Estudio de caso: números primos

Juan Álvarez Rubio

En esta sección se aborda un problema que ilustra la metodología de programación y las facilidades del lenguaje Python revisadas hasta ahora. Se resolverán varios problemas relacionados con los números primos que necesitan métodos computacionales para resolverlos: la primalidad, es decir, determinar si un número entero es o no un número primo; la factorización de un número entero en sus factores primos; y la coprimalidad, es decir, la comprobación de si dos números son primos relativos o primos entre sí.

Primalidad

Un número entero positivo mayor que 1 es un número primo si solo es divisible por sí mismo y por la unidad. Por ejemplo, 13 es un número primo porque solo es divisible por 13 y por 1. El número 2 es el único número par que es primo porque es divisible solo e por 2 y por 1, pero cualquier otro número par no es primo porque es divisible por 2. Por lo tanto, la siguiente es la receta de diseño de una función lógica que determina si un número es o no primo:

```
#esPrimo: int -> bool
#True si n>1 es primo
#ej: esPrimo(2)->True  (par primo)
#ej: esPrimo(6)->False (par no primo)
#ej: esPrimo(13)->True (impar primo)
#ej: esPrimo(9)->False (impar no primo)
def esPrimo(n):
    …
assert esPrimo(2)
assert not esPrimo(6)
assert esPrimo(13)
assert not esPrimo(9)
```

La función debe distinguir si el número es par o impar consultando si es divisible o no por 2, examinando si el resto de la división por 2:

```
def esPrimo(n):
  assert type(n)==int and n>1
  if n%2==0:
      … #numeros pares
  else:
      … #numeros impares
```

Si el número n es par, entonces solo el 2 es primo. Si el número es impar entonces se necesita comprobar si es divisible por alguno de los números impares sucesivos 3, 5, 7, …, etc. Hasta que divisor probar admite distintas alternativas: n, n/2, raíz cuadrada de n. El último caso es el más eficiente y tiene una justificación matemática: si existe un divisor mayor que √n entonces tendría que existir también un divisor menor que √n. Por lo tanto, se necesita una función interna recursiva que pruebe los posibles divisores:

```
def esPrimo(n):
  assert type(n)==int and n>1
  if n%2==0:
      #numero par
      return n==2 #2 es el único par primo
  else:
      #numero impar
      def _esPrimo(n,divisor):
          if divisor*divisor > n:
              return True
          elif n % divisor == 0:
              return False
          else:
              return _esPrimo(n,divisor+2)
      return _esPrimo(n,3)
```

La función interna _esPrimo recibe en el parámetro divisor inicialmente el impar 3 que se incrementa en 2 en cada llamada recursiva. La condición `divisor*divisor>n` permite detectar el término realizando una sola multiplicación entre enteros. Alternativamente, la condición equivalente `divisor>math.sqrt(n)` calcularía la raíz cuadrada con un algoritmo de aproximaciones sucesivas de números reales.

Las dos funciones pueden escribirse en menos líneas utilizando instrucciones if sin else. Adicionalmente, la función interna _esPrimo define el parámetro divisor con el valor inicial 3, de manera que la primera vez puede invocarse omitiendo el segundo argumento:

```
def esPrimo(n):
  assert type(n)==int and n>1
  if n%2==0: return n==2
  def _esPrimo(n,divisor=3):
      if divisor*divisor > n: return True
      if n % divisor == 0: return False
      return _esPrimo(n,divisor+2)
  return _esPrimo(n)
```

Las funciones pueden reescribirse utilizando solo expresiones de tipo bool. Al respecto, un número primo debe cumplir con la condición de tener el valor 2 o ser impar primo. La función interna también puede evaluar una sola condición:

```
def esPrimo(n):
  assert type(n)==int and n>1
  def _esPrimo(n,divisor=3):
    return divisor*divisor>n \
        or n%divisor!=0 and  _esPrimo(n,divisor+2)
  return n%2!=0 and _esPrimo(n) or n==2
```

La función recibirá habitualmente argumentos impares por lo que la condición primero evalúa si es un número impar primo. Al respecto, _esPrimo(n) se evalúa solo si la primera condición es True, es decir si n es impar. Por su parte, la condición n==2 se evalúa solo si el operador and entrega False.

Usos de la función de primalidad

La función esPrimo facilita la resolución de otros problemas. Por ejemplo, para mostrar los primos en un rango de enteros se puede escribir la siguiente función:

```
#primosEnRango: int int ->
#escribir primos entre x e y
#ej primosEnRango(2,10) escribe 2 3 5 7
def primosEnRango(x,y):
    assert type(x)==int and x>=2 and type(y)==int
    if x>y: return
    if esPrimo(x): print(x)
    primosEnRango(x+1,y)
```

La función esPrimo también puede utilizarse en una función que entrega el primo siguiente a un número:

```
#siguientePrimo: int -> int
#primo siguiente a n > 1
#ej: siguientePrimo(7)->11, siguientePrimo(2)->3
def siguientePrimo(n):
    assert type(n)==int and n>1
    if n%2==0:
        sgte=n+1
    else:
        sgte=n+2
    if esPrimo(sgte):
        return sgte
    else:
        return siguientePrimo(sgte)
```

```
assert siguientePrimo(7)==11
assert siguientePrimo(2)==3
```

La función se puede optimizar probando sólo con los números impares, es decir, avanzando de dos en dos, hasta encontrar un número primo (situación que las matemáticas garantizan que siempre ocurrirá).

```
def siguientePrimo(n):
    assert type(n)==int and n>1
    def _siguientePrimo(n):
        if esPrimo(n): return n
        return _siguientePrimo(n+2)
    if n%2==0:
        return _siguientePrimo(n+1)
    else:
        return _siguientePrimo(n+2)
```

Factores Primos

Los números primos son la fuente primaria para la representación de los números enteros, es decir, todo número entero se puede representar como un producto de factores primos. Por ejemplo, el número 24 se puede representar como 2 x 2 x 2 x 3, es decir, como el producto de 2^3 y 3^1, en que los factores 2 y 3 son números primos. En el caso de un número primo el único factor es el mismo número, por ejemplo, 5 es 5^1.

