



南京邮电大学
Nanjing University of Posts and Telecommunications

Python程序设计 (混合式)



for



杨尚东

南京邮电大学计算机学院, 数据科学与工程系

shangdongyang.github.io

2024/10/24

交互模式vs脚本模式

□ 交互模式

- ✓ 直接在命令行输入python或python3, 每次回车之后运行一行代码

□ 脚本模式

- ✓ 编辑后缀为.py的文件
- ✓ 然后通过 python [文件名].py运行

□ Windows推荐用windows terminal

- ✓ MAC推荐用Iterm
- ✓ 自学一些简单的Linux shell命令
→ ls, pwd, cd, which, ps...

目录

- **基本语法**
 - **对象、类型、变量和赋值**
 - **运算符和表达式**
 - **字符串**
 - **流程控制**
- **组合数据类型**
- **函数**
- **异常处理和文件操作**
- **面向对象基础**
- **数值计算库NumPy**

Python语法基本特点

- 使用**缩进（四个空格）**来表示程序的层次结构。不同于C语言，**Python变量可不声明直接使用**
- **同一层次代码块，缩进所包含空格数量必须一致**

```
# 代码2.1 接收用户从键盘上输入的整数，并判别其奇偶性
num = int(input("请您输入一个整数: "))
if num%2 == 0:
    print(num, "是一个偶数")
    print("这里执行的是程序中的第 1 个分支")
else:
    print(num, "是一个奇数")
    print("这里执行的是程序中的第 2 个分支")
```

- 输入函数input(), 从键盘输入一个字符串
 - ✓ int()函数用于把字符串转换成整数
- 输出函数print(), 输出后自动换行

为什么Python使用缩进

- Guido van Rossum 认为使用缩进进行分组非常**优雅**，并且大大提高了普通Python程序的**清晰度**
- 取代了其他语言中的 {} 或关键词，**减少了冗余符号**，提高了代码**可读性和一致性**

```
if (x <= y)
    x++;
    y--;
z++;
```

如果条件为真，则只执行 `x++` 语句，但缩进会使你认为情况并非如此。即使是经验丰富的C程序员有时会长时间盯着它，想知道为什么即使 `x > y`，`y` 也在减少。

因为没有开始/结束括号，所以Python不太容易发生编码式冲突。在C中，括号可以放到许多不同的位置。如果您习惯于阅读和编写使用一种风格的代码，那么在阅读（或被要求编写）另一种风格时，您至少会感到有些不安。

许多编码风格将开始/结束括号单独放在一行上。这使得程序相当长，浪费了宝贵的屏幕空间，使得更难以对程序进行全面的了解。理想情况下，函数应该适合一个屏幕（例如，20--30行）。20行Python可以完成比20行C更多的工作。这不仅仅是由于缺少开始/结束括号 -- 缺少声明和高级数据类型也是其中的原因 -- 但缩进基于语法肯定有帮助。

Python对象及类型

□ Python 的世界里，一切皆**对象**

✓ 每个对象各包含一个 **identity**、**type** 和 **value**

记住该记住的，忘记该忘记的。
改变能改变的，接受不能改变的。
Remember what should be remembered,
and forget what should be forgotten. Alter
what is changeable, and accept what is
mutable.
——杰罗姆·大卫·塞林格 《麦田里的守望者》

```
>>> a = 30
>>> b = 30
```

```
>>> a is b
True
```

```
>>> a == b
True
```

```
>>> a = 300
>>> b = 300
```

```
>>> a is b
False
```

```
a = 300; b = 300
```

```
>>> a is b
True
```

Python对象及类型

□ 数据类型初探

```
>>> L = []
>>> dir(L)
['_add_', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
 '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']
>>> [d for d in dir(L) if '_' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 're

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

>>> L.append(1)
>>> L
[1]
```

`dir()` 函数用于获取一个对象的所有属性和方法的列表。它返回一个包含对象的属性（包括方法）的字符串列表。这个列表通常用于了解对象的可用功能，帮助你在编程中探索和使用它。

Python对象及类型

□ Python 所有**数据**都由**对象**或**对象间关系**来表示

□ Python3.x 中常见的内置标准类型

- ✓ 整型 (int)
- ✓ 浮点型 (float)
- ✓ 复数型 (complex)
- ✓ 布尔型 (bool)
- ✓ 字符串 (str)
- ✓ 元组 (tuple) 、不可变集合 (frozenset)
- ✓ 列表 (list) 、可变集合 (set) 、字典 (dict)

不可变类型

- 可 hash (不可变长度)
- 不支持新增
- 不支持删除
- 不支持修改
- 支持查询

可变类型

- 不可 hash (可变长度)
- 支持新增
- 支持删除
- 支持修改
- 支持查询

```
# 演示可变类型, list数据类型
L = ['Adam', 'Lisa', 'Bart']
for i in range(5):
    L.append('Bart' )
    print(L)
L.remove('Bart')
print(L, '\n')
```

```
# 演示不可变类型, tuple数据类型
T = ('Adam', 'Lisa', 'Bart')
for i in range(5):
    print(T.__add__('Bart2'))
    T = T + ('Bart' ,)
    print(T)
```


Python对象及类型

#代码2.2 在屏幕上输出各种对象的类型

```
print("100 is", type(100))
print("3.14 is", type(3.14))
print("5+2j is", type(5+2j))
print("True and False are", type(False))
print("\'I love Python.\' is", type("I love Python."))
print("[1,2,3] is", type ([1, 2, 3]))
print("(1,2,3) is", type ((1, 2, 3)))
print("{1,2,3} is", type ({1, 2, 3}))
print("frozenset({1,2,3}) is", type(frozenset({1, 2, 3})))
print("{'name':'Tom','age':18} is", type({'name':'Tom','age': 18}))
```

```
100 is <class 'int'>
3.14 is <class 'float'>
5+2j is <class 'complex'>
True and False are <class 'bool'>
'I love Python.' is <class 'str'>
[1,2,3] is <class 'list'>
(1,2,3) is <class 'tuple'>
{1,2,3} is <class 'set'>
frozenset({1,2,3}) is <class 'frozenset'>
{'name':'Tom','age':18} is <class 'dict'>
```

- ✓ **type()是Python的内置函数，返回对象所属类型**

Python变量和关键字

□ 标识符的命名规则

- ✓ 由大写和小写字母、下划线 _ 以及数字组成
- ✓ 不可以数字打头，可以包含Unicode字符（中文）
- ✓ 不可以和Python中的关键字重复

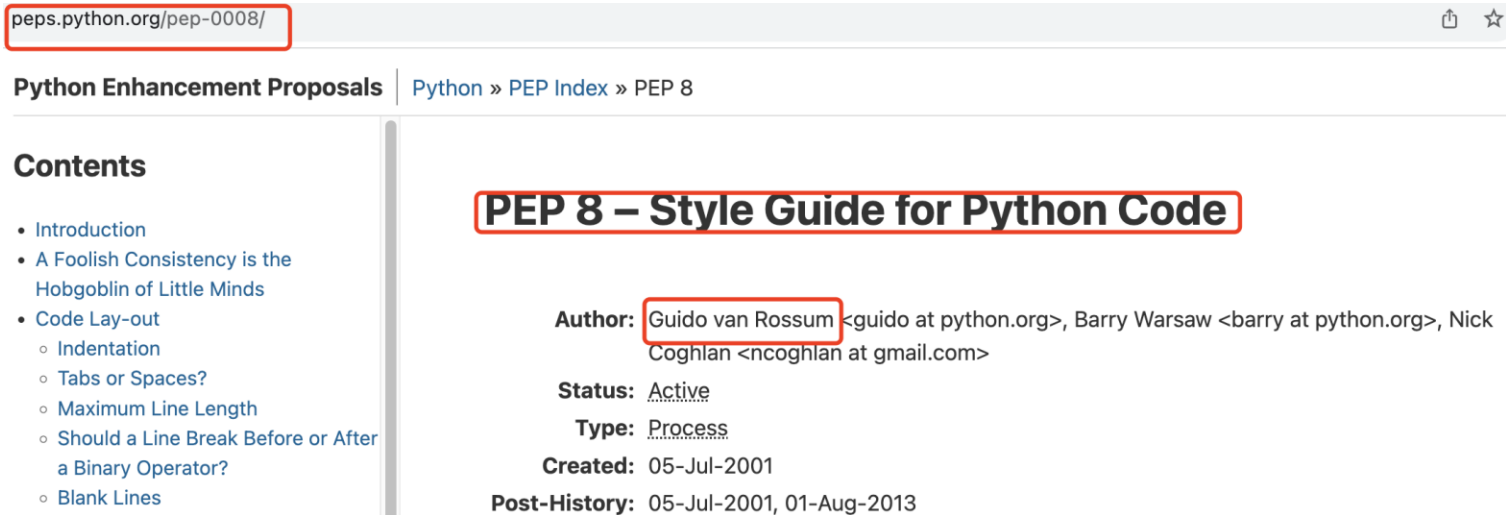
□ Python中的**关键字**，程序中有特殊作用的字符组合，不能作为标识符

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Python变量和关键字

□ Python的PEP8编码规范

- ✓ 每种语言都有其编码规范，PEP8就是Python官方指定的编码规范



The screenshot shows a web browser window with the URL `peps.python.org/pep-0008/` in the address bar. The page title is "Python Enhancement Proposals" and the breadcrumb is "Python » PEP Index » PEP 8". The main heading is "PEP 8 – Style Guide for Python Code". The author information is "Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>". The status is "Active", the type is "Process", the creation date is "05-Jul-2001", and the post-history is "05-Jul-2001, 01-Aug-2013". A table of contents is visible on the left side of the page.

