

Programare 1

Testare, exceptii
Cursul 7

Despre ce am
discutat în cursul
precedent?

Funcții,

- apelul funcțiilor,
- context.
- Variabile locale și globale

Funcții recursive

Module și pachete

Despre ce o
să discutăm
astăzi?

Testare

Depanare

Excepții

Aserțiuni

Calitate?

- Stăm într-un apartament unde sunt gândaci 🐞 și dorim să facem o supă. Din nefericire gândacii tot pică în supă. Cum procedăm ?
 - Verificăm dacă sunt gândaci 🐞 în supă
 - testare
 - Ținem vasul acoperit
 - Programare defensivă
 - Curățăm bucătăria
 - Eliminăm sursa gândacilor

PROGRAMARE DEFENSIVĂ

- Scriem **specificațiile** funcțiilor
- **Modularizăm** programele
- **Validăm** intrările și ieșirile (asertiuni)

TESTARE/VALIDARE

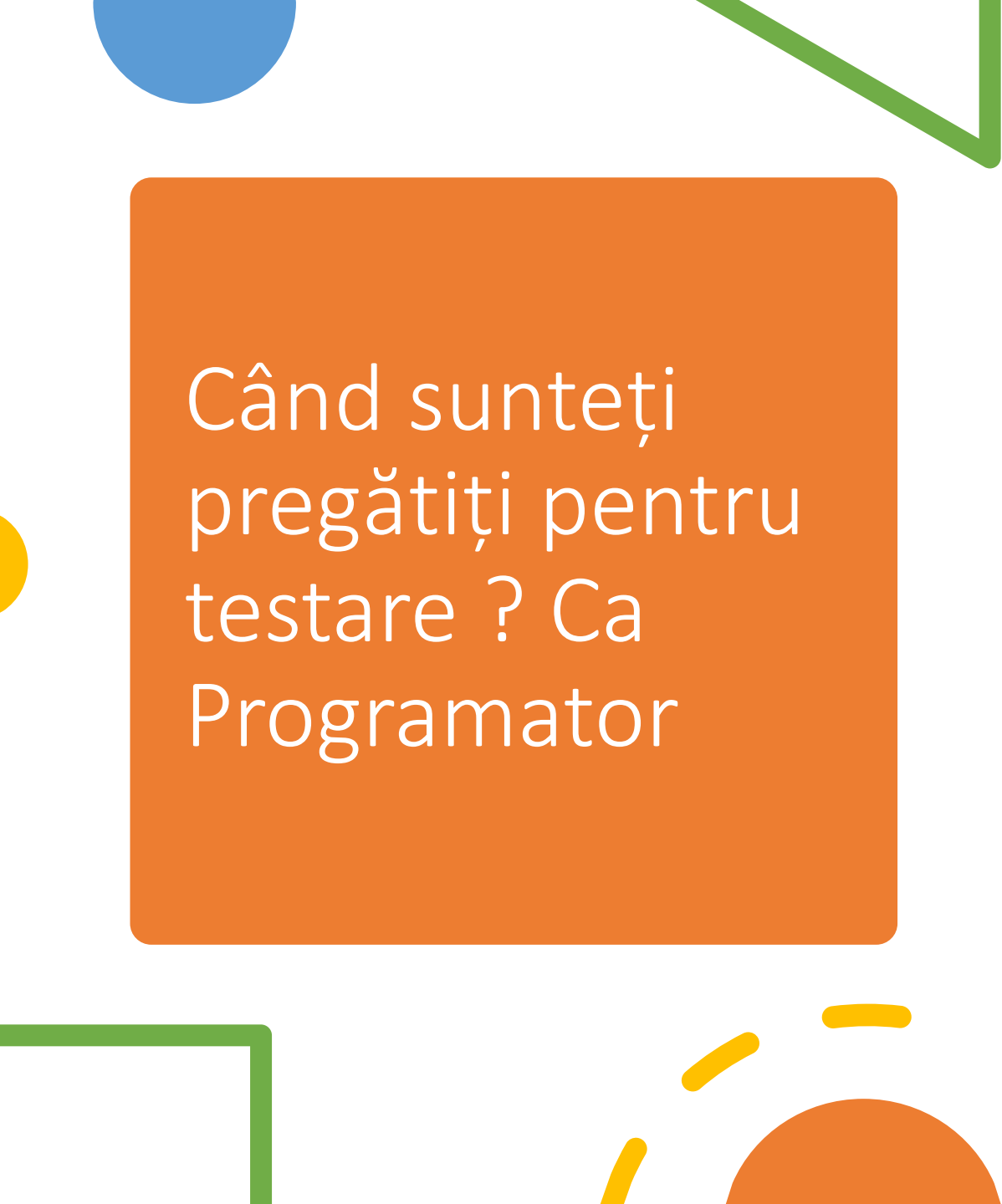
- **Comparam intrările/ieșirile** cu specificația
- “Nu merge!”
- “Cum pot strica intenționat programul ?”

