从python转到Scala简明教程[译]

原作者项目地址 PythonToScala

这个教程主要是通过 `python` 和 `scala` 的对比来了解 `scala` 语言, 给从前学 `python` 的朋友转向 `scala` 的一个引导

这个教程借鉴了很多 Scala for theImpatient, 的内容

需要注意的是这个教程主要从 `python` 的思维方式来对比 `scala` 但是我们不能用写 `python` 的方式来写 `scala` ,要了解更多的 `scala` 细节,用写 `scala` 的方式写 `scala` 多阅读Twitter 的文档Effective Scala

推荐按下面顺序阅读章节内容:

1. Variables and Arithmetic
2. Conditionals
3. Functions
4. Strings
5. Sequences
6. Maps
7. Tuples
8. Exceptions
9. Classes

下面有一些比较好和比较重要的介绍 `scala` 思想的文章推荐延伸阅读:

- **Pattern Matching** This is like a switch statement on turbo, and is very powerful and oft used. The Scala Cookbook has really great practical examples.
- **Auxiliary Constructors** Classes can have multiple constructors that operate on different argument types/number of args.

- **Case Classes** as an immutable, record-like data-structure that can be pattern-matched.
- **Scala Collections** There is a lot of power in all of the methods available to data structures like Vector, Array, List, Sequence, Set, etc. Just take a look at all of the available methods.

英文 gitbook 地址 GitBook

联系方式:

> 邮箱 `yishenggudou@me.com`
>
> 微博 `http://weibo.com/zhanghaibo/`

Variables变量申明

简单过下,`Scala`和`python`不同的地方是`Scala`有可变类型和不可变类型

1. `val`ues (immutable) 不可变
2. `var`iables (mutable) 可变

Python:

```
>>> foo = "Apples"
>>> baz = foo + " and Oranges."
>>> baz
'Apples and Oranges.'
>>> baz = "Only Grapes."
```

Scala:

```
scala> val foo = "Apples"
foo: String = Apples

scala> val baz = foo + " and Oranges."
baz: String = Apples and Oranges.


scala> baz
res60: String = Apples and Oranges.


// In Scala, vals are immutable
scala> baz = "Only Grapes."
<console>:13: error: reassignment to val
       baz = "Only Grapes."


// Create a var instead
scala> var baz = "Apples and Oranges."
baz: String = Apples and Oranges.

scala> baz = "Only Grapes."
baz: String = Only Grapes.

scala> var one = 1
one: Int = 1

scala> one += 1

scala> one
res21: Int = 2
```

`Scala`中我们可以使用强类型申明来代替编译器自己判断:

Scala

```
scala> val foo: String = "Apples"
foo: String = Apples
```

Python和 Scala 都允许多重引用但是还是有一些区别 Python and Scala will both let you perform multiple assignment. However, be careful with Python

and pass by reference! You'll usually want to unpack rather than perform multiple assignment.

Scala:

```
scala> val foo, bar = Array(1, 2, 3)
foo: Array[Int] = Array(1, 2, 3)
bar: Array[Int] = Array(1, 2, 3)

// foo and bar reference different pieces of memory; changing one
scala> bar(0) = 4

scala> bar
res70: Array[Int] = Array(4, 2, 3)

scala> foo
res71: Array[Int] = Array(1, 2, 3)
```

在python中等号前后直接解包匹配

```
>>> foo, bar = [1, 2, 3], [1, 2, 3]
# Are they referencing the same memory?
>>> foo is bar
False
# What happens when you change bar?
>>> bar[0] = 4
>>> bar
[4, 2, 3]
>>> foo
[1, 2, 3]


# You *can* assign both foo and bar the same value, but they refe
>>> foo = bar = [1, 2, 3]
>>> foo is bar
True
>>> bar[0] = 4
>>> bar
[4, 2, 3]
>>> foo
[4, 2, 3]
```

Scala和 Python 的操作符直接有区别

1. `python` 中 `a + b` 实际上是 `a.__add__(b)`
2. `scala` 中 `a + b` 实际上上 `a.+(b)`

也就是说在`scala` 中 `+` 实际上是一个方法名字

Python

```
>>> foo = 1
# What's happening behind the scenes?
>>> foo.__add__(4)
5
```

Scala

```
scala> val foo = 1
foo: Int = 1

scala> foo + 1
res72: Int = 2
// What's happening behind the scenes:
scala> foo.+(1)
res73: Int = 2
```

Conditional Expressions条件表达式

`Scala` 的 `if/else/else-if` 是 c 语言风格的,但是也和 python 有很多相似 的地方

Python

```
>>> x = 0
# Inline: expression_if_true if condition else expression_if_fals
>>> foo = 1 if x > 0 else -1
>>> foo
-1
# Expressions broken across multiple lines
if x == 1:
    foo = 5
elif x == 0:
    foo = 6
>>> foo
6
>>>
if isinstance('foo', str):
    print('Foo is string')
else:
    print('Foo is not string')

Foo is string
```

