Programación 1

Tecnicatura Universitaria en Inteligencia Artificial

2022

Apunte 6

1. Funciones

1.1. Introducción

A medida que hemos avanzado en el curso, hemos afrontado problemas cada vez más complejos. La soluciones a estos problemas se han convertido en programas de, también, mayor complejidad, en el sentido en que hemos usado mayor cantidad de líneas de código, hemos usado diferentes estructuras de control y de datos y esto posiblemente a impactado en la cantidad de código repetido en nuestras soluciones, transformando a la complejidad en un nuevo problema a abordar.

Una estrategia para afrontar la complejidad es que los problemas se dividan en problemas menores y se resuelvan por separado. Los algoritmos usualmente son divididos a partir de piezas o componentes denominados *funciones*. Las funciones son subalgoritmos del algoritmo original.

Hay diferentes motivos para dividir un algoritmo en funciones:

- Las funciones evitan la repetición de instrucciones al reutilizarlas mediante una llamada a la función, por lo que el algoritmo se vuelve más compacto.
- Las funciones son algoritmos más fáciles de diseñar, corregir, mantener y modificar.
- Las funciones ocultan los detalles de su especificación, de manera que para utilizarlas sólo es necesario conocer su interfaz.

El ocultamiento de los detalles se refiere a esconder la especificación de cómo es ejecutada una función. Imagina una función como una caja negra de la cual nos interesa sólo su forma de interactuar con el medio que le rodea, entendiendo qué es lo que hace, pero sin dar importancia a cómo lo hace.

El poder de las funciones es que los usuarios pueden suponer que realizan su tarea sin necesidad de entender los detalles internos. ¿Sabes cómo realiza sus cálculos la función `sin x` incluida en tu calculadora? No tienes que saberlo, únicamente asumes que funciona.

Piensa cuánto más difícil será manejar un coche si se tuvieras que entender, antes de poder utilizarlo, cómo funciona la dirección hidráulica. Por lo regular, el único interés de un usuario es manejarlo y no cómo funciona internamente.

Los ingenieros podrían haber agregado en cada generación de nuevas tecnologías una gran cantidad de botones e interruptores, pero sabiamente han mantenido la misma *barrera de abstracción*, es decir, las mismas entradas y salidas: pedal derecho acelera y pedal izquierdo frena. Dicha barrera de abstracción ha permanecido a través de varias generaciones tecnológicas, porque es una interfaz sencilla que oculta detalles innecesarios.

En computación, **una función es un subalgoritmo que realiza una tarea específica y que devuelve un resultado**.

1.2. Funciones integradas (built-in)

El intérprete de Python reconoce un conjunto de funciones que están siempre disponibles, son las llamadas **funciones integradas** (o built-in, en inglés).

abs, *max*, *min* y *len* son ejemplos de funciones de *Python*: la función *abs* permite calcular el valor absoluto de un número, *max* y *min* permiten obtener el máximo y el mínimo entre un conjunto de números, y *len* permite obtener la longitud de una cadena de texto.

Ejercicio
Evalúe:

1. `abs(-28)`
2. `max([2,7,-5,-4])`
3. `min([2,7,-5,-4])`
4. `len("Programación 1")`

Ejercicio
En las unidades anteriores hemos usado otras funciones integradas. ¿Cuáles? Busque ejemplos y pruébelos (Ayuda: puede buscar información en la página oficial de Python https://docs.python.org/3.10/library/functions.html)

Ejercicio
Para obtener la documentación de una instrucción podemos utilizar la función *help* en el intérprete interactivo de python, como por ejemplo, *help('for')*, *help('return')*. Si le pasamos por parámetro el nombre de una función (por ejemplo *help(abs)* o *help(min)*) nos dará la documentación de esa función. Vea la documentación de las funciones con las que trabajó en los ejercicios anteriores.

1.3. Declaración e invocación de funciones propias

Podemos definir nuestras propias funciones. Para ello hacemos uso de *def*.

```
def nombre_funcion(parámetros):
    código
    return retorno
```

La primera línea de la definición de una función la denominaremos, de ahora en más, *encabezado* de la función. El encabezado contiene el nombre de la función y la lista de parámetros.

Una función en *Python* puede recibir *0 o más parámetros* (expresados entre entre paréntesis, y separados por comas), efectúa una *operación* y devuelve *0, 1 o más resultados*.

Ahora que vamos a crear nuestras propias funciones, tal vez alguien se interese en ellas y podamos compartírselas. Las funciones pueden ser muy complejas, y leer código ajeno no es tarea fácil. Es por ello por lo que es importante documentar las funciones. Es decir, añadir comentarios para indicar cómo deben ser usadas.