`peps.python.org/pep-0008/`

Python Enhancement Proposals | Python » PEP Index » PEP 8

Contents

- Introduction
- A Foolish Consistency is the Hobgoblin of Little Minds
- Code Lay-out
 - Indentation
 - Tabs or Spaces?
 - Maximum Line Length
 - Should a Line Break Before or After a Binary Operator?
 - Blank Lines

PEP 8 – Style Guide for Python Code

Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>

Status: Active

Type: Process

Created: 05-Jul-2001

Post-History: 05-Jul-2001, 01-Aug-2013

<https://peps.python.org/pep-0008/>

Python变量和关键字

- **变量和函数名**应小写，如果包含多个单词，应使用下划线进行分隔
- **常量**应大写，如果包含多个单词，应使用下划线进行分隔

```
# Good
student_name = "Alice"
def calculate_average(score_list):
    return sum(score_list) / len(score_list)
```

```
# Good
PI = 3.14159
MAXIMUM_SPEED = 120
```

- **类名**应使用驼峰式命名法，即首字母大写，如果类名包含多个单词，首字母大写
- **方法名和实例变量**应小写，如果包含多个单词，应使用下划线进行分隔。

```
# Good
class StudentProfile:
    pass
```

```
# Good
class StudentProfile:
    def init(self, name, age):
        self.student_name = name
        self.student_age = age
```

Python变量赋值

□ 赋值语句可以将**变量**和**对象**进行**关联**

```
#代码2.3 使用赋值语句建立变量和对象之间的关联
#这是使用科学记数法表示浮点数0.0178的方法
var = 1.78E-2
print(var, type(var))
#复数的表示中, 带有后缀j或者J的部分为虚部
var = 3+7j
print(var, type(var))
#字符串数据既可以用单引号, 也可以用双引号进行定义
var = "人生苦短, 我用Python"
print(var, type(var))
```

```
0.0178 <class 'float' >
(3+7j) <class 'complex' >
人生苦短, 我用Python <class 'str' >
```

```
#代码2.4 赋值语句的连续赋值和多重赋值
#用于将不同的变量关联至同一个对象
x = y = z = 100
print(x, y, z)
#用于将不同的变量关联至不同的对象
a, b, c = 7, 8, 9
print(a, b, c)
```

```
100 100 100
7 8 9
```

Python变量赋值

- 由于多重赋值中的赋值操作并无先后顺序，所以可以使用多重赋值语句交换多个变量的关联关系，例如代码 2.5 所示

```
#代码 2.5 使用多重赋值交换多个变量的关联关系  
x, y = 3, 5  
print("交换之前 x 与 y 的值为: ", x, y)  
x, y = y, x  
print("交换之后 x 与 y 的值为: ", x, y)
```

```
交换之前 x 与 y 的值为: 3 5  
交换之后 x 与 y 的值为: 5 3
```

Python设计

□ 数据类型

- ✓ Python的变量不用声明，对象是储存在内存中的实体。但我们并不能直接接触到该对象。我们在程序中写的对象名，只是指向这一对象的引用(reference)。
- ✓ Python对象的类型和内存都是在运行时确定的

□ 编译

- ✓ 当Python程序运行时，编译的结果则是保存在位于内存中的PyCodeObject中，当Python程序运行结束时，Python解释器则将PyCodeObject写回到pyc文件中。
- ✓ 当Python程序第二次运行时，首先程序会在硬盘中寻找对应的pyc文件，如果找到，则直接载入，否则就重复上面的过程。

Python设计

□ 指针

- ✓ Python函数的参数是引用传递（除非对于不可变类型）

□ 内存管理

- ✓ Python采用了类似Windows内核对象一样的方式来对内存进行管理。每一个对象，都维护这一个对指向该对象的引用的计数
- ✓ 当内存中有不再使用的部分时，垃圾收集器就会把他们清理掉。它会去检查那些引用计数为0的对象，然后清除其在内存的空间。

推荐资料：<https://docs.python.org/zh-cn/3.7/faq/design.html>

Python运算符和表达式

- 表达式可以由多个**运算对象**和**运算符**构成
- Python按照运算符的**优先级**求解表达式的值
 - ✓ Python常见运算符（从最低优先级到最高优先级）
 - ✓ Python没有三目运算符，但是可以用if else

运算符	运算符描述
if -- else	条件运算符
or	逻辑或运算
and	逻辑与运算
not x	逻辑非运算
in、not in、is、is not、<、<=、>、>=、!=、==	比较运算
	按位或运算
^	按位异或运算
&	按位与运算
<<、>>	移位运算
+、-	算术运算符：加、减
*、/、//、%	算术运算符：乘、除、整除、取模（求余数）
+x、-x、~x	单操作数运算符：正、负、按位非运算
**	乘方运算符

Python运算符和表达式

#代码2.6 表达式举例

```
print(1+2*3-4)      #由于乘号的优先级高于加号和减号, 所以2*3会优先计算
print(2*3**3)      #乘方运算的优先级比乘法运算还要高
a, b = 10, 20
print(a if a>b else b) #输出变量a和b中数值较大的那一个
print(True+1)      #运算过程中, True和False可以被自动转换成1和0使用
print(3>2>1)      #连续的比较运算, 其结果与表达式3>2 and 2>1等价
print((3>2)>1)     #这条表达式计算的是子表达式3>2的值是否大于1, 即True>1
```

```
3
54
20
2
True
False
```

试一试

```
>>> 0.2+0.4==0.6
>>> round(0.2+0.4, 1) == 0.6
>>> 0xAF
>>> 9**0.5
>>> -10%3
>>> -3**2
>>> 4.0+3
>>> 12 and 34
```

Python字符串

□ 字符串类型的对象使用**一对单引号**、**一对双引号**或者**一对三引号**进行定义

✓ 使用三引号定义字符串的时候可以包含换行符

```
>>> s='Hello World'
>>> print(s)
Hello World
>>> s="Hello World"
>>> print(s)
Hello World
```

```
>>> s='''Hello World'''
>>> print(s)
Hello World
>>> s='Hello World'''
>>> print(s)
Hello World
>>> s="Hello World'''
```

```
<stdin> 1
s="Hello World'''
^
```

□ Python字符串转义字符

转义字符	含义	转义字符	含义
\'	单引号	\t	水平制表符
\"	双引号	\v	垂直制表符
\\	字符 "\" 本身	\r	回车符
\a	响铃	\f	换页符
\b	退格符	\ooo	以最多3位的八进制数作为编码值对应的字符
\n	换行符	\xhh	以必须为2位的十六进制数作为编码值对应的字符

Python字符串举例

□ Python字符串是不可变的

```
#代码2.7 定义字符串的几种方法
str1 = '单引号可以用来定义字符串'
str2 = "双引号也可以用来定义字符串"
print(str1, str2)
str3 = '''三引号定义的字符串可以直接换行
这是字符串的第二行
这是字符串的第三行'''
print(str3)
print("转义字符\n表示换行符") 原样字符串
print(r"转义字符\n表示换行符") 格式字符串
#格式标记.2f表示保留小数点后2位小数
print(f"半径为5的圆的面积是{3.14*5**2:.2f}")
```

```
单引号可以用来定义字符串 双引号也可以用来定义字符串
三引号定义的字符串可以直接换行
这是字符串的第二行
这是字符串的第三行
转义字符
表示换行符
转义字符\n表示换行符
半径为5的圆的面积是78.50
```

Python字符串不可变

□ 有几个优点

- ✓ 一个是**性能**：知道字符串是不可变的，意味着我们可以在创建时为它分配空间，并且存储需求是固定不变的。这也是元组和列表之间区别的原因之一
- ✓ Python 中的字符串被视为与数字一样**“基本”**。任何动作都不会将值 8 更改为其他值，在 Python 中，任何动作都不会将字符串“8”更改为其他值

```
>>> a = 'Hello World!'
>>> a[0] = 'Y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> a += '?'
>>> a
'Hello World!?'
>>> a.replace('H', 'Y')
'Yello World!?'
>>> a
'Hello World!?'
```

Python字符串运算

- f-string, 亦称为格式化字符串常量, 从Python3.6新引入, 目的是使格式化字符串操作更加简便
- f-string在形式上是以f或F修饰符引领的字符串: **格式: f"{表达式}"**

```
#代码2.8 与字符串有关的运算
#字符串之间用加法运算连接, 结果为连接后的字符串
print("ABCD"+"1234")
#字符串和整数进行乘法运算, 结果为字符串复制多遍并连接
print("Hi"*3)
str1 = "I love Python!"
#获取字符串str1中的首个字符
print(f"str1[0] = {str1[0]}")
#获取字符串str1的最后一个字符
print(f"str1[-1] = {str1[-1]}")
#获取字符串中索引号从2开始到6之前结束的子字符串, 即"love"
print(f"str1[2:6] = {str1[2:6]}")
```

```
ABCD1234
HiHiHi
str1[0] = I
str1[-1] = !
str1[2:6] = love
```

Python字符串运算

- `f"{}"` 和 `"{}.format()"` 都是 Python 中用于格式化字符串的两种常用方式
 - ✓ `f"{}"` 在 3.6 版本后引入，简洁高效
 - ✓ `"{}.format()"` 在 2.7 版本后引入，兼容旧版

```
name = "Alice"
age = 30
# 使用 f-string
result = f"My name is {name} and I am {age} years old."
# 使用 .format
result = "My name is {} and I am {} years old.".format(name, age)

number = 3.14159
# 使用 f-string
formatted_number = f"{number:.2f}"
# 使用 .format
formatted_number = "{:.2f}".format(number)
```

Python字符串的常用函数

□ 字符串对象有一系列已定义好的函数可直接使用

#代码2.9 字符串对象的常用方法

```
str1 = "i have an apple."  
print(f"str1.capitalize() = {str1.capitalize()}") #用于将字符串的首字母大写  
print(f"str1.title() = {str1.title()}") #用于将字符串中每个单词的首字母大写  
print(f"str1.count('a') = {str1.count('a')}") #用于统计指定的字符在字符串中出现的次数  
print(f"str1.startswith('i am') = {str1.startswith('i am')}") #用于判定字符串是否以指定的参数开始  
print(f"str1.endswith('apple.') = {str1.endswith('apple.')}") #用于判定字符串是否以指定的参数结尾  
print(f"str1.find('apple') = {str1.find('apple')}") #用于查找指定的子字符串并返回其起始位置  
print(f"str1.find('pear') = {str1.find('pear')}") #若无法找到指定的子字符串, find()方法会返回-1  
print(f"str1.split() = {str1.split()}") #对字符串进行切割, 默认为所有的空字符, 包括空格、换行(\n)、制表符(\t)等。作为分隔符  
print(f"','.join(['a','b','c']) = {'','.join(['a','b','c'])}") #用指定的字符串将若干字符串类型的对象进行连接  
print(f"'ABcd'.upper() = {'ABcd'.upper()}") #将字符串中的字母转换成大写  
print(f"'ABcd'.lower() = {'ABcd'.lower()}") #将字符串中的字母转换成小写  
print(f"' ABcd '.strip() = {' ABcd '.strip()}") #删除字符串前后的特殊字符, 比如空格和换行符
```

```
str1.capitalize() = I have an apple.  
str1.title() = I Have An Apple.  
str1.count('a') = 3  
str1.startswith('i am') = False  
str1.endswith('apple.') = True  
str1.find('apple') = 10  
str1.find('pear') = -1  
str1.split() = ['i', 'have', 'an', 'apple.']  
','.join(['a','b','c']) = a,b,c  
'ABcd'.upper() = ABCD  
'ABcd'.lower() = abcd  
' ABcd '.strip() = ABcd
```


