Lecture 3　Function & OOP & Exceptions & Files

金融科技协会　2018 年 11 月 4 日

目录

一 函数

函数通过**组织代码**来定义**可重用代码**以达到**简化代码**的效果。例如，假设你需要分别对 1 到 10，15 到 25，30 到 49 的自然数求和，如果不使用函数，则代码将会如下所示 (图 1)：

```
1  sum1 = 0
2  for i in range(1,11):
3      sum1 += i
4  print(sum1)
5
6  sum2 = 0
7  for i in range(15,25):
8      sum2 += i
9  print(sum2)
10
11 sum3 = 0
12 for i in range(30,50):
13     sum3 += i
14 print(sum3)
```

```
55
195
790
[Finished in 0.2s]
```

图 1: 代码具有很高的重复度

可以发现，上述代码的三块求和部分，除 range() 返回的序列内容不同外，其他部分均十分相似。因此，我们可以编写一个实现求和的**函数**，以达到简化代码的效果，如下 (图 2)：

```
1  def getSum(i1, i2):
2      result = 0
3      for i in range(i1, i2+1):
4          result += i
5      return result
6
7  print(getSum(1,10))
8  print(getSum(15,20))
9  print(getSum(30,49))
```

```
55
105
790
[Finished in 0.1s]
```

图 2: 使用函数简化代码

上述使用函数的代码，明显比最开始的代码简洁，而实现的功能完全相同。

总之，函数是为实现一个操作而集合在一起的语句集；调用函数时，系统会进入函数，执行函数中的语句，并返回相应的结果。

1.1 定义函数

函数定义的基本模式 (图 3)：

```
def functionName(list of parameters):
    # function body
```

图 3: 函数的定义

即，定义函数包括以下要素：

- 函数头：包括**函数名**与**参数 (形式参数)**
- 函数体；

具体的，下面的名为 **max**的函数，有两个参数 **num1**与 **num2**，函数体将比较这两个参数的大小，并返回其中较大的那个数的值 (图 4)：

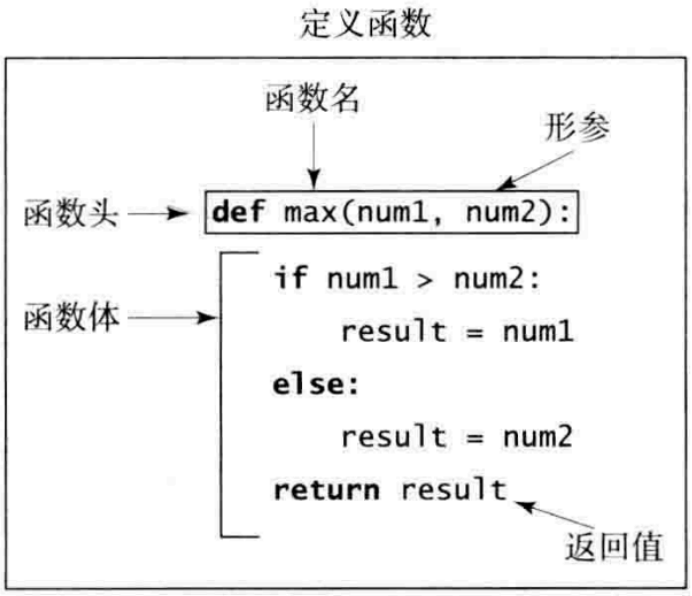


图 4: 定义 max 函数

一些函数具有用 **return**标识的返回值，而另一些函数则没有——他们往往只会被要求完成一定的操作而不返回值，例如 **print()**函数 (图 5)。

```
>>> returnValue = print("Sam is a boy.")
Sam is a boy.
>>> returnValue
>>> returnValue == None
True
>>>
```

图 5: print 函数无返回值 (返回 None)

因此，通过一个函数是否有返回值，我们可以把函数分为**带返回值的函数**与**无返回值函数**。

1.2 调用函数

调用一个函数，即使用函数名来调用一个函数，来执行函数中的代码。

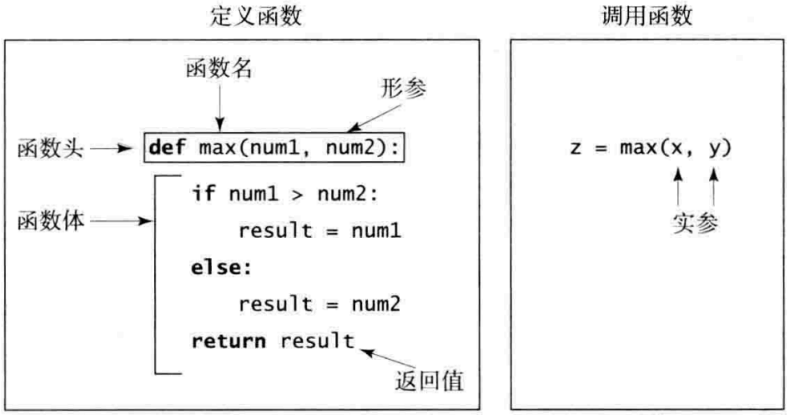


图 6: 调用 max 函数

上图 (图 6) 右侧即为函数调用的形式，将 **x**与 **y**作为**实际参数**传入 **max**函数，**max**函数内的语句被执行，最终返回一个结果，该结果被赋给变量 **z**。

