分支结构、函数和模块

主要内容

- **分支结构**
  - 比较运算符
  - 单分支/双分支/多分支结构
  - 逻辑运算符
- 函数
  - 函数定义和调用
  - 函数参数
  - 新的格式化方法
  - 名字空间
- 模块和import语句
- 编程风格
- 内置数学函数和math模块

比较(关系）运算符

注意： == != <= >= 运算符中要求之间**不能有空格**

- 比较运算符运算的结果为True或False（bool类型）
- 对象的值是否相等：
  - x == y, **x != y**: 等于、不等于
  - 两个对象可以是不同类型
- 两个对象是否同一个对象
  - x **is** y, x **is not** y：等价于判断id(x) == id(y), id(x) != id(y)
    - 不建议 not x is y
  - 两个对象可以是不同类型
- 判断数值类型的对象的值的大小：
  - 对象都是数值类型(不包括复数，复数间无法比较大小）时可以比较
  - x < y, x <= y, x > y, x >= y： 小于、小于等于、大于、大于等于
- **比较运算符**可以连用，连用时不考虑传递性，仅仅**相邻数之间的比较**
  - x < y < z 表示 x < y 且 y < z
  - x < y > z 表示 x < y 且 y > z
  - x > y is z 表示 x > y且y is z ,即 x > y and y is z

```
>>> x, y, z = 12, 18, 13
>>> x < y, x <= y, x > y, x >= y      # (True, True, False, False)
>>> x < y < z,  x < y > z      # (False, True)
```

- **先算后比**：算术运算符的优先级相比比较运算符优先级更高
- 所有比较运算符的优先级相同

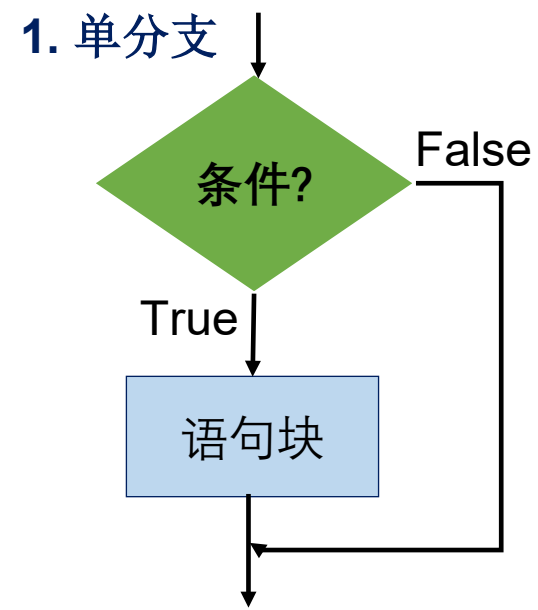
```
>>> 4 ** 2 + 8 ** 2 <  5 ** 2 + 7 ** 2
False
```

条件执行

- if 后面为**条件表达式**，**行尾的冒号**表示代码块的开始
- 有些语言要求条件表达式用括号包含，python并不需要
- 顺序执行代码块中的代码，代码块**整体缩进相同的位置**
- **结束缩进**回到if语句开始的位置，表示代码块的结束
- 建议以**4个空格**为基本缩进单位，不建议采用制表符缩进，更不能混合使用

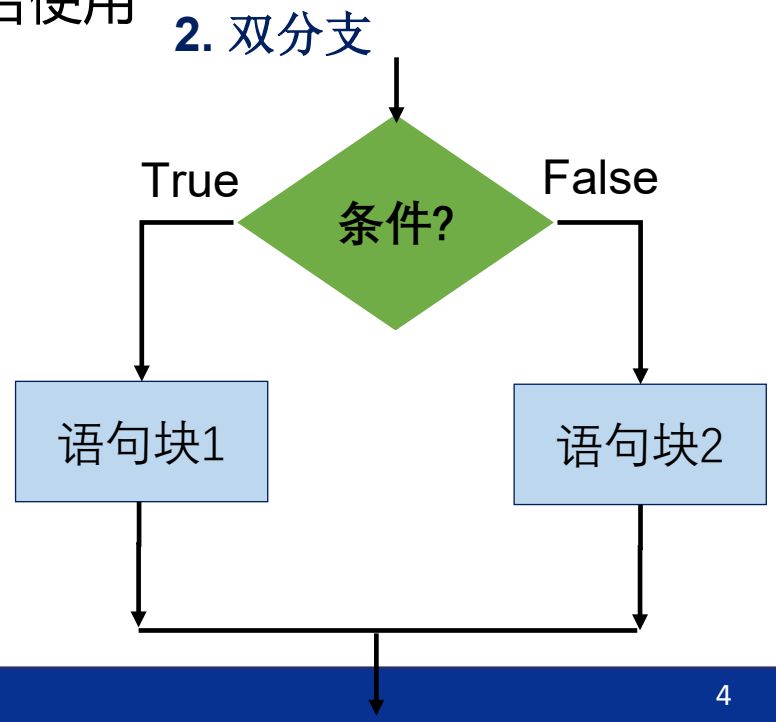
```
if expr:
    statement
    ...
    statement

if expr:
    statement(s)
else:
    statement(s)
```




```
gpa = float(input("gpa? "))
if gpa >= 2.0:
    print("Application accepted.")
```

```
gpa = float(input("gpa? "))
if gpa >= 2.0:
    print("Welcome to Mars University!")
else:
    print("Application denied.")
```



条件表达式

- 出现在if/while中作为决定分支走向或循环结束的条件的条件表达式expr
  - 一般包括比较运算符，运算的结果是True/False
  - 也可以是任何表达式，会进行真值判断 bool(expr)
  - 一个对象一般真值判断为真，除非：
    - 如果该对象包含了__bool__方法，由调用该方法的返回值确定
    - 如果该对象包含了__len__方法，调用该方法，如果返回0，则真值判断为假
  - **真值判断为False的对象：False/None/值为0的数值对象/长度为0的空序列对象(比如空字符串)**
  - 真值判断为True的对象：True/**非0数值对象、非空序列对象**

```
if __name__ == '__main__':
    main()
if 2:  # 相当于 if bool(2)
    print('真值判断为真'))
```

```
if i != 0:
    print(i, '!=0')

if len(s) != 0:
    print(s)
```

```
# pythonic version
if i:
    print(i, '!=0')

if s:
    print(s)
```

真值判断为

假：0 0.0 0j '' [] () set() {} range(0)

真：1 -5 'abc' [1,2] (1,) {one:'one'}

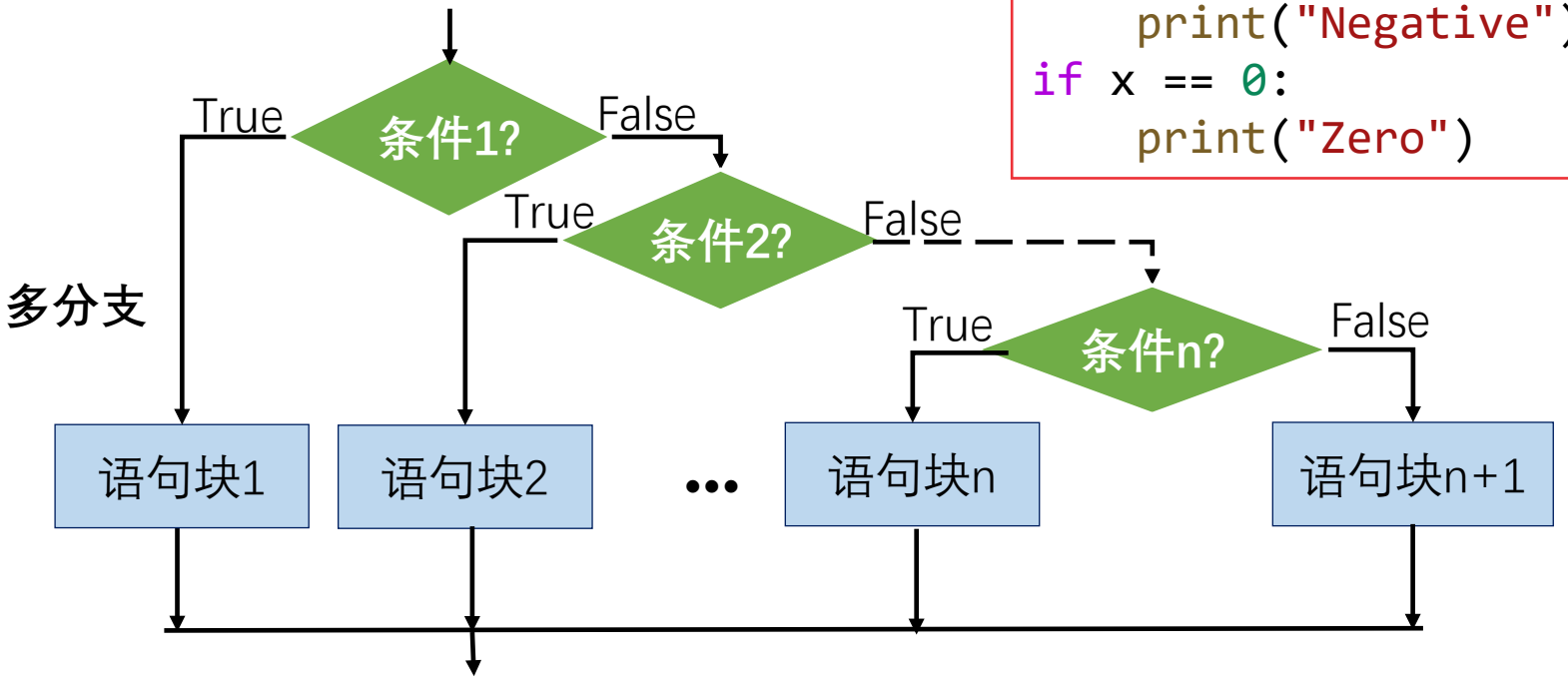
print

if/else语句嵌套

- 如果条件表达式之间是独立的且可以执行语句块的任意组合时，可以采用顺序的多个if语句

```
a = 4; b = 5; c = 7
if a % 2:
    print(a, '为奇数')
if b % 2:
    print(b, '为奇数')
if c % 2:
    print(c, '为奇数')
```

