



# Python 101

Curs 1 - Introducere în Python  
20.03.2023

# Despre Hackademy

- Cursuri: **CCNA, Python 101, Web 101.**
- Evenimente: Vezi pagina de Facebook:

<https://www.facebook.com/hackademy.ro>

o



# Meet our NetAcad

- Toate informațiile cursului se găsesc în același loc.

[https://lms.netacad.com/course/view.php  
?id=892148](https://lms.netacad.com/course/view.php?id=892148)

# Despre echipă

## Instructori:

Predescu Ioan-Alexandru    Filip Emil Florentin  
Bîrchi Sorin-Ioan-Alexandru

## Infrastructură:

Tulpan Andrei



# Despre curs

# Pentru Început...

-  Discord + NetAcad
  - Curs
  - Materiale și anunțuri
-  Luni 18:00 – 21:00
  - Quiz de recapitulare din cursul precedent
  - Curs + Demo
  - Laborator
-  Punetăți întrebări oricând
-  Feedback la fiecare curs

# Calendarul cursului

Nr. curs	Titlu	Săptămână
1	Introducere - Sintaxa - Colecții	23.10.2023
2	Paradigme de Programare	30.10.2023
3	Programare orientată pe obiecte	06.11.2023
4	Module	13.11.2023
5	Flask	20.11.2023
6	Workshop Git	04.12.2023
7'	<i>Examen</i>	11.12.2023
8	<i>Workshop Pitch-uri</i>	18.12.2023
9	Prezentarea Proiectelor	15.01.2024

# Punctaj

- Parcurs - **3p**
  - Laboratoare - **2p**
  - Quiz de Prezenta - **1p**
- Proiect - **4p**
  - Prezentare - **1.5p**
  - Calitatea codului - **1.5p**
  - Demo - **1p**
- Examen - **4p**
  - Hackerrank

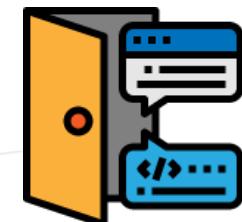
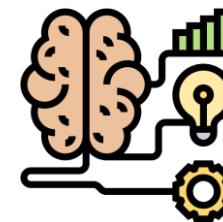
Minimum **7p** și **prezentarea proiectului** pentru promovarea cursului.



# Introducere în Python

# Utilizări

- Prototipare
- Automatizare
- Machine Learning
- Backend pentru aplicații web



# Limbajul Python

- Vom folosi Python 3
- Este un limbaj interpretat
- Un cod Python este transformat într-un format intermediar, numit bytecode, care este trecut în limbaj mașină pentru fiecare arhitectură

# Dezavantajele Python

- Interpretarea costă timp.
- Gestiunea memoriei prin garbage collector.
- Ineficient în platformele mobile.

# Avantajele Python

- Flexibilitate.
- Ușor de folosit și învățat.
- Biblioteci diverse.
- Comunitate activă.
- Centrat pe rezolvarea problemei, ci nu pe probleme de sintaxă, memorie etc.

# Comentarii în cod

- Pentru comentarii pe o singură linie se folosește caracterul “#”.

# Acesta este un comentariu pe o linie.

- Pentru comentarii pe mai multe linii se folosește “”””.

””””

Acesta este un comentariu foarte,  
foarte lung.

””””

# Declararea variabilelor

- Pentru declararea variabilelor se folosește operatorul de atribuire “=”.
- O variabilă poate fi folosită doar după ce a fost declarată.

```
a = 42
```

```
b = "Fred"
```

# Tipuri primitive de date

- Integer:
- Float:
- Bool:
- String:

x = 2

y = 2.2

is\_empty = False

name = "Fred"

Putem afla tipul de date al unei variabile folosind funcția **type**.

```
is_full = True  
type(is_full) # <class 'bool'>
```

# Conversie între tipuri

Cele mai folosite conversii sunt:

- de la string la int:

- `i = int("123")` # 123

- de la string la float:

- `f = float("7.23")` # 7.23

- de la int / float / bool la string:

- `s = str(3)` # "3"

- `s = str(3.14)` # "3.14"

- `s = str(True)` # "True"

# Tiparea limbajelor (1)

## Tipare dinamică

- Tipul unei variabile se stabilește la atribuire.

