金融计算机语言讲义

高强 (mutecamel@gmail.com) 首都经济贸易大学金融学院

第 13 课 语句 (statements) 和文件 (file) 操作

学习任务

本周是第 15 周。请一边阅读以下讲义，一边输入代码练习。本讲义特意将 Python 输出全部隐去，就是希望同学们亲手输入代码，亲眼观看程序输出，这样才能收获经验。本讲义中的练习题将构成期末闭卷考试 (占期末总评 60%) 的题库，所以请务必认真对待学习任务，不能按时完成任务者后果自负。

这一周我们学习 Python 的各种语句，以及 Python 文件操作的有关概念。

1. 表达式语句 (expression statements)

1. 在交互 (interactive) 模式 REPL (read-evaluate-print-loop) 下，一个表达式 (expression) 经常作为一个语句 (statement)，用于查看计算结果。例如：

```
>>> str  # 表达式：变量名
>>> 300*(1 + 0.05)**10  # 表达式：运算符
>>> locals()  # 表达式：函数调用
>>> '-'.join('abcd')  # 表达式：方法调用
>>> {c: ord(c) for c in set('hello')}  # 表达式：花括号围场推导式
```

   但在脚本 (script) 模式下，一个仅进行 **计算** (computation) 的表达式 (expression) 语句 (statement) 没有任何实际意义，因为计算出的新对象 (object) 没有被变量 (variable) 引用 (reference)，转眼就会被 **垃圾回收** (garbage collected)，既不发生输入输出 (IO)，也不改变内存状态。

2. 进行 **操作** (operation) 的表达式 (expression) 语句 (statement) 在交互 (interactive) 模式和脚本 (script) 模式下都有实际意义，尤其是进行输入输出 (IO) 时。例如：

```
>>> print('hello')  # 表达式语句：无论在交互模式还是脚本模式下都会在屏幕终端输出
>>> l = []  # 赋值语句
>>> l.append('a')  # 表达式语句：没有输入输出，但有操作，在交互模式和脚本模式下都有意义
```

2. 赋值语句 (=)

给表达式 (expression) 计算出的对象 (object) 取名 (name)，即建立引用 (reference)，使对象 (object) 随后不被垃圾回收 (garbage collected)，也允许随后的语句 (statements) 进一步使用或操作该对象 (object)，就需要赋值 (assignment)。Python 赋值 (assignment) 语句 (statement) 使用方法多样，下面分 9 种不同方式进行介绍。

1. **单变量赋值**。这种赋值的方式是，在等号 `=` 左边写变量名称 (variable name)，右边写表达式 (expression)，使左边变量名 (variable name) 指向右边表达式 (expression) 计算出的对象 (object)。这种方式最简单也最常用。例如：

   ```
   >>> l = dir(str)
   >>> l.append('fake')
   >>> dir(str)
   ```

   这种赋值的效果，是在当前的 **局部访问范围** (locals) 字典 (dict) 内更新 (update) 一个键值对 (key-value pair)，键 (key) 是变量名称 (variable name) 字符串 (str)，值 (value) 是变量 (variable) 所引用 (reference) 的对象 (object)。例如：

   ```
   >>> locals()  # 局部访问范围字典
   >>> 'blah' in locals()  # 局部访问范围内没有 'blah' 这个键
   False
   >>> blah  # 报错 NameError
   >>> blah = list('practice makes perfect')  # 赋值语句
   >>> blah  # 查看 blah 的内容
   >>> 'blah' in locals()  # 局部访问范围内有了 'blah' 这个键
   True
   >>> locals()['blah'] is blah  # 同一个对象
   True
   >>> locals().update([('blah', 123)])  # 以软编码的方式赋值
   >>> blah  # 查看 blah 的内容，已变为 123
   ```

2. **多变量赋相同值**。这种赋值 (assignment) 方式就是连写多个等号 `=`，把最右边的等号 `=` 右边的表达式 (expression) 计算结果赋给多个等号 `=` 左边的变量名 (variable name)。这种方式并不很常用，因为一个对象 (object) 通常只要有一个引用 (reference) 就够了。这种赋值 (assignment) 方式一般常用于不可变 (immutable) 对象。例如：

   ```
   >>> x = y = z = {'a': 1, 'b': 2, 'c': 3}  # 语法允许，但没必要
   >>> x is y is z  # is 也可以连用
   >>> x = y = z = 0  # 同时初始化三个变量
   >>> x == y == z  # == 也可以连用
   ```

3. **多变量赋不同值**。这种赋值方式，在等号 `=` 左边用逗号 (comma) 分隔多个变量名 (variable names)，右边写一个可迭代 (iterable) 对象，把迭代 (iterate) 取出的值逐个赋给等号 `=` 左边的变量 (variables)，变量的个数与迭代出的对象的个数必须相等。例如：

   ```
   >>> x, y = 2, 8  # 未加括号的元组
   ```

```
>>> x, y
>>> x, y = y, x  # y, x 打包成元组，赋值给 x, y，是常用的变量交换技巧
>>> i = list('abc')  # 列表
>>> x, y, z = i  # 列表解包赋值
>>> i = set('abc')  # 集合
>>> x, y, z = i  # 集合解包赋值，但顺序不保证
>>> i = {ord(c): c for c in 'abc'}  # 字典
>>> i  # 查看字典 i
>>> x, y, z = i  # 字典解包赋值，不保证顺序
>>> x, y, z  # 查看 x, y, z，迭代只取出 keys
>>> x, y, z = i.values()  # 视图解包赋值
>>> x, y, z  # 查看 x, y, z，迭代只取出 values
>>> i = (c*5 for c in 'abc')  # 生成器
>>> x, y, z = i  # 生成器解包赋值
>>> list(i)  # 生成器被消耗干净
>>> x, y, z = i  # 迭代取出的对象不够用来赋值
```

4. **等号左边 `*` 打包**。在等号 `=` 左边的某 一个 变量 (variable) 前加 `*`，可以用来把等号 `=` 右边的多个值打包成列表 (list) 赋值给该变量 (variable)。等号 `=` 左边 `*` 只能使用一次，否则会有歧义。例如：

```
>>> s = 'abcde'
>>> *x, y, z = s
>>> x
>>> x, *y, z = s
>>> y
>>> x, y, *z = s
>>> z
>>> *x, y, *z = s  # 有歧义，语法错误
```

5. **等号右边 `*` 解包**。在等号 `=` 右边的变量前加 `*`，使用的是解包运算符的概念。例如：

```
>>> s = '123'
>>> *x, y = 0, s
>>> x  # 查看 x 的内容
>>> y  # 查看 y 的内容
>>> *x, y = 0, *s
>>> x  # 查看 x 的内容
>>> y  # 查看 y 的内容
```

