `'Inteligenta Artificiala'` este înlocuit cu stringul `'IA'`. Merită menționat faptul că stringul original rămâne nemodificat (este imutabil).

```
text_original = 'Inteligenta Artificiala - Inteligenta Artificiala in
Ingineria Energetica - I.A. IN I.E.'
print(text_original)
element_vechi = 'Inteligenta Artificiala'
element_nou = 'IA'
# se modifica toate substringurile
text_modificat = text_original.replace(element_vechi, element_nou)
print(text_modificat)
# se modifica doar primul substring
text_modificat = text_original.replace(element_vechi, element_nou, 1)
print(text_modificat)
OUTPUT:
Inteligenta Artificiala - Inteligenta Artificiala in Ingineria Energetica -
I.A. IN I.E.
IA - IA in Ingineria Energetica - I.A. IN I.E.
IA - Inteligenta Artificiala in Ingineria Energetica - I.A. IN I.E.
```

`met_str.29. obiect_str.rfind(valoare, start, stop)` – returnează poziția (indexul) ultimei (cea mai din dreapta) apariții a unui substring indicat prin parametrul `valoare`. Dacă valoarea nu este găsită, metoda va returna `-1`. Parametrii `start` și `stop` sunt opționali și, dacă sunt specificați, indică începutul și sfârșitul zonei de căutare. Implicit, `start` este 0 și `stop` este `len(obiect_str) - 1` – sfârșitul șirului de caractere analizat (`obiect_str`). Se folosește în

mod similar cu metoda find() – met_str.7, diferența fiind că a doua returnează indexul primei apariții a substringului. În exemplul următor sunt exemplificate ambele metode.

```
text_original = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce nulla dolor, ipsum cursus vel ipsum vitae, ipsum convallis semper dui'
print(text_original)
cuvant_cautat = 'ipsum'
print('ultima paritie a cuvantului cautat este la indexul:', text_original.rfind(cuvant_cautat))
print('prima paritie a cuvantului cautat este la indexul:', text_original.find(cuvant_cautat))
print('ultima paritie a cuvantului "inginer" este la indexul:', text_original.rfind("inginer"))
OUTPUT:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce nulla dolor, ipsum cursus vel ipsum vitae, ipsum convallis semper dui
ultima paritie a cuvantului cautat este la indexul: 106
prima paritie a cuvantului cautat este la indexul: 6
ultima paritie a cuvantului "inginer" este la indexul: -1
```

met_str.30. obiect_str.rindex(valoare, start, stop) – returnează cel mai mare index al argumentului valoare (ultima apariție). Faptul că metoda rfind() returnează -1 în condițiile în care elementul nu este găsit, poate fi o sursă de apariție a erorilor de logică în dezvoltarea diferitelor aplicații. O alternativă este utilizarea metodei rindex() care funcționează analog, dar generează o eroare de tip ValueError: substring not found în cazul în care nu găsește elementul în șirul de caractere analizat. În cazul în care substringul este găsit, cele două metode se comportă identic.

```
text_original = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce nulla dolor, ipsum cursus vel ipsum vitae, ipsum convallis semper dui'
print(text_original)
cuvant_cautat = 'ipsum'
print('ultima paritie a cuvantului cautat este la indexul:', text_original.rfind(cuvant_cautat))
print('prima paritie a cuvantului cautat este la indexul:', text_original.rindex(cuvant_cautat))
print('ultima paritie a cuvantului "inginer" este la indexul:', text_original.rindex("inginer"))
OUTPUT:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce nulla dolor, ipsum cursus vel ipsum vitae, ipsum convallis semper dui
ultima paritie a cuvantului cautat este la indexul: 106
prima paritie a cuvantului cautat este la indexul: 106
ValueError: substring not found
```

```
text = "Inteligenta Artificiala"
pozitie_element_gasit = text.rindex("a", 3, 13)
print(pozitie_element_gasit)
OUTPUT:
10
```

Drept aplicație practică, ambele metode se pot folosi pentru analiza domeniului unui site web sau a unei adrese de e-mail, după cum este prezentat în fragmentul următor de cod.

```
def verifica_email(adresa_email):
    domeniul = adresa_email.rfind('@', 1)
    tld_string = adresa_email[domeniul:]

    if tld_string == "@upb.ro":
        return "Face parte din domeniul mail@upb.ro"
    else:
        return f"Nu face parte din domeniul mail@upb.ro: {tld_string}"

email1 = 'mihai.carutasiu@upb.ro'
email2 = 'carutasiu@gmail.com'

print(verifica_email(email1))
print(verifica_email(email2)
OUTPUT:
Face parte din domeniul mail@upb.ro
Nu face parte din domeniul mail@upb.ro: @gmail.com
```

`met_str.31. obiect_str.rjust(lungime, caracter(e))` – returnează o copie a `obiect_str` aliniată la dreapta (*right justified*). Se analizează analog cu metoda `ljust()` - `met_str.24` care returnează o copie a `obiect_str` aliniată la stânga (*left justified*).

`met_str.32. obiect_str.rpartition(separator)` – returnează un tuplu care conține trei elemente formate din împărțirea șirului de caractere în funcție de parametrul `separator`. Singura diferență dintre metoda `rpartition()` și `partition()` – `met_str.27` este aceea că `rpartition()` folosește ultima apariție a parametrului `separator` în textul dat, în timp ce `partition()` folosește prima apariție a acestuia. Analiza este analoagă.

`met_str.33. obiect_str.rsplit(separator, maxplit)` – returnează o listă ce conține caracterele `string`-ului inițial. Parametrul `separator` (opțional – implicit este spațiu) indică elementul care se folosește drept delimitator, metoda împărțind caractere în funcție de acesta. Metoda împarte șirul de caractere din dreapta acestuia (*right split*). Parametrul `maxsplit` (opțional – implicit este `-1`) indică numărul maxim de împărțiri. Dacă acest parametru este specificat, lista returnată va avea un număr de maximul `maxsplit+1` itemi.

```
text_original = 'Inteligenta Artificiala-Ingineria Energetica - I.A. IN I.E.'
text_rsplit = text_original.rsplit()
print('String-ul impartit este: ', text_rsplit)
print('String-ul impartit are lungimea', len(text_rsplit))

print(verifica_email(email1))
print(verifica_email(email2)
OUTPUT:
String-ul impartit este:  ['Inteligenta', 'Artificiala-Ingineria', 'Energetica', '-', 'I.A.', 'IN', 'I.E.']
String-ul impartit are lungimea 7
```

În exemplul anterior se folosește metoda `rsplit()` fără a utiliza vreun argument opțional și returnează o listă ce conține toate elementele din șirul inițial separate de spațiu, însemnând șapte itemi – atâtea cuvinte câte sunt în stringul inițial. În exemplul următor se utilizează drept separator caracterul `-`, rezultând o listă cu doar trei elemente. În cel de-al treilea exemplu este utilizat și cel de-al doilea argument opțional – `maxplit` – care va indica numărul maxim de împărțiri, rezultând `maxplit+1` elemente.

```
text_original = 'Inteligenta Artificiala-Ingineria Energetica - I.A. IN I.E.'
text_rsplit = text_original.rsplit(sep='-')
print('String-ul impartit este: ', text_rsplit)
print('String-ul impartit are lungimea', len(text_rsplit))

text_original = 'Inteligenta Artificiala-Ingineria Energetica - I.A. IN I.E.'
text_rsplit = text_original.rsplit(sep='-', maxsplit=1)
print('String-ul impartit este: ', text_rsplit)
print('String-ul impartit are lungimea', len(text_rsplit))
OUTPUT:
String-ul impartit este: ['Inteligenta Artificiala', 'Ingineria Energetica ',
' I.A. IN I.E.']
String-ul impartit are lungimea 3
String-ul impartit este: ['Inteligenta Artificiala-Ingineria Energetica ', '
I.A. IN I.E.']
String-ul impartit are lungimea 2
```

`met_str.34. obiect_str.rstrip(caractere)` – returnează o copie a stringului inițial fără spațiile goale (sau `caractere` – dacă este menționat parametrul) de la sfârșitul său (*right strip*).

```
text_original = 'Inteligenta Artificiala-Ingineria Energetica -      '
print(text_original, len(text_original))
text_rstrip = text_original.rstrip()
print(text_rstrip, len(text_rstrip))
OUTPUT:
Inteligenta Artificiala-Ingineria Energetica -       51
Inteligenta Artificiala-Ingineria Energetica - 46
```

```
text_original = 'Inteligenta Artificiala-Ingineria Energetica --------'
print(text_original, len(text_original))
text_rstrip = text_original.rstrip('-')
print(text_rstrip, len(text_rstrip))
OUTPUT:
Inteligenta Artificiala-Ingineria Energetica -------- 53
Inteligenta Artificiala-Ingineria Energetica  45
```

`met_str.35. obiect_str.split(separator, maxplit)` – returnează o listă ce conține caracterele `string`-ului inițial. Funcționează analog cu `rsplit()`, singura diferență fiind cazul în care se utilizează parametrul opțional `maxplit` – metoda `split()` separă din stânga parametrului, în timp de `rsplit()` din dreapta acestuia, așa cum reiese din fragmentul de cod:

```
text_original = 'I.A. - Ingineria Energetica - I.A. IN I.E.'
text_rsplit = text_original.rsplit(sep='-', maxsplit=1)
text_split = text_original.split(sep='-', maxsplit=1)
print('Cu metoda rplit: ', text_rsplit)
print('Cu metoda split: ', text_split)
OUTPUT:
Cu metoda rplit:  ['I.A. - Ingineria Energetica ', ' I.A. IN I.E.']
Cu metoda split:  ['I.A.', 'Ingineria Energetica - I.A. IN I.E.']
```

met_str.36. obiect_str.**splitlines**(keeplinebreakers) – returnează o listă pentru fiecare linie a stringului inițial. Parametrul opțional keeplinebreakers specifică dacă rupturile de linie sunt incluse (True) sau nu (False) – implicit este False. În Python, o nouă linie într-un șir de caractere este indicat prin specificarea caracterului special newline: '\n', care va indica interpretorului Python să treacă pe linia următoare într-un text. Într-un editor de text specific și matur (precum Microsoft Word sau Notepad), caracterul newline este realizat prin simpla apăsare a tastei Enter. În exemplul de cod următor se creează un string care conține 3 linii evidențiate prin existența a două caractere newline. Utilizând metoda **splitlines**() cu și fără menționarea argumentului opțional, se va genera o listă ce conține trei elemente – fiecare linie a șirului inițial. Prin specificarea argumentului opțional se poate observa faptul că primele două linii conțin inclusiv caracterele specifice newline.

```
text_original = 'Inteligenta Artificiala\nIngineria Energetica\nI.A. IN I.E.'
print(text_original)
text_splitlines = text_original.splitlines()
print(text_splitlines)

text_splitlines = text_original.splitlines(True)
print(text_splitlines)
OUTPUT:
Inteligenta Artificiala
Ingineria Energetica
I.A. IN I.E.
['Inteligenta Artificiala', 'Ingineria Energetica', 'I.A. IN I.E.']
['Inteligenta Artificiala\n', 'Ingineria Energetica\n', 'I.A. IN I.E.']
```

met_str.37. obiect_str.**startswith**(value, start, stop) – returnează True dacă șirul/subșirul de caractere începe cu caracterul/caracterele indicate prin parametrul value și False altfel – implicit este False. Parametrii start (implicit 0) și stop (implicit **len**(obiect_str) - 1) sunt opționali și specificarea lor permite generarea unui subșir de caractere din șirul inițial. Merită menționat faptul că parametrul value poate fi și sub forma unui tuplu de valori și metoda va returna False dacă stringul/substringul începe cu oricare dintre valorile specificate în tuplu.

```
text_original = 'Inteligenta Artificiala-Ingineria Energetica-I.A. IN I.E.'
print('Textul incepe cu "ingeli"?', text_original.startswith('inteli'))
print('Textul[12:20] incepe cu "Ar"?', text_original.startswith('Ar', 12, 20))
OUTPUT:
Textul incepe cu "ingeligenta"? False
Substringul dintre 12:20 incepe cu "Ar"? True
```

```
string = "stiinta datelor"
rezultat = string.startswith(('a', 'e', 'i', 'o', 'u'))
print(rezultat)
OUTPUT:
False
```

met_str.38. `obiect_str.strip(carctere)` – returnează o copie a stringului inițial fără spațiile goale (sau `caractere` – dacă este menționat parametrul) indiferent dacă sunt la începutul sau sfârșitul textului. Aceasta este singura diferență între această metodă și metoda analizată anterior `rstrip()`.

```
text = "_____inginer_____"
text_strip = text.strip("_")
print('Cu metoda strip(): ', text_strip)

text_rstrip = text.rstrip("_")
print('Ce metoda rstrip():', text_rstrip)
OUTPUT:
Cu metoda strip():  inginer
Ce metoda rstrip(): _____inginer
```

met_str.39. `obiect_str.swapcase()` – returnează o copie a stringului în prima literă este majusculă, exact ca un titlu. Metoda nu primește niciun parametru și ignoră orice alt caracter în afară de caracterele alfabetice.

```
text_original = 'Inteligenta Artificiala - Ingineria Energetica'
print(text_original)
print(text_original.swapcase())
OUTPUT:
Inteligenta Artificiala - Ingineria Energetica
iNTELIGENTA aRTIFICIALA - iNGINERIA eNERGETICA
```

met_str.40. `obiect_str.title()` – returnează o copie a stringului în care literele mici sunt transformate în majuscule și viceversa. Metoda nu primește niciun parametru și ignoră orice alt caracter în afară de caracterele alfabetice.

```
text_original = 'inteligenta Artificiala - Ingineria Energetica'
print(text_original)
print(text_original.title())
OUTPUT:
inteligenta Artificiala - Ingineria Energetica
Inteligenta Artificiala - Ingineria Energetica
```

met_str.41. `obiect_str.upper()` – returnează o copie a stringului care conține toate caracterele alfabetice majuscule. Metoda nu primește niciun parametru și ignoră orice alt caracter în afară de caracterele alfabetice.

```
text_original = 'inteligenta Artificiala - Ingineria Energetica'
print(text_original)
print(text_original.upper())
OUTPUT:
inteligenta Artificiala - Ingineria Energetica
INTELIGENTA ARTIFICIALA - INGINERIA ENERGETICA
```

met_str.42. obiect_str.zfill(len) – returnează o copie a stringului în care sunt adăugate cifre 0 până când șirul de caractere va ajunge la lungimea specificată prin parametrul len – parametru necesar. În exemplul următor este inițializată o variabilă de tip string care are lungimea 11 caractere. Utilizarea funcției zfill() cu argumentul 20 returnează un nou șir de caractere care are lungimea totală 20, dintre care 8 cifre 0.

```
text_original = 'inteligenta'
print(text_original, 'are lungimea', len(text_original))
print(text_original.zfill(20), 'are lungimea', len(text_original.zfill(20)))
OUTPUT:
inteligenta are lungimea 11
000000000inteligenta are lungimea 20
```

4.3. Liste: list

Listele reprezintă o altă categorie de date **secvențiale** Python, având anumite similarități cu șirurile de caractere. Listele reprezintă obiecte neomogene care permit stocarea simultană a mai multor tipuri date (int, float, string, obiecte, liste, dicționare etc), fiind în esență cele mai simple containere din sintaxa limbajului Python. Mai mult, listele pot conține duplicate. Sunt comparabile cu matricele sau vectorii (tablouri de date) cu dimensiuni dinamice din matematică și care prezintă anumite caracteristici specifice. Listele Python sunt:

↔ indexabile – pot fi accesate prin specificarea indecșilor;
↔ ordonate – ordinea elementelor se păstrează chiar și după manipulare;
↔ dinamice – li se pot modifica oricând numărul de elemente;
↔ eterogene – pot conține obiecte de diverse tipuri;
↔ mutabile – pot fi modificate după ce au fost create;
↔ concatenabile – pot fi imbricate mai multe liste (adunate, multiplicate etc.);
↔ multidimensionale – pot avea mai mult de o dimensiune (vectori/matrice).

Pentru a instanția o listă în Python este suficient de a specifica numele variabilei (indicatorului) care va indica spațiul din memorie unde va fi stocată lista, urmat de funcția constructor specifică list() - f38 sau de două paranteze drepte – []. În exemplul următor sunt create două liste goale utilizând ambele metode de inițializare.

```
lista_mea = list()
lista_mea = []
print(type(lista_mea))
OUTPUT:
<class 'list'>
```

În exemplul din fragmentul următor cod este creată o listă care conține toate tipurile de date încorporate din Python și utilizarea funcției `len()` pentru a returna lungimea acesteia – numărul total de elemente conținute. Mai mult, lista este indexată pentru a extrage anumite elemente (care mai apoi, pot fi atribuite unor variabile).

```
lista_elemente = [list(), int(), float(), str(), set(), frozenset(), com-
plex(), dict()]
print(f'Lista are {len(lista_elemente)} elemente.' )
# se indexeaza lista pentru a se extrage obiectul 3 - index 2
elementul_3 = lista_elemente[2]
print('Al treilea element - indexul 2 este:', elementul_3)
# se felieaza lista - ultimul index nu este inclus!
elemente_3_5 = lista_elemente[2:5]
print('Elemtele 3, 4 si 5 sunt:', elemente_3_5)
OUTPUT:
Lista are 8 elemente.
Al treilea element - indexul 2 este: 0.0
Elemtele 3, 4 si 5 sunt: [0.0, '', set()]
```

OBSERVAȚIE 1. Ca și în cazul obiectelor tip string și listele sunt indexabile din 0.

OBSERVAȚIE 2. Merită observat faptul la că utilizarea funcțiilor constructor specifice pentru diverse tipuri de date, unele mărimi sunt obiecte goale ale acelor tipuri de date - `float()` returnează `0.0`, `str()` returnează `''`, etc.

LIST COMPREHENSION – COMPREHENSIUNEA LISTELOR

Există și o a treia metodă de creare a listelor specifică limbajului de programare Python, și anume utilizarea mecanismului de comprehensiune (*list comprehension*). Această abordare permite utilizarea unei sintaxe mai scurte (scrisă într-o linie de cod) când se dorește generarea unor liste pe baza unor elemente iterabile existente, putând fi inclusiv procesate în baza unor expresii logice. Sintaxa generală este de forma:

```
lista = [expresie1 for item in iterabil if conditie == True]
```

În exemplul din fragmentul următor de cod este inițializat un string care este atribuit variabilei `text` – acesta va fi utilizat drept `iterabil`. Folosindu-se sintaxa pentru crearea unei liste comprehensive, se generează o listă care va fi atribuită variabilei `vocale`. Practic, se iterează în interiorul listei prin iterabil, aplicând condiția pentru fiecare element – condiția pentru acest caz specific este ca itemul curent să aparțină uneia dintre valorile specificate în lista cu vocale (`lista_vocale = ['a', 'e', 'i', 'o', 'u']`). În cazul în care condiția va returna `True` (itemul este o vocală), valoarea analizată la iterația curentă va fi alipită listei, în caz contrar itemul va fi ignorat. Se afișează pe ecranul terminalului componența listei.

```
text = 'Inteligenta Artificial - I.A. in I.E.'
lista_vocale = ['a', 'e', 'i', 'o', 'u']
vocale_text = [litera for litera in text if litera.lower() in lista_vocale]
print(vocale_text)
OUTPUT:
['I', 'e', 'i', 'e', 'a', 'A', 'i', 'i', 'i', 'a', 'I', 'A', 'i', 'I', 'E']
```

Un alt exemplu concret este utilizarea comprehensiunii listelor pentru a popula o listă cu valorile în grade `Fahrenheit` și `Kelvin` dintr-o listă cu valorile în grade `Celsius`. Acestea vor fi populate sub forma unei liste interioare. În plus, pentru o lizibilitate crescută, se poate utiliza și funcția încorporată `round()` - `f53` cu argumentul `3` (număr maxim de zecimale).

```
Celsius = [34.2, 0, 100.3, 75]
F_K=[[round(((float(9)/5)*t + 32), 3), round((t+273.15), 3)] for t in Celsius]
print(F_K)
OUTPUT:
[[93.56, 307.35], [32.0, 273.15], [212.54, 373.45], [167.0, 348.15]]
```

Un alt exemplu practic pentru a accentua utilizabilitatea acestui mecanism este de generare a unei liste ce conține tupluri (sau alte liste imbricate) formate din numere pitagoreice (triplete de numere naturale ce verifică relația din teorema lui Pitagora). Fragmentul de cod următor prezintă o astfel de posibilitate. Se definește inițial o variabilă care are rolul de număr maxim pentru a genera un interval numeric (utilizând funcția `range()` - `f50`) în care se caută tripletele pitagoreice. Se generează o listă utilizând mecanismul de comprehensiune și se verifică dacă relația lui Pitagora este îndeplinită, utilizând-o drept expresie condiționată `if`.

```
limita_maxima = 20
numere_pitagoreice = [[x, y, z] for x in range(1,limita_maxima) for y in
range(x, limita_maxima) for z in range(y, limita_maxima) if x**2 + y**2 ==
z**2]
print(numere_pitagoreice)
OUTPUT:
[[3, 4, 5], [5, 12, 13], [6, 8, 10], [8, 15, 17], [9, 12, 15]]
```

Se poate folosi o expresie condiționată de tipul `if` – `else` pentru a opera două expresii care alterează datele înainte de a le alipi la listă, dar sintaxa se va modifica, având forma:

```
lista = [expresie1 if conditie == True else expresie2 for item in iterabil]
```

În fragmentul următor de cod este prezentat un astfel de exemplu. Se creează inițial o listă care conține o serie de numere întregi. Utilizând sintaxa anterioară, numerele impare sunt ridicate la pătrat și alipite listei denumite `patrate_impare`. Condiția impusă este ca itemul din iterația curentă să aibă rezultatul împărțirii numărului la 2 diferit de 0 – însemnând că număr este impar (`numar%2 != 0`). Dacă expresia condiționată returnează `False`, în locul numărului original se alipește la listă rezultatul expresiei `numar-1` – practic generând numere impare.

```
lista_numere = [1, 14, 18, 13, 51, 8, 10, 17]
impare = [numar**2 if numar%2 != 0 else numar-1 for numar in lista_numere]
print(impare)
OUTPUT:
[1, 13, 17, 169, 2601, 7, 9, 289]
```

INDEXAREA LISTELOR

În esență, o listă simplă poate fi asimilată cu un vector din matematică, dar există și posibilitatea de a crea liste imbricate, rezultând în matrice (dacă au două dimensiuni), cuburi (cu trei

dimensiuni) sau tensori (peste 4 dimensiuni). În continuare sunt prezentate liste imbricate de diverse dimensiuni și modalitatea de indexare a acestora. Indexarea elementelor în cazul acestor tipuri de liste se face de la lista cea mai exterioară către interior (cum este simbolizat în figura 16). Presupunând faptul că `lista_1` este cea exterioară și `lista_n` cea mai interioară indexarea va fi:

`lista_nD[index_lista_1][index_lista_2][index_lista_3]...[index_lista_n].`

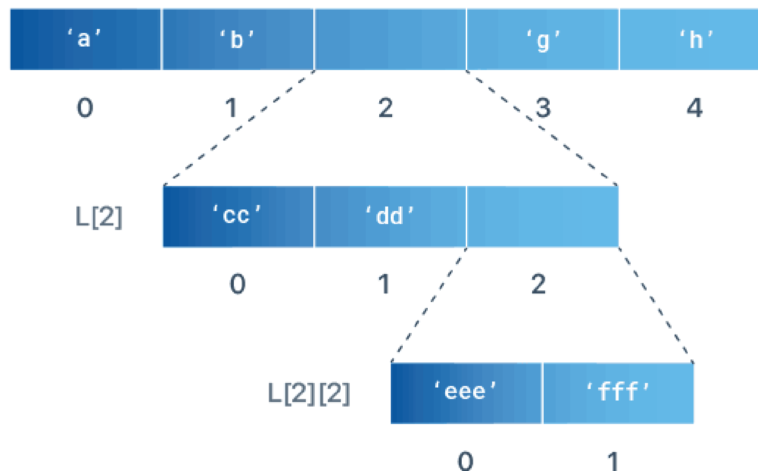
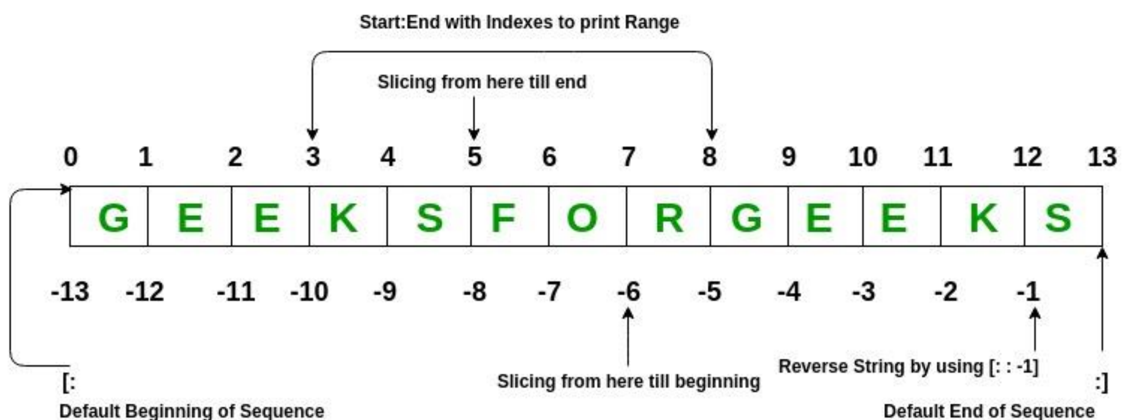


