介绍

2017 年 9 月 1 日

1 目标

- 回顾计算机科学的思想, 提高编程和解决问题的能力。
- 理解抽象化以及它在解决问题过程中发挥的作用
- 理解和实现抽象数据类型的概念
- 回顾 Python 编程语言

2 快速开始

从第一台通过接入网线和交换机来传递人的指令的计算机开始，我们编程思考的方式发生了许多变化。与社会的许多方面一样，计算技术的变化为计算机科学家提供了越来越多的工具和平台来实践他们的工艺。计算机的快速发展诸如更快的处理器，高速网络和大的存储器容量已经让计算机科学家陷入高度复杂螺旋中。在所有这些快速演变中，一些基本原则保持不变。计算机科学关注用计算机来解决问题。毫无疑问你花了相当多的时间学习解决问题的基础知识，以此希望有足够的能力把问题弄清楚并想出解决方案。你还发现编写代码通常很困难。问题的复杂性和解决方案的相应复杂性往往会掩盖与解决问题过程相关的基本思想。本章着重介绍了其他两个重要的部分。首先回顾了计算机科学与算法和研究数据结构所必须适应的框架，特别是我们需要研究这些主题的原因，以及如何理解这些主题有助于我们更好的解决问题。第二，我们回顾 Python 编程语言。虽然我们不提供详尽的参考，我们将在其余章节中给出基本数据结构的示例和解释。

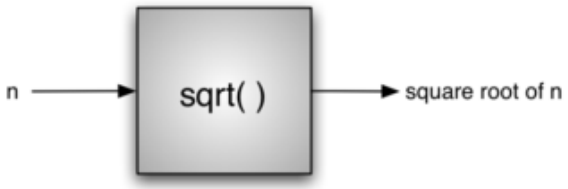
3 什么是计算机科学

计算机科学往往难以定义。这可能是由于在名称中不幸使用了“计算机”一词。正如你可能知道的，计算机科学不仅仅是计算机的研究。虽然计算机作为一个工具在学科中发挥重要的支持作用，但它们只是工具。计算机科学是对问题，解决问题以及解决问题过程中产生的解决方案的研究。给定一个问题，计算机科学家的目标是开发一个算法，一系列的指令列表，用于解决可能出现的问题的任何实例。算法遵循它有限的过程就可以解决问题。计算机科学可以被认为是对算法的研究。但是，我们必须谨慎地包括一些事实，即一些问题可能没有解决方案。虽然证明这种说法正确性超出了本文的范围，但一些问题不能解决的事实对于那些研究计算机科学的人是很重要的。所以我们可以这么定义计算机科学，是研究能被解决的问题的方案和不能被解决问题的科学。通常我们会说这个问题是可计算的，当在描述问题和解决方案时。如果存在一个算法解决这个问题，那么问题是可计算的。计算机科学的另一个定义是说，计算机科学是研究那些可计算和不可计算的问题，研究是不是存在一种算法来解决它。你会注意到，“电脑”一词根本没有出现。解决方案是独立于机器而言的。计算机科学，因为它涉及问题解决过程本身，也是抽象的研究。抽象使我们能够以分离所谓的逻辑和物理角度的方式来观察问题和解决方案。基本思想跟我们常见

的例子一样。假设你可能已经开车上学或上班。作为司机，汽车的用户。你为了让汽车载你到目的地，你会和汽车有些互动。进入汽车，插入钥匙，点火，换挡，制动，加速和转向。从抽象的角度，我们可以说你所看到的是汽车的逻辑视角。你正在使用汽车设计师提供的功能，将你从一个地方运输到另一个位置。这些功能有时也被称为接口。另一方面，修理汽车的技工有一个截然不同的视角。他不仅知道如何开车，还必须知道所有必要的细节，使我们认为理所当然的功能运行起来。他需要了解发动机是如何工作的，变速箱如何变速，温度是如何控制的等等。这被称为物理视角，细节发生在“引擎盖下”。当我们使用电脑时也会发生同样的情况。大多数人使用计算机写文档，发送和接收电子邮件，上网冲浪，播放音乐，存储图像和玩游戏，而不知道让这些应用程序工作的细节。他们从逻辑或用户角度看计算机。计算机科学家，程序员，技术支持人员和系统管理员看计算机的角度截然不同。他们必须知道操作系统如何工作的细节，如何配置网络协议，以及如何编写控制功能的各种脚本。他们必须能够控制底层的细节。这两个示例的共同点是用户态的抽象，有时也称为客户端，不需要知道细节，只要用户知道接口的工作方式。这个接口是用户与底层沟通的方式。作为抽象的另一个例子，Python 数学模块。一旦我们导入模块，我们可以执行计算

```
>>> import math
>>> math.sqrt(16)
4.0
>>>
```