6. **给对象属性赋值**。对 Python 解释器 (interpreter) 内置 (built-in) 的少数基本 (basic) 类型 (class) 及其实例 (instance) 的属性 (attribute) 不能赋值，但对大多数 Python 定义的类型 (class) 及其实例 (instance) 的属性 (attribute) 可以赋值。赋值的方法是在等号 `=` 左边写对象的属性 (attribute) 访问 (reference) 表达式 (expression)。例如：

```
>>> s = 'abc'
>>> s.token = 'secret'  # 字符串是基本内置类型，不能给字符串的属性赋值
>>> import os
>>> os.token  # os 模块没有 token 属性
>>> os.token = 'secret'  # 可以给 os 模块的属性赋值
>>> os.token  # 查看 os 模块的 token 属性
>>> class Dog:  # 定义一个 Dog class
...     pass  # 占位语句，什么也不做，以便符合语法要求
...
>>> dog = Dog()  # 初始化一个 dog 实例
>>> dog.name = 'Max'  # 给 dog 实例的属性赋值
>>> dog.name  # 查看 dog 实例的 name 属性
```

给属性赋值还有另一种 **软编码** (soft coded) 方法，即使用 `setattr` 内置函数 (built-in function)。例如：

```
>>> setattr(dog, 'age', 2)
>>> dog.age  # 查看 dog 实例的 age 属性
```

7. **给对象提取赋值**。在等号 `=` 左边写对象 (object) 提取 (subscription) 表达式 (expression)，可以给提取 (subscripting) 的元素 (element) 赋值。当然，前提是该对象 (object) 可变 (mutable)。例如：

```
>>> s = 'hello'
>>> s[1]
>>> s[1] = 'E'  # 字符串是不可变类型，提取赋值失败
>>> q = list(s)
>>> q[1] = 'E'  # 列表是可变类型，可以提取赋值
>>> q
>>> s = ''.join(q)
>>> s
>>> d = {}
>>> d['name'] = 'John'  # 字典是可变类型，用提取赋值给字典增加键值对是非常典型的用法
>>> d  # 查看 d 的内容
```

8. **给对象切片赋值**。在等号 `=` 左边写对象 (object) 切片 (slicing) 表达式 (expression)，右边写一个可迭代 (iterable) 对象，可以给切片 (slicing) 中的元素 (elements) 逐一赋值。当然，前提是该对象 (object) 是可变的 (mutable)。例如：

```
>>> s = 'hello'
>>> s[1:4]
>>> s[1:4] = 'ELL'  # 字符串是不可变类型，切片赋值失败
>>> q = list(s)
>>> q[1:4]
>>> q[1:4] = 'ELL'  # 列表可以切片赋值，'ELL' 是可迭代
>>> ''.join(q)
>>> q = list(s)
>>> q[::2] = 'HLO'  # 也可以跳跃赋值
>>> ''.join(q)
```

9. **通过运算符赋值**。把运算符 (operator) 和等号 `=` 连用，例如 `+=`，效果是先使用变量 (variable) 进行运算，再把运算的结果重新赋值 (assign) 给变量 (variable)。例如：

```
>>> i = 5
>>> i += 3  # 等价于 i = i + 3
>>> i
>>> s = '-'  # s 不可变没有关系，可以重新给 s 赋值
>>> s *= 20
>>> s
```

   解释器 (interpreter) 所支持的赋值 (assigning) 运算符 (operators) 共包括：`**=`、`*=`、`/=`、`@=`、`//=`、`%=`、`+=`、`-=`、`<<=`、`>>=`、`&=`、`^=`、`|=` 等。

3. 删除引用语句 (`del`)

`del` 语句 (statement) 与赋值 (assignment) 语句 (statement) 恰好相反，用于删除 (delete) 已经建立好的对象 (object) 引用 (reference) 关系。相对而言，`del` 语句更多用于从列表 (list) 或字典 (dict) 中删除元素。

1. **删除变量**。此用法是在 `del` 关键词 (keyword) 后写一个或多个变量名 (variable names)，用于从当前局部访问范围 (local scope) 内删除这些变量 (variable)。例如：

```
>>> x, y, z = 'a', 42, [7, 2, 9]
>>> 'x' in locals()
>>> del x  # 删除一个变量
>>> 'x' in locals()
>>> x  # 报错 NameError
>>> del y, z  # 同时删除两个变量
```

   由于 Python 解释器 (interpreter) 具有垃圾回收 (garbage collection) 功能，通常很少有必要明确 (explicitly) 删除 (delete) 变量 (variable)，因此 `del` 语句这样使用的频率并不高。

2. **删除属性**。此用法是在 `del` 关键词 (keyword) 后写属性访问 (attribute access) 表达式 (expression)，用于从对象上删除某一属性。例如：

```
>>> import os
>>> 'path' in dir(os)  # os 模块具有 path 属性
>>> type(os.path)  # path 是 os 模块下的子模块
>>> del os.path  # 从 os 对象上删除 path 属性
>>> os.path  # 报错 AttributeError
```

除非是在使用 Python 进行 **元数据编程** (meta-programming)，否则一般没必要从对象 (object) 上删除某个属性 (attribute)，因此 `del` 语句的这种用法很不常用。

3. **删除提取**。此用法是在 `del` 关键词 (keyword) 后写提取 (subscription) 表达式 (expression)，用于从可变 (mutable) 对象 (object) 中删除提取 (subscription) 位置所对应的元素 (element)。例如：

```
>>> q = [4, 2, 6]
>>> del q[1]  # 从列表指定索引处删除元素
>>> q
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> del d['b']  # 从字典指定键处删除键值对
>>> d
```

这种通过 `del` 语句对可提取的 (subscriptable) 可变 (mutable) 对象 (object) 进行操作，在实际中相对而言是比较常见的。可以产生类似效果的是调用 (call) 列表 (list) 或字典 (dict) 的 `pop` 方法 (method)。

4. **删除切片**。此用法是在 `del` 关键词 (keyword) 后写切片 (slice) 表达式 (expression)，用于从可变 (mutable) 对象 (object) 中删除切片 (slice) 位置所对应的元素 (element)。例如：

```
>>> q = list('abcdefg')
>>> q
>>> del q[::2]
```

4. 导入语句 (`import`)

导入 (import) 语句 (statement) 用于从 Python 路径 (`sys.path`) 下寻找软件包 (package) 或者模块 (module)，加载 (load) 其代码，并建立一个变量 (variable) 用于引用 (reference) 所加载的软件包 (package)、模块 (module)、类 (class)、函数 (function) 或其他任何对象 (object)。例如：

```
>>> import os
>>> os  # 模块
>>> import os.path
>>> os.path  # 模块
>>> import os.path as path
>>> path  # 模块
>>> import pandas as pd  # 导入并赋值 (若导入失败则需要在终端运行 pip install pandas 先进行安装)
>>> pd  # 软件包的模块文件名为 __init__.py
```

```
>>> from collections import deque
>>> deque  # 类
>>> from copy import deepcopy
>>> deepcopy  # 函数
>>> from sys import path
>>> path  # 寻找软件包或模块的路径列表
>>> print(*sys.path, sep='\n')
>>> from sympy import *  # 尽量少用，一般来说，相当于扩充内置函数
```

导入 (import) 语句 (statement) 经常写在 `.py` 文件的最开始，用于从 **Python 标准库** (Python standard library) 或 **第三方库** (third-party libraries) 导入功能更丰富的各种对象，”避免重新发明轮子” (don’t reinvent the wheel)。

