07 - Funcions

October 20, 2015

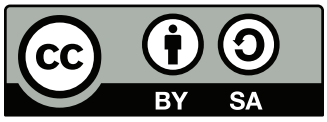


Figure 1: BY-SA

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

1 Funcions

Les funcions permeten la reutilització de codi: escriure codi que fa una tasca definida i que es pot utilitzar moltes vegades. D'aquesta manera el codi es pot fer més compacte (evitant repeticions tedioses) i més legible.

1.1 Definició d'una funció

Per a implementar una funció en un programa Python cal:

1. Una línia on es defineix la funció usant la paraula clau **def** seguida del nom de la funció
2. Una descripció de la funció, marcada entre triples cometes
3. Les sentències que composen la funció. **Important:** les línies de codi de la sentència han d'estar indentades per indicar que pertanyen a la funció. Si s'acaba la indentació s'enten que s'acaba la funció

   def Name(Parameters): ‘” docstring ‘” Statements return

El nom pot ser una paraula o un grup de paraules. L'única restricció és que no pot ser Python reserved words.

```
identifier ::=  (letter|"_") (letter | digit | "_")*
letter     ::=  lowercase | uppercase
lowercase  ::=  "a"..."z"
uppercase  ::=  "A"..."Z"
digit      ::=  "0"..."9"
```

Docstring i return no són necessaris.

1.2 Cridant una funció

Definint una funció només dóna el seu nom, especifica els paràmetres que s'han d'inclure a la funció i estructura els blocs de codi. Pots executar-la cridant-la des d'una altra funció or directament des de prompt.

Exemple:

```
In [1]: def my_function(s):
            print("My string: {}".format(s))

        my_function("Hello")
```

```
My string: Hello
```

Si volem dibuixar un quadrat amb una tortuga:

```
In [2]: import turtle

        def drawSquare():
            wn = turtle.Screen()
            alex = turtle.Turtle()

            alex.forward(50)
            alex.right(90)
            alex.forward(50)
            alex.right(90)
            alex.forward(50)
            alex.right(90)
            alex.forward(50)
            alex.right(90)

        drawSquare()

        turtle.mainloop()
```

Hem dibuixat el quadrat, però que passa si volem dibuixar el quadrat i després fer una nova acció. En aquest cas podem passar la tortuga a paràmetre:

```
In [3]: import turtle

        def drawSquare(t):
            t.forward(50)
            t.right(90)
            t.forward(50)
            t.right(90)
            t.forward(50)
            t.right(90)
            t.forward(50)
            t.right(90)
```

```
In [4]: wn = turtle.Screen()
        alex = turtle.Turtle()

        drawSquare(alex)

        turtle.mainloop()
```

Ara podem crear una figura geomètrica:

```
In [5]: wn = turtle.Screen()
        alex = turtle.Turtle()

        drawSquare(alex)
        alex.right(45)
        drawSquare(alex)
        alex.right(45)
        drawSquare(alex)
        alex.right(45)
        drawSquare(alex)
        alex.right(45)
        drawSquare(alex)
        alex.right(45)
        drawSquare(alex)
        alex.right(45)
        drawSquare(alex)
        alex.right(45)
        drawSquare(alex)
        alex.right(45)

        turtle.mainloop()
```

Amb la definició de funció `drawSquare` no podem canviar la mida. Per solucionar-ho podem afegir un nou paràmetre:

```
In [6]: def drawSquare(t, size):
            t.forward(size)
            t.right(90)
            t.forward(size)
            t.right(90)
            t.forward(size)
            t.right(90)
            t.forward(size)
            t.right(90)

        wn = turtle.Screen()
        alex = turtle.Turtle()

        smallSquareSize = 80
        largeSquareSize = 120

        drawSquare(alex, smallSquareSize)
        alex.right(45)
        drawSquare(alex, largeSquareSize)
        alex.right(45)
        drawSquare(alex, smallSquareSize)
        alex.right(45)
        drawSquare(alex, largeSquareSize)
        alex.right(45)
        drawSquare(alex, smallSquareSize)
        alex.right(45)
        drawSquare(alex, largeSquareSize)
        alex.right(45)
        drawSquare(alex, smallSquareSize)
        alex.right(45)
```

```
drawSquare(alex, largeSquareSize)
alex.right(45)

turtle.mainloop()
```