Scala

```
scala> val x = 0
x: Int = 0

// Inline: variable assignment expression
scala> val foo = if (x > 0) 1 else -1
foo: Int = -1

scala> var baz = 1
baz: Int = 1

// REPL paste mode
scala> :paste
// Entering paste mode (ctrl-D to finish)
// Kernighan & Ritchie brace style preferred for multi-line expre
if (x == 0) {
  baz = 5
}
// Exiting paste mode, now interpreting.

scala> baz
res90: Int = 5
// But for simple expressions, try to keep them to one line
scala> if (x == 0) baz = 6

scala> baz
res94: Int = 6

scala>

if (foo.isInstanceOf[String]) {
    print("Foo is a string!")
} else if (foo.isInstanceOf[Int]) {
    print("Foo is an int!")
} else {
    print("I dont know what foo is...")
}

Foo is a string!
```

While loops 看起来差不多:

Python

```
n = 0
nlist = []
while n < 5:
    nlist.append(n)
    n += 1
>>> nlist
[0, 1, 2, 3, 4]
```

下面是 scala 版本

Scala

```
n: Int = 0

// ArrayBuffer is a mutable array that acts like Python lists.
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer

scala> var nlist = ArrayBuffer[Int]()
nlist: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

scala>

while (n < 5) {
    nlist += n
    n += 1
}

scala> nlist
res115: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(0
}
```

Scala 的 `For` 循环写法和 python 一样飘逸,但是没有了强制缩进的问题我们终于 可以把 `for` 写在一行之内搞定啦.

Python

```
foo = "Apple"
n = 0
for a in foo:
    n += 1
>>> n
5
```

Scala

```
scala> val foo = "Apple"

scala> var n = 0

scala>
for (x <- foo) {
    n += 1
}

scala> n
res140: n: Int = 5

// This would actually be better expressed in a single line
scala> n = 0

scala> for (x <- foo) n += 1

scala> n
res141: n: Int = 5
```

python 的列表推导式,在 scala 里面可以用 yield 实现,

Python

```
>>> [f + 1 for f in [1, 2, 3, 4, 5]]
[2, 3, 4, 5, 6]
```

Scala

```
scala> for (f <- Array(1, 2, 3, 4, 5)) yield f + 1
res59: Array[Int] = Array(2, 3, 4, 5, 6)
```

python的`itertools`这个库里面有很多实用的函数比如`zip`这个我们看看scala 里面怎么用:

Python

```
foo, bar = [1, 2, 3], ['a', 'b', 'c']
foobars = {}
for f, b in zip(foo, bar):
    foobars[b] = f
>>> foobars
{'a': 1, 'c': 3, 'b': 2}

# It's more Pythonic to use a comprehension
>>> {b: f for f, b in zip(foo, bar)}
{'a': 1, 'c': 3, 'b': 2}
```

Scala

```
val foo = Array(1, 2, 3)
val bar = Array("a", "b", "c")

import scala.collection.mutable.Map
// Let's go ahead and specify the types, since we know them
var foobars= Map[String, Int]()

for (f <- foo; b <- bar) foobars += (b -> f)
scala> foobars
res5: scala.collection.mutable.Map[String,Int] = Map(b -> 3, a ->

// This is really powerful- we're not limited to two iterables
val baz = Array("apple", "orange", "banana")
val mapped = Map[String, (Int, String)]()
for (f <- foo; b <- bar; z <- baz) mapped += (z -> (f, b))
scala> mapped
res7: scala.collection.mutable.Map[String,(Int, String)] = Map(ba

// It's worth noting that Scala also has an explicit zip method
val arr1 = Array(1, 2, 3)
val arr2 = Array(4, 5, 6)

scala> arr1.zip(arr2)
res240: Array[(Int, Int)] = Array((1,4), (2,5), (3,6))
```

Python' 的 `enumerate` 有时候挺好用的, 在 `Scala` 里面可以用 `zipWithIndex`：

Python

```
>>> [(x, y) for x, y in enumerate(["foo", "bar", "baz"])]
[(0, 'foo'), (1, 'bar'), (2, 'baz')]
```

Scala

```
scala> for ((y, x) <- Array("foo", "bar", "baz").zipWithIndex) yi
res27: Array[(Int, String)] = Array((0,foo), (1,bar), (2,baz))

// Note that simply calling zipWithIndex will return something si
scala> Array("foo", "bar", "baz").zipWithIndex
res31: Array[(String, Int)] = Array((foo,0), (bar,1), (baz,2))
```

Scala 中也可以实现列表推导中的条件判断

Python

```
foo = [1,2,3,4,5]
bar = [x for x in foo if x != 3]
>>> bar
[1, 2, 4, 5]
```

Scala

```
val foo = Array(1, 2, 3, 4, 5)
var bar = ArrayBuffer[Int]()
for (f <- foo if f != 3) bar += f
scala> bar
res136: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1

// You can stack guards
scala> for (x <- (1 to 5).toArray if x != 2 if x != 3) yield x
res44: Array[Int] = Array(1, 4, 5)
```

函数式编程使用 `map` 函数隐式循环在某些场景下不错

Scala

```
scala> for (c <- Array(1, 2, 3)) yield c + 2
res56: Array[Int] = Array(3, 4, 5)

scala> Array(1, 2, 3).map(_ + 2)
res57: Array[Int] = Array(3, 4, 5)
```

Functions函数

Disclaimer: in the following section I use "function" to refer to both Scala functions (defined with `=>`) and Scala methods (defined with `def`) somewhat interchangeably.