下面的语句都是跟流程控制 (flow control) 有关。

5. 迭代语句 (`for`)

`for` 语句 (statement) 是一种复合 (compound) 语句 (statement)，对可迭代 (iterable) 对象进行迭代 (iterate)，每迭代 (iterate) 取出下一个 (next) 对象 (object)，命名为迭代变量 (iterating variable)，就执行一遍所复合的 (compound) 语句 (statements)。复合的 (compounded) 语句 (statements) 要求缩进 (indent)，且要保持相同的缩进量 (indentation) (通常为 4 个空格)，否则为语法错误。例如：

```
>>> import random
>>> x = 0
>>> for i in range(10):  # 不要忘记冒号
...     x += random.gauss(0, 1)
...     print(x)
```

这个例子的作用是生成 10 个标准正态分布的随机数，依次累加到变量 `x` 上，每一步都把结果 `print` 在终端 (terminal) 上。

6. 循环语句 (`while`)

`while` 语句 (statements) 也是一种复合 (compound) 语句 (statement)，对 `while` 关键词 (keyword) 后的表达式 (expression) 进行逻辑判断 (truth-value testing)，若为逻辑真 True 则执行复合的 (compounded) 语句 (statements)，执行完后再次对 `while` 后的表达式 (expression) 进行逻辑判断……直至判断出逻辑假 False，才不再执行复合的 (compounded) 语句 (statements)，结束循环。例如：

```
>>> import random
>>> x = 10
>>> while x > 1:
>>>     print('o'*(79 if x >= 79 else int(x)))
>>>     x += random.gauss(0, 1)
```

这个例子的作用是不断生成标准正态分布的随机数，依次累加到变量 `x` 上，每一步都在一行上 `print` 出 `x` 个 `'o'` (最多 79 个)，直至 `x <= 1` 才结束循环。

7. 条件语句 (`if`)

`if` 语句 (statement) 也是一种复合 (compound) 语句 (statement)，对 `if` 关键词 (keyword) 后的表达式 (expression) 进行逻辑判断 (truth-value testing)，若为逻辑真 True 则执行复合的 (compounded) 语句 (statements)，若为逻辑假 False 则不执行。例如：

```
>>> count = 0
>>> for a in dir(str):
...     if not a.startswith('_'):
...         count += 1
...         print(str(count).ljust(2), a)
```

这个例子的作用是用 `count` 计数，并输出 `str` 类型 (class) 的所有普通属性 (attributes) (不以 `'_'` 开始) 的名称 (names)。

8. 条件从句 (`elif`)

在 `if` 语句 (statement) 后可以使用多个 `elif` 从句 (clause)，建立起多条件分支 (branch) 的执行流程。例如：

```
>>> age = 12
>>> if age < 4:
...     price = 0
... elif age < 18:
...     price = 5
... elif age < 65:
...     price = 10:
... elif age >= 65:
...     price = 5
...
>>> print('Your ticket price is ${}.'.format(price))
```

这个例子是根据年龄 `age` 计算门票价格 `price`。若年龄小于 4 岁则免票；否则，若年龄小于 18 岁则半票；否则，若年龄小于 65 岁则全票；否则，若年龄大于等于 65 岁则半票。

9. 其他从句 (`else`)

在 `if` 语句的最后还可以使用一个 `else` 从句 (clause)，用于在前面的 `if` 和/或 `elif` 判断都不满足的情况下执行某些复合 (compounded) 语句 (statements)。例如：

```
>>> age = 17
>>> if age >= 18:
...     print('You are old enough to vote!')
... else:
...     print('Sorry, you are too young to vote.')
```

这个例子根据年龄 `age` 是否大于等于 18，或者执行第一个 `print`，或者执行第二个 `print`。

10. 循环跳过语句 (`continue`)

`continue` 语句 (statement) 只能在 `for` 语句 (statement) 或 `while` 语句 (statement) 的复合 (compounded) 语句 (statements) 里使用，作用是跳过最近的一层循环，进入下一轮循环重新执行复合 (compounded) 语句 (statements)。例如：

```
>>> for i in range(128):
...     c = chr(i)
...     if not c.isprintable():
...         continue
...     print(i, c, end='\t')
```

这个例子的作用是 `print` 出 1～127 的 ASCII 码范围内的所有可打印字符。

11. 循环跳出语句 (`break`)

`break` 语句 (statement) 只能在 `for` 语句 (statement) 或 `while` 语句 (statement) 的复合 (compounded) 语句 (statements) 里使用，作用是跳出最近的一层循环，执行复合 (compounded) 语句 (statements) 下面的语句 (statements)。例如：

```
>>> while True:
...     s = input('Input: ')
...     if s == '':
...         break
...     print('Echo:  ' + s)
```

这个例子的作用是不断 `print` 用户输入的内容，直到用户直接输入回车为止。

12. 循环语句的其他从句 (`else`)

在 `for` 语句 (statement) 或 `while` 语句 (statement) 之后还可以跟 `else` 从句 (clause)，该从句 (clause) 下的复合 (compounded) 语句 (statements) 只在循环正常完成的情况下才执行，若循环是从 `break` 语句跳出结束的，则不执行。这种句式主要用于通过循环 (`for` 或者 `while`) 寻找某个条件 (`if`) 跳出 (`break`)，若找不到 (未跳出) 才执行 `else` 从句 (clause) 的内容。例如：

```
>>> q = ['print', '__builtins__', 'Exception']
>>> for n in q:
...     if not n.isidentifier():
...         print('Not an identifier: ', n)
...         break
... else:
...     print('All identifiers verified.')
```

这个例子的作用是从列表 `q` 中寻找不是标识符 (identifier) 的字符串，若找到则 `print` 一种结果，若找不到则 `print` 另一种结果。下面的例子显示的就是另一种结果：

```
>>> q = ['print', '__builtins__', 'Exception', 'os.path']
>>> for n in q:
...     if not n.isidentifier():
...         print('Not an identifier: ', n)
...         break
... else:
...     print('All identifiers verified.')
```

13. 异常捕捉语句 (`try`)

