Introducción a la Programación

Prof. Agustín Gravano

Primer semestre de 2022

Clase teórica 4: Funciones

Funciones

En clases pasadas vimos expresiones como las siguientes:

```
len('hola')      # devuelve 4
math.sqrt(4.0)   # devuelve 2.0
```

`len(·)` y `math.sqrt(·)` son dos ejemplos de funciones:

- Son programas dentro de otro programa. Reciben otros nombres, como *subprogramas*, *subrutinas*, *procedimientos* o *métodos*.
- No necesariamente son *funciones* en el sentido matemático, del mismo modo que las *variables* de programación tampoco tienen mucha relación con las *variables* matemáticas.
- Son una herramienta muy importante en la programación: nos permiten abstraer funcionalidades.

Concepto de abstracción

Por ejemplo, ¿qué sabemos de `len(·)` y de `math.sqrt(·)`?

- `len(·)` recibe un string s, y retorna un entero igual a la cantidad de caracteres de s.
- `math.sqrt(·)` recibe un float x, y retorna un float aproximadamente igual a la raíz cuadrada de x.

Sabemos **qué** hacen estas funciones: conocemos su especificación, y con eso nos alcanza para usarlas.

Posiblemente no tengamos idea de **cómo** funcionan o de **cómo** están implementadas. ¡Pero eso no tiene por qué importarnos! Esa es la esencia del concepto de abstracción.

Vimos muchos más ejemplos: +, *, /, `·[·]`, etc., son todas funciones. Python las ofrece (por comodidad) como operadores infijos. Pero vale todo lo mismo que dijimos para `len(·)` y `math.sqrt(·)`.

Usamos el concepto de abstracción naturalmente, casi sin pensarlo: una vez definida una función, si está clara su especificación (qué hace), nos podemos olvidar de su implementación (cómo funciona).

Funciones nuevas

Con frecuencia durante la resolución de un problema, identificamos una funcionalidad que sería conveniente abstraer: un problema particular que podría ser resuelto mediante una función.

Lo primero que debemos hacer es especificar esa función: describir con precisión qué se supone que debe hacer.

Ejemplo: Volvamos al problema de convertir una distancia de millas a kilómetros. Es esperable que, en ciertos contextos, nos interese abstraer esta funcionalidad, para aplicarla muchas veces en forma directa.

Escribamos la especificación de una función que resuelva ese problema:

Encabezado:	`millas_a_kilómetros(d:float)` $\rightarrow$ `float`
Requiere:	`d` ≥ 0
Devuelve:	el resultado (aproximado) de convertir `d` de millas a kilómetros, sabiendo que 1mi $\approx$ 1.60934km.

Importante: Por ahora **no** nos interesa **cómo** implementar esta función.

Conjunto de ejemplos

Encabezado: `millas_a_kilómetros(d:float)` $\rightarrow$ `float`
Requiere: $d \geq 0$
Devuelve: el resultado (aproximado) de convertir `d` de millas a kilómetros, sabiendo que $1\text{mi} \approx 1.60934\text{km}$.

Lo siguiente que haremos es construir un conjunto de ejemplos que describan el comportamiento esperado de la función.

`d`	`resultado`
0.0	0.0
10.0	16.0934
50.0	80.4670
123.4	198.5926

Especificación de una función

Una especificación tiene tres partes:

- Encabezado: Indica el nombre de la función, el nombre, tipo y número de los parámetros, y el tipo del valor de retorno.
- Requiere: Describe restricciones sobre los parámetros de la función.
- Devuelve: Describe el valor de retorno de la función.

Además, con el conjunto de ejemplos de uso, procuraremos lograr un buen cubrimiento, más o menos representativo de los posibles valores de entrada (los argumentos) de la función y sus salidas correspondientes (los valores de retorno).

Estos ejemplos se transformarán después en casos de test, que usaremos en la etapa de verificación de la función.

Algoritmo para resolver el problema

Ahora que ya definimos con claridad en qué consiste el problema, procedemos a pensar un algoritmo que lo resuelva.

Esto es simple en este ejemplo: alcanza con hacer una cuenta sencilla para convertir millas a kilómetros.