- 如果要根据条件表达式决定执行多个语句块中的其中某1个或0个（不包含else子句），则采用多分支结构
  - 关键字elif是else if的缩写
  - 条件i 实际上等于 前面的条件都为假 再加上条件i




```
if x > 0:
    print("Positive")
if x < 0:
    print("Negative")
if x == 0:
    print("Zero")
```

branch.py

```
if x > 0:
    print("Positive")
else:
    if x < 0:
        print("Negative")
    else:
        if x == 0:
            print("Zero")
```

```
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
elif x == 0:  # else:
    print("Zero")
```

if/else语句嵌套与多分支结构

```
if  not (0 <= percent <= 100):
    raise AssertError('invalid percent')
```

- 如果条件表达式之间是独立的且可以执行语句块的任意组合时，可以采用顺序的多个if语句
- 如果要根据条件表达式决定执行多个语句块中的其中某1个或0个，则采用多分支结构

```
percent = float(input("What percentage did you earn? "))

if percent >= 90:
    print("You got an A!")

if percent >= 80:
    print("You got a B!")

if percent >= 70:
    print("You got a C!")

if percent >= 60:
    print("You got a D!")

if percent < 60:
    print("You got an F!")
```

```
percent = float(input("What percentage did you earn? "))
assert 0 <= percent <= 100, 'invalid percent'

if percent >= 90:
    print("You got an A!")
else:
    if percent >= 80:
        print("You got a B!")
    else:
        if percent >= 70:
            print("You got a C!")
        else:
            if percent >= 60:
                print("You got a D!")
            else:
                if percent < 60:  # 一定小于60
                    print("You got an F!")
```

branch.py

if/else语句嵌套与多分支结构

- 多分支结构中，条件i 实际上等于 前面的条件都为假 再加上条件i
- 注意条件表达式的顺序，下面的例子中如果percent >= 60(+分支）与>= 90调换，则60分以上都是等级D

```
percent = float(input("What percentage did you earn? "))
assert 0 <= percent <= 100, 'invalid percent'

if percent >= 90:
    print("You got an A!")
elif percent >= 80:
    print("You got a B!")
elif percent >= 70:
    print("You got a C!")
elif percent >= 60:
    print("You got a D!")
else: # percent < 60
    print("You got an F!")
```

branch.py

```
if  percent >= 60:
    print("You got a D!")
elif percent >= 80:
    print("You got a B!")
elif percent >= 70:
    print("You got a C!")
elif percent >= 90:
    print("You got an A!")
else: # percent < 60
    print("You got an F!")
```

选择结构常见错误：错误缩进

```
import math
radius = -20
if radius >= 0:
    area = radius * radius * math.pi
print("The area is", area)
```

```
import math
radius = -20
if radius >= 0:
    area = radius * radius * math.pi
    print("The area is", area)
```

```
i = 1
j = 2
k = 3
if i > j:
    if i > k:
        print('A')
    else:
        print('B')
```

缩进不一样，执行的结果也会不一样!!

左边没有输出
右边打印B

```
i = 1
j = 2
k = 3
if i > j:
    if i > k:
        print('A')
else:
    print('B')
```

python不允许在条件表达式中出现赋值运算 =

```
>>> if a=3:pass
SyntaxError: invalid syntax
```

拓展：python3.8引入assignment expression or named expression

```
if a:=3:
    print(a)
```

if/else结构的代码重构

- 将公共的代码移出if/else结构以删除冗余

branch.py

```
if money < 500:
    print("You have $", money, "left.")
    print("Cash is dangerously low. Bet carefully.")
    bet = int(input("How much do you want to bet?"))
elif money < 1000:
    print("You have $", money, "left.")
    print("Cash is somewhat low. Bet moderately.")
    bet = int(input("How much do you want to bet?"))
else:
    print("You have $", money, "left.")
    print("Cash is in good shape. Bet liberally.")
    bet = int(input("How much do you want to bet?")
```

```
print("You have $", money, "left.")
if money < 500:
    print("Cash is dangerously low. Bet carefully.")
elif money < 1000:
    print("Cash is somewhat low. Bet moderately.")
else:
    print("Cash is in good shape. Bet liberally.")
bet = int(input("How much do you want to bet?"))
```

逻辑(布尔)运算符

```
if expr1:
    expr = expr2
else:
    expr = expr1   # expr1真值为False
```

expr1为真时expr2,否则 expr1

and :逻辑与 expr1 and expr2 and expr3　比如n>= 0 and n < 10

- and运算采用左结合律，即从左到右进行运算
- 只有**所有条件(表达式)真值判断为真，**最终真值判断结果才为**真值**
- 短路逻辑(short-circuit logic) 或惰性求值(lazy evaluation)
  - 如果前面真值判断为假，不管后面条件(表达式)怎样，最终也是假值。后面表达式不需要计算
- 注意与其他语言不同，and/or运算得到的结果并不一定是True或者False，而是某个表达式的值(用于if/while中的条件时再进行真值判断)
  - and 得到的是最后一个决定真值判断的表达式的值，即第一个假的表达式（已经知道最终为假值)或者最后一个表达式（最终的真值判断由最后一个表达式决定）

bool(expr1)	bool(expr2)	bool(expr1 and expr2)	**expr1 and expr2**
True	True	True	**expr1为真, 取值为expr2**
True	False	False	**expr1为真, 取值为expr2**
False	True	False	**expr1为假, 取值为expr1**
False	False	False	**expr1为假, 取值为expr1**

```
>>> 4 and 5
5
>>> 4 and 0
0
>>> 0.0 and 5
0.0
>>> '' and 5
''
```

and : 返回第一个假**(None、空或者数值0)**的表达式或者最后一个表达式

逻辑与运算示例

```
# 用户输入一个[0, 10)范围内的整数，如果正确great，否则wrong
number = int(input('please input a number[0, 10)'))
if number >= 0:
    if number < 10:
        print('Great!')
    else:
        print('Wrong!')
else:
    print('Wrong!')

if number >= 0 and number < 10: # 0 <= number < 10
    print('Great!')
else:
    print('Wrong!')
```

conditional.py

短路逻辑: 如果前面的表达式已经确定了最终真值判断的结果，则后面的表达式无需计算

```
x = 40
if (x > 10 and
    input('Print Value(y/n)?') == 'y'):
    print(x)

if x != 0 and 100 / x > 2:
    print('Great')
```

逻辑(布尔)运算符

- **and :逻辑与运算** expr1 and expr2 and expr3
  - **返回第一个假(None、空或者数值0)的表达式或者最后一个表达式**
- **or: 逻辑或运算 expr1 or expr2 or expr3**
  - 采用左结合律，即从左到右进行运算
  - 只要其中任一条件真值判断为真，则最终真值判断为真值，否则为假值
  - 短路逻辑：前面条件如果为真，后面的表达式不需要进行计算
  - **返回第一个真（非None、非空或者非0）的表达式或者最后一个表达式**

bool(expr1)	bool(expr2)	bool(expr1 or expr2)	expr1 or expr2
True	True	True	**expr1为真, expr1**
True	False	True	**expr1为真, expr1**
False	True	True	**expr1为假, expr2**
False	False	False	**expr1为假, expr2**

```
if expr1:
    expr = expr1
else:
    expr = expr2   # expr1真值为False
```

对象为空对象(字符串、列表等)时采用缺省值

```
name = input('Please enter your name: ') or '<unknown>'
```

逻辑(布尔)运算符

- **not: 逻辑非** not expr
  - expr真值判断为真值时结果为False， expr真值判断为假值时结果为True
  - True: not False, not 0, not [], not (), not '' not range(0)
  - False: not True, not -1, not [1], not (1,), not 'hello' not range(1)
- **if/else三元运算符:** value1 if condition else value2

当条件表达式condition的真值判断为True时，表达式的值为value1，否则值为value2

- 短路逻辑：不会同时对value1 和value2 求值
- 其他语言的三元表达式的语法一般为 condition?value1:value2

```
>>> x = 4
>>> x ** 2 if x % 2 == 0  else x ** 3
16
```

```
if condition:
  expr = value1
else:
  expr = value2
```

- 一元的not优先级最高，and第二，or优先级次之，三元的ifelse优先级最低
  - expr1 and expr2 or expr3 相当于 (expr1 and expr2) or expr3
  - not expr1 and not expr2 相当于 (not expr1) and (not expr2)

运算符优先级

- 优先级越低，越最后计算
- 除了**为右结合外，其他运算符都是左结合
- lambda < if-else < 逻辑运算符(or,and,not) < 比较运算符 < 两元位运算 < 算术运算 < 一元运算 < 求幂运算 < 下标、切片、函数调用、字面量定义等
- 赋值看成运算符时其优先级最低，采用右结合方式
- 首先定义对象，然后才可以运算 →字面量优先级最高
- 函数调用、下标和切片、属性等返回对象 →优先级次之
- 有了对象，可以进行计算→算术运算符次之，一元的优先级更高
- 计算后可以进行比较 →比较运算符比算术运算符更低
- 比较以及成员判断的结果为True/False，可以进行逻辑运算→逻辑运算比比较运算符低，单目逻辑运算符(not)更高
- 三元运算符及lambda表达式优先级最低

从低到高

运算符	描述
lambda	lambda表达式
if-else	3元条件表达式
or	布尔“或”
and	布尔“与”
not x	布尔“非”，
in，not in	成员测试，与下面同一性和比较相同优先级
is，is not	同一性测试
<，<=，>，>=，!=，==	比较
|	按位或
^	按位异或
&	按位与
<<，>>	移位
+，-	加法与减法
*，/，//，%	乘法、除法与取余
+x，-x，~x	正负号，按位求反，一元运算或单目运算
**	求幂
x.attribute	属性
x[index] x[index:index]	下标和切片
f(arguments...)	函数调用
(experession,...)	元组字面量
[expression,...]	列表字面量
{key:datum,...}	字典字面量
'expression,...'	字符串字面量

逻辑运算符使用技巧

- 在设计条件表达式时，若能大概预测不同条件失败的概率或者根据表达式运算的开销，可将多个条件根据"and"和"or"运算的**短路特性**进行组织，提高程序运行效率
  - 表达式运算开销小的应该在前面进行求值
  - and运算时失败概率高，or运算时成功概率高的表达式应该在前面求值
- **德摩根定理(De Morgan's Law)**

```
not (expr1 and expr2) = not expr1 or  not expr2
not (expr1 or  expr2) = not expr1 and not expr2
```

conditional.py

```
i = 36
if i % 17 != 2 or i % 13 != 10:
    print(i, "不满足：被17整除余2且被13整除余10")
else:
    print(i, "被17整除余2且被13整除余10")
```

```
i = 36
if not (i % 17 == 2 and i % 13 == 10):
    print(i, "不满足：被17整除余2且被13整除余10")
else:
    print(i, "被17整除余2且被13整除余10")
```

```
if expr1 and expr2:
    blah blah...
else:
   pass
```

可以改写为：

```
if not expr1 or not expr2:
    pass
else:
    blah blah...
```

逻辑运算符使用技巧

- 分配律(distributive law)

expr1 and (expr2 or expr3) = (expr1 and expr2) or (expr1 and expr2)

expr1 or (expr2 and expr3) = (expr1 or expr2) and (expr1 or expr3)

```
(male == 'M' and age > 75) or (male == 'M' and age < 12)
male == 'M' and (age > 75 or age < 12)
```

问题：判断某一年是否闰年?