`try` 语句 (statement) 也是一种复合 (compound) 语句 (statement)，必须跟至少一个 `except` 从句 (clause)，可以跟一个 `else` 从句 (clause)，也可以跟一个 `finally` 从句 (clause)。`try` 语句的作用是捕捉异常 (exception)，如果复合 (compounded) 语句 (statements) 执行过程中报错 (raise)，且这个异常 (exception) 在某个 `except` 从句 (clause) 中已有指定要求捕捉，那么解释器 (interpreter) 将取消 (cancel) 这个异常 (exception) 的报错 (raise)，转而执行 `except` 从句 (clause) 里的复合 (compounded) 语句 (statements)。如果 `try` 语句中没有报错 (raise exception)，则执行 `else` 从句 (clause) 里的复合 (compounded) 语句 (statements)。无论 `try` 语句有没有报错 (raise exception)，只要指定有 `finally` 从句 (clause)，都会执行里面的复合 (compounded) 语句 (statements)，这些语句 (statements) 通常被用于手动释放不能被解释器 (interpreter) 垃圾回收 (garbage collection) 的系统资源，例如关闭文件或数据库连接。例如：

```
>>> s = 'ok'
>>> int(s)  # 注意报错 ValueError
>>> try:
...     x = int(s)
... except ValueError:  # 注意 可自行验证
...     print('s cannot be converted to integer')
```

```
...
>>> x  # 变量 x 未赋值成功
>>> 'ValueError' in dir(__builtins__)  # ValueError 是一个 built-in name
>>> [
...     n for n, o in vars(__builtins__).items()  # vars 函数的返回值是字典
...     if isinstance(o, type) and issubclass(o, BaseException)  # 要求 o 是类且是 BaseException 的子类
... ]
>>> # 若以上替换成 issubclass(o, BaseException) and isinstance(o, type) 则会报错，请考虑为什么
>>> try:
...     x = int(s)
... except KeyError:
...     print('This will not happen.')
...
>>> # 由于捕捉的是 KeyError，x = int(s) 发生的 ValueError 仍旧报错
>>> try:
...     x = int('100')
... except ValueError:
...     print('This will not happen.')
... else:
...     print('x =', x)
>>> # try 语句内不报错则执行 else 从句
>>> f = open('file1.txt', mode='w')
>>> f.write(321)  # 报错 TypeError
>>> try:
...     f.write(321)
... except BaseException:  # BaseException 是所有异常的基类
...     print('Error in writing to file.')
... finally:
...     f.close()  # 不论 try 中是否发生异常，都会执行此语句释放资源
...
>>> issubclass(TypeError, Exception)
```

14. 情境语句 (`with`)

`with` 语句 (statement) 也是一种复合 (compound) 语句 (statement)，后面必须跟一个情境管理器 (context management)。所谓情境管理器 (context manager)，是指具有 `__enter__` 和 `__exit__` 方法 (method)，由 `__enter__` 方法 (method) 完成资源申请或环境准备工作并返回一个进入 (enter) 情境 (context) 内部以后可以使用的对象 (object)，由 `__exit__` 方法 (method) 负责在情境 (context) 退出 (exit) 时进行资源释放或环境清理工作。不论 `with` 语句 (statement) 是正常完成还是异常退出，`__exit__` 方法 (method) 总会执行，从而确保情境 (context) 恢复到进入前的状态。

文件对象 (file object) 就是一个非常典型的情境管理器 (context manager)。在 `with` 语句 (statement) 里使用文件对象 (file object)，可以确保不论正常执行还是异常退出，文件资源都一定会被释放。支持情境管理 (context manager) 是文件对象 (file object) 本身就定义好的功能。例如：

```
>>> f = open('file2.txt', mode='w')
>>> '__enter__' in dir(f)
>>> '__exit__' in dir(f)
>>> f.close()
>>> with open('file2.txt', mode='w') as fobj:  # 情境语句的使用方式
...     fobj.write('abc')
...
>>> # 在 with 情境下使用 fobj 就不用担心 fobj 的关闭问题
>>> fobj.closed  # 情境外 fobj 肯定会被自动关闭
```

一般来说，内存 (memory) 资源都可以被垃圾回收 (garbage collected)，但磁盘 (dict)、网络 (network) 等外部资源都无法被垃圾回收 (garbage collected)，所以这些资源经常结合着 `with` 语句 (statement) 使用，以达到资源自动释放的目的。其实通过 `try` 语句 (statement) 和 `finally` 从句 (clause) 也可以达到类似的效果，但不如 `with` 语句简单直观。例如，上面的代码基本上等价于：

```
>>> fobj = open('file2.txt', mode='w')
>>> try:
...     fobj.write('abc')
... finally:
...     fobj.close()
```

15. 断言语句 (`assert`)

`assert` 语句 (statement) 是编程中的一种简单调试 (debug) 工具，用于确保某条件判断成立，若不成立，则报错 `AssertionError`，若成立，则什么也不发生。

1. **简单断言**。在 `assert` 关键词 (keyword) 之后写任意表达式 (expression)，会对该表达式 (expression) 进行逻辑判断 (truth-value testing)，若为逻辑假 False 则报错。例如：

   ```
   >>> s = 'abc'
   >>> assert len(s) == 4  # 逻辑判断 False，报错 AssertionError
   >>> assert s.startswith('a')  # 逻辑判断 True，什么也不发生
   ```

2. **附带对象断言**。在 `assert` 关键词 (keyword) 之后写两个表达式 (expression)，用逗号 (comma) 隔开，第一个表达式进行逻辑判断 (truth-value testing)，第二个表达式为逻辑假 False 时异常 (exception) 对象 (object) 所附带的对象 (object)。例如：

   ```
   >>> s = set('abc')
   >>> import collections.abc
   >>> assert isinstance(s, collections.abc.Sequence), type(s)  # 报错附带 type(s)
   ```

16. 报错语句 (`raise`)

`raise` 语句 (statement) 也有助于调试 (debug)，同时与 `try` 语句结合使用也是控制程序流程的一种有力工具。可以使用 `if` 语句 (statement) 判断某种条件，满足条件则用 `raise` 语句 (statement) 报错，可以主动地终止 Python 程序运行，避免继续计算出无意义的结果。例如：

```
>>> raise NameError
>>> raise RuntimeError('my error infomation')
>>> v = input('Your number: ')
>>> if not v.isdecimal():
...     raise ValueError('Input number must be a positive integer.')
```

17. 占位语句 (`pass`)

`pass` 语句 (statement) 在执行 (execute) 时什么也不做，它的存在只是为了满足 Python 基于缩进 (indentation) 的复合 (compounding) 语法 (syntax) 要求。例如：

```
>>> s = 'ok'
>>> try:
...     x = int(s)
... except ValueError:
...     pass  # 语法要求必须写这个语句才能什么也不做
...
>>> print(x)  # 对 x 进行的赋值语句没有运行成功
```

这个例子能够捕捉 `int(s)` 中报错 (raise) 的 `ValueError`，但在 `except` 从句里什么也不做，相当于只是忽略这个异常 (exception)。

但上面的例子里不写 `pass` 语句，则会报错 (raise) 缩进错误 (`IndentationError`)。例如：

```
>>> s = 'ok'
>>> try:
...     x = int(s)
... except ValueError:
...
...
>>> # 报错 IndentationError
```

18. 文件对象

此处讨论的文件 (file)，仅限于保存在磁盘 (disk) 上的的文件。Python 解释器 (interpreter) 解释 Python 代码而进行的一切活动，归根到底只是一次进程 (process)；进程运行中需要输入 (input) 的数据通常要来自于文件读取 (read)，而需要输出 (output) 的数据通常要被写入 (write) 至文件；不进行文件读写 (I/O) 操作的 Python 进程几乎不会有任何 *实际* 用途。所以，理解清楚文件读写的有关概念、熟练掌握文件读写的基本操作，非常必要。

