Introducció al Python

Python

`www.python.org`
Python és un llenguatge de programació genèric de gran difusió que pot ser usat en qualsevol tipus de tasca que no requereixi accés directe al hardware del sistema o processament en temps real o que impliqui el desenvolupament i manteniment d'una gran infrastructura software per part de molta gent. La principal causa d'aquestes limitacions (i també dels seus avantatges en el altres àmbits) és que té un sistema dèbil de control de la semàntica estàtica del codi.
És un llenguatge simple, amb una corba d'aprenentatge ràpida, i disposa d'un gran nombre de mòduls que fan possible la seva aplicació a molt diverses àrees d'aplicació, des del càlcul científic al desenvolupament de plataformes web.
Python és un llenguatge interpretat. Això vol dir que les intruccions que el programador escriu no són intruccions que la plataforma que el conté pugui entendre directament i per tant necessiten ser traduides per un altre programa a instruccions comprensible per la màquina. Aquesta característica el fa flexible.

Elements Bàsics

Programa Un programa en Python és una seqüència de definicions i comandes.

Comanda Una comanda és una instrucció directa a l'interpret per fer alguna cosa. El procés per determinar el resultat d'una comanda es diu **avaluació**.

Literal Un literal és una entitat el valor de la qual és ella mateixa i que per tant no cal avaluar. `3` és un literal.

Objectes Els objectes són les entitats bàsiques que manipula Python. Tots els objectes tenen un **tipus** que defineix quines operacions podem aplicar-lis. Els tipus poden ser *escalars* o *no escalars*. Els primers són indivisibles, els segons tenen estructura interna.

Tipus escalars Python té varios tipus escalars:

`int` S'usa per representar els nombres enters: `3`, `1245` o `-23`.

`float` S'usa per representar els nombres reals: `3.0`, `1245.2325` o `-23.56`.

`bool` S'usa per representar valors els valors Booleans `True` o `False`.

`None` És un tipus amb un únic valor.

Operadors Els operadors ens permeten combinar objectes, formant **expressions**, cada una de les quals denota un objecte, anomenat **valor**, d'un determinat tipus. L'expressió `3 * 2` denota l'objecte `5` de tipus `int`. Els operadors dels tipus `int` i `float` són:

`i+j` S'usa per representar la suma. Si tant `i` com `j` són `int`, el resultat és `int`. Si algun dels dos és `float`, el resultat és `float`.

`i-j` S'usa per representar la resta. Si tant `i` com `j` són `int`, el resultat és `int`. Si algun dels dos és `float`, el resultat és `float`.

`i*j` S'usa per representar el producte. Si tant `i` com `j` són `int`, el resultat és `int`. Si algun dels dos és `float`, el resultat és `float`.

`i//j` S'usa per representar la divisió entera (retorna el quocient i s'oblida del reste): el valor de `6//2` és `3` i el valor de `6//4` és `1`.

`i/j` S'usa per representar la divisió.

`i%j` S'usa per representar la resta (o mòdul) de la divisió entera de `i` per `j`.

`i**j` S'usa per representar i^j. Si tant `i` com `j` són `int`, el resultat és `int`. Si algun dels dos és `float`, el resultat és `float`.

`>,<,>=,<=, ==, !=` Són els operadors de comparació, el resultat dels quals és de tipus `bool`.

Tots aquests operadors segueixen l'ordre de precedència habitual.

Els operadors del tipus `bool` són:

`a and b` Si tant `a` com `b` són `True`, el resultat és `True`. Si algun dels dos és `False`, el resultat és `False`.

`a or b` Si `a` o `b` són `True`, el resultat és `True`. Si els dos són `False`, el resultat és `False`.

`not a` Si `a` és `True`, el sesultat és `False`. Si `a` és `False`, el resultat és `True`.