Exemplu: Javascript

```
let name = "static";
name = 2; // Valid
```

## Tipare statică

- Tipul unei variabile se stabilește la definire.

Exemplu: C++

```
string name = "static";
name = 2; // Eroare
```

# Tiparea limbajelor (2)

## Tipare slabă

- Tipurile variabilelor se modifică fără conversie explicită.

### Exemplu: Javascript

```
let x = "4";
let y = 20;

/* "420" */
console.log(x + y);
```

## Tipare puternică

- Tipurile variabilelor se modifică doar prin conversie explicită.

### Exemplu: C/C++

```
string x = "4";
int y = 20;

/* "420" */
cout << x + to_string(y);
```

# Tiparea din Python

Tiparea în Python este:

- puternică - prin conversie explicită.

```
x = "4"  
y = 20  
print(x + str(y)) # "420"
```

- dinamică - tip stabilit la atribuire.

```
name = "Fred"  
name = 2 # Valid
```

# Tipuri de operatori

Operatori:

- aritmetici
- pe biți
- de atribuire
- de comparație
- logici
- pe string-uri

# Operatori aritmetici

$x = 7$

$y = 2$

Operator	Descriere	Exemplu	Rezultat
$+$	adunare	$x + y$	9
$-$	scădere	$x - y$	5
$*$	înmulțire	$x * y$	14
$/$	împărțire cu virgulă	$x / y$	3.5
$//$	împărțire întreagă	$x // y$	3
$\%$	restul împărțirii	$x \% y$	1
$**$	ridicare la putere	$x ** y$	49

# Operatori pe biți

`x = 7 # 00000111`

`y = 2 # 00000010`

Operator	Descriere	Exemplu	Rezultat
<code>&amp;</code>	și	<code>x &amp; y</code>	<code>2 # 00000010</code>
<code> </code>	sau	<code>x   y</code>	<code>7 # 00000111</code>
<code>^</code>	xor	<code>x ^ y</code>	<code>5 # 00000101</code>
<code>~</code>	not	<code>~x</code>	<code>-8 # 11111000</code>
<code>&lt;&lt;</code>	Shiftare la stânga	<code>x &lt;&lt; y</code>	<code>28 # 00011100</code>
<code>&gt;&gt;</code>	Shiftare la dreapta	<code>x &gt;&gt; y</code>	<code>1 # 00000001</code>

# Operatori de atribuire

Operatorii de atribuire se formează punând “=” după operatorii aritmetici sau pe biți.

`x = 7`

`y = 2`

Operator	Exemplu	Valoarea lui x
<code>+=</code>	<code>x += y</code>	9
<code>%=</code>	<code>x %= y</code>	1
<code>**=</code>	<code>x **= y</code>	49
<code>&lt;=&gt;</code>	<code>x &lt;=&gt; y</code>	28
...	...	...

# Operatori de comparație

$x = 7$

$y = 2$

Operator	Descriere	Exemplu	Rezultat
<code>==</code>	egal cu	$x == y$	False
<code>!=</code>	diferit de	$x != y$	True
<code>&gt;</code>	mai mare	$x > y$	True
<code>&lt;</code>	mai mic	$x < y$	False
<code>&gt;=</code>	mai mare sau egal	$x >= y$	True
<code>&lt;=</code>	mai mic sau egal	$x <= y$	False

# Operatori logici

x = True

y = False

Operator	Descriere	Exemplu	Rezultat
and	și	x and y	False
or	sau	x or y	True
not	not	not x	False

# Operații pe string-uri

`x = "Ce "`

`y = "faci?"`

Operator	Exemplu	Rezultat
<code>+</code>	<code>x + y</code>	<code>"Ce faci?"</code>
<code>*</code>	<code>x * 2</code>	<code>"Ce Ce "</code>

# Operații pe string-uri(2)

x = "Fred"

y = "El e Fred."