18.1 创建

文件 (file) 位于 *磁盘* (disk)，Python 需要根据文件 (file) 处于文件系统 (file system) 的路径 (path)，在 *内存* 中创建出一个指向该路径 (path) 的对象 (object)，借此对象 (object) 的 `read` 和 `write` 方法 (method) 来完成对磁盘文件的读写 (I/O) 操作 (operation)。这样的对象叫做 **文件对象** (file object)，通常通过内置函数 (built-in function) `open` 来创建。

```
>>> dir(__builtins__)  # 可以看到内置模块 __builtins__ 确实包含有 open 这个属性
>>> open  # 可以看到 open 确实是一个内置变量 (built-in variable)，这里没有报告 NameError
>>> help(open)  # 建议强迫自己读一读官方文档，既能提高英语水平，又能提高自学能力；读不懂没关系，坚持读

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
...
```

文档第一行显示的就是函数 (function) 的 **签名** (signature)，其中 `open` 是函数 (function) 名称 (name)，`file`, `mode`, `buffering`, `encoding`, `errors`, `newline`, `closefd`, `opener` 等是形参 (parameter) 名称 (name)。

调用 (call) `open` 函数 (function) *至少* 需要给形参 (parameter) `file` 传递一个字符串 (str) 实例 (instance) 作为实参 (argument)，用来指定需要打开的文件 (file) 的路径 (path)：

```
>>> open()  # 缺少必要参数
>>> open('a')  # 无效路径，文件 a 不存在
>>> open('/etc/passwd')  # 有效文件路径，取得文件对象
>>> help(_)  # 查看文件对象 (file object) 的文档，可以看到创建的是 TextIOWeapper 类型的对象
```

18.2 模式

调用 (call) `open` 函数 (function) 创建文件对象 (file object) 时，*可以* 给形参 (parameter) `mode` 传递一个字符串 (str) 实例 (instance) 作为实参 (argument)，用来指定打开文件 (file) 的 **模式** (mode)。可选择的文件模式 (mode) 有以下几种：

`mode` 实参	说明
`'r'` 或 `'rt'`	只读 (read-only) 文本 (text) 模式，支持文本读取，不支持文本写入
`'w'` 或 `'wt'`	只写 (write-only) 文本 (text) 模式，支持文本写入，不支持文本读取，删除已有文件
`'x'` 或 `'xt'`	只写 (write-only) 文件 (text) 模式，支持文本写入，不支持文本读取，已有文件报错
`'a'` 或 `'at'`	只写 (write-only) 文件 (text) 模式，支持文本写入，不支持文本读取，已有文件追加
`'r+'` 或 `'r+t'`	读写 (read-and-write) 文本 (text) 模式，同时支持读取和写入
`'w+'` 或 `'w+t'`	读写 (read-and-write) 文本 (text) 模式，同时支持读取和写入，删除已有文件
`'rb'`	只读 (read-only) 二进制 (binary) 模式，支持字节读取，不支持字节写入
`'wb'`	只写 (write-only) 二进制 (binary) 模式，支持字节写入，不支持字节取，删除已有文件
`'xb'`	只写 (write-only) 二进制 (binary) 模式，支持字节写入，不支持字节读取，已有文件报错
`'ab'`	只写 (write-only) 二进制 (binary) 模式，支持字节写入，不支持字节读取，已有文件追加
`'r+b'`	读写 (read-and-write) 二进制 (binary) 模式，同时支持读取和写入
`'w+b'`	读写 (read-and-write) 二进制 (binary) 模式，同时支持读取和写入，删除已有文件

简单来说，`'r'` 代表 “只读”，`'w'` 代表 “删除只写”，`'x'` 代表 “创建只写”，`'a'` 代表 “追加只写”，`'r+'` 代表 “可读可写”，`'w+'` 代表 “删除读写”，`'t'` 代表 “文本” (默认值，可省略)，`'b'` 代表 "二进制”。由于我们以操作文本为主，下面对每种文本模式各举一个例子：

以只读 (read-only) 文本 (text) 模式打开文件：`mode='r'` (可省略)

```
$ echo 'This is my text.' > file1.txt  # 创建文件，内容为文本 This is my text.
$ cat file1.txt  # 查看 file1.txt 的文本内容
$ python3  # 启动 Python 解释器进入交互模式
>>> f = open('file1.txt')  # 创建文件对象，以只读文本模式打开 file1.txt 文件
>>> text = f.read()  # 调用 f 对象的 read 方法读取文本，让名称 text 指向其返回值
>>> print(text)  # 在终端输出 text 所指向的字符串实例
>>> f.write('\nHere is another line.')  # 调用 f 对象的 write 方法会报错，因为只读
>>> import os  # 导入 os 内置模块
```

```
>>> help(os.path.expanduser)  # 查看函数 os.path.expanduser 的文档
>>> f = open(os.path.expanduser('~/.bash_history'))  # 创建文件对象
>>> print(f.read())  # 在终端输出 ~/.bash_history 文件的文本内容
>>> f.close()  # 关闭文件，释放从操作系统申请的资源
>>> f.read()  # 再次调用 read 方法会报错，因为文件已关闭，向操作系统申请的资源已释放
>>> print(os.__file__)  # 在终端输出 os 模块的 .py 源码文件路径
>>> f = open(os.__file__)  # 创建文件对象，以只读文本模式打开 os 模块的源代码
>>> text = f.read()  # 读取文本
>>> print(text)  # 在终端输出 os 模块的源代码
```

磁盘文件是一种资源，Python 进程需要向操作系统申请 (`open`) 并成功才能获得，使用完毕之后需要 *手动* 释放 (`close`)。Python 解释器 (interpreter) 虽然有 **垃圾回收** (garbage collection, GC) 功能，但回收的仅限于内存对象 (object)，并不包括像文件 (file) 这样的外部资源。Python 解释器 (interpreter) 进程 (process) 虽然在退出时会释放所有已申请但尚未释放的资源，但还是建议 Python 程序员自己，在使用完毕后及时释放资源，以免资源被过久占用，从而影响其他进程 (process) 运行。

以只写 (write-only) 文本 (text) 模式打开文件，删除已有文件：`mode='w'`

```
>>> f = open('file2.txt', mode='w')  # 创建文件对象，以只写文本模式打开 file2.txt 文件
>>> f.write('Here is my title')  # 调用 f 对象的 write 方法写入文本，返回写入的 Unicode 字符数

# 启动另外一个终端，Python 解释器所在的终端不要关闭
$ cat file2.txt  # 查看 file2.txt 的文本内容，结果显示为空，并没有真正写入文件

# 切换回 Python 解释器所在的终端
>>> f.close()  # 调用 f 对象的 close 方法，关闭文件，缓存在文件对象内的文本内容此时才真正写入磁盘

# 切换至另一个终端
$ cat file2.txt  # 查看 file2.txt 的文本内容，结果显示 Here is my title，说明文本写入成功

# 切换回 Python 解释器所在的终端
>>> f2 = open('file2.txt', mode='w')  # 创建另一个文件对象，以只写文本模式打开 file2.txt 文件

# 切换至另一个终端
$ cat file2.txt  # 查看 file2.txt 的文本内容，结果显示为空，说明现有文件已被删除并重新创建

# 切换回 Python 解释器所在的终端
>>> f2.write('\u6b64\u81f4')  # 写入文本，使用了字符串字面值的 unicode 转义，返回写入的字符数
>>> f2.write(', ')  # 再次写入文本，将在之前的文本缓存后面追加，返回写入的字符数
>>> f2.write('\n\t\t\u656c\u793c')  # 继续追加文本，返回写入的字符数
>>> f2.read()  # 调用 f 对象的 read 方法会报错，因为只写
>>> f2.close()  # 关闭文件，并把保存在文件对象内的文本缓存写入磁盘
```

```
25
26  # 切换至另一个终端
27  $ cat file2.txt  # 查看 file2.txt 的文本内容，文本写入成功
```