Figura 16. Exemplu de indexare listă imbricată

```
# lista 1D
lista_1D = [1, 2, 10]
print(f'Ultimul element din lista este: {lista_1D[-1]}')
# lista 2D
lista_2D = [1, 2, 3, [11, 12, 13]]
print(f'Elementul al doilea din utlima lista este: {lista_2D[-1][1]}')
# lista 3D
lista_3D = [[[1, 2, 3]], [[11, 22, 33]], [[111, 222, 333]]]
print(f'Primul element din a doua lista este: {lista_3D[1][0][0]}')
OUTPUT:
Ultimul element din lista este: 10
Elementul al doilea din utlima lista este: 12
Primul element din a doua lista este: 11
```



Figură 17. Indexarea pozitivă/negativă a unei liste - preluare https://www.geeksforgeeks.org/

ITERAREA LISTELOR

Fiind un tip secvențial de date, listele Python pot fi iterate identic cu obiectele tip `string`, fiind permisă parcurgerea listei element cu element sau în funcție de indecși prin utilizarea instrucțiunilor `for` și `while` – cu mențiunea că este de preferat utilizarea buclei `for`. În primul exemplu din fragmentul următor de cod este prezentată prima posibilitate de parcurgere iterativă a listei – bucla `for`. În cel de-al doilea exemplu este utilizată bucla `while` (trebuie avut grijă la faptul că este posibil ca bucla să itereze la infinit dacă nu se specifică în mod corect condiția de oprire: incrementarea indexului în acest caz).

```
print('Utilizare bucla for:')
list_abrevieri = ['A.C', 'C.C', 'I.A.', 'kW', 'I.E.']
for abreviere in list_abrevieri:
    print(abreviere, end='/')
print('\nUtilizare bucla while:')
index = 0
while index < len(list_abrevieri):
    print(list_abrevieri[index], end='/')
    index += 1
```

```
OUTPUT:
Utilizare bucla for:
A.C/C.C/I.A./kW/I.E./
Utilizare bucla while:
A.C/C.C/I.A./kW/I.E./
```

În exemplul următor este inițializată o listă ce conține 4 liste interioare, fiecare fiind compusă dintr-un `string` – id-ul studentului și un `int` – media de admitere. Lista inițială se iterează cu o buclă `for`, rezultând toate sublistele constituente, apoi cu ajutorul unui al doilea `for` – intern, se va bucla în listele interioare, realizând un anumit algoritm – se verifică dacă elementul din iterația curentă este de tip `str` utilizând funcția încorporată `isinstance()` - `f34`. Dacă funcția returnează `True`, atunci valoarea item va fi afișată pe ecran. În caz concret, se afișează un mesaj: `itemul nu este strig. 7.75 <class 'float'>`.

```
# se intitializeaza o lista formata din 4 liste interioare
lista_studenti = [
    ['001', 9.87],
    ['002', 8.98],
    ['003', 7.75],
    ['004', 9.14]
]
# se itereaza prin listele interne cu ajutorul a doua bucle for
for student in lista_studenti:
    print(student)
    for item in student:
        if isinstance(item, str):
            print(item)
        else:
            print('itemul nu este strig.', item, type(item))
```

OUTPUT:
['001', 9.87]
001
itemul nu este strig. 9.87 <class 'float'>
['002', 8.98]
002
itemul nu este strig. 8.98 <class 'float'>
['003', 7.75]
003
itemul nu este strig. 7.75 <class 'float'>
['004', 9.14]
004
itemul nu este strig. 9.14 <class 'float'>

OPERAȚII CU LISTE

Caracterul mutabil al listelor le oferă o flexibilitate mai mare în momentul în care se dorește modificarea acestora (comparabil cu obiectele imutabile). Cu alte cuvinte, listele în Python pot fi alterate local, fără a fi nevoie ca interpretorul intern Python să genereze o copie a obiectului respectiv. Pentru a modifica o listă, este suficient de a specifica indexul din listă pentru elementul care se dorește a fi modificat și apoi atribuită o altă valoare care va înlocui acest element. Mai mult, prin partiționarea listei se pot modifica mai multe elemente simultan, așa cum este detaliat în fragmentul următor de cod. Trebuie observat faptul că, chiar dacă lista a fost modificată, adresa de referință din memorie nu a fost modificată (lucrul generat prin utilizarea funcției încorporate `id()` - `f31`). Spre deosebire de modificarea prin indexare, unde se poate modifica un element singular, modificarea prin felierea listei parcurge două etape: **șterge** porțiunea de listă specificată prin feliere și **inserează** în acel loc noile valori specificate. Acest lucru este important pentru a explica de ce numărul elementelor inserate nu trebuie neapărat să fie egal cu numărul element înlocuite. Drept urmare, se poate indica un număr mai mic de elemente de a fi înlocuite decât elementele care le înlocuiesc, caz în care, pe lângă elemente, lista își va modifica inclusiv dimensiunea – numărul de elemente. Acest lucru este posibil deoarece valorile de inserat sunt preluate înainte de a le șterge pe cele indicate – lista întâi va conține toate elementele, apoi le va șterge pe cele de înlocuit.

```
# se creeaza lista originala
lista_numerica = [1, 2, 3, 4, 5, 6]
print(f'lista initiala: {lista_numerica} - id: {hex(id(lista_numerica))}')
# se inlocuieste al treilea element
lista_numerica[2] = 33
print(f'lista modificata {lista_numerica} - id: {hex(id(lista_numerica))}')
# se modifica toate elementele de la indexul 3 la sfarsit
# se adauga si un element in plus
lista_numerica[3:]=[44, 55, 66, 77]
print(f'lista modificata{lista_numerica} - id: {hex(id(lista_numerica))}')
```

OUTPUT:

```
lista initiala: [1, 2, 3, 4, 5, 6] - id: 0x181e781b5c0
lista modificata [1, 2, 33, 4, 5, 6] - id: 0x181e781b5c0
lista modificata[1, 2, 33, 44, 55, 66, 77] - id: 0x181e781b5c0
```

Pe lângă modificarea listelor, interpretorul Python permite o serie de alte operații uzuale cu liste, exemplificate în fragmentul următor de cod.

```
# crearea unei liste si atribuirea la o variabila
lista_test = ['inginer','energie',1,3]
print('Lista initiala este:', lista_test)
# aflarea numarului de elemente din lista
lungime_lista = len(lista_test)
print(f'Lista analizata are {lungime_lista} elemente.')
# contatenarea a doua liste
lista_concatenata = lista_test + ['inteligenta',3,4]
print(f'Noua lista este: {lista_concatenata} cu lungimea: {len(lista_concate-
nata)}')
# stergere prin atribuirea unei liste goale
lista_concatenata[4:] = []
print(f'Lista concatenata este {lista_concatenata} cu lungimea: {len(li-
sta_concatenata)}')
# multiplicarea unie liste
print(f'Lista multiplicata cu 2: {lista_test*2}')
# partitionarea listelor
print(f'Elementele din mijlocul listei sunt: {lista_test[1:3]}')
# verificare apartenenta obiect in lista
print(f'"e" este in {lista_test}? {"e" in lista_test}')
```

```
OUTPUT:
Lista initiala este: ['inginer','energie',1,3]
Lista analizata are 4 elemente.
Noua lista este: ['inginer','energie',1,3,'inteligenta',3,4] cu lungimea: 7
Lista concatenata este ['inginer', 'energie', 1, 3] cu lungimea: 4
Lista multiplicata cu 2: ['inginer','energie',1,3,'inginer','energie',1,3]
Elementele din mijlocul listei sunt: ['energie', 1]
"e" este in ['inginer','energie',1,3]? False
```

METODE SPECIFICE LISTELOR

Ca fiecare clasă încorporată în sintaxa de bază a limbajului de programare Python, și clasa `list` dispune de o serie de metode predefinite menită să ușureze lucrul cu aceste obiecte. În continuare toate metodele specifice listelor Python vor fi detaliate.

`met_list.1. obiect_list.append(element)` – inserează argumentul `element` la sfârșitul obiectul de tip `list`. Argumentul poate fi orice obiect Python, dar va fi adăugat ca element singular (chiar dacă argumentul este compus dintr-un obiect tip container, acesta va fi adăugat la sfârșitul listei ca un singur element). Metoda nu returnează nicio valoare – `None`.

```
# adaugarea unui element la sfarsitul unei liste
list_initiala = [1, 2, 3, 4]
print(f'lista initiala {list_initiala} are {len(list_initiala)} elemente')
list_initiala.append(5)
print(f'lista modificata {list_initiala} are {len(list_initiala)} elemente')
# adaugarea unei liste la sfarsitul listei initiale
list_initiala.append([11, 22, 33, 44, 55])
print(f'lista modificata {list_initiala} are {len(list_initiala)} elemente')
OUTPUT:
lista initiala [1, 2, 3, 4] are 4 elemente
lista modificata [1, 2, 3, 4, 5] are 5 elemente
lista modificata [1, 2, 3, 4, 5, [11, 22, 33, 44, 55]] are 6 elemente
```

`met_list.2.` `obiect_list.clear()` – șterge toate elementele din obiectul tip `list`. Metoda nu primește niciun parametru și nu returnează nicio valoare – `None`. Merită menționat faptul că metoda a fost implementată pentru variantele mai recente de Python – începând cu varianta 3.3, pentru variantele anterioare se utiliza instrucțiunea `del obiect_list`.

```
# stergerea tuturor elementelor dintr-o lista
list_initiala = ['inginer', 'energetica', 'stiinda datelor']
print(f'lista initiala {list_initiala} are {len(list_initiala)} elemente')
list_initiala.clear()
print(f'lista initiala {list_initiala} are {len(list_initiala)} elemente')
OUTPUT:
lista initiala ['inginer', 'energetica', 'stiinda datelor'] are 3 elemente
lista initiala [] are 0 elemente
```

`met_list.3.` `obiect_list.copy()` – returnează o **copie superficială** (shallow copy) a obiectului tip `list` indicat. Nu acceptă niciun parametru. Este extrem de utilă pentru crearea duplicatelor listelor inițiale și modificarea acestor duplicate fără a altera listele inițiale. După cum se poate observa în primul exemplu din fragmentul următor de cod, utilizarea metodei `copy()` permite păstrarea listei inițiale nemodificate, lucru ce vine în contrast cu cea de-a doua modalitate de copiere a unui obiect Python, și anume atribuirea directă utilizând semnul `=` , când, odată modificată copia listei, se modifică și lista inițială. Această modalitate se numește **copiere adâncă** (deep copy). Acest lucru se întâmplă deoarece, utilizând atribuirea, ambele variabile vor indica fix același obiect, în timp ce utilizarea metodei `copy()` creează un obiect specific nou, pe o altă zonă de memorie care are aceleași elemente ca obiectul original. Ambele variante sunt detaliate în fragmentul următor de cod.

```
# utilizand metoda .copy()
patrate_impare = [numar**2 for numar in [1,2,25,7,23,4,44] if numar%2 != 0]
print(patrate_impare, 'id:', hex(id(patrate_impare)))
patrate_impare_copie = patrate_impare.copy()
patrate_impare_copie.append(10)
print(patrate_impare_copie,'id:', hex(id(patrate_impare_copie)))
print(patrate_impare, 'id:', hex(id(patrate_impare)))
OUTPUT:
[1, 625, 49, 529] id: 0x1ab4b932e40
[1, 625, 49, 529, 10] id: 0x1ab4b92bec0
[1, 625, 49, 529] id: 0x1ab4b932e40
```

```
# utilizand atribuirea
patrate_impare = [numar**2 for numar in [1,2,25,7,23,4,44] if numar%2 != 0]
print(patrate_impare, 'id:', hex(id(patrate_impare)))
patrate_impare_copie = patrate_impare
patrate_impare_copie.append(10)
print(patrate_impare_copie,'id:', hex(id(patrate_impare_copie)))
print(patrate_impare, 'id:', hex(id(patrate_impare)))
OUTPUT:
[1, 625, 49, 529] id: 0x1ab4b94be00
[1, 625, 49, 529, 10] id: 0x1ab4b94be00
[1, 625, 49, 529, 10] id: 0x1ab4b94be00
```

O a treia variantă de copiere superficială a unei liste este de a utiliza indexarea prin feliere, de a se alege toate elementele din lista originală și de atribuire a acestor elemente unei alte variabile. Acest exemplu este prezentat în fragmentul de cod următor.

```
#utilizarea felierii
lista_obiecte = ['inginerie', 0, 6.7]
lista_obiecte_noua = lista_obiecte[:]

lista_obiecte_noua.append('energetica')

print('Lista originala:', lista_obiecte, hex(id(lista_obiecte)))
print('Lista copiata:', lista_obiecte_noua, hex(id(lista_obiecte_noua)))
OUTPUT:
Lista originala: ['inginerie', 0, 6.7] 0x1ab4bf54cc0
Lista copiata: ['inginerie', 0, 6.7, 'energetica'] 0x1ab4bf2b940
```

`met_list.4.` `obiect_list.count(item)` – returnează de câte ori parametrul `item` apare în obiectul tip `list` analizat. `item` poate fi orice obiect specific Python. În fragmentul de cod următor sunt prezentate două abordări pentru a realiza acest lucru: metoda `count()` și iterarea prin listă cu o buclă `for` pentru a compara și observa cum funcționează intern metoda. Se analizează un obiect tip string, care va fi convertit la o listă utilizând funcția constructor `list()`. În obiectul tip list rezultat se dorește a se afla de câte ori apare vocala `"i"`, pentru aceasta utilizându-se ambele abordări.

```
text = 'inginerie energetica - inteligenta artificiala'
lista_text = list(text)

# utilizarea metodei .count()
print(f'Metoda 1: vocala "i" apare in text de {lista_text.count("i")} ori')

# utilizarea buclei iterative for
# se initializeaza un contor cu valoarea 0
contor = 0
# se itereaza prin lista
for element in lista_text:
    # daca elementul de la iteratia curenta este i, contorul se incrementeaza
    if element == 'i':
        contor += 1

print(f'Metoda 1: vocala "i" apare in text de {contor} ori')
OUTPUT:
Metoda 1: vocala "i" apare in text de 9 ori
Metoda 2: vocala "i" apare in text de 9 ori
```

OBSERVAȚIE. Dacă se introduce mai mult de 1 element drept parametru, metoda va genera o eroare de tip `TypeError: list.count() takes exactly one argument (2 given)`. Totuși se pot căuta obiecte (containere) formate din mai multe elemente, parametrul va consta dintr-o listă, tuplu, dicționar etc. `lista_text.count(("i","2"))` – se încearcă determinarea numărului de apariții ale unui tuplu cu două elemente de fapt, și nu două elemente distincte!

`met_list.5.` `obiect_list.extend(iterabil)` – extinde obiectul de tip `list` prin adăugarea tuturor elementelor din parametrul `iterabil`. Are funcționalitatea similară cu metoda `append()` - `met_list.1`, diferența constând în numărul de elemente care sunt adăugate. Dacă metoda `append()` adaugă **un singur** element, `extend()` extinde lista adăugând oricâte elemente sunt în parametrul `iterabil`, lungimea ei crescând cu numărul de elemente din parametru.

```
lista_obiecte = ['inginerie', 0, 6.7]
obiecte_adaugat = ['energetica', 13, 'algoritmi']
print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')

# folosind metoda append()
lista_obiecte.append(obiecte_adaugat)
print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')
OUTPUT:
['inginerie', 0, 6.7] are 3 elemente
['inginerie', 0, 6.7, ['energetica', 13, 'algoritmi']] are 4 elemente
```

```
lista_obiecte = ['inginerie', 0, 6.7]
obiecte_adaugat = ['energetica', 13, 'algoritmi']
print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')
# folosind metoda extend()
lista_obiecte.extend(obiecte_adaugat)
print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')
OUTPUT:
['inginerie', 0, 6.7] are 3 elemente
['inginerie', 0, 6.7, 'energetica', 13, 'algoritmi'] are 6 elemente
```

OBSERVAȚIE 1. Deoarece ambele metode returnează `None`, rezultatele nu se pot atribui altor variabile.

OBSERVAȚIE 2. Ambele metode `append()` și `extend()` alterează pe loc obiectul pe care sunt apelate și nu returnează nicio valoare. Drept urmare, apelarea consecutivă a celor două metode pe lista din exemplul următor va genera următorul output:

```
['inginerie', 0, 6.7] are 3 elemente
['inginerie', 0, 6.7, ['energetica', 13, 'algoritmi inteligenti']] are 4
elemente

['inginerie', 0, 6.7, ['energetica', 13, 'algoritmi inteligenti'], 'energetica',
13, 'algoritmi inteligenti'] are 7 elemente
```

EXEPLICAȚIA: Lista inițială are 3 elemente, la care se adaugă o listă cu elemente prin utilizarea metodei `append()`, rezultând 4 elemente în total. Utilizând apoi metoda `extend()`, listei alterate deja i se adaugă și cele 3 elemente din lista `obiecte_adaugat`, rezultând 7 elemente în total după utilizarea celor două metode.

Intern, extend() funcționează precum este prezentat în fragmentul de cod următor: se iterează utilizând o buclă for prin obiectul iterativ și fiecare element se adaugă la lista inițială. Acest procedeu, deși eficient, duce la o rulare mai lentă a metodei comparativ cu append().

```
lista_obiecte = ['inginerie', 0, 6.7]
obiecte_adaugat = ['energetica', 13, 'algoritmi']

print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')

for item_adaugat in obiecte_adaugat:
    lista_obiecte.append(item_adaugat)

print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')
OUTPUT:
['inginerie', 0, 6.7] are 3 elemente
['inginerie', 0, 6.7, 'energetica', 13, 'algoritmi'] are 6 elemente
```

Există și o a treia posibilitate de a extinde o listă pe lângă cele două metode menționate: concatenarea efectivă a celor două:

```
lista_obiecte = ['inginerie', 0, 6.7]
obiecte_adaugat = ['energetica', 13, 'algoritmi']
print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')
lista_obiecte += obiecte_adaugat
print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')
OUTPUT:
['inginerie', 0, 6.7] are 3 elemente
['inginerie', 0, 6.7, 'energetica', 13, 'algoritmi'] are 6 elemente
```

met_list.6. obiect_list.index(element, star, stop) – returnează poziția (indexul) **primei** apariții a elementului specificat drept argument. Parametrul element (obligatoriu) poate fi orice obiect Python în timp ce parametrii start și stop sunt opționali și permit specificarea spațiului de căutare. Ca valori implicite: start este 0 și stop este lungimea obiect_list - 1. Dacă elementul nu este găsit în listă, metoda va genera o eroare de tip **ValueError**: 13 is not in list.

```
lista_obiecte = ['inginerie', 0, 6.7, 3.4, 0, 3.0, 0]
print(f'Prima valoare de 0 este pe indexul: {lista_obiecte.index(0)}')
print(f'Pozitia este: {lista_obiecte.index(0) + 1}')
index_cautat = lista_obiecte.index(13, 2, 5)
print(index_cautat)
OUTPUT:
Prima valoare de 0 este pe indexul: 1
Pozitia este: 2
...
ValueError: 13 is not in list
```

met_list.7. obiect_list.insert(pozitie, element) – inserează parametrul element pe poziția indicată. Ambii parametri sunt necesari și metoda nu returnează nicio valoare. Este o versiune a metodei append() care, suplimentar, permite specificarea poziției, sub formă de index, pe care se dorește a insera noul element. Trebuie specificat faptul că indicarea poziției = 0, este echivalentă cu inserarea unui element nou pe prima poziție în listă (indexare din 0).

```
lista_obiecte = ['inginerie', 0, 'energie termica', 3.14, 0]
print('Pana in inserarea obiectului')
print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')
# se adauga un element pe pozitia 1 - index 0
lista_obiecte.insert(0, 'electro')
print('Dupa inserarea obiectului string')
print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')
# se adauga un obiect tip lista pe pozitia 4 - index 3
lista_obiecte.insert(3, [5, 6, 7])
print('Dupa inserarea obiectului lista')
print(f'{lista_obiecte} are {len(lista_obiecte)} elemente')
```

```
OUTPUT:
Pana in inserarea obiectului
['inginerie', 0, 'energie termica', 3.14, 0] are 5 elemente
Dupa inserarea obiectului string
['electro', 'inginerie', 0, 'energie termica', 3.14, 0] are 6 elemente
Dupa inserarea obiectului lista
['electro','inginerie',0,[5, 6, 7], 'energie termica', 3.14, 0] are 7 elemente
```

Merită observat faptul că în mod similar cu metoda `append()`, metoda `insert()` adaugă doar un element la lista inițială.

`met_list.8. obiect_list.pop(pozitie)` – returnează elementul de pe poziția specificată prin argumentul `pozitie` – are sensul de index. Argumentul este opțional, implicit acesta având valoarea `-1` (returnează ultimul element din listă). **Mai mult, metoda alterează și lista, ștergând elementul de pe poziția specificată.** Dacă se specifică un index mai mare decât lungimea listei curent, metoda va genera eroare de tip `IndexError: pop index out of range`.

```
specializari_energetica = [
'Termo',
'Electro',
'Hidro',
'Nucleara'
]
print('Specializarile sunt:', specializari_energetica)
# se sterge al doilea element - index 1
element_sters = specializari_energetica.pop(1)
print('Dupa stergerea celei de-a treia specializari:')
print(specializari_energetica)
print('Elementul sters:', element_sters)
# se sterge ultimil element - fara specificare argument
utlimul_element = specializari_energetica.pop()
print('Dupa stergerea ultimelei specializari:')
print(specializari_energetica)
print('Elementul sters:', utlimul_element)
```

```
OUTPUT:
Specializarile sunt: ['Termo', 'Electro', 'Hidro', 'Nucleara']
Dupa stergerea celei de-a treia specializari:
['Termo', 'Hidro', 'Nucleara']
Elementul sters: Electro
Dupa stergerea ultimelei specializari:
['Termo', 'Hidro']
Elementul sters: Nucleara
```

Trebuie menționat faptul că indexarea negativă este perfect posibilă. În exemplul următor este șters același element ca în primul exemplu, doar că indexul este specificat utilizând un număr negativ: se pleacă de la -1 (ultimul element) și se decrementează până la primul.

```
specializari_energetica = ['Termo', 'Electro', 'Hidro', 'Nucleara']
print('Specializarile sunt: ', specializari_energetica)
# se sterge al doilea element - index -3
element_sters = specializari_energetica.pop(-3)
print('Dupa stergerea celei de-a treia specializari:')
print(specializari_energetica)
print('Elementul sters:', element_sters)
OUTPUT:
Specializarile sunt: ['Termo', 'Electro', 'Hidro', 'Nucleara']
Dupa stergerea celei de-a treia specializari:
['Termo', 'Hidro', 'Nucleara']
Elementul sters: Electro
```

`met_list.9.` `obiect_list.remove(element)` – șterge prima apariție în listă a parametrului obligatoriu `element` (poate fi orice obiect Python existent în obiectul tip `list` analizat). Dacă există mai multe elemente identice, celelalte nu vor fi șterse.