- 能被400整除
- 或者能被4整除，但不能被100整除

```
if year % 400 == 0 or year % 4 == 0 and year % 100 != 0:
    leap = True
else:
    leap = False
```

```
leap = year % 400 == 0 or year % 4 == 0 and year % 100 != 0
```

conditional.py

```
def is_leap_year(year):
    return year % 400 == 0 or year % 4 == 0 and year % 100 != 0
 # return year % 400 == 0 or (year % 4 == 0 and year % 100 != 0)
```

逻辑运算符使用常见错误

- if expr == value1 or expr == value2 or expr == value3:

错误写成 if expr == value1 or value2 or value3:

- 类似地： **if expr < value1 or expr < value2**:

错误写成:

if expr < value1 or value2 :

优先级：相比再逻辑运算，相当于 if (expr < value) or value2

如果expr >= value1则 value2的真值作为条件表达式的真值

if expr < (value1 or value2): # 首先求解value1 or value2，然后与expr比较

主要内容

- 分支结构
  - 比较运算符
  - 单分支/双分支/多分支结构
  - 逻辑运算符
- **函数**
  - 函数定义和调用
  - 函数参数
  - 新的格式化方法
  - 名字空间
- 模块和import语句
- 编程风格
- 内置数学函数和math模块

函数

- 在实际开发中，有许多操作是完全相同或非常相似的，仅仅是要处理的数据不同
  - 通过将可能需要反复执行的代码封装为函数，实现代码的复用，保证代码的一致性
- 问题分解：将问题分解为可管理的部分并单独解决每个部分来构建问题的解决方案
  - 每个函数解决一个相对独立的问题，不要太复杂
- 首先声明(定义)函数：**描述**了要解决某个问题需要执行的语句集合
- 调用函数：**执行**函数中包含的语句

```
print("+------+")
print("|      |")
print("|      |")
print("+------+")
print()
print("+------+")
print("|      |")
print("|      |")
print("+------+"
```

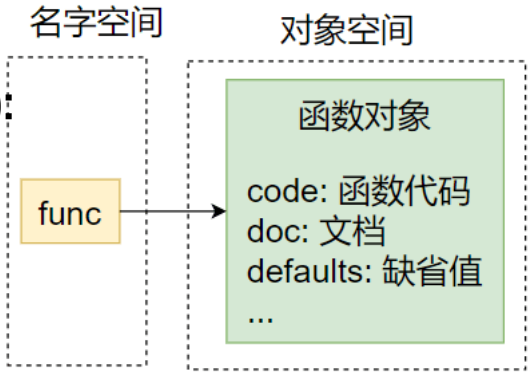
```
def print_box():
    print("+------+")
    print("|      |")
    print("|      |")
    print("+------+")

print_box()
print()
print_box()
```

```
def is_leap_year(year):
    return year % 400 == 0 or year % 4 == 0 and year % 100 != 0
```

函数定义

```
def func_name([parameters]):
    '''docstring'''  # optional
    function body
```



- def语句定义(define)函数：
  - 函数名
  - 函数的参数：
    - 可为0个参数，可为1或多个参数，参数之间以逗号(,)隔开
    - 在函数调用时才会知道该参数具体所指向的对象
  - 函数体：给出了调用该函数时要执行的语句
    - 一般包括多行
      - 相对于def关键字必须保持缩进(建议4个空格)
      - 解除缩进到def同样的位置时表示函数体结束
    - 如果函数体比较简单(如只有一行语句)时，也可与def在同一行
  - **创建函数对象，保存函数的代码**
  - 当前名字空间**引入名字**对应该函数对象
  - 检查语法错误，存储相关信息。函数体中的代码并没有执行
  - 函数也是一等公民(First-class Citizen)

functions.py

```
def f1():
    print('f1')

def f1(): print('f1')

def f2():
    pass

def cubic(x):
    return x ** 3

def abs2(x, y):
    z = x + y
    if z >= 0:
        return z
    else:
        return -z

abs2(2, -4*6)
```

- 如果同一个函数名有多个定义时，在访问该名字时之前的最后一个函数定义有效
- 函数的函数体中必须有一个语句，可使用**pass语句**，它是空语句，表示什么也不做

函数定义：文档字符串

- 文档字符串：提供友好的提示和使用帮助
  - 如果一个**模块源程序**的第一个语句为字符串字面量，则该字符串会保存在**变量__doc__**中
  - 如果一个函数的**函数体**的第一个语句为字符串字面量，则该字符串会保存在**函数对象的__doc__属性**中
  - 注意作为函数体时，字符串表达式的最开始必须要缩进，后续行则并没有强制，但建议缩进

- help()函数传递的是**函数对象时**会显示函数对象的文档字符串，而传递的是**模块对象时**显示模块对象的文档字符串以及相关函数的文档字符串
  - 会将每行前面的空格类字符移走
- 集成开发环境的编辑器界面一般也会在calltip处显示相应的文档字符串

```
>>> import docstring
>>> help(docstring)
>>> help(docstring.distance)
```

```
>>> docstring.distance(
(x1, y1, x2, y2)
返回两点间的距离
```

docstring.py

```
#! /usr/bin/env python3

'''模块文档字符串__doc__:第一个语句为字符串表达式时'''

import math

def distance(x1, y1, x2, y2):
    """返回两点间的距离

    传递的参数:第一个点的X轴和Y轴坐标，第二个点的X轴和Y轴坐标
    """
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

if __name__ == '__main__':
    dist = distance(0,0, 4.5, 4.5)
    print('(0,0)与(4.5,4.5)间的距离为', round(dist, 2))
```

函数定义：return语句

- 函数体内可包含return语句： return [expression_list]
  - 在调用函数执行函数体的代码时，如果执行的是return语句，结束函数的执行并返回
  - 关键字return后的表达式计算后的结果作为函数的返回值
  - return后面如果没有表达式，则等价于return None
    - None的类型为NoneType， NoneType类型只有一个值None
- 函数体内可以没有return语句
  - 如果执行到函数体结束也无return语句，则等价于最后执行语句return None
- 在函数调用时顺序执行函数体内的代码，执行到return语句时或函数体最后时，结束执行该函数，返回return语句所指出的对象，将该返回值传递给调用者
- 除非在函数调用的过程中出现了异常，函数一定会有返回值，只是有的函数可能返回值为None

```
def max_(x, y):
    if x > y:
        return x
    else:
        return y
def max2_(x, y):
    if x > y:
        print(x)
    else:
        print(y)
```

functions.py

```
>>> max_(2, 4)
4
>>> max2_(2, 4)
4
>>> print(max_(2, 4))
4
>>> print(max2_(2, 4))
4
None
```

交互式console：如果执行的语句为表达式obj，且其的值**不为None**时调用**print(repr(obj))**

函数调用

```
def func_name([parameters]):
    '''docstring'''  # optional
    function body

func_name(arguments)
```

- 函数的三个基本要素： 传递的参数/完成的功能/ 返回值
- 函数调用：**func_name(arguments)**
  - 实参(argument) 应该与形参(parameter)对应，表示传递实参给出的对象到相应的形参
  - 为了避免side effect，函数调用时为本次调用创建一个新的名字空间（本地名字空间）
    - 变量属于某个名字空间，不同名字空间的变量名可以相同
  - 函数调用时向其传递实参，python采用**赋值传递(pass by assignment)**的策略
    - 形参变量=实参变量，即**(本次函数调用时的本地名字空间的）形参变量与(调用者所在名字空间的)实参变量**指向同一个对象。形参变量的使用范围为函数体
    - 如果传递过来的是**可变对象**，函数体内可以对该可变对象修改，需要**特别小心**
    - 如果传递过来的是**不可变对象**，在函数内部无法修改，则**不用担心**会不小心更改了该对象
  - 函数调用的时候，不是执行下一条语句，而是跳到函数体，然后**执行相应的函数体中的代码**，在结束或return语句时返回**回到调用的地方继续执行下一条语句**
  - 函数调用如同加法运算一样，也是**一种表达式**，其运算后得到的对象为执行函数体代码直到返回时通过**return语句得到的对象**，从而可以进一步运算或者通过赋值语句将其与某个变量绑定

```
def f(x, y):
    z = x + y
    return z
```

```
x = y = 3
y = f(x + 3, y)
```

