Argenitna Programa 4.0

Universidad Nacional de San Luis

Desarrollador Python

Lenguaje de Programación Python: Soporte Funcional

Autor:

Dr. Mario Marcelo Berón

UNIDAD 7

LENGUAJE DE PROGRAMACIÓN PYTHON: SOPORTE FUNCIONAL

7.1. Introducción

¿Qué es la Programación Funcional? Esta es una pregunta difícil de responder pero se podría decir que la *Programación Funcional* es lo que el programador realiza cuando programa en lenguajes como *Lisp*, *Scheme*, *Clojure*, *Haskell*, *ML*, *OCAML*, *Erlang*, etc. Esta es una posible respuesta que no aclara demasiado el concepto. Desafortunadamente, es muy difícil dar una respuesta esclarecedora respecto a lo que es la Programación Funcional. Esto es verdad aún para los programadores que usan el paradigma de programación funcional. También se puede usar como intento de respuesta la constrastación con lenguajes imperativos como C, Pascal, TCL, etc) o con lenguajes orientados a objetos aunque en algunas casos un lenguaje puede incluir ambos tipos de paradigma de programación.

Se puede caracterizar a la Programación Funcional como aquella forma de concebir la programación que tiene las siguientes características:

- Las funciones son *first class* es decir todo lo que se puede hacer con los

datos es posible realizar con las funciones (tal como pasar una función como parámetro a otra función).

- La recursión se usa como estructura de control primaria. En algunos lenguajes no existen los loops.
- Existe un foco sobre las listas. Las listas son frecuentemente usadas con recursión sobre sublistas como un sustituto de los loops.
- Los lenguajes de programación funcional *puros* no tienen efectos colaterales. Esta característica excluye la asignación, otros valores que permitan seguir el estado de las variables.
- La programación funcional desaconseja el uso de sentencias, en su lugar trabaja con el uso de expresiones (funciones más argumentos). En el caso puro un programa es una expresión (más algunas definiciones).
- La programación funcional se preocupa por el *¿Qué?* se computa antes que *¿Cómo?* se computa.
- La programación funcional utiliza funciones de orden superior, es decir funciones que operan con funciones que operan funciones.

Los defensores de la programación funcional sostienen que todas esas características permiten un desarrollo más veloz, más corto, menos propenso a errores. Los científicos informáticos que estudian programación funcional sostienen que con este paradigma es más sencillo demostrar propiedades de los sistemas. Un concepto crucial en programación funcional es el de *Función Pura* es decir una función que siempre retorna el mismo resultado para los mismos argumentos. Es decir, lo más parecido a una función.

Python definitivamente no es un *Lenguaje de Programación Funcional Puro* los efectos colaterales están dispersos en casi todos los programas. Python tampoco es un *Lenguaje de Programación Funcional* sino más bien un lenguaje con soporte multiparadigma que permite utilizar construcciones funcionales cuando el programador lo desea.

Python brinda facilidades para mezclar diferentes estilos de programación.

7.2. Flujo de Control

En los lenguajes imperativos y orientados a objetos un bloque de código consiste generalmente de loops (*for* o *while*), asignaciones de variables dentro de esos loops, modificación de estructura de datos tales como listas, conjuntos, diccionarios, sentencias de bifurcación (*if*, *if-elif-else*, *try-except-finally*). Lo anterior parece que facilita el razonamiento. Sin embargo, el problema surge con los *efectos colaterales* que aparecen con las variables y las estructuras de datos. Variables y estructuras de datos permiten modelar los conceptos a partir de contenedores físicos del mundo real. No obstante, también es difícil razonar precisamente respecto del estado de los datos en un punto específico del programa.

Una solución consiste en no focalizar sobre la construcción de colecciones de datos sino en *¿Qué?* consiste esa colección de datos.

7.3. Encapsulación

Una forma obvia de focalizar en el *¿Qué?* antes que en el *¿Cómo?* consiste en refactorizar el código, y colocar los datos en un lugar más aislado como por ejemplo una función o un método.

Considere el siguiente trozo de código escrito utilizando el paradigma de programación imperativo.

Programa Imperativo

```
#Configurar los datos para comenzar
colección=obtenerEstadoInicial()
estadoVar = None
for dato in conjuntoDeDato:
 if condición(estadoVar):
  estadoVar = caluclarAPartirDe(dato)
  nuevo = modificar(dato, estadoVar)
  colección.agregar(nuevo)
 else:
  nuevo =modificarDeManeraDiferente(dato)
  colección.agregar(nuevo)

# Trabajar con los datos
for cosa in colección:
 procesar(cosa)
```

Para eliminar el *¿Cómo?* se debe sacar la construcción de los datos del alcance actual y colocarlo en una función que se pueda pensar de forma independiente.

Encapsulación

```
# Construcción de los Datos
def hacerUnaColección(conjuntoDeDatos):
 colección = obtenerEstadoInicial()
 estadoVar = None
 for dato in conjuntoDeDato:
  if condición(estadoVar):
   estadoVar = caluclarAPartirDe(dato, estadoVar)
   nuevo = modificar(dato, estadoVar)
   colección.agregar(nuevo)
  else:
   nuevo = modificarDeManeraDiferente(dato)
   colección.agregar(nuevo)
  return colección

# Trabajar con los datos
for cosa in hacerUnaColección(conjuntoDeDatos):
 procesar(cosa)
```

No se ha cambiado el programa, ni la cantidad de líneas de código sino más bien se a centrado en *¿Qué crea hacerUnaColección()?*

7.4. Comprensiones

Usar comprensiones es frecuentemente una forma de hacer el código más compacto y desplazar el foco desde el *¿Cómo?* hacia el *¿Qué?*. Una comprensión es una expresión que usa palabras claves como los loops y los bloques condicionales pero que invierte su orden para centrarse sobre los datos antes que en el procedimiento. Simplemente cambiando la forma de expresión puede tener un alto impacto respecto de como se razona el código y el esfuerzo requerido para entenderlo. El operador ternario también realiza una reestruc-

turación similar del foco a través del uso de las mismas palabras claves en un orden diferente.

Código Imperativo

```
colección = list()
for dato in conjuntoDeDato:
 if condición(dato):
  colección.append(dato)
 else:
  nuevo = modificar(dato)
  colección.append(nuevo)
```

Una forma más compacta del código anterior se muestra a continuación:

Comprensión

```
colección = [d if condición(d) else \
             modificar(d) for d in conjuntoDeDatos]
```

La última solución, además de tener menos líneas de código, permite pensar en *¿Qué es una colección?* en lugar de *¿Cuál es el estado de la colección en un punto determinado de la iteración?*

7.5. Generadores

Las comprensiones *Generadoras* tienen la misma sintaxis que las *Comprensiones de Listas* aunque no hay corchetes al rededor de ellas. No obstante, en algunos contextos es necesario colocar paréntesis.

Los generadores son *perezosos* es decir ellos son descripciones de *¿Cómo obtener el dato?* es decir no se genera hasta que explícitamente se pregunte por él a través de un llamado a *next()* o una iteración. Esta característica salva memoria y difiere la computación hasta que realmente se necesita.

Por ejemplo:

Generadores

```
líneas = (líneas for línea in
              leerLínea(archivo)
              if condiciónCompleja(línea))
```

Para usos típicos, el comportamiento es el mismo que consturir una lista pero el tiempo de ejecución es mejor.

El generador también se puede implementar de forma imperativa como se muestra a continuación.

Generador-Imperativo

```
def obtenerLíneas(archivo):
 línea = leerLínea(archivo)
 while True:
  try:
   if condiciónCompleja(línea):
    yield línea
   línea = leerLínea(archivo)
  except StopIteration:
   raise
líneas = obtenerLíneas(archivo)
```

La versión imperativa no es complicada pero muestra el *detrás de las escenas* el *¿Cómo?* de un loop *for* sobre un iterable.

Una versión orientada a objetos se muestra a continuación.

Generador-Orientado a Objeto

```
class ObtenerLíneas(object):
 def __init__(self, archivo):
  self.archivo = archivo
  self.línea = None

 def __iter__(self):
  return self

 def __next__(self):
  if self.línea is None:
   self.línea = leerLínea(archivo)
  while not condiciónCompleja(self.línea):
   self.línea = leerLínea(self.archivo)
  return self.línea
líneas = ObtenerLíneas(archivo)
```

Como se puede observar en las versiones imperativas y orientadas a objetos el centro está en el *¿Cómo?* en lugar de en el *¿Qué?*

7.6. Diccionarios y Conjuntos

Los diccionarios y conjuntos también se pueden crear con comprensiones.

