test-e.ro

CAIET ELEV

Lecția 19

Subprograme — rol, caracteristici, concepte de bază

*Un subprogram este o bucată de program căreia îi dăm un **nume** și care rezolvă o **subproblemă** bine delimitată: se scrie o singură dată, cu **def**, și se apelează de câte ori e nevoie, pentru valori diferite.*

DISCIPLINA Informatică **CLASA** a IX-a · mate-info **LIMBAJ** Python 3

NUMELE ELEVULUI .....

CUM FOLOSEȘTI CAIETUL

Rezolvă exercițiile pe hârtie: urmărești execuția pas cu pas, recunoști componentele unui subprogram și scrii singur soluțiile. Noutatea lecției este `def` — definirea unui subprogram — împreună cu **parametrii** și cu `return`. Rămân la dispoziție `while`, `for ... in range()`, `for x in lista`, `if/elif/else`, `%`, `//`, `ord()` / `chr()` și listele (indexare, `len`, metodele listei). Regula de aur a lecției: **tot ce intră în subprogram intră prin parametri, tot ce iese, iese prin** `return`.

CUPRINS

1 Ce știm deja: numere, cifre și liste .....
2 De ce avem nevoie de subprograme .....
3 Anatomia: antet, corp, apel .....
4 Parametri și returnarea rezultatelor .....
5 Variabile locale și globale .....
6 Avantajele modularizării .....
7 Cod ghidat · șase probleme .....

SECȚIUNEA 1 · REACTUALIZARE: CIFRE, LISTE, PAȘI

1 Ce știm deja: numere, cifre și *liste*

Înainte de lecția nouă ne asigurăm că avem la îndemână instrumentele: parcurgerea cifrelor unui număr cu `while`, parcurgerea unei liste cu indici și numărarea pașilor unei bucle.

Primul instrument este parcurgerea cifrelor. Ultima cifră se obține cu `n % 10`, iar restul numărului cu `n // 10`. Repetând cele două operații într-un `while`, ajungem la fiecare cifră, de la dreapta spre stânga, până când numărul devine 0.

```
n = 907
while n > 0:
    c = n % 10
    n = n // 10
    print(c)
```

Al doilea instrument este parcurgerea unei liste. `len(v)` ne dă numărul de elemente, `v[i]` ne dă elementul de pe poziția `i`, iar indicii încep de la 0 și merg până la `len(v) - 1`. Când nu ne interesează poziția, ci doar valorile, scriem direct `for x in v:`.

```
v = [4, 7, 10, 3]
i = 0
while i < len(v):
    print(v[i])
    i = i + 1
```

ATENȚIE

Parcurgerea cifrelor **distruge** numărul: la ieșirea din buclă, `n` a ajuns 0. Dacă mai ai nevoie de valoarea inițială, o salvezi într-o a doua variabilă înainte de buclă. Ține minte capcana aceasta — la finalul lecției vei vedea că există o cale prin care problema dispare de la sine.

DE REȚINUT

Numărul de pași al unei bucle e o măsură a costului. Bucla `while d <= n:` face `n` pași, în timp ce `while d * d <= n:` face doar aproximativ √n pași — pentru `n = 100`, zece pași în loc de o sută.

A

EXERCIȚIUL A · GRILĂ

Instrumentele din lecțiile trecute

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Pentru lista `v = [4, 7, 10, 3]`, ce valoare are `len(v)`?

   `3 — indicele ultimului element` `4 — numărul de elemente` `24 — suma valorilor` `10 — cel mai mare element`

2. Pentru `v = [2, 5, 8]`, ce afișează `print(v[1])`? `2` `8` `5` `eroare`

3. De câte ori se execută corpul buclei de mai jos, pentru `n = 100`?

```
n = 100
d = 1
while d * d <= n:
    d = d + 1
```

   `de 100 de ori` `de 50 de ori` `de 10 ori` `de 9 ori`

4. De ce parcurgem cifrele unui număr cu `while`, și nu cu `for`?

   `pentru că nu știm dinainte câte cifre are numărul`

   `pentru că` `for` `nu merge cu numere întregi`

   `pentru că un număr are întotdeauna zece cifre` `pentru că` `% 10` `dă numărul de cifre`

HINT *`len(v)` numără elementele, nu indicii, iar indicii încep de la 0. La întrebarea 3, bucla merge cât timp `d * d <= 100`.*

