07 - Decomposition, Abstraction, Functions

Hasta ahora, hemos introducido números, asignaciones, entrada/salida, comparaciones y construcciones de bucle. ¿Cómo de potente es este subconjunto de Python? En un sentido teórico, es tan potente como puedas necesitar, es decir, es Turing completo. Esto significa que si un problema puede resolverse utilizando computación, puede resolverse utilizando sólo los mecanismos lingüísticos que ya has visto.

Pero que algo pueda hacerse no significa que deba hacerse. Aunque, en principio, cualquier cálculo puede implementarse utilizando sólo estos mecanismos, hacerlo es muy poco práctico. En el último capítulo, vimos un algoritmo para encontrar una aproximación a la raíz cuadrada de un número positivo.

Uso de la búsqueda por bisección para aproximar la raíz cuadrada de x

```
# Find approximation to square root of x
if x < 0:
    print("Does not exist")
else:
    low = 0
    high = max(1, x)
    ans = (high + low) / 2

    while abs(ans ** 2 - x) >= epsilon:
        if ans ** 2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2

    print(ans, "is close to square root of", x)
```

Es un código razonable, pero carece de utilidad general. Sólo funciona para los valores asignados a las variables x y épsilon.

Esto significa que si queremos reutilizarlo, tenemos que copiar el código, posiblemente editar los nombres de las variables, y pegarlo donde queramos. No podemos utilizar fácilmente este cálculo dentro de otro cálculo más complejo. Además, si queremos calcular raíces cúbicas en lugar de raíces cuadradas, tenemos que editar el código. Si queremos un programa que calcule raíces cuadradas y cúbicas (o que calcule raíces cuadradas en dos lugares diferentes), el programa contendría múltiples trozos de código casi idéntico.

El siguiente código adapta el código anterior para imprimir la suma de la raíz cuadrada de `x1` y la raíz cúbica de `x2`. El código funciona, pero no es bonito.

Suma de una raíz cuadrada y una raíz cúbica

```
# Find square root of x1
if x1 < 0:
    print("Does not exist")
else:
    low = 0
    high = max(1, x1)
    ans = (high + low) / 2

    while abs(ans ** 2 - x1) >= epsilon
        if ans ** 2 < x1:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2

x1_root = ans

x2 = -8

# Find cube root of x2
if x2 < 0:
    is_pos = False
    x2 = -x2
else:
    is_pos = True

low = 0
high = max(1, x2)
ans = (high + low) / 2

while abs(ans ** 3 - x2) >= epsilon
    if ans ** 3 < x2:
        low = ans
    else:
        high = ans
    ans = (high + low) / 2

if is_pos:
    x2_root = ans
else:
    x2_root = -ans
    x2 = -x2

print("Sum of square root of", x1, "and cube root of", x2, "is close to",
x1_root + x2_root)
```

Cuanto más código contenga un programa, más posibilidades hay de que algo salga mal, y más difícil es mantener el código. Imaginemos, por ejemplo, que hubiera un error en la implementación inicial de la

búsqueda por bisección, y que el error saliera a la luz al probar el programa. Sería demasiado fácil arreglar la implementación en un lugar y no darse cuenta de que hay código similar en otro sitio que necesita reparación.

Afortunadamente, Python proporciona varias características lingüísticas que hacen relativamente fácil generalizar y reutilizar código. La más importante es la función.

Funciones y alcance

Ya hemos utilizado una serie de funciones incorporadas, por ejemplo, `max` y `abs`. La posibilidad de que los programadores definan y utilicen sus propias funciones, como si estuvieran incorporadas, supone un salto cualitativo en cuanto a comodidad.

Definición de función

En Python cada definición de función es de la forma:

```
1   def nombre de la función (lista de parametros):
2       Cuerpo de la función
```

Por ejemplo, podríamos definir la función `max_val` mediante el código

```
1   def max_val(x, y):
2       if x > y:
3           return x
4       else:
5           return y
```

`def` es una palabra reservada que indica a Python que una función está a punto de ser definida. El nombre de la función (`max_val` en este ejemplo) es simplemente un nombre que se usa para referirse a la función. La convención PEP 8 es que los nombres de función deben estar en *minúsculas con las palabras separadas por guiones bajos para mejorar la legibilidad*.