假设： c为caller名字空间；f1为这次调用名字空间

f1.x = c.x + 3

f1.y = c.y

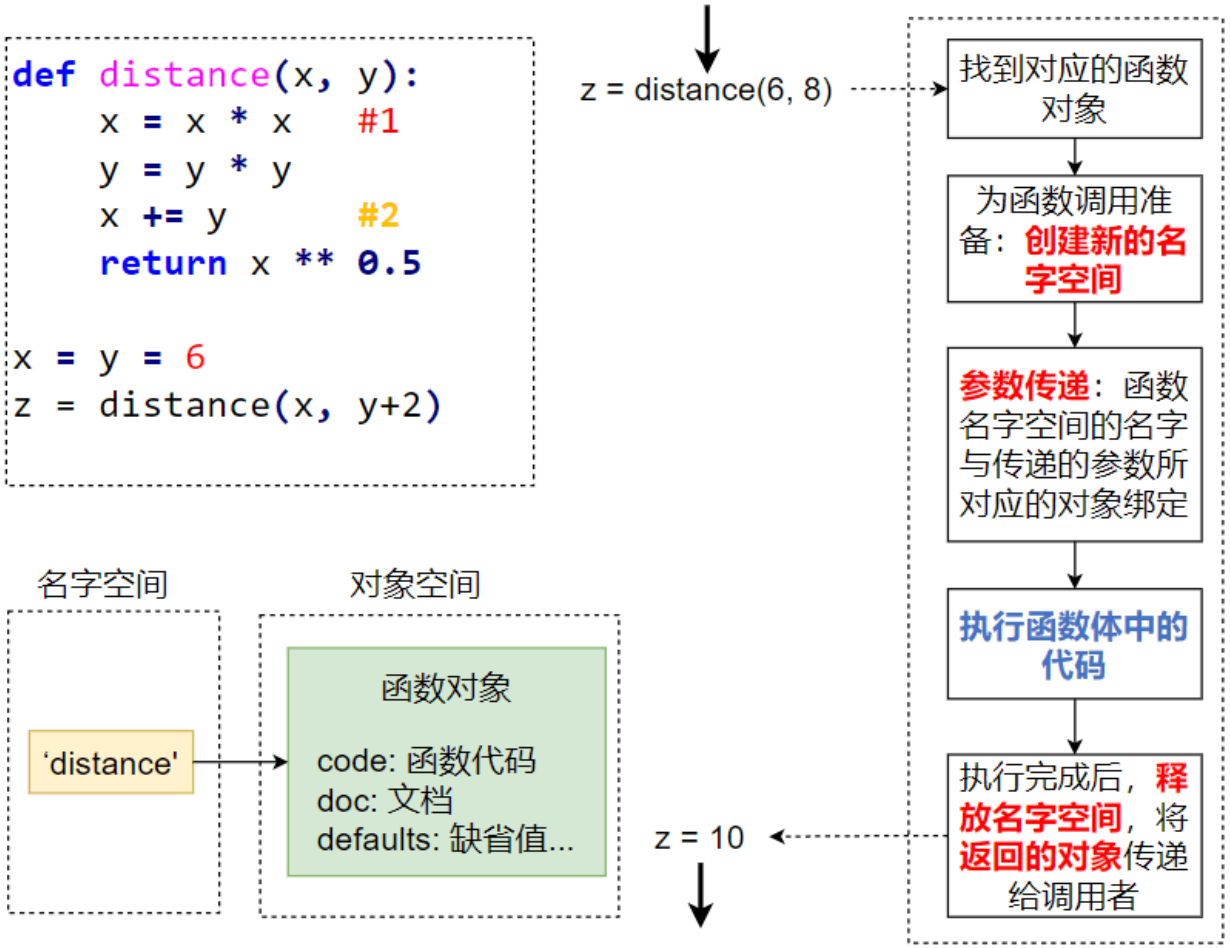
函数调用

- 函数体内可以再调用其他函数，这些函数再调用其他函数
  - 在函数体内部甚至可以调用自己（称为递归函数）
  - 绝大部分情况下，各次函数调用之间是独立的。系统会保存函数调用时的状态（通过堆栈stack frame），在调用完成之后移除该次调用时的状态信息
  - python能够保证函数调用按照预定的顺序来执行，而不会迷失
- 在函数调用时，首先要按照顺序计算各个参数的值后，才进行实际的函数的函数调用

```
def cubic(n):
        return n**3
n = 3
result = cubic(n)
result = cubic(cubic(n))
result = cubic(cubic(cubic(n)))
```

```
def distance(x, y):
    x = x * x    #1
    y = y * y
    x += y       #2
    return x ** 0.5

x = y = 6
z = distance(x, y+2)
```



函数的参数

- 变量不需要声明类型
- 函数定义时，只需要指定参数的名字，而不需要指定参数的类型
  - 形参的类型完全由调用者传递的实参类型决定
  - 函数编写如果有(非语法方面)的问题，只有在调用时才能被发现
  - 传递某些参数时可能执行正确，而传递另一些类型的参数时可能出现错误
- 函数定义时，参数（形参）可以是：
  - 普通(位置)参数
  - 缺省值参数，相应位置没有参数传递时使用缺省值
  - 可变长度位置参数 *args，调用时相应位置可以传递0个或者多个位置参数（实参）
  - 可变长度字典参数 **kwargs，调用时可以传递0个或多个关键字参数（实参）
  - keyword-only参数，如果要传递值时只能通过关键字参数（实参）方式传递

以后介绍

- 函数调用时，参数（实参）可以是：
  - 普通（位置）参数
  - 关键字参数

```
def func(a, b, c, x='x', y='y'):
    print(a, b, c, x, y)

func(1, 2, 3, y=5)
```

函数定义：位置形参和缺省值形参

- 定义函数时，仅包含形参名的参数称为**位置（positional）参数**，根据其出现的位置顺序来逐个匹配相应的实参(argument,函数调用时传递的参数)

```
def func(a, b, c, x='x', y='y'):
    print(a, b, c, x, y)
```

- **缺省值或默认值参数**，格式为 arg=default
  - 给出了形参的名字以及缺省值
  - 可以将缺省值参数看成一个“**特殊的位置参数**”，如果调用时没有传递该参数，则形参指向函数定义时指定的缺省值对象
  - **缺省值保存在函数对象的属性__defaults__**中，该属性为元组类型，保存了所有缺省值参数的当前值
  - arg为函数调用时创建的**新名字空间内部的名字**，表达式default是在**函数定义时所在名字空间**处进行计算的
  - 如果缺省值参数的缺省值为不可变对象，缺省值无法更改，不会有什么问题；如果缺省值参数的缺省值为**可变对象，则需要特别注意（以后会展开）**

```
def func(a, n=n+1):
    print(a, n)
```

- 缺省值参数右边不能再有位置参数
  - 即最前面是位置参数（如果有），然后是缺省值参数(如果有)
  - 采取这种设计的原因是调用时无法区分传递的实参是否为缺省值形参传递的，还是为后面的形参传递的

```
>>> def f(a, x='x', y): print(a, x, y)
SyntaxError: non-default argument follows default argument
```

函数调用：位置实参

函数调用时传递参数(称为实参)有位置参数和关键字参数两种方式：

- **位置参数**：按照参数出现的顺序逐个与函数定义时的参数(形参)对应

args.py

```
def func(a, b, c, x='x', y='y'):
    print(a, b, c, x, y)
```

```
>>> func(1, 2)
Traceback (most recent call last):
  File "<pyshell#87>", line 1, in <module>
    func(1, 2)
TypeError: func() missing 1 required positional argument: 'c'
>>> func(1, 2, 3)
1 2 3 x y
>>> func(1, 2, 3, 4)
1 2 3 4 y
>>> func(1, 2, 3, 4, 5)
1 2 3 4 5
>>> func(1, 2, 3, 4, 5, 6)
Traceback (most recent call last):
  File "<pyshell#91>", line 1, in <module>
    func(1, 2, 3, 4, 5, 6)
TypeError: func() takes from 3 to 5 positional arguments but 6 were given
```

函数调用：关键字实参(keyword argument)

- **调用函数时**的第2种参数传递方式：**关键字参数**
  - 格式name=expr，表示调用时名为name的形参为表达式expr求解后的对象
- 调用时可以使用位置参数和关键字来传递参数，但是与函数定义时类似，**关键字参数右边不能有位置参数**,即调用时参数的顺序为位置参数(如果有)、关键字参数
- 在函数定义中有多个缺省值参数时通过关键字参数可以明确传递哪些参数，哪些采用缺省值
- 通过**关键字参数可以按参数名字**传递值，实参顺序可以和形参顺序不一致，避免了用户需要牢记位置参数顺序的麻烦
- 注意**关键字参数**与函数定义时的**缺省值参数**的区别，虽然两者形式一致，但两者之间**不等同**
  - 函数定义时可不采用缺省值参数，但可通过关键字参数调用
  - 函数定义时可采用缺省值参数，但调用时可选择采用关键字参数或者不采用关键字参数传递

```
def func(a, b, c, x='x', y='y'):
    print(a, b, c, x, y)
func(1, 2, 3)
func(1, 2, 3, y=5)
```

函数调用：实参与形参的匹配

函数调用时，根据传递的实参按照以下规则匹配形参：

- 首先按照**位置**匹配形参中的位置参数和缺省值参数
- 接下来根据**关键字参数的名字**匹配形参中的各个参数
- 函数定义中**尚未匹配的缺省值参数**设置为**缺省值**
- 如果仍然有**尚未匹配**的形参和实参则报错
- 每个参数**只能匹配一次**

```
def func(a, b, c, x='x', y='y'):
    print(a, b, c, x, y)

>>> func(1, 2, 3)
1 2 3 x y
>>> func(1, 2, 3, y=5)
1 2 3 x 5
>>> func(b=1, a=2, c=3, y=5)
2 1 3 x 5
>>> func(1, c=3, b=2, x=5)
1 2 3 5 y
>>> func(1, b=2, x=5)
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    func(1, b=2, x=5)
TypeError: func() missing 1 required positional argument: 'c'
```

主要内容

- 分支结构
  - 比较运算符
  - 单分支/双分支/多分支结构
  - 逻辑运算符
- 函数
  - 函数定义和调用
  - 函数参数
  - **新的格式化方法**
  - 名字空间
- 模块和import语句
- 编程风格
- 内置数学函数和math模块

recap: 内置函数print

print(value1, value2, …, sep=‘ ’, end=‘\n’, file=sys.stdout)