B

EXERCIȚIUL B · DRY-RUN

Câte elemente impare are lista

DRY-RUN

Urmărește bucla pas cu pas. Coloanele `i` și `v[i]` îți sunt date; tu completezi dacă elementul este impar (scrie DA sau NU) și valoarea contorului `c` după acel pas. Primul rând e completat ca exemplu. Pe ultimul rând, unde condiția buclei a devenit falsă, scrie o liniuță.

```
v = [4, 7, 10, 3]
c = 0
i = 0
while i < len(v):
    if v[i] % 2 != 0:
```

```
6           c = c + 1
7       i = i + 1
8   print(c)
```

i (dat)	v[i] (dat)	v[i] este impar?	c după acest pas
0	4	NU	0
1	7		
2	10		
3	3		
4	–	–	

→ Ce afișează programul? _____

HINT *contorul `c` crește doar când `v[i] % 2 != 0`; altfel rămâne cu valoarea de pe rândul anterior.*

SPRIJIN · SCRIE CODUL

Suma cifrelor unui număr

SCRIERE COD

Se citește un număr natural `n` (`n >= 0`). Afișează suma cifrelor lui `n`. Ex.: pentru `n = 1234` se afișează `10`, pentru că 1 + 2 + 3 + 4 = 10.

HINT *pentru `n = 0` bucla nu pornește deloc, iar suma rămâne 0 — exact rezultatul corect.*

SOLUȚIA TA · PYTHON

SPRIJIN · SCRIE CODUL

Câte elemente pare are lista

SCRIERE COD

Se citește un număr natural `n` (`n >= 1`), apoi `n` numere naturale, fiecare pe câte o linie. Afișează câte dintre ele sunt pare. Ex.: pentru `n = 4` și numerele 4, 7, 10, 3 se afișează `2`.

HINT *zero este număr par, deci intră la numărătoare.*

SOLUȚIA TA · PYTHON

2 SECȚIUNEA 2 · PRIMA DEFINIȚIE CU DEF

De ce avem nevoie de *subprograme*

Când aceeași prelucrare apare de trei ori într-un program, o scriem de trei ori. Programul funcționează — dar devine greu de citit și, mai ales, greu de corectat.

Să presupunem că vrem suma cifrelor pentru trei numere citite. Fără instrumente noi, singura variantă e să copiem bucla de trei ori, schimbând doar numele variabilelor. Programul are peste douăzeci de linii și, dacă descoperim o greșeală în algoritm, trebuie să o corectăm în trei locuri. Uităm unul — și programul dă rezultate corecte în două cazuri din trei, ceea ce e mai rău decât o eroare vizibilă.

SUBPROGRAMUL

Un subprogram este o secvență de instrucțiuni căreia îi dăm un **nume** și care rezolvă o **subproblemă** bine delimitată: primește niște valori, execută niște instrucțiuni, trimite înapoi un rezultat. Îl scriem o singură dată și îl folosim de câte ori avem nevoie, pentru valori diferite.

```
1  def suma_cifrelor(n):
2      s = 0
3      while n > 0:
4          s = s + n % 10
5          n = n // 10
6      return s
8  print(suma_cifrelor(1234))
9  print(suma_cifrelor(907))
10 print(suma_cifrelor(2025))
```

Algoritmul apare o singură dată. Cele trei linii de la final nu mai spun *cum* se calculează suma cifrelor, ci doar *că* se calculează — iar aceasta e exact ce vrem să citim când urmărim logica programului.

DE REȚINUT

Un subprogram bun rezolvă **o singură** subproblemă, iar numele lui spune limpede care. `suma_cifrelor` e un nume bun. `calcul` nu e.

ATENȚIE

Definiția unui subprogram nu îl execută. Scrisă singură, secvența care începe cu `def` nu afișează nimic și nu calculează nimic — e doar o rețetă pusă deoparte. Ea intră în acțiune abia când e chemată prin nume.

A

EXERCIȚIUL A · GRILĂ

Rolul unui subprogram

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Care este rolul principal al unui subprogram într-un program?

să memoreze mai multe valori deodată, ca o listă

să grupeze sub un nume o secvență care rezolvă o subproblemă

să facă programul să se execute de mai multe ori

să înlocuiască `while` când nu știm numărul de pași

2.

Un program calculează suma cifrelor pentru trei numere, repetând aceeași buclă de trei ori. Care e dezavantajul cel mai concret?

