Authors Sonia Estradé
José M Gómez
Ricardo Graciani
Frank Güell
Manuel López
Xavier Luri
Josep Sabater

Tots els llenguatges de programació tenen tipus de dades bàsiques. Aquests tipus bàsics permeten enmagatzemar, manipular i intercanviar informació dins el programa o entre ells. Els tipus bàsics estan molt relacionats amb la CPU, i cada llenguatge intenta utilitzar-los tantes vegades com sigui possible. Això és el que fa Python.

D'altra banda, és necessari enmagatzemar la informació entre operacions en algun lloc, és el que s'anomena variables. Aquestes permeten enmagatzemar els resultats intermitjos mentre estem processant un algoritme.

Tipus numèrics

Aquests representen valors matemàtics, i s'utilitzen per fer càlculs. Els tipus utilitzats són:

Integer

S'utilitza per operar amb nombres sencers. Treballen amb una precisió infinita, és a dir, poden treballar amb qualsevol nombre de dígits. Exemples en són:

- Suma

```
In [1]: 1 + 1
```

```
Out[1]: 2
```

- Producte

```
In [2]: 3 * 5
```

```
Out[2]: 15
```

- Creació d'una variable sencera

```
In [1]: # Creant una variable sencera per testejar nombre sencers.
        integer_var = 4

        integer_var
```

```
Out[1]: 4
```

- Determinar el tipus d'una variable sencera

```
In [4]: type(integer_var)
```

```
Out[4]: int
```

```
In [5]: integer_var + 5
Out[5]: 9
```

Float

Són nombres de coma flotant, i s'utilitzen per representar nombres reals. Tenen una part sencera, una fraccional i exponent. Exemples en són:

- Crear una variable de coma flotant

```
In [6]: float_var = 2.1

        float_var
Out[6]: 2.1
```

- Crear una variable amb un nombre representat en notació exponencial

```
In [7]: float_exp = 6.023e23

        float_exp
Out[7]: 6.023e+23
```

- Determinar el tipus

```
In [8]: type(float_exp)
Out[8]: float
```

- Divisió de variables

```
In [9]: float_var/float_exp
Out[9]: 3.4866345674912835e-24
```

Complex

Els nombres complexes són natius a Python, i s'utilitzen com els altres tipus (`j` representa la part imaginària):

- Creant una variable

```
In [10]: complex_var = 1.5 + 0.5j
```

- Demanant la part real

```
In [11]: complex_var.real
Out[11]: 1.5
```

- Demanant la part imaginària

```
In [12]: complex_var.imag
Out[12]: 0.5
```

- Determinant el tipus

```
In [13]: type(1 + 0j)
Out[13]: complex
```

Important: Tingueu en compte que `1 + 0j` es pot considerar un nombre real, ja que no té part imaginària, Python considera que és complex.

Booleans

```
In [14]: 3 > 4
Out[14]: False
```

```
In [15]: test = (3 > 4)
```

```
In [16]: test
Out[16]: False
```

```
In [17]: type(test)
Out[17]: bool
```

Operadors numèrics

Així l'entorn Python pot ser la teva calculadora, amb les operacions aritmètiques bàsiques `+`, `-`, `*`, `/`, `%` (mòdul) i `**`(exponent) implementades de forma natural. Per `+`, `-`, `*` i `%` si el nombre del mateix tipus és utilitzat, els resultats són sempre del mateix tipus. Exceptuant `/` i `**` que cambiaran el tipus així que sigui necessari per representar el resultat esparat de forma correcte:

```
In [18]: 7/3
Out[18]: 2.3333333333333335
```

En aquest cas el resultat és de coma flotant, per donar el resultat que millor s'ajusta.