这是一个程序抽象的例子。我们不一定知道如何计算平方根，但我们知道函数是什么以及如何使用它。如果我们正确地执行导入，我们可以假设该函数将为我们提供正确的结果。我们知道有人实现了平方根问题的解决方案，但我们只需要知道如何使用它。这有时被称为“黑盒子”视图。我们简单地描述下接口：函数的名称，需要什么（参数），以及将返回什么。细节隐藏在里面



4 什么是编程

编程是将算法编码为符号，编程语言的过程，以使得其可以由计算机执行。虽然有许多编程语言和不同类型的计算机存在，第一步是需要有解决方案。没有算法就没有程序。

计算机科学不是研究编程。然而，编程是计算机科学家的一个重要能力。编程通常是我们为解决方案创建的表现形式。因此，这种语言表现形式和创造它的过程成为该学科的基本部分。

算法描述了依据问题实例数据所产生的解决方案和产生预期结果所需的一套步骤。编程语言必须提供一种表示方法来表示过程和数据。为此，它提供了控制结构和数据类型。

控制结构允许以方便而明确的方式表示算法步骤。至少，算法需要执行顺序处理，决策选择和重复控制迭代。只要语言提供这些基本语句，它就可以用于算法表示。

计算机中的所有数据项都以二进制形式表示。为了赋给这些字符串含义，我们需要有数据类型。数据类型提供了对这个二进制数据的解释，以便我们能够根据解决的问题思考数据。这些底层的内置数据类型（有时称为原始数据类型）为算法开发提供了基础。

例如，大多数编程语言为整数提供数据类型。内存中的二进制数据可以解释为整数，并且能给予一个我们通常与整数（例如 23,654 和 -19）相关联的含义。此外，数据类型还提供数据项参与的操作的描述。对于整数，诸如加法，减法和乘法的操作是常见的。我们期望数值类型的数据可以参与这些算术运算。通

常我们遇到的困难是问题及其解决方案非常复杂。这些简单的，语言提供的结构和数据类型虽然足以表示复杂的解决方案，但通常在我们处理问题的过程中处于不利地位。我们需要一些方法控制这种复杂性，并能给我们提供更好的解决方案。

5 为什么要学习数据结构和抽象数据类型

为了管理问题的复杂性和解决问题的过程，计算机科学家使用抽象使他们能够专注于“大局”而不会迷失在细节中。通过创建问题域的模型，我们能够利用更好和更有效的问题解决过程。这些模型允许我们以更加一致的方式描述我们的算法将要处理的数据。之前，我们将过程抽象称为隐藏特定函数的细节的过程，以允许用户或客户端在高层查看它。我们现在将注意力转向类似的思想，即数据抽象的思想。抽象数据类型（有时缩写为 ADT ）是对我们如何查看数据和允许的操作的逻辑描述，而不用考虑如何实现它们。这意味着我们只关心数据表示什么，而不关心它最终将如何构造。通过提供这种级别的抽象，我们围绕数据创建一个封装。通过封装实现细节，我们将它们从用户的视图中隐藏。这称为信息隐藏。

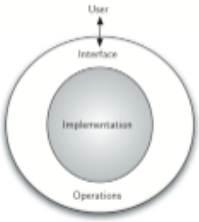


Figure 2 展示了抽象数据类型是什么以及如何操作。用户与接口交互，使用抽象数据类型指定的操作。抽象数据类型是用户与之交互的 shell。实现隐藏在更深的底层。用户不关心实现的细节。