Operație	Explicație	Rezultat
x in y	Verifică dacă x este conținut în y	True
len(x)	determină lungimea sirului x	4
y.find(x)	Întoarce indicele primei apariții a lui x în y (sau -1, în caz că nu există)	5

# Operații pe string-uri(3)

x = "Fred"  
"ed"

c1 =

c2 = "am"

Operație	Explicație	Rezultat
x.upper()	Transformă toate literele mici ale lui x în litere mari	"FRED"
x.lower()	Transformă toate literele mari ale lui x în litere mici	"fred"
x.replace(c1, c2)	Înlocuiește toate aparitiile lui c1 din x cu c2	"Fram"

# Instructiuni de control

- Instructiuni pentru ramificarea executiei:
  - if, elif, else
- Instructiuni de repetitie:
  - for, while
- Instructiuni speciale:
  - break, continue, return

# Ramificare și context

- Ramificarea execuției se face prin instrucțiunile if, elif, else.
- Fiecare instrucțiune trebuie urmată de “:”.
- Contextul este definit prin tab-uri.

```
if 2 == 3:  
    print("Sunt în if")  
elif 2 == 2:  
    print("Sunt în elif")  
else:  
    print("Sunt în else")  
print("Nu mai sunt în if")
```

# Instructiunea for (1)

- Instructiune cu număr cunoscut de pași.
- Putem stabili numărul de pași cu funcția **range**.
- **range(n)** întoarce valorile de la 0 la **n-1**.

```
for i in range(3) :  
    print(i)
```

```
# 0  
# 1  
# 2
```

# Instructiunea for (2)

- **range(start, stop)** întoarce valorile de la **start** la **stop-1**.

```
for i in range(3, 6):  
    print(i)  
# 3  
# 4  
# 5
```

# Instructiunea for (3)

- **range(start, stop, p)** întoarce valorile de la **start** la **stop - 1** cu pasul **p > 0**.

```
for i in range(3, 10, 2):  
    print(i)  
# 3  
# 5  
# 7  
# 9
```

# Instructiunea for (4)

- **range(start, stop, p)** întoarce valorile de la **start** la **stop + 1** cu pasul **p < 0**.

```
for i in range(10, 3, -2):  
    print(i)  
# 10  
# 8  
# 6  
# 4
```

# Instructiunea for (5)

- Execuția poate fi sărită cu instrucțiunea **continue**.

```
for i in range(3, 10):  
    if i % 2 == 0:  
        continue  
        print(i)  
    # 3  
    # 5  
    # 7  
    # 9
```

# Instructiunea for (6)

- Execuția poate fi oprită cu instrucțiunea **break**.

```
for i in range(3, 10):  
    if i == 6:  
        break  
        print(i)  
    # 3  
    # 4  
    # 5
```

# For pentru string-uri

- Pentru a itera prin caracterele unui string, folosim sintaxa **for c in s**, în care variabila **c** va lua pe rând toate caracterele din **s**.

```
s = "ABC"  
for c in s:  
    print(c + "1")  
    # A1  
    # B1  
    # C1
```

# Instructiunea while (1)

- Instructiune cu conditie de executie.

```
i = 0
while i < 3:
    print(i)
    i += 1
```

```
# 0
# 1
# 2
```

# Instructiunea while (2)

- Execuția poate fi sărită cu instrucțiunea **continue**.

```
i = -1
while i < 4:
    i += 1
    if i == 2:
        continue
    print(i)
#
# 0
# 1
# 3
```

# Instructiunea while (3)

- Execuția poate fi oprită cu instructiunea **break**.

```
i = 0
while i < 4:
    if i == 2:
        break
    print(i)
    i += 1

# 0
# 1
```

# Citirea de la tastatură

- Putem citi de la tastatură folosind funcția `input`.
- Input returnează **mereu** string-uri.

```
name = input()  
number = int(input("Introduceti un numar: "))
```

# Afișarea pe ecran

Putem afișa pe ecran folosind funcția **print**.

```
print("Python")    # Python
print(True)        # True
print(3.14)         # 3.14
print(3)           # 3
```

# Afișarea pe ecran (2)

- Pentru a schimba caracterul ce se pune după print, putem să folosim end = (implicit avem linie nouă).

```
print("Py", end="-")
print("thon", end=".")  
# Py-thon.
```