Python选择结构

□ 用if...elif...else来构造选择结构的程序代码

```
#代码2.10 将百分制成绩转换成五级制成绩
score = float(input("请输入一个百分制成绩: "))
if score >= 90:
    print(f"成绩{score}对应的五级制成绩是优秀")
elif score >= 80:
    print(f"成绩{score}对应的五级制成绩是良好")
elif score >= 70:
    print(f"成绩{score}对应的五级制成绩是中等")
elif score >= 60:
    print(f"成绩{score}对应的五级制成绩是及格")
else:
    print(f"成绩{score}对应的五级制成绩是不及格")
```

```
请输入一个百分制成绩: 59.5
成绩59.5对应的五级制成绩是不及格
```

Python选择结构

□ 用if...elif...else来构造选择结构的程序代码

X python没有switch语句

为什么Python中没有switch或case语句？

你可以通过一系列 `if... elif... elif... else` 轻松完成这项工作。对于switch语句语法已经有了一些建议，但尚未就是否以及如何进行范围测试达成共识。有关完整的详细信息和当前状态，请参阅 [PEP 275](#)。

对于需要从大量可能性中进行选择的情况，可以创建一个字典，将case 值映射到要调用的函数。例如：

```
def function_1(...):
    ...

functions = {'a': function_1,
            'b': function_2,
            'c': self.method_1, ...}

func = functions[value]
func()
```

Python选择结构

□ 通过组合多个 **if...elif...else** 结构进行嵌套使用

```
#代码2.11 判断从键盘上输入的三个整数构成的三角形形状
a,b,c = input("请输入组成三条边, 用空格分隔: ").split()
a,b,c = int(a), int(b), int(c)
a,c,b = min(a,b,c), max(a,b,c), a+b+c-min(a,b,c)-max(a,c,b)
if a+b<=c:
    print("输入的整数无法构成三角形")
else:
    if a==b==c:
        print("输入的整数构成一个等边三角形")
    elif a==b:
        print("输入的整数构成一个等腰三角形")
    elif a**2+b**2==c**2:
        print("输入的整数构成一个直角三角形")
    else:
        print("输入的整数构成一个普通三角形")
```

Python不能在表达式中赋值

Python循环结构

□ 用关键字**while**或者**for**可以用来构造循环结构

- ✓ while语句实现累加
- ✓ for语句实现累加

```
#代码2.12 使用while循环完成1至100所有整数求和
n = 1
s = 0
while n<=100:
    s += n    #与s = s + n等价
    n += 1    #与n = n + 1等价
print("1到100所有整数的和是：",s)
```

```
#代码2.13 使用for循环完成1至100所有整数求和
s = 0
for n in range(1,101):
    s += n
print("1到100所有整数的和是：",s)
```

- ✓ **range(1,101)**所表示的从1到101以内（不含101）的所有整数

Python循环结构

- 如果**range()函数只有一个参数**，则该参数表示返回结果的终止值（不含）
- 如果**range()函数有两个参数**，则第1个参数表示返回结果的起始值，第2个参数表示返回结果的终止值（不含）
- 如果**range()函数有三个参数**，则第1个参数表示返回结果的起始值，第2个参数表示返回结果的终止值（不含），第3个参数表示每次步进的增量值

```
#代码2.14 求100以内所有奇数的和  
#sum()函数的功能为对其参数进行求和  
s = sum(range(1,100,2))  
print("100以内所有奇数的和是: ",s)
```

for...else...

for 临时变量 in 序列:
 重复执行的代码

.....

else:
 循环正常结束后要执行的代码

- 若**循环正常结束**，执行else下方缩进的代码
- 若**循环被break终止**，else下方缩进的代码将不执行

```
# Example using break statement
for i in range(5):
    print(i)
else:
    print("循环已完成\n")
```

```
0
1
2
3
4
循环已完成
```

for...else...

- 若**循环正常结束**，执行else下方缩进的代码
- 若**循环被break终止**，else下方缩进的代码将不执行

```
for i in range(5):  
    if i == 3:  
        print("达到3, break\n")  
        break  
    print(i)  
else:  
    print("循环已完成\n")
```

```
0  
1  
2  
达到3, break
```

```
# Example using continue statement  
for i in range(5):  
    if i == 3:  
        print("达到3, continue")  
        continue  
    print(i)  
else:  
    print("循环已完成")
```

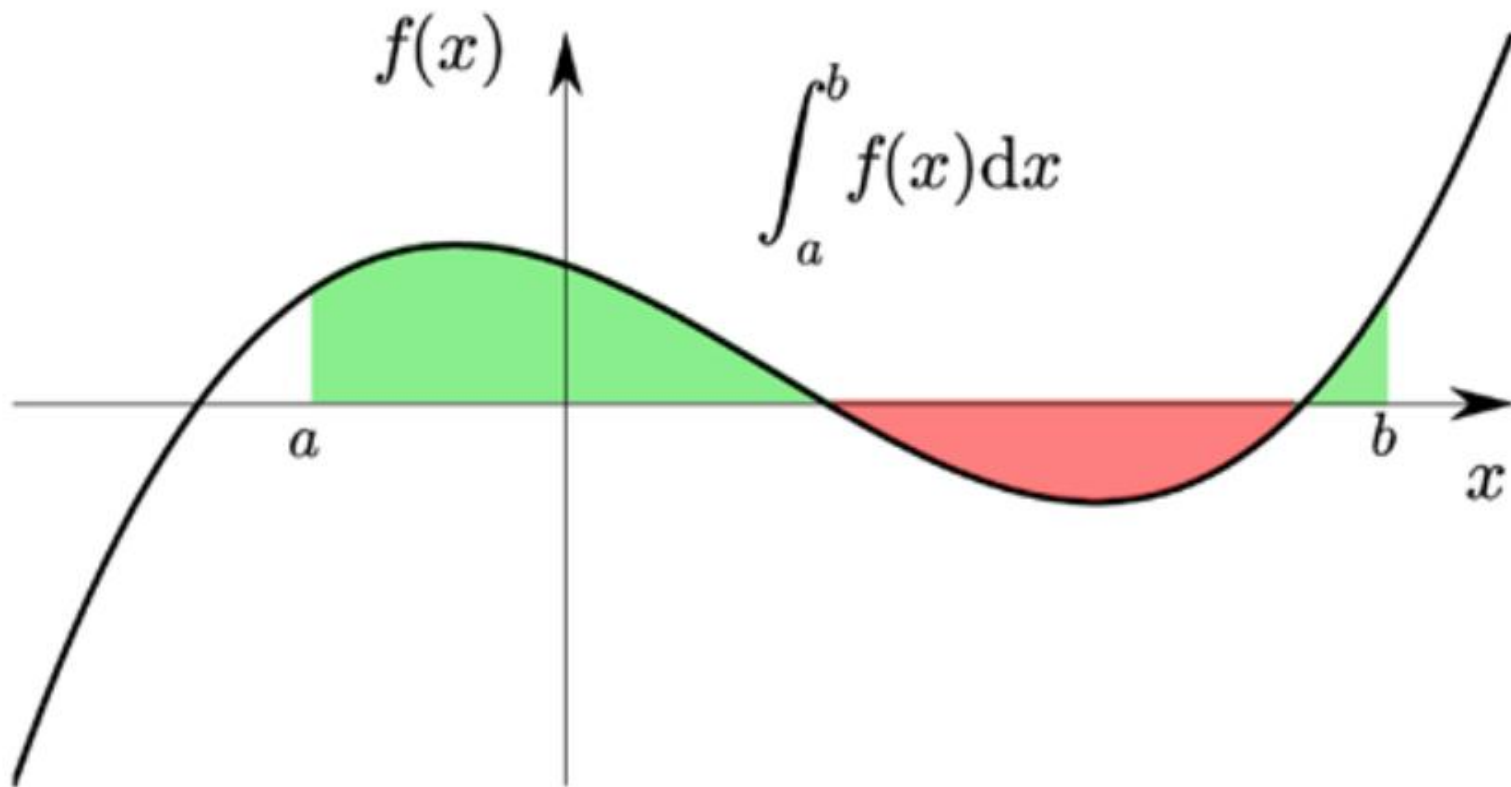
```
0  
1  
2  
达到3, continue  
4  
循环已完成
```

Python循环结构

- **break**语句会终结最近的外层循环，如果循环有可选的else子句，也会跳过该子句
- **continue**语句会终止当前循环中剩下的语句，并继续执行最近的外层循环的下一个轮次

```
#代码2.15 输出从小到大排列的第100个素数
count = 0
num = 2
while True:
    #下方的for循环用于判断num是否为素数，如果是则计数器count加1
    for i in range(2,num):
        if num%i==0:
            break
    else:
        count += 1
    #如果计数器小于100，num加1，且终止当前循环中剩下的语句，并继续执行循环的下一个轮次
    if count<100:
        num += 1
        continue
    #剩下的代码只有在上方if不成立的时候才会被执行到，即计数器count为100的情况下
    print(num,"是从小到大排列的第100个素数")
    break #输出完毕后，使用break语句终结外层的while循环
```


一起读代码



变量间的区别

□ 无下划线变量

- ✓ 无下划线变量为**公有变量**

□ 前面单下划线

- ✓ `_xx`: 前置单下划线, 又称**口头私有变量**, 私有化属性或方法的一种, 一般来讲, 变量名`_xx`从被看作是“私有的, 在模块或类外不可以使用。当变量是私有的时候, 用`_xx`来表示变量是很好的习惯。类对象和子类可以访问, 这并不能完全做到真正的私有, 只是约定俗成的而已, 这样写表示不希望这个变量在外部被直接调用