La secuencia de nombres entre paréntesis que sigue al nombre de la función (`x,y` en este ejemplo) son los parámetros formales de la función. Cuando se utiliza la función, los parámetros formales están vinculados (como en una sentencia de asignación) a los parámetros reales (a menudo denominados argumentos) de la invocación de la

función (también denominada llamada a la función). Por ejemplo, la invocación

```
max_val(3, 4)
```

vincula `x` a `3` e `y` a `4`.

El *cuerpo de una función es cualquier fragmento de código de Python*. Sin embargo, existe una sentencia especial, `return`, que *sólo puede usarse dentro del cuerpo de una función*.

Una llamada a función es una expresión, y como todas las expresiones, tiene un valor. Ese valor es devuelto por la función invocada. Por ejemplo, el valor de la expresión `max_val(3,4) * max_val(3,2)` es 12, porque la primera invocación de `max_val` devuelve el `int 4` y la segunda devuelve el `int 3`. Observe que la ejecución de una sentencia `return` termina una invocación de la función.

Recapitulando, cuando se llama a una función.

1. Las *expresiones que componen los parámetros reales se evalúan*, y los *parámetros formales de la función se vinculan a los valores resultantes*. Por ejemplo, la invocación `max_val(3+4, z)` ligará el parámetro formal `x` a `7` y el parámetro formal `y` al valor que tenga la variable `z` cuando se ejecute la invocación.
2. El *punto de ejecución* (la siguiente instrucción a ejecutar) se desplaza desde el punto de invocación hasta *la primera sentencia del cuerpo de la función*.
3. El *código en el cuerpo de la función se ejecuta hasta que se encuentra una sentencia* `return`, en cuyo caso el valor de la expresión que sigue al `return` se convierte en el valor de la invocación de la función, o no hay más sentencias que ejecutar, en cuyo caso la función devuelve el valor `None`. (Si no hay ninguna expresión tras el `return`, el valor de la invocación es `None`).
4. El *valor de la invocación es el valor devuelto*.
5. El *punto de ejecución se transfiere de nuevo al código inmediatamente posterior a la invocación*.

Los **parámetros** permiten a los programadores escribir código que no accede a objetos específicos, sino a cualquier objeto que el invocador de la función decida utilizar como parámetros reales. Esto se denomina **abstracción lambda**.

El siguiente código contiene una función que tiene tres parámetros formales y devuelve un valor, llámese resultado, tal que `abs(resultado`

`** potencia - x) >= épsilon`.

Una función para encontrar raíces

```
def find_root(x, power, epsilon):
    # Find interval containing answer

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

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

    # Use bisection search

    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
```

El siguiente código contiene código que se puede utilizar para probar si `find_root` funciona según lo previsto. La función de prueba `test_find_root` tiene aproximadamente la misma longitud que `find_root`. Para los programadores inexpertos, escribir funciones de prueba a menudo parece una pérdida de esfuerzo. Sin embargo, los programadores experimentados saben que invertir en escribir código de prueba suele reportar grandes beneficios. Ciertamente es mejor que sentarse frente al teclado y escribir casos de prueba en el intérprete de comandos una y otra vez durante la depuración (el proceso de averiguar por qué un programa no funciona, y luego arreglarlo). Observe que, como estamos invocando a `test_find_root` con tres tuplas (es decir, secuencias de valores) de longitud tres, una llamada comprueba 27 combinaciones de parámetros.

Por último, dado que `test_find_root` comprueba si `find_root` devuelve una respuesta adecuada e informa del resultado, evita al programador la tediosa tarea, propensa a errores, de inspeccionar visualmente cada salida y comprobar si es correcta.

