Orientación a Objetos en Python

Algoritmia
Grado en Ingeniería Informática
Universidad de Burgos

Juan José Rodríguez Diez

Contenido

Abstracción

- Tratamiento implícito de las abstracciones, *duck typing*.
- No hay comprobación de tipos en compilación.
- Asumir que el objeto soporta un conjunto de comportamientos conocidos.
- Error en tiempo de ejecución si las suposiciones fallan.

Duck typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

[Goodrich et al., 2013, pág 60]

Clases base abstractas

- *Abstract Base Class* (ABC).
- No pueden ser instanciadas.
- Define uno o más métodos que todas las implementaciones de la abstracción deben tener.
- Clases *concretas* que heredan de la abstracta y proporcionan implementaciones de los métodos declarados en la clase abstracta.
- El módulo *abc* proporciona soporte formal para ABCs.
- En el módulo *collections* hay ABCs e implementaciones concretas de tipos abstractos de datos.

[Goodrich et al., 2013, pág 60]

Encapsulación

- Los componentes software no deberían revelar los detalles internos de sus implementaciones.
- Soporte escaso en Python.
- Por convenio, los miembros de una clase que empiezan por un carácter único de subrayado se asume que no son públicos.
  - E.g., *_secreto*.
  - Esos miembros no deberían utilizarse desde fuera de la clase.
- Al generar documentación se omiten esos miembros.

[Goodrich et al., 2013, pág 60]

Estilo (I)

- Guía de estilo: [van Rossum et al., 2001]
- Bloques indentados por 4 espacios, evitar tabuladores.
- Clases: nombre singular, primera letra en mayúsculas (e.g. *Fecha*).
  - Si varias palabras concatenadas, en "*CamelCase*" (e.g., *CreditCard*).
- Funciones (incluyendo miembros de clases) en minúsculas.
  - Si varias palabras, separar por subrayado. E.g., *realizar_pago* .
  - Habitualmente, verbo que describa el efecto.
    - Si solo devuelve un valor, se podría usar el nombre del valor. E.g., *sqrt* en vez de *calculate_sqrt*

[Goodrich et al., 2013, pág 65]

Estilo (II)

- Identificadores de objetos individuales (e.g., parámetros, variables): sustantivo en minúsculas.
  - E.g., *precio*.
  - Separar palabras por subrayados.
- Identificadores de constantes: todo en mayúsculas y separados por subrayados.
  - E.g., *MAX_SIZE*.
- Comentarios de una línea con #.
- Comentarios de varias líneas: cadenas con múltiples lineas (triples comillas).
  - Primera sentencia de un módulo, clase o función: *docstring*.

[Goodrich et al., 2013, págs 65-66]

Clases

- Palabra reservada **class**, seguida del nombre de la clase, dos puntos y el cuerpo indentando.
- El cuerpo incluye las definiciones de todos los métodos de la clase.
- Los métodos se definen como funciones, pero con un parámetro especial denominado *self*.
  - Identifica a la instancia particular sobre la que se invoca el método.
- Al llamar al método, un parámetro menos.

[Goodrich et al., 2013, págs 69]

Ejemplo de clase (I)

```
1  class CreditCard:
2    """ A consumer credit card."""
3
4    def __init__ ( self , customer, bank, acnt, limit ):
5      """ Create a new credit card instance.
6
7  The initial balance is zero.
8
9      customer  the name of the customer (e.g., 'John Bowman')
10     bank      the name of the bank (e.g., 'California Savings')
11     acnt      the acount identifier (e.g., '5391 0375 9387 5309')
12     limit     credit limit (measured in dollars)
13     """
```

[Goodrich et al., 2013, pág 70]

Ejemplo de clase (II)

```
14        self._customer = customer
15        self._bank = bank
16        self._account = acnt
17        self._limit = limit
18        self._balance = 0
19
20    def get_customer( self ):
21        """Return name of the customer."""
22        return self._customer
23
24    def get_bank( self ):
25        """Return the bank's name."""
26        return self._bank
```

[Goodrich et al., 2013, pág 70]

Ejemplo de clase (III)

```
27    def get_account( self ):
28        """ Return␣the␣card␣identifying␣number␣(typically␣stored
29    ␣␣␣␣as␣a␣string)."""
30        return  self ._account
31
32    def  get_limit ( self ):
33        """ Return␣current␣credit␣limit . """
34        return  self . _limit
35
36    def  get_balance( self ):
37        """ Return␣current␣balance. """
38        return  self ._balance
```

[Goodrich et al., 2013, pág 70]

Ejemplo de clase (IV)

```
39    def charge( self ,  price ):
40        """ Charge␣given␣price␣to␣the␣card,␣assuming␣sufficient
41    ␣␣␣␣credit␣limit .
42
43    ␣␣␣␣Return␣True␣if␣charge␣was␣processed;␣False␣if␣charge␣was
44    ␣␣␣␣denied.
45        """
46        if  price  +  self ._balance  >  self . _limit :
47            # if charge would exceed limit , cannot accept charge
48            return False
49        else:
50            self ._balance  +=  price
51            return True
```