□ 前面双下划线

- ✓ `__xx`: 前置双下划线, **私有化属性或方法**, 只有内部可以访问, 外部不能访问。

变量间的区别

□ 前后都有双下划线

- ✓ `__xx__`: 以双下划线开头, 并且以双下划线结尾的, 是**特殊变量** (这就是在python中强大的魔法方法), 特殊变量是可以直接访问的, 对于普通的变量应当避免这种命名风格

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y +
other.y)

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(5, 6)

result = v1 + v2 # 实际调用 v1.__add__(v2)
print(result) # 输出: Vector(7, 9)
```

□ 后置下划线

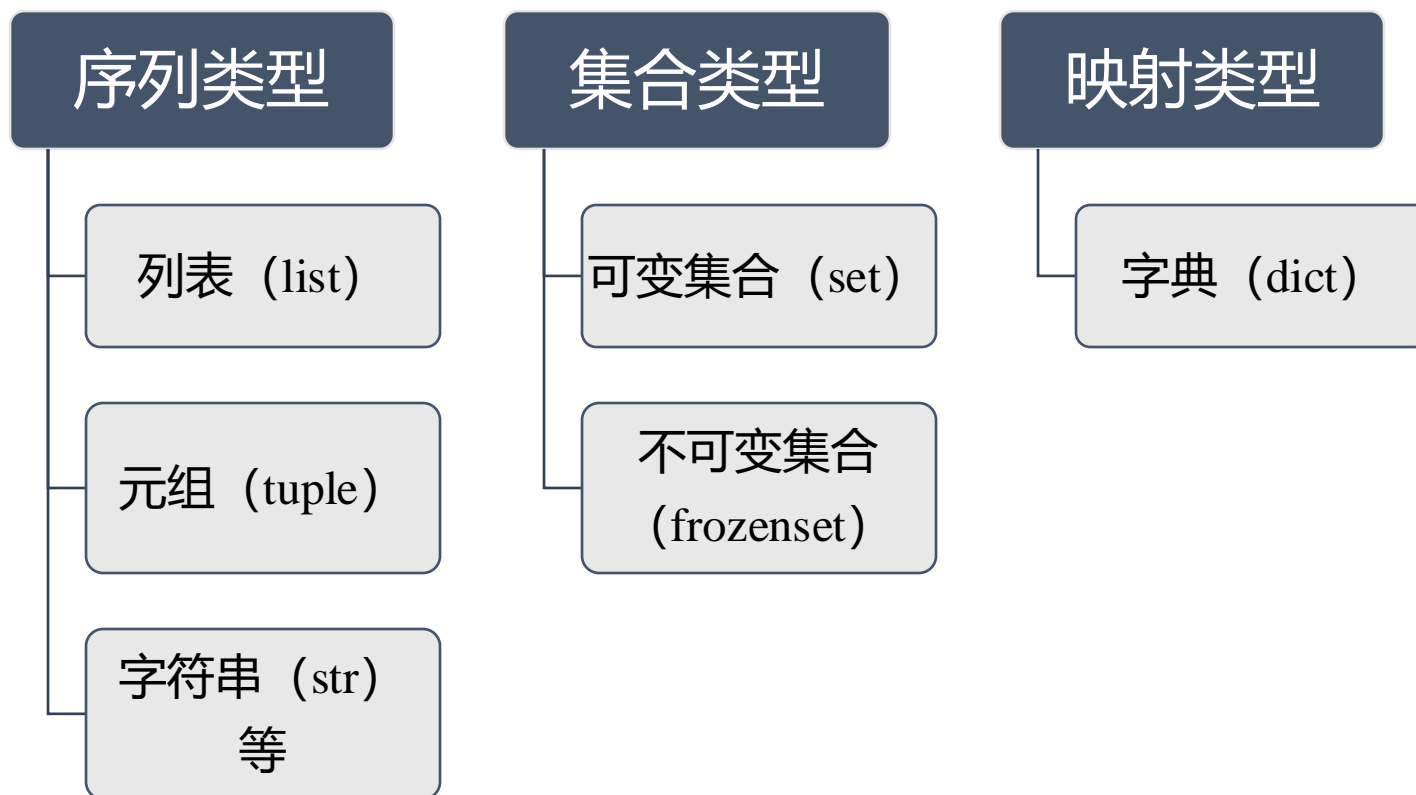
- ✓ `xx_`: 后置单下划线, 用于**避免与Python关键词的冲突**

目录

- **基本语法**
 - **对象、类型、变量和赋值**
 - **运算符和表达式**
 - **字符串**
 - **流程控制**
- **组合数据类型**
- **函数**
- **异常处理和文件操作**
- **面向对象基础**
- **数值计算库NumPy**

组合数据类型

- 组合数据类型可以将多个数据组织起来，根据数据组织方式的不同，Python的组合数据类型可分成三类：**序列**类型、**集合**类型和**映射**类型



组合数据类型——列表

- 列表 (list) 是一种**序列数据类型**，是一种**可变数据类型**，内部元素可以任意增删和修改
- 列表对象的**定义和基本运算**如下：

#代码2.16 列表对象的定义和基本运算

```
lst1 = [] #定义一个空列表
```

```
#len()函数可以返回参数对象中包含元素的数量
```

```
print(f"len(lst1) = {len(lst1)}")
```

```
#list()函数可以将参数对象转换成列表
```

```
lst1 = list("Python!")
```

```
#列表也是序列类型对象，所以支持索引和切片
```

```
print(f"lst1[2] = {lst1[2]}, lst1[-2] = {lst1[-2]}")
```

```
print(f"lst1[2:-2] = {lst1[2:-2]}")
```

```
lst1 = [1,2,3]
```

```
lst2 = [1,2,3]
```

```
#通过id()函数我们可以取得对象的身份标识
```

```
print(f"{id(lst1)=}, {id(lst2)=}")
```

```
print("lst1和lst2是否相等：", lst1==lst2) # True
```

```
print("lst1和lst2是否相同：", lst1 is lst2) # False
```

```
#不同的变量与同一个列表对象关联
```

```
lst1 = lst2 = [1,2,3]
```

```
#无论操作变量lst1或者lst2，其实都是在操作同一个列表
```

```
lst1[0] = 100
```

列表的常用方法

□ 列表对象具有一系列定义好的方法可以直接使用

```
#代码2.17 列表对象的常用方法
```

```
lst1 = list("Python")
```

```
lst1.append("666") #追加元素到列表中
```

```
lst1.insert(1, "is") #插入元素到列表中
```

```
lst1.remove("i") #从列表中移除指定的元素
```

```
#从指定位置弹出列表元素，默认弹出列表的  
最后一个元素
```

```
print(f'lst1.pop(0) = {lst1.pop(0)}')
```

```
#将参数转换成列表元素后，连接到原列表的  
末尾
```

```
lst1.extend([1, 2, 3])
```

```
#对列表元素按照ASCII码从小到大的顺序进行排序
```

```
lst1.sort()
```

```
#在列表中对指定的内容进行计数
```

```
print("'o'在lst1中出现了", lst1.count("o"), "次")
```

```
#在列表中查找指定的内容，返回其索引值
```

```
print("'t'在lst1中的索引值是：", lst1.index("t"))
```

```
lst1 = [1,2,3]
```

```
#通过copy()方法可以复制原列表对象，得到一个新的  
列表对象
```

```
lst2 = lst1.copy()
```

```
lst2[0] = 100
```

```
print(f'{lst1 = }, {lst2 = }')
```

Python解释器

```
>>> L = []
>>> dir(L)
['_add_', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
 '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 're

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

>>> L.append(1)
>>> L
[1]
```

`dir()` 函数用于获取一个对象的所有属性和方法的列表。它返回一个包含对象的属性（包括方法）的字符串列表。这个列表通常用于了解对象的可用功能，帮助你在编程中探索和使用它。

列表推导式

□ 列表推导式将**列表元素的生成规则放置在一对方括号内完成列表的创建**

```
#代码2.18 列表推导式的使用
```

```
lst1 = [n for n in range(1,10) if n%2==1] #创建列表包含10以内的奇数
```

```
lst2 = [5*n for n in range(5)] #创建等差数列构成的列表
```

```
lst3 = [3**n for n in range(5)] #创建等比数列构成的列表
```

```
#判断元素是否在列表里
```

```
5 in lst2
```

思考

□ Python列表是数组还是链表？

列表是如何在CPython中实现的？

CPython的列表实际上是可变长度的数组，而不是lisp风格的链表。该实现使用对其他对象的引用的连续数组，并在列表头结构中保留指向该数组和数组长度的指针。

这使得索引列表 `a[i]` 的操作成本与列表的大小或索引的值无关。

当添加或插入项时，将调整引用数组的大小。并采用了一些巧妙的方法来提高重复添加项的性能；当数组必须增长时，会分配一些额外的空间，以便在接下来的几次中不需要实际调整大小。

基于列表的栈和队列

□ 栈

- ✓ 先进后出, 后进先出
- ✓ 入栈: `append()`
- ✓ 出栈: `pop()`

□ 队列

- ✓ 先进先出
- ✓ 入队: `append()`
- ✓ 出队: `pop(0)`

append和insert, 哪个效率更高, 为什么?