Codigo para probar find_root

```
def test_find_root(x_vals, powers, epsilons):
    for x in x_vals:
```

```
        for p in powers:
            for e in epsilons:
                result = find_root(x, p, e)
                if result == None:
                    val = "No root exists"
                else:
                    val = "Okay"
                    if abs(result ** p - x) > e:
                        val = "Bad"
                print(f'x = {x}, power = {p}, epsilon = {e}: {val}')

x_vals = (0.25, 8, -8)
powers = (1, 2, 3)
epsilons = (0.1, 0.001, 1)
test_find_root(x_vals, powers, epsilons)
```

Ejercicio manual

Utilice la función `find_root` de "Una función para encontrar raíces" para imprimir la suma de aproximaciones a la raíz cuadrada de 25, la raíz cúbica de -8 y la raíz cuarta de 16. Usa 0.001 como épsilon.

Mi aproximación al ejercicio:

```
def find_root(x, power, epsilon):
    # Find interval containing answer

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

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

    # Use bisection search

    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

print(find_root(25, 2, 0.001) + find_root(-8, 3, 0.001) + find_root(16, 4,
0.001))
```

Como ya tenemos implementada la función solo debemos invocarla con los argumentos que necesitamos.

> **Ejercicio manual**
>
> Escribe una función `is_in` que acepte dos cadenas como argumentos y devuelva `True` si cualquiera de las cadenas aparece en cualquier parte de la otra, y `False` en caso contrario.
>
> Sugerencia: quizás quieras usar el operador incorporado `str in`.

Mi aproximación al ejercicio:

```
def is_in(str1, str2):
    if str1 in str2 or str2 in str1:
        return True
    else:
        return False
```

Si la cadena `str1` o `str2`, esta contenida dentro de la otra devuelvo `True` de lo contrario `False`.

> **Ejercicio manual**
>
> Escribe una función para probar is_in.

Mi aproximación al ejercicio:

```
def test_is_in():
    str1_list = ["cat", "abc", "x", "hello", "dog", "hi", "python"]
    str2_list = ["concatenate", "ab", "xyz", "hello", "cat", "hello", "java"]

    for str1 in str1_list:
        for str2 in str2_list:
            result = is_in(str1, str2)
            print(f'is_in("{str1}", "{str2}") = {result}')
```

Compruebo si cada elemento de la lista lo contiene en cada elemento de la otra lista.

Argumentos y valores por defecto de las palabras clave

En Python, hay dos maneras en que los parámetros formales se unen a los parámetros reales. El método más común, que es el que hemos usado hasta ahora, se llama **posicional**: el primer parámetro formal se vincula al primer parámetro real, el segundo formal al segundo real, etc. Python también admite **argumentos de palabra clave**, en los que los formales se asocian a los reales utilizando el nombre del parámetro formal. Considere la definición de función

```
def print_name(first_name, last_name, reverse):

    if reverse:
        print(last_name + ', ' + first_name)
    else:
        print(first_name, last_name)

```

La función `print_name` asume que `first_name` y `last_name` son cadenas y que `reverse` es un booleano. Si `reverse == True`, imprime apellido, nombre; en caso contrario, imprime nombre apellido.

Cada una de las siguientes es una invocación equivalente de `print_name`:

```
print_name('Olga', 'Puchmajerova', False)
print_name('Olga', 'Puchmajerova', reverse = False)
print_name('Olga', last_name = 'Puchmajerova', reverse = False)
print_name(last_name = 'Puchmajerova', first_name = 'Olga', reverse =
False)
```

Aunque los argumentos de palabra clave pueden aparecer en cualquier orden en la lista de parámetros reales, **no es legal seguir un argumento de palabra clave con un argumento que no sea de palabra clave**. Por lo tanto, se produciría un mensaje de error

```
print_name('Olga', last_name = 'Puchmajerova', False)
```

Los argumentos de palabra clave se suelen utilizar junto con valores de parámetros por defecto. Podemos, por ejemplo, escribir

```
def print_name(first_name, last_name, reverse = False):
    if reverse:
        print(last_name + ', ' + first_name)
    else:
        print(first_name, last_name)
```

Los valores por defecto permiten a los programadores llamar a una función con un número de argumentos inferior al especificado. Por ejemplo,

```
print_name('Olga', 'Puchmajerova')
print_name('Olga', 'Puchmajerova', True)
print_name('Olga', 'Puchmajerova', reverse = True)
```

imprimirá

```
Olga Puchmajerova
Puchmajerova, Olga
Puchmajerova, Olga
```

Las dos últimas invocaciones de `print_name` son semánticamente equivalentes. La última tiene la ventaja de proporcionar algo de documentación para el quizás misterioso argumento True. En términos más generales, el uso de argumentos de palabra clave reduce el riesgo de vincular involuntariamente un parámetro real al parámetro formal incorrecto. La línea de código

```
print_name(last_name = 'Puchmajerova', first_name = 'Olga')
```

no deja ninguna ambigüedad sobre la intención del programador que la escribió. Esto es útil porque llamar a una función con los argumentos correctos en el orden incorrecto es un error común. El valor asociado a un parámetro por defecto se calcula en el momento de definir la función.