[Goodrich et al., 2013, págs 71-73]

Ejemplo de clase (V)

```
52    def make_payment(self, amount):
53        """ Process␣customer␣payment␣that␣reduces␣balance."""
54        self._balance -= amount
55
56 if __name__ == '__main__':
57    wallet = []
58    wallet.append(CreditCard('John␣Bowman', 'California␣Savings',
59                             '5391␣0375␣9387␣5309', 2500) )
60    wallet.append(CreditCard('John␣Bowman', 'California␣Federal',
61                             '3485␣0399␣3395␣1954', 3500) )
62    wallet.append(CreditCard('John␣Bowman', 'California␣Finance',
63                             '5391␣0375␣9387␣5309', 5000) )
```

[Goodrich et al., 2013, pág 71]

Ejemplo de clase (VI)

```
64  for val in range(1, 17):
65      wallet[0].charge(val)
66      wallet[1].charge(2*val)
67      wallet[2].charge(3*val)
68
69  for c in range(3):
70      print('Customer = ', wallet[c].get_customer())
71      print('Bank = ', wallet[c].get_bank())
72      print('Account = ', wallet[c].get_account())
73      print('Limit = ', wallet[c].get_limit())
74      print('Balance = ', wallet[c].get_balance())
75      while wallet[c].get_balance() > 100:
76          wallet[c].make_payment(100)
77          print('New balance = ', wallet[c].get_balance())
78      print()
```

[Goodrich et al., 2013, pág 73]

Constructor

- Creación de instancias:

```
1 cc = CreditCard( 'John Doe', '1st Bank',
2   '5391 0375 9387 5309', 1000)
```

- Método especial, denominado *__init__* .
- Establecer el estado del objeto creado.
- Cinco variables de instancia: *_customer*, *_bank*, *_account*, *_limit* , *_balance*.
- Los identificadores sin cualificar (e.g., *customer*) se refiere al parámetro en el espacio de nombres local.

[Goodrich et al., 2013, pág 71]

Sobrecarga de operadores (I)

- En las clases predefinidas, se pueden usar varios operadores. E.g: $a+b$.
- Al definir una clase, podemos considerar el uso de estos operadores.
- Métodos con nombres especiales. E.g, *__add__*.
- $a + b$ se convierte en una llamada a un método $a.\_\_add\_\_(b)$
- Si operaciones con tipos distintos (e.g., `3 * 'a'`), se intenta con el tipo de la izquierda.
  - Si no está definido, con el tipo de la derecha. E.g., *__rmul__*
- La distinción entre *__mul__* y *__rmul__* es útil cuando las operaciones no son conmutativas.
  - E.g., multiplicar matriz por vector.

[Goodrich et al., 2013, pág 74]

Sobrecarga de operadores (II)

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)
a += b	a.__iadd__(b)	
a −= b	a.__isub__(b)	
a *= b	a.__imul__(b)	
...	...	

[Goodrich et al., 2013, pág 75]

Sobrecarga de operadores (III)

+a	a.__pos__()
−a	a.__neg__()
~a	a.__invert__()
abs(a)	a.__abs__()
a < b	a.__lt__(b)
a <= b	a.__le__(b)
a > b	a.__gt__(b)
a >= b	a.__ge__(b)
a == b	a.__eq__(b)
a != b	a.__ne__(b)
v in a	a.__contains__(v)
a[k]	a.__getitem__(k)
a[k] = v	a.__setitem__(k,v)
del a[k]	a.__delitem__(k)

a(arg1, arg2, ...)	a.__call__(arg1, arg2, ...)
len(a)	a.__len__()
hash(a)	a.__hash__()
iter(a)	a.__iter__()
next(a)	a.__next__()
bool(a)	a.__bool__()
float(a)	a.__float__()
int(a)	a.__int__()
repr(a)	a.__repr__()
reversed(a)	a.__reversed__()
str(a)	a.__str__()

[Goodrich et al., 2013, pág 75]

Clase *Vector* (I)

```
1  class Vector:
2      """Represent a vector in a multidimensional space."""
3
4      def __init__(self, d):
5          if isinstance(d, int):
6              self._coords = [0] * d
7          else:
8              try:      # we test if param is iterable
9                  self._coords = [val for val in d]
10             except TypeError:
11                 raise TypeError('invalid parameter type')
```

[Goodrich et al., 2013, pág. 78]

Clase *Vector* (II)

```
12    def __len__(self):
13        """Return the dimension of the vector."""
14        return len(self._coords)
15
16    def __getitem__(self, j):
17        """Return jth coordinate of vector."""
18        return self._coords[j]
19
20    def __setitem__(self, j, val):
21        """Set jth coordinate of vector to given value."""
22        self._coords[j] = val
```

[Goodrich et al., 2013, pág. 78]

Clase *Vector* (III)

```
23    def __add__( self , other ):
24        """Return sum of two vectors."""
25        if len( self ) != len(other ):    # relies on __len__ method
26            raise ValueError('dimensions must agree')
27        result = Vector(len( self ))  # start with vector of zeros
28        for j in range(len( self )):
29            result [j] = self [j] + other[j]
30        return result
```

- *other* puede que no sea de la clase *Vector*.