```
l = []
```

```
for i in range(1000000):
```

```
    l.append(i)
```

```
l = []
```

```
for i in range(1000000):
```

```
    l.insert(0, i)
```

组合数据类型——元组

- 元组 (tuple) 也是一种序列数据类型，但是它是一种**不可变数据类型**，其**元素不可增删改**

#代码2.19 元组对象的定义和基本运算

#定义空元组、一元组、二元组

```
tup1,tup2,tup3 = tuple(),(1,),(2,3)
```

#通过+运算连接两个旧元组，得到一个新元组

```
tup1 = tup2 + tup3
```

```
tup1 = tuple("Python!")
```

#元组也支持索引和切片运算

```
print(f"{tup1[2]}, {tup1[3:6]}")
```

#由于元组是不可变对象，元素不可以修改，所以该语句报错

```
tup1[2] = 'a'
```

#代码2.20 元组对象的常用方法

#tuple()函数可以将参数对象转换成元组

```
tup1 = tuple("Beautiful is better than ugly.")
```

#统计指定的参数在元组中出现的次数

```
print(f" tup1.count('t') = {tup1.count('t')}")
```

#查找指定的参数，并返回其在元组中的索引值

```
print(f" tup1.index('a') = {tup1.index('a')}")
```

列表与元组的区别

□ 性能

- ✓ 由于元组是**不可变的**，它们通常比列表具有更高的性能，因为不需要考虑元素的添加或删除
- ✓ 列表的**可变性**可能导致更多的内存分配和复制操作，从而在某些情况下影响性能

□ 应用场景

- ✓ 列表通常用于需要动态增长和修改元素的情况，例如存储一组项目或记录
- ✓ 元组通常用于表示不可更改的数据集，例如坐标、日期时间、函数的多返回值等

组合数据类型——字典

- 字典 (dict) 是表示**映射关系的类型**，由**键-值**对构成字典中的元素，是一种**可变数据类型**
- 字典不是序列类型，**其中元素不能使用索引值进行访问，而是通过键访问对应元素，键必须唯一，不可变**

#代码2.21 字典对象的定义和基本运算

```
dct1 = {} #定义一个空字典
```

#字典元素的键可以是整数、字符串和元组等不可变对象

```
dct1 = {1:"Python", "Tom":"Male", (3,4):"Red"}
```

```
dct1[(3,4)] = "Blue" #字典的元素是可以被修改的
```

```
del dct1[1] #字典的元素可以被删除
```

```
dct1[2] = "Java" #字典中可以随时添加元素
```

```
print(f"After change: {dct1}")
```

#如果一个容器对象中的每个元素都是包含2个元素的对象，则可以被转换成字典

```
dct2 = dict([[1,"Python"],["Tom","Male"],[(3,4),"Red"]])
```

#使用zip()函数可以在两个对象之间建立元素的对应关系，从而创建字典

```
dct3 = dict(zip(["Tom","Jack","Rose"],["Male","Male","Female"]))
```

字典的常用方法

#keys()方法可以返回字典中的所有键

```
print(f"{dct1.keys() }")
```

#values()方法可以返回字典中的所有值

```
print(f"{dct1.values() }")
```

#items()方法可以返回字典中的所有键-值对

```
print(f"{dct1.items() }")
```

#get()方法用来返回以参数1作为键的元素值，如果找不到则返回参数2的内容

```
print(f"{dct1.get('Tom','Unknown') }")
```

```
print(f"{dct1.get('Jerry','Unknown') }")
```

#copy()方法通过拷贝原字典数据创建一个新的字典对象

```
dct2 = dct1.copy()
```

```
print(f"{dct2}")
```

#popitem()方法用于弹出最后被加入字典的元素

```
print(f"{dct2.popitem()}")
```

#pop()方法用于弹出字典中的指定元素，并返回其值

```
print(f"{dct2.pop('Tom')}")
```

#update() 作用是使用参数对象的内容对字典进行更新

```
dct2.update({'Jerry':'Male'})
```

#clear()方法可以将字典中的元素清空

```
print(f"{dct2.clear()}")
```

#fromkeys()用参数1作为键，参数2作为值，创建字典

```
dct3 = dict.fromkeys(["Alice","Mary"],"Female")
```

#setdefault()用于以参数作为键-值对插入元素

```
print(f"{dct3.setdefault('Alice','Male')}")
```


字典是如何实现的？

字典是如何在CPython中实现的？

CPython的字典实现为可调整大小的哈希表。与B-树相比，这在大多数情况下为查找（目前最常见的操作）提供了更好的性能，并且实现更简单。

字典的工作方式是使用 `hash()` 内置函数计算字典中存储的每个键的hash代码。hash代码根据键和每个进程的种子而变化很大；例如，“Python”的hash值为-539294296，而“python”（一个按位不同的字符串）的hash值为1142331976。然后，hash代码用于计算内部数组中将存储该值的位置。假设您存储的键都具有不同的hash值，这意味着字典需要恒定的时间 -- $O(1)$ ，用Big-O表示法 -- 来检索一个键。

使用list还是dict?

□ 场景：学生管理系统，每个学生有学号、姓名、班级、成绩等信息

□ 功能1：对于学生成绩进行排序

```
self.students.sort(key=lambda x: x.score)
```

□ 功能2：根据学号修改成绩等信息

```
self.students[student_id]['score'] = new_score
```

组合数据类型——集合

- 集合是一种非序列容器对象，其中的**每个元素都是唯一**的，可变集合（set）表示该集合创建后依然可以对其中元素进行增减或修改，不可变集合（frozenset）表示该集合一旦创建，元素便无法修改

#代码2.24 集合对象的定义和基本运算

```
set1 = {1,2,3,4,5} #定义可变集合对象
```

```
set2 = {3,4,5,6,7}
```

```
print(f"{3 in set1}")
```

```
print(f"{set1|set2}") #求两个集合的并集
```

```
print(f"{set1&set2}") #求两个集合的交集
```

```
print(f"{set1-set2}") #求两个集合的差集
```

```
print(f"{set2-set1}")
```

```
set2 = set1
```

```
print(f"{set1}, {set2}")
```

#判断集合set1是否为集合set2的真子集

```
print(f"{set1 < set2}")
```

#判断集合set1是否为集合set2的子集

```
print(f"{set1 <= set2}")
```

```
set3 = frozenset({1,2,3}) #创建不可变集合
```

```
print(f"{set3}")
```

#不可变集合是无法对元素进行修改的，所以程序报错

```
set3.add(4)
```

集合的常用方法

#代码2.25 集合对象的常用方法

set1 = {1,2,3,4,5} #定义可变集合对象

set1.add(6) #向可变集合中添加元素

print(f"After add: {set1}")

#从可变集合中删除元素，如果该元素不存在就报错

set1.remove(6)

#从可变集合中删除元素，如果该元素不存在，什么也不做

set1.discard(6)

set2 = {3,4,5,6,7}

#求两个集合的并集

print(f"{set1.union(set2)}")

#求两个集合的交集

print(f"{set1.intersection(set2)}")

#求两个集合的差集

print(f"{set1.difference(set2)}")

#用参数中的对象更新可变集合

set1.update(set2)

#判断set2是否是set1的子集

print(f"{set2.issubset(set1)}")

#判断set1是否是set2的父集

print(f"{set1.issuperset(set2)}")

Python函数定义和调用

- Python函数与C语言函数基本相似，定义的关键字是**def**
- Python函数的**形式参数可以设置默认值**

#代码2.26 从键盘上输入一个正整数，判断其是否为素数

```
def is_prime(n):  
    for i in range(2,n):  
        if n%i==0:  
            return False  
    return True  
num = int(input("请输入一个大于2的正整数: "))  
print(f"{num}是素数" if is_prime(num) else f"{num}  
不是素数")
```

#代码2.27 参数的默认值

```
def fun(a=10, b=20, c=30):  
    return f"{a} + {b} + {c} = {a+b+c}"  
print(fun(15, 25, 35)) #没有使用参数默认值  
print(fun(15, 20)) #参数c使用了默认值30  
print(fun(b=15)) #参数a和参数c使用了默认值  
print(fun()) #所有参数都采用默认值
```

引用传递和值传递

□ Python中的变量是没有类型的，我们可以把它看做一个(*void)类型的指针，变量是可以指向任何对象的，而对象才是有类型的。

□ Python中的对象有不可变对象 (number, string, tuple等) 和可变对象之分 (list, dict等) 。

1. **不可变对象**作为函数参数，相当于C语言的**值传递**。

2. **可变对象**作为函数参数，相当于C语言的**引用传递**。（因列表是可变数据类型，作为参数传递实则是传递对列表的引用，修改更新列表值后依旧引用不变）

□ Python函数可以return多个值

引用传递和值传递

#值传递（传递的是不可变对象） 示例

```
def modify_value(my_value):  
    my_value = my_value + 1  
  
my_integer = 5  
print("Original Integer:", my_integer)  
  
modify_value(my_integer)  
print("Modified Integer:", my_integer)
```

```
Original Integer: 5  
Modified Integer: 5
```

引用传递（传递的是对象的引用） 示例

```
def modify_list(my_list):  
    my_list.append(4)  
  
my_list = [1, 2, 3]  
print("Original List:", my_list)  
  
modify_list(my_list)  
print("Modified List:", my_list)
```

```
Original List: [1, 2, 3]  
Modified List: [1, 2, 3, 4]
```

匿名函数与lambda关键字

□ **lambda**表达式的功能是创建一个非常短小的函数应用，将参数和返回值的
关系进行约定

```
g = lambda x: x+1
```

```
g(5)
```

```
#代码2.28 匿名函数lambda表达式的应用
```

```
ages = {"Tom":20, "Jack":19, "Rose":18, "Mary":21}
```

```
#默认的排序方式是按照键的从小到大顺序
```

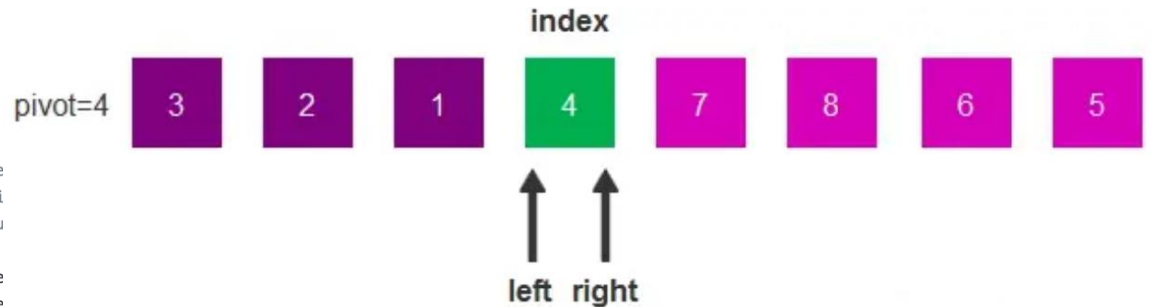
```
print(f"{sorted(ages)}")
```

