06.JerarquiaHerencia

July 29, 2024

1 Jerarquía de clases y herencia

La jerarquía de clases es una familia de clases que heredan métodos y atributos entre sí.

Una jerarquía de clases es una familia de clases estrechamente relacionadas organizadas de manera jerárquica.

Un concepto clave es la *herencia*, lo que significa que las clases hijo pueden heredar atributos y métodos de las clases padre. Una estrategia típica es escribir una clase general como clase base (o clase principal) y luego dejar que los casos especiales se representen como subclases (clases secundarias). Este enfoque a menudo puede ahorrar mucha escritura y duplicación de código.

1.1 Ejemplo. Clases para líneas y parábolas

En este ejemplo crearemos una clase que represente y evalúe líneas rectas $y = c_0 + c_1 x$.

[ ]:
```
import numpy as np
from prettytable import PrettyTable

class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1

    def __call__(self, x):
        return self.c0 + self.c1 * x

    def table(self, a, b, n):
        """ Devuelve una tabla con n puntos dado a <= x <= b """
        tabla = PrettyTable(field_names = ['x', 'f(x)'], float_format='.6')
        for x in np.linspace(a, b, n):
            y = self(x)
            tabla.add_row([x, y])
        return tabla

    def __str__(self):
        return f'y = {self.c0} + {self.c1}x'
```

La clase tiene su constructor estándar y un método especial `__str__` para la impresión de la función:

[ ]:
```
linea = Line(1,2)
print(linea)
```

```
y = 1 + 2x
```

Tiene un método especial `__call__` que permite la evaluación de la función:

[ ]:
```
linea(3)
```

[ ]:
```
7
```

Y tiene un método table que construye una tabla tipo `PrettyTable` con la evaluación de la función en un rango de valores de acuerdo a los valores recibidos por los argumentos a, b, y n.

[ ]:
```
linea.table(a=1, b=12, n=20)
```

[ ]:
```
+-----------+-----------+
|     x     |    f(x)   |
+-----------+-----------+
|  1.000000 |  3.000000 |
|  1.578947 |  4.157895 |
|  2.157895 |  5.315789 |
|  2.736842 |  6.473684 |
|  3.315789 |  7.631579 |
|  3.894737 |  8.789474 |
|  4.473684 |  9.947368 |
|  5.052632 | 11.105263 |
|  5.631579 | 12.263158 |
|  6.210526 | 13.421053 |
|  6.789474 | 14.578947 |
|  7.368421 | 15.736842 |
|  7.947368 | 16.894737 |
|  8.526316 | 18.052632 |
|  9.105263 | 19.210526 |
|  9.684211 | 20.368421 |
| 10.263158 | 21.526316 |
| 10.842105 | 22.684211 |
| 11.421053 | 23.842105 |
| 12.000000 | 25.000000 |
+-----------+-----------+
```

Digamos que ahora se requiere escribir una clase similar para la evaluación de la parábola $y = c_0 + c_1x + c_2x^2$, el código será el siguiente:

[ ]:
```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0=c0, self.c1=c1, self.c2=c2

    def __call__(self, x):
```

```
        return self.c0 + self.c1*x + self.c2*x**2

    def table(self, a, b, n):
        """ Devuelve una tabla con n puntos dado a <= x <= b """
        tabla = PrettyTable(field_names = ['x', 'f(x)'], float_format='.6')
        for x in np.linspace(a, b, n):
            y = self(x)
            tabla.add_row([x, y])
        return tabla

    def __str__(self):
        return f'y = {self.c0} + {self.c1}x + {self.c2}x²'
```

Observe que la mayoría del código es la misma excepto por las partes que involucran a c_2. Quizá copiar, pegar y modificar el código no sea tardado ni problemático, pero tal repetición de código es una mala práctica.

Imáginese que fuese necesario cambiar la funcionalidad de la generación de la tabla o corregir un error en el código replicado, sería necesario cambiar en todos los lugares donde el código fue replicado. Esto sería lento, impráctico y una posible fuente de más errores.

Por ello, se puede reutilizar el código de la clase `Line` para la construcción de la clase `Parabola` mediante la herencia.