抽象数据类型（通常称为数据结构）的实现将要求我们使用一些程序构建和原始数据类型的集合来提供数据的物理视图。正如我们前面讨论的，这两个视角的分离将允许我们将问题定义复杂的数据模型，而不给出关于模型如何实际构建的细节。这提供了独立于实现的数据视图。由于通常有许多不同的方法来实现抽象数据类型，所以这种实现独立性允许程序员在不改变数据的用户与其交互的方式的情况下切换实现的细节。用户可以继续专注于解决问题的过程。

6 为什么要学习算法

计算机科学家经常通过经验学习。我们通过看别人解决问题和自己解决问题来学习。接触不同的问题解决技术，看不同的算法设计有助于我们承担下一个具有挑战性的问题。通过思考许多不同的算法，我们可以开始开发模式识别，以便下一次出现类似的问题时，我们能够更好地解决它。算法通常彼此完全不同。考虑前面看到的 sqrt 的例子。完全可能的是，存在许多不同的方式来实现细节以计算平方根函数。一种算法可以使用比另一种更少的资源。一个算法可能需要 10 倍的时间来返回结果。我们想要一些方法来比较这两个解决方案。即使他们都工作，一个可能比另一个“更好”。我们建议使用一个更高效，或者一个只是工作更快或使用更少的内存的算法。当我们研究算法时，我们可以学习分析技术，允许我们仅仅根据自己的特征而不是用于实现它们的程序或计算机的特征来比较和对比解决方案。在最坏的情况下，我们可能有一个难以处理的问题，这意味着没有算法可以在实际的时间量内解决问题。重要的是能够区分具有解决方案的那些问题，不具有解决方案的那些问题，以及存在解决方案但需要太多时间或其他资源来合理工作的那些问题。经常需要权衡，我们需要做决定。作为计算机科学家，除了我们解决问题的能力，我们还需要了解解决方案评估技术。最后，通常有很多方法来解决问题。找到一个解决方案，我们将一遍又一遍比较，然后决定它是否是一个好的方案。

7 Python 介绍

Python 是一种高级的、动态的多范型编程语言。Python 代码很多时候看起来就像是伪代码一样，因此你可以使用很少的几行可读性很高的代码来实现一个非常强大的想法。举个例子，下面是使用 Python 来实现非常经典的快速排序算法的代码。

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
# Prints "[1, 1, 2, 3, 6, 8, 10]"
```

7.1 Python versions

目前有两种不同的 Python 版本被支持，分别是 2.7 和 3.4。令人不解的是，Python 3.0 引入了一些不可向下兼容的变换，所以使用 2.7 写的代码不一定能在 3.4 版本上运行，反之亦然。

你可以在命令行中使用一下命令来检查一下自己的 Python 版本:

```
python --version
```

7.2 基本数据类型

和很多语言一样，Python 也有几种数据类型包括：整型、浮点型、布尔型和字符型。这些数据类型的表现方式和大家熟悉的其他语言一样。

7.2.1 Numbers

整型和浮点型的工作方式和其他语言一样。

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"
x += 1
print(x)  # Prints "4"
x *= 2
print(x)  # Prints "8"
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

注意：Python 没有类似于++和--的一元运算的操作。Python 也内置了长整型和复数。

7.2.2 Booleans

Python 的布尔型逻辑和其他语言都一样，不过使用了英文单词替换了符号 (&&, ||, etc.):

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
```

7.2.3 Strings

Python 对字符型的支持非常强大。

```
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.
print(hello)       # Prints "hello"
print(len(hello))  # String length; prints "5"
hw = hello + ' ' + world  # String concatenation
print(hw)  # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string formatting
print(hw12)  # prints "hello world 12"
```