```
#在sorted函数中，可以通过指定key参数对应的函数来改变排序的依据，以  
lambda表达式的返回值，即年龄进行排序
```

```
print(f"{sorted(ages, key=lambda x: ages[x])}")
```


一起读代码

```
1 """
2 A pure Python implementation of the quick sort algorithm
3
4 For doctests run following command:
5 python3 -m doctest -v quick_sort.py
6
7 For manual testing run:
8 python3 quick_sort.py
9 """
10 from __future__ import annotations
11
12
13 def quick_sort(collection: list) -> list:
14     """A pure Python implementation of quick sort algorithm
15
16     :param collection: a mutable collection of comparable it
17     :return: the same collection ordered by ascending
18
19     Examples:
20     >>> quick_sort([0, 5, 3, 2, 2])
21     [0, 2, 2, 3, 5]
22     >>> quick_sort([])
23     []
24     >>> quick_sort([-2, 5, 0, -45])
25     [-45, -2, 0, 5]
26     """
27     if len(collection) < 2:
28         return collection
29     pivot = collection.pop() # Use the last element as the
30     greater: list[int] = [] # All elements greater than pi
31     lesser: list[int] = [] # All elements less than or equ
32     for element in collection:
33         (greater if element > pivot else lesser).append(ele
34     return quick_sort(lesser) + [pivot] + quick_sort(greate
35
36
37 if __name__ == "__main__":
38     user_input = input("Enter numbers separated by a comma:\n").strip()
39     unsorted = [int(item) for item in user_input.split(",")]
40     print(quick_sort(unsorted))
```



https://github.com/TheAlgorithms/Python/blob/master/sorts/quick_sort.py

目录

- Python基本语法
 - 对象、类型、变量和赋值
 - 运算符和表达式
 - 字符串
 - 流程控制
- 组合数据类型
- 函数
- 异常处理和文件操作
- 面向对象基础
- 数值计算库NumPy

Python异常处理

- 程序在执行过程中会**遇到未知的错误**，Python语言通过异常处理机制在运行程序发生错误时

#代码2.29 异常处理机制的使用

```
lst1 = [2,0]
```

```
for i in range(3):
```

```
    print(f"循环的第{i+1}次运行: ")
```

```
    try:
```

```
        print(f"{36/lst1[i] = }")
```

```
    except Exception as e:
```

```
        print("程序产生了异常: ", e)
```

```
    else:
```

```
        print("程序没有产生异常")
```

```
    finally:
```

```
        print("无论有没有产生异常, 这里的代码都会被执行")
```

Python异常捕获

□ try的工作原理是，当开始一个try语句后，python就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，**try子句先执行**，接下来会发生什么依赖于执行时是否出现异常

✓ **如果当try后的语句执行时发生异常**

→ python就跳回到try并执行第一个匹配该异常的except子句，异常处理完毕，控制流就通过整个try语句（除非在处理异常时又引发新的异常）

→ 如果在try后的语句里发生了异常，却没有匹配的except子句，异常将被递交到上层的try，或者到程序的最上层（这样将结束程序，并打印默认的出错信息）

✓ **如果在try子句执行时没有发生异常**，python将执行else语句后的语句（如果有else的话），然后控制流通过整个try语句

✓ **try-finally 语句无论是否发生异常都将执行最后的代码**

什么情况下使用try...except...

□ 问题：对班级同学进行点名，用字典记录每位同学的姓名和出勤次数

- ✓ count = {}
- ✓ 功能1：点名的时候，如果出勤，则+1
- ✓ 功能2：在文件中记录本次的点名，在下一次课重新读取该文件

```
# 功能1：点名时，如果出勤，则+1
def mark_attendance(student_name):
    try:
        count[student_name] += 1
    except KeyError:
        # 如果学生不在字典中，将其添加并初始化为1
        count[student_name] = 1
```

```
# 功能2：下一次课重新读取该文件
def load_attendance_from_file(filename):
    try: with open(filename, "r") as file:
        for line in file:
            student, attendance = line.strip().split(": ")
            count[student] = int(attendance)
    except FileNotFoundError: # 如果文件不存在，不做任何操作
        pass
```

什么情况下使用异常捕获

如果没有引发异常，则try/except块的效率极高。实际上捕获异常是昂贵的。在2.0之前的Python版本中，通常使用这个习惯用法：

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

只有当你期望dict在任何时候都有key时，这才有意义。如果不是这样的话，你就是应该这样编码：

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

什么情况下使用异常捕获

□ 观点1

- ✓ The point is that using try/except statements is in many cases much more natural (more "Pythonic") than if/else, and you should get into the habit of using them where you can.

①try/except语句在 Python 中的表现可以用海军少将Grace Hopper的妙语解释：“请求宽恕易于请求许可。”在做一件事时去处理可能出现的错误，而不是在开始做事前就进行大量的检查，这个策略可以总结为习语“看前就跳（Leap Before You Look）”。

□ 观点2

- ✓ 作为使用者，应该按照所使用库的习惯来决定用那种方式处理错误，如果库使用throw则try，如果库使用return则if
- ✓ 而自己的模块，开发时间短，脚本为临时性的则try-except；如代码比较关键要求稳定，如出错需要详细的错误提示则if-else
- ✓ 如果涉及到文件、网络等外部操作，尽量多用try-except

Python文件处理

1. r (只读模式):

- **用途:** 以只读模式打开文件。
- **文件要求:** 文件必须存在, 如果文件不存在, 会抛出 `FileNotFoundError`。
- **指针位置:** 文件的指针会指向文件的开头。

2. w (只写模式):

- **用途:** 以写入模式打开文件。
- **文件要求:** 如果文件不存在, 会创建新文件; 如果文件存在, 会清空文件内容。
- **指针位置:** 文件指针指向文件的开头, 原有内容会被覆盖。

3. a (追加模式):

- **用途:** 以追加模式打开文件。
- **文件要求:** 如果文件不存在, 会创建新文件; 如果文件存在, 数据会追加到文件末尾。
- **指针位置:** 文件指针会移动到文件末尾, 新的内容将从末尾追加, 不会修改原有内容。

4. r+ (读写模式):

- **用途:** 以读写模式打开文件, 允许读取和写入。
- **文件要求:** 文件必须存在, 如果文件不存在, 会抛出 `FileNotFoundError`。
- **指针位置:** 文件指针位于开头, 允许同时读写文件内容。
- **注意:** 写操作不会自动清空文件内容, 写入操作会覆盖从指针位置开始的内容。

5. w+ (读写模式):

- **用途:** 以读写模式打开文件, 允许读取和写入。
- **文件要求:** 如果文件不存在, 会创建新文件; 如果文件存在, 文件内容会被清空。
- **指针位置:** 文件指针位于开头, 原有内容会被清空。

6. a+ (读写追加模式):

- **用途:** 以读写和追加模式打开文件, 允许读取和追加内容。
- **文件要求:** 如果文件不存在, 会创建新文件。
- **指针位置:** 文件指针位于文件末尾, 追加的内容从末尾开始写入; 但读操作可以通过 `seek()` 移动指针, 进行任意位置的读取。

Python文件处理

- 与C语言相同，Python支持以文本文件或者二进制文件操作，一般过程包括：
 - ✓ 使用`open()`函数打开文件对象
 - 文件路径
 - 打开方式：w,r,a,w+,r+,a+,wb,rb,ab,wb+,rb+,ab+（+：读写，b：二进制）
 - ✓ 读/写文件
 - `write()`（写入字符串）、`writelines()`（写入字符串列表）
 - `read()`：读取整个文件，将文件内容放到一个字符串变量中
 - `readline()`：每次读取一行；返回一个字符串对象，保持当前行的内存
 - `readlines()`：一次性读取整个文件；自动将文件内容分析成一个行的列表
 - ✓ 使用`close()`函数关闭文件对象
 - 若使用关键字`with`包含文件操作的程序，则关闭文件的代码可以省略

Python读写文件

□ 使用with

- ✓ with表达式其实是try-finally的简写形式
- ✓ with本身并没有异常捕获的功能，但是如果发生了运行时异常，它照样可以关闭文件释放资源。

```
# 不使用 with 关键字
f = open('file.txt', 'r')
try:
    data = f.read()
except FileNotFoundError:
    print("文件未找到，检查文件路径。")
finally:
    # 需要手动关闭文件
    f.close()
```

```
# 使用 with 关键字
with open('file.txt', 'r') as f:
    data = f.read()
```

```
try:
    with open('file.txt', 'r') as f:
        data = f.read()
except FileNotFoundError:
    print("文件未找到，检查文件路径。")
```

Python读写文件

□ 使用with

#代码2.30 将数据写入文件

with open("data.txt", "w") as file:

#写入单个字符串

file.write("人生苦短, 我用Python. \n")

poem = [

"静夜思 (唐 李白)\n",

"床前明月光, \n",

"疑是地上霜. \n",

"举头望明月, \n",

"低头思故乡. \n"

]

#写入容器对象中的字符串

file.writelines(poem)

#代码2.31 从文件中读取数据

with open("data.txt", "r") as file:

#从文件中读取所有内容

str1 = file.read()

print("str1 =", repr(str1))

#将读取位置恢复到文件的首部

file.seek(0)

#从文件中读取一行内容

str2 = file.readline()

print("str2 =", repr(str2))

file.seek(0)

#从文件中读取所有内容到列表中

str3 = file.readlines()

print("str3 =", repr(str3))

Python面向对象程序设计

□ 面向对象程序设计中，先定义类再使用该类实例，**关键字：class**