Ara podem crear una espiral augmentant la mida cada vegada que cridem la funció `drawSquare`:

```
In [7]: import math

        wn = turtle.Screen()
        alex = turtle.Turtle()

        squareGain = math.sqrt(2)-1
        squareSize = 20

        drawSquare(alex, squareSize)
        alex.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(alex, squareSize)
        alex.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(alex, squareSize)
        alex.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(alex, squareSize)
        alex.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(alex, squareSize)
        alex.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(alex, squareSize)
        alex.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(alex, squareSize)
        alex.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(alex, squareSize)
        alex.right(45)

        turtle.mainloop()
```

És important tenir en compte que la definició de funció s'ha d'haver fet abas d'usar-la. Sinó no es pot uitlitzar:

```
In [8]: helloWorld()
```

```
        def helloWorld():
            print("Hello World!!!")
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-8-638c477440db> in <module>()
    ----> 1 helloWorld()
          2
          3 def helloWorld():
          4     print("Hello World!!!")


        NameError: name 'helloWorld' is not defined
```

El que diu la funció no es pot executar a no ser que la funció sigui cridada.

```
In [9]: def helloWorld():
            print("Hello World!!!")

In [10]: helloWorld()

Hello World!!!
```

Una funció també pot cridar altres funcions com en els exemples `drawSquare()`. També podem crear una funció per crear espirals quadrades:

```
In [11]: def squaredSpiral(t, size):
             squareGain = math.sqrt(2)-1
             squareSize = size

             drawSquare(alex, squareSize)
             t.right(45)

             stepSize = squareSize*squareGain
             squareSize = squareSize + stepSize
             drawSquare(t, squareSize)
             t.right(45)

             stepSize = squareSize*squareGain
             squareSize = squareSize + stepSize
             drawSquare(t, squareSize)
             t.right(45)

             stepSize = squareSize*squareGain
             squareSize = squareSize + stepSize
             drawSquare(t, squareSize)
             t.right(45)
```

```
        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(t, squareSize)
        t.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(t, squareSize)
        t.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(t, squareSize)
        t.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(t, squareSize)
        t.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(t, squareSize)
        t.right(45)

        stepSize = squareSize*squareGain
        squareSize = squareSize + stepSize
        drawSquare(t, squareSize)
        t.right(45)
```

```
In [12]: import turtle

         wn = turtle.Screen()
         alex = turtle.Turtle()

         squaredSpiral(alex, 20)

         turtle.mainloop()
```

Quan s'analitza un programa, no s'hauria de llegir des de dalt cap a baix. S'ha de seguir el flux de l'execució.

També és important saber que és possible tenir qualsevol nombre de paràmetres, però amb cura.

1.3 Sentència return

Una funció pot retornar un resultat. Per això és necessari afegir la sentència `return` :

```
In [13]: def parabola(x, a, b, c):
             y = a*x**2 + b*x + c

             return y

         parabola(5, 1, 3, 2)
```

```
Out[13]: 42
```

```
In [14]: result = parabola(5, 1, 3, 2)
         second_value = result * 4
         print(second_value)
```

```
168
```

```
In [15]: def first_function():
             y = 1*5**2 + 2*5 + 3
             return y

         result = first_function()
```

```
In [16]: def second_function(result):
             second_value = result * 4
             print(second_value)

             second_function(result)
         152
```

```
Out[16]: 152
```

Una funció acaba quan la sentència return és trobada, o quan la indentació finalitza. Sino es troba la sentència `return` una funció donarà `None`.

`None` és freqüentment utilitzat a python per representar la absència d'una valor.