需要注意，文件对象 (file object) 是在 *内存* 而并不是在 *磁盘*。调用 `write` 方法只是改变文件对象 (file object) 所持有的 **缓存** (buffer)，*并不必然* 写入到磁盘。只有在缓存 (buffer) 满后，文件对象 (file object) 才会 *自动* 向操作系统申请写入磁盘，并清空缓存 (buffer)。设立缓存 (buffer) 的目的是减少 I/O 调用次数，因为磁盘 (disk) 读写的速度相对于内存 (memory) 读写的速度而言要慢许多倍 (约几十倍)。若要 *手动* 将缓存 (buffer) 写入磁盘并清空，可以调用文件对象 (file object) 的 `flush` 方法 (`close` 方法在关闭文件对象之前，其实会自动调用 `flush` 方法)。

```
>>> f3 = open('file3.txt', mode='w')  # 创建文件对象，以只写文本模式打开 file3.txt 文件
>>> f3.write('Welcome to')  # 调用 write 方法，写入到缓存

# 切换至另一个终端
$ cat file3.txt  # 查看 file3.txt 的文本内容，结果显示为空，说明缓存尚未写入磁盘

# 切换回 Python 解释器所在的终端
>>> f3.flush()  # 调用 flush 方法，把缓存写入磁盘并清空，文件并不关闭，可以继续写入

# 切换至另一个终端
$ cat file3.txt  # 查看 file3.txt 的文本内容，结果显示 Welcome to，说明文本写入成功

# 切换回 Python 解释器所在的终端
>>> f3.write(' China.') 再次调用 write 方法，可以继续写入缓存
>>> f3.close()  # 关闭文件，写入磁盘
>>> f3.write()  # 再次调用 write 方法会报错，因为文件已关闭，向操作系统申请的资源已释放
```

以只写 (write-only) 文本 (text) 模式打开文件，已有文件报错：`mode='x'`

因为 `mode='w'` 模式会首先删除已有文件，为防止误删已有文件，可以选择 `mode='x'` 模式，在遇到已有文件的情况下会报告 (raise) FileExistsError 错误：

```
>>> f3 = open('file3.txt', mode='w')
>>> f3.write('Text exists.')
>>> f3.close()
>>> f = open('file3.txt', mode='x')  # 因为 file3.txt 已存在，所以会报错
>>> quit()
$ cat file3.txt  # 查看 file3.txt 文件的文本，看到内容并没有被删除
Text exists.
```

以只写 (write-only) 文本 (text) 模式打开文件，追加已有文件：`mode='a'`

如果写入文本文件前不希望删除已有内容，而是希望在任何已有内容的后面继续追加 (append) 文本，可以选择 `mode='a'` 模式。若文件原本不存在，则 `'a'` 模式与 `'w'` 模式的行为将完全相同。

```
>>> f3 = open('file3.txt', mode='w')
>>> f3.write('Text exists.')
>>> f3.close()
>>> f = open('file3.txt', mode='a')  # 追加模式
>>> f.write(' ')
>>> f.write('continue...\nfor another line.')
>>> f.close()
>>> quit()
$ cat file3.txt  # 查看 file3.txt 文件的文本，看到原内容之后有新追加的文字
Text exists. continue...
for another line.
```

以读写 (read-and-write) 文本 (text) 模式打开文件：`mode='r+'`

读写 (read-and-write) 模式仅用于打开文件后 *同时* 需要读取 (read) 和写入 (write) 的情形。这种情形其实非常少见。就算是 *修改 已有* 文件，通常也是分成三步来做：1. 读取文字，2. 修改文字，3. 写入文字，并不需要 *同时* 读写。例如：

```
$ cat file3.txt
Text exists. continue
for another line.

$ python3
>>> # 步骤1. 读取文件
>>> f = open('file3.txt')
>>> text = f.read()
>>> f.close()  # 读取完毕之后应该立刻关闭文件释放资源
>>> # 步骤2. 修改文本
>>> text  # 文件关闭不影响内存中的 text 字符串对象
>>> text = text.upper()  # 调用 text 对象的 upper 方法，返回变为大写的新对象，并重新命名为 text
>>> text  # 查看 text 名称现在所指的对象
>>> # 步骤3. 写入文件
>>> f = open('file3.txt', mode='w')
>>> f.write(text)
>>> f.close()  # 写入文件到磁盘并关闭文件释放资源
>>> quit()

$ cat file3.txt
TEXT EXISTS. CONTINUE
FOR ANOTHER LINE.
```

确实需要 *同时* 读写的时候，可以使用 `mode='r+'` 模式。

```
>>> f = open('file3.txt', mode='r+')
>>> f.read()  # 调用 read 方法不报错
>>> f.write(' Bye.')  # 调用 write 方法也不报错
>>> f.close()
>>> quit()

$ cat file3.txt
TEXT EXISTS. CONTINUE
FOR ANOTHER LINE. Bye.
```

以读写 (read-and-write) 文本 (text) 模式打开文件，删除已有文件：`mode='w+'`

`mode='w+'` 与 `mode='r+'` 的区别仅在于前者会删除已有文件。

```
$ cat file3.txt
TEXT EXISTS. CONTINUE
FOR ANOTHER LINE. Bye.

$ python3
>>> f = open('file3.txt', 'w+')
>>> f.read()  # 调用 read 方法不报错，但读不出任何字符，因为现有文件已被删除
>>> f.write('hi')  # 调用 write 方法也不报错
>>> f.close()
>>> quit()

$ cat file3.txt
hi
```

18.3 读写位置

构成磁盘文件的实际上是二进制 (binary) 字节串 (byte stream)。文件对象 (file object) 在进行读写操作 (read/write) 时会保持着读写到的 **位置** (position)，即读取 (read) 或写入 (write) 到了第几个 **字节** (byte)。位置 (position) 总是从 0 开始，每读取 (read) 或写入 (write) 一个字符 (character)，位置 (position) 就会根据所读写的字节 (bytes) 数增加若干，具体增加几取决于所使用的字符编码 (encoding)。用文件对象 (file object) 的 `tell` 方法可以获取当前位置 (position)，用 `seek` 方法可以移动位置 (position) 至指定的字节数。

```
>>> f = open('file4.txt', mode='w')  # 创建文件对象，只写模式
>>> f.write('你好这是我的abcdefg')
>>> help(f.tell)
>>> f.tell()  # 查看文件的当前位置，注意与 write 方法返回的字符数并不同，tell 方法返回的是字节数
>>> f.flush()  # 将缓存写入磁盘

# 切换到另一个终端
$ cat file4.txt
你好这是我的abcdefg