Generador-Orientado a Objeto

```
>>> {i:chr(65+i) for i in range(6)}
{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'}
>>> {chr(65+i) for i in range(6)}
{'A', 'B', 'C', 'D', 'E', 'F'}
```


7.7. Recursión

Los programadores funcionales frecuentemente intentan expresar el flujo de control con recursión en vez de loops. De esta manera, se evita alterar el estado de las variables o de las estructuras de datos dentro de un algoritmo y más importante aún permite centrarse en el *¿Qué?* antes que en el *¿Cómo?* de una computación. Sin embargo, cuando se desea estudiar el estilo recursivo es necesario distinguir entre los casos donde la recursión es solo una iteración con otro nombre y aquellos casos donde un problema puede fácilmente ser particionado en problemas más pequeños, y cada uno de ellos puede ser abordado de manera similar. Existen dos razones por las que se haría esta distinción. Por una parte, usar la recursión como una forma de recorrer una secuencia de elementos, aún siendo posible, no es pitónica. Por otra parte, Python es lento con la recursión y tiene una profundidad del stack limitada. Si bien dicha profundidad se puede cambiar con *sys.setrecursionlimit()* (el valor por defecto es 1000) es probablemente un error. Python carece de una característica interna denominada *tail call elimination* que hace que la profundidad de la recursión sea computacionalmente eficiente. A continuación se muestra un ejemplo trivial donde la recursión se utiliza como una clase de iteración.

Recursión

```
def generarSuma(listaDeNúmeros, comienzo=0):
 if len(listaDeNúmeros) == 0:
  return []
 return [listaDeNúmeros[0]+comienzo]+\
          generarSuma(listaDeNúmeros[1:],\
          listaDeNúmeros[0]+comienzo)

print(generarSuma([10,20,30]))
print(generarSuma([1,2,3],20))
```


Poco se puede recomendar en este enfoque, una iteración sería más legible.

Versión Iterativa

```
def generarSuma2(listaDeNúmeros,comienzo=0):
 r=[]
 for i in range(len(listaDeNúmeros)):
  comienzo=listaDeNúmeros[0]+comienzo
  r=r+[comienzo]
 return r

print(generarSuma2([10,20,30]))
print(generarSuma2([1,2,3],20))
```

Además, es perfectamente razonable que se invoque esta función con una longitud mayor que 1000. No obstante, en otros casos, el estilo recursivo, aún sobre operaciones secuenciales, expresa algoritmos más intuitivamente y en una forma que es más fácil de razonar.

Recursión - Factorial - Versión Recuersiva e Impertiva

```
def factorialR(N):
 "Función Factorial Recursiva"
 assert isinstance(N, int) and N >= 1
 return 1 if N <= 1 else N * factorialR(N-1)

def factorialI(N):
 "Función Factorial Iterativa"
 assert isinstance(N, int) and N >= 1
 producto = 1
 while N >= 1:
  producto *= N
  N -= 1
 return producto
```

Si bien el algoritmo se puede expresar fácilmente con un producto de variables, la versión recursiva es más cercana al *¿Qué?* que al *¿Cómo?*. Los detalles del cambio de N en la versión iterativa es una tarea de mantenimiento y no de la naturaleza de la computación[1].

Comentario

Una de las versiones más rápidas de *factorial* en Python está en el estilo funcional y también expresa el *¿Qué?* del algoritmo.

```
from functools import reduce
from operator import mul
def factorialHOF(n):
 return reduce(mul, range(1, n+1), 1)
```

Donde la recursión es convincente, y algunas veces la única forma obvia

[1]La versión iterativa probablemente es más rápida y permite alcanzar fácilmente el límite de la recursión.

de expresar una solución, es cuando el problema permite aplicar una aproximación *divide y conquista*. Esto es cuando el problema permite aplicar una computación similar a dos mitades o a colecciones más grandes. En esos casos, la profundidad de la recursión es O($log_n N$) del tamaño de la colección lo cual es poco probable que sea demasiada profundidad.

Quick - Sort

```
def quicksort(lst):
 "Quicksort sobre una lista"
 if len(lst) == 0:
  return lst
 pivot = lst[0]
 pivots = [x for x in lst if x == pivot]
 small = quicksort([x for x in lst if x < pivot])
 large = quicksort([x for x in lst if x > pivot])
 return small + pivots + large
```

Algunos nombres son usados en el cuerpo de la función para mantener valores y nunca se cambian. La definición se puede escribir en una sola expresión pero sería poco legible.

Consejo

Es buena práctica analizar diferentes posibilidades para la expresión recursiva - especialmente para las versiones que evitan la necesidad de variables o colecciones de datos mutables - siempre que el problema sea particionable en subproblemas. No es buena idea en Python usar la recursión meramente hacer una iteración con otro nombre.

7.7.1. Ejemplos de uso de Recursión

En ciencias de la computación, un árbol binario es una estructura de datos en la cual cada nodo puede tener un hijo izquierdo y un hijo derecho (ver figura7.1). No pueden tener más de dos hijos (de ahí el nombre "binario").

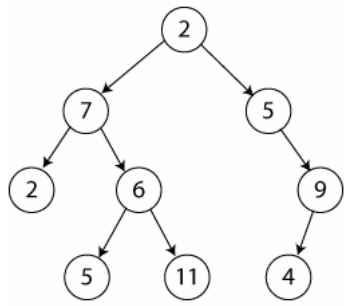

Figura 7.1: Árbol Binario

Si algún hijo tiene como referencia a null, es decir que no almacena ningún dato, entonces este es llamado un nodo hoja o externo. En el caso contrario el hijo es llamado un nodo interno. Usos comunes de los árboles binarios son los *árboles binarios de búsqueda* y los *heap binarios*.

En teoría de grafos, se usa la siguiente definición: *Un árbol binario es un grafo conexo, acíclico y no dirigido tal que el grado de cada vértice no es mayor a 2*. De esta forma solo existe un camino entre un par de nodos.

Un árbol binario con raíz es como un grafo que tiene uno de sus vértices, llamado raíz, de grado no mayor a 2 y grado de entrada 0 (es decir no le llegan arcos). Con la raíz seleccionada, cada vértice tendrá un único padre, y nunca más de dos hijos.

Recorridos en un Árbol Binario

Un recorrido en un árbol binario es una operación que consiste en visitar todos sus vértices o nodos, de tal manera que cada vértice se visite una sola vez. Se distinguen tres tipos de recorrido: *inorden*, *posorden* y *preorden*. En cada recorrido se tiene en cuenta la posición de la raíz (de ahí su nombre) y que siempre se debe ejecutar primero el hijo izquierdo y luego el derecho.

- Recorrido *inorden*. Este recorrido se realiza así: primero recorre el subárbol izquierdo, segundo visita la raíz y por último, va al subárbol derecho. En síntesis: hijo izquierdo — raíz — hijo derecho.

- Recorrido *preorden*. Este recorrido se realiza así: primero visita la raíz;

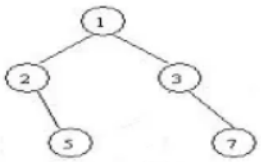

figura 12.5a

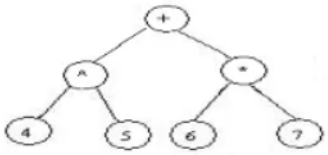

figura 12.5b

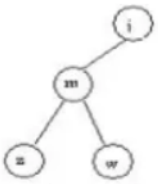

figura 12.5c

RECORRIDOS	Figura 12.5a	Figura 12.5b	Figura 12.5c
INORDEN	2-5-1-3-7	4^5+6*7	z m n i
PREORDEN	1-2-5-3-7	+^4 5*6 7	i m z n
POSORDEN	5-2-7-3-1	4 5 ^ 6 7*+	z n m i

Figura 7.2: Árbol Binario

segundo recorre el subárbol izquierdo y por último va a subárbol derecho. En síntesis: raíz — hijo izquierdo — hijo derecho.

- Recorrido *posorden*. Primero recorre el subárbol izquierdo; segundo, recorre el subárbol derecho y por último, visita la raíz. En síntesis: hijo izquierdo– hijo derecho — raíz.

La figura 7.2 presenta ejemplos de recorridos sobre árboles binarios.

Uno de los principales problemas cuando se desarrolla software consiste en la elección de la representación computacional para la estructura de datos que se está estudiando. Se dice que es un problema dado que la elección de la representación puede afectar la *complejidad temporal* del algoritmo, la *complejidad espacial* y también la *algoritmia*. En este contexto, no se pretende elaborar la mejor representación de un árbol sino más bien ilustrar como dicha estructura de datos puede ser representada y luego definir un algoritmo que implemente un recorrido sobre un árbol binario.

Un nodo del árbol puede ser representado por una lista compuesta por tres elementos:

- El valor del nodo, es decir la información que se almacena en el nodo.
- Una lista que represente el subárbol izquierdo.

- Una lista que represente el subárbol derecho.

A continuación se muestran diferentes árboles representados con los lineamientos descritos con anterioridad.