> **Ejercicio manual**
>
> Escribe una función `mult` que acepte uno o dos ints como argumentos. Si se llama con dos argumentos, la función imprime el producto de los dos argumentos. Si se llama con un argumento, imprime ese argumento.

Mi aproximación al ejercicio:

```
def mult(int1, int2 = None):
    if int2 is None:
        return int1
    else:
        return int1 * int2
```

Variable Número de argumentos

Python tiene una serie de funciones incorporadas que operan con un número variable de argumentos. Por ejemplo,

```
min(6,4)
min(3,4,1,6)
```

son ambos legales (y evalúan lo que crees que hacen). Python facilita a los programadores la definición de sus propias funciones que aceptan un número variable de argumentos. El operador de **descompresión** `*` permite que una función acepte un número variable de argumentos posicionales. Por ejemplo,

```
def mean(*args):
    # Assumes at least one argument and all arguments are numbers
    # Returns the mean of the arguments
    tot = 0
    for a in args:
        tot += a
    return tot/len(args)
```

imprime `1.5` `-1.0`. Tenga en cuenta que el nombre que sigue al `*` en la lista de argumentos no tiene por qué ser `args`. Puede ser cualquier nombre. Para la media, podría haber sido más descriptivo escribir `def mean(*numbers)`.

Alcance

Veamos otro pequeño ejemplo:

```
def f(x): # name x used as formal parameter
    y = 1
    x = x + y
    print('x =', x)
    return x

x = 3
y = 2
z = f(x) # value of x used as actual parameter

print('z =', z)
print('x =', x)
print('y =', y)
```

Cuando se ejecuta, este código imprime

```
x = 4
z = 4
x = 3
y = 2
```

¿Qué ocurre aquí? En la llamada de `f`, el parámetro formal `x` está ligado localmente al valor del parámetro real `x` en el contexto del

cuerpo de la función `f`. *Aunque los parámetros reales y formales tienen el mismo nombre, no son la misma variable*. Cada función define un *nuevo espacio de nombres*, también llamado **ámbito**. El parámetro formal `x` y la variable local `y` que se utilizan en `f` sólo existen dentro del ámbito de la definición de `f`. La sentencia de asignación `x = x + y` dentro del cuerpo de la función vincula el nombre local `x` al objeto `4`.

Las asignaciones en `f` no tienen ningún efecto sobre los enlaces de los nombres `x` e `y` que existen fuera del ámbito de `f`.

He aquí una forma de verlo:

1. En el nivel superior, es decir, el nivel del intérprete de comandos, una tabla de símbolos mantiene un registro de todos los nombres definidos en ese nivel y sus vinculaciones actuales.
2. Cuando se llama a una función, se crea una nueva tabla de símbolos (a menudo llamada marco de pila). Esta tabla mantiene un registro de todos los nombres definidos dentro de la función (incluidos los parámetros formales) y sus vinculaciones actuales. Si se llama a una función desde el cuerpo de la función, se crea otro marco de pila.
3. Cuando la función finaliza, su marco de pila desaparece.

En Python, siempre se puede determinar el ámbito de un nombre mirando el texto del programa. Esto se llama **ámbito estático o léxico**.
El siguiente código contiene un ejemplo que ilustra las reglas de ámbito de Python.