În fragmentul următor de cod este inițializată o listă conținând prescurtările domeniilor ce se pot studia în Facultatea de Energetică. Utilizatorul trebuie să aleagă o specializare existentă apelând funcția încorporată `input()` - `f32` împreună cu metoda specifică stringurilor `capitalize()` - `met_str.1`. Opțiunea aleasă de utilizator este mai apoi utilizată drept argument al metodei `remove()` pentru a șterge această opțiune din totalul posibilităților. În acest exemplu utilizatorul a introdus stringul `'termo'`, care a fost capitalizat, rezultând `'Termo'`, obiect ce se regăsește în lista inițială, fiind în prealabil șters. Dacă elementul nu există în listă, metoda va genera o eroare de tip `ValueError: list.remove(x): x not in list`.

```
specializari_energetica = ['Termo', 'Electro', 'Hidro', 'Nucleara']
print('Specializari dispobilile:', specializari_energetica)
# stergerea elementului 'Termo' utilazand remove
specializare = input('Ce sepecializare alegeti?').capitalize()
specializari_energetica.remove(specializare)
print('Specializari dispobilile acum:', specializari_energetica)
OUTPUT:
Specializari dispobilile: ['Termo', 'Electro', 'Hidro', 'Nucleara']
Specializari dispobilile acum: ['Electro', 'Hidro', 'Nucleara']
```

`met_list.10.` `obiect_list.reverse()` – inversează ordinea de sortare a elementelor. Metoda nu acceptă niciun parametru și nu returnează nicio valoare, ea acționând local asupra listei originale, alterând-o.

```
lista_obiecte = ['inginerie', 0, 'energie termica', 3.14, 0]
print('Lista originala:', lista_obiecte)
# se inverseaza ordinea obiectelor
lista_obiecte.reverse()
print('Lista originala inversata:', lista_obiecte)
OUTPUT:
Lista originala: ['inginerie', 0, 'energie termica', 3.14, 0]
Lista originala inversata: [0, 3.14, 'energie termica', 0, 'inginerie']
```

Există două alternative viabile la utilizarea acestei metode: inversarea listei folosind indexarea prin feliere (se selectează toate elementele și se parcurge invers lista) și inversarea listei apelând funcția încorporată `reversed()` - `f52` (generează un obiect iterabil, apărând necesitatea de a se utiliza o instrucțiune iterativă pentru a genera componentele). În fragmentul următor sunt exemplificate ambele metode alternative, specificând faptul că utilizarea metodei specifice listelor este cea mai indicată și simplu de utilizat, celelalte două având o utilitate mai generală. Totuși, operarea celor două metode alternative prezintă un avantaj: spre deosebire de metoda specifică `reverse()`, acestea permit stocarea obiectului inversat într-o altă variabilă, lista originală rămânând nealterată (nicio metodă nu acționează asupra ei). În funcție de cerințele problemei, se utilizează una dintre cele trei variante exemplificate în continuare.

```
lista_obiecte = ['inginerie', 0, 'energie termica', 3.14, 0]
print('Lista originala:', lista_obiecte)

# inversare prin metoda specifica reverse()
lista_obiecte.reverse()
print('Lista originala inversata:', lista_obiecte)

# inversare prin indexare si parcurgere inversa
# se va inversa lista inversate cu metoda .reverse()
lista_inversata = lista_obiecte[::-1]
print('Lista inversata - indexare:', lista_inversata)

# inversare cu functia incorporata reversed()
lista_inversata_2 = []

for item in reversed(lista_obiecte):
    lista_inversata_2.append(item)
print('Lista inversata - reversed():', lista_inversata_2)
OUTPUT:
Lista originala: ['inginerie', 0, 'energie termica', 3.14, 0]
Lista originala inversata: [0, 3.14, 'energie termica', 0, 'inginerie']
Lista inversata - indexare: ['inginerie', 0, 'energie termica', 3.14, 0]
Lista inversata - reversed(): ['inginerie', 0, 'energie termica', 3.14, 0]
```

`met_list.11. obiect_list.sort(reverse=True|False, key=functie)` – sortează elementele dintr-un obiect tip `list` în ordine crescătoare (dacă parametrul `reverse` este `False` – implicit) sau crescătoare (dacă parametrul `reverse` este specificat cu valoarea `False`). Metoda modifică lista direct și nu returnează nicio valoare (returnează `None`). Parametrul opțional `key` este o funcție ce deservește drept criteriu în sortarea elementelor, asigurând o logică specifică.

```
lista_obiecte = ['energie termica', 'energie electrica', 'energie nucleara']
print('Lista originala:\n', lista_obiecte)
lista_obiecte.sort()
print('Lista sortata crescator:\n', lista_obiecte)
lista_obiecte.sort(reverse=True)
print('Lista sortata descrescator:\n', lista_obiecte)
OUTPUT:
Lista originala:
 ['energie termica', 'energie electrica', 'energie nucleara']
Lista sortata crescator:
 ['energie electrica', 'energie nucleara', 'energie termica']
Lista sortata descrescator:
 ['energie termica', 'energie nucleara', 'energie electrica']
```

OBSERVAȚIE. Metoda se poate utiliza doar pe liste omogene (care conțin numai un tip de obiecte Python), în caz contrar va genera o eroare de tip `TypeError: '<' not supported between instances of 'int' and 'str'`.

În exemplul următor de cod se dorește sortarea elementelor dintr-o listă în funcție de lungimea elementului, dar în ordine inversă – de la cel mai lung element la cel mai scurt. Pentru aceasta se utilizează funcția încorporată `len()`, specificată drept argumentul pozițional opțional `key`. A se observa faptul că funcția nu este apelată ci doar specificată – numele funcției nu este urmat de parantezele rotunde. Funcția utilizată poate avea și o complexitate mai mare, caz în care va fi necesară definirea ei *a priori* sau utilizarea unei funcții `lambda`.

```
specializari_energetica = [
  'Energetica si Tehnologii Nucleare - ETN',
  'Managementul Energiei - ME',
  "Ingineria Sistemelor Electroenergetice - ISE",
  'Termoenergetica - TE',
  'Energetica si Tehnologii de Mediu - ETM',
  'Energetica si Tehnologii Informatice - ETI',
  'Energetica si Ingineria Fluidelor - EIF'
]
# se sorteaza funcia in ordine iversa lungimii strin-ului
specializari_energetica.sort(reverse=True, key=len)
for specializare in specializari_energetica:
  print(specializare)
OUTPUT:
Ingineria Sistemelor Electroenergetice - ISE
Energetica si Tehnologii Informatice - ETI
Energetica si Tehnologii Nucleare - ETN
Energetica si Tehnologii de Mediu - ETM
Energetica si Ingineria Fluidelor - EIF
Managementul Energiei - ME
Termoenergetica - TE
```

În exemplul următor se creează o listă cu `tupluri`, fiecare `tuplu` fiind format la rândul lui din două `string`-uri (denumire specializare și abreviere) și un `int` (numărul de studenți). Pentru sortarea specifică, se definește o funcție care va genera ultimul element din tupluri și se va indica prin intermediul parametrului `key`. În consecință, sortarea se va face în ordinea descrescătoare (`reverse=True`) a numărului de studenți înrolați la acele specializări. În al doilea exemplu, se modifică funcția și se va sorta crescător (`reverse=True` – nespecificat) în funcție de literele care compun abrevierea (al doilea element din `tupluri`). De observat că sortarea șirurilor de caractere se face în funcție de primul caracter și se sortează în ordine alfabetică.

```
specializari_energetica = [
  ('Energetica si Tehnologii Nucleare', 'ETN', 76),
  ('Managementul Energiei', 'ME', 130),
  ("Ingineria Sistemelor Electroenergetice", 'ISE', 100),
  ('Termoenergetica', 'TE', 74),
  ('Energetica si Tehnologii de Mediu', 'ETM', 68),
  ('Energetica si Tehnologii Informatice','ETI', 134),
  ('Energetica si Ingineria Fluidelor', 'EIF', 86)
]
# se defineste o functie care returneaza ultimul element
def ultim_element(iterabil):
  return iterabil[-1]

# se sorteaza de la numarul cel mai mare la cel mai mic (invers)
specializari_energetica.sort(reverse=True, key=ultim_element)
for specializare in specializari_energetica:
  print(specializare)
OUTPUT:
('Energetica si Tehnologii Informatice', 'ETI', 134)
('Managementul Energiei', 'ME', 130)
('Ingineria Sistemelor Electroenergetice', 'ISE', 100)
('Energetica si Ingineria Fluidelor', 'EIF', 86)
('Energetica si Tehnologii Nucleare', 'ETN', 76)
('Termoenergetica', 'TE', 74)
('Energetica si Tehnologii de Mediu', 'ETM', 68)
```

```
specializari_energetica = [
  ('Energetica si Tehnologii Nucleare', 'ETN', 76),
  ('Managementul Energiei', 'ME', 130),
  ("Ingineria Sistemelor Electroenergetice", 'ISE', 100),
  ('Termoenergetica', 'TE', 74),
  ('Energetica si Tehnologii de Mediu', 'ETM', 68),
  ('Energetica si Tehnologii Informatice','ETI', 134),
  ('Energetica si Ingineria Fluidelor', 'EIF', 86)
]
# se defineste o functie care returneaza ultimul element
def ultim_element(iterabil):
  return iterabil[1]
# se sorteaza de la numarul cel mai mare la cel mai mic (invers)
specializari_energetica.sort(key=ultim_element)
```

```
for specializare in specializari_energetica:
  print(specializare)
OUTPUT:
('Energetica si Ingineria Fluidelor', 'EIF', 86)
('Energetica si Tehnologii Informatice', 'ETI', 134)
('Energetica si Tehnologii de Mediu', 'ETM', 68)
('Energetica si Tehnologii Nucleare', 'ETN', 76)
('Ingineria Sistemelor Electroenergetice', 'ISE', 100)
('Managementul Energiei', 'ME', 130)
('Termoenergetica', 'TE', 74)
```

Există o alternativă la utilizarea metodei specifice `sort()` – funcția încorporată `sorted()` – `f57`, care, având un caracter mai general, returnează un obiect care mai apoi poate fi atribuit unei variabile, lista originală rămânând neschimbată.

```
lista_obiecte = [0.0012, 17.56, -3.14, 0, -1.34]
print('Lista originala:', lista_obiecte)
# se utilizeaza functia sorted in functie de modulul elementului
lista_sortata = sorted(lista_obiecte, key=abs)
print('Lista dupa sortate cu functia sorted():', lista_sortata)
print('Lista originala dupa sortare:', lista_obiecte)
# se sorteaza pe loc lista fara key=abs
lista_obiecte.sort()
print('Lista dupa sortare cu metoda sort():', lista_obiecte)
OUTPUT:
Lista originala: [0.0012, 17.56, -3.14, 0, -1.34]
Lista dupa sortate cu functia sorted(): [0, 0.0012, -1.34, -3.14, 17.56]
Lista originala dupa sortare: [0.0012, 17.56, -3.14, 0, -1.34]
Lista dupa sortare cu metoda sort(): [0, 0.0012, -1.34, -3.14, 17.56]
```

4.4. Dicționare: `dict`

Deși listele sunt printre cele mai flexibile tipuri de date containere și pot conține colecții eterogene de itemi, utilizarea lor în practică este mai scăzută urmare a faptului că indexarea este singura metodă de a analiza valorile componente. În foarte multe cazuri din practică se preferă un alt tip de indexare a valorilor componente din containere – indexare pe baza unor elemente fixe, denumite **chei**, care au un nume cu o însemnătate mai mare decât indexarea pozițională (indexare comparabilă cu găsirea unor definiții într-un dicționar). Dicționarele Python sunt, deci, colecții de date asociative care mapează anumite **valori** de **chei unice**, generând astfel perechi **unice** cheie-valoare (key-value). Alte caracteristici ale dicționarelor:

- ↔ `mapare` – singura clasă din Python care deservește mapării datelor;
- ↔ `indexabile` – pot fi accesate valorile specificând cheia respectivă;
- ↔ `itemii` – compuși din perechi cheie-valoare;
- ↔ `ordonate` – ordinea elementelor se păstrează, deși nu este atât de importantă;
- ↔ `dinamice` – li se poate modifica dinamic numărul de elemente;
- ↔ `eterogene` – pot conține obiecte de diverse tipuri;
- ↔ `mutabile` – pot fi modificate după ce au fost create;
- ↔ `unice` – nu pot conține duplicate de perechi cheie-valoare;
- ↔ `cheile` – trebuie să fie unice și compuse din elemente imutabile[1].

[1]*Din punct de vedere tehnic, nu este corect să spunem că un obiect trebuie să fie imuabil pentru a fi folosit drept cheie de dicționar. Mai precis, un obiect trebuie să fie hashabil, ceea ce înseamnă că poate fi transmis unei funcții hash. O funcție hash preia date de dimensiuni arbitrare și le mapează la o valoare relativ mai simplă de dimensiune fixă, numită valoare hash (sau pur și simplu, hash), care este utilizată pentru căutarea și compararea tabelelor.*

↔ `valorile` – pot fi orice obiect și pot avea orice configurație acceptată de sintaxa Python;

↔ `hashable` – sunt în fapt tabele de referințe la obiecte (hash).

Pentru a instanția un dicționar în Python este suficient de a specifica numele variabilei (indicatorului) care va indica spațiul din memorie unde va fi stocat dicționarul, urmat de funcția constructor specifică `dict()` - `f15` sau de două acolade – `{}`. În exemplul următor sunt create două dicționare goale utilizând ambele metode de instanțiere.

```
dictionar = dict()
print(f'{dictionar} este de tipul {type(dictionar)}')
dictionar2 = {}
print(f'{dictionar2} este de tipul {type(dictionar2)}')
OUTPUT:
{} este de tipul <class 'dict'>
{} este de tipul <class 'dict'>
```

În exemplul următor sunt create două instanțe a două dicționare care mapează id-ul unic al unui student din Facultatea de Energetică de un dicționar intern reprezentând perechi – disciplină (abreviere) și notă obținută, rezultând un dicționar imbricat. De reținut faptul că o listă nu permite această funcționalitate. Există o diferență sesizabilă între cele două metode de instanțiere a dicționarelor:

- prin utilizarea acoladelor - `{}` - trebuie trecută fiecare pereche sub forma:

```
dictionar = {
    <key1>: <value2>,
    <key2>: <value2>,
      .
      .
      .
    <key>: <value>
}
```

```
note_studenti = {
    1: {'TCM': 8, 'EIT': 7, 'MMNE': 10, 'CI': 9, 'EEC': 8},
    2: {'TCM': 5, 'EIT': 8, 'MMNE': 9, 'CI': 10, 'EEC': 10},
    3: {'TCM': 7, 'EIT':8, 'MMNE': 7, 'CI': 7, 'EEC': 10},
    4: {'TCM': 7, 'EIT':8, 'MMNE': 7, 'CI': 7, 'EEC': 10},
    5: {'TCM': 4, 'EIT':5, 'MMNE': 10, 'CI': 6, 'EEC': 6}
}
print(f'Dictionarul are {len(note_studenti)} perechi key-values')
OUTPUT:
Dictionarul are 5 perechi key-values
```

- prin utilizarea funcției constructor implicită `dict()` - `f15`– trebuie trecut argumentul sub forma unei secvențe de perechi cheie-valoare (o listă sau un tuplu care conține perechile cheie-valoare). Doar în cazul în care valorile sunt obiecte de tip string se poate folosi și cea de-a treia abordare, în care se utilizează atribuirea clasică (semnul =) între chei și valori.

```
dictionar = dict((
    (<key1>, <value2>),
    (<key2>, <value2>),
      .
      .
      .
    (<key>, <value>)
))
```

sau

```
dictionar = dict(
    <key1> = <str1>,
    <key2> = <str2>,
      .
      .
      .
    <key>, <str>
)
```

```
note_studenti_2 = dict(
    (
        (1, {'TCM': 8, 'EIT': 7, 'MMNE': 10, 'CI': 9, 'EEC': 8}),
        (2, {'TCM': 5, 'EIT': 8, 'MMNE': 9, 'CI': 10, 'EEC': 10}),
        (3, {'TCM': 7, 'EIT':8, 'MMNE': 7, 'CI': 7, 'EEC': 10}),
        (4, {'TCM': 7, 'EIT':8, 'MMNE': 7, 'CI': 7, 'EEC': 10}),
        (5, {'TCM': 4, 'EIT':5, 'MMNE': 10, 'CI': 6, 'EEC': 6})
    )
)
print(f'Dictionarul are {len(note_studenti)} perechi key-values')
print(f'Sunt cele doua dict egale? {note_studenti == note_studenti_2}')
print(f'Sunt cele doua dict identice? {note_studenti is note_studenti_2}')
OUTPUT:
Dictionarul are 5 perechi key-values
Sunt cele doua dict egale? True
Sunt cele doua dict identice? False
```

DICT COMPREHENSION – COMPREHENSIUNEA DICȚIONARELOR

Ca și în cazul listelor, există o modalitate specifică limbajului de programare Python de a crea aceste mapări de date, caracterizată prin simplitate și lizibilitate – **comprehensiunea dicționarelor** (*dictionary comprehension*). Dacă în cazul creării listelor utilizând mecanismul de comprehensiune este folosit un obiect iterabil, în cazul dicționarelor, comprehensiunea necesită un obiect tip `dic`, `zip` sau `map` pentru a genera alt dicționar. În timpul execuției acestui procedeu, itemii din dicționarul original pot fi incluși condiționat în noul dicționar și fiecare item în parte poate fi alterat după caz. Sintaxa generală a mecanismului de comprehensiune al dicționarelor (de menționat faptul că metoda specifică `items()`) returnează toate perechile chei-valori:

```
dict_nou = {key: value for (key, value) in dict_vechi.items()}
```

Se specifică faptul că indicatorii `key: value` pot avea orice denumire atâta timp cât se păstrează consecvența și în corpul buclei `for` se regăsesc aceleași denumiri.

În exemplul din fragmentul următor de cod este instanțiat un dicționar compus din 5 itemi (perechi chei-valori). Utilizându-l, se propune mecanismul de comprehensiune al dicționarelor pentru a crea un nou dicționar care are drept valori pătratele valorilor din dicționarul inițial. Spre exemplificare, este prezentată și modalitatea clasică de formare a unui astfel de dicționar plecând de la o listă de valori și un dicționar gol. Merită observat faptul că primul exemplu este mult mai concis și mai rapid de scris decât utilizarea clasică. Mai mult, există necesitatea utilizării unei funcții adiționale: `zip()` - `f62`.

```
dict_initial = {'a': 11, 'b': 22, 'c': 33, 'd': 44, 'e': 55}
print(dict_initial)
# Double each value in the dictionary
dict_compus = {key: value**2 for (key, value) in dict_initial.items()}
print(dict_compus)
OUTPUT:
{'a': 11, 'b': 22, 'c': 33, 'd': 44, 'e': 55}
{'a': 121, 'b': 484, 'c': 1089, 'd': 1936, 'e': 3025}
```

```
# se instantiaza doua liste cu cate 5 valori
lista_valori = [11, 22, 33, 44, 55]
lista_chei = ['a', 'b', 'c', 'd', 'e']
# se instantiaza un dictionar gol
dictionar_patrate = dict()

for (key, value) in zip(lista_chei, lista_valori):
    dictionar_patrate[key] = value**2
print(dictionar_patrate)
OUTPUT:
{'a': 121, 'b': 484, 'c': 1089, 'd': 1936, 'e': 3025}
```

La fel ca și în cazul comprehensiunii listelor, și dicționarele pot utiliza acest mecanism în coroborare cu declarații condiționale, având sintaxa asemănătoare.

```
dict_nou = {key: value for (key, value) in dict_nou.items() if conditie_key}
```

În fragmentul de cod următor este generat un dicționar dintr-un dicționar deja existent, dar în care se omite valoarea de la cheia `'b'`, adăugând în comprehensiune declarația condiționată `if value != 22` – se omite itemul a cărui valoare este `22`. În plus, se pot utiliza mai multe declarații condiționale `if`, adițional putând fi legate între ele prin `and` sau `or`.

```
dict_initial = {'a': 11, 'b': 22, 'c': 33, 'd': 44, 'e': 55}
# Double each value in the dictionary
dict_compus = {key:value**2 for (key, value) in dict_initial.items() if value
!= 22}
print(dict_compus)
OUTPUT:
{'a': 121, 'c': 1089, 'd': 1936, 'e': 3025}
```

```
dict_initial = {'a': 11, 'b': 22, 'c': 33, 'd': 44, 'e': 55}
# Double each value in the dictionary
dict_compus = {key:value**2 for (key, value) in dict_initial.items() if value
!= 22 and value!=44}
print(dict_compus)
OUTPUT:
{'a': 121, 'c': 1089, 'e': 3025}
```

În cazul în care este necesară o declarație condiționată mai complexă, formată din `if` și `else`, sintaxa mecanismului de comprehensiune se modifică și declarația se introduce în prima parte, legându-se direct de indicativul cheilor, așa cum este prezentat în continuare.

```
dict_nou = {k:(val1 if expresie_true else val2) for (k,v) in dict_nou.items()}
```

Pentru a exemplifica această sintaxă mai complexă, este propusă următoarea premisă: o centrală de achiziții de date a radiației solare stochează valorile din perioadele nocturne cu o eroare, în loc de valoarea normală 0, în baza de date se înregistrează valoarea -1, lucru ce va induce erori în cazul în care se dorește utilizarea acestor date în diverse studii. Pentru aceasta, se poate utiliza comprehensiunea dicționarelor pentru a modifica doar valorile care sunt negative. Inițial s-au creat două liste: prima (`ore`), care utilizează mecanismul de comprehensiune al listelor și generează o listă populată din obiecte string ce reprezintă orele dintr-o zi, și a doua (`valor_radiatie`), care conține valorile orare înregistrate de piranometrul cu defect. Aceste liste sunt utilizate drept constituente pentru un dicționar denumit `radiatia_solara` care va conține perechi de valori de tipul `ora` – `valoare_radiație`. În a doua fază, după afișarea datelor și observarea faptului că valorile de pe timpul nopții sunt negative, se utilizează mecanismul de comprehensiune al dicționarelor și se ajustează această eroare filtrând valorile care vor popula noul dicționar (`radiatia_solara_`) în baza declarației condiționale `if` – `else` (dacă valoarea din dicționarul inițial este negativă, dicționarul nou va fi populat cu valoarea 0 și va păstra valoarea neschimbată în caz contrar).

```
ore = [str(num+1)+':00' for num in range(24)]
valori_radiatie = [-1, -1, 0, 0, -1, 12.4, 95.5, 345, 500, 734, 790, 950,
1078, 1130, 1015, 1001, 767, 690, 260, 37, 0, -1, 0, -1]
radiatia_solara = dict(zip(ore, valori_radiatie))

radiatia_solara_ = {k:(0 if v<0 else v) for (k,v) in radiatia_solara.items()}
for ora, valoare in radiatia_solara_.items():
    print(ora, ':', valoare)
OUTPUT:
1:00 : 0
2:00 : 0
3:00 : 0
4:00 : 0
5:00 : 0
6:00 : 12.4
7:00 : 95.5
8:00 : 345
9:00 : 500
10:00 : 734
11:00 : 790
```

```
12:00 : 950
13:00 : 1078
14:00 : 1130
15:00 : 1015
16:00 : 1001
17:00 : 767
18:00 : 690
19:00 : 260
20:00 : 37
21:00 : 0
22:00 : 0
23:00 : 0
24:00 : 0
```

Merită menționat faptul că nu este necesară utilizarea a două dicționare, ci fragmentul de cod poate fi prescurtat dacă se utilizează instrucțiunea: `radiatia_solara_ = {k:(0 if v<0 else v) for (k,v) in dict(zip(ore, valori_radiatie)).items()}`. În acest caz, dicționarul inițial se generează direct în sintaxa de comprehensiune a dicționarului, prin utilizarea funcției constructor `dict()` cu argument rezultatul funcției `zip()`. Deși pentru începători poate părea complicat, este extrem de uzual de a folosi funcții imbricate care vor fi executate de la cea mai interioară spre cea mai exterioară, fiecare folosind ca argument rezultatul returnat de funcția internă.

OBSERVAȚIE. Avertismente privind utilizarea mecanismului de comprehensiune a dicționarelor Python. Chiar dacă acest mecanism Python este excelent pentru a scrie cod elegant, ușor de citit, nu este mereu cea mai indicată alegere. Trebuie avut în vedere că:
- uneori poate determina o lentoare a rulării codului și un consum mai mare de memorie;
- contrar așteptărilor, uneori poate reduce lizibilitatea codului;
- sunt de preferat alte abordări (buclele iterative) când codul implică logică mai complexă.

