```
# 用井字符开头的是单行注释

""" 多行字符串用三个引号
    包裹，也常被用来做多
    行注释
"""

####################################################
## 1. 原始数据类型和运算符
####################################################

# 整数
3  # => 3

# 算术没有什么出乎意料的
1 + 1  # => 2
8 - 1  # => 7
10 * 2  # => 20

# 但是除法例外，会自动转换成浮点数
35 / 5  # => 7.0
5 / 3  # => 1.6666666666666667

# 整数除法的结果都是向下取整
5 // 3     # => 1
5.0 // 3.0 # => 1.0 # 浮点数也可以
-5 // 3  # => -2
-5.0 // 3.0 # => -2.0

# 浮点数的运算结果也是浮点数
3 * 2.0 # => 6.0

# 模除
7 % 3 # => 1

# x 的 y 次方
2**4 # => 16

# 用括号决定优先级
(1 + 3) * 2  # => 8

# 布尔值
True
False
```

```
# 用 not 取非
not True  # => False
not False  # => True

# 逻辑运算符，注意 and 和 or 都是小写
True and False #=> False
False or True #=> True

# 整数也可以当作布尔值
0 and 2 #=> 0
-5 or 0 #=> -5
0 == False #=> True
2 == True #=> False
1 == True #=> True

# 用==判断相等
1 == 1  # => True
2 == 1  # => False

# 用!=判断不等
1 != 1  # => False
2 != 1  # => True

# 比较大小
1 < 10  # => True
1 > 10  # => False
2 <= 2  # => True
2 >= 2  # => True

# 大小比较可以连起来!
1 < 2 < 3  # => True
2 < 3 < 2  # => False

# 字符串用单引双引都可以
"这是个字符串"
'这也是个字符串'

# 用加号连接字符串
"Hello " + "world!"  # => "Hello world!"

# 字符串可以被当作字符列表
"This is a string"[0]  # => 'T'
```

```
# 用.format 来格式化字符串
"{} can be {}".format("strings", "interpolated")

# 可以重复参数以节省时间
"{0} be nimble, {0} be quick, {0} jump over the {1}".format("Jack", "candle
stick")
#=> "Jack be nimble, Jack be quick, Jack jump over the candle stick"

# 如果不想数参数，可以用关键字
"{name} wants to eat {food}".format(name="Bob", food="lasagna") #=> "Bob
wants to eat lasagna"

# 如果你的 Python3 程序也要在 Python2.5 以下环境运行，也可以用老式的格式化语法
"%s can be %s the %s way" % ("strings", "interpolated", "old")

# None 是一个对象
None  # => None

# 当与 None 进行比较时不要用 ==，要用 is。is 是用来比较两个变量是否指向同一个对象。
"etc" is None  # => False
None is None  # => True

# None，0，空字符串，空列表，空字典都算是 False
# 所有其他值都是 True
bool(0)  # => False
bool("")  # => False
bool([]) #=> False
bool({}) #=> False


####################################################
## 2. 变量和集合
####################################################

# print 是内置的打印函数
print("I'm Python. Nice to meet you!")