Para ello debemos usar la triple comilla al principio de la función. Se trata de una especie de comentario que podemos usar para indicar como la función debe ser usada. No se trata de código, es un simple comentario un tanto especial, conocido como *docstring*.

Ahora cualquier persona que tenga nuestra función, podrá llamar a la función *help()* y obtener la ayuda sobre cómo usarla. Quedaría algo así:

```
def nombre_funcion(parámetros):
    """
    Descripción de la función. Cómo debe ser usada,
    qué parámetros acepta y qué devuelve
    """
    código
    return retorno
```

Empecemos por la función más sencilla de todas. Una función sin parámetros de entrada y que no devuelve ningún valor, sino que simplemente imprime un mensaje en pantalla.

```
def di_hola():
    """
    La función di_hola es una función que no recibe parámetros
    y no devuelve ningún resultado
    Sirve para imprimir en pantalla el mensaje Hola
    """
    print("Hola")
```

Hemos declarado o definido la función. El siguiente paso es llamarla con `di_hola()`

Si lo realizamos veremos que se imprime *"Hola"*.

Hemos declarado o definido la función. El siguiente paso es llamarla con

```
di_hola()
```

Si lo realizamos veremos que se imprime *"Hola"*.

Ejercicio
Ejecute la función `di_hola` y obtenga su documentación.

Ejercicio
Cree una función que se llame *menu*, que muestre en pantalla un menú con las siguientes opciones:

1. Ingresar de datos
2. Modificar de datos
3. Listar datos
4. Salir

No se olvide de agregar la documentación de *menu*.

1.4. Retorno y envío de valores

El uso de return permite realizar dos cosas:

- Salir de la función y transferir la ejecución de vuelta a donde se realizó la llamada.
- Devolver uno o varios resultados, fruto de la ejecución de la función.

En lo relativo a lo primero, una vez se llama a *return* se para la ejecución de la función y se vuelve o retorna al punto donde fue llamada. Es por ello por lo que el código que va después del *return* no es ejecutado en el siguiente ejemplo.

```
def mi_funcion():
    print("Entra en mi_funcion")
    return
    print("No llega")
```

Por ello, sólo llamamos a return una vez que hemos acabado de hacer lo que teníamos que hacer en la función.

Ejercicio Ejecute la función `mi_funcion`. Qué ve? Cómo modificaría el código para que se imprima el mensaje "No llega" en pantalla?

Por otro lado, se puede devolver un resultado. Normalmente las funciones son llamadas para realizar unos cálculos en base a una entrada, por lo que es interesante poder devolver esos resultados a quien llamó a la función.

Ejemplos simples:

```
def di_hola_v2():
    """
    La función di_hola_v2 es una función que no recibe parámetros
    y devuelve el string "Hola"
    """
    return "Hola"

def duplicar(x):
    """
    La función duplicar es una función que un número x
    y devuelve el doble de x
    """
    return 2*x
```

Ejercicio
Proponga una función que se llame `C_a_F` que tome una temperatura expresada en grados Celsius y devuelva esa misma temperatura expresada en grados Fahrenheit (Ayuda: si no recuerda las equivalencias, busque en internet)

Ejercicio
Proponga una función que se llame `hms_a_s` que tome una duración expresada en horas, minutos y segundos y devuelva esa misma duración expresada en segundos.

También es posible devolver más de un valor, separados por ,. En el siguiente ejemplo tenemos una función que calcula la suma y media de tres números, y devuelve los dos resultados.

```
def suma_y_media(a, b, c):
    """
    La función suma_y_media es una función que recibe 3 números
    y devuelve la suma de los 3 valores y su media
    """
    suma = a+b+c
```

```
    media = suma/3
    return suma, media
```

Cuando la función debe devolver múltiples resultados, se empaquetan todos juntos en una n-upla (secuencia de valores separados por comas) del tamaño adecuado.

Para guardar los valores resultantes de la función podemos usar una variable (tupla) o podemos usar diferentes variables, como en el siguiente ejemplo.

```
sum, med = suma_y_media(9, 6, 3)
```

Luego podemos operar con dichas variables o imprimir los resultados en pantalla.

Ejercicio
Proponga una función que se llame `s_a_hms` que tome una duración expresada en segundos y devuelva esa misma duración expresada en horas, minutos y segundos.