[ ]:
```
class Parabola(Line):
    pass
```

El término `pass` es una palabra clave que indica a Python que la clase se ha dejado intencionalmente vacía. Sin embargo, aunque no haya código en la definición de la clase Parabola, ésta clase no está vacía. De esta forma `Parabola` ha heredado los atributos c_0 y c_1 y los métodos `__init__`, `__call__`, `__str__` y `table` de la clase `Line`.

En este sentido se dice que `Line` es la clase base (padre o superclase) y `Parabola` es una subclase (clase hijo o clase derivada). En ese momento, `Parabola` es una copia exacta de `Line` pero es posible modificarla para los propósitos de su uso. Para ello es necesario agregar su propio constructor `__init__`, sus métodos `__call__` y `__str__` de acuerdo a lo requerido.

[ ]:
```
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        super().__init__(c0, c1)    # La clase Linea almacena c0 y c1
        self.c2 = c2

    def __call__(self, x):
        return super().__call__(x) + self.c2*x**2

    def __str__(self):
        return super().__str__() + f' + {self.c2}x²'
```

[ ]:
```
par = Parabola(1,2,3)
print(par.table(1,2,10))
print(par)
```

```
+----------+-----------+
|    x     |    f(x)   |
+----------+-----------+
| 1.000000 |  6.000000 |
| 1.111111 |  6.925926 |
| 1.222222 |  7.925926 |
| 1.333333 |  9.000000 |
| 1.444444 | 10.148148 |
| 1.555556 | 11.370370 |
| 1.666667 | 12.666667 |
| 1.777778 | 14.037037 |
| 1.888889 | 15.481481 |
| 2.000000 | 17.000000 |
+----------+-----------+
y = 1 + 2x + 3x²
```

Observe que para maximizar la reutilización del código se ha llamado los métodos de la superclase `Line` y añadido las partes faltantes. Siempre será posible tener acceso a los métodos del padre mediante la función `super()`.

Es posible llamar directamente al padre mediante su nombre, pero será necesario agregar en los argumentos el contexto del objeto mediante `self`. Esto debido a que el nombre genérico no tiene una relación con la subclase, mientras que con `super()` si se hace una referencia directa a la superclase.

Esto es:

```
super().__init__(c0, c1)        # super() hace referencia a la superclase del objeto
Line.__init__(self, c0, c1)     # Se hace referencia a la clase `Line`, que en general no tien
```

En términos generales sería lo siguiente:

```
SuperClassName.method(self, arg1, arg2, ...)
super(arg1, arg2, ...)
```

1.2 El verdadero significado de la herencia

Desde un punto de vista práctico, la herencia permite reutilizar código y minimizar la duplicidad del mismo. Pero desde un punto de vista teórico, la herencia representa la relación que hay entre dos clases.

Esto significa que si `Parabola` es una subclase de `Line`, un objeto `Parabola` también es un objeto de `Line`. En otras palabras, la clase `Parabola` es un caso especial de la clase `Line`, por lo tanto cualquier instancia de `Parabola` es una instancia de `Line` pero no viceversa.

[ ]:
```
l = Line(-1,1)
print(isinstance(l, Line))
```

```
print(isinstance(l, Parabola))
```

```
True
False
```

[ ]:
```
p = Parabola(1,2,3)
print(isinstance(p, Parabola))
print(isinstance(p, Line))
```

```
True
True
```

[ ]:
```
print(issubclass(Parabola, Line))
print(issubclass(Line, Parabola))
```

```
True
False
```

[ ]:
```
print(p.__class__ == Parabola)
print(p.__class__.__name__)
```

```
True
Parabola
```

Se ha dicho que una subclase es un caso especial de una superclase. En el ejemplo, la clase `Parabola` es un caso específico de la clase `Line`, sin embargo matemáticamente una parábola no es un caso específico de una linea, en realidad una línea es un caso específico de una parábola cuando $c_2 = 0$.

Dado este hecho quizá valga la pena redefinir las clases para corregir esta discrepancia.