实际参数：简称实参，与形参相对。以 **max**函数为例，函数头中的参数是形参，这里即 **num1**与 **num2**，他们的作用是占位符；实参是调用函数时实际传入函数的参数值，这里即 **x**与 **y**，他们的值 (或者是引用，见 Lecture2 的可变对象与不可变对象一节，可变对象传入引用，不可变对象传入值) 被赋给形参 **num1**与 **num2**，**num1**与 **num2**带着他们的值 (或引用) 进入函数体，进行运算，最终返回一个结果。

具体看一个函数调用时，程序运行顺序的例子 (图 7 & 图 8)：

```
1  # Return the max of two numbers
2  def max(num1, num2):
3      if num1 > num2:
4          result = num1
5      else:
6          result = num2
7
8      return result
9
10 def main():
11     i = 5
12     j = 2
13     k = max(i, j) # Call the max function
14     print("The larger number of", i, "and", j, "is", k)
15
16 main() # Call the main function
```

```
The larger number of 5 and 2 is 5
```

图 7: 运行顺序

对于**带返回值的函数**，我们通常将其调用结果赋给一个变量，或者打印出来，例如：

- $upperStr = "abcde".upper()$
- $print("abcde".upper())$

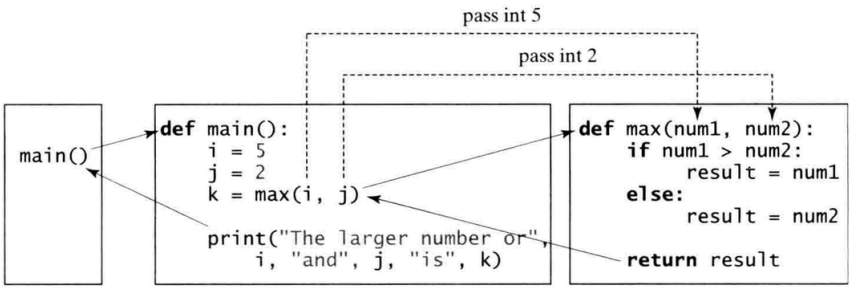


图 6-2 当函数被调用时，程序控制权就转移到这个函数，
当函数结束时程序控制权又转移到函数被调用的地方

图 8: 运行顺序

对于**不带返回值的函数**，我们通常使用一条语句直接调用，例如：

- $print(”Helloworld”)$

1.3 函数参数

函数实参是作为**位置参数**或**关键字参数**被传递的。

1.3.1 位置参数

位置参数要求参数按它们**在函数头的顺序**进行传递。

例如，下面的 **printN**函数，有两个参数 **strA**与 **N**，其作用是将 **strA**打印 **N**次 (注意，由于 python 是动态类型语言，所以不用标注参数的类型)。调用函数 **printN**时，我们可以按照**位置参数**的规则传递实参，即：**实参必须和函数头中定义的形参在顺序、个数、类型上匹配：**

```
1  # 将一个字符串(strA)打印多次(N)
2  def printN(strA, N):
3      for _ in range(N):
4          print(strA)
5
6
7  printN("Hello world", 5)
```

```
Hello world
Hello world
Hello world
Hello world
Hello world
[Finished in 0.3s]
```

图 9: printN 函数

基于位置，上图 (图 9) 将**”Hello world”**传给了形参 **strA**，将 **5**传给了形参 **N**；而下图 (图 10) 则将**”Hello world”**传给了形参 **N**，将 **5**传给了形参 **N**，由此引发错误：