1.5. Argumentos, parámetros, valor y referencia

Ahora volvamos sobre la definición que vimos de la función `di_hola`. Vamos a complicar un poco las cosas agregando un parámetro de entrada en el encabezado de la función `di_hola`. Ahora si pasamos como entrada un nombre, se imprimirá *"Hola"* seguido del nombre.

```
def di_hola_v3(nombre):
    print("Hola", nombre)
```

Si invocamos la función de la siguiente manera:

```
di_hola_v3("Juan")
```

imprimirá el mensaje en pantalla: *"Hola Juan"*.

Las variables que aparecen en el encabezado de una función, conforman la lista de *parámetros*. Las valores correspondientes a la llamada a la función se denominan *argumentos*. En el ejemplo anterior, *nombre* es el único parámetro de la función `di_hola_v3` y *"Juan"* es el argumento.

Ejercicio
Cree una función que se llame *menu_v2* similar al que ya implementó, pero que además de la opciones muestre el id de la persona logueada en el sistema. El *id* deberá ser un parámetro de entrada de la función. Deberá mostrar, por ejemplo, para un *id = 30567342* el siguiente menú:

Menú (id 30567342):

1. Ingresar de datos
2. Modificar de datos
3. Listar datos
4. Salir

Ejercicio
Proponga una función que recibiendo un palabra la imprima 1000 veces.

Ejercicio
Proponga un función *producto* que tome dos números y devuelva su producto

Ejercicio
Proponga un función `area_tri` que tome dos números que representan la base y altura de un triángulo y devuelva su área.

Ejercicio
Proponga un función `factorial` que tome un entero positivo o cero y calcule su factorial. El factorial de un número n se obtiene la siguiente multiplicación: `1 x 2 x ... x n`.

Python permite pasar argumentos de diferentes formas. A continuación las explicamos.

Argumentos por posición

Los argumentos por posición o posicionales son la forma más básica e intuitiva de pasar parámetros. Si tenemos una función @italicresta que recibe dos parámetros:

```
def resta(a, b):
    return a-b
```

se puede llamar de la siguiente manera:

```
resta(5, 3)
```

Al tratarse de parámetros posicionales, se interpretará que el primer argumento es la `a` y el segundo la `b`.

El número de parámetros es fijo, por lo que si intentamos llamar a la función con un solo argumento, dará error. Tampoco es posible usar más argumentos de los tiene la función definidos, ya que no sabría que hacer con ellos. Por lo tanto si lo intentamos, Python nos dirá que toma 2 posicionales y estamos pasando 3, lo que no es posible.

Argumentos por nombre

Otra forma de llamar a una función, es usando el nombre del parámetro con `=` y el valor del argumento. El siguiente código hace lo mismo que el código anterior, con la diferencia de que los argumentos no son posicionales.

```
resta(a=3, b=5)
```

Al indicar en la llamada a la función el nombre parámetro y el valor que recibe, el orden ya no importa, y se podría llamar de la siguiente forma.

```
resta(b=5, a=3)
```

Como es de esperar, si indicamos un parámetro que no ha sido definido, tendremos un error.

Ejercicio
Invoque a las funciones anteriores usando argumentos posicionales y por nombre (elija Ud los valores de los argumentos, pero preste atención al tipo de dato de cada parámetro).

Argumentos por defecto

Tal vez queramos tener una función con algún parámetro opcional, que pueda ser usado o no dependiendo de diferentes circunstancias. Para ello, lo que podemos hacer es asignar un valor por defecto a dicho parámetro. En el siguiente caso `c` valdría cero salvo que se indique lo contrario.

```
def suma(a, b, c=0):
    return a+b+c
```

Dado que el parámetro `c` tiene un valor por defecto, la función puede ser llamada con sólo dos argumentos.

```
suma(4,3)
```

Podemos incluso asignar un valor por defecto a todos los parámetros

```
def suma_v2(a=3, b=5, c=0):
    return a+b+c
```

Por lo que se podría llamar a la función sin ningún argumento de entrada.

```
suma_v2()
```

Las siguientes llamadas a la función @italicsuma$_v$2*tambinsonvlidas*

```
suma_v2(1)
suma_v2(4,5)
suma_v2(5,3,2)
```

O haciendo uso de lo que hemos visto antes y usando los nombres de los argumentos.

```
suma_v2(a=5, b=3)
```

Paso por valor y referencia

En muchos lenguajes de programación existen los conceptos de paso por valor y por referencia que aplican a la hora de como trata una función a los parámetros que se le pasan como entrada. Su comportamiento es el siguiente:

- Si usamos un *parámetro pasado por valor*, se creará una copia local de la variable, lo que implica que cualquier modificación sobre la misma no tendrá efecto sobre la original.
- Con un *parámetro como referencia*, se actuará directamente sobre la variable pasada, por lo que las modificaciones afectarán a la variable original.