Variables Les variables, en Python, són un mecanisme per associar un nom a un objecte. Considerem aquestes comandes:

```
pi = 3.14
radi = 11.2
area = pi * (radi ** 2)
```

El que fa Python és, primer, associar els noms `pi` i `radi` a dos objectes de tipus `float` i després associar el nom `area` a un altre objecte de tipus `float`. Si a continuació escrivim:

```
radi = 0.0
```

Python desfà l'associació anterior del nom `radi` i l'associa a un objecte diferent de tipus `float`. En Python **una variable és només un nom**. Un objecte pot tenir un nom, diversos noms, o cap nom associat amb ell.

Els noms de les variables poden estar formats per lletres majúscules i minúscules, dígits (tot i que no poden estar el principi), i el caràcter _. Les variables `jordi` i `Jordi` són noms diferents. Hi ha un conjunt de paraules que no es poden usar com a noms de variables perquè estan reservades: `and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, with, while` i `yield`.

Comentaris Podem posar comentaris sobre el codi que escrivim usant el símbol `#` a principi de linia:

```
# Calcul de l'area d'un cercle
pi = 3.14
radi = 11.2
area = pi * (radi ** 2)
```

Assignació múltiple Python ens permet assignar noms als objectes en *paral·lel*:

```
x, y = 2, 3
x, y = y, x
```

Si executem aquestes intruccions, el contingut final a `x` és `3` i a `y` és `2`.

El tipus `Str`

El tipus no escalar `Str` serveix per representar seqüències de caràcters o *strings*. Els literals de tipus `Str` es poden escriure de dues maneres: `'abcd'` o `"abcd"`. El literal `'1'` representa el caràcter i no el nombre!
La longitud d'un *string* es pot saber amb la funció `len`: `len('abc')` és `3`.
Hi ha alguns operadors numèrics que es poden aplicar també a aquest tipus. Aquesta abstracció s'anomena **sobrecàrrega**. Considerem:

```
b,c = 'a' + 'a', 'a'*3
```

Llavors, el contingut de `b` és `'aa'` i el de `c` és `'aaa'`.

Indexació La indexació és l'operació que ens permet extreure caràcters individuals d'un *string*. Per exemple, `'abc'[0]` és `'a'` i `'abc'[2]` és `'c'`. És important observar que donat un string `s` la primera posició és `0` i l'última `len(s)-1`. Si escrivim `'abc'[3]` és produirà un error per voler accedir a una posició inexistent. Els indexos negatius s'interpreten en ordre invers: `'abc'[-1]` és `'c'`.

Slicing L'*slicing* és l'operació que ens permet extreure *substrings* de qualsevol mida. Si `s` és un string, `s[start:end]` denota el substring que comença a la posició `start` i acaba a `end-1`: `'abc'[1:3]` és `'bc'`.

Input Python usa `input` per obtenir dades del teclat de l'ordinador. Aquesta funció fa que el programa s'aturi fins que l'usuari introdueix un string pel teclat de l'ordinador i tracta l'entrada com un string.

Condicionals

Fins ara hem vist seqüències linials d'instruccions, que són executades per l'interpret Python una darrera l'altra. Si volem un esquema d'execució condicional, en arbre, necessitem especificar tres parts:

1. Un test (que és una expressió de tipus `bool`).
2. Un bloc de codi que s'executi quan l'expressió prengui el valor `True`.
3. Un bloc de codi, opcional, que s'executi quan l'expressió prengui el valor `False`.