支持可变长的参数，将多个参数转换为字符串并且输出

- 在输出时，这些值之间以sep分隔；sep缺省为空格
- 所有值输出完之后，输出end参数，end缺省为换行
- 缺省输出到标准输出(屏幕)，也可通过file参数指定输出到相应的文件中
- 返回None

```
# 附加Hello, world到文件mytest.txt
fp = open('mytest.txt', 'a+')
print('Hello,world!', file = fp)
fp.close()
```

```
x, y, z='Mike', 4, 3.14
print(x, y, z)                          #输出Mike 4 3.14
print(x, y, z, sep=', ')                #输出Mike, 4, 3.14
print(x, y, z, sep=', ', end='****')    #输出Mike, 4, 3.14*****
print(1, 2, 3, sep='')                  #在上一行最后输出123
print(1, 2, 3, sep='+', end='=')        #输出1+2+3=
print(1 + 2 + 3)                        #在上一行最后输出6
print()                                 #输出空行
```

recap:格式化输出

目前为止已经了解的格式化输出方法

- print(obj1, obj2, ..., sep=' ', end='\n') 通过**sep**和**end**进行格式化
- print(str(obj1) + sep + str(obj2) ...) 利用+ *运算符**组合成一个大的字符串**后输出
- print("Art: %5d, Price: %8.2f" % (453, 59.058)) 利用**%运算符**进行格式化

format_string % value
format_string % (value1, value2, ...)
format_string % map 与字典对象结合，不介绍

第一种格式中如果value包含了其他运算符时，需要注意运算的优先级顺序，如 '%f ‘ % 3.0 * 4.5

- 格式化字符串format_string由固定文本以及格式说明符混合而成。最终的字符串中：
  - 固定文本原封不动
  - 格式说明符部分被替换成运算符%后面的对应位置的值按照指定方式转换之后的字符
- 格式说明符：说明后面的参数中如何转换为字符串
  - 格式： **% [flags] [width] [.precision] type**
- 如果格式化字符串中有多个格式说明符，则%后面的运算数应该是元组tuple对象，格式为(value1, value2, ..., valueN)

格式化输出：%运算符

- 最小宽度和精度如果要**动态构造**，而不是写在格式化字符串中时，可以用*代替，表示具体的取值从后面的元组中获得，即取下一个值

```
>>> '%10.*f' % (2, 3.1415)
'      3.14'
>>> '%*.*f' % (10, 2, 3.1415)
'      3.14'
```

- 格式化的值还可以通过字典对象传递，使用更加方便

format_string % value
format_string % (value1, value2, …)
format_string % map

格式说明符： **% [flags] [width] [.precision] type**

% [(key)] [flags] [width] [.precision] type

(key)用于第三种格式，表示要格式化的值为后面的map对象中key所对应的对象，即map[key]

```
>>> "name:%(name)s  phone:%(phone)s" % {'name': 'stone', 'phone':'55555335'}
'name:stone  phone:55555335
```

%格式化的局限

- 格式化字符串中的格式说明符与后面的值按照位置进行对应
  - 某个值要格式化多次时要重复多次
  - 增加或删除格式说明符时，或者调整先后顺序时，也要相应地调整后面的值的顺序
- 只能支持右对齐和左对齐，且填充字符只能是空格（对于%d %f等还可填充0）
- 除了%s/%d/%f/%e等，无法增加新的类型的格式说明符

字符串对象的format方法

format_string.format(p0,p1,... k0=v0,k1=v1,...)

要格式化的值通过函数调用时的参数传递完成：
- 可通过位置参数传递，还可使用关键字参数传递
- 可以使用函数调用时的序列解包（以后介绍）

格式化字符串从 %[(key)][flags][width][.precision]typecode 变为 {[fieldname][!conversion][:formatspec]}

- { }包含的部分给出了如何格式化某个值，该值格式化后的结果替换{ }部分
- **fieldname**: 确定（自动或人为指定）**当前位置格式化的值**对应哪个位置参数(数字)或关键字参数(名字)
  - 如果省略，表示值是采用自动编号所指出的位置参数。**位置参数编号从0开始**。自动编号时以**左括号为基准自动编号**，每次看到一个左括号开始的要自动编号的值时，编号自动加一。**但注意自动编号和人为编号不能混合使用。**比如"{} {1}".format(1, 2)会报错
  - fieldname后还可以添加.attr或[index]，访问值的属性或者值的下标对应的元素
  - !conversion (不介绍)表示要格式化的值是首先转换后再格式化的，可取sra，分别对应str/repr/ascii函数
- 格式化部分 **:formatspec** 是可选的，格式为 :[[fill]align][sign][#][0][width][,][.precision][typecode]
  - 如果省略，表示采用要格式化的值所支持的**缺省格式化方法**，大部分情况下调用str转换为字符串
- 如果要输出{或者}，使用两个对应的大括号，即**{{或者}}**

```
>>> "First %s, Second %d, Third %5.2f" % ('idle', 90, 93.4)
'First idle, Second 90, Third 93.40'
>>> "First {0:s}, Second {1:d}, Third {2:5.2f}".format('idle', 90, 93.4) #位置参数编号从0开始
'First idle, Second 90, Third 93.40'
>>> "First {}, Second {}, Third {:5.2f}".format('idle', 90, 93.4)    #位置参数也可自动编号
'First idle, Second 90, Third 93.40'
```

字符串对象的format方法：示例

```
>>> "%6.2f or %6.3f" % (3.1415, 3.1415)
'  3.14 or  3.142'
>>> "{0:6.2f} or {0:6.3f}".format(3.1415)   # 不需要重复传递多个值
'  3.14 or  3.142'
>>> "Art: {a:5d},  Price: {p:8.2f}".format(a=453, p=59.058)  # 关键字参数
'Art:   453,  Price:    59.06'
>>> print("Second {1}, First {0}, Third {2}\nArt: {a:5d},  Price: {p:8.2f}".format('idle', 90, 93.4,
a=453, p=59.058 ))  # 位置参数和关键字参数结合使用
Second 90, First idle, Third 93.4
Art:   453,  Price:    59.06
>>> '{{{} > {}}} {}'.format(54, 42, 54 > 42)   # {{替代以 {
'{54 > 42} True'
```

fieldname后还可以添加.attr或[index]，首先得到相应值的属性或者其下标对应的元素，然后格式化

```
>>> c = 3-5j
>>> '%s, 实部 %s 虚部 %s.' % (c, c.real, c.imag)
'(3-5j), 实部 3.0 虚部 -5.0.'
>>> '{0}, 实部 {0.real} 虚部 {0.imag}.'.format(c)
'(3-5j), 实部 3.0 虚部 -5.0.'
>>> coord = (3, 5)
>>> 'X: %s;  Y: %s' % (coord[0], coord[1])
'X: 3; Y: 5'
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3; Y: 5'
```

字符串对象的format方法：格式化说明符

```
>>> help('FORMATTING')
```

%运算符	字符串的format方法
[-] [0] [+] [#][width] [.precision] typecode	[[fill]align][sign][#][0][width][,或_][.precision][typecode]

%格式中

- **缺省为右对齐，可以通过-指定左对齐**
- **缺省填充空格。对于%d %f等可填充0**

align: 对齐方式，指定了最小宽度才有意义

- < 左对齐 大部分对象缺省为左对齐
- > 右对齐 数字类型缺省为右对齐
- ^ 中间对齐
- = 用于数字类型，表示右对齐，且填充字符在符号(+-，如果有)之后，如- 45

fill: 对齐时使用的填充字符，缺省为空格

- sign:符号，可取+-和空格。+表示总是包含符号；-表示仅负数有符号(缺省)；空格表示负数有符号，正数时为空格
- # 用于数字类型。在格式化整数(o/x/b等)时前面添加前缀(0o/0x/0b)
- 0表示在数字前填充足够多的0
- **,或_**用于数字类型，表示每千位加逗号或下划线
- typecode，与%类似，只是：
  - **省略时**选择要格式化的值所支持的格式化方法格式化
  - **s仅用于字符串类型**
  - 新加入 b将整数格式化成二进制字符串
  - **% 百分比**，比如'{:.2%}'.format(0.456)的结果为'45.60%'
  - 可以是要格式化的对象所**自定义的typecode**
    - 相应的对象类型中实现__format__(self, format)方法

字符串对象的format方法：格式化说明符动态构造

- %运算符中最小宽度和精度如果要**动态构造**，而不是写在格式化字符串中时，可以用*代替，表示具体的取值从后面的value元组中获得，取下一个参数

```
>>> '%*.*f' % (10, 2, 3.1415)
'    3.14'
```

- 字符串对象的format方法通过嵌套支持格式化说明符的动态构造，但仅允许嵌套一层

```
>>> '{0:{1}}'.format(12345, 10)    # '{0:10}'.format(12345)
'     12345'
>>> '{:{:s}{align}{width}}'.format('hello', '*', align='^', width=20)
'*******hello********'
# 通过自动编号和关键字参数传递参数
# 嵌套的部分解析完成后等价于  {:*^20}.format('hello')
```

字符串对象的format方法

- %为格式化运算符
- 字符串的format方法为实例方法
  - 可看成(特殊的)函数对象，隐藏了的参数为字符串对象
  - 可以暂时保存字符串的format方法到某个变量，即**该变量指向该函数对象**

```
>>> "Art: {a:5d},  Price: {p:8.2f}".format(a=453, p=59.058)
'Art:   453,  Price:    59.06'
>>> format_func = "Art: {a:5d},  Price: {p:8.2f}".format
>>> format_func
<built-in method format of str object at 0x05861090>
>>> format_func(a=453, p=59.058)
'Art:   453,  Price:    59.06'
```

```
weather = [("Monday","rain"),("Tuesday","sunny"),
 ("Wednesday",  "sunny"),("Thursday", "rain"),("Friday","Cloudy")]

formatter = "Weather of '{0[0]}' is '{0[1]}'".format
for item in weather:
    print(formatter(item))
```

```
Weather of 'Monday' is 'rain'
Weather of 'Tuesday' is 'sunny'
Weather of 'Wednesday' is 'sunny'
Weather of 'Thursday' is 'rain'
Weather of 'Friday' is 'Cloudy'
```

```
“Weather of ‘{0[0]}’ is ‘{0[1]}’”.format(("Monday","rain"))
```

内置函数format

format(value,format_spec='')　　将value按照format_spec进行格式化

- 等价于　'{:format_spec}'.format(value)
- format_spec与前面字符串的format方法中的格式一致（注意不包括字段部分及冒号）

  [[fill]align][sign][#][0][width][,][.precision][typecode]

```
>>> '{:6.2f}'.format(3.1415)
'  3.14'
>>> format(3.1415, '6.2f')
'  3.14'
>>> '%6.2f' % 3.1415
'  3.14'
```