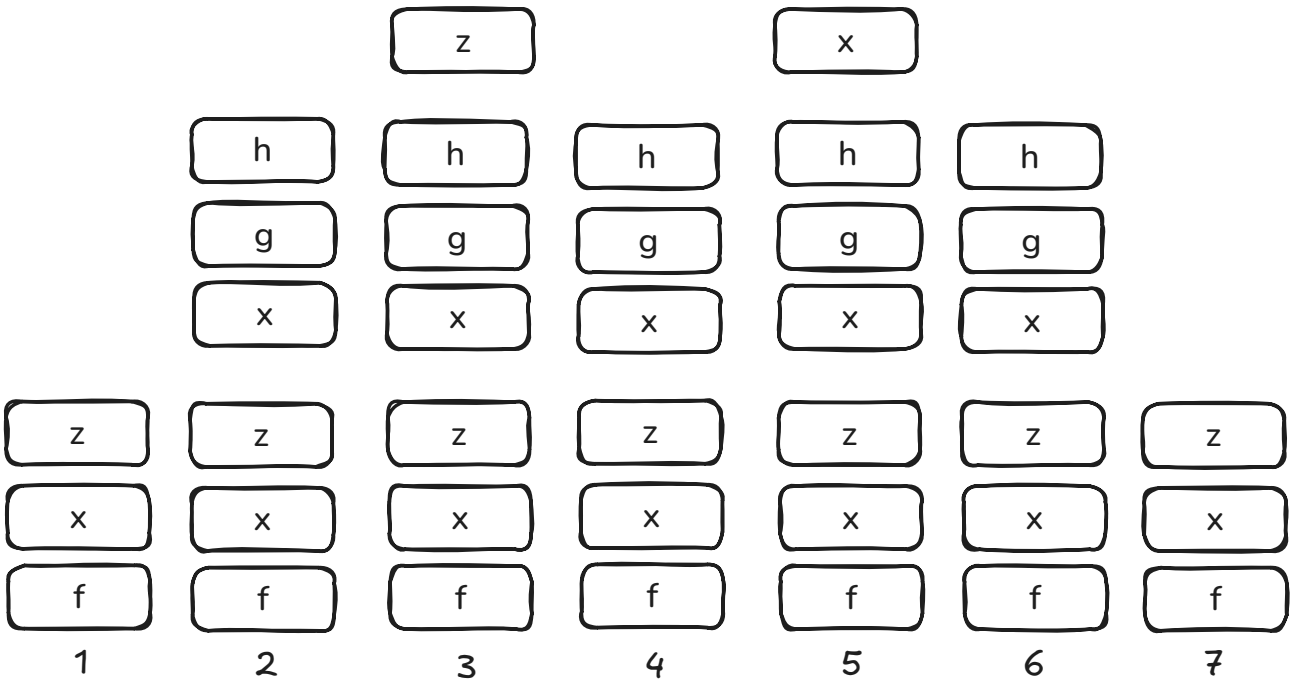
Ambitos anidados

```
def f(x):
    def g():
        x = "abc"
        print("x =", x)
    def h():
        z = x
        print("z =", z)
    x = x + 1
    print("x =", x)
    h()
    g()
    print("x =", x)
    return g

x = 3
z = f(x)
print("x =", x)
print("z =", z)
```

```
19  z()
```

La primera columna del código contiene el conjunto de nombres conocidos fuera del cuerpo de la función `f`, es decir, las variables `x` y `z`, y el nombre de la función `f`. La primera sentencia de asignación une `x` a `3`.



La sentencia de asignación `z = f(x)` evalúa primero la expresión `f(x)` invocando a la función `f` con el valor al que está ligado `x`. Cuando se introduce `f`, se crea un marco de pila, como se muestra en la columna 2. Los nombres en el marco de pila son `x` (el parámetro formal x, no la x en el contexto de llamada), `g`, y `h`. Las variables `g` y `h` están ligadas a objetos de tipo función. Las propiedades de estas funciones vienen dadas por las definiciones de función dentro de `f`.

Cuando `h` se invoca desde `f`, se crea otro marco de pila, como se muestra en la columna 3. Este marco contiene sólo la variable local `z`. ¿Por qué no contiene también `x`? Un nombre se añade al ámbito asociado a una función sólo si ese nombre es un parámetro formal de la función o una variable ligada a un objeto dentro del cuerpo de la función. En el cuerpo de `h`, `x` sólo aparece a la derecha de una sentencia de asignación. La aparición de un nombre (x en este caso) que no está ligado a un objeto en cualquier parte del cuerpo de la función (el cuerpo de h en este caso) hace que el intérprete busque en el marco de pila asociado con el ámbito dentro del cual se define la función (el marco de pila asociado con f). Si se encuentra el nombre (que es en este caso), se utiliza el valor al que está ligado (4). Si no se encuentra, aparece un mensaje de error.

