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

Operacions i Codis d'estil

Operacions bàsiques (aritmètiques)

Python implementa les sis operacions aritmètiques bàsiques amb els simbols següents:

- Suma: +
- Resta: -
- Producte *
- Divisió: /
- Exponenciació: **
- Mòdul: % (resta de la divisió)

Aquestes opreracions requereixen dos operands i retornen un resultat numèric el tipus del qual depèn dels anteriors, com hem vist en el tema anterior. El programa següent conté exemples d'ús d'aquests operadors:

In [3]:
```
# Definim dues variables de treball
x = 10
y = 20

# Operacions aritètiques bàsiques
print(x+y) # Suma
print(x-y) # Resta
print(x*y) # Producte
print(x/y) # Divisió
print(x**y) # Exponenciació
print(y%x) # Mòdul
```

```
30
-10
200
0.5
10000000000000000000
0
```

Exercici: intenteu fer un programa que resolgui equacions de segon grau

$$ax^2 + bx + c = 0$$

Solucions:

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \qquad x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

Nota: per calcular arrels quadrades en Python useu l'expressió d'aquestes com una potència $\sqrt{x}$ $x^{\frac{1}{2}}$

Exercici: una implementació alternativa de les solucions d'una equació de segon grau és la solució quadràtica (http://mathworld.wolfram.com/QuadraticEquation.html):

$$q = -\frac{1}{2}[b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}]$$

$$x_1 = \frac{q}{a}$$

$$x_2 = \frac{c}{q}$$

Aquesta solució és numèricament més estable quan s'usen valors de coma flotant. Implementeu-la i compareu els resultats amb l'anterior.

Particularitats de la divisió d'enters

Quan s'usa l'operador de divisió "/" amb enters el resultat en general no és un enter. En aquest cas Python converteix el resultat en un *float* en comptes de retornar un enter.

In [2]:
```
x = 1
y = 2
print(x/y)
```

```
0.5
```

Nota important: en versions anteriors de Python el comportament de la divisió d'enters no era aquest. El resultat era un enter i podieu trobar casos com `1/2 = 0`. Aneu amb compte amb aquesta peculiaritat si useu versions anteriors de Python o altres llenguatges de programació com Java

Experimenteu amb la divisió i els altres operadors amb els vostres propis exemples

Operacions addicionals

El signe "-" davant d'un valor serveix per canviar-li el signe

In [7]:
```
x = 10
y = -x
print(y)
```

```
-10
```

L'operador "//" és la **divisó d'enters**, en aquest cas Python sempre arrodoneix el resultat al valor inferior més proper.

In [9]:
```
# Divisió normal
a = 1.
b = 2.
c = a/b
print(c)
print(type(c))

d= a//b
print(d)
print(type(d))
```

```
0.5
<class 'float'>
0.0
<class 'float'>
```

Operacions a nivell de bit

Aquestes operacions treballen amb les variables tractant el seu contingut a nivell de bit. Solen usar-se en programes que treballen a baix nivell, tot i que ocasionalment poden ser útils per a programes científics.

- and: &
- or: |
- xor: ^
- not: ~
- Desplaçament esquerra: <<
- Desplaçament dreta: >>

And: realiza un "and" logic bit a bit

0 & 0	= 0
0 & 1	= 0
1 & 0	= 0
1 & 1	= 1

In [11]:
```
a = 1 # 00000001
b = 1 # 00000001
c = 2 # 00000010
print(a & b) # 00000001
print(a & c) # 00000000
```

```
1
0
```

Or: realitza un "or" logic bit a bit

0 | 0	= 0
0 | 1	= 1
1 | 0	= 1
1 | 1	= 1

In [12]:
```
a = 1 # 00000001
b = 1 # 00000001
c = 2 # 00000010

print(a | b) # 00000001
print(a | c) # 00000011
```

```
1
3
```

Xor: realitza un "or" exclusiu bit a bit

0 ^ 0	= 0
0 ^ 1	= 1
1 ^ 0	= 1
1 ^ 1	= 0

In [13]:
```
a = 1 # 00000001
b = 1 # 00000001
c = 2 # 00000010

print(a ^ b) # 00000000
print(a ^ c) # 00000011
```

```
0
3
```

Not: fa una negació bit a bit d'un valor

~0	= 1
~1	= 0

Noteu que en aquest exemple es veu l'efecte de la representació dels enters en complement a 2

In [14]:
```
a = 1 # 00000001
b = 2 # 00000010

print(~a) # 11111110
print(~b) # 11111101
```

```
-2
-3
```

Desplaçament: desplaçen *n* bits a dreta o esquerra. Els bits de l'extrem es perden.