[Goodrich et al., 2013, pág. 78]

Clase *Vector* (IV)

```
1   def __eq__( self , other ):
2       """Return True if vector has same coordinates as other."""
3       return self._coords == other._coords
4
5   def __ne__( self , other ):
6       """Return True if vector differs from other."""
7       return not self == other # rely on existing __eq__
8
9   def __str__ ( self ):
10      """Produce string representation of vector. """
11      return '<' + str( self._coords)[1:-1] + '>'
12              # adapt list representation
```

[Goodrich et al., 2013, pág. 78]

Clase *Vector* (V)

```
13    def __neg__( self ):
14        """ Return copy of vector with all coordinates negated. """
15        result = Vector(len( self ))    # start with vector of zeros
16        for j in range(len( self )):
17            result [j] = -self[j]
18        return result
```

[Goodrich et al., 2013, pág. 78]

Clase *Vector* (VI)

```
19    def __lt__(self, other):
20        """Compare vectors based on lexicographical order."""
21        if len(self) != len(other):
22            raise ValueError('dimensions must agree')
23        return self._coords < other._coords
24
25    def __le__(self, other):
26        """Compare vectors based on lexicographical order."""
27        if len(self) != len(other):
28            raise ValueError('dimensions must agree')
29        return self._coords <= other._coords
```

[Goodrich et al., 2013, pág. 78]

Clase *Vector* (VII)

```
30  # the following demonstrates usage of a few methods
31  v = Vector(5)    # construct five-dimensional <0, 0, 0, 0, 0>
32  v[1] = 23        # <0, 23, 0, 0, 0> (based on use of __setitem__)
33  v[-1] = 45       # <0, 23, 0, 0, 45> (also via __setitem__)
34  print(v[4])      # print 45 (via __getitem__)
35  u = v + v        # <0, 46, 0, 0, 90> (via __add__)
36  print(u)         # print <0, 46, 0, 0, 90>
37  total = 0
38  for entry in v: # implicit iteration via __len__ and __getitem__
39      total += entry
```

[Goodrich et al., 2013, pág. 78]

Iteradores (I)

- Iterador: método *__next__*
  - Devuelve el siguiente elemento o lanza *StopIteration* .
- No es habitual implementar directamente clases iterador.
- Generadores, producen automáticamente un iterador sobre los valores generados.
- Iterador automático para cualquier clase que defina *__len__* y *__getitem__*

[Goodrich et al., 2013, pág. 79]

Iteradores (II)

```
1  class SequenceIterator:
2    """ An iterator for any of Python's sequence types."""
3
4    def __init__ (self, sequence):
5      """ Create an iterator for the given sequence."""
6
7      self._seq = sequence
8          # keep a reference to the underlying data
9
10     self._k = -1
11         # will increment to 0 on first call to next
```

[Goodrich et al., 2013, pág. 79]

Iteradores (III)

```
12    def __next__( self ):
13        """ Return␣the␣next␣element,␣or␣else␣raise␣StopIteration
14    ␣␣␣␣error. """
15        self ._k += 1                       # advance to next index
16        if  self ._k < len( self ._seq ):
17          return( self ._seq[ self ._k]) # return the data element
18        else :
19          raise  StopIteration ()    # there are no more elements
20
21    def  __iter__ ( self ):
22        """ By␣convention,␣an␣iterator␣must␣return␣itself␣as␣an
23    ␣␣␣␣iterator. """
24        return  self
```

[Goodrich et al., 2013, pág. 79]

Clase *Range* (I)

```
1  class Range:
2      """ A class that mimic's the built-in range class. """
3
4      def __init__ ( self , start , stop=None, step=1):
5          """ Initialize a Range instance.
6      Semantics is similar to built-in range class.
7          """
8          if step == 0:
9              raise ValueError('step cannot be 0')
10
11         if stop is None:     # special case of range(n)
12             # should be treated as if range(0,n)
13             start , stop = 0, start
```

[Goodrich et al., 2013, pág. 81]

Clase *Range* (II)

```
14        # calculate the effective length once
15        self._length = max(0, (stop - start + step - 1) // step)
16
17        # need knowledge of start and step (but not stop) to
18        # support __getitem__
19        self._start = start
20        self._step = step
21
22    def __len__(self):
23        """Return number of entries in the range."""
24        return self._length
```