# Afișare string-uri (1)

Pentru a insera parametrii într-un string, putem folosi **f-strings** (>= Python 3.6).

```
nume = "Peter"  
nota = 10  
s = f"{nume} are nota {nota}."  
  
print(s)      # Peter are nota 10.
```

# Afișare string-uri (2)

O altă metodă de a insera parametrii într-un string este folosind `.format` în care punem parametrii la final.

```
nume = "Peter"  
nota = 10  
s = "{ } are nota { }.".format(nume, nota)  
  
print(s)      # Peter are nota 10.
```



# Pauză

# Functii și colecții

# Colecții

- Nu putem stoca toată informația în variabile.
- Trebuie să stocăm date într-un mod ordonat și după reguli fixe.
- O grupare de date este o colecție.

# Tuplu (1)

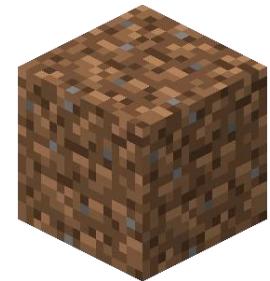
- Este o structură de date cu mai multe câmpuri care pot fi de orice tip.
- Câmpurile pot fi accesate prin index (primul index este 0).
- Câmpurile nu pot fi modificate individual.
- Odată creat, nu pot fi adăugate câmpuri unui tuplu.

# Tuplu (2)

Un tuplu se creează cu () .

```
# Bloc din Minecraft - (x, y, z, type)
```

```
x = (1, 2, 3, "Dirt")
```



```
print(x[3]) # Dirt
```

```
print(x) # (1, 2, 3,  
'Dirt')
```

```
x[3] = "Diamond Ore" # eroare
```

```
x = (x[0], x[1], x[2], "Diamond Ore")
```



# Listă (1)

- Structură de date ce permite stocarea a oricâte elemente de orice tip.
- Elementele pot fi accesate prin index.
- Elementele pot fi modificate.

# Listă (2)

O listă se creează cu `[]`.

```
l1 = []  
print(l1)      # []
```

```
l2 = [1, 2, 3]  
print(l2)      # [1, 2,  
3]
```

```
l3 = ["Red", 2]  
print(l2)      # ['Red',  
2]
```

Lungimea unei liste se obține cu funcția `len`.

```
len(l1)      # 0
```

```
len(l2)      # 3
```

```
len(l3)      # 2
```

# Listă (3) - Accesare

Elementele listei pot fi accesate prin index, care poate fi atât pozitiv, cât și negativ.  
(primul index este 0).

```
l = [1, 2, 3, 4, 5]
```

```
print(l[1])          # 2  
print(l[4])          # 5
```

```
print(l[-1])         # 5 (ultimul element)  
print(l[-2])         # 4 (penultimul element)  
print(l[-5])         # 1 (primul element)
```

Pentru un index mai mare decât 4 sau mai mic decât -5, se obține următoarea eroare: "list index out of range".