1.4 Com podem saber que fa una funció?

Primer cal posar un nom clar. però no és suficient. Per aquesta raó el que hem d'incloure és un comentari just després de la declaració. Això s'anomena `docstring`:

```
In [17]: def myFunction(parameter0, parameter1):
             """My function is used to perform a sequence of statements.
             The parameters that it uses are:
             parameter0: It is of type real and gives the...
             parameter1: It is a string that describes...
             My function returns a value of type int..."""
             result = 0
             print("My nice function...")

             return result
```

Al notebook, aquesta informació és donada posant el nom de la funció seguit del interrogant.

```
In [18]: myFunction?
```

Python també mostrarà aquest docstring si uitlitzem la funció help:

```
In [19]: help( myFunction )
```

```
Help on function myFunction in module __main__:

myFunction(parameter0, parameter1)
    My function is used to perform a sequence of statements.
    The parameters that it uses are:
    parameter0: It is of type real and gives the...
    parameter1: It is a string that describes...
    My function returns a value of type int...
```

```
In [20]: def displaceAndTurn(t, distance, angle):
             '''Function that based on the turtle (t) moves a distance (distance) and
             afterwards rotates an angle (angle). It provides no result'''
             t.forward(distance)
             t.left(angle)
```

```
In [21]: import turtle

         wn = turtle.Screen()
         alex = turtle.Turtle()

         displaceAndTurn(alex, 100, 90)

         turtle.mainloop()
```

```
In [22]: import turtle

         wn = turtle.Screen()
         alex = turtle.Turtle()

         displaceAndTurn(alex, 100, 90)
```

```
In [23]: displaceAndTurn(alex, 30, 60)
```

```
In [24]: turtle.mainloop()
```

1.5 Identificació de funcions

La identificació de funcions és un aspecte important de la programació, i una de les tasques més confuses. En aquesta secció, intentarem definir una estratègia per identificar les funcions i millorar la codificació.

La estratègia proposada segueix una aproximació de dalt a baix i de baix a dalt. El primer pas és identificar el problema a solucionar. Usualment un problema és massa complexe per ser solucionat en un pas. Per aquesta raó, el problema s'ha de dividir en petits problemes fins que puguin ser solucionats de cop.

Aquest procés és equivalent a solucionar un problema matemàtic. Si hem de trobar la solució d'un problema, primer analitzarem la informació de la seva expressió. D'acord amb aquesta informació analitzarem quins passos o tasques són necessaris per solucionar el problema. Si una tasca o pas és massa gran, l'haurem de subdividir en problemes més petits. Finalment tots els subproblemes seran a un nivell atòmic, això vol dir que seran suficientment simples per ser solucionats mitjançant eïnes que tinguem a mà. Això s'anomena aproximació des de dalt a baix.

A continuació començarem a construir la solució solucionant cada subproblema atòmic, i donant els resultats als següents passos o tasques. Això ens permetrà trobar las solució final. Això és una aproximació des de baix a dalt, on construirem la solució des dels fonaments.

És important tenir en compte que ambdues aproximacions usualment no es fan una després del altre, ja que són barrejades, a vegades un problema que sembla ser atòmic s'ha de subdividir altre cop. I també podem haver llibraries que donaran solucions a un problema.

Considerarem cada subproblema una funció. Per tant una funció pot ser una extensió d'una funció si soliciona un problema atòmic, o una funció nòdul si soluciona un problema utilitzant altres funcions. Com a resultat, solucionar un problema requereix un arbre de funcions que intercanvii la informació per assolir el resultat desitjat.

Un exemple podria ser dibuixar una dummy.

1.5.1 Definició del problema

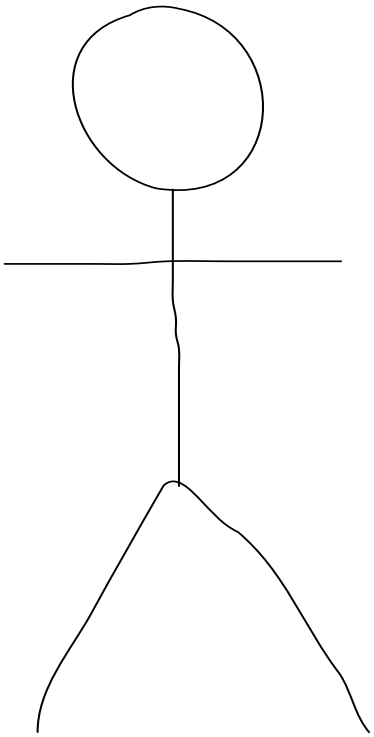
Com hem dit volem dibuixar un dummy, per tant hem de definir la funció `dummy()`. Hem de pensar també amb els arguments. Com a primera aproximació posarem la tortuga. A continuació pensarem quines tasques ha de fer. Com a primera aproximació dibuixarem un dummy basic:

In [25]:
```
import turtle

def dummy(t):
    '''dummy draws a dummy on the screen using the turtle t.'''
    # TODO: draw the dummy
    pass #Indicates that nothing has to be done
```

In [26]:
```
from IPython.display import SVG
SVG('dummy.svg')
```

Out[26]:



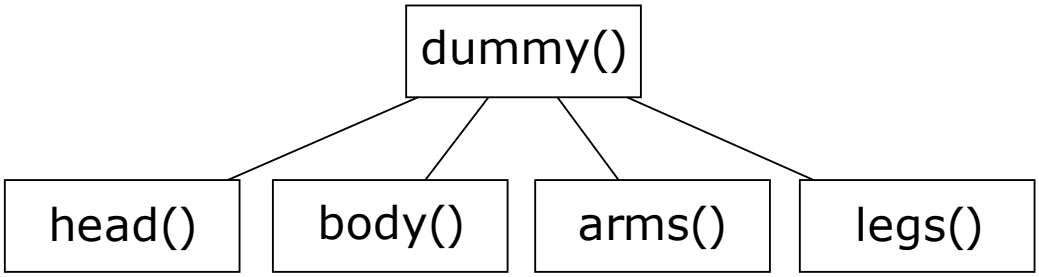
Podem veure que està composta per un cap fet amb un cercle, el cos que és una línia, les braços dibuixats amb una sola línia, i les cames utilitzant una 'v' invertida:

- Head
- Body
- Arms
- Legs

La funció dummy esdevé un arbre:

In [27]:
```
SVG('dummy_tree.svg')
```

Out[27]:



Els arguments tindran en compte la tortuga i la mida, per assegura una correcta relació:

```
In [28]: def head(t, size):
             '''head draws a head.'''
             # TODO: draw the head
             pass

         def body(t, size):
             '''head draws a body.'''
             # TODO: draw the body
             pass


         def arms(t, size):
             '''head draws a head.'''
             # TODO: draw the arms
             pass


         def legs(t, size):
             '''head draws a head.'''
             # TODO: draw the legs
             pass
```

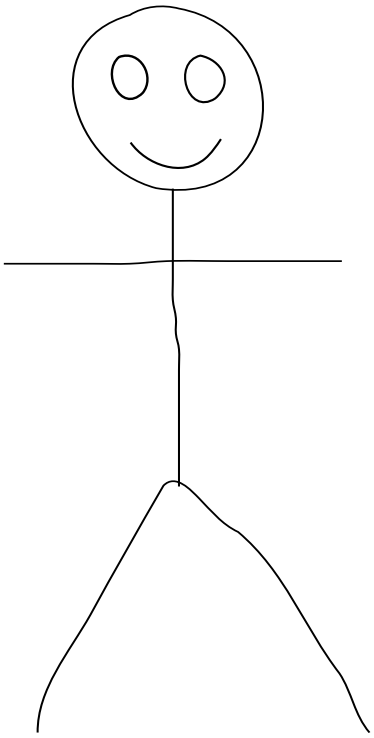
Ara hem d'omplir les funcions per dibuixar els diferents elements.

Problema: omple la funció per dibuixar els diferents elements.
Nota: Per simplificar, dins la funció dummy mou la tortuga cap al punt inicial on les subtasques han de començar. Dins les funcions cridades fes els moviments necessaris, i recorda de tornar a la posició inicial.

Ara, imaginem que volem dibuixar un somriure al dummy inicial:

```
In [29]: SVG('smiling_dummy.svg')

Out[29]:
```



Per fer-ho inclourem noves funcions dins la funció `head()` :

- Eyes
- Smile

Problema: escriu les funcions que permetran dibuixar una cara somrient.

Seguint aquest procés, podem millorar el dummy fins assolir el nivell de detall desitjat. També podem afegir l'argument mida a la funció dummy, i calcular les mides d'altres elements proporcionalmet.