La función factoresPrimos escribirá los factores primos de un número en la forma indicada en los siguientes ejemplos:

factoresPrimos (24)
factor: 2 potencia: 3
factor: 3 potencia: 1

factoresPrimos(5)
factor: 5 potencia: 1

La función determinará los factores primos de un número probando con los números primos 2, 3, 5, 7, 11, etc. El primer factor primo 2, se puede asignar como el valor por omisión de un parámetro, y los siguientes se pueden generar con la función siguientePrimo:

```
#factoresPrimos: int ->
#escribe factores primos de n>=1
#ej: factoresPrimos(24) escribe 2**3 y 3**1
def factoresPrimos(n,factor=2):
    assert type(n)==int and n>=1
```

```
#caso base
if n==1: return
#calcular y escribir potencia p de factor primo
…
#recursión
factoresPrimos(n//factor**p, siguientePrimo(factor))
```

El cálculo y escritura de la potencia del factor primo se puede implementar de la siguiente manera:

```
#calcular y escribir potencia p de factor primo
def potencia(n,factor):
    if n % factor != 0: return 0
    return 1 + potencia(n//factor, factor)
p=potencia(n,factor)
if p>0: print('factor:',factor,'potencia:',p)
```

La función interna recursiva de nombre potencia cuenta la cantidad de divisiones enteras que se pueden realizar entre el número y el factor primo y corresponde a la potencia del factor por la que hay que dividir el número original.

La función factoresPrimos puede utilizarse en una función que escriba la descomposición en factores primos de un rango de números enteros. Por ejemplo, la invocación factoresPrimosRango(25,30) produce el siguiente resultado:

```
factores primos de 25
factor: 5 potencia: 2

factores primos de 26
factor: 2 potencia: 1
factor: 13 potencia: 1

factores primos de 27
factor: 3 potencia: 3

factores primos de 28
factor: 2 potencia: 2
factor: 7 potencia: 1

factores primos de 29
factor: 29 potencia: 1

factores primos de 30
factor: 2 potencia: 1
factor: 3 potencia: 1
```

```
factor: 5 potencia: 1
```

La función que produce el resultado anterior se puede escribir de la siguiente manera:

```
#factoresPrimosRango: int int ->
#escribe factores primos de números entre x e y
#ej: factoresPrimosRango(25,30) escribe factores de 25,...,30
def factoresPrimosRango(x,y):
    assert type(x)==int and x>=2 and type(y)==int
    if x>y: return
    #escribir factores primos de x
    print('factores primos de',x)
    factoresPrimos(x)
    print() #escribe línea en blanco
    factoresPrimosRango(x+1,y)
```

Coprimalidad

Dos números enteros distintos mayores que 1 son coprimos (primos relativos o primos entre sí) si su único divisor común es la unidad. Por ejemplo, los números 4 y 9 son coprimos porque no tienen divisores comunes distintos de 1. Por su parte 18 y 24 no son primos relativos porque tienen los divisores comunes 2, 3 y 6. Por supuesto, dos números primos son también coprimos, por ejemplo 5 y 11.

Una manera sencilla de implementar una función que devuelva True si dos números son primos entre sí es usar una función que calcule el máximo común divisor, y si entrega un resultado igual a 1, entonces son coprimos:

```
from mcd import *
#sonCoprimos: int int -> bool
#True si x e y no tienen divisores comunes
#ej: sonCoprimos(4,9)->True
#ej: sonCoprimos(18,24)->False
def sonCoprimos(x,y):
  assert type(x)==int and x>1
  assert type(y)==int and y>1
  assert x != y
  return mcd(x,y)==1 #mcd: maximo común divisor
assert sonCoprimos(4,9)
assert not sonCoprimos(18,24)
```

La función sonCoprimos se puede utilizar para escribir todos los pares de números que son coprimos en un rango de enteros. Por ejemplo, la invocación coprimosEnRango(2,5) escribe:

```
2 3
2 5
3 4
3 5
4 5
```

La función necesita de una función interna recursiva que escribe los coprimos de un número del rango:

```
#coprimosEnRango: int int ->
#escribe coprimos entre números x e y
#ej: coprimosEnRango(2,5) escribe 2 3,2 5,3 4,3 5,4 5
def coprimosEnRango(x,y):
    assert type(x)==int and x>1
    assert type(y)==int and y>1
    #caso base
    if x>=y: return
    #escribe coprimos de x entre x+1 e y
    def coprimosDe(n,x,y):
        if x>y: return
        if sonCoprimos(n,x): print(n,x)
        coprimosDe(n,x+1,y)
    coprimosDe(x,x+1,y)
    #recursión
    coprimosEnRango(x+1,y) #recursión
```