08 - Functions as Objects

Uso de funciones para modularizar el código

Hasta ahora, todas las funciones que hemos implementado han sido pequeñas. Caben perfectamente en una sola página. A medida que implementamos funcionalidades más complicadas, *es conveniente dividir las funciones en múltiples funciones*, cada una de las cuales hace una cosa simple.

Para ilustrar esta idea, de forma algo superflua, dividimos `find_root` en tres funciones separadas. Cada una de las funciones tiene su propia especificación y cada una tiene sentido como entidad independiente. La función `find_root_bounds` encuentra un intervalo en el que debe estar la raíz, `bisection_solve` utiliza la búsqueda por bisección para buscar en este intervalo una aproximación a la raíz, y `find_root` simplemente llama a las otras dos y devuelve la raíz.

¿Es esta versión de `find_root` más fácil de entender que la implementación monolítica original? Probablemente no. Una buena regla general es que *si una función cabe cómodamente en una sola página, probablemente no necesite subdividirse* para ser fácilmente comprensible.

```
def find_root_bounds(x, power):
    """ x a float, power a positive int
    return low, high such that low**power <=x and high**power >= x
    """

    low = min(-1, x)
    high = max(1, x)

    return low, high

def bisection_solve(x, power, epsilon, low, high):
    """x, epsilon, low, high are floats
    epsilon > 0
    low <= high and there is an ans between low and high s.t.
        ans**power is within epsilon of x
    returns ans s.t. ans**power within epsilon of x"""

    ans = (high + low) / 2

    while abs(ans**power - x) >= epsilon:
        if ans**power < x:
```

```
            low = ans
        else:
            high = ans
        ans = (high + low) / 2

    return ans

def find_root(x, power, epsilon):
    """Assumes x and epsilon int or float, power an int,
        epsilon > 0 & power >= 1
    Returns float y such that y**power is within epsilon of x.
        If such a float does not exist, it returns None"""

    if x < 0 and power % 2 == 0:
        return None #Negative number has no even-powered roots

    low, high = find_root_bounds(x, power)

    return bisection_solve(x, power, epsilon, low, high)
```

Las funciones como objetos

En Python, las funciones son **objetos de primera clase**. Esto significa que *pueden tratarse como objetos de cualquier otro tipo*, por ejemplo, `int` o `list`. Tienen tipos, por ejemplo, la expresión `type(abs)` tiene el valor `<type “built-in_function_or_method”>`; pueden aparecer en expresiones, por ejemplo, como el lado derecho de una sentencia de asignación o como argumento de una función; pueden ser devueltas por funciones; etc.

El uso de funciones como argumentos permite un estilo de codificación denominado **programación de orden superior**. Nos permite escribir funciones que tienen una utilidad más general. Por ejemplo, la función `bisection_solve` puede reescribirse para que pueda aplicarse a tareas distintas de la búsqueda de raíces.

```
def bisection_solve(x, eval_ans, epsilon, low, high):
    """ x, epsilon, low, high are floats
        epsilon > 0
        eval_ans a function mapping a float to a float
        low <= high and there is an ans between low and high s.t.
            eval(ans) is within epsilon of x
        returns ans s.t. eval(ans) within epsilon of x"""

    ans = (high + low) / 2

    while abs(eval_ans(ans) - x) >= epsilon:
```

```
        if eval_ans(ans) < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2

    return ans
```

Empezamos sustituyendo el parámetro entero `power` por una función, `eval_ans`, que convierte flotantes en flotantes. A continuación, sustituimos cada instancia de la expresión `ans ** power` por la llamada a la función `eval_ans(ans)`.

Si quisiéramos utilizar el nuevo `bisection_solve` para imprimir una aproximación a la raíz cuadrada de 99, podríamos ejecutar el código

```
def square(ans):
    return ans ** 2

low, high = find_root_bounds(99, 2)

print(bisection_solve(99, square, 0.01, low, high))
```

Tomarse la molestia de definir una función para hacer algo tan simple como elevar un número al cuadrado parece un poco tonto. Por suerte, Python permite crear **funciones anónimas** (es decir, funciones que no están vinculadas a un nombre) utilizando la palabra reservada **lambda**. La forma general de una expresión lambda es

```
lambda sequence of variable names : expression
```

Por ejemplo, la expresión `lambda x, y: x*y` devuelve una función que devuelve el producto de sus dos argumentos. Las expresiones **lambda** se utilizan con frecuencia como argumentos de funciones de orden superior. Por ejemplo, podríamos sustituir la llamada anterior a `bisection_solve` por

```
print(bisection_solve(99, lambda ans: ans ** 2, 0.01, low, high))
```

Ejercicio manual

Escriba una expresión lambda que tenga dos parámetros numéricos. Si el segundo argumento es igual a cero, debe devolver `None`. En

caso contrario debe devolver el valor de dividir el primer argumento por el segundo argumento. Sugerencia: utilice una expresión condicional.