这里说的函数包括`=>`申明和`def`申明的函数

Scala 的函数部分除了在申明函数的时候需要指定参数类型之外,其他的和 python 出入不大.

Python

```
def concat_num_str(x, y):
    if not isinstance(x, str):
        x = str(x)
    return x + y
>>> concat_num_str(1, "string")
'1string'
```

Scala 的函数在没有显式return的情况下会吧最后一个求值的表达式的值作为函数的 返回值:

Scala:

```
def concat_num_str(x:Int, y:String) = x.toString + y
scala> concat_num_str(1, "string")
res146: String = 1string

// What happens if we try to pass the incorrect type?
scala> concat_num_str("string", num)
<console>:19: error: type mismatch;
 found   : String("string")
 required: Int
              concat_num_str("string", num)
                             ^
<console>:19: error: not found: value num
              concat_num_str("string", num)
```

如果多行的函数需要用大括号包起来,这点是和 python 不一样,python:

Scala:

```
import scala.collection.mutable.Map
val str_arr = Array("apple", "orange", "grape")

def len_to_map(arr:Array[String]) = {
    var lenmap:Map[String, Int] = Map()
    for (a <- arr) lenmap += (a -> a.length)
    lenmap
}

scala> len_to_map(str_arr)
res149: scala.collection.mutable.Map[String,Int] = Map(orange ->
```

With Scala, you can specify a return type, and *have* to do so in recursive funcs:

Scala:

```
def factorial(n: Int): Int = if (n <= 0) 1 else n * factorial(n -
scala> factorial(5)
res152: Int = 120

// It's nice to specify the return type as a matter of habit
scala> def spec_type(x: Int, y:Double): Int = x + y.toInt
spec_type: (x: Int, y: Double)Int

scala> spec_type(1, 3.4)
res33: Int = 4
```

函数传参数,参数可以有预值:

Python

```
def make_arr(x, y, fruit="apple", drink="water"):
    return [x, y, fruit, drink]
>>> make_arr("orange", "banana")
['orange', 'banana', 'apple', 'water']
>>> make_arr("orange", "banana", "melon")
['orange', 'banana', 'melon', 'water']
>>> make_arr("orange", "banana", drink="coffee")
['orange', 'banana', 'apple', 'coffee']
```

Scala

```
def make_arr(x:String, y:String, fruit:String = "apple", drink:St

scala> make_arr("orange", "banana")
res154: Array[String] = Array(orange, banana, apple, water)

scala> make_arr("orange", "banana", "melon")
res155: Array[String] = Array(orange, banana, melon, water)

scala> make_arr("orange", "banana", drink="coffee")
res156: Array[String] = Array(orange, banana, apple, coffee)

// As with Python, non-named parameters need to be supplied befor
scala> make_arr("orange", drink="coffee", "banana")
<console>:19: error: not enough arguments for method make_arr: (x
Unspecified value parameter y.
              make_arr("orange", drink="coffee", "banana")
```

`*args` 方式传参, `Scala` 也是支持的

Python:

```
def sum_args(*args):
    return sum(args)

>>> sum_args(1, 2, 3, 4, 5)
15
```

Scala:

```
def sum_args(args:Int*) = args.sum
scala> sum_args(1, 2, 3, 4, 5)
res159: Int = 15
```

在 python 里面需要先解包

Python:

```
>>> sum_args([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sum_args
TypeError: unsupported operand type(s) for +: 'int' and 'list'

>>> sum_args(*[1, 2, 3])
6
```

Scala:

```
scala> sum_args(Array(1, 2, 3))
<console>:17: error: type mismatch;
 found   : Array[Int]
 required: Int
              sum_args(Array(1, 2, 3))

scala> sum_args(Array(1, 2, 3): _*)
res161: Int = 6
```

有些函数没有返回值的时候不能

Scala:

```
def proc_func(x:String, y:String) {print(x + y)}
proc_func("x", "y")
```

Scala supports anonymous functions the same way that Python's `lamda` functions work:

Python:

```
>>> concat_fruit = lambda x, y: x + y
>>> concat_fruit('apple', 'orange')
'appleorange'
```

Scala:

```
scala> val concat_fruit = (x: String, y: String) => x + y
concat_fruit: (String, String) => String = <function2>

scala> concat_fruit("apple", "orange")
res4: String = appleorange
```

函数一等公民,函数可以作为参数传给其他参数

Python:

```
def apply_to_args(func, arg1, arg2):
    return func(arg1, arg2)
>>> apply_to_args(concat_fruit, 'apple', 'orange')
'appleorange'
```

Scala:

```
scala> def applyToArgs(func: (String, String) => String, arg1: St
applyToArgs: (func: (String, String) => String, arg1: String, arg

scala> applyToArgs(concat_fruit, "apple", "banana")
res9: String = applebanana

// You can apply anonymous functions as well
scala> def applySingleArgFunc(func: (Int) => Int, arg1: Int): Int
applySingleArgFunc: (func: Int => Int, arg1: Int)Int

scala> applySingleArgFunc((x: Int) => x + 5, 1)
res11: Int = 6
```

函数可以作为函数返回值

Python:

```
def concat_curry(fruit):
  def perf_concat(veg):
      return fruit + veg
  return perf_concat

>>> curried = concat_curry('apple')
>>> curried('spinach')
'applespinach'
>>> curried('carrot')
'applecarrot'
```

