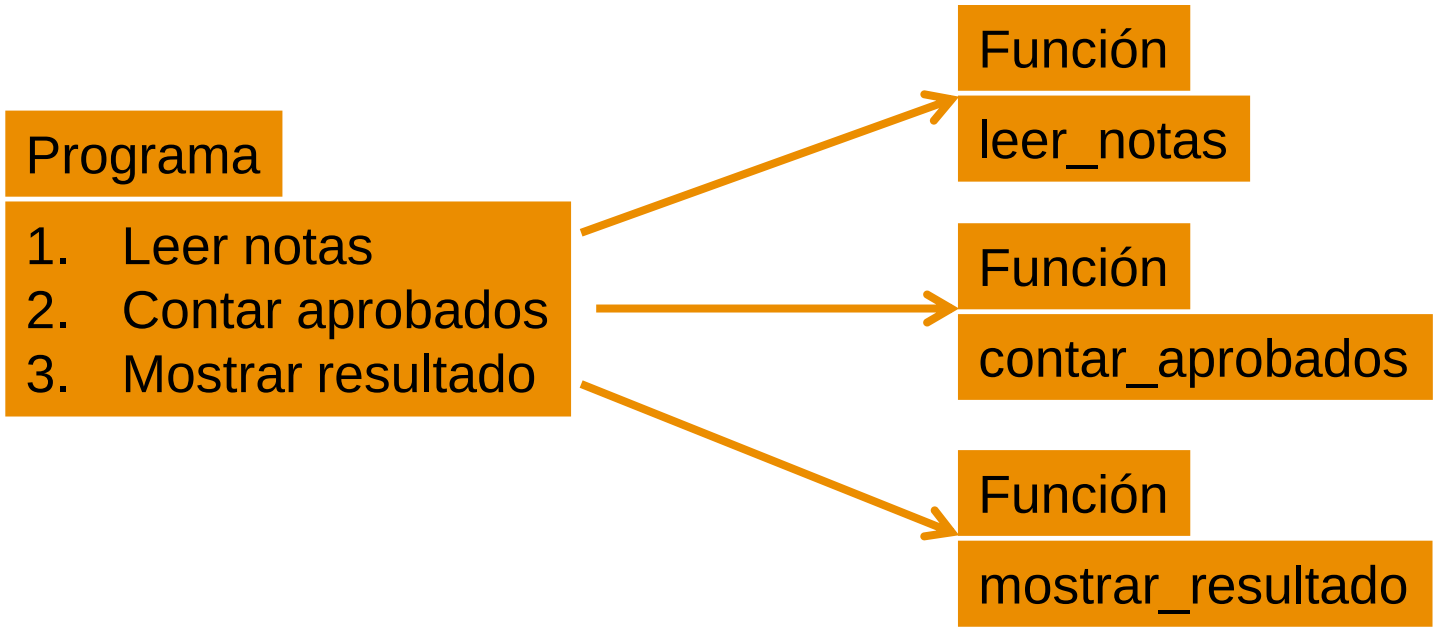
2.5 Definición y uso de subprogramas y funciones

- 2.1 Abstracción de problemas para su programación. Conceptos fundamentales
- 2.2 Variables, expresiones, asignación
- 2.3 Uso de entrada/salida por consola
- 2.4 Manejo de estructuras básicas de control de flujo: secuencial, alternativa y repetitiva
- 2.5 Definición y uso de subprogramas y funciones. Ámbito de variables
- 2.6 Entrada/salida a ficheros
- 2.7 Tipos y estructuras de datos básicas: arrays

Funciones

Una **función** es un fragmento de código de un programa que resuelve un subproblema con entidad propia.

Ejemplo: se pide hacer un programa que lea la nota de todos los alumnos, cuente el número de aprobados y lo muestre en pantalla