Python refuză să execute aceeași buclă de mai multe ori

rezultatele sunt corecte doar pentru primul număr

o corectură trebuie făcută în trei locuri, iar unul poate fi uitat

programul cere calculatoare cu multă memorie

3. Ce înseamnă că un subprogram este „reutilizabil"?

poate fi chemat de oricâte ori, din locuri diferite, pentru valori diferite

se șterge din memorie după prima folosire

își schimbă singur instrucțiunile la fiecare apel

poate fi scris o singură dată pe zi

HINT *duplicarea nu strică programul imediat, dar îl face fragil: o corectură trebuie aplicată în toate copiile.*

B

EXERCIȚIUL B · COMPLETARE

Definiția unui subprogram

COMPLETARE

Completează golurile folosind cuvintele din rezervă (două cuvinte rămân nefolosite).

`indice` `rezultat` `tabel` `nume` `subproblemă`

Un subprogram grupează sub un _____ o secvență de instrucțiuni care rezolvă o _____. Programul principal îi trimite valori concrete și primește înapoi un _____, fără să fie nevoie să știe cum a fost obținut.

HINT *cele două cuvinte nefolosite vin din lucrul cu liste, nu din definiția unui subprogram.*

C

EXERCIȚIUL C · SCRIE CODUL

Folosește subprogramul dat

SCRIERE COD

Copiază subprogramul `mesaj_cifre` de mai jos, exact așa cum e scris. Sub el, citește două numere naturale `a` și `b` (`>= 1`), fiecare pe câte o linie, apoi cheamă subprogramul pentru fiecare dintre ele, în ordinea citirii. Ex.: pentru `a = 1234` și `b = 7` se afișează `4` pe primul rând și `1` pe al doilea.

```python
def mesaj_cifre(n):
    c = 0
    while n > 0:
        c = c + 1
        n = n // 10
    print(c)
```

HINT *subprogramul dat nu are* `return`*: își face treaba afișând direct rezultatul. Cele două apeluri se scriu* **neindentate** *, la marginea din stânga — ele fac parte din programul principal, nu din definiție.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 3 · ANTET, CORP, APEL

3 Anatomia unui *subprogram*

Orice subprogram Python are aceeași formă, iar formele se recunosc după indentare. Trei nume acoperă tot: antetul, corpul și apelul.

ANTETUL · CORPUL · APELUL

Antetul este prima linie a definiției. Începe cu cuvântul cheie `def`, urmat de numele subprogramului, de o pereche de paranteze rotunde care conține lista **parametrilor**, și se încheie obligatoriu cu `:`. **Corpul** cuprinde toate instrucțiunile scrise sub antet, deplasate la dreapta cu patru spații; se termină acolo unde indentarea revine la marginea din stânga. **Apelul** este instrucțiunea prin care programul principal cere execuția: numele subprogramului, urmat de paranteze rotunde cu valorile concrete. Apelul **nu** face parte din definiție și nu este indentat.

```
def numar_cifre(n):        # antet
    c = 0                  # corp
    while n > 0:           # corp
        c = c + 1          # corp
        n = n // 10         # corp
    return c               # corp: trimite rezultatul
print(numar_cifre(2025))   # apel
```

Aceeași definiție, **în pseudocod** — convențiile pe care le folosim la analiză și proiectare au și ele o formă pentru subprograme:

```
subprogram numar_cifre(n) returnează c
  c ← 0
  cât timp n > 0 execută
    c ← c + 1
    n ← [n / 10]
  sfârșit cât timp
sfârșit subprogram
Apel numar_cifre(2025)
```

Antetul, corpul și apelul sunt aceleași trei componente: se schimbă doar notația. Pseudocodul îl folosim când **proiectăm**, Python când **implementăm**.

Ordinea în care se execută instrucțiunile nu mai este, de acum, ordinea în care sunt scrise. La întâlnirea unui apel, programul principal se **oprește**, sare în corpul subprogramului, îl execută până la `return`, ia valoarea primită și **revine exact în punctul de unde a plecat**, continuând calculul început.

ATENȚIE

Un subprogram poate fi apelat numai după ce definiția lui a fost **executată** — adică după ce Python a trecut peste linia cu `def`. Practic: scrii toate definițiile la începutul fișierului, iar programul principal sub ele. Dacă apelezi un nume pe care Python nu-l cunoaște încă, programul pornește, dar se oprește chiar în momentul apelului cu un `NameError` — o **eroare de executare**, nu una de sintaxă.