String 对象还有很多常用的方法，例如:

```
s = "hello"
print(s.capitalize())  # Capitalize a string; prints "Hello"
print(s.upper())       # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding with spaces; prints "
hello"
print(s.center(7))     # Center a string, padding with spaces; prints " hello "
print(s.replace('l', '(ell)'))  # Replace all instances of one substring with
another;
                               # prints "he(ell)(ell)o"
print('  world '.strip())  # Strip leading and trailing whitespace; prints "world"
```

7.3 容器（Containers）

Python 包含了几个内置的容器类型：列表 (lists)、字典 (dictionaries)、集合 (sets)、元组 (tuples)

7.3.1 列表（list）

list 在 Python 中几乎等价于数组，但是是可调整大小的同时也可以包容不同类型的元素。

```
xs = [3, 1, 2]    # Create a list
print(xs, xs[2])  # Prints "[3, 1, 2] 2"
print(xs[-1])     # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'     # Lists can contain elements of different types
print(xs)         # Prints "[3, 1, 'foo']"
xs.append('bar')  # Add a new element to the end of the list
```

```
print(xs)          # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()       # Remove and return the last element of the list
print(x, xs)       # Prints "bar [3, 1, 'foo']"
```

切片（Slicing）：除了每次访问一个列表元素，Python 还提供了一种简洁的语法去访问子列表，这被称为 slicing:

```
nums = list(range(5))     # range is a built-in function that creates a list of
integers
print(nums)               # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])          # Get a slice from index 2 to 4 (exclusive); prints "[2,
3]"
print(nums[2:])           # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])           # Get a slice from the start to index 2 (exclusive);
prints "[0, 1]"
print(nums[:])            # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])          # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]        # Assign a new sublist to a slice
print(nums)               # Prints "[0, 1, 8, 9, 4]"
```

我们将会在 numpy arrays 章节中再看到 slicing.

Loops：你可以用如下方法遍历列表元素：

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
# Prints "cat", "dog", "monkey", each on its own line.
```

如果你想使用循环结构得到元素的索引以及内容，使用内置的 “enumerate” 方法

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

列表推导：在编程的时候我们经常会想要将一种数据类型转换成另一种。举个简单的例子，思考下面这段代码，它计算了数据的平方：

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)   # Prints [0, 1, 4, 9, 16]
```

你也可以使用列表推导写出更简单的代码：

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)   # Prints [0, 1, 4, 9, 16]
```

列表推导还可以包含判断逻辑：

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
```

```
print(even_squares)  # Prints "[0, 4, 16]"
```

7.3.2 字典（Dictionaries）

字典由键值对组成。就像 JAVA 里面的 map，以及 JavaScript 中的 Object。使用方法如下：

```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['cat'])       # Get an entry from a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary has a given key; prints "True"
d['fish'] = 'wet'     # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
# print(d['monkey'])  # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
del d['fish']         # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

Loops：根据字典的键很容易进行迭代。

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

如果你想获得键以及对应的值，使用 items 方法

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

字典推导（Dictionary comprehensions）：和推导列表差不多，不过可以更简单的构建字典。

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)  # Prints "{0: 0, 2: 4, 4: 16}"
```

7.3.3 集合（Sets）

sets 是离散元的无序集合。下面举个简单的例子

```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints "True"
print('fish' in animals)  # prints "False"
animals.add('fish')       # Add an element to a set
print('fish' in animals)  # Prints "True"
print(len(animals))       # Number of elements in a set; prints "3"
animals.add('cat')        # Adding an element that is already in the set does
nothing
print(len(animals))       # Prints "3"
animals.remove('cat')     # Remove an element from a set
print(len(animals))       # Prints "2"
```

Loops：迭代集合的语法和迭代列表的一样，但是当集合是无序的时候，在便利集合中的元素时不要有排序的预期。

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

集合推导（Set comprehensions）：和列表以及字典一样。我们可以很简单的构建一个集合：

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)  # Prints "{0, 1, 2, 3, 4, 5}"
```

7.3.4 元组（Tuples）

元组是一个有序列表。在很多方面和列表一样，有一点非常重要的不同就是元组可以在字典中被用作键而列表不行。例子：

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)        # Create a tuple
print(type(t))    # Prints "<class 'tuple'>"
print(d[t])       # Prints "5"
print(d[(1, 2)])  # Prints "1"
```