```
# 代码2.32 shop.py定义一个类，用来描述水果商店
```

```
class FruitShop:
```

```
    def __init__(self, name, fruitPrices):
```

```
        self.fruitPrices = fruitPrices
```

```
        self.name = name
```

```
        print(f'Welcome to {name} fruit shop')
```

```
    def getCostPerPound(self, fruit):
```

```
        if fruit not in self.fruitPrices:
```

```
            return None
```

```
        return self.fruitPrices[fruit]
```

```
    def getPriceOfOrder(self, orderList):
```

```
        totalCost = 0.0
```

```
        for fruit, numPounds in orderList:
```

```
            costPerPound = self.getCostPerPound(fruit)
```

```
            if costPerPound != None:
```

```
                totalCost += numPounds * costPerPound
```

```
        return totalCost
```

```
    def getName(self):
```

```
        return self.name
```

```
#2.33使用自定义的类创建对象
```

```
import shop
```

```
shopName = 'the Berkeley Bowl'
```

```
fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}
```

```
berkeleyShop = shop.FruitShop(shopName, fruitPrices)
```

```
applePrice = berkeleyShop.getCostPerPound('apples')
```

```
otherName = 'the Stanford Mall'
```

```
otherFruitPrices = {'kiwis':6.00, 'apples': 4.50, 'peaches': 8.75}
```

```
otherFruitShop = shop.FruitShop(otherName, otherFruitPrices)
```

```
otherPrice = otherFruitShop.getCostPerPound('apples')
```

```
print(otherPrice)
```

```
print(f'Apples cost ${ otherPrice : .2f} at { otherName }.')
```

```
print("My, that's expensive!")
```

Python类的继承

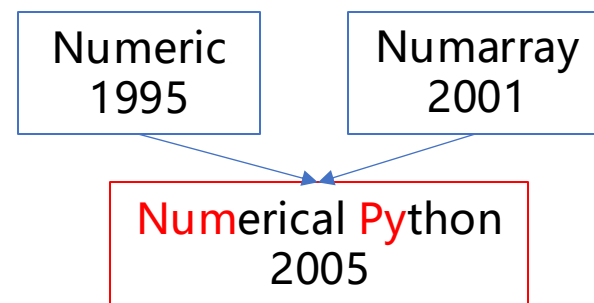
- 面向对象程序设计的一个非常强的优势是在代码复用方面，通过扩展或修改一个已经定义好的类来建立新的类，不需要将已有类的功能再次实现，这种机制成为**继承** (Inheritance)
- `super()` 函数是用于调用父类（超类）的一个方法

```
#代码2.34 类的继承实例
class Dog:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print(f"I am {self.name}.")
class BarkingDog(Dog):
    def bark(self):
        print("汪汪汪~~")
dog = BarkingDog("旺财")
dog.greet()
dog.bark()
```

数值计算库NumPy

□ NumPy是一个用于数值计算的第三方模块

- ✓ 创建数组或矩阵
- ✓ 对数组或矩阵对象进行函数运算
- ✓ 完成数值积分或线性代数的运算
- ✓



□ 安装命令: `pip install numpy`

NumPy多维数组创建

- 通过NumPy模块中的`array()`函数可以依据列表、元组、或其他序列类型对象中的数据，创建数组（ndarray）对象，ndarray对象和列表不一样，其中元素的数据类型必须保持一致

```
#代码 2.35 array()函数创建数组
```

```
import numpy as np
```

```
#创建1维数组
```

```
arr1 = np.array([1,2,3])
```

```
#创建2维数组
```

```
arr2 = np.array([[1,2],[3,4],[5,6]])
```

```
#查看数组的元素类型
```

```
print(f"{arr2.dtype}")
```

```
#以指定的数据类型创建数组
```

```
arr3 = np.array([1,2,3], dtype='float64')
```

```
#数据类型float64指的是64位的双精度浮点数
```

```
print(f"{arr3.dtype}")
```

```
#代码 2.36 创建特征数组对象
```

```
#使用arange()函数创建连续数值构成的数组，
```

```
#三个参数分别表示起始值、终止值（不包含）、步进值
```

```
print(f"{np.arange(10, 20, 2)=}")
```

```
print(f"{np.arange(0.1, 1, 0.2)=}")
```

```
#使用linspace()函数创建等差数列数组，
```

```
#三个参数分别表示起始值、终止值、元素数量
```

```
print(f"{np.linspace(0, 1, 5)=}")
```

```
#使用logspace()函数创建等比数列数组，
```

```
#前两个参数分别是以10的幂表示的起始值和终止值
```

```
print(f"{np.logspace(0, 1, 5)=}")
```

```
print(f"{np.zeros([3, 2])=}")#使用zeros()函数创建全零数组
```

```
print(f"{np.ones([2, 3])=}") #使用ones()函数创建全1数组
```

```
print(f"{np.identity(3)=}") #使用identity()函数创建对角矩阵
```

```
print(f"{np.diag([1, 2, 3, 4])=}") #使用diag()函数创建对角线数组
```

NumPy数组常用属性和数据类型转换

□ NumPy创建好的ndarray数组对象具有一系列有用的属性

```
#代码2.37 ndarray对象的属性
import numpy as np
arr1 = np.array([[1,2,3],[4,5,6]])
print(f"{arr1.dtype=}") #dtype属性表示元素的类型
print(f"{arr1.ndim=}") #ndim属性表示数组的维度
print(f"{arr1.shape=}") #shape属性表示数组每一维的大小
print(f"{arr1.size=}") #size属性表示数组元素的数量

arr1.shape = 3,2 #通过设置shape属性, 可以改变数组元素的排列
print(f"After shape: {arr1.shape=}")

#astype()方法可以按照指定的类型得到新数组
arr2 = arr1.astype(np.float64)
print(f"After astype: {arr1.dtype=}, {arr2.dtype=}")
```


NumPy生成随机数

□ NumPy中包含了random模块，可生成随机数

#代码2.38 random模块中函数举例

```
import numpy as np
arr1 = np.arange(15)
np.random.shuffle(arr1) #将有序的数组随机打乱
```

```
#choice()函数用于在样本中随机进行抽取
print(f"{np.random.choice(arr1, size=(2,3))}")
```

```
#randint()函数用于在指定的范围内构成随机整数数组
print(f"{np.random.randint(10, 20, size=(3,3))}")
```

```
#permutation()函数用于直接生成一个乱序数组
print(f"{np.random.permutation(10)}")
```

```
#normal()函数用于产生正态分布的随机数，
#第一个参数是期望值，第二个参数是标准差
print(f"{np.random.normal(100, 10, size=(2,3))}")
```

#uniform()函数用于产生均匀分布的随机数，前两个参数分别是区间的起始值和终止值

```
print(f"{np.random.uniform(10, 20, size=(2,3))}")
```

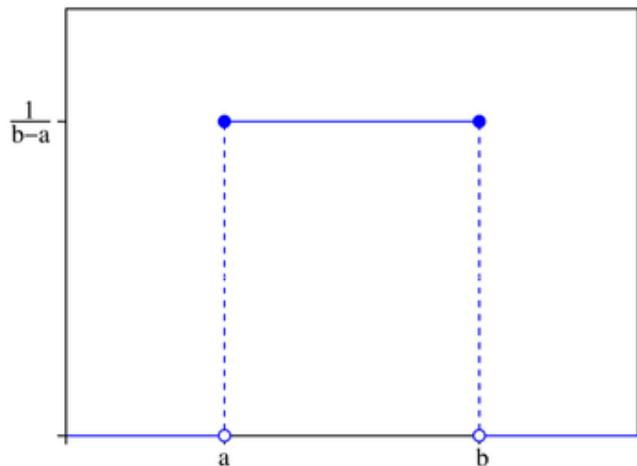
```
#poisson()函数用于产生泊松分布的随机数，
#第一个参数用于指定λ系数
print(f"{np.random.poisson(2.0, size=(2,3))}")
```

```
#beta()函数用于产生beta分布的随机数
print(f"{np.random.beta(1, 3, size=(2,3))}")
```

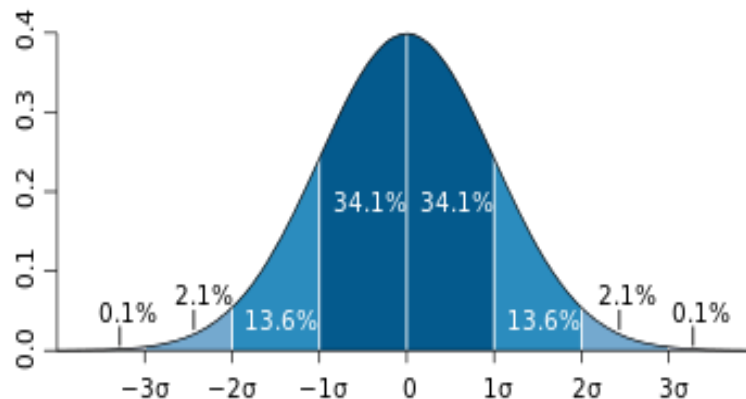
```
#chisquare()函数用于产生卡方分布的随机数，
#第一个参数表示自由度数
print(f"{np.random.chisquare(3, size=(2,3))}")
```

```
#gamma()函数用于产生gamma分布的随机数
print(f"{np.random.gamma(2, 2, size=(2,3))}")
```

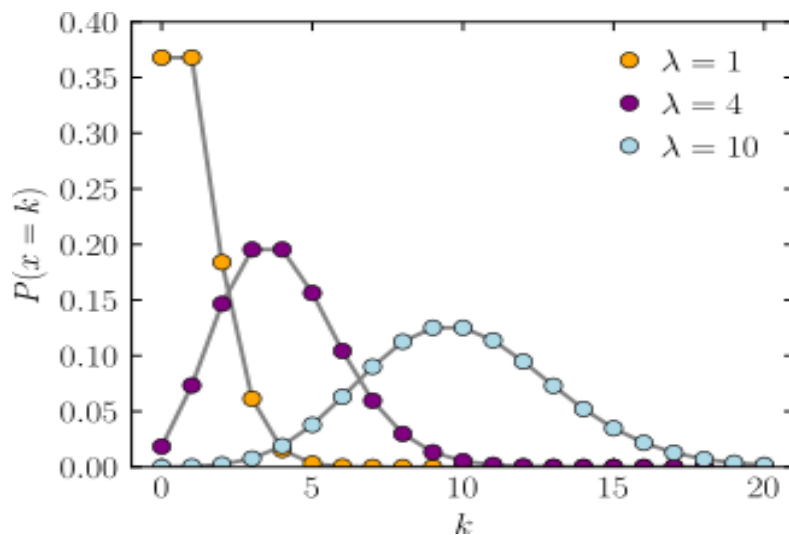
不同的分布



均匀分布



正态分布：主要用于误差分析



泊松分布：单位时间内事件发生的次数的概率分布
 λ 是单位时间内随机事件的平均发生率

NumPy数组变换

- 通过`reshape()`方法改变其数组维度,
- 数据散开`ravel()`方法
- 数据扁平化`flatten()`方法 (分配了新的内存)

#代码2.39 改变数组维度和数据散开示例