字符串的方法：填充和对齐

- center/ljust/rjust/zfill：返回新字符串，原字符串居中或左对齐或右对齐，并使用指定字符（默认空格）填充，保证长度至少为width
- expandtabs([tabsize]): 将\t转换为多个(tabsize)空格，缺省为8个

方法	描述	等价于
s.center(width[,fillchar])	居中对齐	'{:{}^{}}'.format(s, fillchar, width)
s.ljust(width[,fillchar])	左对齐	'{:{}<{}}'.format(s, fillchar, width)
s.rjust(width[,fillchar])	右对齐	'{:{}>{}}'.format(s, fillchar, width)
s.zfill(width)	右对齐，填充0	rjust(width, '0') 或 '{:0>{}}'.format(s, width)

```
>>> '1\t2\t3'.expandtabs()
'1       2       3'
>>> '1\t2\t3'.expandtabs(2)
'1 2 3'
```

```
>>> s = 'Hello world!'
>>> s.center(20)
'    Hello world!    '
>>> s.center(20, '=')
'====Hello world!===='
>>> '{:=^20s}'.format(s)
'====Hello world!===='
>>> format(s, '=^20s')
'====Hello world!===='
>>> s.ljust(20, '=')
'Hello world!========'
>>> '{:=<20s}'.format(s)
'Hello world!========'
>>> s.rjust(20, '=')
'========Hello world!'
>>> '{:=>20s}'.format(s)
'========Hello world!'
```

f-string(formatted string literal)

python3.6引入，语法： f'……{expression:format_spec}……'

- 与传统字符串定义类似，可以为单双引号以及三引号，只是前面为f或者F
- 大括号包括的部分被替代：**表达式expression**运算后的结果按照后面的 **:format_spec**进行格式化，:format_spec可省略，表示采用缺省方式格式化
- 大括号内不允许使用转义字符\
- {{表示{，}}表示}

fstring.py

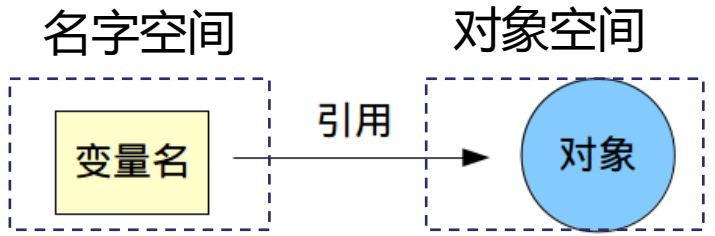
```
name = 'Fred'
age = 50
print(f'My name is {name}, my age next year is {age+1}.')

print(f'{"Name":<15}{"Score":<15}')

import math
print(f'pi={math.pi:.5f}, log(2)={math.log(2,10):.5f}'
```

```
My name is Fred, my age next year is 51.
Name           Score
pi=3.14159, log(2)=0.30103
```

名字空间(namespace)



- **名字空间（namespace）**是名字（变量）和对象的映射
- 允许有多个名字空间，一个名字可以在多个名字空间里面出现，代表不同名字空间的名字
- 模块的**全局名字空间**：globals()可以查看当前的全局名字空间的名字
  - 模块被加载(import)或者以脚本方式运行时创建
  - **赋值类语句不在任意函数的函数体内出现时，其所引入(定义或赋值)的变量**，称为全局变量
  - 全局变量的有效范围为定义它的模块
- 函数的**本地名字空间**：locals()可以查看当前本地名字空间的名字
  - 在函数调用时创建，在调用结束时销毁
  - 一般而言，**赋值类语句出现在函数内部，且在函数内部没有使用global或nonlocal语句声明的变量**(包括参数)属于该名字空间，称为局部变量
  - 局部变量的有效范围为函数体
- **内置(built-in)名字空间**：
  - 在进入解释器时创建，退出解释器时结束
  - 包含了内置函数以及异常类型，**dir(__builtins__)可以查看该名字空间的名字**

locals()	当前名字空间的名字与对象的映射
globals()	全局名字空间的名字与对象的映射

名字空间：赋值类语句

- 3个问题：
  - 有哪些类型的名字空间？全局、本地和内置名字空间
  - **引入的名字属于哪个名字空间？**
  - 表达式中出现未限定名时，该名字是哪个名字空间的名字？
- 赋值类语句（赋值语句、import语句、函数定义等引入名字的语句）所处的位置决定了往哪个名字空间中引入名字
  - 赋值类语句没有出现在函数体内，其引入的名字就属于全局名字空间
  - 如果赋值类语句出现在函数体内，
    - 引入的名字(假设为var)属于本地名字空间，**不管其出现的位置**
    - global var，表示var应该在全局名字空间。在函数体内的赋值类语句给var赋值时，改变的是全局名字中的var的映射
    - 拓展：nonlocal var，表示var在外层（以及外层的外层等）函数的名字空间。函数嵌套时内层函数要改变或访问外层函数的变量时使用

```
g = 1

def f(x):
    t = 4
```

```
def f(x):
    global g
    g = 4
```

```
def f(x):
    def p(x):
        nonlocal t
        t += 1
        print(t)
    t = 4
    p(1)
```

非限定名的绑定

由于名字空间中的名字的有效范围会有重叠，在**访问变量**(注意不是**赋值**)时，怎么知道某处代码中的名字到底是属于哪个名字空间呢?

- **LEGB规则**：根据其所出现的位置，遵循 Local →Enclosing→Global→Builtin 原则
- 按照顺序搜索相应的名字空间，直到到达限定的名字空间范围或找到匹配为止

- 表达式出现在函数体外，全局/内置名字空间查找
- 表达式出现在函数体内，则：
  - 函数体中有global语句，全局/内置名字空间
  - 函数体中有nonlocal语句，外层函数名字空间
  - 函数体中有赋值类语句(不管出现的先后顺序），本地名字空间
  - 否则从内往外，外层/全局/内置名字空间
- 在定义函数的时候分析赋值类/global/nonlocal语句，函数对象中记录"函数体中的名字到底应该属于哪个名字空间"
- **迟来绑定(late binding)**：只有在对表达式进行求解才会绑定。如果为自由变量，表达式求解或者其所在名字空间销毁的时候绑定

名字空间

Builtin：在python环境的内置模块builtins中定义的变量（内置函数等）

Global(模块)：在函数外部定义的变量或者在函数体内通过**global**声明过

Enclosing: 外层函数内部赋值的变量或通过**nonlocal**声明过，往外层搜索

Local: 函数内部赋值的变量，且没有通过global/nonlocal声明过

非限定名的绑定

- 调用f1(2,2) 时：
  - 全局名字空间： sys, math, x, y, z, f1, times
  - 本地名字空间中的名字有x, y, x1, z
  - 访问的名字x, x1, y, z都在本地名字空间中查找
  - print和times会在全局或内置名字空间中查找

- 调用f2(2, 2)时：
  - 全局名字空间： sys, math, x, y, z, f1, f2, times
  - 本地名字空间： x, y, x1, z
  - f2的函数体无法直接访问全局名字空间的x/y/z

    拓展 sys.modules[__name__].z: 根据模块名访问模块对象，然后通过属性的方式访问

    globals()['z']:通过globals函数得到维护全局名字空间的字典，然后名字作为下标访问
  - f2的函数体可以访问全局变量times/sys/math，内置函数print/globals

scope.py

```
import sys
import math
x = y = z = 0
def f1(x, y):
    x1 = x * x
    z = x1 + y*y
    print('z =', z)
    print('times =', times) #全局变量

times = 1
f1(2,2)
```

```
def f2(x, y):
    x1 = x * x
    z = x1 + y*y
    print('z =', z)

    # 访问同名的位于全局名字空间的变量
    print(sys.modules[__name__].z)
    print(globals()['z’])

    print('times =', times)  # 全局变量

times = 2
f2(2,2)
```

非限定名的绑定

- 赋值类语句出现在函数体，如果没有global或nonlocal语句，**不管其出现的位置**，记录其属于本地变量的信息于函数对象
- 函数体中通过赋值类语句修改全局变量，需要在使用前使用**global语句**显式声明
- 拓展：函数体中通过赋值类语句修改外层函数中的变量，需要在使用前使用nonlocal语句显式声明
- 如果函数体内没有赋值类语句，则基于LEGB原则搜索(实际为EGB），可以访问外层函数的变量或全局变量，可不需要global语句

```
def f3(x, y):
    x1 = x * x
    z = x1 + y*y
    print('z =', z)

    print('times =', times)
    times = times + 1
    # UnboundLocalError: local variable
'times' referenced before assignment
```

```
def f4(x, y):
    x1 = x * x
    z = x1 + y*y
    print('z =', z)

    global times
    print('times =', times)
    times = times + 1
```

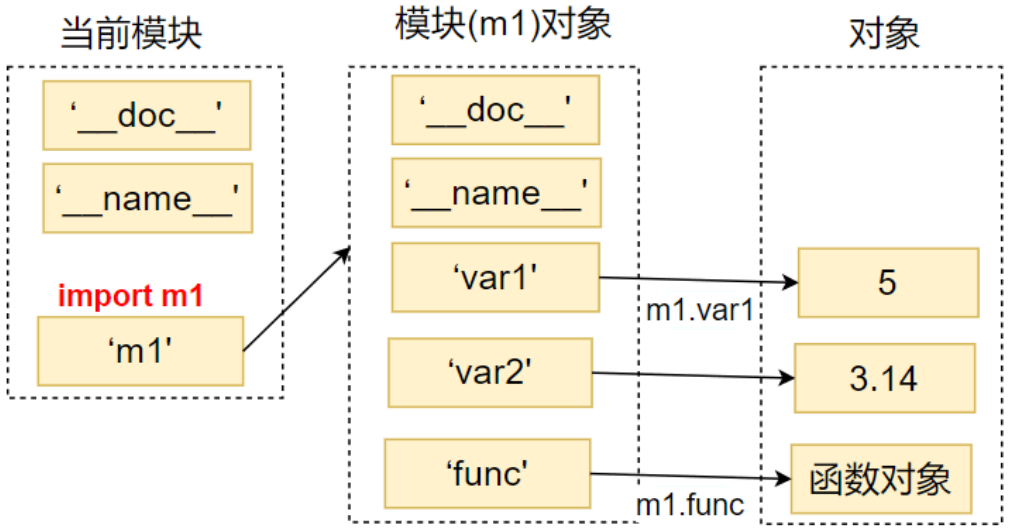
- 尽量**减少全局变量的使用**，通过**参数传递体现调用者和函数实现之间的关系**，减少依赖和副作用