SUBPROGRAM, FUNCȚIE, PROCEDURĂ

„Subprogram" este termenul din programa școlară, „funcție" cel din documentația Python. Distincția utilă: subprogramele care **returnează un rezultat** (`numar_cifre`) se numesc funcții, iar cele care doar **execută o acțiune**, fără să trimită înapoi un rezultat util (`mesaj_cifre`), se numesc proceduri. În Python amândouă se scriu la fel — diferența o face prezența instrucțiunii `return`.

A

EXERCIȚIUL A · DRY-RUN

Ce se întâmplă la fiecare apel

DRY-RUN

Bucla cheamă subprogramul `dublu` la fiecare pas. Coloana `i` îți este dată; tu completezi valoarea returnată de apelul `dublu(i)` și valoarea variabilei `s` după acel pas. Primul rând e completat ca exemplu. Pe ultimul rând, unde bucla s-a oprit și apelul nu se mai face, scrie o liniuță.

```
1  def dublu(x):
2      return x + x
3  n = 3
5  i = 1
6  s = 0
7  while i <= n:
8      s = s + dublu(i)
9      i = i + 1
10 print(s)
```

i (dat)	dublu(i) returnează	s după acest pas
1	2	2
2		
3		
4	–	

→ Ce afișează programul? _____

HINT *la fiecare pas programul principal se oprește, cheamă* `dublu(i)`*, primește înapoi* `i + i` *și adaugă valoarea la* `s`*.*

B

EXERCIȚIUL B · ASOCIERE

Recunoaște componentele

ASOCIERE

Programul este format din patru fragmente. Scrie în căsuță litera descrierii potrivite pentru fiecare fragment. Fiecare literă se folosește o singură dată.

☐	`def numar_cifre(n):`
☐	`c = c + 1`
☐	`return c`
☐	`print(numar_cifre(2025))`

A Cere execuția subprogramului, cu o valoare concretă

B O instrucțiune de calcul din interiorul definiției, care nu încheie subprogramul

C Prima linie a definiției: numele și lista de valori primite

D Trimite rezultatul înapoi și oprește imediat subprogramul

HINT *un subprogram se recunoaște după formă: prima linie începe cu* `def`*, instrucțiunile de sub ea sunt deplasate la dreapta, iar apelul stă la marginea din stânga.*

C

EXERCIȚIUL C · SCRIE CODUL

Scrie corpul, antetul îl primești

SCRIERE COD

Scrie un subprogram cu antetul exact `def ultima_cifra(n):`, al cărui corp returnează ultima cifră a numărului primit. Apoi citește un număr natural `n` (`n >= 0`) și afișează ultima lui cifră, folosind subprogramul. Ex.: pentru `n = 1234` se afișează `4`.

HINT *corpul are o singură instrucțiune. Un subprogram nu trebuie să fie lung ca să fie util — trebuie doar să aibă o treabă clară. Antetul se scrie exact ca în cerință, cu două puncte la final, iar corpul se indentează cu patru spații.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 4 · INTRAREA ȘI IEȘIREA

4 Parametri și returnarea *rezultatelor*

Un subprogram devine cu adevărat util atunci când primește date din afară și trimite un rezultat înapoi. Cele două drumuri au fiecare numele lui: **parametrii** la intrare, `return` la ieșire.

Parametrul este numele scris între parantezele din antet. La apel, el primește valoarea trimisă de programul principal. Pentru **numere**, parametrul se comportă ca o copie proprie a valorii: orice modificare făcută asupra lui în subprogram nu conduce la modificarea variabilei din exterior. Un subprogram poate avea oricâți parametri, despărțiți prin virgulă; **ordinea contează** — prima valoare de la apel ajunge la primul parametru.

```
1  def cmmdc(a, b):
2      while b != 0:
3          r = a % b
4          a = b
5          b = r
6      return a
8  print(cmmdc(48, 18))
9  print(cmmdc(18, 48))
```

INSTRUCȚIUNEA RETURN

Face două lucruri deodată: trimite valoarea înapoi la locul apelului **și** oprește imediat subprogramul. Instrucțiunile scrise după `return`, pe același nivel, nu se mai execută niciodată. Un subprogram poate conține mai multe `return` — se execută doar primul la care ajunge.

