Programarea calculatoarelor si limbaje de programare 1

Tipuri si operatii numerice

Universitatea Politehnica din Bucureşti

Sumar

- ***Numere, notiuni de baza***
- Expresii numerice
- Alte tipuri numerice
- Extensii numerice
- Tipuri dinamice in Python

Clasificare obiecte numerice

- Intregi si numere in virgula mobila (reale)
- Tipul *complex*
- Tipul *Decimal* – numere reale cu precizie fixa
- Tipul *Fraction* – numere rationale cu numarator si numitor
- Tipul *set* – multime
- Tipul *bool* – *True* si *False*
- Functii si module predefinite – *round*, *math*, *random*
- Expresii, intregi cu precizie nelimitata, operatii pe biti, formate *hex*, *oct*, *bin*
- Extensii numerice (ex. *NumPy*)

Constante numerice

Expresie numerica	Semnificatie
1234, -25, 0, 99999999999	Intreg cu oricate cifre
12.34, 2., 2.77e-15, 3E120, 4.0E120	Numere reale
0o377, 0xff, 0b11101110	Octal, hexazecimal, binar (3.X)
1 + 2j, 1.0 + 2.0j, 3J	Numere complexe
set('spam'), {1, 2, 3, 4, 5 }	Multimi
Decimal('3.14'), Fraction(3, 4)	Decimali si fractii rationale
bool(x), True, False	Tipul *bool* – ean

- Numerele reale au punct zecimal sau/si exponent – *e* sau *E*; precizia data de tipul *double* din C (in CPython)
- Intregii au precizie nelimitata, oricate cifre
- Intregii sunt reprezentati in baza 10 de numeratie, dar si in:
  - hexazecimal – prefix 0x sau 0X, cifre 0-9, a-f, A-F

Constante…

- octal – prefix 0o sau 0O, cifre 0 -7
- binar – prefix 0b sau 0B, cifre 0 si 1.

- operatorii predefiniti *hex(I), oct(I), bin(I)* convertesc un intreg la *str* – ing; *int(str, baza)* returneaza un intreg in respectiva baza de numeratie.
- Numerele complexe se reprezinta ca *parte_reala* + *parte_imaginara****j (sau J)***; operatorul predefinit *complex(reala, imaginara)* produce un numar complex.
- alte tipuri folosesc module importate (*decimal*, *fractions*) sau sintaxa specifica (*set*)

Instrumente predefinite

- Operatorii numerici: +, -, *, /, >>, **, &,…
- Functii matematice: pow, abs, round, int, hex, bin,…
- Module matematice: random, math, cmath (complexe),…

Exista si metode specifice tipurilor numerice ex. reali – *as_integer_ratio(), is_integer()*, intregi – *bit_length()*

Precedenta operatorilor Python

Operator (ascendent)	Descriere
yield x	Generator, protocolul *send*
lambda args: expresie	Functie anonima
x if y else z	Selectie ternara (x evaluat daca y este True)
x or y	Sau logic (y evaluat daca x este False)
x and y	Si logic (y evaluat daca x este True)
not x	Negatie logica
x in y, x not in y	Apartenenta (iterabile, multimi)
x is y, x is not y	Testarea identitatii obiectelor
x < y, x <= y, x > y, x >= y	Comparatie; contine, inclus – set
x == y, x != y	Egalitate, diferit – ca valoare

Precedenta…

Operator (asc)	Descriere
x | y	Sau pe biti, uniune – set
x ^ y	Sau exclusiv pe biti, diferenta simetrica – set
x & y	Si pe biti, intersectie – set
x << y, x >> y	Deplasare x cu y biti stanga, repectiv dreapta
x + y	Adunare, concatenare – str
x - y	Scadere, diferenta – set
x * y	Inmultire, repetitie – str
x % y	Rest impartire, format – str
x / y, x // y	Impartire, propriu-zisa sau “floor”
-x, +x	Negatie, identitate

Precedenta…

Operator (asc)	Descriere
~x	Inversiune biti
x ** y	Ridicare x la puterea y
x[i]	Indexare (secvente, mapari)
x[i:j:k]	Slicing
x()	Apel functie, metoda, clasa, instanta
x.atribut	Selectie atribut
()	Tuple, grupari expresii, generator
[]	List, colectii iterative
{}	Dict, set, colectii iterative