Cuando `h` regresa, el marco de la pila asociado con la invocación de `h` desaparece (se retira de la parte superior de la pila), como se muestra en la columna 4. Tenga en cuenta que nunca eliminamos marcos de la mitad de la pila, sino sólo el marco añadido más recientemente. Tenga en cuenta que nunca eliminamos los fotogramas de la mitad de la pila, sino sólo el fotograma añadido más recientemente. Debido a este comportamiento «último en entrar, primero en salir» (LIFO), nos referimos a ella como una pila. (Imagina que cocinas una pila de tortitas. Cuando la primera sale de la plancha, el cocinero la coloca en un plato. A medida que cada tortita sale de la plancha, se apila sobre las tortitas que ya están en el plato. Cuando llegue el momento de comer las tortitas, la primera tortita servida será la que esté encima de la pila, la última que se añada -dejando la penúltima tortita añadida a la pila como la nueva tortita superior, y la siguiente que se sirva-).

Volviendo a nuestro ejemplo de Python, ahora se invoca a `g`, y se añade un marco de pila que contiene la variable local `x` de `g` (columna 5). Cuando `g` retorna, ese marco es vaciado (columna 6). Cuando `f` retorna, el marco de pila que contiene los nombres asociados con `f` es vaciado, devolviéndonos al marco de pila original (columna 7).

Observe que cuando `f` retorna, aunque la variable `g` ya no exista, el objeto de tipo función al que estaba ligado ese nombre sigue existiendo. Esto se debe a que las funciones son objetos, y pueden ser devueltas como cualquier otro tipo de objeto. Así, `z` puede estar ligado al valor devuelto por `f`, y la llamada a la función `z()` puede usarse para invocar la función que estaba ligada al nombre `g` dentro de `f` aunque el nombre `g` no se conozca fuera del contexto de `f`.

Entonces, ¿Qué imprime el código? Imprime

```
x = 4
z = 4
x = abc
x = 4
x = 3
z = <function f.<locals>.g at 0x1092a7510>
x = abc
```

El orden en que aparecen las referencias a un nombre no es relevante. Si un objeto está vinculado a un nombre en cualquier parte del cuerpo de la función (incluso si aparece en una expresión antes de que aparezca como el lado izquierdo de una asignación), se trata como local a esa función. Considere el código

```
def f():
    print(x)
def g():
    print(x)
    x = 1

x = 3
f()
x = 3
g()
```

Imprime 3 cuando se invoca `f`, pero el mensaje de error

```
UnboundLocalError: local variable 'x' referenced before assignment
```

se imprime cuando se encuentra la sentencia `print` en `g`. Esto ocurre porque la sentencia de asignación que sigue a la sentencia `print` hace que `x` sea local a `g`. Y como `x` es local a `g`, no tiene valor cuando se ejecuta la sentencia `print`.

Especificaciones

La **especificación** de una función define un contrato entre el implementador de una función y aquellos que escribirán programas que utilicen la función. Nos referimos a los usuarios de una función como sus clientes. Este contrato consta de dos partes:

- **Supuestos**: Describen las condiciones que deben cumplir los clientes de la función. Normalmente, describen restricciones sobre los parámetros reales. Casi siempre, especifican el conjunto aceptable de tipos para cada parámetro y, no pocas veces, algunas restricciones sobre el valor de uno o más parámetros. Por ejemplo, la especificación de `find_root` puede requerir que la potencia sea un número entero positivo.
- **Garantías**: Describen las condiciones que debe cumplir la función, siempre que haya sido invocada de forma que satisfaga los supuestos. Por ejemplo, la especificación de `find_root` podría garantizar que devuelve `None` si se le pide que encuentre una raíz que no existe (por ejemplo, la raíz cuadrada de un número negativo).