En *Python* las cosas son un poco distintas, y el comportamiento estará definido por el tipo de variable con la que estamos tratando. Veamos un ejemplo de paso por valor.

```
x = 10
def funcion_v1(entrada):
    entrada = 0
funcion_v1(x)

print(x)
```

Iniciamos la `x` a *10* y se la pasamos a `funcion_v1`. Dentro de la función hacemos que la variable valga *0*. Dado que *Python* trata a los *int* como pasados por valor, dentro de la función se crea una copia local de x, por lo que la variable original no es modificada.

Ejercicio
Pruebe la función `funcion_v1` con diferentes valores enteros y analice los resultados.

No pasa lo mismo si por ejemplo x es una lista como en el siguiente ejemplo. En este caso Python lo trata como si estuviese pasada por referencia, lo que hace que se modifique la variable original. La variable original x ha sido modificada.

```
x = [10, 20, 30]
def funcion_v2(entrada):
    entrada.append(40)

funcionv2_(x)
print(x)
```

Ejercicio
Pruebe la función `funcion_v2` con diferentes listas y analice los resultados.

Anotaciones en funciones

Existe una funcionalidad relativamente reciente en Python llamada *function annotation* o *anotaciones en funciones*. Dicha funcionalidad nos permite añadir metadatos a las funciones, indicando los tipos esperados tanto de entrada como de salida.

```
def multiplica_por_3(numero: int) -> int:
    return numero*3
```

Las anotaciones son muy útiles de cara a la documentación del código, pero no imponen ninguna norma sobre los tipos. Esto significa que se puede llamar a la función con un parámetro que no sea int, y no obtendremos ningún error (por ejemplo, si invocamos a la función de esta manera `multiplica_por_3("Hola")` `no obtendemos ningún mensaje avisando el error`).

1.6. Argumentos indeterminados: args y kwargs

Hemos visto que la función puede ser llamada con diferente número de argumentos de entrada, pero esto no es realmente una función con argumentos de longitud variable, ya que existe un número máximo.

Imaginemos que queremos una función suma() como la de antes, pero en este caso necesitamos que sume todos los números de entrada que se le pasen, sin importar si son 3 o 100. Una primera forma de hacerlo sería con una lista.

```
def suma(numeros):
    total = 0
    for n in numeros:
        total += n
    return total
```

Sin embargo, veamos como podemos resolver este problema con `*args` y `**kwargs` en Python.

Uso de *args

Gracias a los `*args` en Python, podemos definir funciones cuyo número de argumentos es variable. Es decir, podemos definir funciones genéricas que no aceptan un número determinado de parámetros, sino que se "adaptan" al número de argumentos con los que son llamados.

De hecho, el `args` viene de arguments en Inglés, o argumentos. Haciendo uso de `*args` en la declaración de la función podemos hacer que el número de parámetros que acepte sea variable.

```
def suma(*args):
    s = 0
    for arg in args:
        s += arg
    return s
```

```
suma(1, 3, 4, 2)
suma(1, 1)
```

Antes de nada, el uso del nombre `args` es totalmente arbitrario, por lo que podrías haberlo llamado como quisieras. Es una mera convención entre los usuarios de Python y resulta frecuente darle ese nombre. Lo que si es un requisito, es usar `*` junto al nombre.

En el ejemplo anterior hemos visto como `*args` puede ser iterado, ya que en realidad es una tupla. Por lo tanto iterando la tupla podemos acceder a todos los argumentos de entrada, y en nuestro caso sumarlos y devolverlos.

Con esto resolvemos nuestro problema inicial, en el que necesitábamos un número variable de argumentos. Sin embargo, hay otra forma que nos proporciona además un nombre asociado al argumento, con el uso de `**kwargs`. La explicamos a continuación.

Uso de **kwargs

Al igual que en `*args`, en `**kwargs` el nombre es una mera convención entre los usuarios de Python. Puedes usar cualquier otro nombre siempre y cuando respetes el `**`.

En este caso, en vez de tener una tupla tenemos un diccionario.

Pero veamos un ejemplo más completo. A diferencia de `*args`, los `**kwargs` nos permiten dar un nombre a cada argumento de entrada, pudiendo acceder a ellos dentro de la función a través de un diccionario.

```
def suma(**kwargs):
    s = 0
    for clave, valor in kwargs.items():
        print(clave, "=", valor)
        s += valor
    return s
```

Si invocamos a la función `suma` de la siguiente manera:

```
suma(a=3, b=10, c=3)
```

Veremos la siguiente salida:

```
a = 3

b = 10

c = 3
```

Como podemos ver, es posible iterar los argumentos de entrada con @italicitems(), y podemos acceder a la *clave* (o nombre) y el *valor* de cada argumento.