```
import numpy as np
```

```
arr1 = np.arange(12)
```

```
arr2 = arr1.reshape(4,3)
```

#参数-1表示该维度的元素个数通过元素总数进行推算

```
arr3 = arr1.reshape(2,-1)
```

#arr4是arr3进行数据散开后的结果

```
arr4 = arr3.ravel()
```

NumPy数组合并与分割

- `hstack()`、`vstack()`和`concatenate()`: 数组合并
- `hsplit()`、`vsplit()`和`split()`: 分别实现数组分隔
- 使用`transpose()`方法和T属性实现矩阵转置

#代码2.40 数组合并示例

```
arr1 = np.arange(4).reshape(2,2)
arr2 = np.arange(4,8).reshape(2,2)
```

```
print(f"{np.hstack([arr1,arr2])}")
```

```
print(f"{np.vstack([arr1,arr2])}")
```

#`concatenate()`函数的参数`axis`为1时表示横向合并,
#为0时表示纵向合并

```
print(f"{np.concatenate([arr1,arr2], axis=1)}")
```

```
print(f"{np.concatenate([arr1,arr2], axis=0)}")
```

#代码2.41 数组分割示例

```
arr1 = np.arange(16).reshape(4,4)
```

```
print(f"{np.hsplit(arr1,2)}")
```

```
print(f"{np.vsplit(arr1,2)}")
```

#`split()`函数的参数`axis`为1时表示横向分割,
#为0时表示纵向分割

```
print(f"{np.split(arr1,2,axis=1)}")
```

```
print(f"{np.split(arr1,2,axis=0)}")
```

NumPy数组的索引和切片

- 一维数组的索引和切片方法与序列类型对象一样
- 多维数组各个维度的索引用逗号分隔即可

```
#代码2.43 一维数组的索引和切片示例
```

```
import numpy as np
arr1 = np.arange(10)

print(f"{arr1}")
print(f"{arr1[5]}")
print(f"{arr1[-2]}")
print(f"{arr1[2:7]}")
```

```
#代码2.44 多维数组的索引和切片示例
```

```
arr1 = np.arange(16).reshape(4,4)

#选取行索引值为1, 列索引值为2的元素
print(f"{arr1[1,2]}")

#选取行、列索引值均在[1,3)区间内的元素
print(f"{arr1[1:3,1:3]}")

print(f"{arr1[2,:]}") #选取索引值为2的那一行的所有元素
print(f"{arr1[:,2]}") #选取索引值为2的那一列的所有元素
print(f"{arr1[arr1 > 5]}") #选取元素值大于5的元素
```

NumPy数组的运算

□ 数组和标量间的运算，广播机制，条件逻辑运算

```
#代码 2.45 数组与标量间的运算示例
#使用循环结构完成10000次乘法运算
arr1 = np.arange(10000)
for i in range(len(arr1)):
    arr1[i] = arr1[i] * 5
print(f"计算完毕后数组的前5项为: {arr1[:5]}")
# 比较对列表*5之后的结果

#使用数组结构完成10000次与标量的乘法运算,
#其运行效率远远优于上述代码
arr1 = np.arange(10000)
arr1 = arr1 * 5
print(f"标量计算完毕数组前5项为: {arr1[:5]}")
```

```
arr1 = np.array([[0,0,0],[1,1,1],[2,2,2]])
arr2 = np.array([1,2,3])
print(f"{arr1=}")
print(f"{arr2=}")
print(f"{arr1 + arr2=}")
```

```
arr1 = np.array([[2,4],[6,7]])
arr2 = np.array([[1,5],[3,8]])
#数组arr3中的元素由arr1和arr2中对应位置
较大的元素构成
arr3 = np.where(arr1 > arr2, arr1, arr2)
print(f"{arr1=}")
print(f"{arr2=}")
print(f"{arr3=}")
```

NumPy数组的读写操作

- 读/写文本文件
- 读/写二进制文件

#代码2.49 NumPy模块中的文本文件读写示例

```
import numpy as np
```

```
arr1 = np.arange(12).reshape(4,3)
```

```
#将数组arr1中的数据写入文本文件arr.txt,
```

```
#数据之间默认以空格分隔, 此处指定以逗号分隔数据
```

```
np.savetxt("arr.txt", arr1, delimiter=',')
```

```
#将文本文件arr.txt中的数据读取出来并与变量arr2关联,
```

```
#并指定以逗号作为数据之间的分隔符
```

```
arr2 = np.loadtxt("arr.txt", delimiter=',')
```

```
print(f"{arr2=}")
```

#代码2.48 NumPy模块中的二进制文件读写示例

```
#将数组arr1保存至arr.npy文件中
```

```
np.save("arr.npy", arr1)
```

```
#将arr.npy中的数据读取出来
```

```
arr2 = np.load('arr.npy')
```

二进制文件的存储、读取更快, 但是不能追加写

NumPy数组排序

- 使用NumPy中数组对象自带的sort()方法与argsort()方法可以轻松地完成数组的排序运算

#代码2.50 使用sort()方法对数组进行排序示例

```
import numpy as np
```

```
arr1 = np.random.randint(10,20,size=10)
```

```
arr1.sort()
```

#数组的sort()方法可以使用指定维度上的元素进行排序

```
arr2 = np.random.randint(10,20,size=(4,4))
```

```
arr2.sort(axis=0)
```

```
arr2.sort(axis=1)
```

#代码2.49 使用argsort()方法对数组排序示例

```
arr2 = arr1.argsort()
```

```
print(f"arr1 中最小元素的索引值为 :  
{arr2[0]=},arr1 中最大元素的索引值为 :  
{arr2[-1]=}")
```


NumPy中的sort函数

```
@array_function_dispatch(_sort_dispatcher)
def sort(a, axis=-1, kind=None, order=None):
    """
    Return a sorted copy of an array.

    Parameters
    -----
    a : array_like
        Array to be sorted.
    axis : int or None, optional
        Axis along which to sort. If None, the array is flattened before
        sorting. The default is -1, which sorts along the last axis.
    kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
        Sorting algorithm. The default is 'quicksort'. Note that both 'stable'
        and 'mergesort' use timsort or radix sort under the covers and, in general,
        the actual implementation will vary with data type. The 'mergesort' option
        is retained for backwards compatibility.

    .. versionchanged:: 1.15.0.
        The 'stable' option was added.

    order : str or list of str, optional
        When `a` is an array with fields defined, this argument specifies
        which fields to compare first, second, etc. A single field can
        be specified as a string, and not all fields need be specified,
        but unspecified fields will still be used, in the order in which
        they come up in the dtype, to break ties.

    Returns
    -----
    sorted_array : ndarray
        Array of the same type and shape as `a`.
```

数据去重和重复数据

- NumPy中unique()函数用于剔除数组的重复元素
- NumPy中tile()函数或repeat()函数完成复制运算

#代码2.52 使用unique函数对数组进行去重
示例

```
import numpy as np
arr1 = np.random.randint(10,20,size=10)

print(f"{np.unique(arr1)}")
```

#代码2.53 使用tile()函数或者repeat()函数对数组进行重复示例

```
import numpy as np
arr1 = np.arange(6).reshape(2,3)
print(f"{np.tile(arr1,3)}")
```

#repeat()函数的使用与tile()函数类似,
#其参数axis用于指定重复的维度

```
print(f"{np.repeat(arr1,2,axis=0)}")
print(f"{np.repeat(arr1,2,axis=1)}")
```

NumPy常用统计函数

- NumPy中提供了很多统计分析函数，常见的有sum()、mean()、std()、var()、min()和max()函数等，需要注意在参数中指定其运算的维度

```
#代码2.54 NumPy中的常用统计函数示例
```

```
import numpy as np  
arr1 = np.arange(10).reshape(2,5)
```

```
#求和运算
```

```
print(f"{np.sum(arr1)=}")  
print(f"{np.sum(arr1,axis=0)=}")  
print(f"{np.sum(arr1,axis=1)=}")
```

```
#计算平均值的运算
```

```
print(f"{np.mean(arr1)=}")  
print(f"{np.mean(arr1,axis=0)=}")  
print(f"{np.mean(arr1,axis=1)=}")
```

```
#计算标准差的运算
```

```
print(f"{np.std(arr1)=}")  
print(f"{np.std(arr1,axis=0)=}")  
print(f"{np.std(arr1,axis=1)=}")
```

Python的缺点

□ 运行速度慢

- ✓ Python是解释型语言，并且屏蔽了很多底层细节
- ✓ 一般慢于C/C++，也慢于JAVA

□ 加密困难

- ✓ 可以通过Cpython等编译
- ✓ 代码混淆
- ✓ Pyexe

□ 复杂问题

- ✓ 解决复杂问题比较吃力，动态语言无法在编译甚至编写阶段就能检查出各种不一致的问题，也无法屏蔽内部信息的可见性

如何为项目选择合适的语言

- 团队成员最熟悉什么语言
- 有没有计算开销大的操作
- 是否会涉及很多子流程和文件管理
- 有紧张的资源限制吗？比如嵌入式系统
- 需要支持什么平台
- 是否涉及特定的领域

总结

- Python语言的基础编程知识
- Python的基本语法
- 各种类型对象的使用方法
- 程序的流程控制结构
- 定义和调用函数的方法
- 使用第三方科学计算模块NumPy进行数组运算和数据统计的基本方法

- 作业: Spoc第二章 (DDL: 见课程主页)

import this

Python 的 “Zen of Python”（Python 之禅）。这是一系列的原则和指导方针，旨在描述 Python 语言的设计哲学和哲学观点。

这些原则是由 Python 社区的核心开发者所制定的，它们强调了代码的可读性、简洁性和清晰性，以及编程时的一些最佳实践。通过运行 `import this`，您可以看到这些原则的列表。

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --
obvious way to do it.
Although that way may not be obvious at first unless
you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a
good idea.
Namespaces are one honking great idea -- let's do
more of those!

Python 之禅

优美优于丑陋,
明了优于隐晦;
简单优于复杂,
复杂优于凌乱,
扁平优于嵌套,
稀疏优于稠密,
可读性很重要!
即使实用比纯粹更优,
特例亦不可违背原则。
错误绝不能悄悄忽略,
除非它明确需要如此。
面对歧义,
拒绝妄加猜测。
任何问题应有一种,
且最好只有一种,
显而易见的解决方法。
尽管这方法一开始并非如此直观,
除非你是荷兰人。
不要一直拖延,
也不要不假思索。
难以解释的, 必然是坏方法。
很好解释的, 可能是好方法。
命名空间是个绝妙的主意,
我们应好好利用它。