[ ]:
```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0 = c0, self.c1, self.c2 = c0, c1, c2

    def __call__(self, x):
        return self.c0 + self.c1*x + self.c2*x**2

    def table(self, a, b, n):
        """ Devuelve una tabla con n puntos dado a <= x <= b """
        tabla = PrettyTable(field_names = ['x', 'f(x)'], float_format='.6')
        for x in np.linspace(a, b, n):
            y = self(x)
            tabla.add_row([x, y])
        return tabla

    def __str__(self):
        return f'y = {self.c0} + {self.c1}x + {self.c2}x²'

class Line(Parabola):
    def __init__(self, c0, c1):
```

```
        super().__init__(c0, c1, 0)
```

1.2.1 Ejemplo: Diferenciación Numérica

Una tarea común en el cómputo científico es la diferenciación e integración que pueden ser resueltas por diversos métodos numéricos. Muchos de esos métodos están muy relacionados entre sí, por lo que pueden ser agrupados en familias de métodos. Una fórmula simple de cálculo es la siguiente:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

que puede ser implementada por la clase:

[ ]:
```
class Derivative:
    def __init__(self, f, h=1e-5):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h)-f(x))/h
```

Para hacer uso del método, sólo creamos una función y una instancia de la clase. Posteriormente hacemos una llamada a la instancia con el valor a evaluar. Por ejemplo:

$$f(x) = e^{-x} \sin(4\pi x)$$
$$f'(1.2) = ?$$

[ ]:
```
from math import exp, sin, pi

def f(x):
    return exp(-x) * sin(4*pi*x)

dfdx = Derivative(f)
print(dfdx(1.2))
```

```
-3.239208844119101
```

Sim embargo, dado que hay diversas fórumlas para encontrar la aproximación a la derivada:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$
$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h},$$
$$f'(x) \approx \frac{4}{3}\frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3}\frac{f(x+2h) - f(x-2h)}{4h}$$

entre otras.

Por ello se puede escribir fácilmente un módulo que ofrezca múltiples fórmulas.

[ ]:
```
class Forward:
    def __init__(self, f, h=1e-5):
        self.f, self.h = f, h

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h)-f(x))/h

class Central2:
    def __init__(self, f, h=1e-5):
        self.f, self.h = f, h

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h)-f(x-h))/2/h

class Central4:
    def __init__(self, f, h=1e-5):
        self.f, self.h = f, h

    def __call__(self, x):
        f, h = self.f, self.h
        return 4/3* (f(x+h) - f(x-h))/(2*h) - 1/3 * (f(x+2*h) - f(x-2*h))*(4*h)
```

El problema en este código es la repetición de código para el constructor. Para resolverlo se puede crear una superclase que contenga el constructor e implementar una subclase por método.

La superclase queda de la siguiente forma:

[ ]:
```
class Diff:
    def __init__(self, f, h=1e-5):
        self.f, self.h = f, h
```

Y las subclases se definen de la siguiente forma:

[ ]:
```
class Forward(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h

class Central2(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/2/h

class Central4(Diff):
    def __call__(self, x):
```

```
        f, h = self.f, self.h
        return (4/3) * (f(x+h) - f(x-h))/(2*h) - (1/3) * (f(x+2*h) - f(x-2*h))/
↪(4*h)
```

[ ]:
```
from math import exp, sin, pi

def f(x):
    return exp(-x) * sin(4*pi*x)

dfdx = Forward(f)
print(dfdx(1.2))

dfdx = Central2(f)
print(dfdx(1.2))

dfdx = Central4(f)
print(dfdx(1.2))
```

```
-3.239208844119101
-3.2391005667389834
-3.2391005760477407
```

Ahora hagamos una concentración de aproximaciones con todas las fórmulas:

[ ]:
```
# from Diff import Forward, Central2, Central4
from math import pi, sin, cos
from prettytable import PrettyTable

H = [(1/2)**i for i in range(10)]
x0 = pi/4

table = PrettyTable(field_names=['h', 'Forward', 'Central 2', 'Central 4'])
for h in H:
    f1 = Forward(sin, h)
    c2 = Central2(sin, h)
    c4 = Central4(sin, h)
    table.add_row([h, f1(x0), c2(x0), c4(x0)])

print(table)
```

```
+-------------+--------------------+--------------------+--------------------+
|      h      |      Forward       |     Central 2      |     Central 4      |
+-------------+--------------------+--------------------+--------------------+
|     1.0     | 0.2699544827129282 | 0.5950098395293859 | 0.6861847232685281 |
|     0.5     | 0.5048856975964859 | 0.6780100988420897 | 0.7056768519463243 |
|     0.25    | 0.611835119448811  | 0.6997640691250939 | 0.707015392552762  |
|    0.125    | 0.6611301360648314 | 0.7052667953545546 | 0.7071010374310415 |
|    0.0625   | 0.6845566203276636 | 0.7066465151141275 | 0.7071064217006517 |
|   0.03125   | 0.6959440534591259 | 0.706991697811663  | 0.7071067587108415 |
```

```
|   0.015625  | 0.7015538499518499 | 0.7070780092891873 | 0.7071067797816953 |
|  0.0078125  | 0.7043374663312676 | 0.7070995881463489 | 0.7071067810987361 |
|  0.00390625 | 0.705723916746507  | 0.7071049829223881 | 0.7071067811810678 |
| 0.001953125 | 0.706415797873774  | 0.7071063316202526 | 0.7071067811862074 |
+-------------+--------------------+--------------------+--------------------+
```