7.3.5 函数

python 中函数的定义使用 def 关键字，如：

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
# Prints "negative", "zero", "positive"
```

我们将会用到带有可选参数的方法，例如：

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True)  # Prints "HELLO, FRED!"
```

7.3.6 类

定义类的语法在 Python 中很简单：

```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

8 Numpy

Numpy 是 Python 中科学计算的核心库。它提供一个高性能多维数据对象，以及操作这个对象的工具。如果你已经熟悉了 MATLAB ，你会发现本教程对于 numpy 起步很有用。

8.1 数组

Numpy 数组可以理解为一个矩阵，所有元素的类型都是一样的，是一个被索引的非负整数的元组。数组的维度大小是数组的 rank ，数组的 shape 是一个整型的元组，给出每个维度数组的大小。

我们可以使用嵌套 Python 列表来初始化一个 numpy 数组，使用方括号来访问元素。

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Numpy 还提供很多方法来创建数组：

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
```

```
print(a)                # Prints "[[ 0.  0.]
                        #          [ 0.  0.]]"

b = np.ones((1,2))      # Create an array of all ones
print(b)                # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)   # Create a constant array
print(c)                # Prints "[[ 7.  7.]
                        #          [ 7.  7.]]"

d = np.eye(2)           # Create a 2x2 identity matrix
print(d)                # Prints "[[ 1.  0.]
                        #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

8.2 数组索引

Numpy 提供了多种方法来索引数组。

切片：和 Python 的列表一样 numpy 数组也是可以被切片的。因为数组可能是多维的，所以对数组的每一个元素都要指定切片：

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

你也可以混合整型索引和切片索引。然而这样做将会得到一个相比原始数组 rank 更低的数组。注意这和 MATLAB 处理数组切片的方法有很大的不同。

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
```

```
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #         [ 6]
                             #         [10]] (3, 1)"
```

整型数组索引：当你使用切片来索引 numpy 数组时，结果的数组视图总是源数组的子数组。相比之下，整型数组索引允许您使用其他数组的数据构建任意数组。例如:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

整型数组索引的一个实用技巧是用来选择或变换矩阵的每一行的一个元素。

```
import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
```

```
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

布尔数组索引：布尔数组索引可以取得一个数组中的任意元素。通常这种索引方式用来选择符合一定条件的元素。例如：

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

8.3 数据类型

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```
import numpy as np
```

```
x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

8.4 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

注意：不同于 MATLAB，* 是逐点乘法，而不是矩阵乘法。我们使用 dot 函数计算向量的的内积、将一个向量乘以一个矩阵、矩阵乘以矩阵。dot 可作为 numpy 模块的一个函数，也可以作为数组对象的一个实例方法：

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Numpy 提供很多有用的函数来实现数组的运算；其中一个最有用的就是 sum：

```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

除了利用数组计算数学函数，我们还经常需要对数组进行 reshape 或处理其中的数据。关于这类操作最简单的例子就是矩阵的转置；去转置一个矩阵，只需使用数组对象的 T 属性：

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

8.5 广播机制（Broadcasting）

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

广播是一种强大的机制，它允许 numpy 在实现算术操作时去处理不同不同 shape 的数组。假设有一个小数组和一个大数组，我们想多次使用小数组去对大数组做某些操作。例如，假设我们想对矩阵的每一行加上一个常向量，可以这么做：

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

可以这么做；然而当矩阵 x 很大时，在 Python 中计算一个显示的循环会很慢。注意对矩阵 x 的每一行加上向量 v 等价于通过按垂直方向复制 v 若干次而叠加成一个矩阵 vv，然后对 x 和 vv 进行逐元加法。我们可以这样实现这一过程：

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

Numpy 广播机制允许我们不需要真正地创建 v 的多重拷贝来执行这一计算。使用广播机制，我们可以这么做：

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

由于广播机制，即使 x 有 shape (4, 3), v 有 shape (3,)，行y = x + v 仍可工作；这就如同 v 有 shape (4, 3)，其中每一行为 v 的拷贝，求和按逐元进行。

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension
6. If this explanation does not make sense, try reading the explanation from the documentation or this explanation.