Scala:

```
scala> def concat_curried(fruit: String)(veg: String): String = f
concat_curried: (fruit: String)(veg: String)String

scala> val curried = concat_curried("apple") _
curried: String => String = <function1>

scala> curried("spinach")
res14: String = applespinach

scala> curried("carrot")
res15: String = applecarrot
```

Strings字符串

Scala 的字符串和 python 的字符串用法差不多

Python

```
>>> foo = "bar"
>>> foo[1]
'a'
>>> len(foo)
3
>>> foo + str(1)
'bar1'
>>> foo.split("a")
['b', 'r']
>>> "foo   ".strip()
'foo'
>>> "String here: {}, Int here: {}".format("foo", 1)
'String here: foo, Int here: 1'
>>> foo.upper()
'BAR'
```

Scala

```
scala> val foo = "bar"
foo: String = bar

scala> foo(1)
res19: Char = a

scala> foo.length
res2: Int = 3

scala> foo + 1.toString
res3: String = bar1

scala> foo.split("a")
res10: Array[String] = Array(b, r)

scala> "fooo   ".trim
res11: String = fooo

scala> "String here: %s, Int here: %d".format("foo", 1)
res13: String = String here: foo, Int here: 1

// Scala lacks some of the convenience operators, but makes it ea
// Note the use of _ here to indicate the each variable passing i
scala> foo.map(_.toUpper)
res18: String = BAR
```

Scala 2.10 支持牛逼的字符串格式化

Scala

```
scala> val fruit = "apple"
fruit: String = apple

scala> val apple_count = 5
apple_count: Int = 5

scala> val veg = "broccoli"
veg: String = broccoli

scala> val broc_count = 10
broc_count: Int = 10

scala> s"I have $apple_count $fruit and $broc_count $veg. In tota
res15: String = I have 5 apple and 10 broccoli. In total I have 1
```

字符串遍历都 easy

Python

```
for c in foo:
    print(c)
b
a
r
```

Scala

```
scala> for (c <- foo) println(c)
b
a
r
```

多行字符串的什么方式一致

Python

```
>>> """
... This is
... a multiline string
... """
'\nThis is\na multiline string\n'
```

Scala

```
scala> """
     | This is
     | a multiline string
     | """
res8: String =
"
This is
a multiline string
"
```

Sequences序列

Scala 的序列类型比 python 的多

1. List (linked-lists)
2. Vectors (immutable arrays)
3. Arrays (mutable arrays of fixed length)
4. ArrayBuffers(mutable arrays of varying length)

需要根据场景选择自己合适的数据结构 Vector 是 Scala中最好的不可变序列结构

Scala:

```
scala> val vector = Vector(1, 2, 3)
vector: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> vector.map(_ + 2)
res30: scala.collection.immutable.Vector[Int] = Vector(3, 4, 5)

scala> vector.count(_ == 2)
res31: Int = 1

scala> Vector(1, 2, 3, 4, 5).drop(3)
res33: scala.collection.immutable.Vector[Int] = Vector(4, 5)

scala> vector.exists(_ == 4)
res35: Boolean = false

scala> vector.take(2)
res8: scala.collection.immutable.Vector[Int] = Vector(1, 2)

scala> Vector(1, 20, 3, 25).reduce(_ + _)
res4: Int = 49

scala> Vector(3, 4, 4, 5, 3).distinct
res5: scala.collection.immutable.Vector[Int] = Vector(3, 4, 5)

scala> Vector(1, 20, 3, 25).partition(_ > 10)
res2: (scala.collection.immutable.Vector[Int], scala.collection.i

scala> Vector("foo", "bar", "baz", "foo", "bar").groupBy(_ == "fo
```

```
res0: scala.collection.immutable.Map[Boolean,scala.collection.imm

scala> Vector(1, 20, 3, 25).groupBy(_ > 10)
res1: scala.collection.immutable.Map[Boolean,scala.collection.imm

scala> vector.filter(_ != 2)
res9: scala.collection.immutable.Vector[Int] = Vector(1, 3)

scala> vector.max
res11: Int = 3

scala> Vector(3, 2, 1).sorted
res17: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> Vector("fo", "fooooo", "foo", "fooo").sortWith(_.length >
res17: scala.collection.immutable.Vector[String] = Vector(fooooo,

scala> vector.sum
res15: Int = 6

scala> Vector(Vector(1, 2, 3), Vector(4, 5, 6)).flatten
res28: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4

// Another way to go about the previous operation
scala> Vector.concat(Vector(1, 2, 3), Vector(4, 5, 6))
res16: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4

scala> Vector(1, 2, 3).intersect(Vector(3, 4, 5))
res7: scala.collection.immutable.Vector[Int] = Vector(3)

scala> Vector(1, 2, 3).diff(Vector(3, 4, 5))
res8: scala.collection.immutable.Vector[Int] = Vector(1, 2)
```

数组的长度在开始的时候确定

Python:

```
# These are contrived- you will rarely see the need for this in P
ten_zeroes = [0]*10
ten_none = [None]*10
```