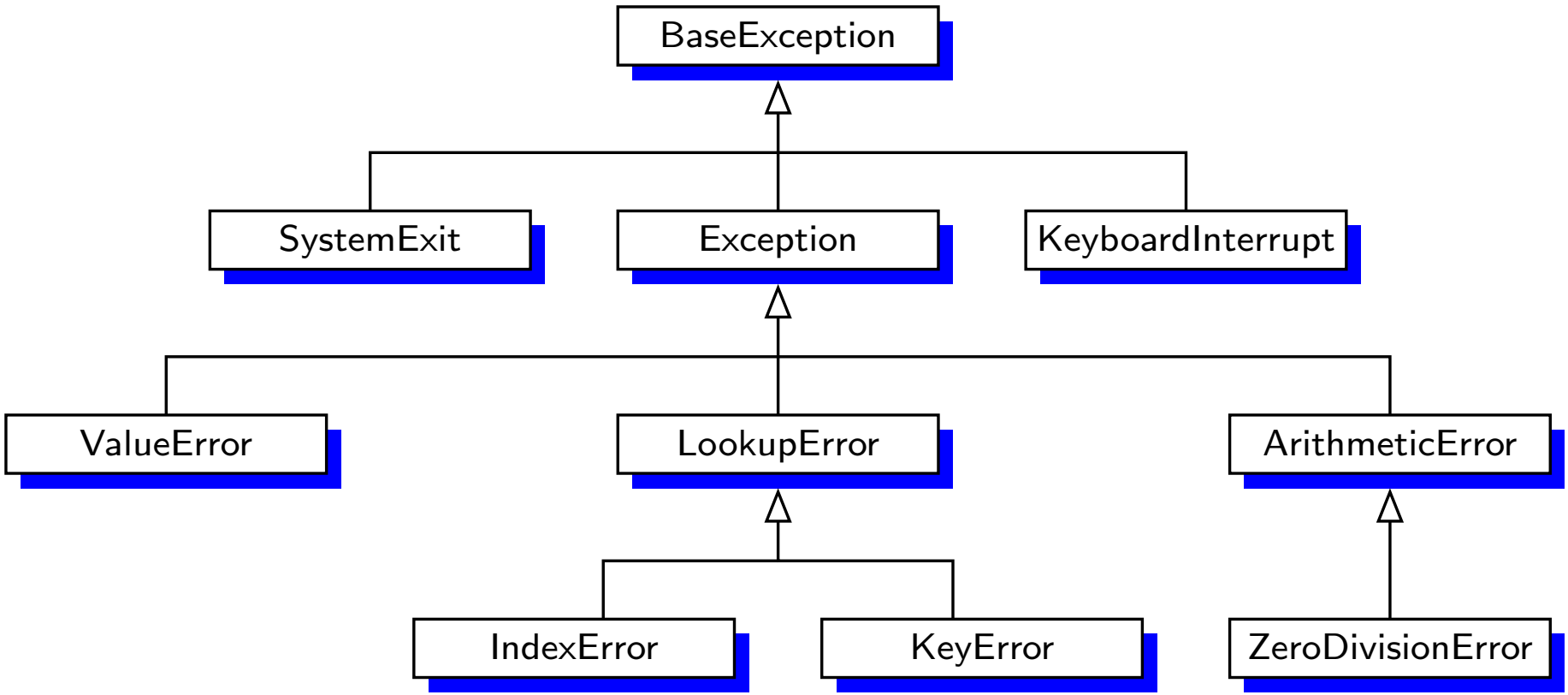
[Goodrich et al., 2013, pág. 81]

Clase *Range* (III)

```
25    def __getitem__( self , k):
26        """Return␣entry␣at␣index␣k␣(using␣standard␣interpretation
27    ␣␣␣␣if␣negative)."""
28        if k < 0:
29            k += len(self)        # attempt to convert negative index
30
31        if not 0 <= k < self._length:
32            raise IndexError('index␣out␣of␣range')
33
34        return self._start + k * self._step
```

[Goodrich et al., 2013, pág. 81]

Jerarquía de excepciones



[Goodrich et al., 2013, pág. 83]

Clase descendiente (I)

Class:	CreditCard	
Fields:	_customer	_balance
	_bank	_limit
	_account	
Behaviors:	get_customer( )	get_balance( )
	get_bank( )	get_limit( )
	get_account( )	charge(price)
	make_payment(amount)	

Class:	PredatoryCreditCard	
Fields:	_apr	
Behaviors:	process_month( )	charge(price)

[Goodrich et al., 2013, pág. 84]

Clase descendiente (II)

```
1  class PredatoryCreditCard(CreditCard):
2    """An extension to CreditCard that compounds interest and fees."""
3    def __init__(self, customer, bank, acnt, limit, apr):
4      """Create a new predatory credit card instance.
5      The initial balance is zero.
6      customer  the name of the customer (e.g., 'John Bowman')
7      bank      the name of the bank (e.g., 'California Savings')
8      acnt      the acount identifier (e.g., '5391 0375 9387 5309')
9      limit     credit limit (measured in dollars)
10     apr       annual percentage rate (e.g., 0.0825 for 8.25% APR)
11     """
12     super().__init__(customer, bank, acnt, limit) # super construc.
13     self._apr = apr
```

[Goodrich et al., 2013, pág. 85]

Clase descendiente (III)

```
14    def charge( self ,  price ):
15        """ Charge␣given␣price␣to␣the␣card,␣assuming␣sufficient
16    ␣␣␣␣credit␣limit.
17
18    ␣␣␣␣Return␣True␣if␣charge␣was␣processed.
19    ␣␣␣␣Return␣False␣and␣assess␣5 fee if charge is denied.
20        """
21        success = super().charge( price )  # call inherited method
22        if not success:
23            self._balance += 5  # assess penalty
24        return success              # caller expects return value
```