主要内容

- 分支结构
  - 比较运算符
  - 单分支/双分支/多分支结构
  - 逻辑运算符
- 函数
  - 函数定义和调用
  - 函数参数
  - 新的格式化方法
  - 名字空间
- **模块和import语句**
- 编程风格
- 内置数学函数和math模块

模块

- 模块(module)：相互之间有联系的函数以及变量组织在一起，放到同一个Python源文件（.py)
  - 不同模块属于不同的**名字空间**，可以避免函数名和变量名冲突
  - 提高了代码的可维护性和可重用性
- 如何使用模块？ import 模块名 [as 别名]　　# import math
  - 系统会寻找模块源文件，**加载并执行**模块中的代码，构造模块对象
  - 在**当前名字空间引入名字**指向模块对象 。 # 变量math指向加载的模块对象
- 如何访问模块(名字空间)中的对象(变量和函数)?
  - 在模块中定义的变量和函数保存在模块对象中
  - 模块中的变量可通过模块对象的属性(attribute)访问
  - 模块中的函数可通过模块对象的方法(method)访问
- 模块和模块之间的联系是通过import语句来建立的
- 未限定的名字遵循LEGB规则确定其属于**当前模块的**



哪个名字空间，其他模块中的名字通过import语句得到模块对象来限定访问

导入模块

- 如果**第一次**导入，则寻找模块源程序，**加载(也就是执行)**模块，保存**模块对象**
  - 从**sys.path**给出的目录列表中查找，找到第一个名字为模块名的源程序，**执行该源程序**
  - 已经导入的模块保存在字典对象**sys.modules**, key为模块名，而value为模块对象
- 可以多次调用import，但是**只加载执行一次**
  - 如果该模块名已经出现在sys.modules或者为内置模块(**sys.builtin_module_names**)，则找到对应的模块对象，添加相应的对象引用
  - 可通过importlib.reload(obj)来重新加载已经加载过的模块，较少使用

```
>>> import sys
>>> sys.path
['', 'D:\\Software\\Python39\\python39.zip', 'D:\\Software\\Python39\\DLLs',
'D:\\Software\\Python39\\lib', 'D:\\Software\\Python39', 'D:\\Software\\Python39\\lib\\site-packages']
```

可以看到首先从当前目录中查找，最后是安装目录下的site-packages

hello.py

```
pi = 3.14
print('loading...', __file__)
if __name__ == '__main__':
    print("Hello World")
```

IDLE中首先Run，将切换到hello.py所在目录

```
>>> import hello
loading...  ...hello.py
Hello World
>>> hello.pi
3.14
>>> sys.modules['hello']
<module 'hello' from ...\PythonClass\\chap1\\hello.py'>
>>> import hello
```

导入模块

- 第一次导入时，在sys.path给出的目录中搜索，找到第一个名字为模块名的源程序，**执行该源程序**
- 注意搜索顺序可能导致没有导入正确的模块
  - 比如base64模块是python平台提供的一个模块，但是如果在sys.path的目录中搜索，经常是首先从当前工作目录开始搜索，如果也有一个base64.py，则导入的是用户提供的base64模块，而不是Python平台提供的模块

hello.py

```
pi = 3.14
print('loading...', __file__)
if __name__ == '__main__':
    print("Hello World")
```

base64.py

```
print('loading my base64.py')
```

通过run module方式运行hello.py，以将工作目录切换到其所在的目录

```
>>>
 RESTART: ... chap1\hello.py
loading... ...chap1\hello.py
Hello World
>>> import base64
loading my base64.py
```

将base64.py改名，比如base64_.py，避免无法使用平台提供的base64模块

系统内置模块

- Python默认安装仅包含部分基本或核心模块，用户可安装大量的扩展模块(以package形式组织)
  - 访问https://pypi.python.org/pypi来查找相应的package
  - 在操作系统的命令行应用程序中执行 **pip install package_name**安装相应的扩展包
- Python启动时，仅加载了很少的一部分模块（包括一些内置的模块比如builtins、sys等），在需要时由程序员显式地加载（可能需要先安装）其他模块
- 这种设计的目的是减小运行的压力，仅加载真正需要的模块和功能，且具有很强的可扩展性
- **sys.builtin_module_names**给出了Python解释器内置的模块
- 尽管内置模块已经加载，但是要使用内置模块的函数仍然要执行 import

```
>>> import sys
>>> sys.builtin_module_names
('_ast', '_bisect', '_codecs', '_codecs_cn', '_codecs_hk', '_codecs_iso2022', '_codecs_jp',
'_codecs_kr', '_codecs_tw', '_collections', '_csv', '_datetime', '_functools', '_heapq', '_imp',
'_io', '_json', '_locale', '_lsprof', '_md5', '_multibytecodec', '_opcode', '_operator', '_pickle',
'_random', '_sha1', '_sha256', '_sha512', '_signal', '_sre', '_stat', '_string', '_struct',
'_symtable', '_thread', '_tracemalloc', '_warnings', '_weakref', '_winapi', 'array', 'atexit',
'audioop', 'binascii', 'builtins', 'cmath', 'errno', 'faulthandler', 'gc', 'itertools', 'marshal',
'math', 'mmap', 'msvcrt', 'nt', 'parser', 'sys', 'time', 'winreg', 'xxsubtype', 'zipimport', 'zlib')
```

Python之禅（The Zen of Python）

```
>>> import this
```

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!

—— Tim Peters

优美胜于丑陋。
显式胜于隐式。
简单胜于复杂。
复杂胜于难懂。
扁平胜于嵌套。
分散胜于密集。
可读性应当被重视。
尽管实用性会打败纯粹性，特例也不能凌驾于规则之上。
除非明确地使其沉默，错误永远不应该默默地溜走。
面对不明确的定义，拒绝猜测的诱惑。
用一种方法，最好只有一种方法来做一件事。
虽然一开始这种方法并不是显而易见的，但谁叫你不是Python之父呢。
做比不做好，但立马去做有时还不如不做。
如果实现很难说明，那它是个坏想法。
如果实现容易解释，那它有可能是个好想法。
命名空间是个绝妙的想法，让我们多多地使用它们吧！

优雅、明确、简单

import语句

import module [as alias]　加载模块，在**当前名字空间引入某些名字**

- import module: 加载模块module，当前名字空间中的变量module指向加载的模块对象
- import module as alias: 加载模块module，变量alias指向加载的模块对象
- 多个import语句可合并在一起，表示加载多个模块，比如import sys, math as math2。但不建议这样使用
- dir(module)可以查看模块对象module中可以通过返回的属性列表。如果调用不带参数的dir()函数，则返回当前名字空间内的所有名字（前面定义的变量）
- help(obj)函数查看任意模块对象（如果传递参数为模块对象）或函数（如果传递参数为函数）的使用帮助

```
>>> import math
>>> math.sin(3)
0.141120080598672
>>> import math as math2
>>> math2.sin(3)
0.141120080598672
```

```
>>> dir(math)
...
>>> help(math)
...
>>> help(math.sin)
...
>>> dir(__builtins__)
```

- 内置模块**builtins**定义了许多内置的类型(int/float等)和内置函数(print/input)等
- 解释器执行了下述语句：

```
import builtins as __builtins__
```

- 作用域搜索的LEGB规则中，**内置名字空间**指的是__builtins__所指向的(模块)对象的名字空间

import语句

from module import obj [as alias]

- 仅从模块module中**导入名字为obj的对象**，当前名字空间引入变量obj或alias，指向该导入的对象
- 访问该对象时不再需要包括模块名，可以减少查询次数，提高执行速度

from math import *

- 从模块中导入所有的对象到当前名字空间
- 多个模块中有同样的对象名时造成混乱，因此谨慎使用
- _xxx **单下划线开始的变量**为保护变量，表示from module import *导入时以单下划线开始的变量排除在外
- **补充**：如果待导入的模块中定义了__all__变量，则在使用from module import *时，只有该变量中列出的名字才可以被其他模块导入

```
__all__ = [
    'abc', 'def'
]
```

```
>>> from math import sin
>>> sin(3)
0.1411200080598672
>>> from math import sin as f, cos
>>> f(3)
0.1411200080598672
>>> cos(0)
1.0
>>> from math import *
```

__name__

拓展：python -m module 以脚本方式执行模块module

除了在源程序中定义的名字外，python解释器还会（在模块真正执行之前）定义一些内部的名字，包括__name__，__doc__，__file__等

模块一般有两种方式执行

- 在IDLE环境中以Run Module方式运行，即该模块为主入口程序，也称为脚本(script)方式运行
  - 主入口(脚本)方式运行时，__name__被设置为“__main__”
  - 每个模块经常会包含测试代码以测试其所定义的函数，这些测试代码在以脚本方式时**应该执行**
- 作为模块被导入(即**被其他模块import**)，如果为第一次导入，则执行该模块源程序中的代码
  - 以import方式执行的模块中自动增加的名字中，__name__被设置为该模块的模块名
  - 模块中经常包含的测试代码在import时**不应该执行**

在模块最后，一般包含：

```
if __name__ == '__main__':
   #以脚本方式运行时才会执行的代码
```

hello.py

```
pi = 3.14
print('loading...', __file__)
if __name__ == '__main__':
    print("Hello World")
```

__name__	模块的名字，以脚本方式运行时设置为"__main__"，import方式运行时设置为**模块的文件名**(不包括.py)
__file__	模块源程序的完整路径名
__doc__	文档字符串，如果模块的第一个语句为字符串字面量，则该字符串作为文档字符串保存在__doc__中

python代码编写规范

IDLE可通过下面方法进行代码块的缩进和反缩进：
- 菜单项：Format➔Indent Region/Dedent Region
- 快捷键： Ctrl + [和 Ctrl +]