因此，在使用位置参数的传递规则时，**一定要注意参数的先后顺序**。

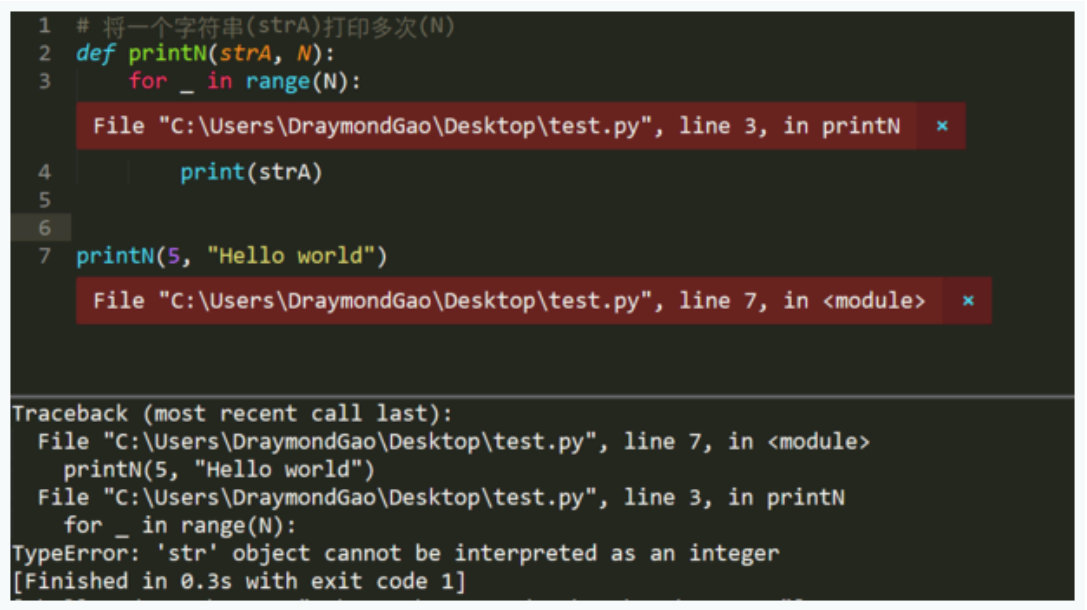


图 10: 位置参数顺序错误

1.3.2 关键字参数

除使用**位置参数**的规则传递参数外，也可以通过**关键字参数**的规则传递参数，即：通过**name=value**的形式传递每一个参数。例如 (图 11)：

图 11: 关键字参数

因为使用了名称标识，所以可以不考虑位置顺序。

另外，位置参数与关键字参数可以混用，但**位置参数不能出现在任何关键字参数之后**，例如函数头如果是 **def f(p1,p2,p3)**，你可以这样调用 **f(30, p2=4, p3=10)**，但不可以 **f(30, p2=4, 10)**。

1.3.3 默认参数

Python 允许定义带默认参数值的函数。设置默认参数后，当函数被调用的时候，相应形参未被传入实参，那么默认值自动被传入 (图 12)：

```
# 求x的N次方的函数
def myPow(x, N=1):
    return x ** N

print(myPow(3))     # 3: 因为没有传入N的值，因此默认为1，即求3的1次方
print(myPow(3,5))   # 243: 求3的5次方
```

```
3
243
[Finished in 0.2s]
```

图 12: 默认参数

1.3.4 变量的作用域

变量的作用域是指该变量可以在程序中**被引用**的范围。

- **局部变量**：在函数内部被定义的变量，只能在函数内部访问；
- **全局变量**：在所有函数之外创建的变量，可以被所有的函数访问；

test.py

```
globalVar = 1

def f1():
    localVar = 2
    print(globalVar)
    print(localVar)

f1()
print(globalVar)
print(localVar)
```

图 13: globalVar 是全局变量，localVar 是局部变量

上图 (图 13) 中，**globalVar**是全局变量，**localVar**是局部变量。**globalVar**可以在函数中访问，但是 **localVar**却不能在函数外访问，会报错 (图 14)：

```
$ python test.py
1
2
1
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    print(localVar)
NameError: name 'localVar' is not defined
```