Scala

```
scala> val init_int = new Array[Int](10)
init_int: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

scala> val init_str = new Array[String](10)
init_str: Array[String] = Array(null, null, null, null, null, nul

scala> Array.tabulate(3)(a => a + 5)
res20: Array[Int] = Array(5, 6, 7)

scala> Array.tabulate(3)(a => a * 5)
res21: Array[Int] = Array(0, 5, 10)

scala> Array.fill(3)(10)
res22: Array[Int] = Array(10, 10, 10)
```

ArrayBuffer 跟 python 的 最像

Python:

```
int_list = []
int_list.append(1)
int_list.extend([2, 3, 4])
int_list.extend((5, 6, 7))
>>> int_list
[1, 2, 3, 4, 5, 6, 7]
>>> int_list.count(1)
>> # Closest thing to Scala trimEnd
>>> int_list = int_list[0:-5]
>>> int_list
[1, 2]
>>> int_list.insert(1, 5)
>>> int_list
[1, 5, 2]
>>> int_list.pop(1)
5
>>> int_list
[1, 2]
>>> int_list.reverse()
>>> int_list
[2, 1]
>>> int_list.sort()
>>> int_list
[1, 2]
>>> max(int_list)
2
>>> min(int_list)
1
>>> int_list = []
```

Scala:

```
import scala.collection.mutable.ArrayBuffer

val int_arr = ArrayBuffer[Int]()
int_arr += 1
int_arr += (2, 3, 4)
int_arr ++= Array(5, 6, 7)

res182: int_arr.type = ArrayBuffer(1, 2, 3, 4, 5, 6, 7)

scala> int_arr.count(_ == 1)
res187: Int = 1
```

```
res187: Int = 1

scala> int_arr.trimEnd(5)

scala> int_arr
res192: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1

scala> int_arr.insert(1, 5)

scala> int_arr
res195: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1

scala> int_arr.remove(1)
res196: Int = 5

scala> int_arr
res197: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1

// Scala will also let you be more flexible with multiple insert/
scala> int_arr.insert(1, 5, 6)

scala> int_arr
res199: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1

scala> int_arr.remove(1, 2)

scala> int_arr
res201: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1

scala> int_arr.reverse
res205: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(2

scala> int_arr.max
res210: Int = 2

scala> int_arr.min
res211: Int = 1

scala> int_arr.reverse.sorted
res215: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1

scala> int_arr.clear

scala> int_arr
res29: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()
```

序列循环

Python:

```
foo = [f for f in range(0, 10, 1)]
>>> foo
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
foo = [f for f in range(0, 10, 2)]
>>> foo
[0, 2, 4, 6, 8]
```

Scala:

```
scala> val foo = for (x <- 0 until 10) yield x
foo: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 2,

scala> val foo = for (x <- 0 until (10, 2)) yield x
foo: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 2, 4,

// Scala also has a "to" operator that creates an inclusive range
scala> 0 to (10, 2)
res22: scala.collection.immutable.Range.Inclusive = Range(0, 2, 4

scala> 0 until (10, 2)
res23: scala.collection.immutable.Range = Range(0, 2, 4, 6, 8)
```

列表推导式

Python:

```
foo = [x + "qux" for x in ["foo", "bar", "baz"] if x != "foo"]
>>> foo
['barqux', 'bazqux']
```

Scala:

```
scala> val foo = for (x <- Vector("foo", "bar", "baz") if x != "f
foo: scala.collection.immutable.Vector[String] = Vector(barqux, b

// Note that the comprehension returns the type that is fed to it
scala> val foo = for (x <- ArrayBuffer("foo", "bar", "baz") if x
foo: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(b

// Note that we could use a more functional approach to operate o
scala> Vector("foo", "bar", "baz").filter(_ != "foo").map(_ + "qu
res25: scala.collection.immutable.Vector[String] = Vector(barqux,
```

多维数组,python部分参见 numpy

Scala:

```
scala> val multdim = Array.ofDim[Int](3, 4)
multdim: Array[Array[Int]] = Array(Array(0, 0, 0, 0), Array(0, 0,

scala> multdim(0)(2) = 15

scala> multdim
res217: Array[Array[Int]] = Array(Array(0, 0, 15, 0), Array(0, 0,
```

Maps

Scala 比 Python 更详细的定义有可变和不可变 Map

Python:

```
fruit_count = {}
fruit_count = {"apples": 4, "oranges": 5, "bananas": 6}
>>> fruit_count["apples"]
4
>>> fruit_count.has_key("apples")
True
>>> fruit_count.get("melons", "peaches")
'peaches'
>>> fruit_count["melons"] = 2
>>> fruit_count.update({"peaches": 8, "pears": 4})
>>> fruit_count
{'peaches': 8, 'melons': 2, 'pears': 4, 'apples': 4, 'oranges': 5
>>> fruit_count.pop("apples")
4
>>> fruit_count
{'peaches': 8, 'melons': 2, 'pears': 4, 'oranges': 5, 'bananas':
>>> fruit_count.keys()
['peaches', 'melons', 'pears', 'oranges', 'bananas']
>>> fruit_count.values()
[8, 2, 4, 5, 6]
```