Si es requereix la divisió d'un sencer, es pot forçar utilitzant l'operador `//`:

```
In [19]: 7//3
Out[19]: 2
```

El remanent es pot calcular utilitzant l'operador mòdul:

In [20]:
```
7%3
```

Out[20]: 1

Com example, podem comprovar la fòrmula: $Dividend = Divisor \ quocient + remanent$

In [21]:
```
dividend = 7
divisor = 3

quotient = dividend//divisor

quotient
```

Out[21]: 2

In [22]:
```
reminder = dividend%divisor

reminder
```

Out[22]: 1

In [23]:
```
divisor*quotient+reminder
```

Out[23]: 7

El resultat és **ok**

Un exemple de cambiar el tipus d'un resultat amb l'operador `**`:

In [24]:
```
5**-2
```

Out[24]: 0.04

És posible operar amb diferents tipus:

- Sencer i coma flotant: resultat coma flotant

In [25]:
```
2 * 3e4
```

Out[25]: 60000.0

- Sencer i complex: resultat complex

In [26]:
```
integer_var * complex_var
```

Out[26]: (6+2j)

- Coma flotant i complex: resultat complex

In [27]:
```
8 * 3j
```

Out[27]: 24j

És possible forçar el tipus utilitzant un casting:

In [28]:
```
int(3.14)
```

Out[28]: 3

In [29]:
```
float(43)
```

Out[29]: 43.0

In [30]:
```
complex(3)
```

Out[30]: (3+0j)

In [31]:
```
complex(4.23)
```

Out[31]: (4.23+0j)

Però no és possible forçar el tipus d'un complex a un altre tipus numeric:

In [32]:
```
float(4 + 0j)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-567fbb0947cd> in <module>()
----> 1 float(4 + 0j)

TypeError: can't convert complex to float
```

In [33]:
```
int(2 + 0j)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-f5d8093ffa9e> in <module>()
----> 1 int(2 + 0j)

TypeError: can't convert complex to int
```

Si un complex s'ha de convertir al quadrat del valor absolut, i després llegir la part real:

In [34]:
```
complex_var
```

Out[34]: (1.5+0.5j)

Primer el conjugat complex del complex_var és calculat:

In [35]:
```
complex_var_conj = complex_var.conjugate()

complex_var_conj
```

Out[35]: (1.5-0.5j)

En aquest cas el conjugat té parèntesis **( )** que volen dir que és una funció associada amb el tipus de complex.

Aleshores el quadrat del valor absolut de la variable complexa és calculada
($|Complex|^2 = Complex \ Complex$):

```
In [36]: absolute_square_var = complex_var*complex_var.conjugate()

         absolute_square_var
Out[36]: (2.5+0j)
```

Aleshores la part real s'obté:

```
In [37]: absolute_square_var.real
Out[37]: 2.5
```

El resultat és ja un valor de coma flotant.

Cadenes

Les cadenes s'utilitzen per representar el text. Les diferents lletres es codifiquen en el que es coneix com *"Characters"*.

Creant una cadena

Differents sintàxis es poden utilitzar per crear una cadena:

- Cuota simple:

```
In [38]: s = 'Hello, how are you?'

         s
Out[38]: 'Hello, how are you?'
```

- Cuota doble:

```
In [39]: s = "Hi, what's up?"

         s
Out[39]: "Hi, what's up?"
```

- Cuota triple:

Permet expandir el text en diferents línies, però s'ha de tenir una cura especial amb el format:

```
In [40]: s = '''Hello,
             how are you?'''

         s
Out[40]: 'Hello,\n    how are you?'
```

En aquest cas, hem inclòs implícitament caràcters de format, com la nova línia `\n`. El resultat es pot veure utilitzant la funció `print` :

```
In [41]: print(s)
         Hello,
             how are you?
```

Els espais abans de *how* s'han inclòs automàticament, i molt probablament això no es volia. Un altre opció és eliminar els espais:

In [42]:
```
s = """Hi,
what's up?"""

print(s)
```

```
Hi,
what's up?
```

Però en aquest cas, el codi Python no és fàcil de llegir.

Tres vegades dobles cometes (""") i tres vegades cometes (''') són correctes sintàcticament a Python. Però, les dobles cometes són recomenades per documentació (també conegut com docstrings), encara que només sigui per una línia.