Reguli de evaluare a expresiilor

- Se bazeaza pe precedenta operatorilor:
  - au prioritate operatorii de la sfarsitul tabelei
  - operatorii din aceeasi linie se evalueaza de la stanga la dreapta, cu exceptia ridicarii la putere si a comparatiilor inlantuite
- Expresiile intre paranteze se evalueaza mai intai
- Expresiile numerice mixte se convertesc la tipul operandului mai complicat:

Reguli…

- In ordinea ascendenta a complexitatii: intregi, reali, numere complexe
- In 3.X nu este permis amestecul de tipuri diferite (*str* cu numeric)

- Operatorii:
  - pot fi inlocuiti in clase noi de obiecte – *overloading*
  - Python foloseste overloading, ex. **+** pentru adunare numere sau concatenare de secvente (striguri)
  - **Polimorfism**: efectul operatiei depinde de obiectele asupra carora se aplica.

Sumar

- Numere, notiuni de baza
- ***Expresii numerice***
- Alte tipuri numerice
- Extensii numerice
- Tipuri dinamice in Python

Expresii si variabile

Variabilele:

- se creeaza prin asignarea de valori
- sunt inlocuite cu valoarea curenta in expresii
- trebuiesc initializate inainte de folosire
- refera obiecte si nu se declara in avans

```
>>> a = 3        #numele de variabile sunt create iara nu declarate
>>> b = 4
>>> a + 1, a - 1 #adunare, scadere
(4, 2)
>>> b * 2, b / 2 #inmultire, impartire
(8, 2.0)
>>> a % 2, b ** 2 #rest, ridicare la putere
(1, 16)
```

```
>>> 2 + 4.0, 2.0 ** b #tipuri mixte
(6.0, 16.0)
>>> c * 2    #raportare eroare pentru variabile neasignate
NameError: name 'c' is not defined
```

Formate de afisare

Grupare operatori; tipuri numerice diferite:

```
>>> b / 2 + a   # / imparte cu zecimale; / se executa inainte de adunare
5.0
>>> b / (2.0 + a)  #expresia dintre paranteze se executa mai intai
0.8
```

Formate de afisare numerice:

```
>>> 1 / 2  #neformatat
0.5
>>> n = 1 / 3
>>> n
0.3333333333333333
>>> print( n )  #cu print()
0.3333333333333333
```

```
>>> '%e' % n  #formatare cu %
'3.333333e-01'
>>> '%4.2f' % n
'0.33'
>>> '{0:4.2f}'.format( n ) #cu metoda
    format() a str
'0.33'
```

Comparatii

Formatele functiilor repr() si str():

```
>>> repr( 'spam' )        #repr() este folosita la afisarea obisnuita
"'spam'"
>>> str( 'spam' )         #str() este folosita de print() pentru afisare
'spam'
```

Comparatia (si inlantuita):

```
>>> 1 < 2  #mai mic decat
True
>>> 2.0 >= 1  #mai mare sau egal, mixt
True
>>> 2.0 == 2.0  #egalitate de valori
True
>>> 1.0 != 1.0  #inegalitate
False
```

```
>>> x = 2
>>> y = 3
>>> z = 4
>>> x < y < z  #inlantuire, echivalenta cu:
True
>>> x < y and y < z
True
```

Impartirea

```
>>> x < y > z  #si pt. rezultat False
False
>>> x < y and y > z
False
>>> 1 < 2 < 3.0 < 4  #inlantuiri, oricate
True
>>> 1 > 2 > 3.0 > 4
False
```

```
>>> 1 == 2 < 3  # 1 == 2 and 2 < 3 !!
False
>>> 1.1 + 2.2 == 3.3  #imprecizie!
False
>>> 1.1 + 2.2
3.3000000000000003
>>> int( 1.1 + 2.2 ) == int( 3.3 ) #OK cu
    conversie la int
True
```

Impartirea (3.X): clasica, “floor”, adevarata:

- **Operatorul /** face impartire cu zecimale
- **Operatorul //** trunchiaza rezultatul (“floor”), returnand intregi sau reali daca vreun operand este de tip real

floor() si trunc()…

```
>>> 10 / 4  #exct, cu toate zecimalele
2.5
>>> 10 / 4.0
2.5
>>> 10 // 4  #fara zecimale, trunchiaza
2
>>> 10 // 4.0  #rezultat de tip real
2.0
```

floor() vs. trunc() – conteaza pt. numere negative:

```
>>> import math
>>> math.floor( 2.5 ) #intregul cel mai
    apropiat si mai mic
2
>>> math.floor( -2.5 )
-3
>>> math.trunc( 2.5 )  #elimina
    zecimalele
2
>>> math.trunc( -2.5 )
-2
>>> 5 / 2, 5 / -2
(2.5, -2.5)
>>> 5 // 2, 5 // -2
(2, -3)
>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)
>>> 5 // 2.0, 5 // -2.0
(2.0, -3.0)
```

Precizie nelimitata, numere complexe

Intregi cu precizie nelimitata:

```
>>> 9999999999999999999 + 1
100000000000000000000
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376
```

Dezavantaj: viteza de executie pentru numere mari; *avantaj*: deja implementat.

Numere complexe:

```
>>> 1j * 1J
(-1+0j)
>>> 2 + 1j * 3
(2+3j)
>>> (2 + 1j) * 3
(6+3j)
```

Sunt de interes pentru domenii ingineresti sau stiintifice.

Se pot procesa cu modulul ***cmath***.

hex(), oct(), bin(), eval()

Reprezentari in hexa, octal, binar:

Valorile memorate sunt aceleasi, indiferent de reprezentare:

>>> **0o1, 0o20, 0o177 #octal**

(1, 16, 127)

>>> **0x01, 0x10, 0x7f #hexa**

(1, 16, 127)

>>> **0b1, 0b10000, 0b1111111 #binar**

(1, 16, 127)

Calculul in diverse baze de numeratie:

>>> **0x2f, (2 * (16 ** 1) + 15 * (16 ** 0))**

(47, 47)

Conversii intreg la string:

>>> **oct(64), hex(64), bin(64)**

('0o100', '0x40', '0b1000000')

Conversii string la intreg cu ***int()***:

>>> **int('64'), int('100', 8), int('40', 16), int('1000000', 2)**

(64, 64, 64, 64)

>>> **int('0x40', 16), int ('0b1000000', 2) #si cu prefix**

(64, 64)

Conversii string la intreg cu ***eval()***:

>>> **eval('0x40')**

64

eval(): mai lenta, interpreteaza strigurile ca fiind cod Python, periculos!

Operatii pe biti

Formatare cu format() si %:

```
>>> '{0:o}, {1:x}, {2:b}'.format( 64, 64, 64 )  #cu metoda format()
'100, 40, 1000000'
>>> '{:o}, {:x}, {:b}'.format( 64, 64, 64 )  #pozitia este superflua
'100, 40, 1000000'
>>> '%o, %x, %X' % (64, 255, 255)  #cu %
'100, ff, FF'
```

Aritmetica pe biti:

Se foloseste in domenii specializate, de ex. sisteme de operare, drivere, retea.

```
>>> x = 1
>>> x << 2  # deplasare stanga
4
>>> x | 2  # sau (1 | 2)
3
>>> x & 3  # si (1 & 3)
1
>>> bin( x | 0b0010 ) # inspectare cu bin()
'0b11'
```

Functii predefinite si module numerice

Functii predefinite:

```
>>> pow( 2, 5 ), 2 ** 5, 2.0 ** 5.0
(32, 32, 32.0)
>>> abs( -35 ), sum( (1, 2, 3, 4) ) #sum, arg iterabil!
(35, 10)
>>> min( 4, 1, 3, 2 ), max( 4, 1, 3, 2 )
(1, 4)
>>> round( 1.569 ), round( 1.469 ), round( 1.569, 2 )  #rotunjire cu 2 zecimale
(2, 1, 1.57)
```

Modulul *math*: import math

```
>>> math.pi, math.e
(3.141592653589793, 2.718281828459045)
>>> math.sin( math.pi / 2 )
1.0
>>> math.sqrt( 144 ), math.sqrt( 2 )
(12.0, 1.4142135623730951)
```