INDEXAREA DICȚIONARELOR

Deoarece dicționarele reprezintă maparea dintre obiectele `chei` și obiectele `valori`, indexarea dicționarelor se realizează diferit față de liste (care se indexează specificând poziția/indexul din listă al elementului dorit). În cazul dicționarelor, indexarea se realizează prin specificarea `cheii` pentru care se dorește obținerea `valorii` – exact ca în cazul obținerii definiției (sau sinonimului, antonimului etc.) dintr-un dicționar real. Sintaxa generală este de forma: `dictionar[cheie]`, unde cheia care se dorește a fi interogată se pune între paranteze drepte `[]`. În exemplul din fragmentul următor de cod este prezentat modul în care se obține valoarea radiației solare de la ora 12:00 și stocarea acestei valori într-o variabilă. Trebuie avut grijă la denumirea cheii interogate. Dacă aceasta nu se regăsește în itemii dicționarului, interpretorul Python va genera o eroare de tip `KeyError: '1'`.

```
# cheie scrisa corect
radiatie_12 = radiatia_solara_ok['12:00']
print(f'Valoarea radiatie solare de la ora 12 este {radiatie_12} W/mp')

# cheie scrisa gresit
radiatie_1 = radiatia_solara_ok['1']
print(f'Valoarea radiatie solare de la ora 12 este {radiatie_12} W/mp')
OUTPUT:
Valoarea radiatie solare de la ora 12 este 950 W/mp

KeyError: '1'
```

Similar cu listele, dicționarele pot fi și ele imbricate (să conțină un dicționar în interiorul altui dicționar), caz în care trebuie specificate ambele chei pentru a prelua valoarea dorită. În fragmentul următor de cod este instanțiat un dicționar ce conține 3 materiale de construcție, fiecare fiind deschis de un alt dicționar intern. Pentru a se obține conductivitatea materialului `2`, trebuie să se specifice prima cheie, urmată de cheia dicționarului intern:

`materiale[2]['conductivitate']`. Mai mult, toate valorile preluate din cheile respective pot fi atribuite altor nume de variabile și utilizate în prealabil în alte porțiuni de cod.

```
materiale = {1: {'denumire': 'vata minerala',
                 'conductivitate': 0.014,
                 'grosimi': [100, 150, 200, 250]},
             2: {'denumire': 'polistiren',
                 'conductivitate': 0.03,
                 'grosimi': [150, 200, 300]},
             3: {'denumire': 'caramida',
                 'conductivitate': 0.01,
                 'grosimi': [50, 150, 300, 550]},
            }

print(f'Conductivitatea materialului {materiale[2]["denumire"]} este
{materiale[2]["conductivitate"]}')

grosimi_material3 = materiale[2]["grosimi"]
print(f'Cate grosimi sunt disponibile pentru {materiale[3]["denumire"]}?')
print(f'Sunt disponibile {len(grosimi_material3)}: {grosimi_material3}')
OUTPUT:
Conductivitatea materialului polistiren este 0.03
Cate grosimi sunt disponibile pentru caramida?
Sunt disponibile 3: [150, 200, 300]
```

ITERAREA DICȚIONARELOR

Chiar dacă nu este un obiect secvențial de date Python (reprezintă o mapare), faptul că dicționarele sunt containere compuse din mai multe obiecte, acestea pot fi iterate cu ajutorul buclelor iterative – `for` și `while` – cu mențiunea că este de preferat utilizarea buclei `for`. Un exemplu de iterare a unui dicționar a fost prezent și în cazul afișării datelor radiației solare din secțiunea DICT COMPREHENSION – COMPREHENSIUNEA DICȚIONARELOR. Trebuie menționat însă faptul că, pentru a afișa toate elementele din dicționar, trebuie să i se specifice metoda încorporată `items()` care returnează tupluri formate din perechi chei-valori; aceasta va fi detaliată în secțiunea următoare. În fragmentul următor de cod este exemplificată o buclă iterativă `for` aplicată pe dicționarul `materiale` instanțiat anterior.

```
for material in materiale.items():
    print(material)
OUTPUT:
(1, {'denumire': 'vata minerala', 'conductivitate': 0.014, 'grosimi': [100,
150, 200, 250]})
(2, {'denumire': 'polistiren', 'conductivitate': 0.03, 'grosimi': [150, 200,
300]})
(3, {'denumire': 'caramida', 'conductivitate': 0.01, 'grosimi': [50, 150, 300,
550]})
```

În situația în care bucla for se utilizează fără a se specifica metoda dicționarelor items(), interpretorul Python va returna doar cheile dicționarului – de fapt, instrucțiunea returnează o listă care conține doar cheile din dicționar, valorile fiind omise. Acest lucru permite ca aceste chei să fie utilizate în aceeași buclă și să se afișeze și valorile. Ambele variante sunt prezentate în fragmentul următor de cod.

```
for material in materiale:
    print(material)

for items in materiale:
    print(items, '---', materiale[items])
OUTPUT:
1
2
3
1 --- {'denumire': 'vata minerala', 'conductivitate': 0.014, 'grosimi': [100, 150, 200, 250]}
2 --- {'denumire': 'polistiren', 'conductivitate': 0.03, 'grosimi': [150, 200, 300]}
3 --- {'denumire': 'caramida', 'conductivitate': 0.01, 'grosimi': [50, 150, 300, 550]}
```

În ipoteza în care atât cheile cât și valorile dintr-un dicționar sunt obiecte imutabile Python, cu ajutorul buclelor iterative acestea se pot interschimba foarte simplu. În exemplul din fragmentul de cod următor este creat un astfel de dicționar – cheile sunt obiecte string și valorile obiecte int, ambele tipuri imutabile. Se instanțiază un dicționar nou care, la fiecare iterație prin elementele dicționarului inițial, va fi populat cu valorile din primul drept chei și cheile drept valori, rezultând interschimbarea celor două elemente.

```
numere = {"one": 1, "two": 2, "three": 3, "four": 4}
print(numere)

numere_interschimbate = dict()
for key, value in numere.copy().items():
    numere_interschimbate[value] = key

print(numere_interschimbate)
    print(material)
OUTPUT:
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
{1: 'one', 2: 'two', 3: 'three', 4: 'four'}
```

Desigur, în Python există o metodă mult mai elegantă și simplă de a realiza acest lucru – utilizând mecanismul de comprehensiune al dicționarelor, care, în esență este tot un mecanism iterativ. În fragmentul următor de cod este prezentat un exemplu cu scop comparativ.

```
numere_inversate = {v:k for (k,v) in numere.items()}
print(numere_inversate)
OUTPUT:
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
{1: 'one', 2: 'two', 3: 'three', 4: 'four'}
```

OPERAȚII CU DICȚIONARE

În exemplele anterioare au fost prezentate o serie de acțiuni posibil a fi efectuate asupra dicționarelor Python, dar acestea nu sunt singulare. În principal, acest tip de obiecte se comportă ca orice container și acceptă toate operațiile specifice. În fragmentul următor de cod sunt detaliate aceste operații, începând cu definirea, popularea și modificarea lor. În primul rând, pentru a instanția un obiect tip dicționar se poate utiliza funcția constructor `dict()` sau două acolade care nu conțin nimic, `{}`, rezultatul fiind același. După creare, pentru a popula un dicționar este suficient a scrie identificatorul variabilei create, urmat de numele cheii dorite între paranteze drepte și atribuirea valorii dorite: `nume_dictionar[cheie] = valoare`. Dicționarele fiind obiecte mutabile, operația de adăugare a noii perechi de cheie și valoare se va realiza modificând dicționarul inițial, așa cum reiese și din fragmentul următor de cod prin specificarea zonei de memorie ocupate de obiect în sine: `hex(id(obiect))`. După cum se poate observa, cheile și valorile pot fi obiecte diferite în cadrul aceluiași dicționar.

```
# se instantiaza un dictionar nou utilizand functia constructor
tipuri_PV = {}
print('Dictionarul initial:')
print(tipuri_PV, '- id:', hex(id(tipuri_PV)))
# popularea dictionarului cu perechi chei-valori
tipuri_PV[1] = [10, 7,75, 0.13]
print("Dictionarul dupa adaugarea primei perechi cheie-valoare:")
print(tipuri_PV, '- id:',hex(id(tipuri_PV)))
tipuri_PV[2] = [100, 15.6, 1.26]
print("Dictionarul dupa adaugarea celei de-a doua perechi cheie-valoare:")
print(tipuri_PV, '- id:', hex(id(tipuri_PV)))
tipuri_PV['3'] = [50, '9.55', '0.55']
OUTPUT:
Dictionarul initial:
{} - id: 0x209b6d5d9c0
Dictionarul dupa adaugarea primei perechi cheie-valoare:
{1: [10, 7, 75, 0.13]} - id: 0x209b6d5d9c0
Dictionarul dupa adaugarea celei de-a doua perechi cheie-valoare:
{1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26]} - id: 0x209b6d5d9c0
```

După ce a fost creat și populat cu valori, asupra dicționarului se pot realiza o serie de operații similare celor aplicate obiectelor tip listă:
- lungimea dicționarului: numărului de perechi cheie-valoare – funcția încorporată `len()`;
- verificare apartenență – `not in` și `in`;
- modificarea valorii unei chei specificate – `nume_dictionar[cheie] = valoare_noua`;
- ștergerea perechii cheie-valoare – `del` și `nume_dictionar[cheie]`.

În plus, o serie de funcții încorporate în sintaxa Python pot fi utilizate în conjuncție cu obiectele tip dicționar și altera. Printre ele sunt: `any()` - f3, `all()` - f2, `sorted()` - f57.

```
# afisarea lungimii dictionarului
print(f'Dictionarul are {len(tipuri_PV)} perechi cheie-valoare.')
# analiza apartenentei unei valori in dictionar
print('Dictionarul contine cheia 4?', 4 in tipuri_PV)
print('Dictionarul nu contine cheia 1?', 1 not in tipuri_PV)
# indexare dictionar in funcie de cheie
print('Valorile corespunzatoare cheii 2 sunt: ', tipuri_PV[2])

# modificarea valorilor unei chei:
print('Dictionarul initial: ')
print(tipuri_PV)
# se modifica valoarea cheii "3"
tipuri_PV['3'] = "Nu mai e o lista"
print('Dupa modificare')
print(tipuri_PV)

# stergerea unei perechi cheie-valoare:
del tipuri_PV['3']
print('Dupa stergere: ')
print(tipuri_PV)
OUTPUT:
Dictionarul are 3 perechi cheie-valoare.
Dictionarul contine cheia 4? False
Dictionarul nu contine cheia 1? False
Valorile corespunzatoare cheii 2 sunt:  [100, 15.6, 1.26]
Dictionarul initial:
{1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26], '3': [50, '9.55', '0.55']}
Dupa modificare:
{1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26], '3': 'Nu mai e o lista'}
Dupa stergere:
{1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26]}
```

METODE SPECIFICE DICȚIONARELOR

Similar tuturor tipurilor de date încorporate în sintaxa Python, și clasa `dict` are definite 11 metode specifice care pot fi utilizate pentru a altera un obiect dicționar. În continuare, aceste metode vor fi detaliate, fiind exemplificate pe următorul dicționar, căruia i se specifică și id-ul – pentru a vizualiza mutabilitatea acestui tip de obiect.

```
tipuri_PV = {}
tipuri_PV[1] = [10, 7,75, 0.13]
tipuri_PV[2] = [100, 15.6, 1.26]
tipuri_PV['3'] = [50, '9.55', '0.55']
print(tipuri_PV)
print('id:', hex(id(tipuri_PV)))
OUTPUT:
{1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26], '3': [50, '9.55', '0.55']}
id: 0x209b6dc8a80
```

met_dict.1. obiect_dict.clear() – elimină toate elementele (perechile de chei-valori) din obiectul dicționar. Metoda nu primește niciun parametru și nu returnează nicio valoare – None.

```
print('Dictionarul initial:', tipuri_PV, '- id:', hex(id(tipuri_PV)))
tipuri_PV.clear()
print('Dictionarul dupa golire:', tipuri_PV, '- id:', hex(id(tipuri_PV)))
OUTPUT:
Dictionarul initial: {1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26], '3': [50,
'9.55', '0.55']} - id: 0x209b6dc8a80
Dictionarul dupa golire: {} - id: 0x209b6dc8a80
```

Este adevărat faptul că se poate goli un dicționar și atribui un dicționar gol {}, dar metoda clear() se aplică pentru referință, și nu pentru dicționar în sine. Acest lucru va goli practic toate referințele la dicționarul respectiv, și nu doar obiectul specific care este reatribuit cu un element gol, după cum este exemplificat în fragmentul de cod:

```
# se atribuie toate date din dicionarul initial
tipuri_PV2 = tipuri_PV
# golirea prin atribuirea unui dictionar gol
tipuri_PV = {}
print('dictionar initial:', tipuri_PV)
print('dictionarul secundar:', tipuri_PV2)
OUTPUT:
dictionar initial: {}
dictionarul secundar: {1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26], '3': [50,
'9.55', '0.55']}
```

```
# se atribuie toate date din dicionarul initial
tipuri_PV2 = tipuri_PV
# golirea prin metoda clear()
tipuri_PV.clear()
print('dictionar initial:', tipuri_PV)
print('dictionarul secundar:', tipuri_PV2)
OUTPUT:
dictionar initial: {}
dictionarul secundar: {}
```

met_dict.2. obiect_dict.copy() – returnează o **copie superficială** *(shallow copy*) a dicționarului specificat. Nu acceptă niciun parametru. Este extrem de utilă pentru crearea duplicatelor dicționarelor inițiale și modificarea acestor duplicate fără a altera obiectele inițiale. Este identică cu metoda copy() - met_list.3 de la analiza obiectelor tip list. Spre deosebire de operatorul atribuire (=), utilizarea metodei creează un dicționar care este completat cu o copie a referințelor din dicționarul original. Utilizarea atribuirii în schimb va crea o referință la dicționarul original, astfel încât, la modificarea primului, ambele vor fi modificate. Acest lucru este detaliat în fragmentul următor de cod.

```
print('original:', tipuri_PV)
# copierea prin atribuire
copie_tipuri_PV = tipuri_PV
# se goleste dictionarul intial
tipuri_PV.clear()
print('original', tipuri_PV)
print('copia:', copie_tipuri_PV)
OUTPUT:
original: {1:[10,7,75,0.13], 2:[100,15.6,1.26], '3':[50,'9.55', '0.55']}
original {}
copia: {}
```

```
print('original:', tipuri_PV)
# copierea prin atribuire
copie_tipuri_PV = tipuri_PV.copy()
# se goleste dictionarul intial
tipuri_PV.clear()
print('original', tipuri_PV)
print('copia:', copie_tipuri_PV)
OUTPUT:
original: {1:[10,7,75,0.13], 2:[100,15.6,1.26], '3':[50,'9.55', '0.55']}
original {}
copia: {1:[10,7,75,0.13], 2:[100,15.6,1.26], '3':[50,'9.55', '0.55']}
```

După cum se poate observa în al doilea exemplu, în copia creată nu a fost modificată.

OBSERVAȚIE. Diferența dintre copie superficială și copie adâncă. Înseamnă că orice modificări aduse unei copii a obiectului nu se reflectă în obiectul original. În Python, acest lucru este implementat folosind funcția `deepcopy()`, în timp ce în copie superficială orice modificări aduse unei copii a unui obiect se reflectă în obiectul original. În Python, acest lucru este implementat folosind funcția `copy()`.

`dict.fromkeys(chei, valori)` – returnează un dicționar populat cu mapările chei-valori specificate ca argumente. Parametrul `chei` este obligatoriu și trebuie să fie un obiect iterabil Python care va furniza cheile noului dicționar, în timp ce parametrul `valori` este opțional și are valoarea implicită `None`. În esență, metoda reprezintă a patra posibilitate de a instanția dicționare pe lângă funcția constructor `dict()`, utilizarea acoladelor `{}` și mecanismul de comprehensiune al dicționarelor.

```
lista_denumiri = ['BP Solar', 'DelSolar', 'SunWorld', 'DelSolar2']
lista_specificatii = [['mono-Si', 100, 15.6, 1.26],
                      ['poli-Si', 200, 13.15, 1.04],
                      ['poli-Si', 50, 9.55, 0.55],
                      ['mono-Si', 130, 11.8, 0.75]]
lista_atribute = ['tehnologie', 'capacitate - W', 'eficienta - %', 'suprafata - mp']
panouri_PV1 = dict.fromkeys(lista_atribute)
panouri_PV2 = dict.fromkeys(lista_denumiri, lista_specificatii[0])

print('- None la valori')
for pv in panouri_PV1.items():
    print(pv)

print('- valori specificate:')
for pv in panouri_PV2.items():
    print(pv)
```

```
OUTPUT:
- None la valori
('tehnologie', None)
('capacitate - W', None)
('eficienta - %', None)
('suprafata - mp', None)
- valori specificate:
('BP Solar', ['mono-Si', 100, 15.6, 1.26])
('DelSolar', ['mono-Si', 100, 15.6, 1.26])
('SunWorld', ['mono-Si', 100, 15.6, 1.26])
('DelSolar2', ['mono-Si', 100, 15.6, 1.26])
```

met_dict.3. `obiect_dict.get(nume_cheie, valoare)` – returnează valoarea itemului a cărui cheie este specificată prin parametrul obligatoriu `nume_cheie`. Parametrul `valoare` este opțional și indică ce valoare să returneze metoda în ipoteza în care dicționarul nu conține cheia specificată – valoarea implicită este `None`.

```
# cazul in care cheia exista in dictionar
valoare_cheie_1 = tipuri_PV.get(1)
print('valoarea de la cheia 1 este:', valoare_cheie_1)
# cazul in care cheia nu exista in dictionar
valoare_cheie_3 = tipuri_PV.get(3, 'Nu exista!')
print('valoarea de la cheia 3 este:', valoare_cheie_3)
OUTPUT:
valoarea de la cheia 1 este: [10, 7, 75, 0.13]
valoarea de la cheia 3 este: Nu exista!
```

După cum se poate observa, metoda `get()` realizează același lucru, cu obținerea valorii prin simpla specificare a cheii `dict[key]` – returnează valorile respective. Totuși există o diferență fundamentală între cele două abordări: în cazul în care metoda `get()` nu găsește cheia specificată în dicționar, va returna `None` sau obiectul din argumentul `valoare`, în timp ce încercarea de indexare a unei chei inexistente generează o eroare de tip **`KeyError`**.

```
valoare_cheie_3 = tipuri_PV[3]
print('valoarea de la cheia 3 este:', valoare_cheie_3)
OUTPUT:

KeyError: 3
```

met_dict.4. `obiect_dict.items(nume_cheie, valoare)` – returnează un obiect iterabil de tip `dict_items` ce conține toate perechile de chei-valori ce compun dicționarul sub forma unor tupluri. Metoda nu acceptă niciun parametru. A fost utilizată în exemple anterioare la iterarea printr-un dicționar sau folosirea mecanismului de comprehensiune pentru generarea acestora.

```
print(tipuri_PV.items())
for tip_PV in tipuri_PV.items():
    print(tip_PV)
OUTPUT:
dict_items([(1, [10, 7, 75, 0.13]), (2, [100, 15.6, 1.26]), ('3', [50, '9.55',
'0.55'])])

(1, [10, 7, 75, 0.13])
(2, [100, 15.6, 1.26])
('3', [50, '9.55', '0.55'])
```

Intern, metoda ține evidența asupra tuturor modificărilor în starea itemilor și, la fiecare apelare, va returna starea curentă. Spre exemplu, după ce a fost ștearsă perechea cu cheia `'3'` și alterată valoarea de la cheia `1`, metoda va returna:

```
del tipuri_PV['3']
tipuri_PV[1] = 'Nu este disponibil'
for tip_PV in tipuri_PV.items():
    print(tip_PV)
OUTPUT:
(1, 'Nu este disponibil')
(2, [100, 15.6, 1.26])
```

`met_dict.5.` `obiect_dict.keys(nume_cheie, valoare)` – returnează un obiect iterabil de tip `dict_keys` ce conține doar cheile ce compun dicționarul. Metoda nu acceptă parametri.

```
print(tipuri_PV.keys())
for key in tipuri_PV.keys():
    print(key)
OUTPUT:
dict_keys([1, 2, '3'])
1
2
3
```

Asemenea cu metoda anterioară, și metoda `keys()` ține intern evidența asupra tuturor modificărilor aduse dicționarului în sine – adăugarea sau ștergerea unor noi perechi de chei-valori și fiecare apelare a acesteia va returna evidența curentă a stării cheilor. Trebuie menționat faptul că nu pot fi modificate cheile dicționarelor (sunt imutabile), dar se pot șterge sau adăuga chei noi și valorile respective. Mai mult, obiectul `dict_keys` este un obiect iterabil fundamentat pe liste Python, deci se pot extrage, prin indexare și/sau feliere, doar anumite elemente ce îl compun, după ce a fost convertită implicit într-o listă – funcția `list()`.

```
print(f'Cheile dictionarului initial: {tipuri_PV.keys()}')
tipuri_PV['tip nou PV'] = [150, 13.4, 70]
print(f'Cheile dictionarului alterat: {tipuri_PV.keys()}')
print(f'Ultima cheie este: {list(tipuri_PV.keys())[-1]}')
OUTPUT:
Cheile dictionarului initial: dict_keys([1, 2, '3'])
Cheile dictionarului alterat: dict_keys([1, 2, '3', 'tip nou PV'])
Ultima cheie este: tip nou PV
```

met_dict.6. obiect_dict.pop(nume_cheie, valoare_default) – șterge itemul a cărui cheie este specificată prin parametrul obligatoriu nume_cheie, dar returnează valoarea lui. Parametrul valoare_default este opțional și specifică ce să returneze cheia în cazul în care nume_cheie nu există în obiectul dict_keys. Dacă acest parametru nu este specificat la apelarea metodei și cheia nu există, interpretorul Python va genera o eroare de tip KeyError. Metoda este similară celei cu același nume specifică obiectelor tip list - met_list.8.

```
print(f'Dictionar initial {tipuri_PV}')
element_sters = tipuri_PV.pop('tip nou PV')
print(f"Dictionar modificat {tipuri_PV}")
print(f'Elementeul care a fost sters: {element_sters}')
# folosire cu indicarea unei valoari implicite
element_sters = tipuri_PV.pop('tip nou PV', 'Nu exista cheia respectiva')
# va afisa mesajul trecut drept parametru
print(element_sters)
# folosire fara indicarea unei valoari implicite
element_sters = tipuri_PV.pop('tip nou PV')
# va afisa mesajul trecut drept parametru
print(element_sters)
OUTPUT:
Dictionar initial {1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26], '3': [50,
'9.55', '0.55'], 'tip nou PV': [150, 13.4, 70]}
Dictionar modificat {1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26], '3': [50,
'9.55', '0.55']}
Elementeul care a fost sters: [150, 13.4, 70]
Nu exista cheia respective

KeyError: 'tip nou PV'
```

met_dict.7. obiect_dict.popitem() – șterge ultimul item care a fost introdus în dicționar – pe principiul *ultimul introdus, primul ieșit* – și îl returnează sub forma unui obiect de tip tuplu ce conține atât cheia cât și valoarea acestuia. Metoda nu acceptă niciun parametru și generează o eroare dacă dicționarul este gol: KeyError: 'popitem(): dictionary is empty'.

```
print(f'Cheile dictionarului initial: {tipuri_PV.keys()}')
# stergerea ultimului item
item_sters1 = tipuri_PV.popitem()
print(f'Cheile dictionarului dupa stergere ultimul item: {tipuri_PV.keys()}')
print(f'Tuplul cu item sters: {item_sters1}')

item_sters2 = tipuri_PV.popitem()
print(f'Cheile dictionarului dupa stergere ultimul item: {tipuri_PV.keys()}')
print(f'Tuplul cu item sters: {item_sters2}')
item_sters3 = tipuri_PV.popitem()
print(f'Tuplul cu item sters: {item_sters3}')
OUTPUT:
Cheile dictionarului initial: dict_keys([1, 2, '3', 'tip nou PV'])
Cheile dictionarului dupa stergere ultimul item: dict_keys([1, 2, '3'])
Tuplul cu item sters: ('tip nou PV', [150, 13.4, 70])
Cheile dictionarului dupa stergere ultimul item: dict_keys([1, 2])
Tuplul cu item sters: ('3', [50, '9.55', '0.55'])
Tuplul cu item sters: (2, [100, 15.6, 1.26])
```

OBSERVAȚIE. Trebuie avut grijă în cazul în care se lucrează cu același dicționar pentru aceste exemple. Dicționarul fiind mutabil, acesta va fi alterat de fiecare metodă aplicată asupra lui și se poate ajunge în situația în care rezultatele obținute să difere față de cele prezentate în carte.

`met_dict.8.` `obiect_dict.setdefault(nume_cheie, val_default)` – returnează valoarea cheii specificată prin parametrul obligatoriu `nume_cheie`. În cazul în care cheia nu se regăsește în obiectul `dict_keys`, metoda returnează returna `None` sau `val_default` dacă acest parametru opțional este specificat. Spre deosebire de `get()`, această metodă nu generează nicio eroare la rulare. Mai mult, de fiecare dată când cheia nu este găsită, metoda alterează dicționarul și adaugă noile perechi de chei-valori, mărindu-i dimensiunea. În exemplul următor este evidențiat și acest aspect.

```
print(f'initial ({len(tipuri_PV)} elemente):\n', tipuri_PV)
# cheia se afla in obiectul dict_keys
valoare_cheie_1 = tipuri_PV.setdefault(1)
print('Valoarea cheii 1 este: ', valoare_cheie_1)
# cheia nu se afla in obiectul dict_keys si nu se specifica paramentrul optio-
nal
valoare_cheie_13 = tipuri_PV.setdefault(13)
print('Valoarea cheii 13 este: ', valoare_cheie_13)
# cheia nu se afla in obiectul dict_keys dar se specifica paramentrul optional
valoare_cheie_3 = tipuri_PV.setdefault(3, 3)
print('Valoarea cheii 3 este: ', valoare_cheie_3)
print(f'la final ({len(tipuri_PV)} elemente):\n', tipuri_PV)
```

```
OUTPUT:
initial (5 elemente):
 {1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26], '3': [50, '9.55', '0.55'], 13:
None, 3: 3}
Valoarea cheii 1 este:  [10, 7, 75, 0.13]
Valoarea cheii 13 este:  None
Valoarea cheii 3 este:  3
la final (5 elemente):
 {1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26], '3': [50, '9.55', '0.55'], 13:
None, 3: 3}
```

`met_dict.9.` `obiect_dict.update(obiect_iterabil)` – actualizează dicționarul curent cu elementele din alt dicționar sau dintr-un iterabil format din perechi de elemente – în general, un obiect tip tuplu (de fapt, este o listă formată din unul sau mai multe tupluri). Dacă metoda este apelată fără a-i fi specificat parametrul, dicționarul inițial nu este modificat. Metoda nu returnează nicio valoarea – returnează `None`.

```
print('initial:\n', tipuri_PV)
# se foloseste metoda update cu alt dictionar
dictionar_adaugat = {4: [200, 12.4]}
tipuri_PV.update(dictionar_adaugat)
print('dupa update:\n', tipuri_PV)
# se foloseste metoda update cu un tuplu de valori
tipuri_PV.update([(5, 53])
print('dupa alt update:\n', tipuri_PV)
OUTPUT:
initial:
 {1: [10, 7, 75, 0.13], 2: [100, 15.6, 1.26], '3': [50, '9.55', '0.55'], 4: 
[200, 12.4, 1.3]}
dupa update:
 {1:[10,7,75,0.13], 2:[100,15.6,1.26], '3':[50,'9.55','0.55'], 4:[200,12.4]}
dupa alt update:
 {1:[10,7,75,0.13],2:[100,15.6,1.26],'3':[50,'9.55','0.55'],4:[200,12.4],5:55}
```

Atenție la modul cum se transmite obiectul de tip tuplu: `tipuri_PV.update([(5, 53])`.

