第六章　結構化資料型態

OOP with Ruby

本章內容

- 陣列型態
- 雜湊型態

陣列型態

- Arrays are indexed collections
  - Store collections of objects
  - Accessible using a integer-based key
- In Ruby, array grows as needed to hold new elements
- You can create and initialize a new array object using an *array literal*
  - A set of elements between square brackets (*nil* is an object)

```
a = [ 1, 'cat', 3.14 ]   # array with three elements
# access the first element
a[0]   →   1
# set the third element
a[2] = nil
# dump out the array
a      →   [1, "cat", nil]
```

如何建立陣列

- Use the special class method []

```
a = Array.[](1,2,3,4)

b = Array[1,2,3,4]

c = [1,2,3,4]
```

- Also, the class method new can take zero, one, or two parameters

```
d = Array.new            # Create an empty array

e = Array.new(3)          # [nil, nil, nil]

f = Array.new(3, "blah")   # ["blah", "blah", "blah"]
```

如何存取陣列

- Array element reference and assignment are done using the class methods [] and []=, respectively
  - Each can take an integer parameter, a pair of integers (start and length), or a range
  - A negative index counts backward from the end of the array, starting at -1
- Also, the special instance method at works like a simple case of element reference

如何存取陣列 (2)

```
a = [1, 2, 3, 4, 5, 6]
b = a[0]              # 1
c = a.at(0)           # 1
d = a[-2]             # 5
e = a.at(-2)          # 5
f = a[9]              # nil
g = a.at(9)           # nil
h = a[3,3]            # [4, 5, 6]
i = a[2..4]           # [3, 4, 5]
j = a[2...4]          # [3, 4]

a[1] = 8              # [1, 8, 3, 4, 5, 6]
a[1,3] = [10, 20, 30]   # [1, 10, 20, 30, 5, 6]
a[0..3] = [2, 4, 6, 8]  # [2, 4, 6, 8, 5, 6]
a[-1] = 12            # [2, 4, 6, 8, 5, 12]
```

修改陣列內容

- Array grows when needed

```
k = [2, 4, 6, 8, 10]
k[1..2] = [3, 3, 3]     # [2, 3, 3, 3, 8, 10]
k[7] = 99               # [2, 3, 3, 3, 8, 10, nil, 99]
```

- Array contains any kind of object, including another array

```
m = [1, 3, 5, 7, 9]
m[2] = [20, 30]         # [1, 3, [20, 30], 7, 9]
# On the other hand...
m = [1, 3, 5, 7, 9]
m[2..2] = [20, 30]      # [1, 3, 20, 30, 7, 9]
```

抽出部分陣列內容

- The method *indices* will take a list of indices (or *indexes*, an alias) and return an array consisting of only those elements

```
x = [10, 20, 30, 40, 50, 60]
y = x.indices(0, 1, 4)        # [10, 20, 50]
z = x.indexes(2, 10, 5, 4)    # [30, nil, 60, 50]
```

刪除陣列內容

- If you want to delete one specific element by index, use *delete_at* method

```
a = [10, 12, 14, 16, 18]

a.delete_at(3)          # Returns 16

# a is now [10, 12, 14, 18]

a.delete_at(9)          # Returns nil (out of range)
```

- If you want to delete all instances of a certain piece of data, *delete* method will do the job

```
b = %w(spam spam bacon spam eggs ham spam)

b.delete("spam")          # Returns "spam"

# b is now ["bacon", "eggs", "ham"]

b.delete("caviar")        # Returns nil
```

刪除陣列內容 (2)

- The *slice!* method accesses the same elements as *slice* but deletes them from the array as it returns their values

```
x = [0, 2, 4, 6, 8, 10, 12, 14, 16]

a = x.slice!(2)                    # 4

# x is now [0, 2, 6, 8, 10, 12, 14, 16]

b = x.slice!(2,3)                  # [6, 8, 10]

# x is now [0, 2, 12, 14, 16]

c = x.slice!(2..3)                 # [12, 14]

# x is now [0, 2, 16]
```

刪除陣列內容 (3)

- The *shift* and *pop* methods can be used for deleting array elements

```
x = [1, 2, 3, 4, 5]
x.pop                  # Delete the last element
# x is now [1, 2, 3, 4]
x.shift                # Delete the first element
# x is now [2, 3, 4]
```

- The *clear* method will delete all the elements in an array
  - It is equivalent to assigning an empty array to the variable, but it's marginally more efficient.

```
x = [1, 2, 3]
x.clear
# x is now []
```

列舉陣列內容

- The Array class has the standard iterator *each*
  - The reverse_each method will iterate in reverse order

```
friends = ["Melissa", "Jeff", "Ashley", "Rob"]
friends.each do |friend|
    puts "I have a friend called " + friend
end

accumulator = []
[1, 2, 3].reverse_each { |x| accumulator << x + 1 }
accumulator              # => [4, 3, 2]
```

陣列長度

- The method *length* (or its alias *size*) will give the number of elements in an array

```
x = ["a", "b", "c", "d"]
a = x.length          # 4
b = x.size            # 4
```

- The method *nitems* is the same except that it does not count nil elements

```
y = [1, 2, nil, nil, 3, 4]
c = y.size            # 6
d = y.length          # 6
e = y.nitems          # 4
```

陣列排序

- The easiest way to sort an array is to use the built-in *sort* method

```
words = [the, quick, brown, fox]

list = words.sort  # ["brown", "fox", "quick", "the"]

# Or sort in place:

words.sort!        # ["brown", "fox", "quick", "the"]
```