Representación de un Árbol Binario

Un árbol vacío se representa con una lista vacía: []
Un árbol con un único nodo se representa como sigue:
[v,[],[]]
donde v es la información que se almacena en un nodo. De esta manera si se almacenan números entonces:
[2,[],[]]
es la representación de un árbol que tiene un único nodo.
El árbol de la figura 7.1 se representa como sigue:
[2,[7,[2,[],[]],[]],[6,[5,[],[]],[11,[][][]]],[5,[],[9,[4,[],[]],[]]]]

Teniendo en cuenta la representación explicada, a continuación se muestra la implementación de un barrido *inorden* en un árbol binario.

Barrido Simétrico en un Árbol Binario

```
def simétrico(árbol):
 if árbol!=[]:
  if árbol[1]!=[]:
   simétrico(árbol[1])
  print(árbol[0])
  if árbol[2]!=[]:
   simétrico(árbol[2])
 else:
  print([])

simétrico([])
simétrico([2,[3,[5,[],[]],[]],[4,[],[6,[],[]]]])
```

7.8. Llamables

El énfasis en la programación funcional está en el llamado a funciones. Python tiene diferentes formas de crear funciones, o algo parecido a una función, es decir algo que pueda ser llamado. Ellas son:

- Funciones creadas con *def* y un nombre en tiempo de definición.
- Funciones anónimas creadas con *lambdas*.
- Instancias de clases que definan un método *__call()__*.
- Clausuras retornadas por fábricas de funciones.
- Métodos estáticos de instancias, o métodos creados con el decorador *@staticmethod* o a través de la clase *__dict__*.
- Funciones generadores.

La lista no es exahustiva pero menciona varias formas de como se puede crear un llamable. Obviamente, un método plano de una instancia de clase también es un llamable, pero por lo general se usa donde el énfasis está en el acceso y modificación del estado mutable. Python es un lenguaje multiparadigma, cuando se define una clase es generalemente para generar instancias que son contenedores de datos que cambian cuando se invoca a un método. Este estilo es de alguna manera opuesto a la aproximación funcional, el cual enfatiza la inmutabilidad y las funciones puras.

Cualquier método que accede a una instancia (en algún grado) para determinar que resultado retorna no es una función pura. Por supuesto, todos los otros tipos de llamables que se discutirán también permiten depender del estado de varias formas. La ventaja de una *función pura* y libre de efectos colaterales es que es más fácil de depurar y probar. Los llamables que libremente mezclan estado con sus resultados no se pueden examinar independiente de su contexto de ejecución para ver como se comportan.

Obviamente, cualquier programa que haga alguna cosa debe tener alguna clase de salida de forma tal de hacer algo útil, por lo tanto los efectos colate-

rales no se pueden evitar, solo se pueden aislar cuando se piensa en términos funcionales.

7.9. Funciones nombradas y lambdas

La forma más obvia de crear llamables en Python son las funciones nombradas y lambdas. La única diferencia entre ellas es que tienen un atributo *.__qualname__* dado que pueden estar ligadas a uno o más nombres. En muchos casos las expresiones *lambda* son usadas en Python solo como *callbacks* [2] y otros usos donde una simple acción está *en línea* dentro del llamado a una función. No obstante, el flujo de control en general se puede incorporar en expresiones lambdas.

[2]En programación, una retrollamada o devolución de llamada, también conocida como posllamada, es una función ejecutable A que se usa como argmento de otra función B. De esta forma, al llamar a B, se ejecutará A

lambda

```
>>> def hola_1(nombre):
.....     print("Hola:", nombre)
.....
>>> hola_2 = lambda nombre: print("Hola:", nombre)
>>> hola_1('David')
Hola David
>>> hola_2('David')
Hola David
>>> hola_1.__qualname__
'hola_1'
>>> hola_2.__qualname__
'<lambda>'
>>> hola_3 = hola_2 # ligar func. a otros nombres
>>> hola_3.__qualname__
'<lambda>'
>>> hola_3.__qualname__ = 'hola_3'
>>> hola_3.__qualname__
'hola_3'
```

Una de las razones por la que las funciones son útiles es que ellas aislan el estado lexicográfico. Esta es una forma limitada de no mutabilidad nada de lo que ud. haga dentro de una función ligará variables de estado fuera de la función. Esta garantía es muy limitada dado que *global* y las sentencias no locales permiten explícitamente que el estado de una función se filtre. Asimismo, muchos tipos de datos son mutables por lo tanto cuando ellos se pasan a una función cambian su contenido. También, la entrada/salida cambia el *estado del mundo* y por eso altera el resultado de las funciones. A pesar de todas las advertencias y límites mencionados con anterioridad, un programador que quiere focalizar en el estilo de programación funcional puede intencionalmente decidir escribir muchas funciones como puras para posibilitar el razonamiento matemático sobre ellas.

7.10. Clausuras y Llamables

Existe un dicho en Ciencias de la Computación de que una clase es *Datos con operaciones* mientras que una clausura es *Operaciones con datos*. En cierto sentido ellos hacen cosas parecidas: colocan lógica y dato en el mismo objeto. Pero hay una diferencia filosófica en los enfoques. Con las clases se enfatiza el estado vinculable y mutable y las clausuras enfatizan la inmutabilidad y las funciones puras. Ninguno de los lados de esta división es absoluto pero diferentes actitudes motivan el uso de cada uno.

Clausuras y Llamables

```
#Una clase que crea instancias de sumadores
#invocables

class Sumador(object):
 def __init__(self, n):
  self.n = n
#El método __call__ permite que los programdores
#escriban clases donde las insntancias se comporten
#como funciones y puedan ser llamadas como una
#función
  def __call__(self, m):
   return self.n + m

sumar5_i = Sumador(5)
# "instancia" - "imperativo"
```

En el ejemplo se construyó un llamable que suma cinco a un argumento que se le pasa como parámetro. El siguiente código muestra el mismo ejemplo pero implementado como una clausura.

Clausuras y Llamables

```
def hacerSumador(n):
 def sumador(m):
  return m + n
 return sumador

sumar5_f = hacerSumador(5)
# "funcional"
```

El comportamiento de *sumar5_f* creado por *hacerSumador()* no está determinado hasta el tiempo de ejecución. Pero una vez que el objeto exista, la clausura se comporta en una forma funcional pura mientras que una instancia de la clase queda dependiente del estado.

Existe poco entendimiento respecto de cómo Python liga las variables en una clausura. Python lo hace por nombre antes que por valor lo cual puede causar confusión, pero tiene una fácil solución. Por ejemplo, si se desea crear diferentes clausuras encapsulando varios datos de la forma que se muestra a continuación no se logran los resultados esperados.

Clausuras y Llamables

```
#Seguramente no es el comportamiento que se espera.
>>> sumadores = []
>>> for n in range(5):
...  sumadores.append(lambda m: m+n)
>>> [sumador(10) for sumador in sumadores]
[14, 14, 14, 14, 14]
>>> n = 10
>>> [sumador(10) for sumador in sumadores]
[20, 20, 20, 20, 20]
```

Afortunadamente, un pequeño cambio permite lograr el objetivo.

Clausuras y Llamables

```
>>> sumadores = []
>>> for n in range(5):
....     sumadores.append(lambda m, n=n: m+n)
....
>>> [sumadores(10) for sumador in sumadores]
[10, 11, 12, 13, 14]
>>> n = 10
>>> [sumador(10) for sumador in sumadores]
[10, 11, 12, 13, 14]
>>> sumar_4 = sumadores[4]
>>> sumar_4(10, 100)
110
```

Es importante notar que se usa el truco de usar argumentos de palabra clave para cambiar el valor lo cual reduce la confusión. El valor predominante para la variable nombrada debe pasarse explícitamente en la llamada misma, ligarse en algún lugar en el flujo del programa.

7.11. Métodos de Clase

Todos los métodos de clases son llamables. Para muchos llamar a un método de una instancia va en contra del estilo de programación funcional. Usualmente se usan métodos porque se desea referenciar a datos mutables que están empaquetados en los atributos de una instancia, y por eso en cada llamado de un método produce un resultado diferente que varía independientemente de los argumentos pasados.

7.11.1. Accesores y Operadores

Aún los accesores creados con el decorador *@property* o de otro modo, técnicamente son llamables, aunque los accesores son llamables con un uso

limitado (desde una perspectiva de la programación funcional). Ellos no tienen artumentos como los *getters* y no retornan un valor como los *setters*.

Accesores

```
class Auto(object):
 def __init__(self):
  self._velocidad = 100

 @property
 def velocidad(self):
  print("La velocidad es:", self._velocidad)
  return self._velocidad

 @velocidad.setter
 def velocidad(self, valor):
  print("Inicializar a:", valor)
  self._velocidad = valor
# >> auto = Auto()
# >>> auto.velocidad = 80 #Sintaxis rara para pasar
                          #un argumento
# Inicializar a: 80
# >>> x = car.velocidad
# La velocidad es: 80
```

En el accesor, se optó por la sintaxis de Python de la asignación para pasar un argumento. La sintaxis es sencilla aunque:

Operadores

```
>>> class IntCharlatán(int):
      def __lshift__(self, otro):
            print("Desplazar", self, "por", otro)
            return int.__lshift__(self, otro)
....
>>> t = IntCharlatán(8)
>>> t << 3
Desplazar 8 por 3
64
```

Todo operador en Python es básicamente un método. Si bien se produce un *Lenguaje Específico del Dominio* más legible, el cual define significados especiales para los operadores no incorpora mejoras para las capacidades de las llamadas a funciones.