Apelul unui subprogram care are `return` este o **expresie**: are o valoare și poate fi folosit oriunde e permisă o valoare — într-o atribuire (`s = suma_cifrelor(n)`), într-o condiție (`if este_prim(n) == 1:`), într-un calcul (`dublu(3) + dublu(4)`) sau direct într-un `print`.

> **ATENȚIE**
>
> Un subprogram fără `return` își execută corpul, dar înapoi trimite o valoare specială, `None`, al cărei înțeles este chiar „niciun rezultat util". `x = mesaj_cifre(1234)` nu dă eroare, însă `x` nu conține numărul de cifre, ci `None`. De aceea, când vrei un **rezultat**, folosești `return`, nu `print`.

> **ATENȚIE · REGULA SE APLICĂ LA NUMERE**
>
> „Ce se schimbă înăuntru rămâne înăuntru" este adevărat pentru valorile numerice, pe care le folosim în toată lecția. Dacă însă îi trimitem unui subprogram o **listă** și îi adăugăm un element cu `append`, modificarea se vede și în programul principal: lista nu se copiază, se împrumută.

A

EXERCIȚIUL A · GRILĂ

Parametri și return

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Ce face instrucțiunea `return` într-un subprogram?

   - `afișează valoarea pe ecran, ca print`
   - `trimite o valoare înapoi la locul apelului și oprește imediat subprogramul`
   - `reia execuția corpului de la prima instrucțiune`
   - `salvează valoarea într-o variabilă globală`

2. Pentru subprogramul de mai jos, ce afișează `print(dublu(3) + dublu(4))`?

```
1  def dublu(x):
2      return x + x
```

   - `7`
   - `24`
   - `14`
   - `nimic — un apel nu poate intra într-o expresie`

3. Ce se întâmplă dacă un subprogram nu conține deloc instrucțiunea `return`?

   - `Python semnalează o eroare de sintaxă înainte de rulare`
   - `returnează automat ultima variabilă modificată în corp`
   - `returnează automat valoarea zero`
   - `corpul se execută, iar înapoi se trimite valoarea specială None`

HINT *la întrebarea 2, apelul este o expresie:* `dublu(3)` *devine 6,* `dublu(4)` *devine 8.*

B

EXERCIȚIUL B · SCRIE CODUL

Produsul cifrelor, cu subprogram

SCRIERE COD

Scrie un subprogram cu antetul exact `def produs_cifre(n):`, care returnează produsul cifrelor numărului primit. Citește apoi un număr natural `n` (`n >= 0`) și afișează pe același rând numărul citit, un spațiu, apoi produsul cifrelor lui. Ex.: pentru `n = 1234` se afișează `1234 24`; pentru `n = 907` se afișează `907 0`.

HINT *acumulatorul pornește de la 1, elementul neutru al înmulțirii. Atenție la* `n = 0`*: bucla* `while n > 0` *nu se execută niciodată, iar subprogramul ar returna 1 — cazul se tratează separat, printr-o verificare explicită înainte de buclă. Rezultatul se trimite înapoi cu* `return`*, nu se afișează din interiorul subprogramului.*

SOLUȚIA TA · PYTHON

★

PERFORMANȚĂ · SCRIE CODUL

Suma cifrelor în orice bază

SCRIERE COD

Scrie un subprogram cu antetul exact `def suma_cifrelor_baza(n, b):`, care returnează suma cifrelor numărului `n` scris în baza `b`. Se citesc `n` (`n >= 0`) și `b` (`2 <= b <= 10`), fiecare pe câte o linie. Ex.: pentru `n = 13` și `b = 2` se afișează `3`, pentru că 13 se scrie 1101 în baza 2.

HINT *nu trebuie să construiești numărul în baza* `b`*. Cifrele lui în baza* `b` *sunt chiar resturile împărțirilor repetate la* `b` *— exact ca la baza 10, unde împărțeai la 10.*

SOLUȚIA TA · PYTHON

5 SECȚIUNEA 5 · SPAȚIUL DE LUCRU PROPRIU

Variabile *locale* și variabile *globale*

Un subprogram are propriul lui spațiu de lucru. Variabilele create acolo apar la apel și dispar la revenire, fără să lase urme. De aici vine cea mai importantă calitate a unui subprogram: **independența**.