In [15]:
```
a = 1 # 00000001
b = 2 # 00000010

print(a << 1) # 00000010
print(b >> 1) # 00000001

print(a << 2) # 00000100
print(b >> 2) # 00000000
```

```
2
1
4
0
```

Operacions amb cadenes (strings)

És important tenir en compte que les operacions anteriors no es poden aplicar a cadenes (valors tipus *string*), encara que el seu contingut sigui la transcripció d'un número. Podeu veure com l'exemple següent dóna un error quan s'executa:

In [19]:
```
cadena = "1.0"

print(cadena-1)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call last
)
<ipython-input-19-787a2117f039> in <module>()
      1 cadena = "1.0"
      2
----> 3 print(cadena-1)

TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

Si teniu una cadena que conté un text que representa un número i voleu fer operacions amb ell, cal transformar la cadena abans de poder fer-ho:

In [21]:
```
cadena="1.0"
valor= float(cadena)

print(valor-1)

cadena2= "1"
valor2= int(cadena2)
print(valor2-1)
```

```
0.0
0
```

Aneu amb compte però, per que si la cadena no representa correctament un valor numèric **simple** si intenteu transformar-la us donarà un error.

In [22]:
```
cadena= "1+2"
valor= int(cadena)
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call last
)
<ipython-input-22-bb26b4d61d03> in <module>()
      1 cadena= "1+2"
----> 2 valor= int(cadena)

ValueError: invalid literal for int() with base 10: '1+2'
```

Tanmateix, l'operador suma si que es pot aplicar a cadenes, però Python no l'interpreta com a una operació aritmètica sinó com una concatenació:

In [23]:
```
cadena1= "ABC"
cadena2= "DEF"

print(cadena1+cadena2)
```

```
ABCDEF
```

De forma similar, l'operador multiplicació aplicat a cadenes s'inerpreta com una repetició:

In [25]:
```
cadena= "12345 "

print(cadena*5)
```

```
12345 12345 12345 12345 12345
```

Operacions booleanes (lògiques)

Permeten fer comparacions de valors o avaluació de condicions més complexes sobre conjunt de valors. El resultat és un valor booleà *True* o *False*.

Seran especialment útils quan introduïm les comandes de control de flux. Permetran condicionar l'execució d'un programa en funció dels valors dels seus resultats. De moment aprendrem com funcionen.

Operadors de comparació

Operador	Funció
x == y	**Igualtat:** retorna 'True' si el valor x és igual al valor y. Altrament retorna 'False'.
x != y	**Desigualtat:** retorna 'False' el valor x és igual al valor y. Altrament retorna 'True'.
x > y	**Major:** retorna 'True' si el valor x és més gran que el valor y. Altrament retorna 'False'.
x < y	**Menor:** retorna 'True' si el valor x és més petit que el valor y. Altrament retorna 'False'.
x >= y	**Major o igual:** retorna 'True' si el valor x és més gran o igual que el valor y. Altrament retorna 'False'.
x <= y	**Menor o igual:** retorna 'True' si el valor x és més petit o igual que el valor y. Altrament retorna 'False'.

In [27]:
```
a = 1
b = 1
c = 2

print("a == b",a==b)
print("a != b",a!=b)
print("a == c",a==c)
print("a != c",a!=c)
print("a > b",a>b)
print("a >= b",a>=b)
print("a < c",a<c)
print("a <= c",a<=c)
```

```
a == b True
a != b False
a == c False
a != c True
a > b False
a >= b True
a < c True
a <= c True
```

Operadors booleans

Serveixen per combinar diverses expressions lògiques donant al final un valor *True* o *False*. Permeten formar condicions complexes a partir de diverses comparacions individuals.

Operador	Funció
and	**I lògic:** retorna *True* si els dos valors booleans als que s'aplica són *True*. Altrament retorna *False*.
or	**O lògic:** retorna *True* si almenys un dels dos valors booleans als que s'aplica és *True*. Altrament retorna *True*.
not	**Negacio:** canvia el valor booleà al que s'aplica pel seu oposat

And

True **and** True	True
True **and** False	False
False **and** True	False
False **and** False	False

In [28]:
```
a = 0
b = 1

# Expressions boleanes: and
print("a==0 and b==1", a==0 and b==1)
print("a==0 and b==0", a==0 and b==0)
print("a>=0 and a<=2", a>=0 and a<=2)
print("a>=0 and a<=2 and b>1", a>=0 and a<=2 and b>1)
```

```
a==0 and b==1 True
a==0 and b==0 False
a>=0 and a<=2 True
a>=0 and a<=2 and b>1 False
```

Or

True **or** True	True
True **or** False	True
False **or** True	True
False **or** False	False

In [29]:
```
a = 0
b = 1