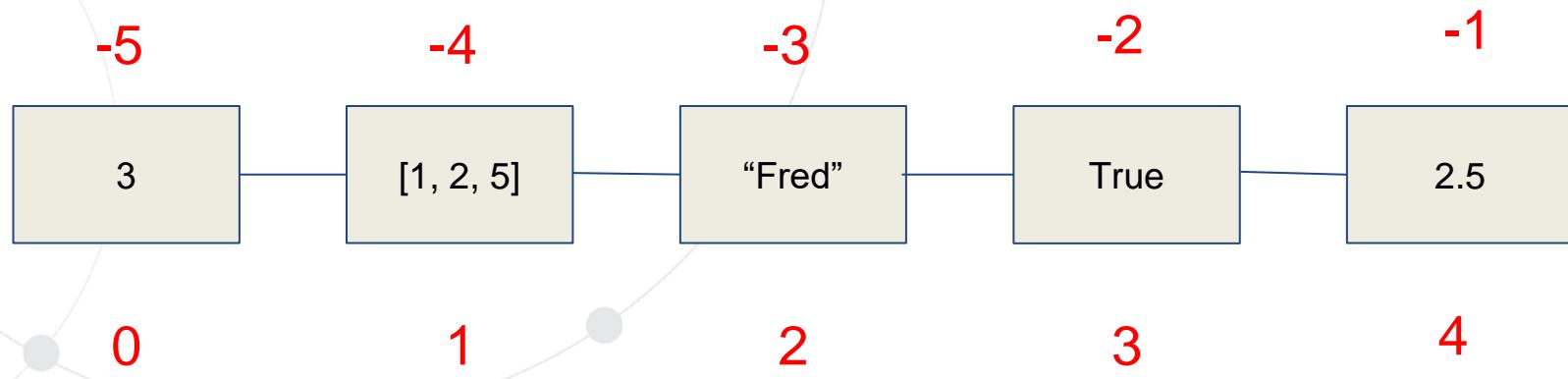
# Listă (4) - Slicing

Putem selecta anumite elemente dintr-o listă folosind slicing, cu sintaxa:

**lista[start:stop:pas],**

```
l = [1, 2, 3, 4, 5]
print (l[1:4:1])    # [2, 3, 4]
print (l[1:4:])     # [2, 3, 4]
print (l[::-2])      # [1, 3, 5]
print (l[::-1])      # [1, 2, 3, 4, 5]
print (l[3:0:-1])    # [4, 3, 2]
```

# Listă (5) - Slicing



```
l = [3, [1, 2, 5], "Fred", True, 2.5]
```

# Inversarea unei liste

Putem inversa o listă folosind **slicing**, astfel: **list[::-1]**.

```
l = [1, 2, 7]
print(l[::-1]) # [7, 2, 1]
```

# Listă (6)

Adăugare element cu **append**.

```
l = [1]
print(l)

l.append(2)
print(l)          # [1,
                  2]
```

# [1]

Adăugare element cu **insert**.

```
l = [3]
print(l)          # [3]

l.insert(0, 4)
print(l)          # [4,
                  3]
```

```
l.insert(2, 5)
print(l)          # [4,
                  3, 5]
```

# Listă (7)

Eliminare element cu **pop**.

```
l = [1, 2, 3, 4]
print(l)      #      [1, 2,
3, 4]
```

```
# Eliminare la un index
l.pop(1)
print(l)      #      [1, 3,
4]
```

```
# Eliminare ultim element
l.pop()
print(l)      #      [1, 3]
```

Eliminare element cu **remove**.

```
l = [1, 1, 2, 3]
print(l)      #      [1, 1,
2, 3]
```

```
l.remove(1)
print(l)      #      [1, 2,
3]
```

```
l.remove(2)
print(l)      #      [1, 3]
```

# Listă (8) - Ștergere

Dacă dorim să ștergem toate elementele unei liste, putem folosi metoda **clear**.

```
l = [1, 2, 3, 4, 5]  
print(l) # [1, 2, 3, 4, 5]  
l.clear()  
print(l) # []
```

# Sortarea unei liste

Sortarea unei liste se face folosind funcția **sorted**. Putem sorta lista în ordine descrescătoare cu argumentul **reverse=True**. Această funcție întoarce lista sortată și nu modifică lista primită.

```
l = [3, 0, -5, 2]
print(sorted(l))
    # [-5, 0, 2, 3]
print(sorted(l, reverse=True) )          #
[3, 2, 0, -5]
print(l)
    # [3, 0, -5, 2]
```

# Iterarea printr-o listă

Pentru a itera prin elementele unei liste, folosim sintaxa **for e in list**, în care variabila **e** va lua pe rând valoarea tuturor elementelor.

```
l = [1, 5, 7]
for e in l:
    print(e)
# 1
# 5
# 7
```

# Căutarea printr-o listă

Pentru a verifica că un element **e** se află într-o listă folosim sintaxa: **e in list**.

```
l = [1, 5, 7]
print(1 in l)    # True
print(2 in l)    # False
```

# Copierea unei liste

Folosind operatorul **=**.

```
a = [3, 4, 5]
b = a

print(b)      # [3, 4, 5]
b[0] = 1
```

```
print(a) # [1, 4, 5]
print(b) # [1, 4, 5]
```

Orice modificare asupra copiei, se reflectă și asupra listei originale.