7.12. Métodos Estáticos de Instancias

Uno de los usos de las clases y sus métodos que está más estrechamente alineado con el estilo de programación funcional es usarlos simplemente como espacio de nombres para mantener una variedad de funciones relacionadas.

Métodos Estáticos

```
import math
class TriánguloRectángulo(object):
 """Clase usada solamente como espacio de nombres
 para funciones relacionadas"""
 @staticmethod
 def hipotenusa(a, b):
  return math.sqrt(a**2 + b**2)

 @staticmethod
  def seno(a, b):
   return a / TriánguloRectángulo.hipotenusa(a, b)

 @staticmethod
 def coseno(a, b):
  return b / TriánguloRectángulo.hipotenusa(a, b)
```

Mantener esta funcionalidad en una clase evita contaminar el espacio de nombres global (o un módulo, etc.) y permite nombrar a una clase o una instancia de la misma para llamar a las funciones puras.

Métodos Estáticos

```
>>> TriánguloRectángulo.hipotenusa(3,4)
5.0
>>> rt = TriánguloRectángulo()
>>> rt.seno(3,4)
0.6
>>> rt.coseno(3,4)
0.8
```

Por lejos, la forma más directa de definir métodos estáticos es usando

decoradores. Sin embargo, en Python 3.x, se pueden utilizar funciones que no han sido decoradas es decir el concepto de funciones no ligadas no es más necesario en las versiones modernas del Python.

Funciones sin Decorar

```
>>> import functools, operator
>>> class Math(object):
...     def product(*nums):
...         return functools.reduce(operator.mul, nums)
...
...     def power_chain(*nums):
...         return functools.reduce(operator.pow, nums)
...
>>> Math.product(3,4,5)
60
>>> Math.power_chain(3,4,5)
3486784401
```

Sin *@staticmethod*, los llamados no funcionarán sobre las instancias dado que se espera que se pase el *self*.

Funciones sin Decorar

```
>>> m = Math()
>>> m.product(3,4,5)
TypeError
Traceback (most recent call last)
<ipython-input-5-e1de62cf88af> in <module>()
----> 1 m.product(3,4,5)
<ipython-input-2-535194f57a64> in product(*nums)
2 class Math(object):
3 def product(*nums):
----> 4 return functools.reduce(operator.mul, nums)
5 def power_chain(*nums):
6 return functools.reduce(operator.pow, nums)
TypeError: unsupported operand type(s) for *:\
 'Math' and 'int'
```

Si el espacio de nombres es un bolsa de funciones puras, no hay razón para no llamarlas a través de una clase antes que una instancia. Pero si ud. usa una mezcla con algunas funciones puras y otras no debería usar el decorador @staticmethod.

7.13. Funciones Generadoras

Una clase especial de funciones en Python son aquellas que contienen una sentencia *yield* las cuales se transforman en generadores. Lo que se retorna a través del llamado a tal función no es un valor común, sino un iterador que produce una secuencia de valores tal como cuando se llama a *next()* o en un loop. Mientras que como con cualquier objeto Python, hay muchas formas de introducir estado en un generador, en principio un generador puede ser puro en el sentido de una función pura. Esto es una función pura que produce una (potencialmente infinita) secuencia de valores antes que un único valor

pero se basa solo en los argumentos que le han sido pasado. No obstante, las funciones generadoras tienen un gran parte de estado interno, esto está en la signatura y en el valor de retorno donde actúan como una caja negra libre de efectos colaterales.

Funciones sin Decorar

```
>>> def obtenerPrimos():
...     "Criba de Eratóstenes perezosa simple"
...     candidato = 2
...     encontrado = []
...     while True:
...         if all(candidato % primo != 0 for primo\\
                   in encontrado):
...             yield candidato
...             encontrado.append(candidato)
...             candidato += 1
>>> primos = obtenerPrimos()
>>> next(primos), next(primos), next(primos)
(2, 3, 5)
>>> for _, primo in zip(range(10), primos):
...     print(primo, end=" ")
...
7 11 13 17 19 23 29 31 37 41
```

Toda vez que se crea un nuevo objeto con *obtenerPrimos()* el iterador produce la misma secuencia perezosa infinita -otro ejemplo debe pasar en la inicialización de valores que afectan el resultado- pero el objeto en si mismo está con estado cuando se consume incrementalmente.

7.14. Despacho Múltiple

Una enfoque muy interesante para la programación de pasos múltiples de ejecución es la técnica llamada *Despacho Múltiple* o algunas veces *Multi-Métodos*. La idea es declarar múltiples signaturas de una función y llamar a la computación que concuerde con los tipos o propiedades de los argumentos del llamado. Esta técnica frecuentemente permite evitar o reducir el uso de bifurcaciones condicionales explícitamente. En lugar de ello se sustituye por el uso de uno o más descripciones de patrones intuitivos de argumentos. Para explicar como el despacho múltiple hace más legible y menos propenso a errores al código, se implementará el juego de piedra/papel/tijera en tres estilos. Se crearán tres clases para jugar el juego para todas las versiones:

Piedra-Papel-Tijera

```
class Cosa(object): pass
class Piedra(Thing): pass
class Papel(Thing): pass
class Tijera(Thing): pass
```

7.15. Muchas Bifurcaciones

Primero se presenta una versión imperativa en donde se puede observar que hay muchas repeticiones, anidamientos, bloques condicionales y en consecuencia es fácil equivocarse.

Piedra-Papel-Tijera

```
def batir(x, y):
 if isinstance(x, Piedra):
  if isinstance(y, Piedra):
   return None # No hay ganador
  elif isinstance(y, Papel):
   return y
  elif isinstance(y, Tijera):
   return x
  else:
   raise TypeError("Segunda Cosa Desconocida")
elif isinstance(x, Papel):
   if isinstance(y, Piedra):
    return x
   elif isinstance(y, Papel):
      return None # No hay ganador
   elif isinstance(y, Tijera):
      return y
   else: raise TypeError("Segunda Cosa Desconocida")
elif isinstance(x, Tijera):
   if isinstance(y, Piedra):
    return y
   elif isinstance(y, Papel):
    return x
   elif isinstance(y, Tijera):
    return None # No hay ganador
   else:
    raise TypeError("Segunda Cosa Desconocida")
else:
 raise TypeError("Primera Cosa Desconocida")
piedra, papel, tijera = Piedra(), Papel(), Tijera()
```


Piedra-Papel-Tijera

```
# >>> batir(papel, piedra)
# <__main__.Papel at 0x103b96b00>
# >>> batir(papel, 3)
# TypeError: Segunda cosa desconocida
```

7.15.1. Delegar a un Objeto

Como un segundo ensayo se puede intentar eliminar alguna repetición con el *Tipado del Pato* de Python. Posiblemente se puedan tener cosas diferentes que compartan un método en común el cual se llama cuando se necesite.

Piedra-Papel-Tijera - Duck Typing

```
class PatoPiedra(Piedra):
 def batir(self, otro):
  if isinstance(otro, Piedra):
   return None # No hay ganador
  elif isinstance(otro, Papel):
   return otro
  elif isinstance(otro, Tijera):
   return self
  else:
   raise TypeError("Segunda Cosa Desconocida")
```


Piedra-Papel-Tijera - Duck Typing

```
class PatoPapel(Papel):
 def batir(self, otro):
  if isinstance(otro, Piedra):
   return self
  elif isinstance(otro, Papel):
   return None # No hay ganador
  elif isinstance(otro, Tijera):
   return otro
  else:
   raise TypeError("Segunda Cosa Desconocida")

class PatoTijera(Tijera):
 def batir(self, otro):
  if isinstance(otro, Piedra):
   return otro
  elif isinstance(otro, Papel):
   return self
  elif isinstance(otro, Tijera):
   return None # No hay ganador
  else:
   raise TypeError("Segunda cosa desconocida")

def batir2(x, y):
 if hasattr(x, 'batir'):
  return x.batir(y)
 else:
  raise TypeError("Primer cosa desconocida")
piedra, papel, tijera = PatoPiedra(), PatoPapel(),\
                        PatoTijera()
```


Piedra-Papel-Tijera - Duck Typing

```
# >>> batir2(piedra, papel)
# <__main__.PatoPapel at 0x103b894a8>
# >>> batir2(3, piedra)
# TypeError: Primer cosa desconocida
```

Como es posible observar no se ha reducido una cantidad de código, no obstante esta versión reduce la complejidad individual de cada llamable, y reduce el nivel de anidamiento de los condicionales en uno. Mucha lógica se coloca en clases separadas antes que en una bifurcación profunda. En POO se puede delegar el despacho a los objetos.