Cum se extrage radical, in trei moduri:

```
>>> math.sqrt( 1234567890123 )
1111111.1061109055
>>> 1234567890123 ** 0.5
1111111.1061109055
>>> pow( 1234567890123, 0.5 )
1111111.1061109055
```

Modulul *random*

Modulul *random*

Generarea de valori aleatorii intre 0 si 1:

```
>>> import random
>>> random.random()
0.7709076947019782
```

Generare intreg intre doua numere:

```
>>> random.randint( 1, 10 )
2
```

Alegere dintr-o secventa:

```
>>> random.choice( [ 'Luceafarul', 'La
steaua', 'Glossa' ] )
'Luceafarul'
```

Amestec aleatoriu:

```
>>> culori = [ 'inima', 'romb', 'trefla', 'pica' ]
>>> random.shuffle( culori )
>>> culori
['pica', 'trefla', 'inima', 'romb']
```

Poate fi folosit pentru simulari de calcule statistice, jocuri video, etc.

Sumar

- Numere, notiuni de baza
- Expresii numerice
- ***Alte tipuri numerice***
- Extensii numerice
- Tipuri dinamice in Python

Decimal, tip numeric

- Numerele decimale sunt numere reale cu un numar fix de zecimale; necesita importarea modulului ***decimal***

```
>>> 0.1 + 0.1 + 0.1 - 0.3          #imprecizie datorata procesorului/hardware
5.551115123125783e-17
>>> print( 0.1 + 0.1 + 0.1 - 0.3 )    #nici print() nu ajuta
5.551115123125783e-17
>>> from decimal import Decimal
>>> Decimal( '0.1' ) + Decimal( '0.1' ) + Decimal( '0.1' ) - Decimal( '0.3' ) #str la Decimal
Decimal('0.0')
```

Rezultatul este precis, cu numarul de zecimale determinat de numarul cu cele mai multe zecimale.

Decimal…

- Setarea globala a preciziei dorite:

>>> **Decimal(1) / Decimal(7) #nr. prea mare de zecimale la conversia de la real la decimal**

Decimal('0.142857142857142857142857142857')

>>> **import decimal**

>>> **decimal.getcontext().prec = 4 #patru zecimale**

>>> **Decimal(1) / Decimal(7)**

Decimal('0.1429')

- Operatii monetare:

>>> **1999 + 1.33**

2000.33

>>> **decimal.getcontext().prec = 2**

>>> **salariu = decimal.Decimal(str (1999 + 1.33))**

>>> **salariu**

Decimal('2000.33')

Decimal…

- Setarea temporara a preciziei dorite – cu instructiunea *with … as*:

```
>>> import decimal
>>> decimal.Decimal( '1.00' ) / decimal.Decimal( '3.00' )  #neconvenabil
Decimal('0.33333333333333333333333333333')
>>> with decimal.localcontext() as ctx:
        ctx.prec = 2
        decimal.Decimal( '1.00' ) / decimal.Decimal( '3.00' )

Decimal('0.33')
>>> decimal.Decimal( '1.00' ) / decimal.Decimal( '3.00' )
Decimal('0.33333333333333333333333333333')
```

Fraction, tip numeric

- Reprezinta numere rationale; se importa modulul *fractions*

```
>>> from fractions import Fraction    #se importa clasa (constructorul)
>>> x = Fraction( 1, 3 )
>>> y = Fraction( 4, 6 )
>>> x
Fraction(1, 3)
>>> y                                 #simplificare automata
Fraction(2, 3)
>>> print( y )
2/3
```

Fraction…

- Expresii matematice

```
>>> x + y
Fraction(1, 1)
>>> x - y
Fraction(-1, 3)
>>> x * y
Fraction(2, 9)
```

- Conversii *str* la *Fraction*

```
>>> Fraction( '.25' )
Fraction(1, 4)
>>> Fraction( '1.25' )
Fraction(5, 4)
>>> Fraction( '.25' ) + Fraction( '1.25' )
Fraction(3, 2)
```