Pero en general, este paso es crucial. En este ejemplo puntual es trivial, pero ¡no se dejen engañar!

Implementación · Encabezado y documentación

Lo siguiente es implementar nuestro algoritmo en Python.

```
1  def millas_a_kilómetros(d:float) -> float:
2      ''' Requiere: d >= 0
3          Devuelve: el resultado (aproximado) de
4                    convertir d de millas a kilómetros.
5      '''
6  [...]
```

El encabezado es casi idéntico al de la especificación.

Documentamos la función mediante lo que se conoce como *docstrings*, que tienen ciertas convenciones básicas y varios estilos distintos:

https://www.python.org/dev/peps/pep-0257/

https://www.datacamp.com/community/tutorials/docstrings-python

En esta materia incluimos dos cosas (por ahora): las condiciones requeridas sobre los argumentos y su resultado.

Implementación

Por último, incluimos el código de la función que resuelve el problema:

```
1  def millas_a_kilómetros(d:float) -> float:
2      ''' Requiere: d >= 0
3          Devuelve: el resultado (aproximado) de convertir d
4          de millas a kilómetros, sabiendo que 1mi~1.60934km
5      '''
6      res:float = d * 1.60934
7      return res
```

La línea 7 es muy importante: una instrucción `return EXPRESIÓN` termina la ejecución de la función y devuelve el valor que resulta de evaluar la `EXPRESIÓN`, que debería ser del mismo tipo indicado en la especificación (`float` en este ejemplo).

Prestar atención a la indentación de la función `millas_a_kilómetros`. Esa la forma en que se delimitan los bloques de código en Python. Otros lenguajes usan llaves { } o palabras reservadas como `begin` y `end`.

Ejecución

```
1  def millas_a_kilómetros(d:float) -> float:
2      ''' Req: d >= 0
3          Dev: el resultado (aprox) de convertir d de mi a km. '''
4      res:float = d * 1.60934
5      return res
6
7  # Cuerpo principal del programa
8  mi:float = 123.4
9  km:float = millas_a_kilómetros(mi)
10 print(str(int(mi))+" mi equivale a "+str(int(km))+" km.")
```

La línea 9 invoca a la función `millas_a_kilómetros` con el argumento `123.4`, de tipo `float`. A continuación ocurren estas cosas:

1. Se reserva un espacio de memoria, dedicado específicamente a esta ejecución de esta función, donde se almacenan sus variables `d` y `res`.
2. Se inicializa la variable `d` con el argumento de la invocación: `123.4`.
3. Se ejecuta el código de la función, actualizando las variables `d` y `res`, según corresponda, dentro de este espacio de memoria.
4. La instrucción `return res` devuelve el valor de `res` en ese momento, termina la ejecución de la función y libera su espacio de memoria.
5. Ya nuevamente fuera de la función, se almacena el resultado en la variable `km` y se imprime un mensaje por pantalla.

Funciones sin valor de retorno

Es posible definir funciones que no devuelvan nada. Por ejemplo:

```
1  def mi_funcion(x:float):
2      ''' Requiere: x!=0
3          Devuelve: Nada
4      '''
5      y:float = 1 / x
```

Notar que el encabezado no tiene tipo del valor de retorno, no hay una instrucción `return`. La ejecución de la función termina luego de ejecutar la última instrucción.

Además, en la documentación se mantiene la palabra `Devuelve`, en este caso seguida de la palabra `Nada`. De este modo, queda explícitamente documentado que la función no tiene un valor de retorno.

Repaso de la clase de hoy

- Uso de funciones ya definidas, conociendo su especificación.
- Especificación de funciones: encabezado, requiere, devuelve.
- Conjunto de ejemplos que describen el comportamiento esperado de la función (futuros casos de test).
- Implementación de una función.
- Ejecución de una función: espacio de memoria propio.

Bibliografía complementaria:

- APPP2, capítulo 3, secciones 5.1 a 5.3 (ignorar recursión por ahora)
- HTCSP3, capítulos 3 y 4 (sin prestar atención a los ciclos (`loops`), que veremos más adelante), secciones 6.1 a 6.4 (ignorar recursión por ahora)

Con lo visto, ya pueden resolver hasta el Ejercicio 2 de la Guía de Ejercicios 2.