Scala:

```
scala> val imm_fruit_count = Map("apples" -> 4, "oranges" -> 5, "
imm_fruit_count: scala.collection.mutable.Map[String,Int] = Map(b

scala> imm_fruit_count("apples")
res219: Int = 4

scala> imm_fruit_count.contains("apples")
res220: Boolean = true

scala> imm_fruit_count.getOrElse("melons", "peaches")
res221: Any = peaches

scala> val mut_fruit_count = scala.collection.mutable.Map[String,
mut_fruit_count: scala.collection.mutable.Map[String,Int] = Map()

scala> mut_fruit_count("apples") = 4

scala> mut_fruit_count += ("oranges" -> 5, "bananas" -> 6)
res223: mut_fruit_count.type = Map(bananas -> 6, oranges -> 5, ap

scala> mut_fruit_count
res224: scala.collection.mutable.Map[String,Int] = Map(bananas ->

scala> mut_fruit_count -= "apples"
res225: mut_fruit_count.type = Map(bananas -> 6, oranges -> 5)

scala> mut_fruit_count.keySet
res228: scala.collection.Set[String] = Set(bananas, oranges)

scala> mut_fruit_count.values
res229: Iterable[Int] = HashMap(6, 5)

// This is a nice feature of Scala Maps:
scala> val defaultMap = Map("foo" -> 1, "bar" -> 2).withDefaultVa
defaultMap: scala.collection.immutable.Map[String,Int] = Map(foo

scala> defaultMap("qux")
res31: Int = 3
```

Scala will let you iterate over maps in a similar way as Python:

Python

```
>>> foo = [k + str(v) for k, v in fruit_count.items()]
>>> foo
['peaches8', 'melons2', 'pears4', 'oranges5', 'bananas6']
```

Scala

```
scala> val foo = ArrayBuffer[String]()
foo: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer()

scala> for ((k, v) <- mut_fruit_count) foo += k.toString + v.toSt

scala> foo
res237: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffe
```

Scala 有一个基于叔的排序 Map 对应 Python 的OrderedDict

Python:

```
import collections
foo = collections.OrderedDict(sorted({"apples": 4, "oranges": 5}.
>>> foo
OrderedDict([('apples', 4), ('oranges', 5)])
```

Scala:

```
// PSA this is immutable
scala> val scores = scala.collection.immutable.SortedMap("oranges
scores: scala.collection.immutable.SortedMap[String,Int] = Map(ap
```

Tuples元组

在 Scala 中元组可以是混合类型的,像 python一样在某些场合下面很好使用

Python:

```
>>> foo = (1, 2.5, "three")
>>> a, b, c = foo
>>> a
1
>>> b
2.5
>>> c
'three'
>>> bar = (4, 5.5, "six")
>>> foobar = ((x, y) for x, y in zip(foo, bar))
>>> foobar
<generator object <genexpr> at 0x10ac21f50>
>>> foobar = tuple(((x, y) for x, y in zip(foo, bar)))
>>> foobar
((1, 4), (2.5, 5.5), ('three', 'six'))
```

Scala:

```
scala> val foo = (1, 2.5, "three")
foo: (Int, Double, String) = (1,2.5,three)

// Accessors are named by position
scala> foo._1
res17: Int = 1

scala> foo._2
res18: Double = 2.5

// Destructuring

scala> val (a, b, c) = foo
a: Int = 1
b: Double = 2.5
c: String = three

scala> val bar = (4, 5.5, "six")
bar: (Int, Double, String) = (4,5.5,six)

// We saw the zip function earlier - it produces a tuple
scala> val pairs = Array(1, 2, 3).zip(Array("four", "five", "six"
pairs: Array[(Int, String)] = Array((1,four), (2,five), (3,six))

scala> for ((k, v) <- pairs) yield k.toString + v
res243: Array[String] = Array(1four, 2five, 3six)
```

Exceptions异常

scala 的异常比 python 简单些.

Python:

```
def is_apple(fruit):
    if fruit != "apple":
        raise ValueError("Fruit is not apple!")

>>> is_apple("orange")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in is_apple
ValueError: Fruit is not apple!
```

Scala:

```
def is_apple(fruit:String) = {
    if (fruit != "apple") throw new IllegalArgumentException("Fru
}

scala> is_apple("orange")
java.lang.IllegalArgumentException: Fruit is not apple!
  at .is_apple(<console>:16)
  ... 33 elided
```

Scala 也可以 `try/catch/finally` 类似 python 的 `try/except/finally`:

Python:

```
def check_fruit(fruit):
    try:
        is_apple(fruit)
        print('No exception raised...')
    except IOError:
        print("Oh no! IOError!")
    except ValueError as e:
        print(e.message)
    finally:
        print('This will excecute regardless of path.')

>>> check_fruit("apple")
No exception raised...
This will excecute regardless of path.
>>> check_fruit("orange")
Fruit is not apple!
This will excecute regardless of path.
```

Scala:

```
import java.io.IOException
def check_fruit(fruit:String) = {
  try {
    is_apple(fruit)
    print("No exception raised...")
  } catch {
    case ex: IllegalArgumentException => print(ex)
    case _: IOException => print("Oh no! IOException!")
  } finally {
    print("This will execute regardless of case.")
  }
}

scala> check_fruit("apple")
No exception raised...This will execute regardless of case.
scala> check_fruit("orange")
java.lang.IllegalArgumentException: Fruit is not apple!This will
```