Mi aproximación al ejercicio:

```
lambda x, y: None if y == 0 else x / y
```

Dado que las funciones son objetos de primera clase, *pueden crearse y devolverse dentro de las funciones*. Por ejemplo, dada la definición de función

```
def create_eval_ans():
    power = input('Enter a positive integer: ')
    return lambda ans: ans**int(power)
```

El código

```
eval_ans = create_eval_ans()
print(bisection_solve(99, eval_ans, 0.01, low, high))
```

imprimirá una aproximación a la raíz enésima de 99, donde n es un número introducido por el usuario.

La forma en que hemos generalizado `bisection_solve` significa que ahora se puede utilizar no sólo para buscar aproximaciones a raíces, sino para buscar aproximaciones a cualquier función monotónica que mapee flotantes a flotantes. Por ejemplo, el siguiente código utiliza `bisection_solve` para encontrar aproximaciones a logaritmos.

```
def log(x, base, epsilon):
    """Assumes x and epsilon int or float, base an int,
        x > 1, epsilon > 0 & power >= 1
    Returns float y such that base**y is within epsilon of x."""
    def find_log_bounds(x, base):

        upper_bound = 0

        while base ** upper_bound < x:
            upper_bound += 1
        return upper_bound - 1, upper_bound

    low, high = find_log_bounds(x, base)
```

```
    return bisection_solve(x, lambda ans: base**ans, epsilon, low, high)
```

Observa que la implementación de `log` incluye la definición de una función local, `find_log_bounds`. Esta función podría haberse definido fuera de `log`, pero como no esperamos utilizarla en ningún otro contexto, nos pareció mejor no hacerlo.

Métodos, excesivamente simplificados

Los métodos son objetos similares a las funciones. Pueden ser llamados con parámetros, pueden devolver valores y pueden tener efectos secundarios.

Por ahora, piense que los métodos proporcionan una sintaxis peculiar para la llamada a una función. En lugar de poner el primer argumento entre paréntesis a continuación del nombre de la función, utilizamos la notación de punto para colocar ese argumento antes del nombre de la función. Introducimos aquí los métodos porque muchas operaciones útiles sobre tipos incorporados son métodos y, por lo tanto, se invocan utilizando la notación de puntos. Por ejemplo, si `s` es una cadena, el método `find` puede utilizarse para encontrar el índice de la primera aparición de una subcadena en `s`. Así, si `s` fuera `“abcbc”`, la invocación `s.find(“bc”)` devolvería `1`. Si se intenta tratar `find` como una función, por ejemplo, invocando `find(s,“bc”)`, se produce el mensaje de error `NameError: name “find” is not defined`.

> **Ejercicio manual**
>
> ¿Qué devuelve s.find(sub) si sub no aparece en s?

Mi aproximación al ejercicio:

Si `sub` no aparece en `s` entonces va a devolver `-1`.

> **Ejercicio manual**
>
> Utilice find para implementar una función que satisfaga la especificación

```
def find_last(s, sub):
    """s and sub are non-empty strings
        Returns the index of the last occurrence of sub in s.
        Returns None if sub does not occur in s"""
```

Mi aproximación al ejercicio:

```
def find_last(s, sub):
    """s and sub are non-empty strings
    Returns the index of the last occurrence of sub in s.
    Returns None if sub does not occur in s"""

    last_index = -1
    index = s.find(sub)

    while index != -1:
        last_index = index
        index = s.find(sub, index + 1)

    if last_index == -1:
        return None
    return last_index
```

Se usa el método `find(sub, start)` para buscar la siguiente aparición de sub en `s`, y vamos guardando el ultimo índice valido, si nunca se encuentra devuelve `None`.

Ejercicio manual de la lectura

Implemente la función que cumpla las siguientes especificaciones:

```
def same_chars(s1, s2):
    """
    s1 and s2 are strings
    Returns boolean True is a character in s1 is also in s2, and vice
    versa. If a character only exists in one of s1 or s2, returns False.
    """
    # Your code here
    for char in s1:
        if char not in s2:
            return False

    for char in s2:
        if char not in s1:
```

```
            return False
    return True

# Examples:
print(same_chars("abc", "cab"))     # prints True
print(same_chars("abccc", "caaab")) # prints True
print(same_chars("abcd", "cabaa"))  # prints False
print(same_chars("abcabc", "cabz")) # prints False
```

El primer bucle se asegura de que no haya letras en `s1` que falten en `s2`. El segundo bucle se asegura de que no haya letras en `s2` que falten en `s1`. Si ambos se cumplen entonces devuelve `True`.