Fraction…

- Precizie in fractii (si decimali); se evita inexactitatile hardware, se fac simplificari automate

```
>>> Fraction( 1, 10 ) + Fraction( 1, 10 ) + Fraction( 1, 10 ) - Fraction( 3, 10 )
Fraction(0, 1)
>>> 1 / 3                          #imprecis
0.3333333333333333
>>> Fraction( 1, 3 )               #precis
Fraction(1, 3)
>>> (1 / 3 ) + (6 / 12)            #imprecis
0.8333333333333333
>>> Fraction( 1 , 3 ) + Fraction( 6 , 12 )  #precis si simplificat
Fraction(5, 6)
```

Fraction…

- Conversii float <-> *Fraction*

```
>>> (2.5).as_integer_ratio()        #float la tuple
(5, 2)
>>> r = 2.5                         #float la Fraction
>>> f = Fraction( *r.as_integer_ratio() )      # * pentru expandare tuplu
>>> f
Fraction(5, 2)
>>> x, float( x )                   #Fraction la float
(Fraction(1, 3), 0.3333333333333333)
>>> Fraction.from_float( 1.75 )     #float la Fraction cu metoda from_float()
Fraction(7, 4)
```

- Conversii automate

```
>>> x + 2                           #Fraction + int  => Fraction
Fraction(7, 3)
```

Fraction…

```
>>> x + 2.0                          #Fraction + float => float
2.3333333333333335
>>> a = x + Fraction( *(4.0 / 3).as_integer_ratio() )
>>> a                                #imprecizie datorata conversiei de la float
Fraction(22517998136852479, 13510798882111488)
>>> a.limit_denominator( 10 )        #limitarea numitorului la valoarea maxima 10
Fraction(5, 3)
```

set

- **Definitie: o colectie neordonata de obiecte unice si fixe (imuabile) – multime.**
- **Reprezentari (3.X):**

```
>>> set( [1, 2, 3, 4] )
{1, 2, 3, 4}
>>> {1, 2, 3, 4}  #ca un dictionar fara valori, doar cu chei.
{1, 2, 3, 4}
>>> set( 'spam' )  #neordonat…
{'s', 'p', 'm', 'a'}
>>> s = {'s', 'p', 'm', 'a'}
>>> s
{'s', 'p', ‘a', ‘m'}
>>> s.add( 'ham' )
>>> s
{'m', 's', 'ham', 'a', 'p'}
```

Operatori pe multimi:

```
>>> s1 = { 1, 2, 3, 4 }
>>> s1 & { 1, 3 }  #intersectie
{1, 3}
>>> { 1, 5, 7, 3 } | s1  #reuniune
{1, 2, 3, 4, 5, 7}
>>> s1 - { 1, 3 }  #diferenta
{2, 4}
>>> s1 > { 1, 3 } #contine
True
```

Set…

Multimea vida:

>>> **s1 - { 1, 2, 3, 4 }**

set()

>>> **type({}) # {} rezervat pt. *dict* !**

<class 'dict'>

>>> **s = set() #initializare cu multimea vida**

>>> **s**

set()

Metode ale tipului set:

>>> **s.add(1.23) #metoda *add()***

>>> **s**

{1.23}

>>> **{ 1, 2, 3 }.union({ 3, 4 }) #union()**

{1, 2, 3, 4}

>>> **{ 1, 2, 3 }.union([3, 4]) #lista**

{1, 2, 3, 4}

>>> **{ 1, 2, 3 }.union(set([3, 4]))**

{1, 2, 3, 4}

>>> **{ 1, 2, 3 }.intersection((1, 3, 5)) #intersection(), tuplu**

{1, 3}

>>> **{ 1, 2, 3 }.issubset(range(-5, 5)) #issubset(), domeniul -5 la 5**

True

Limitari *set*

Limitari set, *frozenset*:

- **Tipurile modificabile, *list*, *dict* si *set* nu pot fi obiecte continute de un set**
- ***tuple* si *frozenset* (tip predefinit), da**

```
>>> s
{1.23}
>>> s.add( [1, 2, 3] )
TypeError: unhashable type: 'list'
>>> s.add( (1, 2, 3) )  #tuplu, da
>>> s
{1.23, (1, 2, 3)}
>>> s | { (1, 2, 3), (4, 5, 6) }  #reuniune cu operatorul |
{1.23, (4, 5, 6), (1, 2, 3)}
>>>
```