2 Programación funcional

2.1 Funciones recursivas

La recursividad es un fenómeno presente en la naturaleza. La recursividad es un concepto donde una función, procedimiento o proceso se define en términos de sí mismo. Es una técnica fundamental en matemáticas y ciencias de la computación, pero también se puede observar en la naturaleza y en otros campos.

2.1.1 Ejemplos de Recursividad en la Naturaleza:

1. **Fractales:**
   - **Helechos:** Las hojas de un helecho son ejemplos clásicos de recursividad. Cada hoja está compuesta de pequeñas hojas que tienen una forma similar a la hoja completa.
   - **Brócoli Romanesco:** Este vegetal tiene una estructura fractal, donde cada florete se asemeja a una versión en miniatura de la planta completa.
2. **Conchas Marinas:**
   - Muchas conchas marinas, como las de los nautilos, muestran patrones recursivos en su estructura, donde el crecimiento de la concha sigue un patrón logarítmico.
3. **Ramas de Árboles:**
   - La ramificación de los árboles es un ejemplo de recursividad. Cada rama principal se divide en ramas más pequeñas, y estas a su vez se dividen en ramas aún más pequeñas, siguiendo un patrón similar.
4. **Ríos y Deltas:**
   - Los sistemas fluviales también muestran recursividad. Un río principal se divide en afluentes, y estos a su vez en arroyos más pequeños, siguiendo una estructura jerárquica.
5. **Sistemas Circulatorios:**
   - El sistema circulatorio de muchos organismos, donde los grandes vasos sanguíneos se dividen en vasos más pequeños y así sucesivamente, es un ejemplo de recursividad biológica.
6. **Nervios y Neuronas:**
   - Las estructuras neuronales, con dendritas que se ramifican en subdendritas y axones que se ramifican en terminales más pequeños, también son recursivas.

2.1.2 Concepto Matemático:

En matemáticas, un ejemplo clásico de recursividad es la secuencia de Fibonacci, donde cada término se define como la suma de los dos términos anteriores:

$$F(n) = F(n-1) + F(n-2) \forall n \geq 2$$

con $F(0) = 0$ y $F(1) = 1$.

La recursividad es una herramienta poderosa porque permite resolver problemas complejos dividiéndolos en subproblemas más simples del mismo tipo.

2.1.3 Ejemplo: Serie de Fibonacci

[ ]:
```python
# Serie de Fibonacci

def Fibonacci(n):
    if n==0 or n==1:
        return n
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)

for i in range(20):
    print(Fibonacci(i))
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
```

2.1.4 Razón áurea (proporción dorada).

La razón áurea, también conocida como número áureo o proporción dorada, es un número irracional denotado por la letra griega phi (φ).

Su valor es aproximadamente 1.6180339887.

Se define algebraicamente como:

$$\varphi = \frac{1+\sqrt{5}}{2}$$

2.1.5 Propiedades:

1. **Proporción Ideal:** El número surge de la división en dos de un segmento guardando las siguientes proporciones: La longitud total $a + b$ es al segmento más largo a, como a es al segmento más corto b, entonces esa proporción es la razón áurea:

$$\frac{a+b}{a} = \frac{a}{b} = \varphi$$

2. **Aparición en la Naturaleza:** La razón áurea aparece en diversas formas en la naturaleza, como en las proporciones de las conchas de nautilo, la disposición de las hojas, y las espirales de las piñas.

3. **Uso en Arte y Arquitectura:** Ha sido utilizada en obras de arte y arquitectura para crear estéticamente agradables proporciones, como en el Partenón de Atenas o en las obras de Leonardo da Vinci.