Classes类

构建 Scala 的 class 比 Java 简单,总的来说还是比较像 Java

Python

```
class Automobile(object):

    def __init__(self, wheels=4, engine=1, lights=2):
        self.wheels = wheels
        self.engine = engine
        self.lights = lights

    def total_parts(self):
        return self.wheels + self.engine + self.lights

    def remove_wheels(self, count):
        if (self.wheels - count) < 0:
            raise ValueError('Automobile cannot have fewer than 0
        else:
            self.wheels = self.wheels - count
            print('The automobile now has {} wheels!').format(sel


>>> car = Automobile()
>>> car.wheels
3
>>> car.total_parts()
7
>>> car.wheels = 6
>>> car.total_parts()
9
>>> car.remove_wheels(7)
---------------------------------------------------------------
ValueError                                Traceback (most recent
<ipython-input-30-24207883207a> in <module>()
----> 1 car.remove_wheels(7)

<ipython-input-27-41c5871dc8ce> in remove_wheels(self, count)
     11     def remove_wheels(self, count):
     12         if (self.wheels - count) < 0:
---> 13             raise ValueError('Automobile cannot have fewe
     14         else:
```

```
      15              self.wheels = self.wheels - count

ValueError: Automobile cannot have fewer than 0 wheels!

>>> car.remove_wheels(2)
The automobile now has 4 wheels!
```

In Scala, use var to make all attributes mutable. Behind the scenes, Scala is creating getters and setters for each:

Scala

```
class Automobile(var wheels:Int = 4, var engine:Int = 1, var ligh

    def total_parts() = {
        // No "self" needed, and implicit return
        wheels + engine + lights
    }

    // Purely side-effecting function, no "=" needed
    def remove_wheels(count:Int) {
        if (wheels - count < 0) {
            throw new IllegalArgumentException("Automobile cannot
        } else {
            wheels = wheels - count
            println(s"Auto now has $wheels wheels!")
        }
    }
}

scala> val car = new Automobile()
car: Automobile = Automobile@77424e9

scala> car.wheels
res64: Int = 4

scala> car.total_parts()
res65: Int = 7

scala> car.wheels = 6
car.wheels: Int = 6

scala> car.total_parts()
res66: Int = 9

scala> car.remove_wheels(7)
java.lang.IllegalArgumentException: Automobile cannot have fewer
  at Automobile.remove_wheels(<console>:19)
  ... 32 elided

scala> car.remove_wheels(2)
Auto now has 4 wheels!
```

可变和不可变 In Scala, if you define a constructor field as val (immutable),

you get a getter but not a setter. This is roughly comparable to defining your own property getter/setters on a Python class (for a great rundown of Python properties and descriptors, check out Chris Beaumont's Python Descriptors Demystified):

Python

```
class Automobile(object):

    def __init__(self, wheels=4):
        # The underscore is convention, but not enforced
        self._wheels = wheels

    @property
    def wheels(self):
        return self._wheels

    @wheels.setter
    def wheels(self, count):
        raise ValueError('This value is immutable!')

>>> car = Automobile()
>>> car.wheels
4
>>> car.wheels = 5
---------------------------------------------------------------------
ValueError                                Traceback (most recent
<ipython-input-26-f07fada773fb> in <module>()
----> 1 car.wheels = 5

<ipython-input-23-87368b68789b> in wheels(self, count)
     11     @wheels.setter
     12     def wheels(self, count):
---> 13         raise ValueError('This value is immutable!')

ValueError: This value is immutable!
```

Scala

```
class Automobile(val wheels:Int = 4){}

scala> val car = new Automobile()
car: Automobile = Automobile@64284ba3

scala> car.wheels
res67: Int = 4

scala> car.wheels = 5
<console>:12: error: reassignment to val
       car.wheels = 5
                  ^
```

Like Java, Scala also allows you to define fields as `private`, which prevents getters/setters from being generated and only allows the field to be accessed within the class. This behavior can be replicated in Python, but you won't often see this pattern actually being used in Python programs- it is understood that class attributes leading with an underscore are "private" and should not be used:

Python

```
class Automobile(object):

    def __init__(self, wheels=4):
        # The underscore is convention, but not enforced
        self._wheels = wheels

    @property
    def wheels(self):
        raise ValueError('You cannot access this value!')

    @wheels.setter
    def wheels(self, count):
        raise ValueError('You cannot access this value!')

>>> car.wheels
---------------------------------------------------------------------
ValueError                                Traceback (most recent
<ipython-input-41-d267b674bbda> in <module>()
----> 1 car.wheels

<ipython-input-39-73ff0f17068d> in wheels(self)
      7     @property
      8     def wheels(self):
----> 9         raise ValueError('You cannot access this value!')
     10
     11     @wheels.setter

ValueError: You cannot access this value!

>>> car.wheels = 5
---------------------------------------------------------------------
ValueError                                Traceback (most recent
<ipython-input-42-f07fada773fb> in <module>()
----> 1 car.wheels = 5

<ipython-input-39-73ff0f17068d> in wheels(self, count)
     11     @wheels.setter
     12     def wheels(self, count):
---> 13         raise ValueError('You cannot access this value!')

ValueError: You cannot access this value!
```