# 在给变量赋值前不用提前声明
# 传统的变量命名是小写，用下划线分隔单词
some_var = 5
some_var  # => 5

# 访问未赋值的变量会抛出异常
# 参考流程控制一段来学习异常处理
```

```
some_unknown_var  # 抛出 NameError

# 用列表(list)储存序列
li = []
# 创建列表时也可以同时赋给元素
other_li = [4, 5, 6]

# 用 append 在列表最后追加元素
li.append(1)    # li 现在是[1]
li.append(2)    # li 现在是[1, 2]
li.append(4)    # li 现在是[1, 2, 4]
li.append(3)    # li 现在是[1, 2, 4, 3]
# 用 pop 从列表尾部删除
li.pop()        # => 3 且 li 现在是[1, 2, 4]
# 把 3 再放回去
li.append(3)    # li 变回[1, 2, 4, 3]

# 列表存取跟数组一样
li[0]  # => 1
# 取出最后一个元素
li[-1]  # => 3

# 越界存取会造成 IndexError
li[4]  # 抛出 IndexError

# 列表有切割语法
li[1:3]  # => [2, 4]
# 取尾
li[2:]  # => [4, 3]
# 取头
li[:3]  # => [1, 2, 4]
# 隔一个取一个
li[::2]   # =>[1, 4]
# 倒排列表
li[::-1]   # => [3, 4, 2, 1]
# 可以用三个参数的任何组合来构建切割
# li[始:终:步伐]

# 用 del 删除任何一个元素
del li[2]   # li is now [1, 2, 3]

# 列表可以相加
# 注意：li 和 other_li 的值都不变
li + other_li   # => [1, 2, 3, 4, 5, 6]
```

```
# 用 extend 拼接列表
li.extend(other_li)   # li 现在是[1, 2, 3, 4, 5, 6]

# 用 in 测试列表是否包含值
1 in li   # => True

# 用 len 取列表长度
len(li)   # => 6


# 元组是不可改变的序列
tup = (1, 2, 3)
tup[0]   # => 1
tup[0] = 3  # 抛出 TypeError

# 列表允许的操作元组大都可以
len(tup)   # => 3
tup + (4, 5, 6)   # => (1, 2, 3, 4, 5, 6)
tup[:2]   # => (1, 2)
2 in tup   # => True

# 可以把元组合列表解包，赋值给变量
a, b, c = (1, 2, 3)     # 现在 a 是 1，b 是 2，c 是 3
# 元组周围的括号是可以省略的
d, e, f = 4, 5, 6
# 交换两个变量的值就这么简单
e, d = d, e     # 现在 d 是 5，e 是 4


# 用字典表达映射关系
empty_dict = {}
# 初始化的字典
filled_dict = {"one": 1, "two": 2, "three": 3}

# 用[]取值
filled_dict["one"]   # => 1


# 用 keys 获得所有的键。因为 keys 返回一个可迭代对象，所以在这里把结果包在 list 里。我们下面会详细介绍可迭代。
# 注意：字典键的顺序是不定的，你得到的结果可能和以下不同。
list(filled_dict.keys())   # => ["three", "two", "one"]
```

```
# 用 values 获得所有的值。跟 keys 一样，要用 list 包起来，顺序也可能不同。
list(filled_dict.values())   # => [3, 2, 1]


# 用 in 测试一个字典是否包含一个键
"one" in filled_dict   # => True
1 in filled_dict   # => False

# 访问不存在的键会导致 KeyError
filled_dict["four"]   # KeyError

# 用 get 来避免 KeyError
filled_dict.get("one")   # => 1
filled_dict.get("four")   # => None
# 当键不存在的时候 get 方法可以返回默认值
filled_dict.get("one", 4)   # => 1
filled_dict.get("four", 4)   # => 4

# setdefault 方法只有当键不存在的时候插入新值
filled_dict.setdefault("five", 5)  # filled_dict["five"]设为 5
filled_dict.setdefault("five", 6)  # filled_dict["five"]还是 5

# 字典赋值
filled_dict.update({"four":4}) #=> {"one": 1, "two": 2, "three": 3, "four":
4}
filled_dict["four"] = 4  # 另一种赋值方法

# 用 del 删除
del filled_dict["one"]  # 从 filled_dict 中把 one 删除


# 用 set 表达集合
empty_set = set()
# 初始化一个集合，语法跟字典相似。
some_set = {1, 1, 2, 2, 3, 4}   # some_set 现在是{1, 2, 3, 4}

# 可以把集合赋值于变量
filled_set = some_set

# 为集合添加元素
filled_set.add(5)   # filled_set 现在是{1, 2, 3, 4, 5}

# & 取交集
```

```
other_set = {3, 4, 5, 6}
filled_set & other_set   # => {3, 4, 5}

# | 取并集
filled_set | other_set   # => {1, 2, 3, 4, 5, 6}

# - 取补集
{1, 2, 3, 4} - {2, 3, 5}   # => {1, 4}

# in 测试集合是否包含元素
2 in filled_set   # => True