```

```
# 切换回 Python 解释器
>>> help(f.seek)
>>> f.seek(0)  # 移动当前位置至文件的最起始
>>> f.write('Bye')  # 再次写入，但注意这次是移动过位置之后的写入
>>> f.close()  # 写入磁盘，关闭文件，释放资源

# 切换到另一个终端
$ cat file4.txt  # 这里 '你' 刚好被替换成 'Bye'
Bye好这是我的abcdefg

# 切换回 Python 解释器
>>> '你'.encode('utf-8')  # '你' 编码后占三个字节
>>> 'Bye'.encode('utf-8')  # B y e 三个字符编码后各占一个字节

>>> f = open('file4.txt')  # 创建文件对象，只读模式
>>> f.read(4)  # 读取 4 个字符，而不是字节
>>> help(f.read)  # 查看 f 对象的 read 方法的文档
>>> f.tell()  # 查看文件的当前位置，注意与 read 方法传入的字符数并不同，tell 方法返回的是字节数
>>> f.read()  # 读取剩余全部字符至文件末尾
>>> f.tell()
>>> f.read()  # 注意再次 read 再读不出任何字符，因为当前位置是文件末尾
>>> f.seek(0)  # 移动当前位置至文件的最起始
>>> f.read(3)  # 注意此次 read 出的是 Bye，因为当前位置是文件起始
>>> f.close()
```

18.4 字符编码

字符编码 (character encoding) 用来将文本 (str) 编码 (encode) 成字节串 (bytes)，或者将字节串 (bytes) 解码 (decode) 成文本 (str)。由于磁盘文件 (disk file) 以字节串 (bytes) 形式存储信息，文本 (text) 模式 (mode) 的文件对象 (file object) 在创建时就需要通过形参 (parameter) `encoding` 指定字符编码 (character encoding)。Python 默认的 (default) 字符编码 (character encoding) 是 `utf-8`。

```
>>> # 以文本模式写入，以二进制模式读取
>>> f = open('file5.txt', mode='w', encoding='gbk')  # 只写文本模式
>>> f.write('指定编码之后，二进制对用户就是透明的')
>>> f.close()
>>> quit()

$ cat file5.txt  # 查看 file5.txt 的文本内容，由于终端默认使用 utf-8 解码，所以显示乱码
������□����z��û�����□???

$ python3
>>> f = open('file5.txt', mode='rb')  # 只读二进制模式
>>> content = f.read()
>>> f.close()
>>> type(content)  # 可以看到 content 是 bytes 类型
>>> content
```

```
>>> content.decode('utf-8')  # 解码失败报错，使用了不匹配的 encoding
>>> content.decode('gbk')  # 解码成功
```

```
>>> 以二进制模式写入，以文本模式读取
>>> text = '指定编码之后，二进制对用户就是透明的'
>>> content = text.encode('gbk')
>>> type(content)  # 可以看到 content 是 bytes 类型
>>> content
>>> f = open('file6.txt', mode='wb')  # 只写二进制模式
>>> f.write(content)
>>> f.close()
>>> quit()

$ cat file6.txt  # 查看 file6.txt 的文本内容，由于终端默认使用 utf-8 解码，所以显示乱码
������□����z��û�����□���

$ python3
>>> f = open('file6.txt')  # 只读文本模式
>>> f.read()  # 报告 UnicodeDecodeError 错误，因为编码不正确
>>> f.close()
>>> f = open('file6.txt', encoding='gbk')  # 只读文本模式，指定了正确的编码
>>> f.read()  # 成功读取出正确的文本
>>> f.close()
```

如果在读取 (read) 或写入 (write) 时希望忽略可能出现的编码 (encode) 解码 (decode) 错误，可以给 `open` 函数的 `errors` 形参 (parameter) 传递字符串 `'ignore'` 作为实参 (argument)。

```
>>> f = open('file6.txt', errors='ignore')  # 只读文本模式，忽略解码错误
>>> f.read()  #  不再报告 UnicodeDecodeError 错误，但读取出来乱码
>>> f.close()
```

必须小心的是，读取 (read) 或写入 (write) 时忽略 (ignore) 编解码 (encoding) 错误 *很可能* 导致数据丢失，必须谨慎使用。实际应用中推荐明确地 (explicitly) 指定编码和明确地 (explicitly) 报错。

```
>>> import this
...
Explicit is better than implicit.
...
Errors should never pass silently.
Unless explicitly silenced.
...
```

18.5 换行符

Unix 操作系统 (如 FreeBSD、Mac OS X、Linux 等等) 约定以 `'\n'` 一个字符作为文本换行 (newline)，Windows 操作系统约定以 `'\r\n'` 两个字符作为文本换行 (newline)，Mac OS X 以前的旧 Mac 操作系统约定以 `'\r'` 一个字符作为文本换行 (newline)。这种不一致给跨平台软件带来一些问题，比如在 Unix 操作系统下创建的文本文件若在 Windows 操作系统下打开会失去所有的换行。

Python 应对这一跨平台问题采取了 **通用换行符** (universal newline) 的概念。默认情况下，在读取 (read) 文本时，`'\n'`、`'\r'`、`'\r\n'` 都会被自动转换为 `\n`，在写入 (write) 文本时，`\n` 字符会被转换为当前操作系统默认的换行符。

如果希望 *关闭* 通用换行符 (universal newline) 功能，即在读取 (read) 或写入 (write) 文本时不希望换行符被自动转换，则应在调用 `open` 函数时给形参 (parameter) `newline` 传递空字符串 `''` 作为实参 (argument)。

```
>>> f = open('file7.txt', mode='w', newline='')  # 只写模式，关闭通用换行符功能
>>> f.write('a\r\nb\rc\nd')
>>> f.close()
>>> f = open('file7.txt', mode='r')  # 只读模式，默认开启通用换行符功能
>>> f.read()  # 注意换行符在读取时被自动转换
>>> f.close()
>>> f = open('file7.txt', mode='r', newline='')  # 只读模式，关闭通用换行符功能
>>> f.read()  # 注意换行符现在不被自动转换
>>> f.close()
```

如果给 `open` 函数的形参 (parameter) `newline` 传递 `'\n'`，`'\r'` 或 `'\r\n'` 作为实参 (argument)，则读取一行 (readline) 时会读取到传递的换行符作为一行结束，但不影响整个文本的读取 (read) 结果，写入 (write) 时则会把所有的 `'\n'` 字符都转换为指定的换行符。

```
>>> f = open('file7.txt', mode='r', newline='\r')  # 只读模式，以 \r 为换行符
>>> f.readline()  # 注意以哪个字符换行
>>> f.readline()  # 再次注意以哪个字符换行
>>> f.seek(0)  # 回到文件起始重新准备读取
>>> f.read()  # 注意读取全部文本时，任何换行符都没有被转换
>>> f.close()
>>> f = open('file8.txt', mode='w', newline='\r')  # 只写模式，以 \r 为换行符
>>> f.write('a\r\nb\rc\nd')
>>> f.close()
>>> f = open('file8.txt', newline='')  # 只读模式，关闭通用换行符功能
>>> f.read()  # 注意现在读取出的是刚才实际写入的文本
>>> f.close()
```