4. **Relación con la Secuencia de Fibonacci:** La razón áurea se relaciona con la serie de Fibonacci a través de la relación de los términos consecutivos de la serie. A medida que avanzas en la secuencia de Fibonacci, la proporción de un número con el anterior se aproxima a la razón áurea.

$$\lim_{n\to\infty} \frac{F(n+1)}{F(n)} = \varphi$$

   Este límite tiende a \$ 1.6180339887\$ cuando n tiende a infinito.

La razón áurea ha fascinado a matemáticos, artistas y arquitectos durante siglos debido a su presencia en patrones naturales y su aplicación en el diseño.

Número áureo

[ ]:
```
n=35
Fibonacci(n)/Fibonacci(n-1)
```

[ ]:
```
1.6180339887499087
```

2.1.6 Ejemplo: Suma recursiva

Para sumar una lista de números se puede utilizar la función nativa `sum` de Python, pero hagamos una propia que utilice el concepto de recursividad.

[ ]:
```
def miSuma(L):
    if not L:
        return 0
    else:
        return L[0] + miSuma(L[1:])

print(miSuma([1,2,3,4,5]))
print(miSuma(list(range(101))))
```

```
15
5050
```

Una versión alternativa del código empleando la expresión tenaria `if-else` de Python.

[ ]:
```python
def miSuma(L):
    return 0 if not L else L[0] + miSuma(L[1:])

print(miSuma([1,2,3,4,5]))
print(miSuma(list(range(101))))
```

```
15
5050
```

Otra versión que soporta no sólo números.

[ ]:
```python
def miSuma(L):
    return L[0] if len(L) == 1 else L[0] + miSuma(L[1:])


print(miSuma([1,2,3,4,5]))
print(miSuma(list(range(101))))
print(miSuma(['hola', 'mundo', 'cruel']))
```

```
15
5050
holamundocruel
```

Una versión que además de no sólo soportar números, utiliza el desempaquetado de la lista.

[ ]:
```python
def miSuma(L):
    first, *rest = L    # Desempaquetado: Toma el primer elemento de la lista L␣
 ↪y lo asigna a first, mientras que el resto de los elementos se asignan a␣
 ↪rest como una lista
    return first if not rest else first + miSuma(rest)

print(miSuma([1,2,3,4,5]))
print(miSuma(list(range(101))))
print(miSuma(['hola', 'mundo', 'cruel']))
```

```
15
5050
holamundocruel
```

2.2 Recursividad vs Ciclos

La recursividad y los ciclos son dos enfoques para repetir acciones en un programa. Aquí están sus diferencias:

2.2.1 Recursividad

1. **Definición:**
   - Una función se llama a sí misma para resolver subproblemas más pequeños.
2. **Estructura:**
   - Requiere un caso base para detener las llamadas recursivas.
   - Cada llamada crea un nuevo contexto en la pila de llamadas.
3. **Ventajas:**
   - Más intuitiva para problemas que tienen una naturaleza divisoria, como el recorrido de árboles o la generación de fractales.
   - Puede ser más fácil de entender y escribir para ciertos problemas.
4. **Desventajas:**
   - Puede causar desbordamiento de pila si la profundidad de la recursión es muy grande.
   - A menudo es menos eficiente en términos de memoria y tiempo debido a las múltiples llamadas a funciones.

2.2.2 Ciclos

1. **Definición:**
   - Utilizan estructuras de control (como `for`, `while`) para repetir acciones.
2. **Estructura:**
   - No involucran llamadas repetidas de función.
   - Mantienen el control en un único contexto de ejecución.
3. **Ventajas:**
   - Más eficiente en términos de uso de memoria, ya que no hay múltiples contextos.
   - Evita problemas de desbordamiento de pila.
4. **Desventajas:**
   - A veces puede ser menos intuitivo para problemas que se modelan naturalmente de manera recursiva.