7.15.2. Concordancia de Patrones

Como un ensayo final, se expresará toda la lógica usando despacho múltiple el cual es más legible aunque hay aún un número de casos por definir.

Piedra-Papel-Tijera - Despacho Múltiple

```
from multipledispatch import dispatch
@dispatch(Piedra, Piedra)
 def batir3(x, y): return None

@dispatch(Piedra, Papel)
def batir3(x, y): return y

@dispatch(Piedra, Tijera)
def batir3(x, y): return x

@dispatch(Papel, Piedra)
def batir3(x, y): return x
```


Piedra-Papel-Tijera - Despacho Múltiple

```
@dispatch(Papel, Papel)
def batir3(x, y): return None

@dispatch(Papel, Tijera)
def batir3(x, y): return x

@dispatch(Tijera, Piedra)
def batir3(x, y): return y

@dispatch(Tijera, Papel)
def batir3(x, y): return x

@dispatch(Tijera, Tijera)
def batir3(x, y): return None

@dispatch(object, object)
def batir3(x, y):
 if not isinstance(x, (Piedra, Papel, Tijera)):
  raise TypeError("Primer Cosa Desconocida")
 else:
  raise TypeError("Segunda Cosa Desconocida")

# >>> batir3(piedra, papel)
# <__main__.DuckPapel at 0x103b894a8>
# >>> batir3(piedra, 3)
# TypeError: Segunda Cosa Desconocida
```


7.15.3. Despacho Basado en Predicados

Una forma exótica de expresar condicionales como decisiones de despacho es incluir los predicados directamente dentro de las signaturas de las funciones (o quizás dentro de los decoradores como con el despacho múltiple). No se tiene información precisa respecto de librerías que implementen este enfoque pero se deja una visión de cómo se podría usar.

Despacho Basado en Predicados

```
from predicative_dispatch import predicate

@predicate(lambda x: x < 0, lambda y: True)
def signo(x, y):
 print("x es negativo; y es", y)

@predicate(lambda x: x == 0, lambda y: True)
def signo(x, y):
 print("x es cero; y es", y)

@predicate(lambda x: x > 0, lambda y: True)
def signo(x, y):
 print("x es positivo; y es", y)
```

Como se puede observar en esta versión hipotética de despacho a través de predicados la bifurcación condicional está en la signatura de la función en sí misma, lo cual produce un código más pequeño más fácil de entender y depurar.

7.16. Evaluación Perezosa

Una característica poderosa de Python es su *Protocolo Iterador*. Esta capacidad es la única pobremente conectada a la programación funcional por

sí, ya que Python no ofrece *Estructuras de Datos Perezosas* como las que proporciona el lenguaje Haskell. Sin embargo, el uso del protocolo iterador -y de muchas funciones primitivas o iterables de la biblioteca estándar- llevan a cabo gran parte del mismo efecto.

En un lenguaje como Haskell, cual es inherentemente evaluado perezosamente se puede definir una lista de números primos como se muestra a continuación.

Despacho Basado en Predicados

```
-- Define una lista de todos los números primos
primos = criba [2 ..]
where criba (p:xs) = p : criba [x | x <- xs,
                                    (x `rem` p)/=0]
```

Si bien esta definición es similar a las comprensiones de Python, la principal diferencia radica en que esta secuencia es infinita y no solo un objeto capaz de producir una secuencia. En particular, se puede indexar un elemento arbitrario de la lista infinita de primos y los valores intermedios serán producidos internamente a medida que se necesiten basados sobre la construcción sintáctica de la lista. La idea descrita con anterioridad se puede replicar en Python como se muestra a continuación.

Evaluación Perezosa

```
from collections.abc import Sequence
class SecuenciaDeExpación(Sequence):
 def __init__(self, it):
  self.it = it
  self._cache = []

 def __getitem__(self, índice):
  while len(self._cache) <= índice:
   self._cache.append(next(self.it))
  return self._cache[índice]

 def __len__(self):
  return len(self._cache)
```

Este nuevo contenedor puede ser perezoso y también indexable.

Evaluación Perezosa

```
>>> primos = SecuenciaDeExpación(obtenerPrimos())
>>> for _, p in zip(range(10), primos):
... print(p, end="␣")
....
2 3 5 7 11 13 17 19 23 29
>>> primos[10]
31
```


Evaluación Perezosa

```
>>> primos[5]
13
>>> len(primos)
11
>>> primos[100]
547
>>> len(primos)
101
```

7.16.1. El Protocolo Iterador

La forma más fácil de crear un iterador - es decir, una secuencia perezosa - consiste en definir una función generador. Simplemente se usa la sentencia *yield* dentro del cuerpo de la función para definir los lugares donde los valores serán producidos. O, técnicamente, la forma más fácil es usar uno de los objetos iterables ya producidos por las funciones primitivas o la librería estándar antes que programar una personalizada. Las funciones generadoras tienen una sintaxis apropiada para definir funciones que retornan un iterador.

Muchos objetos tienen un método llamado *.__iter__()* el cual retorna un iterador cuando se invoca, generalmente a través de la función primitiva *iter()*, o aún más simple iterando sobre el objeto.

Un interador es un objeto retornado por un llamado a *iter(algo)* el cual tiene un método llamado *.__iter__()* que simplemente retorna el objeto en sí mismo y otro método llamado *.__next__()*. La razón de que un iterable tenga el método *.__iter__()* es hacer *iter()* idempotente. Esto es la identidad siempre se mantiene o se dispara una excepción *TypeError('object is not iterable')*.

Protocolo Iterador

```
iter_seq = iter(sequence)
iter(iter_seq) == iter_seq
```

Dado que los comentarios previos son abstractos se presentan algunos ejemplos concretos.

Protocolo Iterador

```
#iterar a través de las líneas de un archivo
>>> perezoso = open('Archivo.txt')
>>> '__iter__' in dir(perezoso) and \
    '__next__' in dir(perezoso)
True
>>> sumar_1 = map(lambda x: x+1, range(10))
>>> sumar_1
# iterate over deferred computations
<map at 0x103b002b0>
>>> '__iter__' in dir(sumar_1) and \
    '__next__' in dir(sumar_1)
True
>>> def hasta_10():
...   for i in range(10):
...     yield i
...
>>> '__iter__' in dir(hasta_10)
False
>>> '__iter__' in dir(hasta_10()) and \
    '__next__' in dir(hasta_10())
True
```


Protocolo Iterador

```
>>> l = [1,2,3]
>>> '__iter__' in dir(l)
True
>>> '__next__' in dir(l)
False
>>> li = iter(l)
#Iterar sobre colecciones concretas
>>> li
<list_iterator at 0x103b11278>
>>> li == iter(li)
True
```

En el estilo de programación funcional - o para legibilidad - escribir iteradores personalizados como funciones generadores es más natural. Sin embargo, es posible crear clases personalizadas que obedezcan al protocolo. Frecuentemente, tienen otros comportamientos los cuales recaen sobre estados y efectos colaterales para que sean significativos.

Protocolo Iterador

```
from collections.abc import Iterable
class Fibonacci(Iterable):
 def __init__(self):
  self.a, self.b = 0, 1
  self.total = 0

 def __iter__(self):
  return self
```


Protocolo Iterador

```
class Fibonacci(Iterable):
....
 def __next__(self):
  self.a, self.b = self.b, self.a + self.b
  self.total += self.a
  return self.a

 def ejecutar_suma(self):
  return self.total

# >>> fib = Fibonacci()
# >>> fib.ejecutar_suma()
# 0
# >>> for _, i in zip(range(10), fib):
# ...print(i, end=" ")
# ...
# 1 1 2 3 5 8 13 21 34 55
# >>> fib.ejecutar_suma()
# 143
# >>> next(fib)
# 89
```

Este ejemplo es trivial pero muestra una clase que implementa un protocolo iterador y también provee un método adicional que retorna un estado respecto de la instancia.

7.16.2. Módulo: Itertools

El módulo itertools consiste de un conjunto poderoso y cuidadosamente diseñado de funciones para realizar el álgebra de iteradores. Es decir permiten combinar iteradores de forma sofisticada sin tener que concretamente instan-

ciar nada más que lo requerido. Además de las funciones básicas en el módulo, la documentación provee un número de cortas, pero fácil de equivocarse, recetas para incorporar funciones adicionales que utilizan una combinación de dos o tres funciones básicas. El módulo *more_itertools* provee funciones adicionales que están diseñadas para evitar dificultades y casos límites.

El objetivo básico de usar los bloques construidos dentro de *itertools* es evitar realizar computaciones antes de que ellas sean requeridas, evitar requerimientos de memoria de colecciones instanciadas grandes, evitar la entrada/-salida potencialmente lenta hasta que sea estrictamente necesaria, y cuando se combinen con funciones o recetas en *itertools* retengan la propiedad.