# Expressions boleanes: or
print("a==0 or b==1", a==0 or b==1)
print("a==0 or b==0", a==0 or b==0)
print("a>=0 or a<=2", a>=0 or a<=2)
print("a==2 or b==2", a==2 or b==2)
print("a>=0 or a<=2 or b>1", a>=0 or a<=2 or b>1)
```

```
a==0 or b==1 True
a==0 or b==0 True
a>=0 or a<=2 True
a==2 or b==2 False
a>=0 or a<=2 or b>1 True
```

not

not True	False
not False	True

In [30]:
```
a = 0
b = 1

# Expressions boleanes: not
print("not a==0", not a==0)
print("not a==1", not a==1)
```

```
not a==0 False
not a==1 True
```

Notes:

- **or** aquest operador només avalua el segón argument si el primer és fals (operador curt-circuit)
- **and** aquest operador només avalua el segón argument si el primer és cert (operador curt-circuit)
- **not** té menys prioritat que els operadors no booleans, de manera que *not a == b* s'interpreta com *not (a == b)*, i *a == not b* és un error de sintaxi

Operacions compostes: compte amb l'ordre d'avaluació!

Els operadors de comparació i booleans es poden combinar per formar sentències lògiques complexes, però **cal anar amb compte amb l'ordre d'avaluació per què pot canviar el resultat**. Es poden usar parèntesis per especificar com s'ha d'avaluar l'expressió.

In [31]:
```
a = 0
b = 1

# Compte amb l'ordre d'avauluació dels operadors
print("not a==1 and b==1",not a==0 and b==0)
print("not (a==1 and b==1)",not (a==0 and b==0))
print("(not a==1) and b==1",(not a==0) and b==0)
```

```
not a==1 and b==1 False
not (a==1 and b==1) True
(not a==1) and b==1 False
```

Codis d'estil

En aquest document llistem les diferents convencions utilitzades en Python a l'hora de programar. Més informació es pot trobar en les Python Developer's Guide (https://www.python.org/dev/peps/pep-008/). Python és l'exemple més clar a l'hora d'implementar el que es coneix com ***programació estructurada***.

La programació estructurada és un paradigma de programació orientada a millorar la claretat, qualitat i temps de desenvolupament en la realització d'un programa. Programar de forma estructurada facilita la lectura de codi i la seva reutilització.

Formateig del codi

Identació

La identació consisteix en introduïr ***quatre espais*** per tal de delimitar l'estructura del programa, permetent establir blocs de codi. La identació en Python és **obligatòria**. Un codi no identat genera el següent error:

```
    File "<ipython-input-1-35a3266bacb1>", line 3
      print(n)
          ^
IndentationError: expected an indented block
```

Però, que vol dir la sentència *delimitar l'estructura del programa, permetent establir blocs de codi*?

El significat d'aquesta sentència és que un determinat conjunt d'instruccions que trobem identat pertany o s'executa depenent del que hi hagi a la instrucció anterior. Veiem un exemple:

```
contador = 0
variable = 1
mentre contador < 5
    Si variable < 3 llavors
        variable = variable + 1
        contador = contador + 1
    En cas contrario
        variable = 1
        print(variable)
```

Tal i com es veu en aquest pseudocodi, les dues comparacions s'executaran sempre que el valor de la variable `contador` sigui menor que 5. Dintre d'aquesta estructura tenim dues comparacions: `Si variable < 3 llavors` i `En cas contrari`. El codi que ve a continuació, només s'executarà si la comparació és correcta.

Mida màxima de la línia

La mida màxima de les línies ha de ser com a molt de 79 caràcters.

In [19]:
```
print('introdueix el valor màxim per executar la \
seqüència anterior')
```

```
introdueix el valor màxim per executar la seqüència anterior
```

Es pot fer servir la barra invertida (\) per indicar que la línia que estem llegint és la mateixa

Importació de llibreries

Quan es volen importar dos o més llibreries, haurien de col·locar-se en linies diferents:

Correcte

```
import sys
import os
```

Incorrecte

```
import sys, os
```

Tot i que sí que és correcte importar parts d'una llibreria d'aquesta forma:

```
from llibreria import A, B
```

Els imports s'han de col·locar sempre en la part superior de l'arxiu o la cel·la on estem treballant, just després de qualsevol comentari o cadena de documentació del codi i just abans de la definició de variables i constants.

Heu de garantir que feu servir les llibreries que importeu.

Comentaris

Un comentari és una construcció del llenguatge de programació destinada a introduïr anotacions llegibles al programador en el codi font de un programa informàtic. Aquestes anotacions són útils pels programadors ja que donen informació de que, quan i com s'executa un determinat codi. Són per tant afegits usualment amb el propòsit de fer el codi font més fàcil d'entendre pel que fa al manteniment o reutilització.

Per aquest motiu, un comentari que contradiu el codi és molt pitjor que no tenir cap comentari. Els comentaris han de ser frases completes i a ser possible en anglès, per afavorir que pugui ser llegit per qualsevol persona independentment de la seva nacionalitat.

Tenim dos tipus de comentaris:

- comentaris de bloc. Són aquells que s'apliquen al codi que es troba a continuació, i s'identen al mateix nivell que el codi que s'està comentant. Cada línia d'un comentari de bloc comença amb un #.
- comentaris de línia. És un comentari que es troba en la mateixa línia on tenim una sentència. Els comentaris en línia haurien de separar-se com a mínim per dos espais de la sentència que comenten. Haurien de començar amb un # i un espai.

Tot seguit veiem exemples de comentaris de bloc i comentaris de línia

In [17]:
```
# Aquest codi calcula el temps
# i la velocitat final quan tenim
# una caiguda lliure i mostra el
# resultat en la consola

g = -9.8
alcada = float(input("introduïr l'alçada"))
v0 = float(input("Introduïr la velocitat inicial"))
v = (v0**2 - 2*g*alcada)**0.5
t = (-v0 + (v0**2 -2*g*alcada)**0.5)/alcada    #compte amb els valors i els signes per evitar nombres complexos
print("la velocitat final és ", v," i el temps és ", t)
```

```
introduïr l'alçada10
Introduïr la velocitat inicial3
la velocitat final és  14.317821063276353  i el temps és  1.13178210632763
54
```

Utilitzeu els comentaris en línia de forma moderada. Aquests comentaris no són útils si indiquen coses òbvies i només serveixen per distreure al programador. No cal fer coses com aquestes:

```
x = x + 1    # incrementa el valor de x en 1
```

Cadenes de documentació

Les cadenes de documentació o ***docstrings*** permeten documentar funcions o classes per tal de conèixer que és i que fan aquests elements. Un docstring s'inicia introduïnt tres cometes (simples o dobles), el text i finalitza de nou amb tres cometes.

Així, donada una funció:

```
def Funcio():
    """
    Aqui introduïm el comentari
    que considerem oportú, definint
    les característiques, el que fa
    i com ho fa
    """
    return 0
```

Posteriorment, quan demanem informació sobre les funcions i/o classes s'obtindrà aquesta informació

In [18]:
```
def Funcio():
    """
    Funció exemple on introduïm les
    característiques principals.
    Normalment s'inclouen els paràmetres
    d'entrada i els paràmetres
    de sortida
    """
    return 0
help(Funcio)
```

```
Help on function Funcio in module __main__:

Funcio()
    Funció exemple on introduïm les
    característiques principals.
    Normalment s'inclouen els paràmetres
    d'entrada i els paràmetres
    de sortida
```

Definició de variables i funcions

Definiu el nom de les variables de manera que indiquin el seu significat. Si la definició està formada per varies paraules, La primera paraula escriviu-la tota en minúscules i, tot seguit, sense espais, les següents paraules separades amb guió baix (_).

Per exemple:

Correcte

```
velocitat = 0
velocitat_inicial = 4    #variable definida en m/s
acceleracio = 2         # acceleracio en m/s2
```

incorrecte

```
a = 0
b = 4
c = 2
```

Fer servir variables com a, b ó c no donen informació, dificultant després la intepretació i l'enteniment del codi. No feu servir accents ni d'altres signes gràfics que puguin donar errors sintàctics.

Pel que fa a les funcions, es defineix també amb minuscules separades amb guió baix (_) i preferentment amb un nom que indiqui l'acció. Així tindrem:

```
def modificar_variable(velocitat):
    return velocitat*2
```

Igual que en el cas de les variables, no feu servir accents ni d'altres signes gràfics, ja que donaran errors sintàctics.

Definició de constants

A diferència d'altres llenguatges de programació, on es pot definir explícitament una constant, en Python no es poden definir. Simplement el que hem de fer és crear la variable i no modificar-la. Com podem saber que el que estem utilitzant és una constant? La regla d'estil ens diu que quan tenim una variable, la definim tota en majúscules. En cas que la definició estigui formada per més d'una paraula, separarem les paraules per un guió baix (_).

Exemple:

In [16]:
```
# ***********************
# Definició de constants
# ***********************
GRAVETAT = 9.8                  # en m/s2
K_BOLTZMANN = 1.380648E-23      # en J/K

# ***********************
# Definició de variables
# ***********************
acceleracio = 1
temperatura = 300
velocitat_inicial = 0
distancia = 0
```

In [ ]:
```

```