Utilizari *set*

- **Colectii iterative pentru *set***

```
>>> { x ** 2 for x in [1, 2, 3, 4] }    #multimea patratelor numerelor din lista
{16, 1, 4, 9}
>>> { x for x in 'spam' }               #iteratie pe un string
{'s', 'p', 'm', 'a'}
>>> s = { x * 4 for x in 'spam' }
>>> s | { 'mmmm', 'qqqq' }              #reuniune
{'pppp', 'qqqq', 'aaaa', 'ssss', 'mmmm'}
>>> s & { 'mmmm', 'qqqq' }              #intersectie
{'mmmm'}
```

- **Utilizari ale tipului *set***

Filtrarea duplicatelor dintr-o secventa:

```
>>> L = [1, 2, 1, 3, 1, 4, 4, 5]  #lista
>>> set( L )
{1, 2, 3, 4, 5}
```

Utilizari…

```
>>> L = list( set( L ) )
>>> L
[1, 2, 3, 4, 5]
```

Izolarea diferentelor intre iterabile (liste, stringuri):

```
>>> set( [1, 3, 5, 7] ) - set( [1, 2, 4, 5, 6] )     #intre liste
{3, 7}
>>> set( 'abcdefg' ) - set( 'abdfghj' )          #intre stringuri
{'e', 'c'}
>>> set( 'spam' ) - set( ['h', 'a', 'm' ] )          #intre string si lista
{'s', 'p'}
>>> set( dir( bytes ) ) - set( dir( bytearray ) )      #tot intre liste..
{'__getnewargs__'}
```

Utilizari…

Testarea egalitatii indiferent de ordine

- cu ***set***:

>>> **L1, L2 = [1, 3, 2, 4, 5], [2, 3, 1, 5, 4]**

>>> **L1 == L2**

False

>>> **set(L1) == set(L2)**

True

- cu functia predefinita ***sorted()***, mai costisitor:

>>> **sorted(L1) == sorted(L2)**

True

>>> **'spam' == 'maps', set('spam') == set('maps'), sorted('spam') == sorted('maps')**

(False, True, True)

Memorarea nodurilor deja vizitate dintr-un graf ciclic !

Utilizari…

Operatii asupra rezultatelor interogarii unei baze de date:

```
>>> ingineri = { 'ion', 'oti', 'ana', 'vic' }
>>> sefi = { 'dan', 'oti' }
>>> 'ion' in ingineri
True
>>> ingineri & sefi
{'oti'}
>>> ingineri | sefi
{'dan', 'oti', 'ion', 'vic', 'ana'}
>>> ingineri - sefi
{'vic', 'ion', 'ana'}
>>> sefi - ingineri
{'dan'}
>>> ingineri > sefi
False
>>> { 'ion', 'oti' } < ingineri
True
>>> (ingineri | sefi) > sefi
True
>>> sefi ^ ingineri  #sau exclusiv
{'vic', 'ana', 'dan', 'ion'}
>>> (sefi | ingineri) - (sefi ^ ingineri)
    #intersectie..
{'oti'}
>>> #Vezi diagrame Venn !
```

bool, tip numeric

- **Definitie: subclasa a tipului *int*, cu doua valori, *True* (1) si *False* (0)**
- **Exemple:**

```
>>> type( True )
<class 'bool'>
>>> isinstance( True, int )
True
>>> True == 1  #ca valoare, da
True
>>> True is 1  #ca obiecte, nu!
False
>>> True or False
True
>>> True + 4
5
```

Sumar

- Numere, notiuni de baza
- Expresii numerice
- Alte tipuri numerice
- ***Extensii numerice***
- Tipuri dinamice in Python

Extensii numerice

Numeric Python – NumPy:

- Ofera tipul matrice, procesare vectoriala, biblioteci de calcul stiintific
- Folosit in cercetare, la NASA, Los Alamos, etc.
- Este considerat mai puternic si flexibil decat Matlab

```
C:\Users\Dan>pip install numpy
Collecting numpy
  Obtaining dependency information for numpy from https://files.pythonhosted.org/packages/32/95/908d0ca
a051beae4f7c77652dbbeb781e7b717f3040c5c5fcaed4d3ed08f/numpy-1.26.1-cp312-cp312-win_amd64.whl.metadata
  Downloading numpy-1.26.1-cp312-cp312-win_amd64.whl.metadata (61 kB)
     ---------------------------------------- 61.2/61.2 kB 653.3 kB/s eta 0:00:00
Downloading numpy-1.26.1-cp312-cp312-win_amd64.whl (15.5 MB)
   ---------------------------------------- 15.5/15.5 MB 14.9 MB/s eta 0:00:00
Installing collected packages: numpy
Successfully installed numpy-1.26.1

[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\Dan>python
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct  2 2023, 13:03:39) [MSC v.1935 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>>
```