VARIABILA LOCALĂ · VARIABILA GLOBALĂ

Orice variabilă creată prin atribuire în corpul unui subprogram — inclusiv parametrii — este **locală**. Există doar cât timp se execută subprogramul și dispare complet la revenire; programul principal nu o poate vedea, nici înainte, nici după. O variabilă creată în programul principal este **globală**: dacă un subprogram folosește un nume pe care nu l-a primit ca parametru și căruia nu îi atribuie nimic, Python îl caută în programul principal și îi **citește** valoarea de acolo — dar nu o poate modifica.

```
1  n = 5
2
```

```
def patrat(n):
    n = n * n
    return n
print(patrat(3))
print(n)
```

Se afișează `9`, apoi `5`. Parametrul `n` a primit valoarea 3, a devenit 9 și a dispărut. Variabila `n` din programul principal a rămas 5.

ATENȚIE · NU CONFUNDA GLOBALA CU LOCALA

Într-un subprogram, un nume este ori global (numai citit), ori local (atribuit) — **pe tot corpul**, nu doar de la atribuire încolo. Dacă îl citești la început și îi atribui ceva mai jos, Python îl consideră local încă de la prima linie și se oprește cu eroarea `UnboundLocalError`. Regula sigură: tot ce intră în subprogram intră prin **parametri**, tot ce iese, iese prin `return`.

ATENȚIE

Din această cauză, un subprogram **nu poate** trimite un rezultat înapoi modificând o variabilă din exterior. Singura cale corectă este `return`, iar programul principal salvează valoarea primită într-o variabilă a lui: `y = calcul(x)`.

A

EXERCIȚIUL A · GRILĂ

Cine vede ce

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Ce afișează secvența de mai jos?

```
n = 7
def patrat(n):
    n = n * n
    return n
print(patrat(5), n)
```

`25 25` `25 7` `49 7` `5 7`

2. O variabilă creată prin atribuire în interiorul unui subprogram este:

`globală: poate fi citită oriunde în programul principal`

`un parametru suplimentar, care trebuie trecut și în antet`

`locală: există doar cât timp se execută subprogramul`

o constantă: valoarea ei nu mai poate fi schimbată

3. Cum trimite un subprogram un rezultat înapoi, corect și sigur?

prin `return`, iar programul principal salvează valoarea într-o variabilă proprie

prin atribuirea unei variabile cu același nume în corp

prin `print`, pentru că afișarea înseamnă și transmitere

scriind numele variabilei pe ultima linie a corpului

HINT *parametrul este o variabilă locală: primește valoarea de la apel, se schimbă în corp și dispare la revenire.*

B

EXERCIȚIUL B · DRY-RUN

Ce vede subprogramul și ce rămâne afară

DRY-RUN

La fiecare pas al buclei se face un apel. Coloana `i` îți este dată; tu completezi valoarea returnată de `patrat(i)` și valoarea variabilei `n` din programul principal, după apel. Primul rând e completat ca exemplu. Pe ultimul rând, unde apelul nu se mai face, scrie o liniuță în prima coloană de completat.

```
1  n = 5
2  def patrat(n):
3      n = n * n
4      return n
5  i = 1
6  while i <= 3:
7      print(patrat(i), n)
8      i = i + 1
```

i (dat)	patrat(i) returnează	n din programul principal
1	1	5
2		
3		
4	–	

HINT *în corpul subprogramului, `n` este parametrul, adică o variabilă locală: ce i se întâmplă acolo nu iese afară. Coloana din dreapta are aceeași valoare pe toate rândurile.*

C

EXERCIȚIUL C · SCRIE CODUL

Apelul nu strică nimic

SCRIERE COD

Se citesc două numere naturale `a` și `b` (`>= 1`), fiecare pe câte o linie. Scrie un subprogram cu antetul exact `def cmmdc(a, b):`, care returnează cel mai mare divizor comun al valorilor primite. Afișează pe primul rând cel mai mare divizor comun, iar pe al doilea rând valorile lui `a` și `b`, despărțite printr-un spațiu. Ex.: pentru 48 și 18 se afișează `6`, apoi `48 18`.

HINT *al doilea rând este dovada: deși parametrii se numesc tot `a` și `b`, iar bucla din corp îi modifică pe amândoi, variabilele din programul principal au rămas cu valorile citite. Schimbul se face cu o variabilă ajutătoare: `r = a % b`, apoi `a = b`, apoi `b = r`.*

SOLUȚIA TA · PYTHON

6 SECȚIUNEA 6 · ȘI PREȚUL EI

Avantajele *modularizării*

A rezolva o problemă **modular** înseamnă a o descompune în subprobleme mici, a scrie câte un subprogram pentru fiecare și a lăsa programul principal să le apeleze în ordinea potrivită. Câștigurile sunt trei, și merită numite separat.