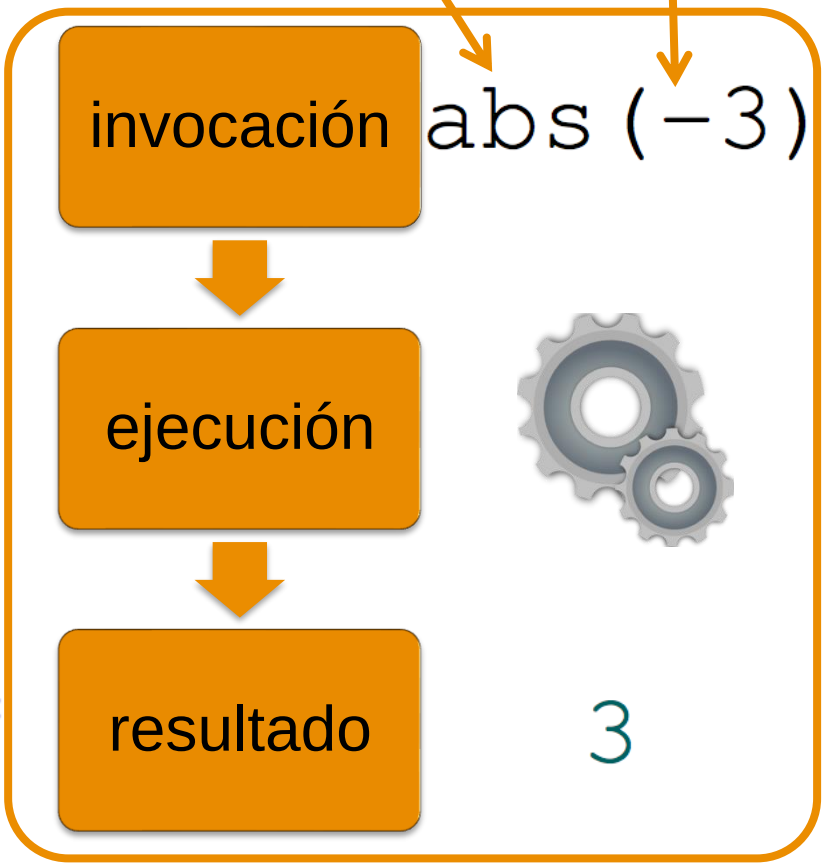
Uso de funciones

- Invocar una función
- Activar una función
- Llamar a una función

= Usar la función

```
>>> abs(-3)
3
>>> abs(round(2.45,1))
2.5
>>> 1+(abs(-3)*2)
7
>>> 2.5 / abs(round(2.45,1))
1.0
>>> 3+str(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: number coercion failed
```

dos argumentos



Funciones que ya existen

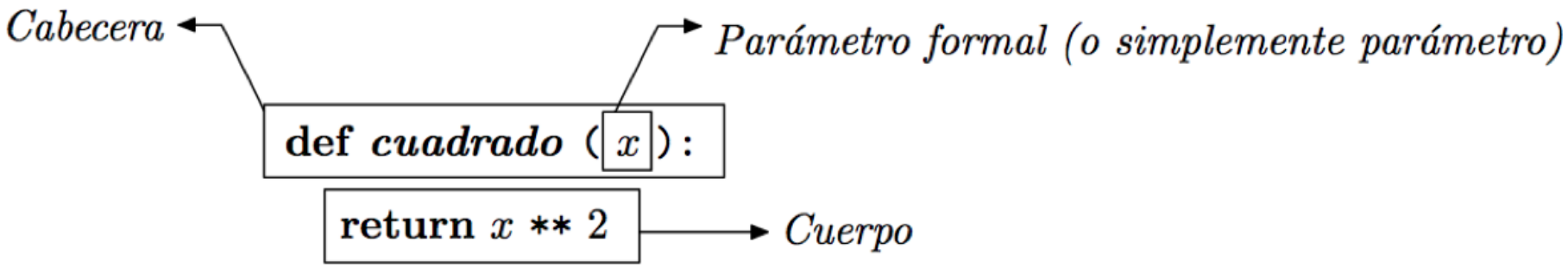
from <módulo> **import** <función>

```
>>> from math import sin
>>> sin(0)
0.0
>>> sin(1)
0.841470984808
>>> cos(0)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: cos
>>> from math import cos
>>> cos(0)
1.0
```