Folosind metoda **copy**.

```
a = [3, 4, 5]
b = a.copy()

print(b)      # [3, 4, 5]
b[0] = 1

print(a) # [3, 4, 5]
print(b) # [1, 4, 5]
```

Modificările asupra copiei **NU** se reflectă și asupra listei originale.

# Concatenarea listelor

Folosind operatorul **+**  
(întoarce lista rezultat).

```
a = [3, 4]  
b = [1, 2]  
a = a + b
```

```
print(a)      # [1, 2,  
3, 4]
```

Folosind metoda **extend**  
(adăugă elementele la finalul  
listei pe care a fost apelată  
metoda).

```
a = [3, 4]  
b = [1, 2]  
b.extend(a)
```

```
print(b)      # [1, 2,  
3, 4]
```

# List comprehension (1)

O modalitate de creare a listelor pe baza unor anumite criterii este prin list comprehension, astfel:

**[expresie for element in list]**

```
l = [i ** 2 for i in range(4)]  
  
print(l)                      # [0, 1, 4, 9]
```

# List comprehension (2)

Într-un list comprehension putem inclusiv să impunem o condiție elementelor din listă.

**[expresie for element in list if condiții]**

```
l = [i ** 2 for i in range(8) if i % 2 == 0]
print(l) # [0, 4, 16, 36]
```

```
l = [i for i in range(20) if i % 2 == 0 and i % 3 == 1]
print(l) # [4, 10, 16]
```

# List comprehension (3)

Putem să construim inclusiv matrice (ca o listă de liste) cu ajutorul list comprehensions.

```
l = [[ i + j for i in range(3) ] for j in range(4) ]  
print(l)
```

```
# [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5]]
```

# Conversia la o listă

Orice colecție poate deveni o listă, prin sintaxa **list(colecție)**.

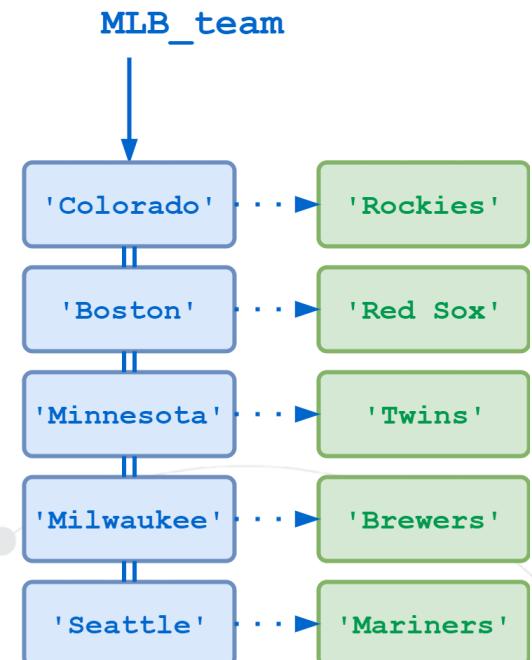
```
t = (1, 2, True)
l = list(t)
print(l)      # [1, 2, True]
```

```
s = "Fred"
l = list(s)
print(s)      # ['F', 'r', 'e', 'd']
```

# Dicționar

Este structura de date ce permite stocarea unei perechi de tipul cheie, valoare.  
Un dicționar se creează cu {}.

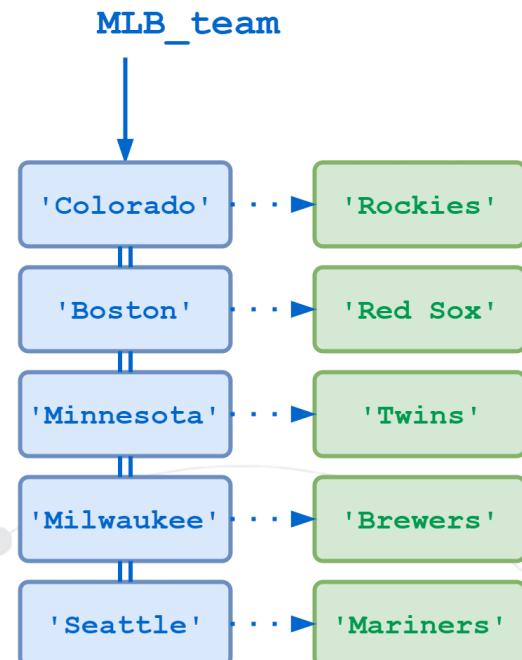
```
team = {}  
print(team) # {}  
  
team = {  
    'Colorado' : 'Rockies',  
    'Boston'    : 'Red Sox',  
    'Minnesota': 'Twins',  
    'Milwaukee': 'Brewers',  
    'Seattle'   : 'Mariners'  
}
```