SciPy:

- vezi https://www.scipy.org/about.html

Sumar

- Numere, notiuni de baza
- Expresii numerice
- Alte tipuri numerice
- Extensii numerice
- ***Tipuri dinamice in Python***

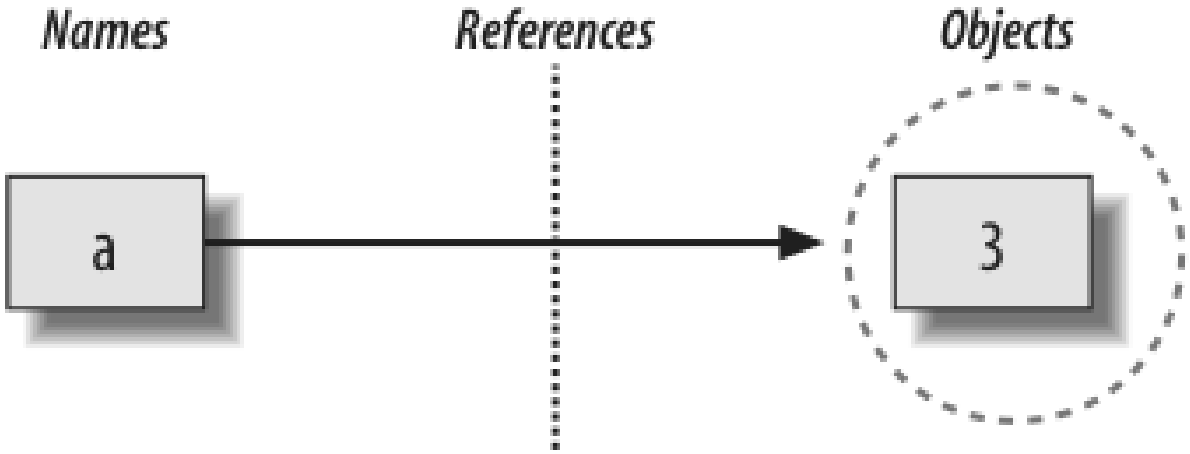
Tipuri Dinamice in Python

- In Python, tipurile de date sunt dinamice, variabilele nu se declara si deci acelasi program se poate executa in contexte diferite – polimorfism.
- Variabilele:
  - Sunt create prin asignarea unei valori.
  - Nu au tipuri asociate ci doar refera obiectele asignate.
  - Sunt inlocuite in expresii cu obiectul referit.
  - Trebuie asignate inaite de folosire.

Variabile si referinte la obiecte

>>> **a = 3** **#numele *a* este atribuit obiectului numeric *3***

- Numele de variabile se afla intr-o tabela, obiectele sunt alocate in memorie, referinta este un pointer catre memoria ocupata de obiect.



Tipurile obiectelor

```
>>> a = 3          #obiect de tip int
>>> a = 'spam'     #obiect de tip str
>>> a = 3.14       #obiect de tip float
```

- Variabila *a* refera, pe rand obiecte de tip diferit
- Tipul apartine obiectului (care contine un pointer catre obiectul int, str, float)
- Obiectele contorizeaza si numarul de referinte curent

Gestionarea memoriei

- Eliberarea memoriei alocata obiectelor care au contorul de referinte zero se face automat, cu tehnica numita *garbage collection*.