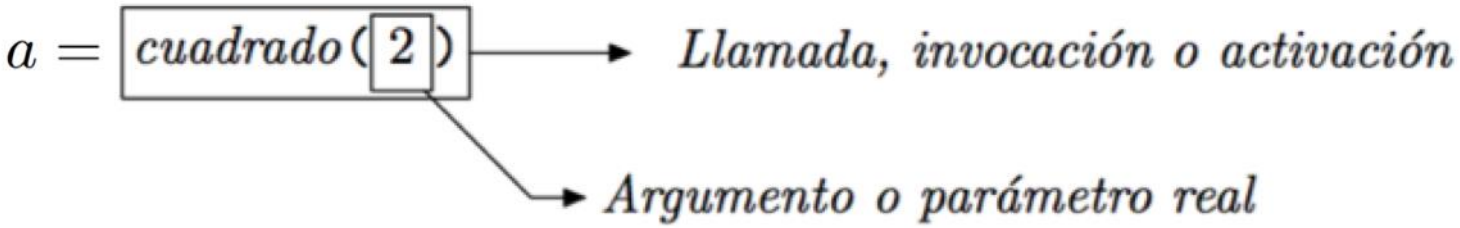
Existen muchos módulos que contienen funciones ya implementadas:

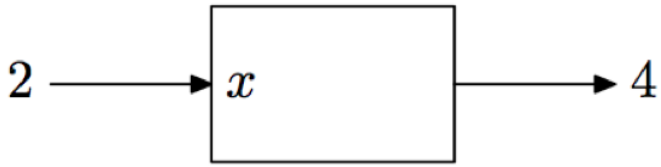
- de interfaces gráficas
- de imagen y sonido
- de bases de datos
- de videojuegos
- …

Definición de nuevas funciones



¡¡¡¡¡¡¡Definir no es invocar!!!!!!!





Una confusión frecuente

Importante: Una confusión frecuente

Supongamos que definimos una función con un parámetro x como esta:

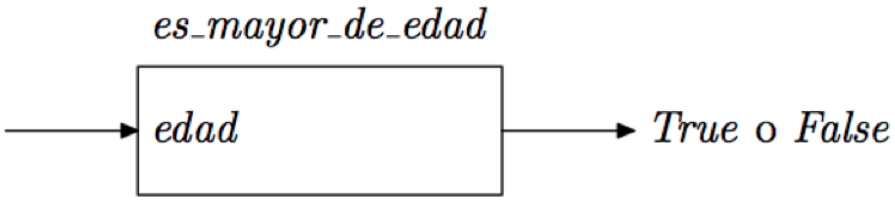
```
1 def cubo (x):
2     return x**3
```

Es frecuente en los aprendices confundir el parámetro x con una variable x. Por tanto, les parece extraño que podamos invocar así a la función:

```
1 def cubo (x):
2     return x**3
3
4 y=1
5 el_cubo = cubo(y)
6 print(el_cubo)
```

¿Cómo es que ahora llamamos y a lo que se llamaba x? No hay problema alguno. Al definir una función, usamos un identificador cualquiera para referirnos al parámetro. Tanto da que se llame x como y.

Otra función



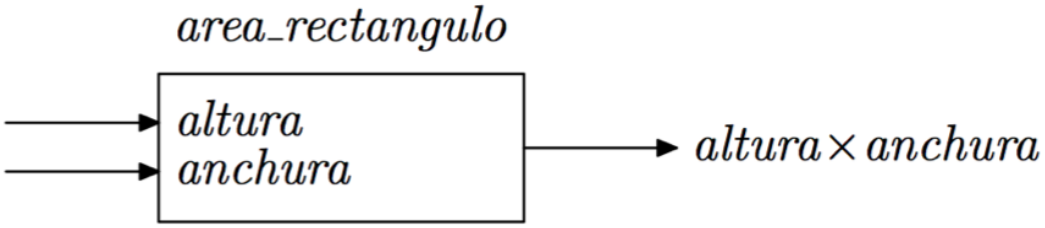
Hacen lo mismo

```
1 def es_mayor_de_edad(edad):
2     if edad < 18:
3         resultado = False
4     else:
5         resultado = True
6     return resultado
```

```
1 def es_mayor_de_edad(edad):
2     if edad < 18:
3         return False
4     else:
5         return True
```

Aparecen dos sentencias **return**: cuando la ejecución llega a cualquiera de ellas, finaliza inmediatamente la llamada a la función y se devuelve el valor que sigue al **return**.