LIZIBILITATE · REUTILIZARE · DEPANARE UȘOARĂ

Lizibilitatea. Programul principal spune *ce* se face, nu *cum*. Linia `if este_prim(x) == 1:` se înțelege dintr-o privire. **Reutilizarea.** Algoritmul se scrie o singură dată: pentru trei numere faci trei apeluri, pentru o mie faci o mie — codul rămâne la fel de scurt. **Depanarea ușoară.** Când rezultatul final e greșit, testezi fiecare subprogram separat, pe valori mici pentru care știi răspunsul.

```
1  def este_prim(n):
2      if n < 2:
3          return 0
4      d = 2
5      while d * d <= n:
6          if n % d == 0:
7              return 0
8          d = d + 1
9      return 1
10 print(este_prim(2), este_prim(9), este_prim(97))
```

Subprogramul de mai sus conține trei instrucțiuni `return`, iar aceasta îl face și mai clar: de îndată ce a găsit un divizor, răspunsul e cunoscut și nu mai are rost să continue.

ATENȚIE · ȘI MODULARIZAREA ARE UN PREȚ

Modularizarea nu înseamnă „cât mai multe subprograme". Un subprogram chemat o singură dată și fără un nume care să însemne ceva nu ajută pe nimeni: fragmentează programul și obligă cititorul să sară dintr-un loc în altul. Regula practică: dacă poți spune într-un cuvânt sau două ce face, merită să existe. Există și un al doilea preț, mai mic: fiecare apel costă puțin timp — pentru un subprogram chemat de un milion de ori într-o buclă, diferența devine măsurabilă.

A

EXERCIȚIUL A · ORDONARE

Etapele rezolvării modulare

ORDONARE

Numerotează pașii de la 1 la 5, în ordinea în care se procedează când rezolvi o problemă prin descompunere în subprograme. Pașii îți apar amestecați.

- [ ] Scriem programul principal, care citește datele și apelează subprogramele în ordinea potrivită.
- [ ] Citim enunțul și stabilim exact ce trebuie afișat la final.
- [ ] Testăm programul întreg și, dacă un rezultat e greșit, corectăm doar subprogramul vinovat.

- [ ] Descompunem problema în subprobleme mici, fiecare cu un rezultat clar.
- [ ] Scriem câte un subprogram pentru fiecare subproblemă și îl testăm separat, pe valori mici.

HINT *ordinea nu e întâmplătoare: descompunerea vine înaintea scrierii, iar testarea fiecărui subprogram vine înaintea testării întregului program.*

B

EXERCIȚIUL B · SCRIE CODUL

Același subprogram, trei apeluri

SCRIERE COD

Scrie un subprogram cu antetul exact `def este_prim(n):`, care returnează 1 dacă `n` este număr prim și 0 în caz contrar. Se citesc trei numere naturale, fiecare pe câte o linie. Afișează, pe câte o linie pentru fiecare număr în ordinea citirii, numărul citit, un spațiu, apoi DA dacă este prim sau NU în caz contrar. Ex.: pentru 2, 9 și 0 se afișează `2 DA`, `9 NU`, `0 NU`.

HINT *numerele 0 și 1 nu sunt prime, iar 2 este singurul număr prim par. Tratează cazul `n < 2` la începutul corpului, altfel bucla nu pornește și subprogramul le declară prime.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 7 · COD GHIDAT

7 Scrie cod care *trece testele*

Șase probleme. Scrie soluția pe hârtie, urmărește-o mental pe exemplul dat și abia apoi tastează-o pe platformă, unde e verificată automat pe 5 cazuri de test.

1

PROBLEMA 7.1 · COD GHIDAT

Suma cifrelor pentru trei numere

SCRIERE COD

Scrie un subprogram cu antetul exact `def suma_cifrelor(n):`, care returnează suma cifrelor numărului primit. Se citesc trei numere naturale, fiecare pe câte o linie. Afișează, pe câte o linie, suma cifrelor fiecărui număr, în ordinea citirii. Ex.: pentru 1234, 907 și 0 se afișează `10`, `16`, `0`.