Les instruccions que implementen aquest esquema s'anomenen **condicionals** i tenen aquesta forma:

```
if a > 3.0:
    b = 0.0
else:
    b = True
```

Python usa l'estructura visual del codi (definida per les identacions de cada línia) com a part de la seva semàntica. Concretament, el codi que s'executa quan l'expressió booleana pren un determinat valor ha d'estar en un nivell superior d'identació que el test. L'identació de Python són 4 caràcters en blanc.
Aquesta regla es pot aplicar a múltiples nivells:

```
if x\%2 == 0:
    if x\%3 == 0:
        print('Divisible␣per␣2␣i␣per␣3')
    else:
        print('Divisible␣per␣2␣i␣no␣per␣3')
elif x\%3 == 0:
    print('Divisible␣per␣3␣pero␣no␣per␣2')
else:
    print('No␣es␣divisible␣ni␣per␣2␣ni␣per␣3')
```

Iteracions

El mecanisme d'iteració es pot implementar de dues formes en Python: mitjançant la instrucció `while` i mitjançant la instrucció `for`.

`while` Aquesta instrucció ens iterarà mentre es compleixi una condició:

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

Aquest codi imprimeix els nombres de `0` a `3`.

for Aquesta instrucció ens iterarà un nombre concret de vegades o fins que troba una instrucció `break`:

```
x = 4
for i in range(0, x):
    print(i)
```

Aquest codi imprimeix els nombres de `0` a `3`.

Càlcul numèric

La manipulació d'objectes de tipus `float` i l'aplicació repetida de càlculs han de ser tractats amb compte. Considerem aquest codi:

```
x = 0.0
for i in range(10):
    x = x + 0.1
print(x == 1.0)
```

El resultat de la seva execució és `False`. Per entendre el motiu, cal entendre com es representen els nombres binaris.

Representació binària d'un nombre enter Un nombre binari es representa per una seqüència de dígits que poden ser `0` o `1`, anomenats bits. El bit de més a la dreta representa el valor 2^0, el següent 2^1 i així successivament. Per tant, el nombre binari `101` correspon a $1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 5$.

Representació binària d'un nombre real Tots els llenguatges de programació representen els nombres reals amb la representació anomenada *punt flotant*, que correspon al producte dels dígits significatius del nombre per un exponent. En el cas binari tant els dígits significatius com l'exponent s'escriuen en forma binària. El problema és que hi ha nombres que no es poden representar de forma exacta (també passa en base 10), com $\frac{1}{10}$.En Python, hi ha 53 bits de precisió, i per tant el nombre `0.1` seria:

`1100110011001100110011001100110011001100110011001100110011001`,

que equival al nombre:

`0.1000000000000000055511151231578270211815834045410156 25`.

Quan en el programa anterior comparavem `x` amb `1.0`, el resultat era fals perque la suma binaria de les diferents representacions de `0.1` no sumen `1.0`.

Aquest fet és important quan fem tests d'igualtat. Sempre que tinguem valors flotants, enlloc de fer `x == y` és millor usar aquesta expressió d'igualtat (aproximada): `abs(x-y) < 0.000001`.

Funcions

En teoria, amb els *nombres, assignacions, input, comparacions i iteracions*, podem implementar ja qualsevol programa. Però no és pràctic. Una de les possibilitats addicionals que proporciona qualsevol llenguatge de programació és la d'agrupar instruccions en *funcions*.

Definició En Python, una funció es defineix de la següent manera:

```
def max(x,y):
    if x > y:
        return x
    else:
        return y
```

`def` és una paraula reservada que indica que anem a definir una funció. Seguidament escrivim el nom que volem donar a la funció. La seqüència de noms que apareixen entre parèntesi després del nom de la funció són els **paràmetres**, que prenen valors en el moment que cridem la funció: `max(3+2,4)`. Seguidament ve el cos de la funció degudament identat. Si volem que la funció retorni valors, hem de posar-hi la instrucció `return` i l'expressió que volem que retorni.

Execució Què passa quam cridem una funció? Primer, s'avaluen les expressions corresponents als paràmetres. Després, el punt d'execució de l'interpret es dirigeix a la primera línia de la funció i s'executa el codi de la funció fins que trobem un `return` o no queden instruccions per executar-se. En aquest darrer cas la funció retorna el valor `None`. Per últim, el punt d'execució torna a la instrucció posterior a la crida de la funció.