`met_dict.10.` `obiect_dict.values()` – returnează un obiect iterabil de tip `dict_values` ce conține toate valorile din dicționar. Metoda nu acceptă niciun parametru. Face parte din aceeași categorie cu metodele `items()` – `met_dict.4` și `keys()`– `met_dict.5`.

```
print(tipuri_PV.values())
for value in tipuri_PV.values():
    print(value)
OUTPUT:
dict_values([[10, 7, 75, 0.13], [100, 15.6, 1.26], [50, '9.55', '0.55']])
[10, 7, 75, 0.13]
[100, 15.6, 1.26]
[50, '9.55', '0.55']
```

Similar metodelor `items()` și `keys()`, `values()` ține intern evidența asupra tuturor modificărilor aduse dicționarului în sine – adăugarea sau ștergerea unor noi perechi de chei-valori și/sau modificarea unei valori – pot fi inclusiv obiecte mutabile. Mai mult, obiectul `dict_values` este un obiect iterabil fundamentat pe liste Python, deci se pot extrage, prin indexare și/sau feliere, doar anumite elemente ce îl compun, după ce au fost convertite implicit într-o listă - funcția `list()`.

EXEMPLU EXHAUSTIV DE UTILIZARE A DICȚIONARELOR

În fragmentul de cod următor se creează un dicționar care are scopul de a stoca date complexe despre diverse tipuri de panouri fotovoltaice: producător, tehnologia PV folosită, capacitatea unitară, eficiența și suprafața unitară. Dicționarul principal este populat cu alt dicționar interior. După instanțiere, dicționarul este populat manual cu prima pereche de date prin atribuirea valorii cheii specificate: `panouri_PV['Schott Poly'] = {}`, dicționarul intern fiind populat cu alte perechi de date. Datele următoare sunt stocate inițial în trei liste: `lista_denumiri`, `lista_atribute` și `lista_specificatii`. Funcția `adauga_elem()` va popula automat dicționarul inițial cu datele din listele respective, astfel: va instanția un dicționar gol folosit pentru a stoca temporar datele, denumit `temp_dict` și un contor denumit `index` (instanțiat cu

valoarea 0). Atâta timp cât contorul are valoarea mai mică decât numărul de elemente din variabila lista_denumiri, temp_dict va fi populat cu perechi de date din lista_atribute și lista_specificatii[index] – lista ce conține specificații trebuie indexată, deoarece la bază aceasta este o listă imbricată – utilizând funcția încorporată **zip**() - f62. Dicționarul principal - panouri_PV - este populat cu copii ale dicționarului interior, utilizând metoda încorporată **copy**(). Acest lucru este necesar deoarece indicatorul este doar un pointer către spațiu de memorie și, în caz contrar, după toate iterațiile, panouri_PV ar fi fost populat **DOAR** cu ultima listă din lista_specificatii, nefiind atribuite toate valorile. Variabila index se va incrementa cu 1 la fiecare iterație pentru a evita apariția iterațiilor infinite. Finalmente, funcția va returna dicționarului panouri_PV, fiind utilizată în conjuncție cu datele din listele definite pentru a popula automat dicționarul. Toate elementele vor fi apoi afișate pe ecranul consolei.

Cum dicționarele nu pot avea duplicate în rândul cheilor, se creează o funcție care va verifica dacă noile perechi de date sunt valide sau nu. Introducând un nume de cheie deja existent, funcția va popula dicționarul cu o cheie puțin modificată prin returnarea - adăugarea unei valori.

```
# se instantiaza un dictionar nou
panouri_PV = dict()
# popularea dictionarului - alterarea lui dupa ce a fost creat
panouri_PV['Schott Poly'] = {
    'tehnologie': 'mono-Si',
    'capacitate - W': 10,
    'eficienta - %': 7.75,
    'suprafata - mp': 0.13
}
# adaugarea elementelor dintr-o lista utilizand funcitia zip()
# se creeaza liste cu denumiri, atribute si specificatii
lista_denumiri = ['BP Solar', 'DelSolar', 'SunWorld', 'DelSolar2']
lista_atribute = ['tehnologie', 'capacitate - W', 'eficienta - %', 'suprafata
- mp']
lista_specificatii = [['mono-Si', 100, 15.6, 1.26],
                      ['poli-Si', 200, 13.15, 1.04],
                      ['poli-Si', 50, 9.55, 0.55],
                      ['mono-Si', 130, 11.8, 0.75]]
```

```
# se creeaza o functie care va popula aceste dictionare
def adauga_elem(lista_denumiri, lista_atribute, lista_specificatii):
    temp_dict = dict()
    index = 0
    while index < len(lista_denumiri):
        for atribut, specificatie in zip(lista_atribute,
lista_specificatii[index]):
            temp_dict[atribut] = specificatie
        panouri_PV[lista_denumiri[index]] = temp_dict.copy()
        index += 1
    return panouri_PV

# se altereaza dictionarul utilizand funcitia definita anterior
```

```
panouri_PV = adauga_elem(lista_denumiri, lista_atribute, lista_specificatii)
for k, v in panouri_PV.items():
    print(k, '-', v)
OUTPUT:
Dictionarul are 3 perechi cheie-valoare.
Schott Poly - {'tehnologie': 'mono-Si', 'capacitate - W': 10, 'eficienta - %': 7.75, 'suprafata - mp': 0.13}
BP Solar - {'tehnologie': 'mono-Si', 'capacitate - W': 100, 'eficienta - %': 15.6, 'suprafata - mp': 1.26}
DelSolar - {'tehnologie': 'poli-Si', 'capacitate - W': 200, 'eficienta - %': 13.15, 'suprafata - mp': 1.04}
SunWorld - {'tehnologie': 'poli-Si', 'capacitate - W': 50, 'eficienta - %': 9.55, 'suprafata - mp': 0.55}
DelSolar2 - {'tehnologie': 'mono-Si', 'capacitate - W': 130, 'eficienta - %': 11.8, 'suprafata - mp': 0.75}
```

4.5. Tupluri: `tuple`

În esență, obiectele de tip tuplu sunt liste **imutabile**, având aceleași caracteristici, dar nu pot fi alterate după ce au fost create. Mai mult, un tuplu este o **secvență** de date, care este:

↔ `indexabil` – poate fi accesat prin specificarea indecșilor – indexare în bază 0;
↔ `ușor` – ocupă spații relativ mici în memorie comparativ cu alte secvențe de date;
↔ `ordonat` – ordinea elementelor se păstrează;
↔ `eterogen` – poate conține obiecte de diverse tipuri;
↔ `iterabil` – poate fi parcurs în bucle iterative;
↔ `imutabil` – nu poate fi modificat după ce a fost creat;
↔ `concatenabil` – pot fi imbricate mai multe tupluri (adunate, multiplicate etc.);
↔ `multidimensional` – poate avea mai mult de o dimensiune (vectori/matrice);
↔ `hash` – este un tablou de referințe la obiecte; pot fi chei pentru dicționare.

Pentru a instanția un tuplu este suficient să se specifice indicatorul (numele variabile), urmat de funcția constructor specifică `tuple()` - `f60` sau de două paranteze drepte - `()`. În exemplul următor sunt prezentate ambele metode de formare a unui obiect tuplu.

```
tuplu1 = tuple()
print(type(tuplu1))
tuplu2 = ()
print(type(tuplu2))
OUTPUT:
<class 'tuple'>
<class 'tuple'>
```

Există și o a treia variantă de instanțiere a unui obiect tuplu – scrierea valorilor constituente separate prin virgulă, dar fără folosirea parantezelor drepte. Totuși, această abordare nu poate fi utilizată pentru generarea unui tuplu fără elemente!

```
tuplu3 = 1, 3, 4
print(type(tuplu3))
OUTPUT:
<class 'tuple'>
```

În exemplul din fragmentul următor de cod este creat un tuplu eterogen ce conține o serie de date Python de diferite tipuri. Numărul de elemente din tuplu se poate determina folosind funcția încorporată `len()` - f37. În plus, este exemplificată eroarea generată în cazul în care se dorește a modifica un element din tuplu prin indexare și reatribuire – eroare `TypeError: 'tuple' object does not support item assignment`.

```
tuplu_eterogen = (1, 1.3, 'text', [1, 0], (0, 5))
print(f'Tuplul {tuplu_eterogen} are {len(tuplu_eterogen)} elemente.')
# se incearca modificarea elementului 2 - indexul 1
tuplu_eterogen[1] = 3
print('Noul tuplu este: ', tuplu_eterogen)
OUTPUT:
Tuplul (1, 1.3, 'text', [1, 0], (0, 5)) are 5 elemente.

TypeError: 'tuple' object does not support item assignment
```

OBSERVAȚIE. Ca și în cazul obiectelor tip string și liste, tuplurile sunt indexabile din 0.

```
# se indexeaza tuplul
print(f'Al doilea element din tuplu este: {tuplu_eterogen[1]}')
OUTPUT:
Al doilea element din tuplu este: 1.3
```

TUPLE COMPREHENSION – COMPREHENSIUNEA TUPLURILOR

Mecanismul de comprehensiune nu se aplică în cazul tuplurilor, adaptarea sintaxei utilizate în cazul listelor și dicționarelor este legală în Python, dar va rezulta un obiect tip generator – v. informația despre cuvântul cheie `yield` - kw.6 și funcția încorporată `next()` - f43. Principalele cauze pentru care nu există comprehensiunea tuplurilor sunt că aceste obiecte sunt imutabile, dar și faptul că utilizarea parantezelor este alocată pentru crearea obiectelor tip generator. Pentru a exemplifica acest aspect se creează inițial un tuplu care conține doar date numerice și se încearcă aplicarea mecanismului de comprehensiune pentru a crea un nou tuplu care împarte la 2 toate elementele din tuplul inițial. Rezultatele sunt prezentate în fragmentul următor de cod.

```
tuplu_numere = 1, 2.4, 8, 9.5, -45, 0.7
print('Primul tuplu este:', tuplu_numere)
tuplu_nou = (numar/2 for numar in tuplu_numere)
print('Al doilea tuplu este:', tuplu_nou)
OUTPUT:
Primul tuplu este: (1, 2.4, 8, 9.5, -45, 0.7)
Al doilea tuplu este: <generator object <genexpr> at 0x000001B3565B5A40>
```

Chiar dacă nu sunt afișate direct cu funcția `print()`, datele conținute în generator pot fi obținute printr-o instrucțiune iterativă sau prin funcția încorporată `next()` - f43.

```
# va returna prima valoare - 0.5
print(next(tuplu_nou))
# va returna a doua valoare - 1.2
print(next(tuplu_nou))
# va returna celelalte valori pana la sfarsitul secventei
for element in tuplu_nou:
    print(element)
OUTPUT:
4.0
4.75
-22.5
0.35
```

Totuși, chiar dacă nu sintaxa uzuală este utilizată în alt scop, există variante de a imita mecanismul de comprehensiune şi pentru generarea tuplurilor. În primul rând se poate utiliza obiectul tip generator rezultat ca argument al funcției specifice constructor `f60`. În acest caz, orice iterator este transformat explicit în tupluri, însemnând că această abordare poate fi aplicată şi pentru liste, dicționare şi seturi – orice iterabil care permite utilizarea mecanismului de comprehensiune pentru generarea obiectelor.

```
tuplu_numere = 1, 2.4, 8, 9.5, -45, 0.7
print('Primul tuplu este:', tuplu_numere)
tuplu_nou = tuple((numar/2 for numar in tuplu_numere))
print('Al doilea tuplu este:', tuplu_nou)
print(f'Obiectul tuplu_nou este de tipul {type(tuplu_nou)}')
OUTPUT:
Primul tuplu este: (1, 2.4, 8, 9.5, -45, 0.7)
Al doilea tuplu este: (0.5, 1.2, 4.0, 4.75, -22.5, 0.35)
Obiectul tuplu_nou este de tipul <class 'tuple'>
```

Cum funcția constructor funcționează, este de aşteptat ca şi instanțierea implicită utilizând un tuplu literal să funcționeze (folosirea parantezelor simple). Dar acest lucru nu este adevărat. Omiterea funcției constructor va returna tot un obiect generator, chiar dacă se folosesc paranteze duble. Totuși, recurgând la un mecanism specific iterabilelor – despachetarea valorilor – se poate omite folosirea funcției constructor. Despachetarea valorilor dintr-un iterabil se realizează utilizând semnul asterisc înaintea iterabilului: `*obiect_iterabil`.

```
tuplu_numere = 1, 2.4, 8, 9.5, -45, 0.7
print('Primul tuplu este:', tuplu_numere)
tuplu_nou = ((numar/2 for numar in tuplu_numere))
print('Al doilea tuplu este:', tuplu_nou)
print(f'Obiectul tuplu_nou este de tipul {type(tuplu_nou)}')
# utilizand despachetarea valorilor
print('Utilizand despachetarea valorilor din generator:')
tuplu_nou = *(numar/2 for numar in tuplu_numere),
print('Al doilea tuplu este:', tuplu_nou)
print(f'Obiectul tuplu_nou este de tipul {type(tuplu_nou)}')
OUTPUT:
Primul tuplu este: (1, 2.4, 8, 9.5, -45, 0.7)
Al doilea tuplu este: <generator object <genexpr> at 0x000001B356636A80>
```

```
Obiectul tuplu_nou este de tipul <class 'generator'>
Utilizand despachetarea valorilor din generator:
Al doilea tuplu este: (0.5, 1.2, 4.0, 4.75, -22.5, 0.35)
Obiectul tuplu_nou este de tipul <class 'tuple'>
```

OBSERVAȚIE 1. Atenție la utilizarea mecanismului de despachetare a datelor: obiectul apelat este în esență un tuplu generat fără paranteze: `tuplu_nou = *(numar/2 for numar in tuplu_numere),` . A nu se uita virgula de la sfârșitul generatorului!
OBSERVAȚIE 2. Utilizarea virgulei este un element specific tuplurilor. Dacă la crearea unui tuplu literal nu este neapărat nevoie a folosi paranteze, utilizarea virgulei este cea care indică interpretorului Python că acel obiect este tuplu, și nu o valoare singulară. Mai mult, când se dorește crearea unui tuplu cu o singură valoare, aceasta trebuie urmată **neapărat** de o virgulă: `(1,)` sau chiar `1,` – fără utilizarea parantezelor – v. exemplul următor.

```
text = ('inginer') # obiect string
print(type(text))
text = ('inginer',) # obiect tuplu
print(type(text))
OUTPUT:
<class 'str'>
<class 'tuple'>
```

Mai mult, se poate utiliza funcția sau mecanismul de despachetare și în conjuncție cu o listă generată folosind LIST COMPREHENSION – COMPREHENSIUNEA LISTELOR, dar și dicționare. În fragmentele de cod următoare sunt exemplificate ambele posibilități cu ambele tipuri de date. E nevoie de remarcat faptul că la utilizarea dicționarelor, dacă nu se specifică altfel prin metoda `163`, doar cheile dicționarului vor fi considerate elemente în tuplu.

```
tuplu_numere = 1, 2.4, 8, 9.5, -45, 0.7
print('Primul tuplu este:', tuplu_numere)
tuplu_nou = tuple([numar/2 for numar in tuplu_numere])
print('Al doilea tuplu este:', tuplu_nou)
print(f'Obiectul tuplu_nou este de tipul {type(tuplu_nou)}')
# utilizand despachetarea valorilor
print('Utilizand despachetarea valorilor din lista:')
tuplu_nou = *([numar/2 for numar in tuplu_numere]),
print('Al doilea tuplu este:', tuplu_nou)
print(f'Obiectul tuplu_nou este de tipul {type(tuplu_nou)}')
OUTPUT:
Primul tuplu este: (1, 2.4, 8, 9.5, -45, 0.7)
Al doilea tuplu este: <generator object <genexpr> at 0x000001B356636A80>
Obiectul tuplu_nou este de tipul <class 'generator'>
Utilizand despachetarea valorilor din lista:
Al doilea tuplu este: (0.5, 1.2, 4.0, 4.75, -22.5, 0.35)
Obiectul tuplu_nou este de tipul <class 'tuple'>
```

```
numere = {"one": 1, "two": 2, "three": 3, "four": 4}
print(numere)
# utilizarea functiei tuple()
numere_inversate = tuple({v:k for (k,v) in numere.items()})
print(numere_inversate)
print(type(numere_inversate))
# utilizarea mecanismului de despachetare al datelor
numere_inversate = *{v:k for (k,v) in numere.items()},
print(numere_inversate)
print(type(numere_inversate))
# utilizarea mecanismului de despachetare al datelor, specificand utilizarea valorilor
numere_inversate = *{v:k for (k,v) in numere.items()}.values(),
print(numere_inversate)
print(type(numere_inversate))
OUTPUT:
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
(1, 2, 3, 4)
<class 'tuple'>
(1, 2, 3, 4)
<class 'tuple'>
('one', 'two', 'three', 'four')
<class 'tuple'>
```

INDEXAREA TUPLURILOR

În esență un obiect tip tuplu se poate indexa identic cu obiectele tip `string` sau `list` – se indexează începând cu valoarea 0 și se pot trunchia/felia. Mai mult, acceptându-se tupluri imbricate, acestea se indexează utilizând indecși multipli: `obiect_tuplu[index1][index2]` – identic cu indexarea listelor imbricate – INDEXAREA LISTELOR. De asemenea, tuplurile acceptă indexare negativă, dar și partiționarea pentru a obține elemente multiple: `obiect_tuplu[start:stop:pas]`. Trebuie menționat însă faptul că partiționarea unui tuplu în scopul de a-i modifica valoarea nu este posibilă, acestea fiind date imutabile. În exemplul următor de cod sunt prezentate exemple din fiecare caz.

```
# se instantiaza un tuplu imbricat
student_id_1 = (
    ("EIT", 8),
    ("TCM", 7),
    ("MMNE", 7),
    ("MAE", 6),
    ("EE", 10),
    ("MH", 7),
    ("TAE", 5),
)
# se indedeaza tuplul pentru a obtine diverse note
print(f'Nota la materia EIT este: {student_id_1[0][1]}')
print(f'Nota la materia TAE este: {student_id_1[-1][1]}')
```

```
# se indedeaza tuplul pentru a obtine diverse note
print(f'Notele la metriile EIT si TMC sunt: {student_id_1[0:2]}')
OUTPUT:
Nota la materia EIT este: 8
Nota la materia TAE este: 5
Notele la metriile EIT si TMC sunt: (('EIT', 8), ('TCM', 7))
```

ITERAREA TUPLURILOR

Fiind date secvențiale, tuplurile permit parcurgerea iterativă a elementelor constituente. Pentru a ilustra acest aspect atribuit tuplurilor se propune un tuplu imbricat format din tupluri interioare cu perechi de obiecte reprezentând materii și note. Tuplul principal se iterează, se vor lua valorile numerice si se va returna media obținută de un student în respectivul semestru universitar. Valorile numerice care servesc drept note se află pe poziția 2 (index 1) în fiecare tuplu intern. La fiecare iterație se adaugă această valoare la o variabilă tip `int` instanțiată cu valoarea `0` (`nota_totala = 0`). Următorul pas este de a calcula efectiv media aritmetică prin împărțirea sumei totale la numărul de materii considerate – în acest caz, fiind reprezentat de lungimea tuplului principal (numărul de elemente). În plus, se utilizează funcția f53 prin care se trunchiază rezultatul la două zecimale specificate prin argumentul `2`.

```
student_id_1 = (
    ("EIT", 8),
    ("TCM", 7),
    ("MMNE", 7),
    ("MAE", 6),
    ("EE", 10),
    ("MH", 7),
    ("TAE", 5),
)
nota_totala = 0
for materie in student_id_1:
    nota_totala += materie[1]
# se calculează media semestrului
medie_semestru = round(nota_totala/len(student_id_1), 2)
print(f'Media in anul III, semestrul 1 este: {medie_semestru}')
OUTPUT:
Media in anul III, semestrul 1 este: 7.14
```

OPERAȚII CU TUPLURI

Obiectele tip tuplu acceptă toate operațiile aferente listelor (mai puțin modificarea după instanțiere) și șirurile de caractere, dar și operații specifice cum ar fi împachetarea și despachetarea datelor. Acestea din urmă au o importanță fundamentală în sintaxa Python, permițând o serie de simplificări în scrierea codului.