Podría pensarse que las estructuras cíclicas son más sencillas y pueden realizar todo aquello que las formas recursivas pueden. Por ejemplo, la versión cíclica de la función suma anterior es:

[ ]:
```
# Suma cíclica
L = [1, 2, 3, 4, 5]

sum = 0
while L:
    sum += L[0]
    L = L[1:]

sum
```

[ ]:
```
15
```

[ ]:
```
# Suma cíclica, versión alternativa
L = [1, 2, 3, 4, 5]

sum = 0
```

```
for x in L:
    sum += x

sum
```

[ ]: 15

Sin embargo hay ocasiones donde la forma recursiva es más simple que una cíclica. Por ejemplo, considere el caso de la suma de números en una lista anidada:

`[1, [2, [3, 4], 5], 6, [7, 8], [[9], [10]]]`

En casos como este la programación cíclica no sirve dado que no es una iteración lineal. Para ello sería necesario agregar más código de manera que los ciclos puedan funcionar. Por otro lado, una versión recursiva se acomoda mejor debido a la construcción recursiva de la propia lista anidada.

Veamos el código:

[ ]:
```
def miSuma(L):
    total = 0
    for x in L:
        if not isinstance(x, list):
            total += x
        else:
            total += miSuma(x)
    return total

print(miSuma([1, [2, [3, 4], 5], 6, [7, 8], [[9], [10]]]))
```

```
55
```

2.3 Llamada indirecta de funciones

Debido a que en Python las funciones son objetos, se pueden escribir programas que las procesen de forma genérica. Las funciones como objeto pueden ser asignadas a otro nombre, pasadas a otras funciones, incrustarlas en estructuras de datos, devueltas por una función, entre otras cosas más.

El nombre utilizado al declarar una función a través de `def`, es sólo una variable asignada al contexto actual. Luego de que el operador `def` sea ejecutado, el nombre sólo es una referencia al objeto.

[ ]:
```
# Llamada directa
def echo(message):
    print(message)

echo("Hola mundo cruel")
```

```
Hola mundo cruel
```

[ ]:
```
# Llamada indirecta
x = echo
x("Hola mundo cruel")
```

```
Hola mundo cruel
```

Debido a que los argumentos se pasan como una asignación de objetos, es posible pasar funciones como argumento para otras funciones.

[ ]:
```
def indirect(func, arg):
    func(arg)

indirect(echo, "Llamada por argumento")
```

```
Llamada por argumento
```

Incluso es posible incluir funciones en estructuras de datos, como si fueran números enteros o cadenas. El siguiente ejemplo, incluye la función dos veces en una lista de tuplas, como una especie de tabla de acciones. Debido a que los tipos compuestos de Python como estos pueden contener cualquier tipo de objeto, tampoco hay ningún caso especial aquí:

[ ]:
```
myList = [(echo, "Hola"), (echo, "mundo"), (echo, "cruel")]
for (func, arg) in myList:
    func(arg)
```

```
Hola
mundo
cruel
```

En el siguiente ejemplo, se define una función como y se devuelve resultado de otra función.

[ ]:
```
def make(label):
    def echo(message):
        print(label + ':' + message)
    return echo

F = make("Hola")
F(" Mundo Cruel")
```

```
Hola: Mundo Cruel
```

En este ejemplo, se define una función `make` que recibe un argumento `label` que se pasa como argumento a la función `echo`. Esta última función recibe el argumento, forma un mensaje y lo imprime. Finalmente la función make devuelve la función `echo`.

2.4 Funciones anónimas `lambda`

Además del parámetro `def` para definir funciones, Python ofrece la expresión `lambda` para definir funciones. Al igual que con `def`, `lambda` permite crear una función para ser utilizada posteriormente pero devuelve la función en lugar de asignarlo a un nombre. Es por ello que se les conoce como *funciones anónimas*.

Para declarar una función anónima, se utiliza la palabra clave `lambda` seguida de uno o más argumentos, y separado con dos puntos la expresión a evaluar por la función.

```
lambda argument1, argument2, ..., argumentN : expression using arguments
```

Las funciones `lambda` operan de forma idéntica a las funciones `def`, pero existen algunas diferencias entre ellas.

- **`lambda` es una expresión, no una sentencia**. Debido a esto, las funciones lambda pueden aparecer en lugares donde la sintáxis de Python no lo permite (dentro de una lista, en los argumentos de una función, entre muchos otros). Debido a que es una expresión, devuelve un valor (una función) que puede ser opcionalmente asignada a un nombre. A diferencia de `def` que siempre asigna la función al nombre indicado.
- **El cuerpo de `lambda` es una sola expresión, no un bloque de sentencias.** El contenido de las funciones `lambda` es similar al de las funciones `def`, pero sólo se escribe la expresión a evlauar sin ser necesario devolverlo explícitamente. Dado que las funciones `lambda` están limitadas a una sola expresión, son menos generales que las funciones `def`. Esta limitante es intencional por diseño, de manera que `lambda` está diseñada para usarse para operaciones simples y `def` para operaciones más largas.