# Accesarea unui element

Pentru a lua valoarea unei key dintr-un dicționar **d**, avem sintaxa **d[key]**. Dacă cheia nu există, vom primi eroare.

```
team = {  
    'Colorado' : 'Rockies',  
    'Boston'    : 'Red Sox',  
    'Minnesota': 'Twins',  
    'Milwaukee': 'Brewers',  
    'Seattle'   : 'Mariners'  
}  
  
print(team['Colorado']) # "Rockies"
```



# Adăugarea într-un dicționar

Putem adăuga o cheie într-un dicționar folosind:

**d[cheie] = valoare**

Dacă valoarea există deja, ea va fi suprascrisă.

```
loot = {}  
loot["white"] = 10  
print(loot) # {'white': 10}  
loot["white"] = 20  
print(loot) # {'white': 20}
```

# Căutarea în dicționar

Pentru a verifica dacă o cheie aparține unui dicționar, este o sintaxă asemănătoare listelor.

**if cheie in d:**

```
loot = {"white" : 10, "red" : 20}  
print("red" in loot)          # True  
print("blue" in loot)        # False
```

# Iterarea printr-un dicționar

## (1)

Pentru a itera prin cheile unui dicționar, folosim **d.keys()**, care întoarce o listă cu cheile din dicționar.

```
loot = {"white" : 10, "red" : 20}
```

```
for key in d.keys():
    print(key)
```

```
# white
# red
```

# Iterarea printr-un dicționar (2)

Pentru a itera prin valorile unui dicționar, folosim **d.values()**, care întoarce o listă cu valorile din dicționar.

```
loot = {"white" : 10, "red" : 20}
```

```
for value in d.values():
    print(value)
```

```
# 10
# 20
```

# Iterarea printr-un dicționar

## (3)

Pentru a itera prin elementele unui dicționar, folosim **d.items()**, care întoarce o listă de tupluri (cheie, valoare) din dicționar.

```
loot = {"white" : 10, "red" : 20}
```

```
for key, value in loot.items():
    print(f'{key} - {value}')
```

```
# white - 10
# red - 20
```

# Set

- Seturile sunt o structură de date, ce oferă posibilitatea de a construi și manipula colecții neordonate de elemente unice.
- Nu acceptă indexare sau slicing.
- Un set se creează cu {}.

# Set - operații

`s = set()`

Operație	Explicație	Exemplu	Rezultat
<code>s.add(x)</code>	Adaugă elementul x la set	<pre>s.add(4) s.add(5) s.add(5) print(s)</pre>	{ 4, 5 }
<code>s.remove(x)</code>	Șterge elementul x din set. Dacă acesta nu există, se va întoarce eroare.	<pre>s.add("Andi") s.add("Marcel") s.remove("Andi") print(s)</pre>	{"Marcel"}
<code>x in s</code>	Verifică dacă elementul x se află în set.	<pre>s.add(5) s.add(6) print(5 in s) print(4 in s)</pre>	False True

# Set - operații matematice

```
s1 = {1, 2, 3, 6}  
s2 = {3, 4, 5}
```

Operație	Explicație	Rezultat
<code>s1.union(s2)</code>	Întoarce un set cu reuniunea celor două multimi.	{1, 2, 3, 4, 5, 6}
<code>s1.intersection(s2)</code>	Întoarce un set cu intersecția celor două multimi.	{3}
<code>s1.difference(s2)</code>	Întoarce un set cu diferența celor două multimi (elementele care sunt în s1 și nu sunt în s2).	{1, 2, 6}



# Întrebări?