10 in filled_set   # => False



####################################################
## 3. 流程控制和迭代器
####################################################

# 先随便定义一个变量
some_var = 5

# 这是个 if 语句。注意缩进在 Python 里是有意义的
# 印出"some_var 比 10 小"
if some_var > 10:
    print("some_var 比 10 大")
elif some_var < 10:    # elif 句是可选的
    print("some_var 比 10 小")
else:                  # else 也是可选的
    print("some_var 就是 10")


"""
用 for 循环语句遍历列表
打印:
    dog is a mammal
    cat is a mammal
    mouse is a mammal
"""
for animal in ["dog", "cat", "mouse"]:
    print("{} is a mammal".format(animal))

"""
"range(number)"返回数字列表从 0 到给的数字
打印:
```

```
    0
    1
    2
    3
"""
for i in range(4):
    print(i)

"""
while 循环直到条件不满足
打印:
    0
    1
    2
    3
"""
x = 0
while x < 4:
    print(x)
    x += 1  # x = x + 1 的简写

# 用 try/except 块处理异常状况
try:
    # 用 raise 抛出异常
    raise IndexError("This is an index error")
except IndexError as e:
    pass    # pass 是无操作，但是应该在这里处理错误
except (TypeError, NameError):
    pass    # 可以同时处理不同类的错误
else:   # else 语句是可选的，必须在所有的 except 之后
    print("All good!")   # 只有当 try 运行完没有错误的时候这句才会运行


# Python 提供一个叫做可迭代(iterable)的基本抽象。一个可迭代对象是可以被当作序列
# 的对象。比如说上面 range 返回的对象就是可迭代的。

filled_dict = {"one": 1, "two": 2, "three": 3}
our_iterable = filled_dict.keys()
print(our_iterable) # => range(1,10) 是一个实现可迭代接口的对象

# 可迭代对象可以遍历
for i in our_iterable:
    print(i)    # 打印 one, two, three
```

```
# 但是不可以随机访问
our_iterable[1]  # 抛出 TypeError

# 可迭代对象知道怎么生成迭代器
our_iterator = iter(our_iterable)

# 迭代器是一个可以记住遍历的位置的对象
# 用__next__可以取得下一个元素
our_iterator.__next__()  #=> "one"

# 再一次调取__next__时会记得位置
our_iterator.__next__()  #=> "two"
our_iterator.__next__()  #=> "three"

# 当迭代器所有元素都取出后，会抛出 StopIteration
our_iterator.__next__() # 抛出 StopIteration

# 可以用 list 一次取出迭代器所有的元素
list(filled_dict.keys())  #=> Returns ["one", "two", "three"]



####################################################
## 4. 函数
####################################################

# 用 def 定义新函数
def add(x, y):
    print("x is {} and y is {}".format(x, y))
    return x + y    # 用 return 语句返回

# 调用函数
add(5, 6)   # => 印出"x is 5 and y is 6"并且返回 11

# 也可以用关键字参数来调用函数
add(y=6, x=5)   # 关键字参数可以用任何顺序


# 我们可以定义一个可变参数函数
def varargs(*args):
    return args

varargs(1, 2, 3)   # => (1, 2, 3)
```

```
# 我们也可以定义一个关键字可变参数函数
def keyword_args(**kwargs):
    return kwargs

# 我们来看看结果是什么：
keyword_args(big="foot", loch="ness")   # => {"big": "foot", "loch":
"ness"}


# 这两种可变参数可以混着用
def all_the_args(*args, **kwargs):
    print(args)
    print(kwargs)
"""
all_the_args(1, 2, a=3, b=4) prints:
    (1, 2)
    {"a": 3, "b": 4}
"""

# 调用可变参数函数时可以做跟上面相反的，用*展开序列，用**展开字典。
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}
all_the_args(*args)   # 相当于 foo(1, 2, 3, 4)
all_the_args(**kwargs)   # 相当于 foo(a=3, b=4)
all_the_args(*args, **kwargs)   # 相当于 foo(1, 2, 3, 4, a=3, b=4)