DEPANARE (DEBUGGING)

- **Studiem cauzele** care au dus la eroare
- “De ce nu funcționează ?”
- “Cum îmi pot repara programul ??”

Pregătirea codului pentru testare și depanare

- De la început concepeți codul în așa fel încât să fie testabil și depanabil
- Descompuneți programul în module care pot fi testate și depănate independent
- Documentați constrângerile modulelor
 - Ce se așteaptă să primească ca intrare ?
 - Ce ne așteptăm să returneze ca ieșire ?
- Documentați ipotezele care au stat în spatele unui anumit cod



Când sunteți pregătiți pentru testare ? Ca Programator

- Asigurați-vă că, **codul rulează**
 - Eliminați toate erorile de sintaxă
 - Eliminați toate erorile de semantică statică
 - Python poate detecta uzual aceste erori
- Pregătiți un **set de rezultate** așteptate
 - Un set de intrări
 - Pentru fiecare intrare pregătiți ieșirile corespunzătoare
- Încercați să vă imaginați **o situație** care v-ar putea strica codul

Să ne uităm la problemă din perspectiva utilizatorului

Cerințe

- Se dau variabilele **v1** si **v2**, care contin cate o lista de numere.
- Cele doua liste au acelasi numar de elemente.
- Creati si afisati o lista cu indicii (indexii) pe care elementele din cele doua liste sunt pare.

Comportament așteptat

- Programul va:
 - Parcurge listele
 - Va testa daca elementele de pe acelasi index sunt pare in ambele liste
 - Hint: poate exista o functie separata pentru a testa daca un numar este par ...
 - DACA ambele numere sunt PARE atunci se stocheaza indexul intr-o lista
- Se va afisa lista de indici alesi

Pasul1 – Un simplu test

Scop

- Familiarizarea cu programul

Cum?

- Verificați stabilitatea minimală a programului: frecvent programul „crapă” din capul locului
- Nu petreceți prea mult timp cu această activitate
- Porniți programul și alegeți două liste
 $v1=[2]$ $v2=[4]$

Rezultatele pasului 1

- Rezultate

?[0]

?[]

[1, 2]

? ..

Raport

- Tip raport(programare, design, sugestie, documentație, hardware, întrebare)
- Gravitate (fatal/serios/minor)
- Rezumatul problemei
- Se poate reproduce ?
- Descrierea problemei
- Rezolvare sugerată (opțional)
- Raportată de ...
- Date

Probleme?

- Nu știm ce face programul
- Nu avem instrucțiuni
- Cum oprim programul ?

Actiuni

- Creem rapoarte
- Cate o problema pe raport

Pas 2 – Ce altceva trebuie testat?

- Intrări valide folosind toate cifrele:

- V1 [1, 2, 3] V2[1, 2, 3]
- V1 [-1, 2, 4] V2[1, 2, 3]
- V1 [999, 222, 32] V2[1, 2, 3]
- V1 [1, 2] V2[1, 2, 3]
- V1 [1] V2[1, 2, 3]
- V1 [] V2[1, 2, 3]
- Etc.

Condiții de frontieră

- Clase de teste:
 - Dacă așteptăm același rezultat de la două teste atunci testăm doar unul din ele
- Testăm varianta cel mai probabil să dea eroare
 - Ne uităm la frontiera unei clase
- Găsirea condițiilor de frontieră
 - Nu există o soluție magică, **ne folosim de experiență**
- Frontiere de programare (din codul sursă) vs. Frontierele de testare (perspectiva utilizatorului)
- Testăm amândouă perspectivele

Pașii următori

Pasul 4: Explorarea cazurilor invalide

- Trecerea de la teste formale la teste informale
- Programul „crapă” în mod semnificativ, motiv pentru care trecem la teste informale
- Testăm mai departe cu cazuri invalide
- Nu trebuie să fim formali deoarece poate trebuie să rescriem programul
- Tot timpul notați rezultatele testării