[Goodrich et al., 2013, pág. 85]

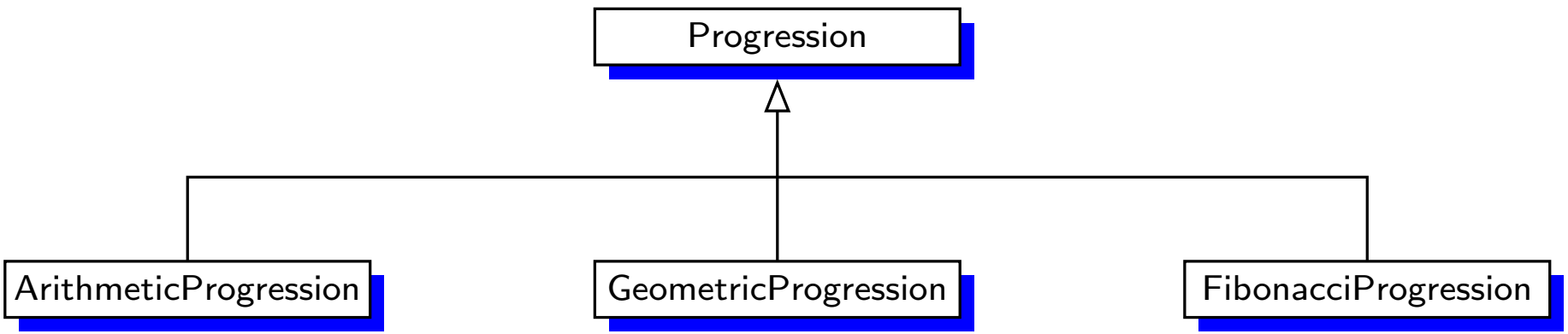
Clase descendiente (IV)

```
25    def process_month( self ):
26        """ Assess␣monthly␣interest␣on␣outstanding␣balance."""
27        if  self ._balance > 0:
28            # if  positive  balance,  convert  APR to monthly
29            #  multiplicative  factor
30            monthly_factor = pow(1 + self._apr, 1/12)
31            self ._balance *= monthly_factor
```

[Goodrich et al., 2013, pág. 85]

Jerarquía de progresiones

- Progresión numérica: secuencia de números, cada número depende del anterior o varios anteriores.
- Valor inicial, como obtener un nuevo valor a partir de los anteriores.



[Goodrich et al., 2013, pág. 87]

Progresión base (I)

```
1  class Progression:
2      """ Iterator producing a generic progression.
3
4      Default iterator produces the whole numbers 0, 1, 2, ...
5      """
6
7      def __init__(self, start=0):
8          """ Initialize current to the first value of the
9          progression.
10         """
11         self._current = start
```

[Goodrich et al., 2013, pág. 88]

Progresión base (II)

```
12    def _advance( self ):
13        """ Update␣self._current␣to␣a␣new␣value.
14
15    ␣␣␣␣This␣should␣be␣overridden␣by␣a␣subclass␣to␣customize
16    ␣␣␣␣progression.
17
18    ␣␣␣␣By␣convention,␣if␣current␣is␣set␣to␣None,␣this␣designates
19    ␣␣␣␣the␣end␣of␣a␣finite␣progression.
20    ␣␣␣␣"""
21        self._current += 1
```

[Goodrich et al., 2013, pág. 88]

Progresión base (III)

```
22    def __next__( self ):
23        """ Return␣the␣next␣element,␣or␣else␣raise␣ StopIteration
24    ␣␣␣␣error.
25        """
26        if  self ._current  is  None:  # our convention to end
27            raise  StopIteration ()
28        else:
29            answer = self ._current  # record current value to return
30            self ._advance()  # advance to prepare for next time
31            return answer  # return the answer
```

[Goodrich et al., 2013, pág. 88]

Progresión base (IV)

```
32    def __iter__ ( self ):
33        """ By␣convention,␣an␣iterator␣must␣return␣itself␣as␣an
34    ␣␣␣␣iterator.
35    ␣␣␣␣"""
36        return self
37
38    def print_progression ( self , n):
39        """ Print␣next␣n␣values␣of␣the␣progression."""
40        print(' ␣'.join(str(next( self )) for j in range(n)))
```

[Goodrich et al., 2013, pág. 88]

Progresión aritmética

```
41 class ArithmeticProgression(Progression):
42     """ Iterator producing an arithmetic progression. """
43
44     def __init__ (self, increment=1, start=0):
45         """ Create a new arithmetic progression.
46 ␣␣␣␣increment␣␣the fixed constant to add to each term
47 ␣␣␣␣start␣␣␣␣␣␣the first term of the progression
48 ␣␣␣␣        """
49         super(). __init__ (start)          # initialize base class
50         self._increment = increment
51
52     def _advance(self):                    # override inherited version
53         """ Update current value by adding the fixed increment."""
54         self._current += self._increment
```

[Goodrich et al., 2013, pág. 89]