# 函数作用域
x = 5

def setX(num):
    # 局部作用域的 x 和全局域的 x 是不同的
    x = num # => 43
    print (x) # => 43

def setGlobalX(num):
    global x
    print (x) # => 5
    x = num # 现在全局域的 x 被赋值
    print (x) # => 6

setX(43)
setGlobalX(6)
```

```python
# 函数在 Python 是一等公民
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_10 = create_adder(10)
add_10(3)   # => 13

# 也有匿名函数
(lambda x: x > 2)(3)   # => True

# 内置的高阶函数
map(add_10, [1, 2, 3])   # => [11, 12, 13]
filter(lambda x: x > 5, [3, 4, 5, 6, 7])   # => [6, 7]

# 用列表推导式可以简化映射和过滤。列表推导式的返回值是另一个列表。
[add_10(i) for i in [1, 2, 3]]  # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5]   # => [6, 7]

####################################################
## 5. 类
####################################################


# 定义一个继承 object 的类
class Human(object):

    # 类属性，被所有此类的实例共用。
    species = "H. sapiens"

    # 构造方法，当实例被初始化时被调用。注意名字前后的双下划线，这是表明这个属
    # 性或方法对 Python 有特殊意义，但是允许用户自行定义。你自己取名时不应该用这
    # 种格式。
    def __init__(self, name):
        # Assign the argument to the instance's name attribute
        self.name = name

    # 实例方法，第一个参数总是 self，就是这个实例对象
    def say(self, msg):
        return "{name}: {message}".format(name=self.name, message=msg)
```

```
    # 类方法，被所有此类的实例共用。第一个参数是这个类对象。
    @classmethod
    def get_species(cls):
        return cls.species

    # 静态方法。调用时没有实例或类的绑定。
    @staticmethod
    def grunt():
        return "*grunt*"


# 构造一个实例
i = Human(name="Ian")
print(i.say("hi"))     # 印出 "Ian: hi"

j = Human("Joel")
print(j.say("hello"))  # 印出 "Joel: hello"

# 调用一个类方法
i.get_species()   # => "H. sapiens"

# 改一个共用的类属性
Human.species = "H. neanderthalensis"
i.get_species()   # => "H. neanderthalensis"
j.get_species()   # => "H. neanderthalensis"

# 调用静态方法
Human.grunt()   # => "*grunt*"


####################################################
## 6. 模块
####################################################

# 用 import 导入模块
import math
print(math.sqrt(16))  # => 4.0

# 也可以从模块中导入个别值
from math import ceil, floor
print(ceil(3.7))  # => 4.0
print(floor(3.7))   # => 3.0

# 可以导入一个模块中所有值
```

```python
# 警告：不建议这么做
from math import *

# 如此缩写模块名字
import math as m
math.sqrt(16) == m.sqrt(16)   # => True

# Python 模块其实就是普通的 Python 文件。你可以自己写，然后导入，
# 模块的名字就是文件的名字。

# 你可以这样列出一个模块里所有的值
import math
dir(math)



####################################################
## 7. 高级用法
####################################################

# 用生成器(generators)方便地写惰性运算
def double_numbers(iterable):
    for i in iterable:
        yield i + i

# 生成器只有在需要时才计算下一个值。它们每一次循环只生成一个值，而不是把所有的
# 值全部算好。这意味着 double_numbers 不会生成大于 15 的数字。
#
# range 的返回值也是一个生成器，不然一个 1 到 900000000 的列表会花很多时间和内存。
#
# 如果你想用一个 Python 的关键字当作变量名，可以加一个下划线来区分。
range_ = range(1, 900000000)
# 当找到一个 >=30 的结果就会停
for i in double_numbers(range_):
    print(i)
    if i >= 30:
        break
```