Pasul 5: Rezumați comportamentul programului

- Pentru uzul testorului
 - Îl ajută să se gândească asupra programului, în vederea realizării unei strategii ulterioare de testare
 - Permite identificarea de noi condiții de frontieră
- De exemplu:
 - Stilul de comunicare al programului este unul concis (în cazul nostru afișează o listă)
 - Programul nu știe să lucreze cu numere negative
 - Programul acceptă orice caracter de la utilizator
 - Programul nu verifică dacă a fost introdus un număr cu virgula

Cauze pentru defecțiuni

Erorile parțiale sunt inevitabile

- Scop: prevenirea erorilor generale
- Structurați-vă codul în așa fel încât să fie solid și de încredere

Câteva cauze pentru erori/defecțiuni:

- Abuzul codului (folosirea în alt scop decât cel pentru care a fost creat)
 - Violarea condițiilor
- Erori în cod
 - Bug-uri, erori de reprezentare, etc
- Erori externe imprevizibile
 - Memorie insuficientă
 - Fișiere lipsă
 - Consumul de memorie

Cum le-am putea clasifica ?

- Eroarea unei componente, a unui subansamblu
- Fără rezultat (e.g., elementul nu este găsit, împărțire la zero)

Clase de teste

Teste Unitare (Unit Testing)

- Testarea fiecărei componente a programului
- Testarea fiecărei funcții în mod independent

Teste de regresie

- Adăugați teste pentru existența bug-urilor rezolvate (cum ați rezolvat un bug încercați să scrieți și testul corespunzător)
- Rerulați testele detectând eventual bug-urile rezolvate anterior si reintroduse

Teste de integrare

- Programul per ansamblu funcționează ? (cu toate componentele testate anterior legate între ele)
- Câteodată sărim la această activitate sărind peste celelalte... ☹️

Abordări pentru testare

Intuiția privind situațiile limită ale problemei

- ```
def este_mai_mare(x, y):
```
- `""" Verificăm dacă  $x > y$  și presupunem că  $x$  și  $y$  sunt întregi  
Returnăm True dacă  $y$  este mai mic decât  $x$ , altfel False """`
  - Vă puteți imagina o situație limită (de frontieră)

## Dacă nu găsim situații limită atunci putem încerca testare aleatoare

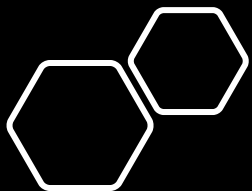
- Probabilitatea ca codul să fie corect crește cu numărul testelor trecut cu succes
- Opțiuni mai bune mai jos

## Testare de tip Black Box

- Testăm de baza specificațiilor
- Suntem un utilizator al programului/funcției

## Testare de tip White Box

- Testăm trasee/scenarii din codul sursă
- programator



# Testare de tip Black Box

```
def sqrt(x, eps):
```

```
 """ Presupune x, eps float,
 • x >= 0, eps > 0
 • Returnează res astfel încât
 • x-eps <= res*res <= x+eps """
```

Tip de testare concepută fără a ne uita la cod sursă

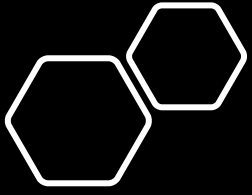
- Poate fi făcută de altcineva decât cel care a implementat, cu scopul de a evita orice interferență (câteodată dezvoltatorul poate fi părtinitor)

Testarea poate fi reutilizată chiar dacă implementarea din spate se schimbă

Trasee de testare prin specificații

- Definirea testelor pe baza specificațiilor
- Considerați situații limită (listă vidă, listă cu un singur element, numere mari, numere mici, etc)





# Testare de tip Black Box

```
def sqrt(x, eps):
 """
 Presupune x, eps float,
 x >= 0, eps > 0
 Returnează res astfel încât
 x-eps <= res*res <= x+eps
 """
```

| CAz                 | x                  | eps                |
|---------------------|--------------------|--------------------|
| limită              | 0                  | 0.0001             |
| Pătrat perfect      | 25                 | 0.0001             |
| Mai mic decât 1     | 0.25               | 0.0001             |
| Rădăcină irațională | 2                  | 0.0001             |
| extreme             | 2                  | $1.0/2.0^{**64.0}$ |
| extreme             | $1.0/2.0^{**64.0}$ | $1.0/2.0^{**64.0}$ |
| extreme             | $2.0^{**64.0}$     | $1.0/2.0^{**64.0}$ |
| extreme             | $1.0/2.0^{**64.0}$ | $2.0^{**64.0}$     |
| extreme             | $2.0^{**64.0}$     | $2.0^{**64.0}$     |