Arguments En Python hi ha dues maneres de passar arguments: *posicionalment* o mitjançant *paraules clau*. Suposem aquesta funció:

```
def printName(firstName, lastName, reverse):
    if reverse:
        print(lastName + ',' + firstName)
    else:
        print(firstName, lastName)
```

Podem cridar la funció *posicionalment*,

`printName('Jordi', 'Vitria', True)`,

i l'interpret assignarà valors als paràmetres en funció de l'ordre, o mitjançant *paraules clau*,

`printName('Jordi', reverse = False, lastName = 'Vitria')`,

on usa els noms.

Scope Aquest concepte correspon a l'**espai de noms** que pot veure una funció. En principi, una funció pot veure els noms associats als seus paràmetres i els de les variables definides en el seu interior. Tots aquests només exisiteixen quan la funció s'està executant.

En Python, les funcions també són objectes i per tant es poden tractar com qualsevol altre objecte: tenen tipus, poden apareixer en expressions, poden ser elements de llistes o es poden passar com a paràmetres d'una altra funció.

Mòduls

Un mòdul és un fitxer `.py` que conté definicions i instruccions Python. Un programa pot accedir a un mòdul amb la instrucció `import` seguida del nom del mòdul.
Cada mòdul té la seva pròpia taula de noms. Per accedir-hi, podem usar la notació `module.name`.
Si executem la instrucció `from module import *` el que estem fent és importar la taula de noms i llavors podem accedir-hi directament.
Les instruccions d'un mòdul s'executen la primera vegada que s'importa el mòdul dins del programa, i només la primera vegada. Si volem que s'executi una altra vegada, podem usar la instrucció `reload(module)`.

Fitxers

Per poder guardar els resultats o les dades d'un programa hem d'usar fitxers, que són dades localitzades en el disc de l'ordinador i que tenen un nom. En Python ho podem fer així:

```
a = open('llistat.txt','r')
```

Amb això indiquem que volem llegir el fitxer `llistat.txt`, que anomenarem `a`. Si hi volguéssim escriure:

```
a = open('llistat.txt','w')
```

Les instruccions més comuns quan tractem fitxers són:
`open('fileName', 'w')`: crea un fitxer per escriure-hi.
`open('fileName', 'r')`: obre un fitxer existent per llegir-lo.
`f.read()`: retorna un string amb els continguts del fitxer `f`.
`f.readline()`: retorna la següent linia del fitxer `f`.
`f.readlines()`: retorna una llista on cada element és una linia del fitxer `f`.
`f.write(s)`: escriu l'string `s` al final del fitxer `f`.
`f.close()`: tanca el fitxer. Fins que no tanquem el fitxer, cap altre programa pot accedir-hi!

Tipus estructurats

Tuples Les tuples són seqüències no mutables (un cop definides no es poden cambiar) d'objectes arbitraris. Es defineixen amb parèntesis: `a = (1, 'jordi', 3.0)`. Les tuples es poden concatenar, indexar i fer-hi *slicing*. La instrucció `for` pot iterar pels elements d'una tupla.

Llistes Les llistes són seqüències mutables d'objectes arbitraris. Els mètodes més comuns són:

`L.append(e)`: afegeix un objecte `e` al final de la llista `L`.
`L.count(e)`: retorna el nombre de vegades que hi ha l'objecte `e` a la llista `L`.
`L.insert (i, e)`: inserta l'objecte `e` a la posició `i` de la llista `L`.
`L.extend(L1)`: afegeix els ítems de la llista `L1` al final de la llista `L`.
`L.remove(e)`: esborra la primera aparició de l'element a la llista `L`.
`L.index(e)`: retorna l'índex de la primera aparició de l'element a la llista `L`.
`L.pop(i:)` elimina i retorna l'element de la posició `i`.
`L.sort()`: ordena els elements de la llista `L`.
`L.reverse()`: aplica una ordenació inversa als elements de la llista `L`.