El uso de los `**kwargs` es muy útil si además de querer acceder al valor de las variables dentro de la función, quieres darles un nombre que brinde una información extra.

1.7. Funciones lambda

Las *funciones lambda* o *anónimas* son un tipo de funciones en *Python* que típicamente se definen en una línea y cuyo código a ejecutar suele ser pequeño.

Citando la documentación oficial:

"Python lambdas are only a shorthand notation if you're too lazy to define a function."

Lo que significa algo así como *"las funciones lambda son simplemente una versión acortada, que puedes usar si te da pereza escribir una función"*

Lo que sería una función que suma dos números como la siguiente:

```
def suma(a, b):
    return a+b
```

Se podría expresar en forma de una función lambda de la siguiente manera:

```
lambda a, b : a + b
```

La primera diferencia es que una función lambda no tiene un nombre, y por lo tanto salvo que sea asignada a una variable, no puede ser invocada. Para ello debemos asignarla a una variable, por ejemplo:

```
suma = lambda a, b: a + b
```

Una vez que tenemos la función, es posible llamarla como si de una función normal se tratase.

```
suma(2, 4)
```

Si es una función que solo queremos usar una vez, tal vez no tenga sentido almacenarla en una variable. Es posible declarar la función y llamarla en la misma línea.

```
(lambda a, b: a + b)(2, 4)
```

1.8. Números aleatorios

A partir de las mismas entradas, la mayoría de los programas generarán las mismas salidas cada vez, que es lo que llamamos comportamiento determinista. El determinismo normalmente es algo bueno, ya que esperamos que la misma operación nos proporcione siempre el mismo resultado.

Nota: Un módulo es no es otra cosa sino un archivo con extensión `.py`*. Un módulo puede definir funciones, clases y variables. En la siguiente unidad definiremos formalmente de qué se trata.*

Para ciertas aplicaciones, sin embargo, querremos que el resultado sea impredecible. Los juegos son el ejemplo obvio, pero hay más.

Conseguir que un programa sea realmente no-determinista no resulta tan fácil, pero hay modos de hacer que al menos lo parezca. Una de ellos es usar algoritmos que generen números pseudoaleatorios. Los números pseudoaleatorios no son verdaderamente aleatorios, ya que son generados por una operación determinista, pero si sólo nos fijamos en los números resulta casi imposible distinguirlos de los aleatorios de verdad.

El módulo `random` proporciona funciones que generan números pseudoaleatorios (a los que simplemente llamaremos *"aleatorios"* de ahora en adelante). La función `random` devuelve un número flotante aleatorio entre *0.0* y *1.0* (incluyendo 0.0, pero no *1.0*). Cada vez que se llama a `random`, se obtiene el número siguiente de una larga serie. Para ver un ejemplo, ejecuta este bucle:

Nota: Para poder invocar las funciones definidas en un módulo debemos indicar al intérprete dónde se encuentran definidas, importando el módulo: ***`import modulo`***

Ejemplo:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Nota: Para invocar a una función de un módulo usamos la notación: ***`modulo.funcion(args)`***

Este programa produce la siguiente lista de 10 números aleatorios entre *0.0* y hasta (pero no incluyendo) *1.0*:

0.11132867921152356

0.5950949227890241

0.04820265884996877

0.841003109276478

0.997914947094958

0.04842330803368111

0.7416295948208405

0.510535245390327

0.27447040171978143

0.028511805472785867

La función random es solamente una de las muchas que trabajan con números aleatorios. La función `randint` toma los parámetros inferior y superior, y devuelve un entero entre inferior y superior (incluyendo ambos extremos).

Ejemplo:

```
random.randint(5, 10)
```

Para elegir un elemento de una secuencia aleatoriamente, se puede usar @italicchoice:

```
t = [1, 2, 3]
random.choice(t)
```

Ejercicio
Proponga una función `tirar_dado` para simular la tirada de un dado. La función no tomará parámetros y devolverá un número entero entre 1 y 6.

2. Bibliografía

- Apunte de la materia Algoritmos y Programación 1, primera materia de programación de las carreras de Ingeniería en Informática y Licenciatura en Análisis de Sistemas dela Facultad de Ingeniería de la UBA: http://materias.fi.uba.ar/7501/apunte%20PYTHON.pdf
- El Libro de Python. Libro digital:https://ellibrodepython.com/
- Documentación de la página oficial de Python: https://docs.python.org/3.10/