Varios parámetros



parámetros separados por comas

definición

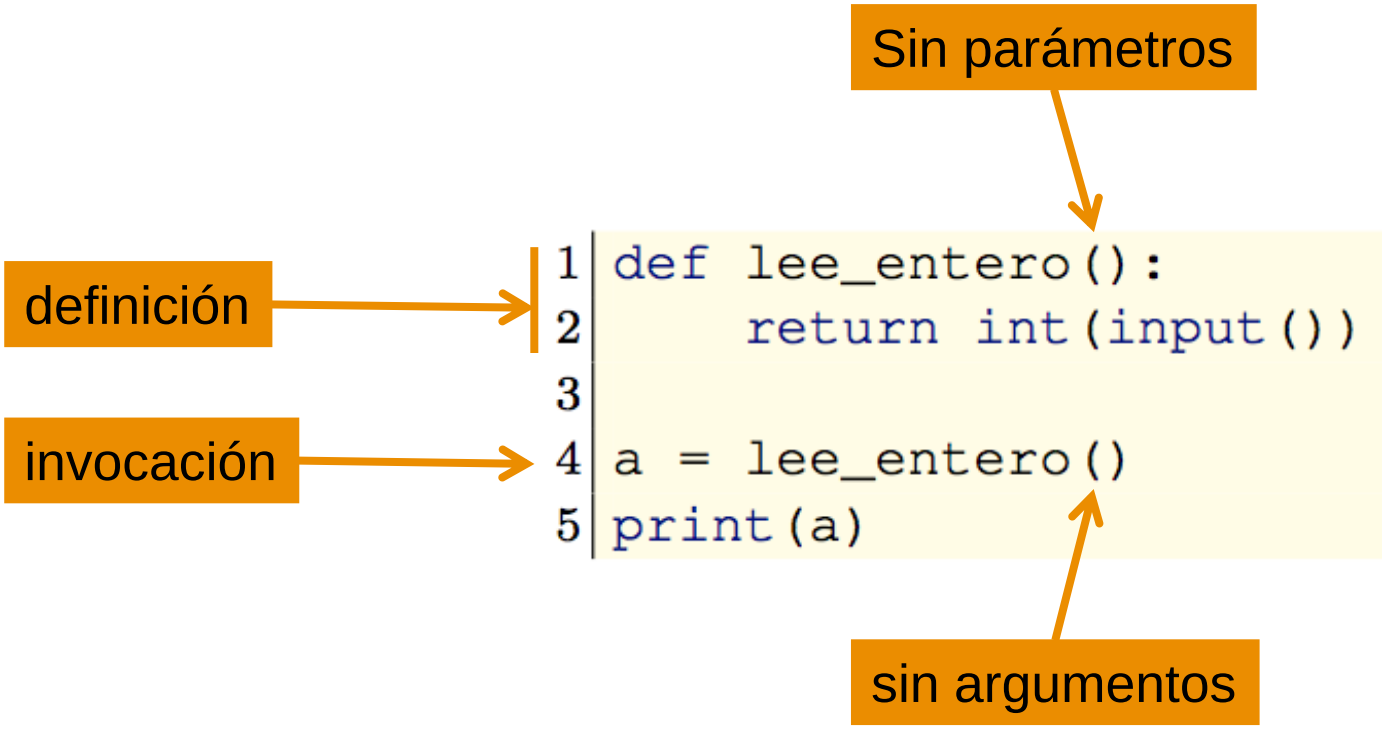
invocación

```
1 def area_rectangulo(altura, anchura):
2     return altura*anchura
3
4 area = area_rectangulo(3,4)
5 print(area)
```

argumentos separados por comas

Funciones sin parámetros

Al definir y al llamar a una función los paréntesis son **obligatorios**



Variables locales y globales

Área de un triángulo

$$\sqrt{s*(s-a)*(s-b)*(s-c)}$$

$$s = (a+b+c)/2$$

a, b y c son los lados

```
1 from math import sqrt
2
3 def area_triangulo(a,b,c):
4     s = (a+b+c)/2.0
5     return sqrt(s*(s-a)*(s-b)*(s-c))
6
7 area = area_triangulo(1,3,2.5)
8 print(area)
9 print(s)
```

La variable s sólo existe en el cuerpo de la función: es una **variable local**

La variable área está definida fuera de cualquier función: es una **variable global**

```
1.1709371246996996
Traceback (most recent call last):
  File "area_triangulo.py", line 9, in <module>
    print(s)
NameError: name 's' is not defined
```