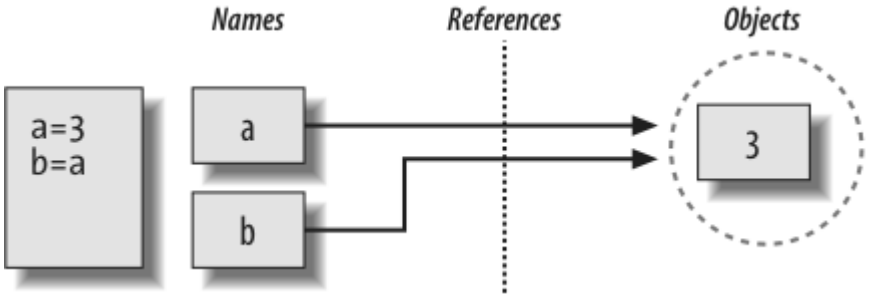
```
>>> x = 59
>>> x = 'spam'          #eliberare obiect 59
>>> x = 3.14            #eliberare obiect 'spam'
>>> x = [1, 2, 3, 4]    #eliberare obiect 3.14
```

- *Garbage collection* detecteaza si referintele ciclice – ex. L.append(L)

Referinte partajate

>>> a = 3 **#*b* refera acelasi obiect – *3* – ca *a***

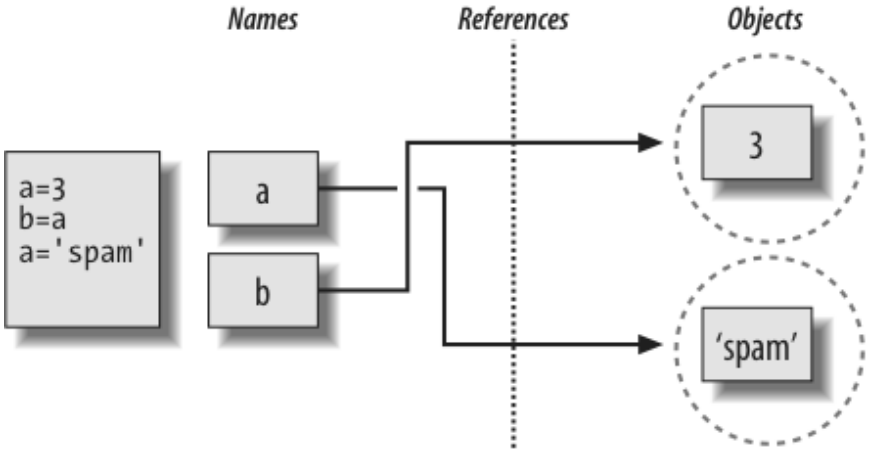
>>> b = a



>>> a = 3

>>> b = a **#*b* ramane neschimbat**

>>> a = 'spam' **#acum *a* refera alt obiect, 'spam'**



Obiecte modificabile

```
>>> L1 = [2, 3, 5]
>>> L2 = L1                 #L2 refera aceeasi lista ca L1
>>> L1[0] = 33              #Primul element din lista este modificat via variabila L1
>>> L1
[33, 3, 5]
>>> L2                      #atat L1 cat si L2 refera acelasi obiect
[33, 3, 5]
```

- Obiectele pot fi **copiate** in loc de a fi referite:

```
>>> L1 = [2, 3, 5]
>>> L2 = L1[:]  #copiere prin slicing
>>> L1[0] = 33
>>> L1
[33, 3, 5]
>>> L2  #neschimbata
[2, 3, 5]
```

Copierea se mai poate face cu:

```
>>> import copy    #modul generic
>>> x = copy.copy( y )
>>> z = copy.deepcopy( y )
```

== si *is*

```
>>> L = [1, 2, 3]
>>> M = L
>>> L == M     #cu ==, egalitate valorica
True
>>> L is M     #cu is, aceeasi referinta
True
>>> L = [1, 2, 3]
>>> M = [1, 2, 3]
>>> L == M     #aceeasi valoare
True
>>> L is M     #dar obiecte diferite
False
```

Observatie: intregii si stringurile mici sunt refolosite (cached):

```
>>> x = 33
>>> y = 33
>>> x == y
True
>>> x is y      #int-ul 33 este cached!
True
```

Referinte “slabe”

- Sunt implementate in modulul standard ***weakref***
- Astfel de referinte nu impiedica stergerea obiectelor – de catre *garbage collector*.
- Se folosesc in cache-uri – temporare – bazate pe dictionare de obiecte relativ mari (referinta din cache ar impiedica stergerea obiectului)