NOTA: Docstrings són comentaris inmediatament a continuació de la definició d'un mòdul, funció, classe o mètode, que expliquen el seu ús i propòsit.

També és posible incloure explícitament caràcters de format:

In [43]:
```
s = "Hi guys\nHow are you?\n\tWe are fine!!! Thanks!!!"

s
```

Out[43]:
```
'Hi guys\nHow are you?\n\tWe are fine!!! Thanks!!!'
```

En aquest cas, hem utilitzat la nova línia de caràcter `\n`, i el caràcter tabulat `\t`. El text resultant és:

In [44]:
```
print(s)
```

```
Hi guys
How are you?
        We are fine!!! Thanks!!!
```

I també dividim les cadenes entre línies:

In [45]:
```
s = "Hi, " \
    "what's up?"

print(s)
```

```
Hi, what's up?
```

I les barrejem:

In [46]:
```
s = "Hi,\n" \
    "what's up?"

print(s)
```

```
Hi,
what's up?
```

Indexant una cadena

Els diferents caràcters dins una cadena es poden accedir utilitzant el que s'anomena indexat:

```
In [47]: a = "hello"
```

L'índex s'introdueix dins els brackets **[]**:

```
In [48]: a[0]
Out[48]: 'h'
```

Important: El primer caràcter és el que està indexat 0. A Python, els índexs sempre comencen amb el valor 0.

```
In [49]: a[1]
Out[49]: 'e'
```

El darrer caràcter és el quart index:

```
In [50]: a[4]
Out[50]: 'o'
```

Si intentem accedir al cinqué index, el resultat pot ser un error:

```
In [51]: a[5]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last
)
<ipython-input-51-b6a934feab86> in <module>()
----> 1 a[5]

IndexError: string index out of range
```

Això indica que hem intentat accedir a un índex incorrecte.

És posible accedir al darrer caracter directament, només utilitzant l'índex -1.

```
In [52]: a[-1]
Out[52]: 'o'
```

Aquesta notació es pot utilitzar fins assolir l'índex 0, que en aquest cas seria el -5:

```
In [53]: a[-5]
Out[53]: 'h'
```

Questió: Que passaria si intentem accedir a l'índex -6?

Breument, l'índex negatiu correspond a contar des del final de la dreta.

Slicing

D'avegades volem llegir parts d'un text. Això es conegut com slicing:

```
In [54]: a = "hello, world!"
```

Per exemple, volem accedir als elements 3, 4, 5. Aleshores els hem d'escriure com:

```
In [55]: a[3:6]
Out[55]: 'lo,'
```

Important: El darrer índex és sempre el previ al requerit.

És posible preguntar la longitud de la sub-cadena:

```
In [56]: len(a[3:6])
Out[56]: 3
```

El primer índex i el darrer ho poden ser implícitament, si no estan inclosos:

```
In [57]: a[:4]
Out[57]: 'hell'
```

```
In [58]: a[4:]
Out[58]: 'o, world!'
```

Podem preguntar també els caràcters a les posicions senars, això es fa afegint uns altres dos punts i un nombre després, en aquest cas 2:

```
In [59]: a[::2]
Out[59]: 'hlo ol!'
```

O des del quart caràcter, cada tres:

```
In [60]: a[4::3]
Out[60]: 'owl'
```

Accents i caràcters especials també es poden fer a mà amb cadenes Unicode (vegeu https://docs.python.org/3.4/howto/unicode.html (https://docs.python.org/3.4/howto/unicode.html)).

Inmutabilitat

Una cadena és **objecte immutable** i no es poden modificar els seus continguts.