# Testare White Box

- Ne folosim de cod pentru a ne concepe testele
- Este considerată completă dacă fiecare cale posibilă din cod a fost testată cel puțin o dată
- Care sunt slăbiciunile acestei abordări?
  - Ciclurile sunt o provocare deoarece este greu de prevăzut de câte ori se execută
  - Căi lipsă
- Recomandări
  - If-uri → Testați toate ramurile unei expresii condiționale
  - Cicluri for → Testați:
    - nu s-a executat corpul ciclului
    - Corpul ciclului s-a executat o singură dată
    - Corpul ciclului s-a executat de mai multe ori
  - Cicluri while

# Testare White Box

---

```
def abs(x):
 """ presupune ca x este un int
 Returnează x dacă x>=0 și -x altfel """
 if x < -1:
 return -x
 else:
 return x
```

- O testare a tuturor ramurilor ar putea să omită un bug (îl puteți observa ?)
- Test al tuturor ramurilor: 2 și -2
- dar abs(-1) returnează în mod eronat -1

# Depanare

Mai greu de deprins  
dar extrem de utilă

Scopul este să avem  
un program fără  
bug-uri

Unelte

Disponibil în IDLE și  
Anaconda

Ipython, PyCharm,

Instrucțiuni de  
afișare  
(logging/jurnalizare)

Gândiți, fiți  
sistematici în  
căutarea voastră

# Instrucțiuni de afisare

O bună abordare testarea ipotezelor

Când să afișăm

- Când intrăm într-o funcție
- Argumentele
- Rezultatele unei funcții

Folosiți divide-et-impera

- Testați în interiorul codului/funcției
- Decideți unde ar putea fi bug-ul în funcție de valori

# Pași pentru depanare

## Studiați codul sursă

- Nu întrebați ce este greșit
- Întrebați-vă cum ați ajuns la răspunsul greșit
- este parte dintr-o familie de erori ?

## Metoda științifică

- Studiați datele disponibile
- Enunțați o ipoteză
- Experimente repetabile
- Alegeți cele mai simple intrări pentru care programul dă greș

# Mesaje de eroare – Ușor

---

- Tentativă de a accesa dincolo de limita unei liste

```
test = [1,2,3]
atunci test[4]
```

→ IndexError

- Tentativă de a convertii la un tip nepotrivit

```
int(test)
```

→ TypeError

- Referențierea unei variabile inexistente

```
a
```

→ NameError

- Amestecul tipurilor de date fara conversiile de rigoare

```
'3' / 4
```

→ TypeError

- Uitam să închidem paranteze, șiruri de caractere, etc

```
a = len([1,2,3]
print(a)
```

→ SyntaxError

# Erori logice

## - Hard

- Gândiți înainte să scrieți cod nou
- Desenați, luați o pauză
- Explicați codul
  - Să înțeleagă altcineva
  - Aplicați metoda „rubber duck” (unde vă luați o rătușcă și îi explicați fiecare linie de cod 🤪 – vedeti cartea „The Pragmatic Programmer”)
  - Sau altfel spus **Aplicati metoda „explicati bunicii”**



# NU

- Scrieți programe **întregi**
- Testați module **întregi**
- Testați programe **întregi**



# DA

- Scrieți o funcție
- Testați funcția, depanați funcția
- Scrieți o funcție
- Testați funcția, depanați funcția
- \*\*\* Faceți teste de integrare \*\*\*

- Schimbați codul
- Țineți minte unde era bug-ul
- Testați codul
- Uitați unde era codul și ce ați modificat mai înainte
- Intrați în panică



- Salvați codul
- Modificați codul
- Scrieți despre bug într-un comentariu
- Testați codul
- Comparați noul cod cu cel vechi

# Excepții și aserțiuni

---

- Ce se întâmplă când execuția unei funcții se izbește de o **condiție neașteptată** ?
- Obținem o excepție ... privind ce trebuia să se întâmple