Împachetarea unui tuplu este, în esență, instanțierea propriu-zisă a tuplului respectiv. Când un obiect de tip tuplu este inițializat cu valori, acestea sunt *împachetate* în identificatorul respectiv. Spre exemplu, crearea unor coordonate spațiale. În exemplul următor de cod sunt împachetate 3 valori numerice într-o variabilă denumită `punct_A`. Aceleași valori sunt mai apoi ***despachetate***

și atribuite unor noi variabile x_A, y_A, z_A. După cum se poate observa, aceste noi variabile au fix valorile conținute în tuplul inițial. Exemplul este completat cu despachetarea unor noi valori dintr-un tuplu pentru variabilele x_B, y_B, z_B – corespunzătoare unui punct B din spațiul euclidian și calculul distanței euclidiene dintre cele două puncte. Trebuie să se aibă în vedere faptul că la despachetarea tuplurilor, numărul de variabile utilizate trebuie să fie egal cu numărul de elemente din tuplu, altfel generându-se o eroare de tipul ValueError: not enough values to unpack (expected 4, got 3) – exemplu pentru cazul în care se utilizează 4 variabile și tuplul conține doar 3 date sau ValueError: too many values to unpack (expected 2) – exemplu pentru cazul în care se utilizează 2 variabile și tuplul conține 3. Pentru a se evita o astfel de eroare este mereu o bună practică a se analiza inițial câte elemente conține tuplul respectiv utilizându-se funcția încorporată f37().

```
import math
# impachetarea tuplului - in esenta instantierea lui
punct_A = 1, 2, 4
# despachetarea tuplului
x_A, y_A, z_A = punct_A
print('x_A:', x_A)
print('y_A:', y_A)
print('z_A:', z_A)
# se despacheteaza un tuplu atribuind direct valorile
x_B, y_B, z_B = 2, 5, -4
# se calculeaza distanta euclidiana dintre cele 2 puncte A, B
distanta = math.sqrt((x_A-x_B)**2 + (y_A-y_B)**2 + (z_A-z_B)**2)
print('Distanta dintre punctele A(1,2,4) si B(2,5,-4) este')
print(round(distanta,3))
OUTPUT:
x_A: 1
y_A: 2
z_A: 4
Distanta dintre punctele A(1,2,4) si B(2,5,-4) este 8.602
```

O întrebuințare a mecanismului de despachetare a tuplurilor foarte des regăsită în programarea în mediul Python este aceea de interschimbare a valorilor dintre două variabile concomitent. Să presupunem că există o variabilă ce conține valoarea 100 și alta care conține --100. Cele două variabile se pot interschimba folosind următoarea sintaxă: numar2, numar1 = numar1, numar2. În acest caz, variabila numar2 va indica către valoarea din numar1 și variabila numar1 va indica către valoarea din numar2.

```
numar1 = 100
numar2 = -100
print('initial numar1:', numar1)
print('initial numar2:', numar2)
numar2, numar1 = numar1, numar2
print('dupa interschimbare numar1:', numar1)
print('dupa interschimbare numar2:', numar2)
OUTPUT:
initial numar1: 100
initial numar2: -100
dupa interschimbare numar1: -100
dupa interschimbare numar2: 100
```

Alternativa la acest mecanism de despachetare a tuplurilor este de a folosi o variabilă temporară care va stoca valoarea uneia dintre variabile. Pentru comparație este exemplificat și acest caz, specificându-se faptul că nu este indicat a se folosi în limbajul de programare Python.

```
numar1 = 100
numar2 = -100
print('initial numar1:', numar1)
print('initial numar2:', numar2)
# variabila temporara
temp = numar1
numar1 = numar2
numar2 = temp
print('dupa interschimbare numar1:', numar1)
print('dupa interschimbare numar2:', numar2)
OUTPUT:
initial numar1: 100
initial numar2: -100
dupa interschimbare numar1: -100
dupa interschimbare numar2: 100
```

Interschimbarea valorilor a două variabile se poate utiliza, printre multe alte aplicații, pentru generarea unei secvențe de numere ce îndeplinesc condiția lui Fibonacci: un număr este suma a două numere anterioare: 0, 1, 1, 2, 3, 5, ... În exemplul de cod următor este creat un script care va genera un număr fix de elemente din secvența Fibonacci. Inițial se utilizează funcția `f32` pentru a se primi informația de la utilizator. Cum această funcție returnează un obiect tip `string`, acesta va trebui să fie convertit explicit la un număr tip `int` utilizând funcția constructor specifică `f33`. Primele două numere din secvență sunt `0, 1` care se vor atribui variabilelor denumite `num1, num2` utilizând mecanismul de despachetare a tuplurilor. Urmează o secvență de expresii logice care afișează diverse mesaje în funcție de numărul introdus de utilizator prin intermediul variabilei `numar_elemente`. Dacă acest număr este mai mare de 1, atunci se vor calcula și afișa toate cele `numar_elemente` din secvența Fibonacci. Numerele sunt calculate utilizând aceeași logică: `urmatorul_numar = num1 + num2`.

```
numar_elemente = int(input('Numarul de elemente din secventa Fibonacci: '))
# primele doula numere din secventa - despachetarea unui tuplu
num1, num2 = 0, 1
# al treilea numar este calculat ca suma primelor 2
urmatorul_numar = num1 + num2
count = 1 # se utilizeaza un contor pentru a opri executia buclei while
if numar_elemente <= 0:
    print('Introduceti un numar mai mare ca 0.')
elif numar_elemente == 1:
    print('Primul element Fibonnaci este', num1)
else:
    while count <= numar_elemente:
        print(urmatorul_numar, end=" ")
        count += 1
        # se updateaza valorile din variabile
```

```
        num1, num2 = num2, urmatorul_numar
        urmatorul_numar = num1 + num2
    print()
OUTPUT:
1 2 3 5 8 13 21 34 75 89
```

Sintaxa Python conține, de asemenea, și un operator de despachetare tupluri – * – care poate fi folosit pentru a spori flexibilitatea acestui mecanism. De exemplu, acest operator se poate utiliza pentru a colecta mai multe valori într-o singură variabilă atunci când numărul de variabile din stânga nu se potrivește cu numărul de elemente din tuplul din dreapta. În exemplul următor se creează un obiect tip tuplu care conține 10 întregi numere consecutive începând din 0 și până la 9. Pentru aceasta s-a utilizat funcția `f50()` cu argumentul `10`. Deși tuplul conține 10 elemente, acestea se despachetează în două variabile. Prima variabilă denumită `primele_numere` este utilizată în conjuncție cu operatorul special * și va conține primele elemente în afară de ultimul care va fi atribuit variabilei `ultimul_numar`.

```
numere = tuple(range(10))
print('tuplul initial:',numere)
*primele_numere, utlimul_numar = numere
print('primele numere:', primele_numere)
print('utlimul numar:', utlimul_numar)
OUTPUT:
tuplul initial: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
primele numere: [0, 1, 2, 3, 4, 5, 6, 7, 8]
utlimul numar: 9
```

Un alt exemplu este utilizarea operatorului de despachetare pentru datele din mijloc, dar și pentru ultimele numere:

```
numere = tuple(range(10))
primul, *numere_mijloc, ultimul = numere
print('primult numar:', primul)
print('numerele de la mijloc:', numere_mijloc)
print('ultimul numar:', ultimul)
OUTPUT:
primult numar: 0
numerele de la mijloc: [1, 2, 3, 4, 5, 6, 7, 8]
ultimul numar: 9
```

```
numere = tuple(range(10))
primul, aldoilea, altreilea, *ultimele = numere
print(f'primul numar: {primul}, al doilea: {aldoilea}, al treilea:
{altreilea}')
print(f'ultimele numere din secventa: {ultimele}')
OUTPUT:
primul numar: 0, al doilea: 1, al treilea: 2
ultimele numere din secventa: [3, 4, 5, 6, 7, 8, 9]
```

De foarte multe ori, în practică funcțiile sau metodele implementate de programator sau încorporate în sintaxa Python returnează mai mult de o singură valoare. Pentru aceasta, este suficient ca valorile indicate după cuvântul cheie `kw.6` să fie separate de virgulă – generându-se astfel un obiect tip tuplu. În fragmentul următor de cod este creată o funcție care va analiza un tip de dată secvențial și va returna valorile minime și maxime. Funcția va accepta drept argument o secvență de numere sau string și va folosi funcțiile încorporate **`f42`** și **`f41`** pentru a genera valoarea minimă și cea maximă din secvență. Aceste valori sunt returnate mai apoi sub forma unui tuplu – pe prima poziție (index 0) fiind minimul și pe a doua poziție (index 1) fiind maximul. Funcționalitatea este demonstrată utilizând o listă ce conține date imaginare despre temperatura exterioară și un șir de caractere care conține un text aleatoriu.

```
def min_max(secventa):
    if len(secventa) <= 1:
        return f'{secventa}, trebuie sa aiba minim 2 elemente'
    return min(secventa), max(secventa)

temperaturi_exterioare = [0, -1, -2.3, 3, 1.3, -2, 3]
temp_minima = min_max(temperaturi_exterioare)[0]
print("temperatura minima este:", temp_minima)
temp_maxima = min_max(temperaturi_exterioare)[1]
print('temperatura maxima este: ', temp_maxima)

sir_caractere = 'inteligenta_artificiala'
litera_minima = min_max(sir_caractere)[0]
print("prima litera:", litera_minima)
litera_maxima = min_max(sir_caractere)[1]
print('utlima litera: ', litera_maxima)
OUTPUT:
temperatura minima este: -2.3
temperatura maxima este:  3
prima litera: _
utlima litera:  t
```

Obiectele de tip tuplu pot fi, de asemenea, **concatenate** (adunare) și **repetate** valorile (înmulțirea) acestora. Merită accentuat faptul că ambele operații creează un nou tuplu și nu modifică tuplul existent – obiect imutabil!

```
tuplu_impare = 1, 3, 5
tuplu_pare = 2, 4, 6
tuplu_adunat = tuplu_impare + tuplu_pare
print('concatenarea tuplurilir:', tuplu_adunat)
print('repetarea tuplului 1 de 2 ori:', tuplu_impare*2)
print('repetarea tuplului 2 de 3 ori:', tuplu_pare*3)
OUTPUT:
concatenarea tuplurilir: (1, 3, 5, 2, 4, 6)
repetarea tuplului 1 de 2 ori: (1, 3, 5, 1, 3, 5)
repetarea tuplului 2 de 3 ori: (2, 4, 6, 2, 4, 6, 2, 4, 6)
```

Ultimele două operații ce se pot aplica tuplurilor sunt **inversarea** și **sortarea**. Pentru aceste două operații se pot utiliza funcțiile încorporate `f52` și `f57`, așa cum este prezentat în fragmentul următor de cod. Merită menționat că inversarea se poate face și prin operațiunea de feliere a întreg tuplului, cu pasul de parcurgere al obiectului egal cu -1: `[::-1]`.

```
print(f'intial: {tuplu_impare}')
tuplu_impare_r = tuple(reversed(tuplu_impare))
print(f'dupa inversare: {tuplu_impare_r}')
print(f'initial: {tuplu_pare}')
tuplu_pare_s = tuple(sorted(tuplu_pare))
print(f'dupa sortare: {tuplu_pare_s}')
tuplu_pare_r = tuplu_pare[::-1]
print(f'dupa inversare: {tuplu_pare_r}')
OUTPUT:
intial: (1, 3, 5)
dupa inversare: (5, 3, 1)
initial: (4, 6, 2)
dupa sortare: (2, 4, 6)
dupa inversare: (2, 6, 4)
```

METODE SPECIFICE TUPLURILOR

Caracterul imutabil al tuplurilor și similitudinea lor cu listele limitează numărul de metode specifice al acestui tip de date la doar două: `count()` și `index()`.

`met_tuple.1.` `obiect_tuple.count(element)` – returnează numărul de apariții al parametrului obligatoriu `element`. Dacă parametrul `element` nu este găsit, va returna 0.

```
tuplu_numere_pare = 2, 2, 4, 6, 2
print(f'numarul 2 apare de {tuplu_numere_pare.count(2)} ori')
print(f'litera "x" apare de {tuplu_numere_pare.count("i")} ori')
OUTPUT:
numarul 2 apare de 3 ori
litera "x" apare de 0 ori
```

`met_tuple.2.` `obiect_tuple.index(element, start, end)` – returnează indexul poziției pe care se regăsește prima apariție a parametrului obligatoriu `element`. Dacă parametrul `element` nu este regăsit, metoda va genera o eroare de tipul `ValueError: tuple.index(x): x not in tuple`. Parametrii opționali `start` și `end` sunt utilizați pentru a specifica spațiul de căutare al elementului. Implicit, aceștia au valorile: `start` = 0 și `end` = numărul de elemente - 1, practic întregul obiect tip tuplu.

```
tuplu_numere_pare = 1, 2, 2, 4, 6, 2
print(f'prima aparitie a numarului 2 este pe pozitia
{tuplu_numere_pare.index(2)}')
OUTPUT:
prima aparitie a numarului 2 este pe pozitia 1
```

```
tuplu_numere_pare = 1, 2, 2, 4, 6, 2
print(f'prima aparitie a numarului 3 este pe pozitia
{tuplu_numere_pare.index(3)}')
OUTPUT:
ValueError: tuple.index(x): x not in tuple
```

4.6. Seturi: set

Ultimul tip de date esențial încorporat în sintaxa limbajului de programare Python este reprezentat de seturile de date care se comportă similar cu mulțimile din matematică – acceptă aceleași tipuri de operații: intersecție, reuniune, adunare etc. În esență, seturile sunt:

↔ `neordonate` – ordinea elementelor nu se păstrează;
↔ `indexabile` – pot fi accesate prin specificarea indecșilor – indexare în bază 0;
↔ `mutabile` – pot fi modificate după ce au fost create;
↔ `neschimbabile` – pot conține **doar** obiecte **imutabile**!;
↔ `elemente unice` – nu pot exista date dublate în interiorul unui set.

Pentru a crea un set, se poate utiliza funcția constructor încorporată `f54()` sau acolade - `{}`. La utilizarea funcției se poate apela orice iterabil existent în sintaxa Python și în cazul în care acest iterabil conține elemente duplicat, funcția păstrează o singură instanță a acelei date. În fragmentul de cod următor este instanțiată o listă care conține mai multe date numerice, printre care 3 numere `1`. După utilizarea funcției `f54()` se poate observa faptul că numerele duplicate au fost eliminate, în timp ce ordinea introducerii elementelor nu s-a păstrat.

```
lista_numere = [1, 3, 2, 1, 6, 8, 1, 4]
print("lista de numere este:", lista_numere)
set_numere = set(lista_numere)
print("setul de numere este", set_numere)
OUTPUT:
lista de numere este: [1, 3, 2, 1, 6, 8, 1, 4]
setul de numere este {1, 2, 3, 4, 6, 8}
```

În exemplul din fragmentul următor de cod este utilizată funcția constructor pentru a instanția un obiect tip `set` plecând de la un obiect tip șir de caractere. Aceleași concluzii se pot observa și aici – obiectele duplicat sunt înlăturate și ordinea elementelor este aleatorie.

```
text = 'inginerie'
print('lista din text:', list(text))
set_text = set(text)
print('set din text:', set_text)
OUTPUT:
lista din text: ['i', 'n', 'g', 'i', 'n', 'e', 'r', 'i', 'e']
set din text: {'g', 'e', 'i', 'n', 'r'}
```

OBSERVAȚIE. Chiar dacă setul de acolade se utilizează și pentru instanțierea dicționarelor, diferența între cele două tipuri de date constă chiar în definirea acestora din urmă: *colecții de date asociative care mapează anumite* ***valori*** *de* ***chei unice****, generând astfel perechi* ***unice*** *cheie-valoare (key-value)*. Drept urmare, interpretorul Python face diferența între cele două tipuri de date în funcție de modul în care acestea sunt definite – în fragmentul următor de cod sunt prezentate ambele posibilități și este evidențiată diferența de obiect instanțiat. Totuși, acoladele nu pot genera un set gol – sunt destinate exclusiv generării unui dicționar gol.

```
text = 'inginerie'
set_text1 = set(text)
print("Utilizand functia set() se genereaza un:", type(set_text1))
set_text2 = {text}
print("Utilizand acoladele {} se genereaza un:", type(set_text1))
dict_text = {text: "energetica"}
print("Daca se specifica o valoarea se genereaza un:", type(dict_text))
set_gol = {}
print('Utilizand acoladele {} fara valoare se genereaza un:', type(set_gol))
OUTPUT:
Utilizand functia set() se genereaza un: <class 'set'>
Utilizand acoladele {} se genereaza un: <class 'set'>
Daca se specifica o valoarea se genereaza un: <class 'dict'>
Utilizand acoladele {} fara valoare se genereaza un: <class 'dict'>
```

Elementele ce compun obiectul set trebuie neapărat să fie obiecte imutabile. În caz contrar, interpretorul Python va genera o eroare: `TypeError: unhashable type: 'list'`. În fragmentul următor de cod este prezentat un asemenea exemplu. Merită menționat faptul că, utilizând funcția încorporată `f54()`, se poate genera un obiect set, deoarece aceasta acceptă ca argument un iterabil pe care îl convertește intern la un obiect tip set.

```
# utilizand functia constructor set() - functioneaza
lista_numere = [1, 3, 2, 1, 6, 8, 1, 4]
set_numere = set(lista_numere)
print(set_numere)
OUTPUT:
{1, 2, 3, 4, 6, 8}
```

```
# utilizand acoladele - nu functioneaza
lista_numere = [1, 3, 2, 1, 6, 8, 1, 4]
set_numere = {lista_numere}
print(set_numere)
OUTPUT:

TypeError: unhashable type: 'list'
```

Diferența majoră între cele două modalități de instanțiere a unui obiect tip set este aceea că, utilizând acoladele – `{}`, fiecare obiect introdus devine un element distinct al setului respectiv, chiar dacă este un iterabil în sine! Comparativ, funcția încorporată `f54()` acceptă un singur element iterabil pe care îl convertește într-un obiect set. În cazul în care se specifică mai mult de un argument, funcția constructor va genera o eroare de tip: `TypeError: set expected at most 1 argument, got 2`. În fragmentul următor de cod sunt prezentate ambele cazuri, încercând instanțierea unui set utilizând două obiecte imutabile – șiruri de caractere. Mai mult, dacă funcția constructor `f54()` nu este utilizată în conjuncție cu un obiect iterabil (cum ar fi, spre exemplu, o dată numerică), aceasta va genera o eroare de tip `TypeError: 'int' object is not iterable` – exemplu pentru utilizarea unui argument tip `int`. De asemenea, acest caz este exemplificat în fragmentul următor de cod.

Sugestie: .v. teoria din capitolul dedicat *FUNCȚII ÎNCORPORATE* - `f54`.

```
# utilizand {} - functioneaza
text1 = 'inginerie'
text2 = 'energetica'
set_text = {text1, text2}
print('Utilizand acoladele {}:')
print(set_text, 'este de tipul', type(set_text))
OUTPUT:
Utilizand acoladele {}:
{'inginerie', 'energetica'} este de tipul <class 'set'>
```

```
# utilizand functia set() - nu functioneaza
text1 = 'inginerie'
text2 = 'energetica'
set_text = set(text1, text2)
print('Utilizand functia set():')
print(set_text, 'este de tipul', type(set_text))
OUTPUT:

TypeError: set expected at most 1 argument, got 2
```

```
print(set(1))
OUTPUT:

TypeError: 'int' object is not iterable
```

ITERAREA SETURILOR

Fiind containere, adică date secvențiale, seturile pot fi foarte ușor parcurse printr-o operație iterativă utilizând bucla `for` sau `while`. În fragmentul următor de cod este prezentată cea mai simplă și utilizată iterare a datelor secvențiale finite: folosirea buclei for. Mai mult, se folosește funcția încorporată `f18()` pentru a returna atât elementul cât și indexul acestuia sub forma unui tuplu. Cum în Python indexarea începe din `0`, pentru a afișa poziția efectivă, indexul reprezentat prin variabila locală `i`, este incrementată cu `1`. A se observa, din nou, distribuția aleatorie a elementelor în obiectul set – dacă la instanțiere ordinea a fost: `'inginerie'`,`'energetica'`, `1`, `4`, `(100, 3, 56)`, la iterare ordinea a devenit: `1`, `4`, `(100, 3, 56)`, `'inginerie'`, `'energetica'`.

```
text1 = 'inginerie'
text2 = 'energetica'
set_complex = {text1, text2, 1, 4, (100, 3, 56)}
for i, element in enumerate(set_complex):
    print(f'elementul de pe pozitia {i+1} este: {element}')
OUTPUT:
elementul de pe pozitia 1 este: 1
elementul de pe pozitia 2 este: 4
elementul de pe pozitia 3 este: (100, 3, 56)
elementul de pe pozitia 4 este: energetica
elementul de pe pozitia 5 este: inginerie
```

INDEXAREA SETURILOR

Dat fiind faptul că seturile sunt imaginea mulțimilor din analiza matematică, indexarea și/sau partiționarea/felierea acestora nu au logică, fiind în afara scopului lor. În cazul în care se încearcă indexarea unui astfel de obiect Python, interpretorul va genera o eroare de tipul `TypeError: 'set' object is not subscriptable`, pentru ambele operații de indexare.

```
text1 = 'inginerie'
text2 = 'energetica'
set_text = {text1, text2}
print('Primul element al setului este:', set_text[1])
OUTPUT:
```

```
TypeError: 'set' object is not subscriptable
```

OPERAȚII CU SETURI ȘI METODE ASOCIATE SETURILOR

Totuși, există operații care sunt specifice lucrului cu obiecte tip set în Python. Acestea sunt similare cu operațiile pe mulțimile din matematică: *reuniune*, *intersecție*, *diferență* etc. Majoritatea operațiilor pentru seturile Python se pot realiza în două moduri: utilizând un operator specific sau o metodă asociată cu acest tip de dată, fapt pentru care ambele metode vor fi prezentate simultan. Pentru exemplificare, se vor considera două seturi care conțin atât elemente comune cât și elemente unice. Fiecare set este compus din elemente imutabile: șiruri de caractere, date numerice și tupluri:

```
# se instantiaza doua seturi distincte
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
print(f'Setul 1: {set1}')
print(f'Setul 2: {set2}')
OUTPUT:
Setul 1: {'inginer', 1.3, 'artificial', (3, 4)}
Setul 2: {(2, 3), 1.3, 'inteligenta', 'inginer'}
```

`met_set.1.` **Reuniunea** seturilor, care în matematică este definită drept mulțimea tuturor elementelor (comune și necomune) ale celor două mulțimi luate o singură dată, are scopul de a combina cele două seturi. Acest lucru se poate realiza atât prin utilizarea operatorului specific ”|„ cât și prin metoda specifică `union()`. Sintaxa generată a metodei este `obiect_set.union(set1, set2, set3,...)` și returnează un nou set compus din elementele comune și necomune ale tuturor seturilor introduse ca argumente.

```
# se instantiaza doua seturi distincte
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
# utilizand operatorul specific |
print('set1 reunit cu set2:', set1 | set2)
# utilizand metoda specifica union()
print('set1 reunit cu set2:', set1.union(set2))
OUTPUT:
set1 reunit cu set2: {1.3,(3, 4),'artificial',(2, 3),'inteligenta','inginer'}
set1 reunit cu set2: {1.3,(3, 4),'artificial',(2, 3),'inteligenta','inginer'}
```

OBSERVAȚIE 1. Chiar dacă în esență ambele metode au același rezultat, există o diferență subtilă între ele: la utilizarea operatorului specific ”|„ ambii operanzi trebuie să fie obiecte de tip set – în caz contrar va genera o eroare de tip: `TypeError: unsupported operand type(s) for |: 'set' and 'list'`. Pe de altă parte, metoda încorporată `union()` acceptă drept argument orice obiect iterabil, îl va converti explicit într-un set, apoi va aplica operația de reuniune pe cele două seturi.

```
# reuniunrea unui set cu un iterabil
# utilizand metoda union() - funcioneaza
print('set1 reunit cu lista:', set1.union([1.3, 1.3, 100]))
# utilizand operatorul specific | - genereaza eroare
print('set1 reunit cu lista:', set1|[1.3, 1.3, 100])
OUTPUT:
set1 reunit cu lista: {1.3, 100, (3, 4), 'artificial', 'inginer'}
TypeError: unsupported operand type(s) for |: 'set' and 'list'
```

OBSERVAȚIE 2. Principiul prezentat anterior se aplică tuturor operațiilor care se pot efectua prin ambele metode: metodele vor accepta orice iterabil ca argument, convertindu-l intern la un set, în timp ce operatorii specifici funcționează doar cu seturi explicite drept operanzi.

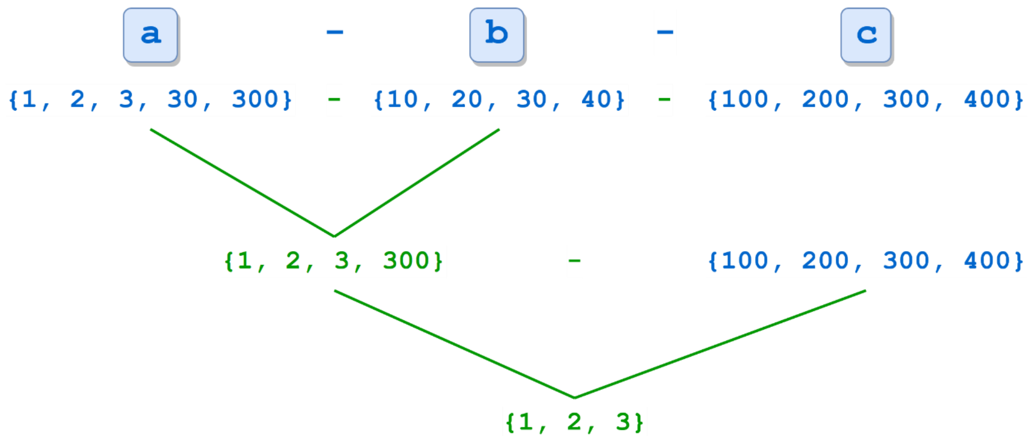
`met_set.2.` **Intersecția** seturilor, care în matematică este definită drept mulțimea ce conține doar elementele comune ale operanzilor mulțimii, se realizează în Python atât prin operandul ”&„ cât și prin metoda specifică `intersection()`. Sintaxa generală a metodei este `obiect_set.intersection(set1, set2, set3,...)` și returnează un nou set compus din elementele comune ale tuturor seturilor introduse ca argumente.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
# utilizand operatorul specific &
print('set1 intersectat cu set2:', set1 & set2)
# utilizand metoda specifica intersection()
print('set1 intersectat cu set2:', set1.intersection(set2))
OUTPUT:
set1 intersectat cu set2: {'inginer', 1.3}
set1 intersectat cu set2: {'inginer', 1.3}
```

`met_set.3.` **Diferența** seturilor, care imită diferența dintre două mulțimi din matematică – definită drept mulțimea elementelor care aparțin primei mulțimi, dar nu aparțin mulțimii a doua, se poate calcula în Python utilizând operandul scădere (–) sau metoda specifică `difference()`.

```
# se instantiaza doua seturi distincte
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
# utilizand operatorul specific -
print('set1 - set2:', set1-set2)
# utilizand metoda specifica difference()
print('set2 - set1:', set2.difference(set1))
OUTPUT:
set1 - set2: {'artificial', (3, 4)}
set2 - set1: {(2, 3), 'inteligenta'}
```

În cazul în care se dorește aplicarea operației de diferențiere pe mai mult de două seturi, metoda specifică nu se poate aplica deoarece acceptă un singur obiect ca argument. În schimb, operatorul scădere se poate utiliza cu succes și succesiunea operațiilor de diferență este de la stânga la dreapta: (`set1` – `set2` – `set3`) va calcula inițial diferența dintre `set1` și `set2`, iar din setul rezultat va fi scăzut `set3`.



Figură 18. Diferența a trei seturi (a, b și c)

`met_set.4.` **Diferența simetrică** a seturilor, definită în matematică drept (`set1 - set2`) reunit cu (`set2 - set1`) – elementele din `set1` și din `set2`, dar care nu se regăsesc în ambele (la intersecția lor). Această operație se poate integra prin intermediul seturilor Python utilizând operatorul specific ”^„ sau metoda specifică **`symetric_difference`**`()`, care returnează un nou set ce conține elementele rezultate în urma operației de diferențiere simetrică. Trebuie menționat faptul că metoda **`symetric_difference`**`()` acceptă drept argument un singur obiect tip set. În cazul în care se dorește aplicarea operației de diferențiere simetrică pe mai mult de două seturi, se poate utiliza operatorul specific: `set1^set2^... set_n`. Similar cu operația de diferențiere simplă, și diferențierea simetrică între mai mult de două seturi se realizează de la stânga la dreapta.