- The sort method assumes that all the elements in the array are comparable with each other
  - A mixed array, such as [1, 2, "three", 4], will normally give a type error

多維陣列

- Ruby doesn't provide explicit support for multi-dimensional arrays
  - You must in essence create an "array of arrays" in which each element of the array holds yet another array

多維陣列範例

```
class Array3

  def initialize
     @store = [[[]]]
  end

  def [](a,b,c)
    if @store[a]==nil ||
       @store[a][b]==nil ||
       @store[a][b][c]==nil
      return nil
    else
      return @store[a][b][c]
    end
  end

  def []=(a,b,c,x)
    @store[a] = [[]] if @store[a]==nil
    @store[a][b] = [] if @store[a][b]==nil
    @store[a][b][c] = x
  end

end

x = Array3.new
x[0,0,0] = 5
x[0,0,1] = 6
x[1,2,3] = 99
```

雜湊型態

- Hashes are known in some circles as *associative arrays*, *dictionaries*, and by various other names
- Hashes are indexed collections
  - Store collections of objects
  - Accessible using an object key
- You may think of a hash as an array with a specialized index, or as a database "synonym table" with two fields, stored in memory

如何建立雜湊

- As with Array, the special class method [] is used to create a hash

```
a1 = Hash.[]("flat",3,"curved",2)

a2 = Hash.[]("flat"=>3,"curved"=>2)

b1 = Hash["flat",3,"curved",2]

b2 = Hash["flat"=>3,"curved"=>2]

c1 = { "flat",3,"curved",2}

c2 = { "flat"=>3,"curved"=>2}

# For a1, b1, and c1: There must be an even number of elements.
```

如何建立雜湊 (2)

- Also, the class method new can take a parameter specifying a default value
  - The default value is not actually part of the hash; it is simply a value returned in place of nil

```
d = Hash.new          # Create an empty hash
e = Hash.new(99)      # Create an empty hash
f = Hash.new("a"=>3) # Create an empty hash
e["angled"]           # 99
e.inspect             # { }
f["b"]                # { "a"=>3}  (default value is
                      #   actually a hash itself)
f.inspect             # { }
```

雜湊的預設值

- The default value of a hash is an object that is referenced in place of nil in the case of a missing key
  - This is useful if you plan to use methods with the hash value that are not defined for *nil*
  - It can be assigned upon creation of the hash or at a later time using the *default*= method

```
a = Hash.new("missing")  # default value object is "missing"
a["hello"]               # "missing"
a.default="nothing"
a["hello"]               # "nothing"
a["good"] << "bye"       # "nothingbye"
a.default                # "nothingbye"
```

雜湊的存取

- The special instance method *fetch* raises an *IndexError* exception if the key does not exist in the Hash object
  - It takes a second parameter that serves as a default value

```
a = { "flat",3,"curved",2,"angled",5}
a.fetch("pointed")                  # IndexError
a.fetch("curved","na")              # 2
a.fetch("x","na")                   # "na"
```

新增雜湊內容

- Hash has class methods [] and []=, just as Array has
  - But they accept only one parameter

```
a = { }
a["flat"] = 3        # { "flat"=>3}
a.[]=("curved",2)    # { "flat"=>3,"curved"=>2}
a.store("angled",5)  # { "flat"=>3,"curved"=>2,"angled"=>5}
```

  - The method store is simply an alias for the []= method
  - The method fetch is similar to the [] method, except that it raises an IndexError for missing keys

刪除雜湊內容

- Use clear to remove all key/value pairs
  - This is essentially the same as assigning a new empty hash, but it's marginally faster
- Use shift to remove an unspecified key/value pair
  - This method returns the pair as a two-element array (or nil if no keys are left)

```
a = { 1=>2, 3=>4}

b = a.shift       # [1,2]

# a is now { 3=>4}
```

刪除雜湊內容

- Use delete to remove a specific key/value pair
  - It accepts a key and returns the value associated with the key removed (if found)
  - If the key is not found, the default value is returned

```
a = { 1=>1, 2=>4, 3=>9, 4=>16}

a.delete(3)                 # 9

# a is now { 1=>1, 2=>4, 4=>16}

a.delete(5)                 # nil in this case
```

列舉雜湊內容

- The Hash class has the standard iterator each

```
{ "a"=>3,"b"=>2} .each do |key, val|
  print val, " from ", key, "; "    # 3 from a; 2 from b;
end
```

- The Hash also has *each_key, each_pair*, and *each_value* (*each_pair* is an alias for *each*)

```
{ "a"=>3,"b"=>2} .each_key do |key|
  print "key = #{ key} ;"      # Prints: key = a; key = b;
end

{ "a"=>3,"b"=>2} .each_value do |value|
  print "val = #{ value} ;"    # Prints: val = 3; val = 2;
end
```

偵測雜湊內容

- Determining whether a key has been assigned can be done with *has_key?* or any one of its aliases: *include?, key?, or member?*

```
a = { "a"=>1,"b"=>2}
a.has_key? "c"        # false
a.include? "a"        # true
a.key? 2              # false
a.member? "b"         # true
```

- It is possible to test for the existence of an associated value using *has_value?* or *value?*

```
a.has_value? 2        # true
a.value? 99           # false
```

偵測雜湊內容 (2)

- Use *empty?* to see whether there are any keys at all left in the hash
- Use *length* or its alias *size* can be used to determine how many there are

```
a = { "a"=>1,"b"=>2}

a.empty?      # false

a.length      # 2
```

本章回顧