HINT *algoritmul se scrie o singură dată, iar programul principal face trei apeluri. Dacă te trezești copiind bucla, oprește-te: exact aceasta încearcă lecția să elimine.*

SOLUȚIA TA · PYTHON

2

PROBLEMA 7.2 · REPARĂ CODUL

Numărul de cifre se oprește prea devreme

SCRIERE COD

Programul de mai jos ar trebui să citească un număr natural `n` (`n >= 1`) și să afișeze câte cifre are. El afișează însă 1 pentru orice număr citit: pentru `n = 1234` afișează 1 în loc de 4. Găsește greșeala, corectează-o și scrie programul reparat.

```
1  def numar_cifre(n):
2      c = 0
3      while n > 0:
4          c = c + 1
5          n = n // 10
6          return c
7
8  n = int(input())
9  print(numar_cifre(n))
```

HINT *uită-te la indentare, nu la formule:* `return` *face două lucruri deodată, iar unul dintre ele explică de ce bucla nu apucă să se termine niciodată.*

PROGRAMUL CORECTAT · PYTHON

3

PROBLEMA 7.3 · COD GHIDAT

Câte numere prime are lista

SCRIERE COD

Scrie un subprogram cu antetul exact `def este_prim(n):`, care returnează 1 dacă `n` este număr prim și 0 în caz contrar. Se citește un număr natural `n` (`n >= 1`), apoi `n` numere naturale, fiecare pe câte o linie. Afișează câte dintre numerele citite sunt prime. Ex.: pentru `n = 5` și numerele 2, 3, 4, 5, 6 se afișează `3`.

HINT *testarea primalității se face cu `while d * d <= n:`, nu cu `while d < n:` — pentru liste lungi, diferența de număr de pași este cea discutată la lecția trecută.*

SOLUȚIA TA · PYTHON

★

PROBLEMA 7.4 · PROVOCARE

Cel mai mare divizor comun al unei liste

SCRIERE COD

Scrie un subprogram cu antetul exact `def cmmdc(a, b):`, care returnează cel mai mare divizor comun al celor două numere primite, folosind algoritmul lui Euclid cu împărțiri. Se citește un număr natural `n` (`n >= 1`), apoi `n` numere naturale, dintre care cel puțin unul diferit de zero. Afișează cel mai mare divizor comun al tuturor numerelor citite. Ex.: pentru `n = 3` și numerele 12, 18, 24 se afișează `6`.

HINT *CMMDC-ul mai multor numere se obține pas cu pas: pornești de la primul element și îl combini, pe rând, cu fiecare dintre celelalte. Varianta cu împărțiri tratează singură zerourile, pentru că* `cmmdc(0, 8)` *este 8.*

SOLUȚIA TA · PYTHON

5 PROBLEMA 7.5 · COD GHIDAT

Este palindrom?

SCRIERE COD

Scrie un subprogram cu antetul exact `def oglindit(n):`, care returnează numărul obținut prin citirea cifrelor lui `n` de la dreapta la stânga. Se citește un număr natural `n` (`n >= 0`). Afișează pe același rând numărul citit, un spațiu, apoi DA dacă numărul este palindrom sau NU în caz contrar. Ex.: pentru `n = 1221` se afișează `1221 DA`; pentru `n = 100` se afișează `100 NU`.

HINT *bucla din corpul subprogramului distruge parametrul, dar variabila* `n` *din programul principal rămâne intactă și poate fi afișată la final, fără nicio variabilă de rezervă. Un număr cu zerouri la coadă nu este palindrom: oglinditul lui 100 este 1.*

SOLUȚIA TA · PYTHON

PROBLEMA 7.6 · PROVOCARE

Câte numere au suma cifrelor un număr prim

SCRIERE COD

Scrie două subprograme, cu antetele exacte `def suma_cifrelor(n):` și `def este_prim(n):`. Primul returnează suma cifrelor numărului primit, al doilea returnează 1 dacă numărul primit este prim și 0 în caz contrar. Se citește un număr natural `n` (`n >= 1`), apoi `n` numere naturale, fiecare pe câte o linie. Afișează câte dintre numerele citite au suma cifrelor un număr prim. Ex.: pentru `n = 4` și numerele 12, 40, 7, 11 se afișează `3`.

HINT *nu combina cei doi algoritmi într-un singur subprogram. Programul principal le cheamă în lanț — rezultatul primului devine parametrul celui de-al doilea.*

SOLUȚIA TA · PYTHON