```
# se instantiaza trei seturi distincte
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
set3 = {1.3, 50, 100}
# utilizand metoda specifica symetric_difference()
print('set2^set1:', set2.symmetric_difference(set1))
# utilizand operatorul specific ^
print('set1^set2^set3:', set1 ^ set2 ^ set3)
OUTPUT:
set2^set1: {(3, 4), (2, 3), 'artificial', 'inteligenta'}
set1^set2^set3: {1.3, 100, (3, 4), (2, 3), 'artificial', 50, 'inteligenta'}
```

`met_set.5.` **Disjuncția** seturilor indică dacă două sau mai multe seturi au elemente comune sau nu. Această operație se realizează utilizând metoda specifică **`isdisjunct`**`()`, care returnează `True` (dacă nu există elemente comune între cele două seturi – seturile sunt disjuncte) sau `False` (dacă există elemente comune între cele două seturi). Mai mult, metoda acceptă drept argument un obiect set. Merită menționat că dacă metoda returnează `True`, este echivalent cu faptul că intersecția celor două seturi rezultă în mulțimea vidă – sau un obiect set gol în Python.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
set3 = {1, 50, 100}
print('set1 si set2 sunt disjuncte?', set1.isdisjoint(set2))
print('set1 si set3 sunt disjuncte?', set1.isdisjoint(set3))
print('intersectia dintre set1 si set3 este', set1.intersection(set3))
OUTPUT:
set1 si set2 sunt disjuncte? False
set1 si set3 sunt disjuncte? True
intersectia dintre set1 si set3 set()
```

OBSERVAȚIE. Nu există operator specific pentru operația de disjuncție a seturilor.

met_set.6. **Subseturile** sunt, conform teoriei seturilor, seturi care au toate elementele conținute în alt set. În Python, acest fapt se poate verifica utilizând operatorii specifici ”<=„ și ”<„ sau, mai intuitiv metoda specifică `issubset()`, care returnează `True` sau `False`, în funcție de valoarea de adevăr a expresiei ”`set1` este subset al `set2`”.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
set3 = {1.3, (3,4)}
# utilizand metoda issubset()
print('set2 este subset al set1?', set2.issubset(set1))
#utilizand operatorul specific <=
print('set3 este subset al set1?', set3 <= set1)
#utilizand operatorul specific <
print('set3 este subset al set2?', set3 < set2)
OUTPUT:
set2 este subset al set1? False
set3 este subset al set1? True
set3 este subset al set2? False
```

OBSERVAȚIE. Diferența dintre operatorii specifici ”<=„ și ”<„ este aceea că operatorul ”<„ verifică dacă un set este strict subset al unui alt set, adică are un număr mai mic de elemente, dar toate sunt conținute în setul mai mare. Cu alte cuvinte, aplicat aceluiași set, acesta va returna `False`. În comparație, `set1` ”<=„ `set1` va returnat `True`, adică toate elementele dintr-un set sunt conținute de el însuși.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
#utilizand operatorul specific <=
print('set1 este subset al set1?', set1 <= set1)
#utilizand operatorul specific <
print('set1 este strict subset al set1?', set1 < set1)
OUTPUT:
set1 este subset al set1? True
set1 este strict subset al set2? False
```

met_set.7. **Supersetul** este inversul subsetului și, în consecință, se poate determina cu unul dintre operatorii ”>=„ și ”>„ sau cu metoda specifică `issuperset()` – care returnează `True` sau `False`, în funcție de valoarea de adevăr a expresiei ”`set1` este supersetul `set2`”. Operatorii se utilizează analog cu analiza subseturilor.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
set3 = {1.3, (3,4)}
# utilizand metoda issuperset()
print('set1 este supersetul set2?', set1.issuperset(set2))
#utilizand operatorul specific >=
print('set1 este supersetul set3', set1 >= set3)
#utilizand operatorul specific >
print('set2 este supersetul set2?', set2 > set3)
OUTPUT:
set1 este supersetul set2? False
set1 este supersetul set3 True
set2 este supersetul set2? False
```

Este de menționat faptul că utilizarea operatorilor permite analiza a mai mult de două seturi simultan, în timp ce metoda specifică `issuperset()` acceptă drept argument un singur obiect de tip set.

MODIFICAREA SETURILOR

Deși trebuie să conțină doar obiecte unicate și imitabile, seturile în sine fac parte din categoria obiectelor mutabile, fiind posibilă modificarea lor local, după ce au fost create obiectele respective. Există integrate în sintaxa limbajului de programare Python o serie de operații și metode specifice care facilitează manipularea acestui tip de date.

`met_set.8.` **Update seturi prin reuniunea** acestora se poate realiza în două moduri: utilizând operatorul specific ”`|=`„ sau metoda `update()`, care acceptă orice număr de obiecte iterabile pe care le convertește la obiecte tip set și le intersectează cu setul pentru care se aplică metoda. Sintaxa generală este `obiect_set.update(set1, set2,...).` Având la bază operația de reuniune a seturilor, operația de update adaugă la `set1` elementele din `set2` cu condiția ca acestea să nu existe deja în `set1`.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
set3 = {100, (3,4)}
print('initial', set1)
# utilizand metoda update()
set1.update(set2)
print("update cu set 2", set1)
#utilizand operatorul specific |=
# set1 este deja updatat cu set2!
set1 |= set3
print('update cu set3', set1)
OUTPUT:
initial {'artificial', 1.3, (3, 4), 'inginer'}
update cu set 2 {1.3, (3, 4),'inginer',(2, 3),'artificial','inteligenta'}
update cu set3 {1.3, 100, (3, 4),'inginer',(2, 3),'artificial','inteligenta'}
```

OBSERVAȚIE. La fiecare operație de update se modifică obiectul local, însemnând că obiectul inițial este alterat. Drept urmare, la a doua operație de update, setul este deja modificat.

```
setA = {1, 3, 5, 7}
setB = {2, 4, 6, 8}
setC = {9, 10}
print('setul original:', setA)
# se adauga elementele din setB si setC simulatan
setA.update(setB, setC)
print('setA dupa update multipu', setA)
OUTPUT:
setul original: {1, 3, 5, 7}
setA dupa update multipu {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

`met_set.9.` **Update seturi prin intersecția** seturilor se poate realiza fie prin operatorul specific ”&=„ sau metoda `intersection_update()`, care acceptă drept argument un număr nespecificat de obiecte iterabile pe care le convertește intern la seturi și aplică operația de intersecție asupra lor – însemnând că setul inițial este alterat păstrând doar elementele pe care le are comune cu celelalte seturi, restul fiind eliminate. Dacă nu există elemente comune, metoda va returna un obiect set gol.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
set3 = {100, (3,4)}
print('initial', set1)
# utilizand metoda intersection_update()
set1.intersection_update(set2)
print("dupa update cu set 2", set1)
#utilizand operatorul specific &=
# set1 este deja updatat cu set2!
set1 &= set3
print('dupa update cu set3', set1)
OUTPUT:
initial {'artificial', 1.3, 'inginer', (3, 4)}
dupa update cu set 2 {1.3, 'inginer'}
dupa update cu set3 set()
```

```
setA = {1, 3, 5, 7}
setB = {2, 4, 6, 8}
setC = {9, 10}
print('setul original:', setA)
# se adauga elementele din setB si setC simulatan
setA.intersection_update(setB, setC)
print('setA dupa update multipu', setA)
OUTPUT:
setul original: {1, 3, 5, 7}
setA dupa update multipu set()
```

`met_set.10.` **Update seturi prin diferențiere simplă** se poate realiza prin operatorul specific ”-=„ sau cu metoda `difference_update()`, care acceptă drept argument un obiect iterabil pe care intern îl transformă într-un set și aplică diferența din setul principal, pe care îl alterează ștergând din el toate elementele din setul rezultat.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
set3 = {100, (3,4)}
print('initial', set1)
# utilizand metoda difference_update()
set1.difference_update(set2)
print("dupa update cu set 2", set1)
#utilizand operatorul specific -=
# set1 este deja updatat cu set2!
set1 -= set3
print('dupa update cu set3', set1)
OUTPUT:
initial {'artificial', 1.3, 'inginer', (3, 4)}
dupa update cu set 2 {'artificial', (3, 4)}
dupa update cu set3 {'artificial'}
```

`met_set.11.` **Update seturi prin diferențiere simetrică** se poate realiza prin operatorul specific ”^=„ sau cu metoda `symmetric_difference_update()`, care acceptă drept argument un obiect iterabil pe care intern îl transformă într-un set și aplică operația de diferență simetrică din setul principal, pe care îl alterează păstrând elementele din setul principal sau din cel rezultat, dar nu din ambele.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
set2 = {'inteligenta', 1.3, (2, 3), 'inginer'}
set3 = {100, (3,4)}
print('initial', set1)
# utilizand metoda symmetric_difference_update()
set1.symmetric_difference_update(set2)
print("dupa update cu set 2", set1)
#utilizand operatorul specific ^=
# set1 este deja updatat cu set2!
set1 ^= set3
print('dupa update cu set3', set1)
OUTPUT:
initial {'artificial', 1.3, 'inginer', (3, 4)}
dupa update cu set 2 {'inteligenta', (3, 4), 'artificial', (2, 3)}
dupa update cu set3 {'inteligenta', 100, 'artificial', (2, 3)}
```

`met_set.12.` **Update seturi prin adăugarea** unui singur element la setul inițial se realizează prin metoda `add()`, care acceptă drept argument un obiect imutabil pe care îl adaugă (aceasta fiind diferența fundamentală între `update()` și `add()`). Trebuie menționat că, datorită faptului că seturile nu acceptă dubluri de elemente, în situația în care se încearcă adăugarea unui element deja existent, acesta nu se adaugă, setul inițial rămânând nealterat. În cazul în care se încearcă folosirea metodei cu un argument care nu este imutabil, se va genera o eroare de tip `TypeError: unhashable type: 'list'`. În fragmentul următor de cod sunt prezentate toate aceste aspecte.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
print('initial', set1)
# se altereaza setul cu metoda add()
set1.add(89)
print("dupa update", set1)
set1.add(1.3)
print("dupa update", set1)
set1.add([1, 3])
print("dupa update", set1)
OUTPUT:
initial {'artificial', 1.3, 'inginer', (3, 4)}
dupa update {1.3, (3, 4), 'inginer', 'artificial', 89}
dupa update {1.3, (3, 4), 'inginer', 'artificial', 89}

TypeError: unhashable type: 'list'
```

`met_set.13.` **Update seturi prin ștergerea** unui singur element din setul inițial se realizează prin metoda `remove()`, care acceptă drept argument un element din set pe care în elimină. Dacă elementul nu există în set, metoda va genera o eroare de tip `KeyError:` .

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
print('initial', set1)
# se altereaza setul cu metoda remove()
set1.remove((3,4))
print("dupa update", set1)
set1.remove((3,4))
print("dupa update", set1)
OUTPUT:
initial {'artificial', 1.3, 'inginer', (3, 4)}
dupa update {'artificial', 1.3, 'inginer'}

KeyError: (3, 4)
```

`met_set.14.` **Update seturi prin ștergerea** unui singur element din setul inițial se poate realiza și utilizând metoda `discard()`, care acceptă drept argument un element din set pe care îl elimină. Spre deosebire de `remove()`, această metodă nu generează nicio eroare în cazul în care elementul nu se găsește în set, ci returnează setul nealterat.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
print('initial', set1)
# se altereaza setul cu metoda remove()
set1.discard((3,4))
print("dupa update", set1)
set1.discard((3,4))
print("dupa update", set1)
OUTPUT:
initial {'artificial', 1.3, 'inginer', (3, 4)}
dupa update {'artificial', 1.3, 'inginer'}
dupa update {'artificial', 1.3, 'inginer'}
```

met_set.15. Update seturi prin ștergerea unui element aleatoriu din setul inițial realizează utilizând metoda pop(), care nu acceptă niciun argument dar returnează elementul șters din set. Dacă setul este gol metoda va genera o eroare de tipul: KeyError: 'pop from an empty set'. Este asemănătoare cu metoda cu același nume din capitolul listelor: met_list.8(). În exemplul următor de cod este creat un set, se iterează prin el folosind numărul de elemente generat de funcțiile f50() și f37(), și se elimină elemente aleatorii utilizând funcția pop(). La sfârșitul iterațiilor, când se încearcă ștergerea unui alt element, metoda va genera eroarea.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
for element in range(len(set1)):
    pop_element = set1.pop()
    print('elementul sters:', pop_element)

pop_element = set1.pop()
OUTPUT:
elementul sters: artificial
elementul sters: 1.3
elementul sters: inginer
elementul sters: (3, 4)

KeyError: 'pop from an empty set'
```

met_set.16. Update seturi prin ștergerea tuturor elementelor dintr-un set se realizează utilizând metoda clear() care nu acceptă niciun argument.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
for element in range(len(set1)):
    pop_element = set1.pop()
    print('elementul sters:', pop_element)

pop_element = set1.pop()
OUTPUT:
elementul sters: artificial
elementul sters: 1.3
elementul sters: inginer
elementul sters: (3, 4)

KeyError: 'pop from an empty set'
```

met_set.17. Copierea unui set se realizează utilizând metoda copy() care nu acceptă niciun argument, dar returnează o copie superficială a setului analizat. Dacă se realizează o modificare asupra copiei, obiectul original nu este alterat.

```
set1 = {'inginer', 1.3, (3, 4), 'artificial'}
print('Numarul de elemente din set1:', len(set1))
set2 = set1.copy()
print('Numarul de elemente din set2:', len(set2))
set2.clear()
print('Numarul de elemente din set1:', len(set1))
print('Numarul de elemente din set2:', len(set2))
OUTPUT:
Numarul de elemente din set1: 4
Numarul de elemente din set2: 4
Numarul de elemente din set1: 4
Numarul de elemente din set2: 0
```

5. Tipuri de date proprii – introducere în programarea obiectuală (clase Python)

Așa cum a fost prezentat succint în subcapitolul § *3. Particularități de sintaxă a limbajului de programare Python – CUVINTELE CHEIE* (mai specific la `kw.5`), acesta, fiind un limbaj de programare orientat pe obiecte (POO), există posibilitatea ca programatorul să își creeze propriile tipuri de date care să mimeze funcționalitatea tipurilor de date încorporate. Prin intermediul claselor se pot crea noi tipuri de date cărora li se pot atribui proprietăți/**atribute** și funcționalități/**metode**. Simplist, o clasă se poate asimila cu un constructor de obiecte, un prototip sau un șablon care modelează obiectele create. Mai specific, o clasă reprezintă ***o încapsulare a datelor simple și comportamentelor***. Pentru cazul unui fluid care are o serie de proprietăți (temperatură, viscozitate, presiune etc.) și funcționalități (curge), fluidul este clasa și aerul este obiectul rezultat din clasa fluid.

Pe lângă faptul că oferă o flexibilitate mai mare în utilizarea datelor, clasele prezintă o serie de avantaje pe care datele încorporate nu le au:

↔ `moștenirea` – se pot transfera proprietăți și funcționalități de la o clasă superioară la una care le preia și le utilizează, modifică etc.;
↔ `compunerea` – reprezintă o colecție de componente care conlucrează și care au un scop precis și bine definit;
↔ `instanțiere` – se pot inițializa oricât de multe obiecte după crearea unei clase;
↔ `instanțiere` – se pot inițializa oricât de multe obiecte după crearea unei clase;
↔ `customizare` – prin mecanismul de `moștenire`, clasele se pot personaliza prin extinderea funcționalității clasei superioare;
↔ `supraîncărcare` – se pot rescrie operațiile clasice pentru a funcționa cu noile obiecte.

5.1. Folosirea claselor

Fiind un limbaj de programare de utilitate generală, Python acceptă atât tipul de programare procedurală (bazată pe folosirea funcțiilor pentru a rezolva problemele date) cât și tipul orientat pe obiecte prin care se creează noi tipuri de date. Acest fapt poate duce la confuzie în rândul programatorilor începători sau al nespecialiștilor. În esență, clasele reprezintă o modalitate elegantă de scriere și organizare a codului și prezintă câteva avantaje pe care programarea procedurală nu le oferă:

a. permit modelarea și rezolvarea problemelor complexe din realitate prin faptul că fiecare clasă poate mapa un obiect real – spre exemplu, un fluid, un perete, un echipament termic, electric etc. Astfel, pot fi rezolvați algoritmi cu complexitate crescută;

b. permit reutilizarea codului și evitarea repetitivității prin posibilitatea particularizării fiecărui obiect creat în parte. Mai mult, se pot defini ierarhii ale claselor și se pot interconecta între ele prin operațiile de moștenire sau extindere. În general, modulele externe sunt bazate pe colecții de clase interconectate;

c. permit încapsularea datelor (atributelor) și comportamentelor (metodelor) într-o singură entitate (clasă) și utilizarea în program prin intermediul instanțelor (obiectelor). Acest lucru oferă o mai mare flexibilitate și siguranță în programare;

d. eficientizează scrierea codului prin abstractizarea detaliilor implementării informațiilor conținute de clasă, permițând, totodată, și procesarea datelor complexe. Acest lucru face posibilă crearea claselor sub forma unor interfețe mai user-friendly (API – *Application Programming Interface*).

Cu toate aceste avantaje, nu este întotdeauna indicată utilizarea claselor în detrimentul funcțiilor clasice. În practică, este de preferat a se evita utilizarea claselor în diverse cazuri, cum ar fi simpla stocare a unor date sau atunci când clasele sunt mult prea simple (au doar o metodă, spre exemplu). De asemenea, este indicat să nu se utilizeze clase pentru a încapsula funcționalități deja disponibile prin funcțiile încorporate sau prin clase importate din diverse module terțe (spre exemplu, modulul care modelează rețelele neuronale artificiale – *tensorflow*). Mai mult, utilizarea claselor are tendința de a spori complexitatea codului în sine – anumiți programatori le evită – și există aplicații întregi extrem de complexe care au la bază doar programarea structurală. Introducerea claselor în aceste cazuri nu este deloc indicată.

5.2. Definirea unei clase – cuvântul cheie `class`

O clasă se definește foarte simplu specificând cuvântul cheie `class` urmat de identificatorul dorit pentru clasa respectivă. După cum a mai fost amintit, convenția pentru denumirea claselor Python este de a folosi stilul `CamelCase` (fiecare cuvânt care compune identificatorul este scris cu majusculă). Instanțierea (crearea obiectelor propriu-zise din clasa creată) obiectelor se realizează puțin diferit față de cazul în care se utilizează un tip de dată încorporat și: trebuie specificată clasa din care se creează obiectul respectiv. Sintaxa generală este de forma:

```
class NumeClasa(ClasaBaza1, ClasaBaza2,...ClasaBazaN):

    atribut = valoare

    def metoda_clasa(self):
        self.atribut = valoare
```

OBSERVAȚIE. Corpul clasei se comportă ca un domeniu separat de domeniul general al denumirilor variabilelor Python – `namespace`. În consecință, identificatorii atributelor și metodelor sunt valabili doar în interiorul clasei și pot fi accesați doar prin intermediul obiectelor create și al clasei în sine.

Se cuvine menționat faptul că o clasă se poate defini fără utilizarea parantezelor ce preced numele dacă aceasta nu moștenește funcționalități de la nicio clasă superioară (de bază). Mai mult, o clasă poate conține doar atribute, doar metode, ambele sau niciun element – clasă goală. Dar este obligatoriu de a utiliza parantezele drepte în momentul în care se instanțiază obiectul! În exemplul următor este creată o clasă goală și instanțiat un obiect simplu, analizându-se și afișând tipul acestuia prin utilizarea funcției încorporate `f61()`.

```
class FluidMonofazic:
    pass

apa = FluidMonofazic()
print('Tipul obiectului apa', type(apa))
OUTPUT:
Tipul obiectului apa <class '__main__.FluidMonofazic'>
```

ATRIBUTELE

Atributele se definesc oriunde în interiorul clasei (se numesc atribute de clasă) și se pot accesa prin intermediul obiectului prin utilizarea notației cu punct: `obiect.nume_atribut`. Trebuie specificat încă de la început faptul că atributele pot fi atât legate de clasă, numindu-se atribute de clasă, cât și legate de instanță în sine – atribute de instanțe.

În fragmentul următor de cod clasa denumită `FluidMonofazic` are definite două atribute de clasă de tip număr întreg. Acestea sunt definite direct în corpul clasei, în afara oricărei metode și sunt apoi accesate de obiectul `apa` și afișate pe ecranul consolei.

```
class FluidMonofazic:
    temperatura = 10 # temperatura in grade Celsius
    densitate = 990 # densitatea in kg/m^3

apa = FluidMonofazic()
print('apa are temperatura', apa.temperatura, 'grade Celsius')
print(f'apa are densitatea {apa.densitate} kg/mc')
OUTPUT:
apa are temperatura 10 grade Celsius
apa are densitatea 990 kg/mc
```

Odată accesate, aceste atribute se pot modifica fără niciun fel de restricție privind valoarea sau tipul de dată utilizat, dar această modificare va fi specifică **doar** obiectului prin care s-a realizat modificarea. În interiorul clasei atributul va avea aceeași valoare și la crearea unui nou obiect, acestuia îi va fi atribuită valoarea inițială. Mai mult, se pot chiar și adăuga noi atribute utilizând același procedeu: `obiect.atribut_nou = valoare`. Din nou, atributul acesta va fi legat doar de obiectul creat în sine și nu va fi specific întregii clase.

```
class FluidMonofazic:
    temperatura = 10 # temperatura in grade Celsius
    densitate = 990 # densitatea in kg/m^3

apa_50 = FluidMonofazic()
print('initial, apa_50 are temperatura', apa_50.temperatura, 'grade Celsius')
# se modifica valoarea atributului temperatura
apa_50.temperatura = 50
print('apa_50 are acum temperatura', apa_50.temperatura, 'grade Celsius')
# se instantiaza un nou obiect
apa_10 = FluidMonofazic()
print('noul obiect:')
print('apa_10 are temperatura', apa_10.temperatura, 'grade Celsius')
OUTPUT:
initial, apa_50 are temperatura 10 grade Celsius
apa_50 are acum temperatura 50 grade Celsius
noul obiect:
apa_10 are temperatura 10 grade Celsius
```

În majoritatea cazurilor practice la crearea claselor se declară o metodă constructor specială denumită `__init__(self)` și care are declarat cel puțin parametrul special `self`. Acest parametru are rolul de a lega obiectul în sine (încă necreat) de atributele sau metodele respective, fiind, în linii mari, referință la obiectul creat. Se amintește faptul că orice atribut sau metodă care are în componență la definire cuvântul cheie `self` este specific obiectelor, și nu claselor! În cazul în care nu se specifică explicit metoda constructor `__init__(self)` este folosită o variantă implicită care se creează automat la utilizarea cuvântului cheie `class`. În exemplul următor clasa definită anterior este alterată și, deși sunt utilizați aceiași parametri, aceștia sunt declarați prin intermediul metodei constructor.

OBSERVAȚIE. Parametrul `self` este denumit așa printr-o convenție, el putând avea practic orice nume, iar programul funcționând la fel. Dar pentru consistență, este indicat a se utiliza denumirea convențională. Ce este în schimb obligatoriu, este ca acest parametru să fie primul!

```
class FluidMonofazic:
    def __init__(self, temperatura, densitate):
        self.temperatura = temperatura
        self.dens = densitate
```

OBSERVAȚIE. A se observa modul în care sunt apoi definite atributele: acestea pot avea același nume ca parametrul metodei `__init__()` – cazul atributului `temperatura` sau alt nume – cazul atributului `dens`. E de menționat că este de preferat prima variantă.

Dacă se utilizează metoda constructor și se definesc atributele prin intermediul acesteia, instanțierea obiectelor se va face obligatoriu specificând valori pentru atribute prin intermediul unor argumente introduse în momentul instanțierii. Dacă sunt omise argumentele, interpretorul Python va genera o eroare de tipul: `TypeError: FluidMonofazic.__init__() missing 2 required positional arguments: 'temperatura' and 'densitate'`. În fragmentul de cod următor este prezentată o modalitate de definire a parametrilor metodei constructor `__init__()` prin care se specifică și tipul de dată necesar instanțierii, și modalitatea corectă de instanțiere a unui obiect cu specificarea atributelor. Trebuie menționat faptul că specificarea explicită a tipului de dată utilizată la instanțiere are un rol pur și simplu informativ, clasa acceptând orice obiect la

instanțierea unui obiect (datorită caracterului dinamic al limbajului de programare Python). În exemplul următor clasa este inițializată cu doi parametri (exceptând parametrul obligatoriu `self`), specificând de fiecare dată tipul `float`. Cu toate acestea, la instanțierea obiectului `aer` se utilizează un argument de tip `string` și interpretorul Python nu generează nicio eroare.