In [61]:
```
a = "hello, world!"

a
```

Out[61]:
```
'hello, world!'
```

In [62]:
```
a[2] = 'z'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-62-d57c4312feba> in <module>()
----> 1 a[2] = 'z'

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

Hom pot emperò crear noves cadenes des de l'original. Podem substituir la primera aparició d'un caràcter:

In [63]:
```
b = a.replace('l', 'z', 1)

b
```

Out[63]:
```
'hezlo, world!'
```

In [64]:
```
a
```

Out[64]:
```
'hello, world!'
```

O totes les aparicions:

In [65]:
```
a.replace('l', 'z')
```

Out[65]:
```
'hezzo, worzd!'
```

Les cadenes tenen moltes mètodes útils, tals com a `.replace` com hem vist a dalt. Recordeu la notació orientada a objecte `a.` i utilitzar el tabulador final o `help(str)` per cercar nous mètodes.

Python ofereix posibilitats avançades per manipular cadenes, buscant patrons o formatejant. Si esteu interessats: https://docs.python.org/3.4/library/stdtypes.html#text-sequence-type-str (https://docs.python.org/3.4/library/stdtypes.html#text-sequence-type-str) i https://docs.python.org/3.4/library/string.html#formatstrings (https://docs.python.org/3.4/library/string.html#formatstrings)

Conversió de cadenes

Les cadenes es poden convertir a d'altres tipus, al igual que els tipus numèrics:

In [66]:
```
int('1')
```

Out[66]:
```
1
```

Però han de seguir el format correcte:

In [67]:
```
int('1.')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-67-f18b6136395b> in <module>()
----> 1 int('1.')

ValueError: invalid literal for int() with base 10: '1.'
```

Altres exemples:

In [68]:
```
complex('3.+4j')
```

Out[68]:
```
(3+4j)
```

In [69]:
```
complex('3. + 4j')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-69-84986758f0fb> in <module>()
----> 1 complex('3. + 4j')

ValueError: complex() arg is a malformed string
```

Això ho podem solucionar utilitzant la funció de reemplaç:

In [70]:
```
complex('3. + 4j'.replace(' ', ''))
```

Out[70]:
```
(3+4j)
```

Nota: En aquest cas, hem aplicat la funció `replace` a la cadena directament, però també podem assignar la cadena a una variable i operar després. El darrer és el preferit.

Substitució de cadenes

És posible convertir una variable a una cadena directament:

In [71]:
```
str(4j)
```

Out[71]:
```
'4j'
```

Però el mètode preferit és utilitzant la substitució de la cadena i formatejant. Això té un millor control de la cadena resultant. Exemples bàsics en són:

In [72]:
```
'An integer: {}; a float: {}; another string: {}'.format(1, 0.1, 'string')
```

Out[72]:
```
'An integer: 1; a float: 0.1; another string: string'
```

In [73]:
```
i = 102
```

In [74]:
```
filename = 'processing_of_dataset_{}.txt'.format(i)
```

```
In [75]: filename
Out[75]: 'processing_of_dataset_102.txt'
```

Es pot aplicar un formateig més complexe, però en parlarem quan parlem d'entrades i sortides.

Dades binàries

Els ordinadors treballen sempre amb dades booleanes ('1' i '0'), també anomenats bits, i operacions booleanes('and', 'or' i 'not'), també anomenats operadors lògics. Com a resultat, per codificar nombres i cadenes, és necessari definir com agrupar els booleans en paquets que permetin codificar més informació.

Com exemple:

> 1 bit (booleà) pot enmagatzemar dos valors: ('1' i '0')
>
> 2 bits poden enmagatzemar quatre valors: ('11', '10', '01' i '00')
>
> 3 bits poden enmagatzemr vuit valors: ('111', '110', '101', '100', '011', '010', '001' i '000')

I així. El nombre de valors (o simbols) es pot determinar mitjançant la fòrmula:

$$m = 2^n$$

On n és el nombre de bits i m nombre de simbols que es poden representar.

Sencers

Ara, Tenim el problema de com codificar un sencer com un nombre binari. Això es pot fer fàcilment utilitzant el següent procediment amb un nombre com el 7. Primer el dividim per 2 (ja que podem codificar dos valors amb un bit):

$$\frac{7}{2} \quad 3$$

per tant

$$7 \quad 3 \; 2 + 1$$

El remanent és 1. Aquest nombre serà el bit menys significant o el bit 0. El 3 resultant ara s'haurà també de dividir:

$$\frac{3}{2} \quad 1$$

per tant

$$3 \quad 1 \; 2 + 1$$

Aquest remanent 1 serà el següent bit o bit 1. Ara podem dividir altre cop el resultat (1):

$$\frac{1}{2} \quad 0$$

per tant

$$1 \quad 0 \; 2 + 1$$

Aquest remanent 1 serà el següent bit o bit 2. El nombre resultant és:

k_2	k_1	k_0
1	1	1

Questió: Quina serà la representació binària de 5? i de 4?

Nombres majors poden ser representats utillitzant un nombre més gran de bits. Python permet calcular-los automàticament. Per exemple, el nombre 234 és representat pel nombre:

```
In [76]: bin(234)
Out[76]: '0b11101010'
```

El resultat és una cadena Python que inclou el prefix '0b' per indicar que és binari.

El valor en aquest cas s'ha de representar per un grup de, al menys, 8 bits:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	1	0	1	0	1	0

Un grup de 8 bits s'anomena byte, i pot representar:

$$(2^8 \quad 256)$$

El rang va del nombre 0 al 255. La dificultat major és que aquest nombres només són positius. Si el nombre és negatiu, Python afageix un signe abans del nombre per representar-lo. Per exemple:

```
In [77]: bin(-234)
Out[77]: '-0b11101010'
```

Questió: Quina serà la representació binària de -9? i de -15?

La representació d'aquesta codificació dins l'ordinador es basa en el complement a dos. Però no hi entrarem en aquesta assignatura.

Per simplificar la representació és posible utilitzar el formar hexadecimal. En aquest cas els bits s'agrupen de 4 en 4. Com 4 bits poden representar 16 simbols, els 10 digits no són suficients, per aquesta raó, s'afegeixen lletres:

dec	bin	hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f

Com a resultat, el nombre 234 és 0b11101010 i en hexadecimal és:

```
1110 is e

1010 is a
```

Per tant, 234 és 0xea (0x indica que és un nombre hexadecimal).

La funció hex fa la conversió:

```
In [78]: hex(234)
Out[78]: '0xea'
```

Nombres reals

En el cas de nombres reals o nombre de coma flotant, Python utilitza la representació IEEE 754 (http://en.wikipedia.org/wiki/IEEE_floating_point) . Aquest standard permet representar nombres entre:

$$\pm 4.9406564584124654 \ 10^{-324} \quad x \quad \pm 1.7976931348623157 \ 10^{308}$$

Un nombre menor que $\pm 4.9406564584124654 \ 10^{-324}$ és considerat 0 amb el mateix signe, i un major és considerat un infinit:

```
In [79]: -1e-325
Out[79]: -0.0
```

In [80]:
```
-1e309
```

Out[80]:
```
-inf
```

Els nombres també es poden mostrar en format hexadecimal utilitzant la funció **`float.hex()`**:

In [81]:
```
float.hex(1.4e5)
```

Out[81]:
```
'0x1.1170000000000p+17'
```

Noteu que l'exponent s'escriu en decimal enlloc de hexadecimal, i que dóna la potència de 2 per la cual es multiplica el coeficient. Per exemple, la cadena hexadecimal 0x1.117p+17 representa el nombre en coma flotant:

$$x = (1\ \ 16^{0} + 1\ \ 16^{-1} + 1\ \ 16^{-2} + 1\ \ 16^{-3})\ \ 2^{17} = 14000 = 1.4\ \ 10^{5}$$

Quan tots els digits després de la coma s'han multiplicat per la corresponent potència de 16. Aquesta notació només és utilitzada quan es requereix una elevada precisió en la representació. En els altres casos, es prefereix la notació decimal.

In [ ]:
```
```