A continuación se presentan un pequeño ejemplo de combinación de unas pocas cosas. Antes que tener la clase *Fibonacci* con estado que mantenga la suma, se crea un iterador perezoso que genere el número y la suma.

Evaluación Perezosa

```
>>> def fibonacci():
...     a, b = 1, 1
...     while True:
...         yield a
...         a, b = b, a+b
...
>>> from itertools import tee, accumulate
>>> s, t = tee(fibonacci())
>>> pares = zip(t, accumulate(s))
>>> for _, (fib, total) in zip(range(7), pares):
...     print(fib, total)
1 1
1 2
2 4
3 7
...
```

La combinación correcta y óptima de las funciones en itertools requiere de un cuidadoso pensamiento pero una vez combinadas dan una gran potencia que da con largos y potencialmente infinitos iteradores que no podrían ser hechos con colecciones concretas.

La documentación del módulo itertools contiene detalles de sus funciones combinatoriales tan bien como un número de recetas cortas para combinarlas.

7.17. Funciones de Orden Superior

En general una *Función de Orden Superior* (HOF: High-Order Function) es una función que toma una o más funciones como argumento y produce una función como resultado. Muchas abstracciones interesantes están disponibles. Dichas abstracciones permiten encadenar y combinar funciones de orden superior de una manera análoga a como se combinan las funciones en itertools para producir nuevos iterables.

Una pocas funciones de orden superior están contenidas en el módulo functools y otras son primitivas. Es común pensar en *map*, *filter()*, y *functools.reduce()* como bloques básicos de construcción de funciones de orden superior. Por esta razón, muchos lenguajes de programación usan esas funciones como sus primitivas (ocasionalmente bajo otros nombres). Las funciones primitivas *map()* y *filter()* son equivalentes a comprensiones y muchos programadores Python encuentran a la comprensión más legible. A continuación se muestran algunos ejemplos.

HOFs

```
# Clásico Estilo de Programación Funcional
transformado = map(tranformación, iterador)
# Comprensión
transformado = (transformación(x) for x in iterador)
# Clásico Estilo de Programación Funcional
filtrado = filter(predicado, iterador)
# Comprehension
filtrado = (x for x in iterador if predicado(x))
```

La función *functools.reduce()* es muy general, muy poderosa, y muy sutil para usar todo su poder. La función toma los sucesivos ítems de un iterable, y los combina de alguna forma. El uso más común de *reduce()* está probablemente dado por *sum* el cual es más compacto:

HOFs-sum

```
from functools import reduce
total = reduce(operator.add, it, 0)
# total = sum(it)
```

Puede o no ser obvio que *map()* y *filter()* son casos especiales de *reduce()*. Observe los ejemplos a continuación:

HOFs-sum

```
>>> add5 = lambda n: n+5
>>> reduce(lambda l, x: l+[add5(x)], range(10), [])
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> # Más simple: map(add5, range(10))
>>> isOdd = lambda n: n%2
>>> reduce(lambda l, x: l+[x] if isOdd(x) else l,
                        range(10),[])
[1, 3, 5, 7, 9]
>>> # Más simple: filter(isOdd, range(10))
```

Las expresiones *reduce()* son extrañas pero también ilustra lo poderosa que es la función en su generaldad: *Cualquier cosa que pueda ser computada a partir de una secuencia de elementos sucesivos puede ser expresada como una reducción.*

7.17.1. Funciones de Orden Superior de Utilería

Una utilidad práctica es *compose()*. Esta es una función que toma una secuencia de funciones y retorna una función que representa la aplicación de cada una de las funciones que se reciben como parámetro a los datos que se reciben como argumento.

HOFs-compose

```
def compose(*funcs):
 """Retorna una nueva función s.t.
 compose(f,g,...)(x) == f(g(...(x)))"""
 def interior(dato, funcs=funcs):
  resultado = dato
  for f in reversed(funcs):
   resultado = f(resultado)
  return resultado
 return interior

# >>> multiplicarPor_2 = lambda x: x*2
# >>> menos_3 = lambda x: x-3
# >>> modulo_6 = lambda x: x%6
# >>> f = compose(mod6, times2, minus3)
# >>> all(f(i)==((i-3)*2)%6 for i in range(1000000))
# True
```

Las funciones primitivas *all()* y *any()* son útiles para preguntar si un predicado se mantiene en los elementos de un iterable. Pero también son buenas para consultar si todos o algunos predicados se cumplen para un ítem de dato en particular de una manera compuesta.

HOFs- all - any

```
all_pred = lambda item, *tests:
                        all(p(item) for p in tests)
any_pred = lambda item, *tests:
                        any(p(item) for p in tests)
```

A continuación se muestran algunos de sus usos:

HOFs- all - any

```
>>> from functools import *
>>> from operator import *
>>> esMenorQue100 = partial(operator.ge, 100)
#Menor que 100?
>>> esMayorQue10 = partial(operator.le, 10)
#Mayor que 10?
>>> from nums import esPrimo
# Implementado en algún lugar
>>> all_pred(71, esMenorQue100, esMayor10, esPrimo)
True
>>> predicados=(esMenorQue100, esMayorQue10,esPrimo)
>>> all_pred(107, *predicados)
False
```

La librería *toolz* tiene una versión más general llamada *juxt()* la cual crea una función que llama diferentes funciones con el mismo argumento y retorna una tupla de resultados.

toolz-juxt

```
>>> from toolz.functoolz import juxt
>>> juxt([menorQue100, mayorQue10, esPrimo])(71)
(True, True, True)
>>> all(juxt([esMenorQue100,esMayor10, esPrimo])(71))
True
>>> juxt([esMenorQue100, esMayorQue10, esPrimo])(107)
(False, True, True)
```

7.18. Funciones Primitivas

A continuación se describen algunas funciones frecuentemente usadas con iteradores.

7.18.1. map

Sintaxis:

map(function, iterable, *iterables)

Tarea: Retorna un iterador que aplica *función* a cada elemento del iterable. Si se le pasan varios iterables la función debe tomar varios argumentos y la misma se aplica a todos los iterables en paralelo. Con múltiples iterables el iterador finaliza cuando finaliza el iterable más corto.

Ejemplo:

map

```
def upper(s):
 return s.upper()

list(map(upper, ['sentencia', 'fragmento']))
['SENTENCIA', 'FRAGMENTO']
```

map-Comentario

Se puede alcanzar el mismo efecto con comprensión de listas.

7.18.2. filter

Sintaxis:

filter(predicado, iterable)

Tarea: Construye un iterador a partir de los elementos de iterable cuyo valor de predicado es verdadero. *iterable* puede ser una secuencia, un contenedor

que permite iteración o un iterador. Si *función* es None, se asume la función identidad y por lo tanto todos los elementos del iterable que son falsos se eliminan.

Ejemplo:

map

```
def esPar(x):
 return (x % 2) == 0

list(filter(esPar, range(10)))
[0, 2, 4, 6, 8]
```

filter-Comentario

Se puede alcanzar el mismo efecto con comprensión de listas.
Note que *filter(funcion, iterable)* es equivalente a la exresión generadora:

```
(item for item in iterable if función(item))
```

Si *función* no es None.

```
(item for item in iterable if item)
```

Si función es None.
La función complementaria itertools.filterfalse() realiza la tarea complementaria a *filter* es decir retorna los elementos de un iterable para los cuales la función produce como resultado *false*.

7.18.3. enumerate

Sintaxis:

enumerate(iterable, comienzo=0)

Tarea: Retorna un objeto enumeración. *iterable* debe ser una secuencia o un objeto que que soporte una iteración. El método *__next()__* del iterador

retornado por *enumerate()* retrona una tupla que contiene una cuenta (desde el comienzo que es por defecto 0) y los valores contenidos en el iterable. **Ejemplo:**

enumerate

```
for item in enumerate(['subject', 'verb', 'object']):
 print(item)
(0, 'subject')
(1, 'verb')
(2, 'object')
```

enumerate-Comentario

enumerate() se usa frecuentemente cuando se hace un loop a través de una lista y se registran los índices de los ítems que cumplen ciertas condiciones.

```
f = open('data.txt', 'r')
for i, line in enumerate(f):
 if line.strip() == '':
  print('Blanco en la línea #%i' % i)
```

7.18.4. sorted

Sintaxis:

sorted(iterable, key=None, reverse=False)

Tarea: Retorna una lista ordenada a partir de los ítems en el iterable. Tiene dos argumentos opcionales los cuales están especificados como argumentos de palabra clave.

Key especifica una función de un argumento que se utiliza para extraer una clave para la comparación. El valor por defecto es *None*.

reverse es un valor booleano. Si es True se realiza un ordenamiento inverso.

Ejemplo:

sorted

```
import random
# Generate 8 random numbers between [0, 10000)
rand_list = random.sample(range(10000), 8)
rand_list
[769, 7953, 9828, 6431, 8442, 9878, 6213, 2207]
sorted(rand_list)
[769, 2207, 6213, 6431, 7953, 8442, 9828, 9878]
sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]
```

7.18.5. all(iter)

Sintaxis:

all(iterable)

Tarea: Retorna True si todos los elementos del iterable son verdaderos o si el iterable es vacío.