```
class FluidMonofazic:
    # se defineste metoda constructor
    def __init__(self, temperatura:float, densitate:float):
        self.temperatura = temperatura
        self.dens = densitate

aer = FluidMonofazic('123', 123)
print(type(aer.temperatura))
print(type(aer.dens))
OUTPUT:
<class 'str'>
<class 'int'>
```

OBSERVAȚIE. Chiar dacă în metoda constructor `__init__()` parametrii au fost denumiți `temperatura` și `densitate`, la apelarea acestora prin intermediul obiectelor se utilizează denumirea specificată în conjuncție cu cuvântul cheie `self` – `self.dens` în cazul discutat (`aer.dens` – `self` devine `aer` după instanțiere!). De aceea, este indicat a se folosi aceeași denumire pentru parametrii metodei și atributele considerate.

Atributele de clasă sunt variabile definite prin intermediul corpului clasei, datele din acesta fiind disponibile atât pentru clasa în sine cât și pentru toate instanțele create, care vor împărți aceste atribute cu clasa în sine. Drept urmare, o modificare internă adusă acestor atribute se va regăsi și în instanțele create. Spre exemplu, dacă se implementează un contor intern care va ține seama de instanțele create din clasă, aceasta se va face sub forma unui atribut de clasă care va fi incrementat de fiecare dată când este apelată metoda constructor. Iată un fragment de cod în care este prezentat un astfel de exemplu.

```
class FluidMonofazic:
    # se defineste un contor
    numar_instante_fluid = 0
    def __init__(self, temperatura:float, densitate:float):
        self.temperatura = temperatura
        self.dens = densitate
        # se apleaza intern atributul de clasa si se incrementeaza
        FluidMonofazic.numar_instante_fluid += 1

apa = FluidMonofazic(10, 990)
print(f'Momentan au fost create {apa.numar_instante_fluid} obiecte')
aer = FluidMonofazic(0, 1.292)
print((f'Momentan au fost create {aer.numar_instante_fluid} obiecte'))
OUTPUT:
Momentan au fost create 1 obiecte
Momentan au fost create 2 obiecte
```

Merită observat faptul că apelarea atributului de clasă se realizează prin intermediul clasei în sine: `FluidMonofazic.numar_instante_fluid += 1`. O altă variantă este utilizarea funcției

încorporate `type()`, care, aplicată unui obiect specific prin parametrul `self` returnează tipul clasei respective. Astfel, sintaxa `type(self).numar_instante_fluid += 1` realizează același lucru și este o alternativă de preferat la prima deoarece este mai robust și evită codarea explicită a clasei. Mai mult, se poate observa faptul că variabila este apelabilă atât prin clasa în sine cât și prin instanțele create.

În contrast cu atributele de clasă, atributele de instanță reprezintă variabilele definite în interiorul unei metode (cel mai indicat este a se folosi metoda de instanțiere `__init__()`) și care sunt legate de obiect în sine. Acestea sunt disponibile doar în cadrul instanțelor create din clasă, caracterizând doar starea instanței, nu și a clasei. În exemplele anterioare `self.temperatura` și `self.dens = densitate` sunt atribute de instanță – a se observa faptul că sunt legate de instanță prin `self`. Mai mult, utilizarea lor în cadrul clasei se realizează tot specificând instanța curentă, ca în cazul metodei definite în continuare: `((self.temperatura_1 + self.temperatura_2)/2`.

Spre deosebire de atributele de clasă, atributele de instanță nu pot fi accesate prin intermediul clasei în sine, interpretorul Python generând o eroare de tipul `AttributeError: type object 'FluidMonofazic' has no attribute 'tempeatura_1'`.

METODELE

Pe lângă atribute, se pot implementa alte metode în cadrul metodei constructor `__init__()`, care, în mod similar, vor fi apelate și vor rula la instanțierea unui nou obiect, putând chiar altera încă de la început. Însă, în cel mai des întâlnite cazuri metodele sunt create în afara metodei constructor și sunt definite ca funcții, utilizând cuvântul cheie `def` și minimum un parametru – același `self` care referențiază obiectul propriu-zis (metode de instanță). Pentru a apela funcția, se utilizează aceeași procedură ca în cazul atributelor: `obiect.metodă()`. Diferența constă în parantezele ce preced numele clasei, tocmai pentru că se apelează o funcție. În esență, o metodă este utilizată pentru a simula comportamentul (a modela o acțiune) unui obiect. În exemplul următor este creată o metodă care calculează temperatura medie a unui fluid supus procesului de încălzire dacă se cunosc temperaturile inițială și finală.

```
class FluidMonofazic:
    # se defineste metoda constructor
    def __init__(self, temperatura_1, temperatura_2):
        self.temperatura_1 = temperatura_1
        self.temperatura_2 = temperatura_2

    # se defineste o metoda care va calcula media aritmetica a atributelor
    def temperatura_medie(self):
        return round((self.temperatura_1 + self.temperatura_2)/2, 2)

aer = FluidMonofazic(10, 85)
temperatura_medie = aer.temperatura_medie()
print(f'La intrare fluidul are are temperatura {aer.temperatura_1} grade C')
print(f'La iesire fluidul are are temperatura {aer.temperatura_2} grade C')
print('Media temperaturilor fluidului este', temperatura_medie, 'grade C')
OUTPUT:
La intrare fluidul are are temperatura 10 grade C
La iesire fluidul are are temperatura 85 grade C
Media temperaturilor fluidului este 47.5 grade C
```

OBSERVAȚIE. Chiar dacă au aceeași denumire, variabila `temperatura_medie` și metoda `temperatura_medie()` sunt tratate diferit de către interpretorul Python.

În Python, există trei tipuri de metode:

↔ metode de instanță (care au primul argument `self`) – cel mai des utilizate;

↔ metode de clasă (care au primul argument `cls`);

↔ metode statice (care nu sunt nici de instanță, nici de clasă).

Metodele se pot defini și cu parametri proprii, care vor fi indicați doar în momentul apelării și nu la instanțierea obiectului în sine. Acești parametri se utilizează pentru definirea funcționalității metodei respective și pot fi declarați inclusiv cu valori implicite, așa cum este cazul metodei care determină fluxul termic al unui fluid pe baza temperaturilor (definite anterior drept atribute), a debitului masic și căldurii specifice. Definirea și apelarea metodelor respectă toate regulile utilizate până acum la definirea și apelarea funcțiilor sau metodelor altor obiecte Python. Trebuie menționat faptul că metoda constructor funcționează și cu valori implicite ale parametrilor, aceștia declarându-se similar cu exemplul anterior.

```
class FluidMonofazic:
    # se defineste metoda constructor cu parametri impliciti
    def __init__(self, temperatura_1=0, temperatura_2=0):
        self.temperatura_1 = temperatura_1
        self.temperatura_2 = temperatura_2

    # se defineste o metoda care va calcula media aritmetica a atributelor
    def temperatura_medie(self):
        return round((self.temperatura_1 + self.temperatura_2)/2, 2)
    # se defineste o metoda care va calcula fluxtul termic primit de fluid
    def calcul_flux(self, debit_masic, c_sp=1000):
        return debit_masic*c_sp*(self.temperatura_2-self.temperatura_1)/1000

# se instantiaza un obiect cu cele 2 atribute obligatorii
apa = FluidMonofazic(10, 50)
# se calculeaza fluxul termic utilizand valoarea implicita a parametrului
# in acest caz nu se mai specifica argumentul respectiv
flux = apa.calcul_flux(4.7)
print(f'Pentru a se incalzi de la {apa.temperatura_1} la {apa.temperatura_2}
apa primeste', end=' ')
print(f'{flux} kW termici.')
# se calculeaza fluxul termic utilizand alta valoare pentur caldura specifica
flux = apa.calcul_flux(4.7, 4800)
print(f'Pentru a se incalzi de la {apa.temperatura_1} la {apa.temperatura_2}
apa primeste', end=' ')
print(f'{flux} kW termici.')
OUTPUT:
Pentru a se incalzi de la 10 la 50 apa primeste 188.0 kW termici.
Pentru a se incalzi de la 10 la 50 apa primeste 902.4 kW termici.
```

Ca și în cazul atributelor, se pot crea și metode specifice obiectelor în sine, acestea nefiind legate de clasă ci definite pe obiect, dar această tehnică nu este indicată și o bună practică este

de a defini toate metodele în interiorul clasei. Cu toate acestea, se pot lega funcții de clase în sine și acestea vor putea fi utilizate. În exemplul următor se definește o clasă, și o funcție separat, urmând ca aceasta să fie legată de clasă prin operația de atribuire. Merită observat faptul că nu este neapărat nevoie ca numele metodei să fie identic cu cel al funcției. În exemplul următor funcția a fost denumită `calcul_flux()`, iar metoda de utilizat în clasă a fost denumită `calculeaza_fluxul_termic`. La apelarea ei în cadrul instanței trebuie avut grijă să se folosească cea de-a doua denumire: `apa.calculeaza_fluxul_termic(4.7)`, în caz contrar interpretorul Python generând o eroare de tipul `AttributeError: 'FluidMonofazic' object has no attribute 'calcul_flux'`. Ca o bună practică, se sugerează ca de fiecare dată când este posibil să se păstreze o constanță între denumiri!

```
# se creeaza clasa cu metoda __init__ și temperatura_medie()
class FluidMonofazic:
    # se defineste metoda constructor
    def __init__(self, temperatura_1, temperatura_2):
        self.temperatura_1 = temperatura_1
        self.temperatura_2 = temperatura_2

    # se defineste o metoda care va calcula media aritmetica a atributelor
    def temperatura_medie(self):
        return round((self.temperatura_1 + self.temperatura_2)/2, 2)

# se creeaza o functie separata care se va lega de clasa
def calcul_flux(self, debit_masic, caldura_specifica=1000):
        return debit_masic*caldura_specifica*(self.temperatura_2-
self.temperatura_1)/1000

# se leaga funcia de clasa, ea devenind o metoda a clasei
FluidMonofazic.calculeaza_fluxul_termic = calcul_flux
apa = FluidMonofazic(10, 50)

flux = apa.calculeaza_fluxul_termic(4.7)
print(f'Pentru a se incalzi de la {apa.temperatura_1} la {apa.temperatura_2}
apa primeste', end=' ')
print(f'{flux} kW termici.')
OUTPUT:
Pentru a se incalzi de la 10 la 50 apa primeste 188.0 kW termici.
```

OBSERVAȚIE. Toate atributele și metodele unei clase Python sunt stocate într-un dicționar de instanță denumit `__dict__`. Acesta a fost automat generat și populat cu metode și atribute implicite în momentul creării clasei. Dacă acestea sunt suprascrise, ele se vor modifica și în dicționar. Pentru a le afișa, se poate utiliza funcția încorporată `print()`. În exemplul următor sunt afișate toate datele din dicționarul clasei `FluidMonofazic`.

```
for item in FluidMonofazic.__dict__:
     print(item)
OUTPUT:
__module__
__init__
temperatura_medie
__dict__
__weakref__
__doc__
calculeaza_fluxul_termic
```

METODE SPECIALE

Sintaxa Python suportă așa-numitele **metode speciale**, denumite și metode **dunder (**de la **d**ouble **under**score) sau **magice**. Aceste metode sunt de cele mai multe ori metode de instanță (necesită specificarea parametrului `self`) și sunt parte fundamentală a mecanismului funcțional al claselor Python – nu necesită definirea lor specifică, acestea fiind create la utilizarea cuvântului cheie `class`, dar interpretorul Python le apelează automat ca răspuns la o operație specifică. Aceste metode speciale se pot diferenția de restul prin utilizarea a două simboluri underscore (__) la începutul și sfârșitul indicatorilor. Două exemple utilizate până acum au fost metodele `__init__()` și `__dict__()`, dar mai există o serie de astfel de metode, printre care `__str__()` și `__repr__()` cu rolul de a reprezenta obiectul creat sub forma unui string Python. Între ele există o diferență majoră: `__str__()` oferă reprezentarea informală a obiectului, în timp ce `__repr__()` asigură reprezentarea formală a acestuia. Cu alte cuvinte, `__str__()` returnează un șir de caractere care reprezintă obiectul într-o manieră lizibilă pentru programator/utilizator, iar `__repr__()` returnează un string care trebuie să permită recrearea obiectului analizat și are un caracter pur de depanare, fiind specific programatorilor. În exemplul din fragmentul următor de cod este utilizată metoda `__str__()` pentru a reprezenta o descriere lizibilă a obiectului tip `FluidMonofazic`.

```
class FluidMonofazic:
    # se defineste metoda constructor
    def __init__(self, nume, temperatura_1, temperatura_2):
        self.nume = nume
        self.temperatura_1 = temperatura_1
        self.temperatura_2 = temperatura_2

    def __str__(self) -> str:
        return f'Fluidul "{self.nume}" are temperatura la intrare
{self.temperatura_1} C si la iesire {self.temperatura_2} C.'

aer = FluidMonofazic('aer', 10, 50)
print(aer)
OUTPUT:
Fluidul "aer" are temperatura la intrare 10 C si la iesire 50 C.
```

Merită observată diferența între crearea unei metode speciale `__str__()` proprii și utilizarea metodei presetare existente la declararea clasei. Dacă în primul caz rezultatul utilizării funcției încorporate `print(aer)` returnează rezultatul din codul anterior (`Fluidul "aer" are`

`temperatura la intrare 10 C si la iesire 50 C`). Mai mult, același rezultat se obține și utilizând funcția încorporată `str(aer)`. Dacă în schimb se utilizează metoda nealterată, `print(aer)` va returna `<__main__.FluidMonofazic object at 0x000002AC74602C50>`.

Pentru mai multe detalii despre metodele speciale se poate consulta link-ul sau link-ul.

5.3. Moștenirea claselor

Un mecanism foarte utilizat în extinderea utilizabilității claselor este acela de **moștenire** care permite încorporarea funcționalității unei/unor clase deja existente în interiorul unei clase noi. Această caracteristică specifică programării obiectuale constă în crearea relațiilor ierarhice între clase, unde o clasă *copil* (subclasă sau clasă derivată) **moștenește** atributele și metodele de la una sau mai multe clase *părinte* (superclasă sau clasă de bază). Sintaxa pentru moștenire este simplă și necesită doar specificarea clasei/claselor părinte între paranteze în momentul definirii clasei copil, ca în exemplul următor.

```
class SubClasa(SuperClasa1, SuperClasa2,..., SuperClasaN):
```

Moștenirea permite reutilizarea codului scris anterior (chiar existent în librării/module externe, după importarea lor în codul sursă curent) și reduce repetarea părților deja dezvoltate. Dezvoltarea în continuare a subclasei se va realiza în mod normal, definind atribute și metode proprii, care vor fi adiționale față de caracteristicile moștenite din superclasă. În exemplul următor se creează o clasă părinte denumită `Fluid` cu o metodă pentru determinarea temperaturii medii a obiectelor generate. Adițional, o subclasă denumită `FluidMonofazic` va moșteni caracteristicile clasei principale și va avea o metodă proprie pentru determinarea vitezei de curgere în funcție de doi parametri: debit volumic și secțiunea de curgere.

```
class Fluid():
    def __init__(self, denumire:str, temp_in:float, temp_out:float) -> None:
        self.denumire = denumire
        self.temp_in = temp_in
        self.temp_out = temp_out

    def calculeaza_temp_medie(self) -> float:
        return f'Temperatura medie este {(self.temp_in + self.temp_out)/2} C'

class FluidMonofazic(Fluid):
        def __init__(self,denumire:str,temp_in:float,temp_out:float) -> None:
            super().__init__(denumire, temp_in, temp_out)

        def calculeaza_viteza(self, debit: float, sectiune: float) -> float:
             return f'Viteza de curgere este {round(debit/sectiune, 3)} m/s'

apa = FluidMonofazic('apa', 38, 75)
print(apa.calculeaza_temp_medie())
print(apa.calculeaza_viteza(13, 3.4))
OUTPUT:
Temperatura medie este 56.5 C
Viteza de curgere este 3.824 m/s
```

După cum se poate observa în exemplul anterior, chiar dacă nu a fost definită în cadrul clasei `FluidMonofazic`, metoda `calculeaza_temp_medie()` poate fi apelată de obiectul creat din această clasă datorită mecanismului de moștenire. Este de observat modul în care se realizează scrierea metodei `__init__()` pentru care trebuie să se specifice legătura cu clasa superioară prin apelarea funcției `super()` – aceasta indică faptul că anumite atribute sunt preluate din clasa superioară (`Fluid` în cazul de față). Mai mult, pe lângă acestea, se pot adăuga și atribute specifice subclasei – așa cum este prezentat cu atributul `viteza` în fragmentul următor de cod.

```
class Fluid():
    def __init__(self, denumire: str, temp_in: float, tem_out: float) -> None:
        self.tip = None
        self.denumire = denumire
        self.temp_in = temp_in
        self.temp_out = temp_out

    def calculeaza_temp_medie(self) -> float:
        return f'Temperatura medie este {(self.temp_in + self.temp_out)/2} C'

class FluidMonofazic(Fluid):
        # atributele mostenite de la subclasa
        def __init__(self, denumire, temp_in, temp_out, viteaza) -> None:
            super().__init__(denumire, temp_in, temp_out)
            # atributul specific sublcasei
            self.viteza = viteaza

        def __str__(self):
            return f'{self.denumire} curge cu viteza {self.viteza} m/s.'

apa = FluidMonofazic('apa', 38, 75, 10)
aer = FluidMonofazic('aer', 10, 50, 3.5)
print(apa.calculeaza_temp_medie())
print(apa)
print(aer.viteza)
OUTPUT:
Temperatura medie este 56.5 C
apa curge cu viteza 10 m/s.
3.5
```

În mecanismul de moștenire al claselor se poate inclusiv suprascrie sau extinde funcționalitatea din superclasă prin intermediul metodelor subclasei. În momentul în care se apelează o metodă sau un atribut, interpretorul Python începe să caute în clasa curentă (subclasa în cazul de față), apoi în superclase în ordinea în care acestea sunt trecute la dezvoltarea subclasei, până când găsește o potrivire de nume, apoi execută instrucțiunea. În cazul în care atributul/metoda căutat/ă nu este găsit/ă în domeniul niciunei clase, interpretorul Python generează o eroare de tipul `AttributeError: 'FluidMonofazic' object has no attribute 'calculeaza_temp_me'`. În fragmentul următor de cod este prezentat un asemenea caz – este adăugată în subclasă o metodă denumită tot `calculeaza_temp_medie()` care va calcula temperatura medie a fluidului, dar va returna rezultatul în grade Kelvin, adăugând 273.15 la rezultat. Spre deosebire de primul

exemplu, la apelarea acestei metode prin intermediul obiectului se va apela metoda din subclasă, aceasta având prioritate. Acesta este un exemplu de suprascriere a metodei din superclasă.

```
class Fluid():
    def __init__(self, denumire: str, temp_in: float, temp_out: float) -> 
None:
        self.tip = None
        self.denumire = denumire
        self.temp_in = temp_in
        self.temp_out = temp_out

    def calculeaza_temp_medie(self) -> float:
        return f'Temperatura medie este {(self.temp_in + self.temp_out)/2} C'

class FluidMonofazic(Fluid):
        # atributele mostenite de la subclasa
        def __init__(self, denumire, temp_in, temp_out) -> None:
            super().__init__(denumire, temp_in, temp_out)

        def calculeaza_temp_medie(self):
             return (self.temp_in + self.temp_out)/2 + 273.15

aer = FluidMonofazic('aer', 10, 75)
print(f'{aer.denumire} cu temp la intrare {aer.temp_in} C si temp la iesire 
{aer.temp_out} C', end=' ')
print(f'are temp medie {aer.calculeaza_temp_medie()} K')
OUTPUT:
aer cu temp la intrare 10 C si temp la iesire 75 C are temp medie 315.65 K
```

Există posibilitatea de moștenire multiplă, atunci când o subclasă moștenește de la mai mult de o superclasă. În acest caz apare riscul de a exista una sau mai multe metode cu aceeași funcționalitate în superclase, putând crea disfuncționalități ale codului. Pentru a ține evidența asupra ordinii în care se vor căuta metode specifice, interpretorul Python are integrat un algoritm intern denumit **ordinea de rezoluție a metodelor** (MRO – *Method Resolution Order*). Acesta determină modul în care interpretorul determină ce metodă să apeleze din ierarhia claselor. Pentru vizualizarea ordinii se poate apela metoda încorporată `.__mro__` ca în exemplul următor. Metoda va returna un obiect de tip tuplu, care va conține clasele în ordinea în care se caută metoda specificată pentru apelare. Spre exemplu, instrucțiunea `FluidBifazic.__mro__` va returna, în ipoteza în care această subclasă va moșteni de la două superclase `Lichid` și `Solid`, următorul rezultat: `(<class '__main__.FluidBifazic'>, <class '__main__.Lichid'>, <class '__main__.Solid'>, <class 'object'>)` indicând faptul că inițial se analizează subclasa (`FluidBifazic`), apoi superclasele de la stânga la dreapta, în ordinea în care au fost specificate la crearea subclasei: `FluidBifazic(Lichid, Solid)`. Căutarea metodei se termină cu analiza clasei încorporate denumite `object`.

În plus, funcționalitatea metodelor din superclasă se poate extinde utilizând funcția `super()` pentru apelarea acesteia și adăugarea noilor elemente. Un exemplu este prezentat în fragmentul următor unde se calculează temperatura medie în grade Kelvin. După crearea instanței denumite

aer specificând denumirea sub forma unui șir de caractere și temperatura la intrare și ieșire, se apelează metoda calculeaza_temp_medie(). Instanțierea făcându-se din subclasă, inițial interpretorul Python va căuta aici metoda și va executa instrucțiunile care o compun. Pe următoarea linie este apelarea metodei cu același nume din clasa superioară utilizând funcția super().calculeaza_temp_medie(). În acest punct, interpretorul Python se va muta către superclasa de unde se moștenește metoda, o va executa (în acest caz, va returna valoarea medie a temperaturii fluidului) și se va întoarce în corpul subclasei pentru a executa următoarea instrucțiune (adăugarea 273.15 K). Rezultatul este în final returnat și se continuă algoritmul. În acest mod, funcționalitatea metodei este extinsă, și nu suprascrisă ca în cazul anterior.

```
class Fluid():
    def __init__(self, denumire, temp_in, temp_out) -> None:
        self.tip = None
        self.denumire = denumire
        self.temp_in = temp_in
        self.temp_out = temp_out

    def calculeaza_temp_medie(self) -> float:
        return (self.temp_in + self.temp_out)/2

class FluidMonofazic(Fluid):
        # atributele mostenite de la subclasa
        def __init__(self, denumire, temp_in, temp_out) -> None:
            super().__init__(denumire, temp_in, temp_out)

        def calculeaza_temp_medie(self):
            return super().calculeaza_temp_medie() + 275.15

aer = FluidMonofazic('aer', 10, 75)
print(f'{aer.denumire} cu temp la intrare {aer.temp_in} si temp la iesire
{aer.temp_out}', end='.')
print(f' Are temperatura medie {aer.calculeaza_temp_medie()} K')
OUTPUT:
aer cu temp la intrare 10 C si temp la iesire 75 C are temp medie 315.65 K
```

SURSE SUPLIMENTARE

1. Mark Lutz, *Learning Python*, O'Reilly, 2013
2. Adriana Stan, *Introducere în Python folosind Google COLAB*, UNTPRESS, 2022
3. Christian Myers, *Python One-Liners*, No Starch Press, 2020
4. Eric Matthes, *Python Crash Course*, 2nd Edition, No Starch Press, 2019
5. Aditya Y. Bhargava, *Grokking Algorithms*, Manning Publications, 2016
6. site-uri web
    a. https://realpython.com/
    b. https://www.geeksforgeeks.org/
    c. https://www.w3schools.com/python/
    d. https://www.coursera.org/

BIBLIOGRAFIE

Badea, A. (2005). *Bazele Transferului de Căldură și Masă.* București: Editura Academiei Române.

Davenport, T. (2012). *Harvard Business Review*. Preluat pe Septembrie 06, 2023, de pe hbr.org: https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century

docs.python.org. (2023). *Python tuturial*. Preluat pe September 19, 2023, de pe docs.pyton: https://docs.python.org/3.11/tutorial/index.html

Parks, D. M. (2017). *Defining Data Science and Data Scientist.* Florida: University of South Florida.

Python Succes Stories. (2023). Preluat de pe Python: https://www.python.org/about/success/

VanderPlas, J. (2017). *Python Data Science Handbook* (ed. First). O'Reilly Media, Inc.

View publication stats