En resum, el procés de solucionar un problema en un ordinador és un procés iteratiu, quant el problema és solucionat in passos diferents seguint els processos de dalt a baix i de baix a dalt per assolir la qualitat desitjada.

1.6 Arguments de la funció

Hem vist la manera més simple de definir els arguments d'una funció en python:

```
def drawSquare(t):
```

La funció té un nombre fixe d'arguments que s'han de donar tots en el mateix ordre quan siguin cridats. Aquests s'anomenen **Required positional arguments**.

```
In [30]: # Function definition is here
         def printme(sentence):
            "This prints a passed string into this function"
            print(sentence)
            return

         # Now you can call printme function
         printme('Hello World!')

         # Now you can call printme function
         printme()
```

```
Hello World!
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-30-1a586cecd3a4> in <module>()
          9
         10 # Now you can call printme function
    ---> 11 printme()


        TypeError: printme() missing 1 required positional argument: 'sentence'
```

1.6.1 Arguments Keyword

D'avegades és convenient passar els arguments en un ordre different al utilitzat en la definició de funció. Això es pot fer identificant els arguments amb els noms dels paràmetres. Per exemple:

```
In [31]: # Function definition is here
         def printme(sentence1, sentence2):
            "This prints a passed string into this function"
            print(sentence1, sentence2)
            return

         # Now you can call printme function
         printme('Hello World!', 'Hey again!')
         print()
         printme(sentence2 = 'Hello World!', sentence1 = 'Hey again!')
```

```
Hello World! Hey again!

Hey again! Hello World!
```

1.6.2 Arguments per defecte

Un argument per defecte és un argument que assumeix un valor per defecte si el valor no és donat en cridar la funció per aquest argument. Si qualsevol dels paràmetres formals en la definició de funció són dclarats amb el format "arg = value," aleshores tindràs la opció de no especificar un valor per aquests arguments en cridar la funció. Si no especifiques un valor, aleshores aquest paràmetre tindrà el valor per defecte donat quna s'executi la funció.

El següent exemple dóna una idea dels arguments per defecte, imprimirà una edat per defecte si no s'ha superat:

```
In [32]: # Function definition is here
         def printinfo( name, age = 35 ):
            "This prints a passed info into this function"
            print("Name: ", name)
            print("Age ", age)
            return

         # Now you can call printinfo function
         printinfo("miki", 50)
         printinfo("miki")
```

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Els arguments per defecte s'han de definir després dels arguments de posició que s'ha de donar sempre en cridar una funció. En cridar el keyword d'un codi, els arguments el poden usar per donar els arguments de posició i defecte.

```
In [33]: # Now you can call printinfo function
         printinfo(age=50, name="miki")
         printinfo(name="miki")
```

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Nota: Arguments no-keyword s'han de donar sempre abans de qualsevol argument keyword.

```
In [34]: printinfo(name="miki", 40)
```

```
  File "<ipython-input-34-fd2f475e39fa>", line 1
    printinfo(name="miki", 40)
                           ^
SyntaxError: non-keyword arg after keyword arg
```

Un ús correcte dels arguments simplifica la reutilització de codi.

1.7 Variables locals i globals

Les variables definides dins d'una funció només són disponibles dins la funció:

```
In [35]: def test_function():
             a = 3

         test_function()

         print(a)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-35-bd3b9212ddb6> in <module>()
  4 test_function()
  5
----> 6 print(a)


NameError: name 'a' is not defined
```

D'altra banda, les variables definides fora d'una funció es poden veure dins una funció:

```
In [36]: b = 5
         def print_b():
             print(b)

         print_b()
```

```
5
```

Però no poden ser modificades:

```
In [37]: c = 3
         def modify_c():
             c = 6
             print(c)
         modify_c()
         print(c)
```

```
6
3
```

Si és necessari modificar una variable global, la funció ha d'incloure la sentència global:

```
In [38]: d = 4
         def modify_global_d():
             global d
             d = 7
             print(d)

         modify_global_d()
         print(d)
```

```
7
7
```

La utilització de global és requerida per minimitzar el risc d'errors.