Ejemplo:

all

```
all([0, 1, 0])
False
all([0, 0, 0])
False
all([1, 1, 1])
True
```


7.18.6. any(iter)

Sintaxis:

any(iterable)

Tarea: Retorna True si algún elemento del iterable es verdadero. Si el iterable es vacío retorna False.

Ejemplo:

any

```
any([0, 1, 0])
True
any([0, 0, 0])
False
any([1, 1, 1])
True
```

7.18.7. zip

Sintaxis:

zip(*iterables, strict=False)

Tarea: Itera sobre diferentes iterables en paralelo. Produce tuplas con un ítem por cada uno. No construye una lista exhaustiva en memoria de todos los iteradores. En cambio, las tuplas se construyen y retornan solo si se requieren. Es decir se realiza una *Evaluación Perezosa*. Esta función se intenta usar cuando todos los iterables tienen la misma longitud. Si los iterables son de distinta longitud la función termina cuando finaliza con el más corto. En consecuencia, el resultado tendrá la longitud igual que el iterable más corto.

strict se utiliza para producir un error cuando los iterables son de diferente longitud. Para lograr este efecto este parámetro tiene que estar inicializado en True.

Ejemplo:

zip

```
zip(['a', 'b', 'c'], (1, 2, 3)) =>
('a', 1), ('b', 2), ('c', 3)

zip(['a', 'b'], (1, 2, 3)) =>
('a', 1), ('b', 2)
```

7.19. Itertools

El módulo *itertools* contiene un número de iteradores comúnmente usados como así también funciones para combinar diferentes iteradores.

Las funciones del módulo se ubican en cuatro amplias clases:

- Funciones que crean un iterador basados en un iterador existente.
- Funciones que tratan elementos de un iterador como argumentos de función.
- Funciones para seleccionar porciones de un iterador.
- Una función para agrupar la salida de un iterador.

7.19.1. Creación de un Iterador

7.19.2. count

Sintaxis:

itertools.count(comienzo, intervalo)

Tarea: retorna una secuencia infinita de valores. Se puede proporcionar el número de comienzo, el cual por defecto es 0, y el intervalo entre números el cual por defecto es 1.

Ejemplo:

itertools.count

```
itertools.count() =>
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
itertools.count(10) =>
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
itertools.count(10, 5) =>
10, 15, 20, 25, 30, 35, 40, 45, 50, 55, ...
```

7.19.3. cycle

Sintaxis:

itertools.cycle(iter)

Tarea: Graba una copia del contenido de un iterable y retorna un nuevo iterador que devuelve sus elementos desde el comienzo hasta el final. El nuevo iterador repetirá esos elementos desde el principio hasta el final.
Ejemplo:

itertools.cycle

```
itertools.cycle([1, 2, 3, 4, 5]) =>
        1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...
```

7.19.4. repeat

Sintaxis:

itertools.repeat(elem, [n])

Tarea: retorna el elemento recibido como parámetro n veces o devuelve el elemento infinitamente si no se provee n.
Ejemplo:

itertools.repeat

```
itertools.repeat('abc') =>
abc, abc, abc, abc, abc, abc, abc, abc, abc, abc, ...
itertools.repeat('abc', 5) =>
abc, abc, abc, abc, abc
```

7.19.5. chain

Sintaxis:

itertools.chain(iterA, iterB, ...)

Tarea: Toma un número de iterables arbitrarios como entrada y retorna todos los elementos del primer iterador, luego todos los elementos del segundo iterador, y así siguiendo hasta que todos los iterables se hayan consumido.
Ejemplo:

itertools.chain

```
itertools.chain(['a', 'b', 'c'], (1, 2, 3)) =>
a, b, c, 1, 2, 3
```

7.19.6. islice

Sintaxis:

itertools.islice(iter, [start], stop, [step])

Tarea: Retorna una rebanada del iterador. Retornará los primeros elementos de iter hasta que se alcance stop. Opcionalmente se puede especificar el elemento de comienzo (start) se obtendrán los elementos comprendidos entre *start* y *stop*. Si además se especifica *step* los elementos se omitirán acordemente. A diferencia de las rebanadas de strings y listas con esta operación no se pueden utilizar índices negativos.

Ejemplo:

itertools.islice

```
itertools.islice(range(10), 8) =>
0, 1, 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8) =>
2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8, 2) =>
2, 4, 6
```

7.19.7. tee

Sintaxis:

itertools.tee(iter, [n])

Tarea: Replica un iterador y retorna n iteradores independientes cada uno de los cuales retornará el contenido de la fuente del iterador. Si no se provee n el valor por defecto es 2. Replicar iteradores requiere salvar contenido del iterador fuente lo que puede consumir mucha memoria si el iterador es grande y uno de los nuevos iteradores e consume más que los otros.

Ejemplo:

itertools.islice

```
itertools.tee( itertools.count() )
#=>iterA , iterB

#iterA ->
#0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

#  iterB ->
#0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```


7.20. LLamado de Funciones sobre Elementos

El módulo *operator* contiene un conjunto de funciones que se corresponden con los operadores de Python. Algunos ejemplos son operator.add(a, b) (suma dos valores a y b), operator.ne(a, b) (a!=b) y operator.attrgetter('id') retorna un llamable que busca el atributo *.id*.

itertools.starmap(func, iter) asume que el iterable retornará tuplas y llama a la función usando esas tuplas como argumentos.

itertools.starmap

```
itertools.starmap(os.path.join,
[('/bin', 'python'), ('/usr', 'bin', 'java'),
('/usr', 'bin', 'perl'), ('/usr', 'bin', 'ruby')])
=>
/bin/python, /usr/bin/java, /usr/bin/perl,
/usr/bin/ruby
```

7.21. Selección de Elementos

Otro conjunto de funciones elige un subconjunto de elementos de un iterador basados en un predicado.

7.21.1. filterfalse

Sintaxis:

itertools.filterfalse(predicate, iter)

Tarea: Es el opuesto a *filter* retorna los elementos para los cuales el predicado da False.

Ejemplo:

itertools.filterfalse

```
itertools.filterfalse(is_even, itertools.count()) =>
1, 3, 5, 7, 9, 11, 13, 15, ...
```

7.21.2. takewhile

Sintaxis:

itertools.takewhile(predicate, iter)

Tarea: Retorna como resultado los elementos de iter mientras se cumpla con el predicado. Una vez que el predicado sea falso el iterador señalará el final de sus resultados.

Ejemplo:

itertools.takewhile

```
def menorQue10(x):
 return x < 10

itertools.takewhile(menorQue10, itertools.count()) =>
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

itertools.takewhile(esPar, itertools.count()) =>
0
```

7.21.3. dropwhile

Sintaxis:

itertools.dropwhile(predicate, iter)

Tarea: Descarta elementos mientras el predicado sea verdadero. Luego retorna el resto de los resultados del iterable.

Ejemplo:

itertools.takewhile

```
itertools.dropwhile(menorQue10, itertools.count()) =>
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...

itertools.dropwhile(esPar, itertools.count()) =>
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

7.21.4. compress

Sintaxis:

itertools.compress(datos, selectores)

Tarea: Toma dos iteradores y retorna solo aquellos elementos *datos* para los cuales el correspondiente elemento de *selectores* es verdadero.

Ejemplo:

itertools.takewhile

```
itertools.compress([1, 2, 3, 4, 5], \
        [True, True, False, False, True]) => \
        1, 2, 5
```

7.22. Funciones Combinatorias

7.22.1. combinations

Sintaxis:

itertools.combinations(iterable, r)

Tarea: Retorna un iterador que da todas las posibles r-tuplas combinaciones de los elementos contenidos en el iterable.
Ejemplo:

itertools.combinations

```
itertools.combinations([1, 2, 3, 4, 5], 2) =>
(1, 2), (1, 3), (1, 4), (1, 5),
(2, 3), (2, 4), (2, 5),
(3, 4), (3, 5),
(4, 5)

itertools.combinations([1, 2, 3, 4, 5], 3) =>
(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4),
(1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5),
(2, 4, 5), (3, 4, 5)
```

itertools.combinations

Los elementos de la tupla quedan en el mismo orden que el iterable. Por ejemplo, el número 1 está siempre antes que 2,3,4 o 5 en el ejemplo anterior.

7.22.2. permutations

Sintaxis:

itertools.permutations(iterable, r=None)

Tarea: Realiza una tarea similar a *combinations* excepto que retorna todos los posibles valores de longitud r.
Ejemplo:

itertools.permutations

```
itertools.permutations([1, 2, 3, 4, 5], 2) =>
(1, 2), (1, 3), (1, 4), (1, 5),
(2, 1), (2, 3), (2, 4), (2, 5),
(3, 1), (3, 2), (3, 4), (3, 5),
(4, 1), (4, 2), (4, 3), (4, 5),
(5, 1), (5, 2), (5, 3), (5, 4)

itertools.permutations([1, 2, 3, 4, 5]) =>
(1, 2, 3, 4, 5), (1, 2, 3, 5, 4), (1, 2, 4, 3, 5),
...
(5, 4, 3, 2, 1)
```

itertools.permutations

Si no se provee un valor de r se utiliza la longitud del iterable.