Clonatge Donada una llista `L1`, la instrucció `L2 = L1` no crea una còpia de la llista, sinó que li assigna un segon nom. Per fer una còpia cal fer això: `L2 = L1(:)`.

```
L1= L2[:]
```

Comprensió de llistes És una forma d'especificar l'aplicació d'una operació a tots els elements d'una llista. Crea una nova llista.

```
L = [x**2 for x in range(1,7)]
```

Operacions en strings, tuples i llistes Les opracions més comuns són:

`seq [i]`: retorna l'ièssim element de la seqüència..
`len(seq)`: retorna la longitud de la seqüència.
`seql + seq2`: concatena dues seqüències.
`n * seq`: retorna una seqüència on `seq` es repeteix `n` vegades.
`seq[start:end]`: retorna un *slice* de la seqüència.
`e in seq`: fa un test sobre el contingut de l'element `e` a la seqüència.
`for e in seq`: itera sobre els elements de la seqüència.

Diccionaris Són uns objectes semblants a les llistes, amb la diferència que els indexos, que s'anomenen **claus**, no han de ser enters, sinó que poden ser valors de qualsevol tipus no mutable:

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3}
```

Podem fer comprensions amb diccionaris:

```
a = {n: n*n for n in range(7)}
# a -> {0:0, 1:1, 2:4, 3:9, 4:16, 5:25, 6:36}
odd_sq = {n: n*n for n in range(7) if n%2}
# odd_sq -> {1:1, 3:9, 5:25}
```

Hi ha dues maneres d'iterar un diccionari:

```
for key in dictionary:
    print(key)
for key, value in dictionary.items():
    print(key,value)
```

Les operacions més importants sobre diccionaris són:

`len(d)`: retorna el nombre d'ítems a `d`
`d.keys()`: retorna una llista amb les claus de `d`.
`d.values()`: retorna una llista amb els valors de d.

`k in d`: retorna `True` si la clau `k` és a `d`.
`d[k]`: retorna el valor que té la clau `k`
`d[k] = v`: associa el valor `v` amb la clau `k`. Si ja hi havia alguna cosa associada a `k` reemplaça el valor.
`del d[k]`: elimina el contingut de la clau `k` de `d`.
`for k in d`: itera sobre les claus de `d`.

Conjunts Els conjunts són col·leccions mutables, no ordenades, d'objectes no mutables.

Podem crear un conjunt sense cap element, `buit = set()` (compte: {} crea un diccionari buit, no un conjunt buit), definir-lo, `c = (1,True,'a')` o convertir una llista:

```
L = [1,2,3]
c = set(L)
```

També podem fer comprensions amb conjunts:

```
s = {e for e in 'ABCHJABDC' if e not in 'AB'}
# --> set(['H', 'C', 'J', 'D'])
s = {(x,y) for x in range(-1,1) \
                for y in range (-1,1)}
# --> set([(0, -1), (0, 0), (-1, 0), (-1, -1)])
```

La forma d'iterar sobre un conjunt és:

```
for item in set:
    print(item)
```

I la de buscar un element:

```
if item in set:
    print(item)
if item no in set:
    print('error')
```

.

Les operacions més importants sobre conjunts són:

`len(s)`: retorna el nombre d'elements del conjunt.

`s.add(item)`: afegeix un element al conjunt.

`s.remove(item)`: elimina un element del conjunt. Si no hi és, genera un error.

`s.discard(item)`: elimina un element del conjunt si hi és.

`s.pop()`: retorna i elimina un element aleatori. Si el conjunt és buit, genera un error.

`s.clear()`: elilmina tots els elements del conjunt.

`s.union(o[, ...])`: retorna la unió dels conjunts.

`s.intersection(o)`: retorna la intersecció dels conjunts.

`s.difference(o)`: retorna els elements que són a `s` però no a `o`.