Scala

```
class Automobile(private val wheels:Int = 4){}

scala> car.wheels
<console>:11: error: value wheels in class Automobile cannot be a
              car.wheels
                  ^

scala> car.wheels = 5
<console>:13: error: value wheels in class Automobile cannot be a
val $ires4 = car.wheels
```

Staticmethods in Scala are handled via "companion objects" for classes, which are named the same as the class itself. Objects are an entire topic of study for the Scala language- I am just touching on them as they related to Python's `staticmethod`, but I recommend that you do some reading on how objects are used in Scala. Also demonstrated in this example is how object fields can be used to mirror Python's class-level attributes:

Python

```
class Automobile(object):

    wheels = 4
    lights = 2

    def __init__(self, name):
        self.name = name

    @staticmethod
    def print_uninst_str():
        print("No 'self' passed to this method, and no instantiat
              " It's static!")

>>> Automobile.print_uninst_str()
No 'self' passed to this method, and no instantiation. It's stati
>>> Automobile.wheels
4
>>> Automobile.wheels = 5
>>> Automobile.wheels
5
```

Scala

```
class Automobile(var name:String){}

object Automobile {
    var wheels = 4
    var lights = 2
    def print_uninst_str() = "No 'self' passed to this method, an
}

scala> Automobile.print_uninst_str
res3: String = No 'self' passed to this method, and no instantiat

scala> Automobile.wheels
res4: Int = 4

scala> Automobile.wheels = 5
Automobile.wheels: Int = 5

scala> Automobile.wheels
res5: Int = 5
```

The following section is going to be a *very* light treatment of inheritance in Scala and Python. I recommend reading more on both languages regarding abstract base classes and traits (Scala) and Method Resolution Order (Python). For now, here are the basics.

Scala supports single inheritance via abstract base classes, which are explicitly named as such. Python can treat any class as an abstract one:

Python

```
class Automobile(object):

    wheels = 4
    lights = 2
    doors = 2

    def __init__(self, color, make):
        self.color = color
        self.make = make

    def towing_capacity(self):
        pass

    def top_speed(self):
        pass

    def print_make_color(self):
        print(" ".join([self.color, self.make]))

class Car(Automobile):

    doors = 4

    def __init__(self, color, model):
        super(Car, self).__init__(color, model)

    def towing_capacity(self):
        return 0

    def top_speed(self):
        return 150


>>> mycar = Car("red", "Toyota")
>>> mycar.doors
4
>>> mycar.towing_capacity()
0
>>> mycar.top_speed()
150
>>> mycar.print_make_color()
Red Toyota
```

Scala

```
abstract class Automobile(val color:String, val make:String) {
    val wheels:Int = 4
    val lights:Int = 2
    val doors:Int = 4

    def towing_capacity: Int
    def top_speed: Int
    def print_make_color():String = return s"$color $make"
}

class Car(color:String, make:String) extends Automobile(color, ma

    override val doors = 4 // Override needed if immutable "val"

    def towing_capacity() = 0
    def top_speed() = 150
}

scala> val mycar = new Car("Red", "Toyota")
mycar: Car = Car@351cdd99

scala> mycar.print_make_color()
res2: String = Red Toyota

scala> mycar.doors
res3: Int = 4

scala> mycar.top_speed
res4: Int = 150

// Abstract classes only support single inheritance!
scala> abstract class SportPackage {}
defined class SportPackage

scala> class Car(color:String, make:String) extends Automobile(co
<console>:9: error: class SportPackage needs to be a trait to be
       class Car(color:String, make:String) extends Automobile(co
```

Generally, you should only use abstract base classes in Scala if you need Java interop or need to pass constructor parameters to the base class.

Otherwise, you should use what Scala calls "Traits", which act like class mixins and enable multiple inheritance in Scala.

Python

```
class Engine(object):

    started = False

    def start(self):
        self.started = True

    def shutdown(self):
        self.started = False

class Transmission(object):

    fluid_level = 0

    def add_fluid(self, amount):
        self.fluid_level = self.fluid_level + amount

# Using Automobile class from earlier
class Car(Automobile, Engine, Transmission):

    doors = 4

    def __init__(self, color, model):
        super(Car, self).__init__(color, model)

    def towing_capacity(self):
        return 0

    def top_speed(self):
        return 150

>>> mycar = Car("Red", "Toyota")

>>> mycar.start()

>>> mycar.started
True
```

```
>>> mycar.fluid_level
0

>>> mycar.add_fluid(50)

>>> mycar.fluid_level
50
```

Scala

```
trait Engine {
    var started:Boolean = false
    // These don't return anything, so no "=" needed
    def start() {started = true}
    def shutdown() {started = false}
}

trait Transmission {
    var fluid_level:Int = 0
    def add_fluid(amount:Int) {fluid_level = fluid_level + amount
}

class Car(color:String, make:String) extends Automobile(color, ma
  with Engine
  with Transmission {

    override val doors = 4 // Override needed if immutable "val"

    def towing_capacity() = 0
    def top_speed() = 150
}

scala> val mycar = new Car("Red", "Toyota")
mycar: Car = Car@38134991

scala> mycar.start()

scala> mycar.started
res2: Boolean = true

scala> mycar.add_fluid(50)

scala> mycar.fluid_level
res4: Int = 50
```

目录