- 统一的编写规范：
  - PEP 8 -- Style Guide for Python Code https://www.python.org/dev/peps/pep-0008/
  - Google Python Style Guide http://google.github.io/styleguide/pyguide.html
- 代码布局(code layout)
  - Python可以在同一行中放置多条语句，语句之间使用分号“;”分割，但为可读起见，不建议在同一行中放置多条语句
  - 每行不超过79个字符，如果一行语句太长，可以在行尾加上\紧跟着换行，这样可以分成多行，但是更建议使用括号来包含多行内容
  - python程序依靠代码块的缩进体现代码间的逻辑关系
    - def/if/while/for等复合语句中,行尾的冒号（后面可以包括空格等）表示下面应该是缩进的代码块
    - 缩进结束表示代码块的结束
    - 同一个级别的代码块的缩进量必须相同
  - 建议以4个空格为基本缩进单位，不建议采用制表符缩进，更不能混合使用制表和空格

```
if a > b and a > c and a > 5  and b > 5 and c > 5  \
   and b > c:
    print('blah')

if (a > b and a > c and a > 5  and b > 5 and c > 5
   and b > c) :
    print('blah')
```

python代码编写规范

空白符：空格、制表(Tab)和换行符
- 换行符：分割语句
- 缩进代码：具有特殊的含义
- 程序中可输入任意数量的空白符，建议遵循一个好的编程风格

- 代码布局(code layout)
  - 必要的空行
    - **不同功能**的代码块之间建议增加**一个空行**以提高可读性
    - 不同的函数定义之间可以使用**两个空行**隔开
  - 必要的空格
    - 建议在**逗号后面**添加一个空格，但是逗号前面没有空格
    - 函数定义时的缺省值参数以及调用时的关键字参数中**=前后**没有空格
    - **赋值语句的=、比较运算符以及逻辑运算符**(and or not) 的前后各一个空格
    - **算术运算符**
      - 如果只有一个算术运算符，运算符两侧前后各一个空格
      - 如果有多个优先级的运算符出现时，**更低优先级的运算符**前后添加一个空格

```
def func(a, b, c, x='x', y='y'):
    print(a, b, c, x, y)

func(1, 2, 3, y=5)
```

```
a = 4 + 5
x = x*2 - 1
hypot2 = x*x + y*y   # +优先级更低
c = (a+b) * (a-b)    # *相比括号优先级更低
```

python代码编写规范

IDLE开发环境中，可快速注释/解除注释大段内容：
- Format➔Comment Out Region/Uncomment Region
- 快捷键： ALT+3 ALT+4

- 一个好的、可读性强的程序应该有注释
  - 以#开始，表示本行#之后的内容为注释
  - 多行注释可以采用长字符串字面量定义。经常在模块开始和函数体开始处添加长注释（三引号），作为文档字符串，文档字符串同样也与代码块有相同的缩进
- 每个import只导入一个模块
  - 首先导入Python标准库模块，如os、 sys、 re
  - 导入第三方扩展库，如numpy、 scipy
  - 导入自己实现的本地模块
- 命名规则：参见前面介绍赋值语句时建议变量名的命名规范
  - 建议采用ASCII字符集中的字符（字母数字和下划线），采用有意义的名字
  - 模块、函数、变量等采用小写字母，如果为多个单词时，单词间以下划线隔开
  - 类型名建议采用CamelCase(驼峰）名，即每个单词首字母大写，比如CapWords
  - 常量一般在模块级定义，建议采用大写形式，比如MAX_OVERFLOW、 TOTAL

主要内容

- 分支结构
  - 比较运算符
  - 单分支/双分支/多分支结构
  - 逻辑运算符
- 函数
  - 函数定义和调用
  - 函数参数
  - 新的格式化方法
  - 名字空间
- 模块和import语句
- 编程风格
- **内置数学函数和math模块**

内置数学函数

	0	0.5	1	1.5	2

- 内置函数来自于内置的模块builtins，可直接通过函数名访问

0.5 + 1.5 = 2
round(0.5) + round(1.5) = 0 + 2 = 2

函数	含义	实例	结果
abs(x)	x的绝对值。如果x为复数，则返回复数的模	abs(-1.2) abs(1-2j)	1.2 2.2360679749979
divmod(a,b)	返回a除以b的商和余数	divmod(5, 3)	(1,2)
pow(x,y[,z])	返回x**y。如果z有，则为pow(x, y) % z	pow(2, 10) pow(2, 10, 10)	1024 4
max(arg1,arg2,*args)	取最大值	max(1, 7, 3, 15, 14)	15
min(arg1,arg2,*args)	取最小值	min(1, 7, 3, 15, 14)	1
sum(iterable[,start]) 暂时不介绍	求和，start如果有，表示加上start	sum((1, 2, 3)) sum((1, 2, 3), 44)	6 50
round(number[,ndigits])	四舍六入五成双取整，如果ndigits则保留ndigits小数。可以为负数，表示保留10的多少次方	round(3.14159) round(3.14159, 4) round(314159, -3)	3 3.1416 314000

```
>>> x, y = divmod(5, 3)
```

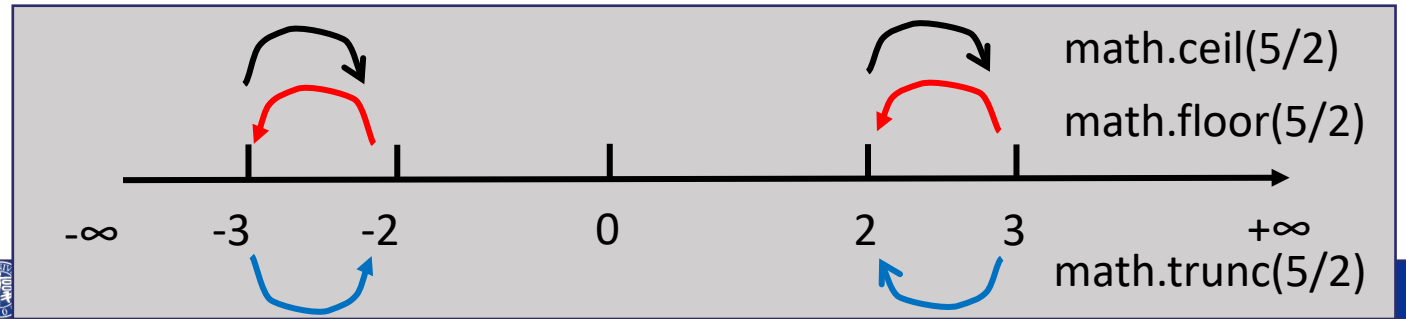
math模块

- math.pi: 数学常量 π
- math.e: 数学常量e

```
>>> import math
>>> math
<module 'math' (built-in)>
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan',
'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
>>> help(math)   # 查看math模块帮助
>>> help(math.sin)  #查看math模块中的sin函数帮助
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

math模块

函数	含义	实例	结果
fabs(x)	绝对值，返回float	fabs(-3)	3.0
ceil(x)	大于等于x的最小的整数(正无穷方向取整)	ceil(1.2), ceil(-1.6)	(2, -1)
floor(x)	小于等于x的最大的整数(负无穷方向取整)	floor(1.8), floor(-2.1)	(1, -3)
trunc(x)	截取为最接近0的整数（原点方向取整）	trunc(1.2), trunc(-2.8)	(1, -2)
sqrt(x)	x(>=0)的平方根	sqrt(2)	1.4142135623730951
exp(x)	e ** x	exp(5)	148.4131591025766
log(x[,base])	以base为底x的对数，base没有则为以e为底的自然对数，即缺省为ln(x)	log(math.e ** 2) log(4, 2)	2.0 2.0
log10/log2(x)	以10或2为底的对数	log10(100), log2(16)	2.0, 4.0
factorial(x)	整数x(>=0)的阶乘, x!，0!为1	factorial(5)	120
gcd(x, y)	整数x和y的最大公约数	gcd(72, 40)	8



- int(x): x为整数或浮点数时，等价于math.trunc(x)
- round(x, ndigits): 四舍六入五成双
- '%.4f' % x 相当于 round(x, 4)

math模块：三角函数、角度转换

函数	含义	实例	结果
sin(x)	x(**弧度**)的正弦	sin(pi/2)	1.0
cos(x)	x的余弦	cos(2*pi)	1.0
tan(x)	x的正切	tan(pi/4)	0.9999999999999999
asin(x)	x的反正弦	asin(1)	1.570796326794896 6
acos(x)	x的反余弦	acos(1)	0.0
atan(x)	x的反正切	atan(1)	0.7853981633974483
degrees(x)	x从弧度转换为角度	degrees(pi)	180.0
radians(x)	x从角度转换为弧度	radians(90)	1.570796326794896 6

浮点数的相等判断

- 十进制的浮点数在计算机内部采用二进制进行运算时会有精度误差

  0.1 + 0.1 + 0.1 == 0.3　　# 比较的结果为False

  - 如何判断浮点数相等呢?
    - 如果差的绝对值非常小，(近似)相等： abs(a-0.3) < 1e-10
    - 采用math模块的isclose函数

    isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)

拓展:

- float类型还提供了多个内置函数，可通过dir(float)查看，其中is_integer()用于判断是否为整数
- fractions模块提供简分数Fraction(分子分母都是整数)
- decimal模块提供高精度浮点数Decimal，用于**十进制数学计算**，保证用户**指定的精度（缺省为28位）**

```
>>> a = 0.1+0.1+0.1
>>> a
0.30000000000000004
>>> a == 0.3
False
>>> abs(a - 0.3)
5.55111512312578e-17
>>> abs(a - 0.3) < 1e-10
True
>>> import math
>>> math.isclose(a, 0.3)
True
>>> a.is_integer()
False
>>> t = 4.0
>>> t.is_integer()
True
```

复数的数学运算

- 复数与int/float一样支持常用的数学运算

```
>>> a = 3 + 4j
>>> b = 5 + 6j
>>> c = a + b
>>> c
(8+10j)
>>> a * b #复数乘法
(-9+38j)
>>> a / b #复数除法
(0.6393442622950819+0.03278688524590165j)
```

- 模块cmath还提供了其他复数的数学运算函数

```
>>> import cmath
>>> dir(cmath)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atanh', 'cos',
'cosh', 'e', 'exp', 'isclose', 'isfinite', 'isinf', 'isnan', 'log', 'log10', 'phase', 'pi', 'polar', 'rect', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```