7.22.3. combinations_with_replacement

Sintaxis:

itertools.combinations_with_replacement(iterable, r)

Tarea: Esta función relaja una restricción diferente: Los elementos se pueden repetir en una tupla.

Ejemplo:

itertools.combinations_with_replacement

```
itertools.combinations_with_replacement(
[1, 2, 3, 4, 5], 2) =>
(1, 1), (1, 2), (1, 3), (1, 4), (1, 5),
(2, 2), (2, 3), (2, 4), (2, 5),
(3, 3), (3, 4), (3, 5),
(4, 4), (4, 5),
(5, 5)
```

7.23. Agrupación de Elementos

7.23.1. itertools.groupby(iter, key_func=None)

Sintaxis:

itertools.groupby(iter, key_func=None)

Tarea: Agrupa los elementos consecutivos de iter que tienen el mismo valor de clave. Retorna un stream de duplas que contienen la clave y el valor y un iterador para los elementos con esa clave. *key_func(elem)* es una función que computa un valor de clave para cada elemento que retorna el iterable. Si la misma no se provee la clave es el mismo elemento.

Ejemplo:

itertools.combinations_with_replacement

```
listaDeCiudades = [
 ('Decatur', 'AL'), ('Huntsville', 'AL'),
 ('Selma', 'AL'), ('Anchorage', 'AK'),
 ('Nome', 'AK'), ('Flagstaff', 'AZ'),
 ('Phoenix', 'AZ'), ('Tucson', 'AZ'),
...
]

def obtenerEstado(ciudad_estado):
 return city_state[1]

itertools.groupby(listaDeCiudades, obtenerEstado) =>
('AL', iterador-1),
('AK', iterador-2),
('AZ', iterador-3), ...

donde
iterador-1 =>
('Decatur', 'AL'), ('Huntsville', 'AL'),
('Selma', 'AL')
iterador-2 =>
('Anchorage', 'AK'), ('Nome', 'AK')
iterador-3 =>
('Flagstaff', 'AZ'), ('Phoenix', 'AZ'),
('Tucson', 'AZ')
```


itertools.permutations

Se asume que el iterable subyacente contiene los elementos ordenados basados en su clave. Note que los iteradores retornados también usan el iterable subyacente. Por lo tanto, se tiene que consumir el *iterator-1* antes que *iterator-2*

7.24. functools

El módulo *functools* contiene algunas HOFs,una de ellas toma como argumento una o más funciones como entrada y retorna una nueva función.

7.24.1. partial

Para programas escritos en un estilo funcional muchas veces se desea variantes de funciones existentes que tienen algunos parámetros fijos. Por ejemplo, si se tiene una función *f(a,b,c)* puede ser deseable crear una función *g(b,c)* que sea equivalente a *f(1,b,c)*. Esto se conoce con el nombre de aplicación de función parcial.

Sintaxis:

functools.partial(func, /, *args, **keywords)

Tarea: Retorna un nuevo objeto parcial el cual cuando se invoca se comporta como *func* con los argumentos **args* y los argumentos de palabra clave ***keywords*. Si se proveen más argumentos se agregan al final de *args*. Si se proveen más argumentos de palabra clave extienden y sobre escriben *keywords*.

functools.partial

```
import functools

def log(mensaje, subsistema):
"""Escribe el contenido de mensaje
 en el subsistema especificado."""
print('%s: %s' % (subsistema, mensaje))
...

logDelServidor= functools.partial(log,
 subsystema='servidor')
logDelServidor('No se puede abrir el Socket')
```

7.24.2. reduce

Sintaxis:

functools.reduce(func, iter, [initial_value])

Tarea: Acumulativamente realiza una operación sobre todos los elementos del iterable, por lo tanto no puede ser aplicada sobre iterables infinitos. *func* es una función que tiene dos argumentos y retorna un valor. *functools.reduce()* toma los primeros dos elementos retornados por el iterador y calcula func(A,B). Luego se requiere el tercer ítem y calcula func(func(A,B),C), combina este resultado con el cuarto elemento y continúa de esta manera hasta que se consuma todo el iterable. Si el iterable no retorna valor se produce una excepción *TypeError*. Si se provee un valor inicial, se utiliza como valor inicial para func (func(initial_value,A)).

functools.reduce

```
import operator, functools
functools.reduce(operator.concat, ['A', 'BB', 'C'])
'ABBC'
```

functools.reduce

```
functools.reduce(operator.concat, [])
Traceback (most recent call last):
...
TypeError: reduce() of empty sequence with no
initial value

functools.reduce(operator.mul, [1, 2, 3], 1)
6
functools.reduce(operator.mul, [], 1)
1
```

7.25. El Módulo operator

El módulo *operator* contiene un conjunto de funciones correspondiente a los operadores de Python. Esas funciones son útiles para el estilo de programación funcional porque evitan que el programador escriba funciones triviales que realicen una operación simple. A continuación se mencionan algunos de los operadores mencionados con anteriorirdad.

- Operaciones Matemáticas: add(), sub(), mul(), floordiv(), abs(), ...
- Operaciones Lógicas: not_(), truth().
- Operaciones a Nivel de Bits: and_(), or_(), invert().
- Comparaciones: eq(), ne(), lt(), le(), gt(), and ge().

- Identidad de Objetos: is_(), is_not().

7.26. Ejercicios

Ejercicio 1: Defina una función lambda que calcule el área de un triángulo. Luego la invoca con un triángulo cuya base es 10 y altura 20.

Ejercicio 2: Defina una función lambda que calcule el perímetro de un cuadrado. Luego invoque la función con un cuadrado cuyo lado vale 20cm.

Ejercicio 3: Escriba una función que genere los n primeros números pares donde n es ingresado por el usuario.

Ejercicio 4: Escriba una función que genere los n primeros números múltiplos de m. Tanto n y m son ingresados por el usuario.

Ejercicio 5: Escriba una función que genere las n primeras potencias de m. Tanto n y m son ingresadas por el usuario.

Ejercicio 6: Escriba un programa que dada una matriz de nxn permita sumar le valor mínimo de cada columna. La matriz es ingresada por el usuario.

Ejercicio 7: Escriba un programa que permita ingresar a la lista n valores aleatorios. El programa debe producir como resultado la suma de los valores. El valor de n es ingresado por el usuario.

Ejercicio 8: Escriba un programa que permita que el usuario ingrese por teclado un árbol binario. La información almacenada en los nodos del árbol es un número entero.

Ejercicio 9: Escriba una función que permita recorrer un árbol de forma preorden. El árbol se encuentra representado como una lista de listas.

Ejercicio 10: Escriba una función que permita recorrer un árbol de forma posorden. El árbol se encuentra representado como una lista de listas.

Ejercicio 11: El siguiente procedimiento explica como se realiza una búsqueda binaria:

Dado un vector A de n elementos con valores $A_0...A_{n-1}$ ordenados de tal forma que $A_0 \leq .. \leq A_{n-1}$ y un valor buscado T, el siguiente procedimiento usa búsqueda binaria para encontrar el índice de T en A.

a) Asignar 0 a L y a R (n - 1).

b) Si L >R, la búsqueda termina sin encontrar el valor.

c) Sea m (la posición del elemento del medio) igual a la parte entera de (L + R)/2.

d) Si A_m <T, igualar L a m + 1 e ir al paso 2.

e) Si A_m >T, igualar R a m – 1 e ir al paso 2.

f) Si A_m = T, la búsqueda terminó, retornar m.

Impleméntla usando el estilo de programación funcional.

Ejercicio 12: Escriba una función que reciba como parámetros una lista l y un número n. La función produce como resultado una triupla donde el primer elemento es una lista que contiene los números menores que n, el segundo elemento es una lista que contiene los números iguales a n, y la tercer componente es una lista que contiene los números mayores que n.

Ejercicio 13: Defina funciones lambda que:

- Sumen dos números.
- Elimine los espacios en blanco de un string
- Devuelva el mayor de dos números.
- Invierta un string.

Ejercicio 14: Escriba una función que permita calcular el máximo común divisor de dos números ingresados por el usuario.

Ejercicio 15: Escriba una función que reciba como parámetro una lista de string ls y un string s. La función retorna como resultado una lista que tiene los elementos de ls sin las ocurrencias de s.

Ejercicio 16: Escriba una función que genere una matriz de 10 x 10 donde cada fila contiene la tabla de multiplicar del nombre de la fila. Así la fila 1 contiene la tabla de multiplicar del 1, la fila 2 contiene la tabla de multiplicar de 2 y así siguiendo.