广播两个数组遵循以下原则：

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length. 当两个数组的 rank 不一致时，
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension
6. If this explanation does not make sense, try reading the explanation from the documentation or this explanation.

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

Here are some applications of broadcasting:

```
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)

# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))

# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

9 SciPy

Numpy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays. SciPy builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications.

The best way to get familiar with SciPy is to browse the documentation. We will highlight some parts of SciPy that you might find useful for this class.

9.1 Image operations

SciPy provides some basic functions to work with images. For example, it has functions to read images from disk into numpy arrays, to write numpy arrays to disk as images, and to resize images. Here is a simple example that showcases these functions:

```
from scipy.misc import imread, imsave, imresize

# Read an JPEG image into a numpy array
img = imread('assets/cat.jpg')
print(img.dtype, img.shape)  # Prints "uint8 (400, 248, 3)"

# We can tint the image by scaling each of the color channels
# by a different scalar constant. The image has shape (400, 248, 3);
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);
# numpy broadcasting means that this leaves the red channel unchanged,
# and multiplies the green and blue channels by 0.95 and 0.9
# respectively.
img_tinted = img * [1, 0.95, 0.9]

# Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))

# Write the tinted image back to disk
imsave('assets/cat_tinted.jpg', img_tinted)
```

9.2 MATLAB files

The functions scipy.io.loadmat and scipy.io.savemat allow you to read and write MATLAB files. You can read about them in the documentation.

9.3 Distance between points

SciPy defines some useful functions for computing distances between sets of points.

The function scipy.spatial.distance.pdist computes the distance between all pairs of points in a given set:

A similar function (scipy.spatial.distance.cdist) computes the distance between all pairs across two sets of points; you can read about it in the documentation.

```
import numpy as np
from scipy.spatial.distance import pdist, squareform
```

```
# Create the following array where each row is a point in 2D space:
# [[0 1]
#  [1 0]
#  [2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

# Compute the Euclidean distance between all rows of x.
# d[i, j] is the Euclidean distance between x[i, :] and x[j, :],
# and d is the following array:
# [[ 0.          1.41421356  2.23606798]
#  [ 1.41421356  0.          1.        ]
#  [ 2.23606798  1.          0.        ]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```

10 Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the matplotlib.pyplot module, which provides a plotting system similar to that of MATLAB.

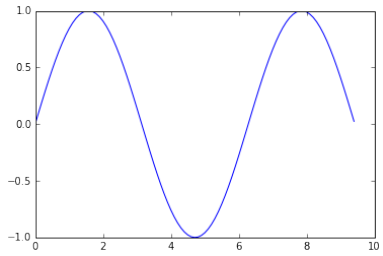
10.1 Plotting

The most important function in matplotlib is plot, which allows you to plot 2D data. Here is a simple example:

```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on a sine curve
```

```
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()  # You must call plt.show() to make graphics appear.
```

Running this code produces the following plot:



With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

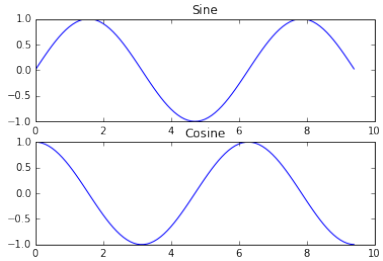
```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

10.2 Subplots

You can plot different things in the same figure using the subplot function. Here is an example:

```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
```

```
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



10.3 Images

You can use the imshow function to show images. Here is an example:

```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt

img = imread('assets/cat.jpg')
img_tinted = img * [1, 0.95, 0.9]

# Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)

# Show the tinted image
plt.subplot(1, 2, 2)

# A slight gotcha with imshow is that it might give strange results
# if presented with data that is not uint8. To work around this, we
# explicitly cast the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```


0
50
100
150
200
250
300
350
400
0
50
100
150
200
0
50
100
150
200
250
300
350
400
0
50
100
150
200