Progresión geométrica

```
55 class GeometricProgression( Progression ):
56     """ Iterator producing a geometric progression."""
57
58     def __init__ ( self , base=2, start=1):
59         """ Create a new geometric progression.
60 base        the fixed constant multiplied to each term
61 start       the first term of the progression
62         """
63         super(). __init__ ( start )
64         self ._base = base
65
66     def _advance( self ):          # override inherited version
67         """ Update current value by multiplying it by the base."""
68         self ._current *= self ._base
```

[Goodrich et al., 2013, pág. 90]

Fibonacci

```
69 class FibonacciProgression(Progression):
70     """ Iterator producing a generalized Fibonacci progression """
71
72     def __init__ (self, first=0, second=1):
73         """ Create a new fibonacci progression.
74 first           the first term of the progression (default 0)
75 second          the second term of the progression (default 1)
76         """
77         super().__init__ (first)    # start progression at first
78         self._prev = second - first  # fictitious previous
79
80     def _advance(self):
81         """ Update current value by taking sum of previous two."""
82         self._prev, self._current = self._current,\
83                                     self._prev + self._current
```

[Goodrich et al., 2013, pág. 91]

Prueba de progresiones (I)

```
84 if __name__ == '__main__':
85     print('Default progression:')
86     Progression().print_progression(10)
87
88     print('Arithmetic progression with increment 5:')
89     ArithmeticProgression(5).print_progression(10)
90
91     print('Arithmetic progression with increment 5 and start 2:')
92     ArithmeticProgression(5, 2).print_progression(10)
```

[Goodrich et al., 2013, pág. 92]

Prueba de progresiones (II)

```
93  print('Geometric progression with default base:')
94  GeometricProgression(). print_progression (10)
95
96  print('Geometric progression with base 3:')
97  GeometricProgression(3). print_progression (10)
98
99  print('Fibonacci progression with default start values:')
100 FibonacciProgression (). print_progression (10)
101
102 print('Fibonacci progression with start values 4 and 6:')
103 FibonacciProgression(4, 6). print_progression (10)
```

[Goodrich et al., 2013, pág. 92]

Clases base abstractas

- Clases que no pueden ser instanciadas.
- En lenguajes estáticamente tipados dan soporte al polimorfismo.
  - Variable declarada del tipo de la clase abstracta, referencia a un objeto de una clase concreta.
- En Python, al no haber declaración de tipos, se tiene polimorfismo sin necesidad de una clase base abstracta.
- No hay tradición de usar clases bases abstractas.
- El módulo *abc* da soporte a la definición de este tipo de clases.

[Goodrich et al., 2013, pág. 93]

collections .*Sequence*

- La clase *collections* .*Sequence* es una clase abstracta que define comportamientos comunes de **list** , **str** y **tuple**.
- Secuencias que suportan el acceso por índice.
- Implementaciones concretas de los métodos: *count*, *index* y *__contains__* .
- Una clase puede heredar esos métodos, que se basan en *__len__* y *__getitem__* .

[Goodrich et al., 2013, pág. 93]

Clase *Sequence* (I)

```
1  from abc import ABCMeta, abstractmethod
2                      # need these  definitions
3
4  class  Sequence(metaclass=ABCMeta):
5     """ Our␣own␣version␣of␣collections.Sequence␣abstract␣base
6  ␣␣class. """
7
8     @abstractmethod
9     def  __len__ ( self ):
10       """ Return␣the␣length␣of␣the␣sequence."""
11
12    @abstractmethod
13    def  __getitem__ ( self ,  j ):
14       """ Return␣the␣element␣at␣index␣j␣of␣the␣sequence."""
```

[Goodrich et al., 2013, pág. 94]

Clase *Sequence* (II)

```
15    def __contains__(self, val):
16        """Return True if val found in the sequence;
17    False otherwise."""
18
19        for j in range(len(self)):
20            if self[j] == val:     # found match
21                return True
22        return False
```

[Goodrich et al., 2013, pág. 94]

Clase *Sequence* (III)

```
23    def index( self , val ):
24        """ Return␣leftmost␣index␣at␣which␣val␣is␣found
25    ␣␣␣␣(or␣raise␣ValueError). """
26
27        for j in range(len( self )):
28            if self [j] == val:    # leftmost match
29                return j
30
31        # never found a match
32        raise ValueError( 'value␣not␣in␣sequence')
```

[Goodrich et al., 2013, pág. 94]

Clase *Sequence* (IV)

```
33    def count( self ,  val ):
34        """ Return the number of elements equal to given value."""
35
36        k = 0
37        for  j  in  range(len( self )):
38            if  self [j] == val:     # found a match
39                k += 1
40
41        return k
```

[Goodrich et al., 2013, pág. 94]