- Tentativă de a accesa dincolo de limita unei liste

- - `test = [1,2,3]`
  - `atunci test[4]` → `IndexError`

- Tentativă de a convertii la un tip nepotrivit

- `int(test)` → `TypeError`

- Referențierea unei variabile inexistente

- `a` → `NameError`

- Amestecul tipurilor de date fara conversiile de rigoare

- `'3'/4` → `TypeError`

- Uitam să închidem paranteze, șiruri de caractere, etc

- `a = len([1,2,3]`
  - `print(a)` → `SyntaxError`

# Alte tipuri de erori - cunoscute

## SyntaxError

- Python nu poate interpreta programul

## NameError

- nu poate fi găsită o variabila locala sau globala

## AttributeError

- nu s-a putut găsi un atribut

## TypeError

- operand-ul nu se potrivește cu tipul așteptat de operator

## ValueError

- Tipul operandului este ok dar valoarea este ilegală

## IOError

- Eroare de la sistemul de intrare/ieșire (ex. fișierul nu a fost găsit)

# Gestiunea excepțiilor

---

- Python oferă mecanisme pentru tratarea erorilor

`try:`

```
a = int(input("Spuneți un număr:"))
b = int(input("Spuneți alt număr:"))
print(a/b)
```

`except:`

```
print("Bug în datele de la utilizator.")
```

- Excepțiile aruncate de orice instrucțiune din corpul unui `try` sunt tratate de o instrucțiune `except` iar execuția continuă în corpul instrucțiunii `except`

# Tratarea excepțiilor specifice

---

- Putem avea mai multe instrucțiuni except pentru gestiunea fiecărui tip de excepție

try:

```
a = int(input("Spuneți un număr:"))
b = int(input("Spuneți alt număr:"))
print("a/b = ", a/b)
print("a+b = ", a+b)
```

```
except ValueError:
 print("Nu am putut convertii în număr.")
```

```
except ZeroDivisionError:
 print("Împărțirea la Zero nu este posibilă")
```

```
except:
 print("S-a întâmplat ceva total neprevăzut.")
```

Only execute if this  
errors come up

For all others errors

# Alte clauze pentru try

---

- `else:`
  - bloc de instrucțiuni executat când blocul instrucțiunii *try* se termină fără nici o excepție
- `finally:`
  - Bloc de instrucțiuni executat întotdeauna după instrucțiunile *try*, *else* și *except*, chiar dacă a fost generată o excepție, `break`, `continue` sau `return`
  - Util pentru cod de „clean-up” care ar trebui să fie executat indiferent ce altceva s-a întâmplat (de exemplu acest cod ar putea să închidă fișierele deschise)

# Ce să facem cu excepțiile ?

## Le ignorăm tacit

- Înlocuim cu valori implicite sau pur și simplu ne face că nu s-a întâmplat nimic • **O idee foarte proastă! Utilizatorul nu știe nimic**

## Returnăm un cod de eroare

- Ce valoare să alegem?
- Complică codul deoarece trebuie să verificăm aceste coduri de eroare

## Oprim execuția, semnalăm o eroare.

- in Python: aruncăm o excepție  
`raise Exception("mesaj informativ")`

# Excepții pentru controlul fluxului

- Nu returnați valori speciale când se întâmplă o eroare, verificând mai târziu codul. (spre deosebire de convențiile din C)
- În schimb, **aruncați o excepție** atunci când nu puteți produce un rezultat în concordanță cu specificațiile funcției

```
raise <numeExcepție> (<argumente>)
```

```
raise ValueError("s-a întâmplat ceva")
```

Cuvânt cheie

Numele erorii

Optional un mesaj de eroare




# Exemplu

---

```
def get_rapoarte(L1, L2):
 """ Presupune: L1 și L2 sunt liste de lungime egală
 Returnează: o listă conținând impartirea L1[i]/L2[i] """
 rapoarte = []
 for index in range(len(L1)):
 try:
 rapoarte.append(L1[index]/L2[index])
 except ZeroDivisionError:
 rapoarte.append(float('nan')) #nan = not a number
 except:
 raise ValueError('get_rapoarte apelat gresit')
 return rapoarte
```

Controlul fluxului  
returnand o noua  
excepție



# Exemple de excepții

- Presupunem ca avem o lista cu studenții dintr-o grupă: fiecare intrare este o listă din două elemente:
  - O listă cu prenumele și numele unui student
  - O listă cu notele studenților

```
test_grades = [[['peter', 'parker'], [80.0, 70.0, 85.0]],
 [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- Cream o nouă listă cu numele, notele și o medie (adăugăm un al treilea element)

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],
 [['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]]
```