A pesar de sus diferencias, las funciones `def` y `lambda` pueden hacer el mismo trabajo:

[ ]:
```
# Función def
def funcion(x,y,z):
    return x+y+z
funcion(3,4,5)
```

[ ]:
```
12
```

[ ]:
```
# Función lambda
funcion = lambda x,y,z : x+y+z
funcion(3,4,5)
```

[ ]:
```
12
```

Otro ejemplo con función `lambda`:

[ ]:
```
def knights():
    title = 'Sir'
    action = lambda x : title + ' ' + x
    return action

act = knights()
act("Robin")
```

[ ]:
```
'Sir Robin'
```

En este ejemplo, la función `knigths` devuelve una función `lambda`. Por ello `act` es nombre de variable que apunta a un objeto función `lambda` asociado en la llamada indirecta. Cuado se invoca `act("Robin")`, la función `lambda` recibe el argumento en `x` y realiza la operación definida.

[ ]:
```
act
```

[ ]:
```
<function __main__.knights.<locals>.<lambda>(x)>
```

2.4.1 ¿Porqué utilizar funciones `lambda`?

Este tipo de funciones resultan útiles en casos donde se quiere incrustar la definición de la función dentro del código que la utiliza. Aunque su uso es opcional, tienden a generar código más simple en escenarios donde se necesita crear bloques de código concisos e independientes.

Las funciones lambda se utilizan comunmente para codificar *jump tables*, que son listas o diccionarios de acciones que se deben realizar bajo demanda. Por ejemplo:

[ ]:
```
L = [lambda x:x**2, lambda x:x**3, lambda x:x**4]   # Lista de 3 funciones

for f in L:
    print(f(2))

print(L[0](3))
```

```
4
8
16
9
```

Este código es muy simple e intuitivo, sin embargo, es posibble obtener el mismo resultado mediante funciones `def`.

[ ]:
```
def f1(x): return x**2
def f2(x): return x**3
def f3(x): return x**4

L = [f1, f2, f3]

for f in L:
    print(f(2))
```

```
4
8
16
```

El resultado es el mismo pero el código es más simple.

Para el caso de los diccionarios veamos el ejemplo:

[ ]:
```
key = 'got'
D = {
    'already': lambda x: x+4,
    'got': lambda x: x*4,
    'one' : lambda x: x**4
}

D[key](3)
```

[ ]: 12

Ahora veamos el caso de funciones `lambda` anidadas, pero primero hagamos un ejemplo con uina función `def` y dentro una `lambda`.

[ ]:
```
def action(x):
    return lambda y : x+y

act = action(99)
print(act)
act(2)
```

```
<function action.<locals>.<lambda> at 0x7fa3ccd81580>
```

[ ]:
```
101
```

Ahora la misma operación pero con un par de funciones `lambda`, en una sola línea.

[ ]:
```
action = lambda x : (lambda y : x+y)
act = action(99)
act(3)
```

[ ]:
```
102
```

2.5 Funciones `map` sobre secuencias

Otra tarea común al utilizar listas es aplicar una operación a todos los elementos y recolectar un resultado. Por ejemplo, actualizar todos los elementos en una lista se puede hacer fácilmente con un ciclo empleando programación convencional.

[ ]:
```
counters = [1,2,3,4]
updated = []

for x in counters:
    updated.append(x+10)
updated
```

[ ]:
```
[11, 12, 13, 14]
```

Pero dado que este tipo de operaciones son muy comúnes, Python ofrece una función nativa que hace este trabajo. La función `map` aplica una función pasada com oargumento a cada uno de los elementos de una lista y devuelve otra lista con los resultados. Por ejemplo:

[ ]:
```
def inc(x):
    return x+10

list(map(inc, counters))
```

[ ]:
```
[11, 12, 13, 14]
```

La función `map` llama a la función pre-existente inc para cada uno de los elementos de la lista y los almacena en una nueva lista.

Dado que `map` espera recibir una función como argumento, es un buen lugar para utilizar una función `lambda`.