Atributos de clase / Clases anidadas

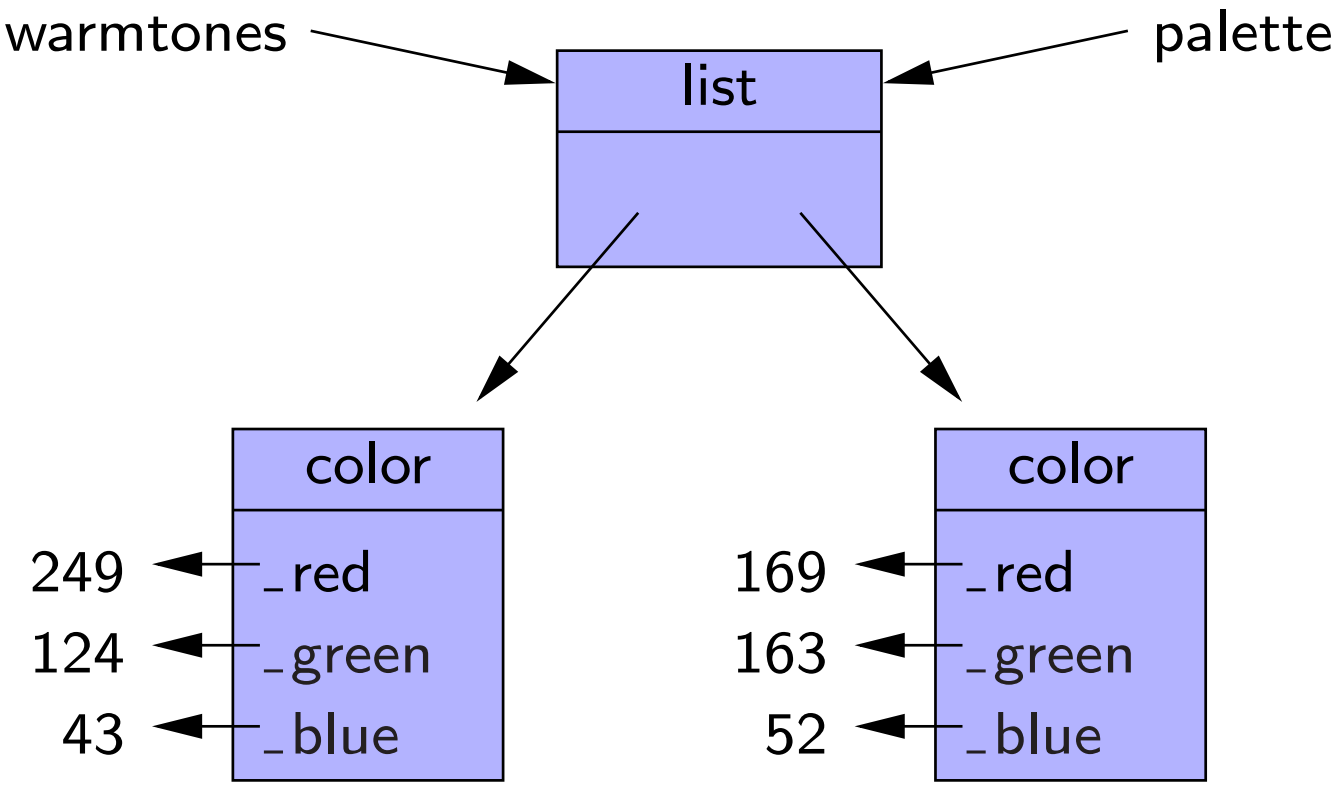
```
1 class PredatoryCreditCard(CreditCard):
2     OVERLIMIT_FEE = 5 # this is a class-level member
3
4     def charge( self , price ):
5         success = super( ).charge( price )
6         if not success:
7             self . balance +=
8                 PredatoryCreditCard.OVERLIMIT_FEE
9         return success
```

```
1 class A:          # the outer class
2     class B:      # the nested class
3         ...
```

[Goodrich et al., 2013, pág. 98]

Copia de objetos (I)