Las funciones son una forma de crear elementos computacionales que podemos considerar primitivos. Proporcionan descomposición y abstracción.

La **descomposición** crea estructura. Nos permite dividir un programa en partes que son razonablemente autónomas y que pueden reutilizarse en diferentes entornos.

La **abstracción** oculta los detalles. Nos permite utilizar un fragmento de código como si fuera una caja negra, es decir, algo cuyos detalles interiores no podemos ver, no necesitamos ver y ni siquiera deberíamos querer ver. La esencia de la abstracción es conservar la información que es relevante en un contexto determinado y olvidar la información que es irrelevante en ese contexto. La clave para utilizar la abstracción de forma eficaz en programación es encontrar una noción de relevancia que sea apropiada tanto para el constructor de una abstracción como para los clientes potenciales de la misma. Ése es el verdadero arte de la programación.

La *abstracción consiste en olvidar*. Hay muchas formas de modelar esto, por ejemplo, el aparato auditivo de la mayoría de los adolescentes.

- **Adolescente dice**: ¿Me prestas el coche esta noche?
- **El padre dice**: Sí, pero vuelve antes de medianoche y asegúrate de que el depósito de gasolina está lleno.
- **El adolescente oye**: Sí.

El adolescente ha ignorado todos esos detalles molestos que considera irrelevantes. La abstracción es un *proceso de muchos a uno*. Si el padre hubiera dicho «Sí, pero vuelve antes de las dos de la madrugada y asegúrate de que el coche está limpio», también se habría abstraído a Sí.

A modo de analogía, imagine que se le pide que elabore un curso introductorio de informática que contenga 25 conferencias. Una forma de hacerlo sería contratar a 25 profesores y pedirles que preparen una clase de una hora sobre su tema favorito. Aunque puede que consigas 25 horas maravillosas, es probable que el conjunto parezca una dramatización de Seis personajes en busca de autor, de Pirandello (o de ese curso de ciencias políticas al que asististe con 15 profesores invitados). Si cada profesor trabajara de forma aislada, no tendría ni idea de cómo relacionar el material de su conferencia con el material tratado en otras conferencias.

De alguna manera, es necesario que todo el mundo sepa lo que hacen los demás, sin generar tanto trabajo que nadie esté dispuesto a participar. Aquí es donde entra en juego la abstracción. Se pueden escribir 25 especificaciones, cada una de las cuales diga qué material deben aprender los estudiantes en cada clase, pero sin dar ningún

detalle sobre cómo debe enseñarse ese material. El resultado no sería pedagógicamente maravilloso, pero al menos tendría sentido.

Así es como las organizaciones utilizan los equipos de programadores para hacer las cosas. Dada la especificación de un módulo, un programador puede trabajar en la implementación de ese módulo sin preocuparse de lo que hacen los demás programadores del equipo. Además, los demás programadores pueden utilizar la especificación para empezar a escribir código que utilice el módulo sin preocuparse de cómo se implementará el módulo.

El siguiente código añade una especificación a la implementación de `find_root`.

Una función definida con especificaciones

```
1   def find_root(x, power, epsilon):
2
3       """Assumes x and epsilon int or float, power an int, epsilon > 0 &
    power >= 1
4          Returns float y such that y ** power is within epsilon of x.
5               If such a float does not exist, it returns None"""
6
7       # Find interval containing answer
8
9       if x < 0 and power % 2 == 0:
10          return None
11
12      low = min(-1, x)
13      high = max(1, x)
14
15      # Use bisection search
16
17      ans = (high + low) / 2
18
19      while abs(ans ** power - x) >= epsilon:
20          if ans ** power < x:
21              low = ans
22          else:
23              high = ans
24          ans = (high + low) / 2
25
26      return ans
```

El texto entre las comillas triples se llama `docstring` en Python. Por convención, los programadores de Python utilizan `docstrings` para proporcionar especificaciones de las funciones. Se puede acceder a estos `docstrings` usando la ayuda de funciones incorporada.

Una de las cosas buenas de los IDEs de Python es que proporcionan una herramienta interactiva para preguntar sobre los objetos incorporados.