[ ]:
```
counters = [1,2,3,4]
list(map(lambda x : x+10, counters))
```

[ ]:
```
[11, 12, 13, 14]
```

Es posible replicar el funcionamiento de `map` con programación ciclos y programación convencional. Sin embargo `map` es una función nativa de Python, por lo que siempre está disponible, funciona siempre igual y es más rápida que su versión equivalente con ciclos `for`.

Además `map` puede utilizarse en situaciones más complicadas. Por ejemplo, si la función recibida como argumento por `map` requiere a su vez más de un argumento, se pueden agregar todos los argumentos necesarios y `map` se los hace llegar a la función.

Por ejemplo, considere la función `pow` que requiere dos argumentos:

[ ]:
```
pow(2,3)

list(map(pow, [1,2,3,4], [5,6,7,8]))
```

[ ]:
```
[1, 64, 2187, 65536]
```

Para una función que requiere n argumentos, `map` espera n secuencias para esa función.

2.6 Herramientas de programación funcional: `filter` y `reduce`

La función `map` es la representación más simple de las funciones nativas de Python para la *programación funcional*, que son herramientas que aplican funciones a secuencias y otros iterables.

Estas herramientas filtran elementos de acuerdo a una función de prueba (filtro) y aplican funciones a pares de elementos y resultados de ejecución (reduce). Debido a que devuelven iterables, range y filter requieren llamadas de lista para mostrar todos sus resultados.

2.6.1 Filter

Por ejemplo, el siguiente filtro selecciona elementos en una secuencia dada por `range(-5, 5)` que son mayores que cero:

[ ]:
```
list(filter(lambda x:x>0, range(-5, 5)))
```

[ ]:
```
[1, 2, 3, 4]
```

Los elementos de la secuencia o iterable que cumplen la condición dada por la función `lambda` son añadidos a la lista resultante. Esta función, al igual que con `map`, puede ser construida con ciclos `for`, pero es más rápida y nativa.

[ ]:
```
res = []
for x in range(-5, 5):
    if x>0:
        res.append(x)
```

```
res
```

[ ]: [1, 2, 3, 4]

2.6.2 Reduce

`reduce` es una función nativa contenida dentro del modulo `functools` y acepta un iterador para proceder pero `reduce` no es un iterador ya que devuelve un sólo resultado.

Por ejemplo, `reduce` llama la suma y multiplicación para los elementos de la lista:

[ ]:
```
from functools import reduce

print(reduce(lambda x,y : x+y, [1,2,3,4]))
print(reduce(lambda x,y : x*y, [1,2,3,4]))
```

```
10
24
```

En cada paso, `reduce` pasa el valor actual de la suma o multiplicación junto con el siguiente valor de la lista a la función `lambda`. Por defecto, el primer valor de la lista inicializa el acumulador para el resultado final.

El siguiente código es una versión equivalente empleando ciclo `for`.

[ ]:
```
L = [1,2,3,4]
res = L[0]
for x in L[1:]:
    res += x
res
```

[ ]: 10

Para fines ilustrativos y entender el funcionamiento de `reduce` a fondo, éste es un código que replica el funcionamiento de `reduce`.

[ ]:
```
def myReduce(function, sequence):
    result = sequence[0]
    for next in sequence[1:]:
        result = function(result, next)
    return result

print(myReduce(lambda x,y : x+y, [1,2,3,4]))
print(myReduce(lambda x,y : x*y, [1,2,3,4]))
```

```
10
24
```

En el código se puede observar que `result` fue inicializado en el primer elemento de la lista. Sin embargo `reduce` acepta un tercer argumento que se utiliza como valor inicial, o incluso final si la lista estuviese vacía.

[ ]:
```
print(reduce(lambda x,y : x+y, [1,2,3,4], 10))
print(reduce(lambda x,y : x*y, [1,2,3,4], 5))
```

```
20
120
```

El valor inicial no debe ser necesariamente un número, puede ser cualquier tipo de dato hasta incluso un objeto. Debe considerarse el tipo que sea dicho valor inicial para las operaciones que `reduce` llevará a cabo con dicho valor inicial.

2.7 Referencias

- Sundnes J., Introduction to Scientific Programming with Python, Springer Open, 2020.
- Lutz M., Learning Python, O'Reilly, 2009.