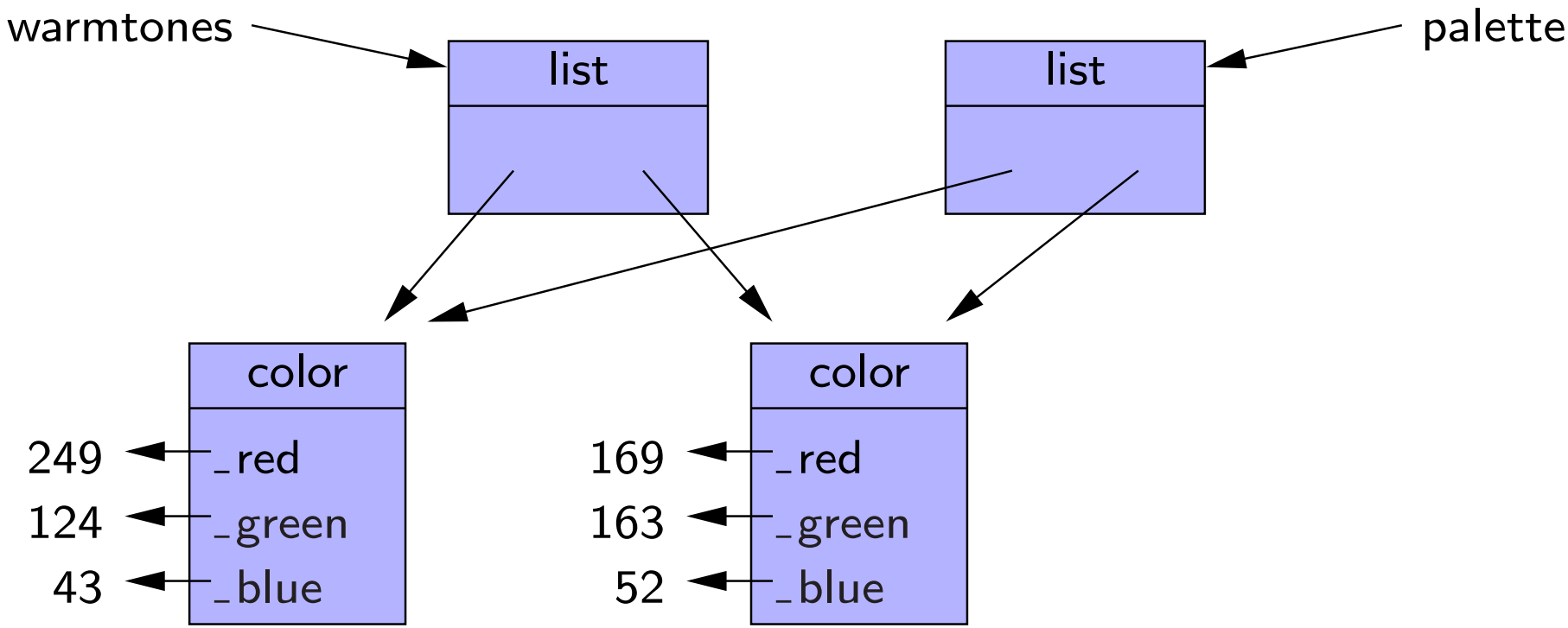
```
1 palette = warmtones # creamos un alias
```



[Goodrich et al., 2013, pág. 101]

Copia de objetos (II)

```
1 palette = list(warmtones) # copia superficial
```

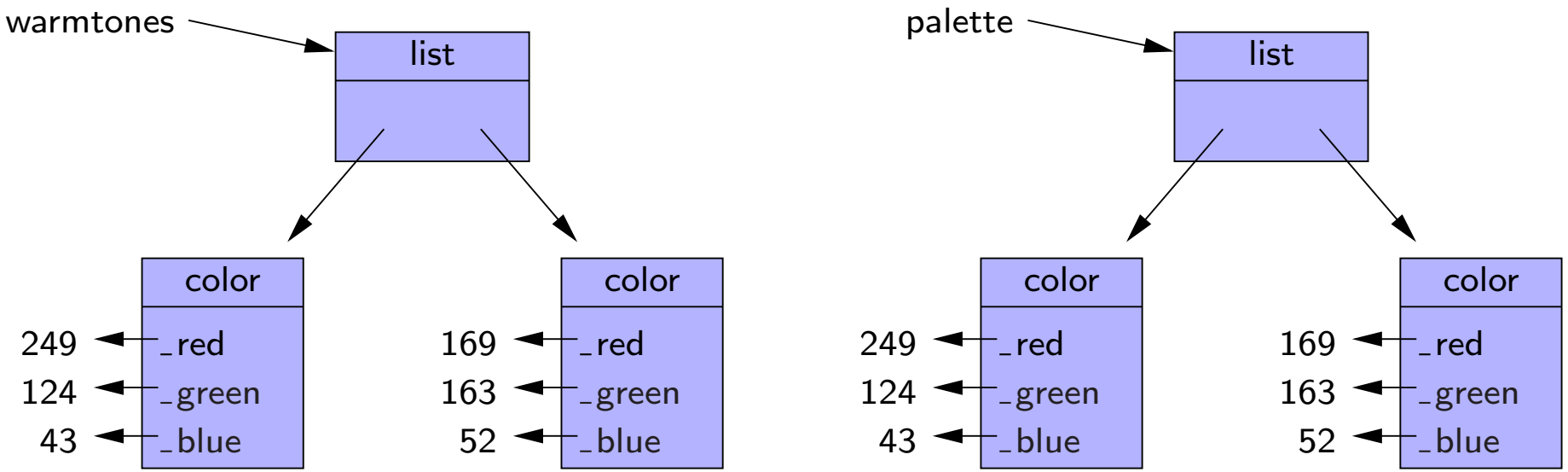


[Goodrich et al., 2013, pág. 101]

Copia de objetos (III)

Paquete *copy*:

```
1 palette = copy.copy(warmtones)      # copia superficial
2 palette = copy.deepcopy(warmtones) # copia profunda
```



Referencias (I)

[Goodrich et al., 2013] El capítulo 2 está dedicado a la orientación a objetos en Python.

[van Rossum et al., 2001] Guía de estilo de Python.


Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. (2013).
Data Structures and Algorithms in Python.
Wiley.
http://bcs.wiley.com/he-bcs/Books?action=index&bcsId=8029&itemId=1118290275.


van Rossum, G., Warsaw, B., and Coghlan, N. (2001).
Style guide for python code.
http://www.python.org/dev/peps/pep-0008/.