Si quieres saber lo que hace una función específica, sólo tienes que escribir `help(objeto)` en la ventana de la consola. Por ejemplo, `help(abs)` produce el texto

```
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

Esto nos dice que `abs` es una función que asigna un único argumento a su valor absoluto. (La / en la lista de argumentos significa que el argumento debe ser posicional.) Si introduce `help()`, se inicia una sesión de ayuda interactiva, y el intérprete presentará el `prompt` `help>` en la ventana de la consola. Una ventaja del modo interactivo es que puede obtener ayuda sobre construcciones de Python que no son objetos. Por ejemplo

```
help> if
The "if" statement
******************

The "if" statement is used for conditional execution:

    if_stmt ::= "if" expression ":" suite
        ("elif" expression ":" suite)*
        ["else" ":" suite]

It selects exactly one of the suites by evaluating the
expressions one
by one until one is found to be true (see section Boolean
operations
for the definition of true and false); then that suite is
executed
(and no other part of the "if" statement is executed or
evaluated).
If all expressions are false, the suite of the "else"
clause, if
present, is executed.

Related help topics: TRUTHVALUE
```

Se puede salir de la ayuda interactiva tecleando `quit`.
Si el código se hubiera cargado en un IDE, al escribir `help(find_root)` en el intérprete de comandos aparecería

```
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
```

La especificación de `find_root` es una abstracción de todas las posibles implementaciones que la cumplen. Los clientes de `find_root` pueden asumir que la implementación cumple con la especificación, pero no deben asumir nada más. Por ejemplo, pueden asumir que la llamada `find_root(4, 2, 0.01)` devuelve un valor cuyo cuadrado está entre `3.99` y `4.01`. El valor devuelto puede ser positivo o negativo, y aunque 4 es un cuadrado perfecto, podría no ser 2 ni -2. Fundamentalmente, si no se cumplen las suposiciones de la especificación, no se puede asumir nada sobre el efecto de llamar a la función. Por ejemplo, la llamada `find_root(8, 3, 0)` podría devolver 2. Pero también podría bloquearse, ejecutarse indefinidamente o devolver un número muy alejado de la raíz cúbica de 8.

Ejercicio manual

Utilizando el algoritmo del código "*Usando la busqueda por biseccion para estimar log base 2*", escriba una función que satisfaga la especificación:

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

    lower_bound = 0
    while base ** lower_bound < x:
        lower_bound += 1

    low = lower_bound - 1
    high = lower_bound + 1

    # Búsqueda por bisección
    ans = (high + low) / 2

    while abs(base ** ans - x) >= epsilon:
        if base ** ans < x:
            low = ans
        else:
            high = ans
```

```
20          ans = (high + low) / 2
21
22      return ans
```

Ejercicio manual de la lectura

Implemente la función que cumpla con las especificaciones a continuación:

Mi aproximación

```
1   def eval_quadratic(a, b, c, x):
2       """
3       a, b, c: numerical values for the coefficients of a quadratic equation
4       x: numerical value at which to evaluate the quadratic.
5       Returns the value of the quadratic a×x² + b×x + c.
6       """
7       # Your code here
8       return a * x ** 2 + b * x + c
9
10  # Examples:
11  print(eval_quadratic(1, 1, 1, 1)) # prints 3
```

```
1   def two_quadratics(a1, b1, c1, x1, a2, b2, c2, x2):
2       """
3       a1, b1, c1: one set of coefficients of a quadratic equation
4       a2, b2, c2: another set of coefficients of a quadratic equation
5       x1, x2: values at which to evaluate the quadratics
6       Evaluates one quadratic with coefficients a1, b1, c1, at x1.
7       Evaluates another quadratic with coefficients a2, b2, c2, at x2.
8       Prints the sum of the two evaluations. Does not return anything.
9       """
10      # Your code here
11      result = eval_quadratic(a1, b1, c1, x1) + eval_quadratic(a2, b2, c2,
    x2)
12      print(result)
13      return result
14
15  # Examples:
16  two_quadratics(1, 1, 1, 1, 1, 1, 1, 1) # prints 6
17  print(two_quadratics(1, 1, 1, 1, 1, 1, 1, 1)) # prints 6 then None
```