# Exemplu

```
[[['peter', 'parker'], [80.0, 70.0, 85.0]],
[['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

```
def get_stats(class_list):
 new_stats = []
 for elt in class_list:
 new_stats.append([elt[0], elt[1], avg(elt[1])])
 return new_stats

def avg(grades):
 return sum(grades)/len(grades)
```

# Eroare dacă un student nu are nici o notă

- Primim eroare dacă Dacă unul sau mai mulți **studenți nu au nici o notă**

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],
 [['bruce', 'wayne'], [10.0, 8.0, 74.0]],
 [['captain', 'america'], [8.0, 10.0, 96.0]],
 [['deadpool'], []]]
```

- primim `ZeroDivisionError`: impartire la zero deoarece lista pentru *deadpool* este vida

```
return sum(grades) / len(grades)
```

Lungimea este 0

# Soluție: Semnalăm eroarea afișând o eroare

- Decidem să **notificăm** utilizatorul atunci când a apărut o problemă

```
def avg(grades):
 try:
 return sum(grades)/len(grades)
 except ZeroDivisionError:
 print('atenție: nu avem note')
```

- Rulând pe niște date de test

atenție: nu avem note

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
 [['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.833333334],
 [['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
 [['deadpool'], [], None]]
```

**Semnalăm eroarea**

**None deoarece avg nu a  
returnat nimic**

# Soluție: schimbăm abordarea

- Decidem să **notificăm** utilizatorul atunci când a apărut o problemă

```
def avg(grades):
 try:
 return sum(grades)/len(grades)
 except ZeroDivisionError:
 print('atenție: nu avem note')
 return 0.0
```

- Rulând pe niște date de test

```
'atenție: nu avem note'
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.833333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], 0.0]]
```

*încă semnalăm eroarea*

*avg returnează acum 0*

# Aserțiuni

---

Vrem să ne asigurăm că **ipotezele** privind starea unui calcul sunt așa cum ne așteptăm

---

Folosim instrucțiunea **assert** pentru a arunca excepția **AssertionError** atunci când ipoteza nu este satisfăcută

---

Un bun exemplu de programare defensivă

# Exemplu

---

```
def avg(grades):
 assert len(grades) != 0, 'nu avem note'
 return sum(grades)/len(grades)
```

*Funcția se termină  
imediat ce condiția nu  
mai este satisfăcută*

- Generează `AssertionError` dacă primește o listă vidă ca argument
- Altfel rulează ok



# Aserțiuni și Programare Defensivă

---

Aserțiunile nu îi permit unui programator sa controleze comportamentul în situații neprevăzute

---

Scopul este ca execuția **să se oprească** oricând dăm de o situație neprevăzută

---

Sunt folosite uzual pentru **verificarea argumentelor** unei funcții dar pot fi folosite oriunde

---

Pot fi folosite pentru **verificarea rezultatelor** unei funcții, cu scopul prevenirii propagării valorilor eronate

---

Ne permit să determinăm mai ușor sursa unui bug

# Aserțiuni și Programare Defensivă

- Verifică
  - Precondiții
  - Postcondiții
  - Invariantul
  - Alte proprietăți cunoscute ca fiind adevărate
- Verificare statică manuală (& și cu unelte)
- Verificare dinamică în timpul execuției

```
assert index >= 0;
```

```
assert size % 2 == 0, "Dimensiune
eronată pentru listă"
```

- Scrieți aserțiunile în timp ce scrieți cod

# Unde să folosim aserțiuni

- Scopul este să găsim bug-uri cat mai repede după ce le-am introdus și să facem evidentă locația unde au apărut
- Folosiți ca un **supliment pentru testare**
- Aruncați **excepții** când utilizatorul furnizează **date greșite**
- Folosiți **aserțiuni** pentru
  - Verificarea **tipului** argumentelor sau valorilor
  - Verificați că **invariantii** sunt satisfăcuți
  - Verificați **constrângerile** datelor de ieșire
  - Verificați **încălcările** constrângerilor algoritmului (ex să nu fie duplicate în listă)

# Excepții

- Folosiți **excepții**
  - Folosite într-un context larg și imprevizibil
  - Este fezabilă verificarea precondițiilor
- Folosiți **precondiții** atunci când:
  - Verificare explicită este prea costisitoare
    - Ex: verificare dacă o listă este sortată
  - Folosite într-un context mai restrâns în care apelurile pot fi verificate
- Evitați precondițiile pentru că:
  - Apelantul poate viola precondițiile
  - Programul poate eșua într-un mod imprevizibil și potențial periculos
  - Ne dorim ca programul să eșueze cât mai devreme posibil
- Cum diferă precondiții și excepțiile pentru un client ?

# Bibliografie

- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/>