19. 路径操作

与文件 (file) 紧密相关的概念是文件 (file) 处在文件系统 (file system) 中的 **路径** (path)。这里只简要介绍一些常用的路径 (path) 操作 (operating) 函数 (function)。

```
>>> import os
```

```
>>> os.getcwd()  # 获取当前工作目录
>>> os.listdir('/')  # 获取指定路径下的目录名和文件名
>>> os.listdir('/usr')
>>> os.chdir('/usr')  # 改变当前工作目录
>>> os.getcwd()
>>> p = os.__file__
>>> p  # 路径字符串
>>> os.path.exists(p)  # 计算路径 p 定位的文件是否存在
>>> os.path.exists('/do/not/exist')  # 这是个不存在的文件路径
>>> os.path.dirname(p)  # 计算路径 p 的 目录名 部分
>>> os.path.basename(p)  # 计算路径 p 的 文件名 部分
>>> os.path.expanduser('~/folder')  # 把 ~ 符号根据环境变量展开成用户目录
>>> os.chdir(os.path.expanduser('~'))
>>> os.path.getsize(p)  # 获取文件大小 (字节数)
>>> os.path.isabs(p)  # 测试路径 p 是否为绝对路径
>>> os.path.isdir(p)  # 测试路径 p 是否为目录
>>> os.path.isdir('/usr/bin')
>>> os.path.isdir('/do/not/exist')
>>> os.path.isfile(p)  # 测试路径 p 是否为文件
>>> os.path.join('/usr', 'local', 'bin/', 'python3.7.2/')  # 连接路径
>>> os.path.join('/usr', 'local', '/bin/', 'python3.7.2/')  # 注意起始的 / 表示根目录
>>> os.path.split(p)  # 在最后一个 / 处拆分路径
>>> os.path.split('/usr/local/bin')
>>> os.path.split('/usr/local/bin/')
>>> os.path.splitext(p)  # 将路径 p 拆分为 文件名 和 扩展名
>>> os.path.splitext('os.py')  # 文件名最后一个 . 后面的字符为其 扩展名
>>> os.path.splitext('os.py.tar.gz')
>>> os.path.splitext('.bash_profile')  # 以 . 开头的文件名或目录名在 Unix 下代表隐藏文件
>>> os.path.abspath('.bash_profile')  # 基于当前工作目录计算 相对路径 的 绝对路径
>>> os.path.abspath('.')  # 一个点 . 在路径中代表 当前目录
>>> os.path.splitext('.')
>>> os.path.abspath('..')  # 两个点 .. 在路径中代表 上级目录
>>> os.path.splitext('..')
>>> os.path.splitext('/usr/local/bin')
>>> help(os.path)  # 浏览文档，查看全部路径操作函数
>>> help(os)  # 浏览文档，查看全部文件系统函数，包括创建目录、文件重命名、修改权限等
```

20. 路径对象

从 Python 3.4 版本起，Python 增加了一个新的标准库 `pathlib`，提供了功能强大的 **路径对象** (path object)，使得以往基于 `os` 和 `os.path` 标准库进行的以函数 (function) 调用 (call) 为主的操作，可以转为更直观的面向对象操作。下面简单举几个例子，更多的请自行用 `help` 查看文档。

```
>>> import pathlib
```

```
>>> p = pathlib.Path('week06')  # 基于 相对路径 创建 路径对象 并使用名称 p 指向该对象
>>> p.exists()  # 判断路径 (目录或文件) 是否存在
>>> p.mkdir()  # 创建目录
>>> sorted(os.listdir('.'))  # 检查 week06 目录确实已被创建
>>> p.absolute()  # 计算 p 的绝对路径
>>> p  # p 被指定的是相对路径
>>> fp = p / 'file1.txt'  # 用 / 运算符可以方便地连接路径，生成新的路径对象
>>> fp  # 查看 f 路径对象及其相对路径
>>> f = fp.open(mode='w', encoding='gbk', newline='\r\n')  # 创建一个只写模式的文件对象
>>> f.write('你好\n欢迎\n')  # 写入字符串
>>> f.close()  # 关闭文件，写入磁盘，释放资源
>>> fp.read_text(encoding='gbk')  # 调用 read_text 方法直接以通用换行符模式读取文本，省去打开关闭步骤
>>> fp.rename('file.del.txt')  # 调用 rename 方法直接将文件重命名
>>> fp  # 注意 fp 的相对路径并没有跟随改变
>>> fp.exists()  # fp 对象指向的文件已不存在
>>> list(p.iterdir())  # p 所指的 week06 目录下并不存在 file.del.txt 文件
>>> import os
>>> os.listdir()  # 注意在当前工作目录下找到了 file.del.txt 文件，这是因为传给 rename 方法的 相对路径
```

21. 自测题

21.1 简答题

请简述 `open` 函数具有哪些形参 (parameter)，各个形参 (parameter) 什么含义，分别代表什么功能。

21.2 简答题

请简述 `open` 函数支持哪些模式 (mode)，并讨论他们之间的区别。

21.3 简答题

请写代码，打开并读取 `os` 模块的源代码，并将其显示在终端上。

21.4 简答题

如何理解文件对象 (file object) 和磁盘文件 (disk file) 之间的关系？我们为什么要手动关闭文件对象?

21.5 简答题

请写代码，以写入 (write) 为例，演示文件缓存 (buffer) 的概念，即演示 “写入文件” 与 “写入磁盘” 的区别。

21.6 简答题

请写代码，以读取 (read) 为例，演示文件位置 (position) 的概念，包括获取当前位置、移动当前位置等操作。

21.7 简答题

请写代码，以编码 (encoding) 的概念为基础，演示读/写文本 (text) 文件与读/写二进制 (binary) 的 *等价* 操作。

21.8 简答题

请写代码，演示通用换行符 (universal newline) 的概念，包括打开和关闭该功能，以及读写之间的差异。先演示写入，再演示读取。

21.9 简答题

请写 Python 代码，获取当前工作目录，再把当前工作目录切换至文件系统根目录，再把当前目录切换至用户根目录。

21.10 简答题

请写 Python 代码，展示 `/etc` 目录下的内容 (目录和文件)，并判断该目录下的 `profile` 名称是目录还是文件。

21.11 简答题

请写 Python 代码，在当前目录下新建一个 `week15` 目录，在 `week015` 目录下新建一个名为 `temp.py` 的文本文件，向其中写入文本 `print('Hello World!')`，然后将其删除。

21.12 简答题

请写 Python 代码，计算 `datetime` 模块的源代码的目录名、文件名和扩展名，并在此基础上获取其所在目录下的其他目录名和文件名。

21.13 简答题

请写 Python 代码，使用路径对象 (path object)，用一条语句创建出目录 `./week06/lecture01/trash`，并在其下创建 `greeting.exe` 文件，在其内部写入字符串 `'无效命令'` 用 `'utf-16'` 编码得出的二进制字节串。