图 14: localVar 无法在函数外被访问

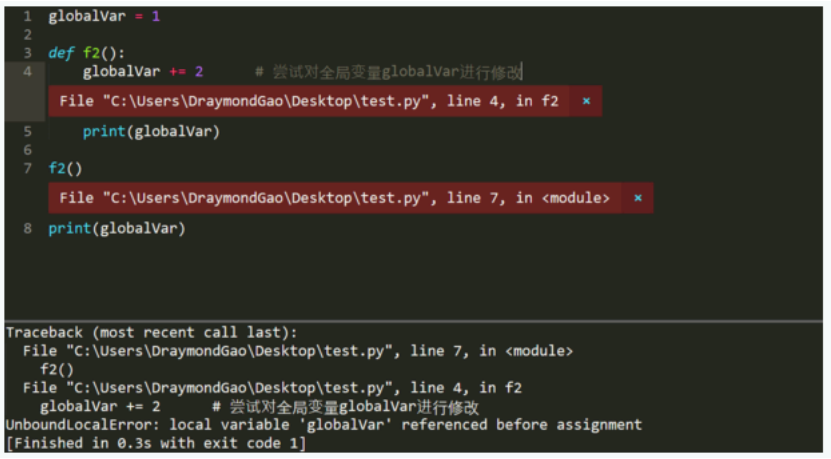


图 15: 无法在函数内部修改全局变量 globalVar

全局变量可以在函数中被访问，但是不能被修改 (图 15)：

如果在函数内部创建一个与全局变量相同名称的变量，则该新建的变量将“统治”该函数，即访问该名称，即访问的是函数内部创建的那个局部变量，如下图 (图 16)，调用 **f2**将不会打印**我在函数外**，而是打印在函数内部创建的同名局部变量的值**我在函数里**：

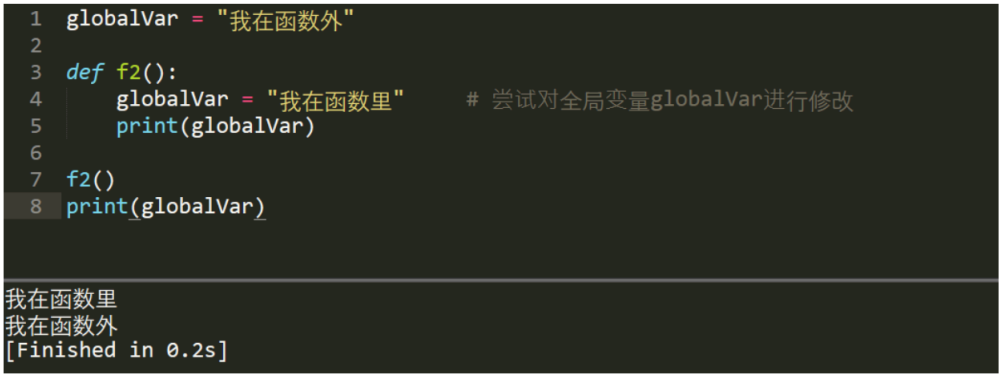


图 16: 覆盖

综上，访问一个变量是有顺序的：**同层级搜索，没有再向上层搜索，不会向下层搜索**。

关于 global

1. 将局部变量的作用域绑定为全局，即在函数中创建的变量可以在函数外使用它 (图 17)。

```
1  def f2():
2      global localVar
3      localVar = "我明明在函数里啊"
4      print(localVar)
5
6  f2()
7  print(localVar)      # 不会报错
```

```
我明明在函数里啊
我明明在函数里啊
[Finished in 0.4s]
```

图 17: 将局部变量的作用域绑定为全局

2. 在函数内修改全局变量 (图 18)。

```
1  globalVar = "我在函数外面"
2
3  def f2():
4      global globalVar
5      globalVar += "！！！"
6      print(globalVar)
7
8  f2()
```

```
我在函数外面！！！
[Finished in 0.2s]
```

图 18: 在函数内修改全局变量

1.3.5 通过传引用来传递参数

调用函数的时候：**形参 = 实参**，实质是一个赋值的过程。

赋值过程的实质其实比较麻烦——**赋值的实质是赋变量或字面量的引用值**，如下 (图 19，**id()**函数用于获取对象的内存地址，即引用值，可以理解为对象的唯一身份标识，id 不同的两个对象是不同的对象)：

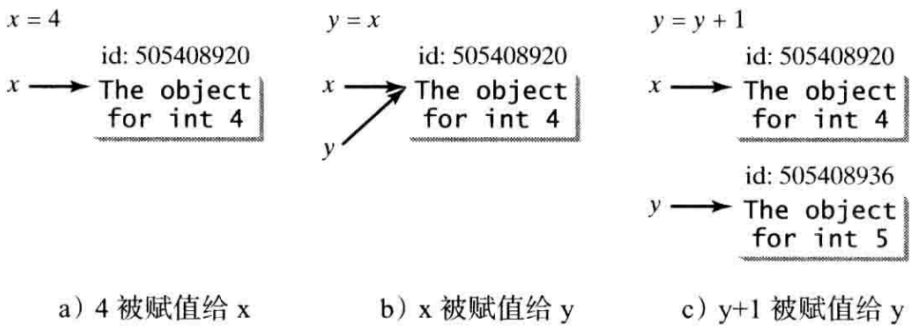


图 19: 赋值的实质是赋引用值

从上图中可以看出：

- Python 中的变量，其储存的实际上不是对象的值，而是对象的引用；
- x 是一个整型变量，其值为 4 ——x 引用了一个 int 对象，这个 int 对象的值是 4；
- 整数是不可变对象。

对于可变对象需要注意!

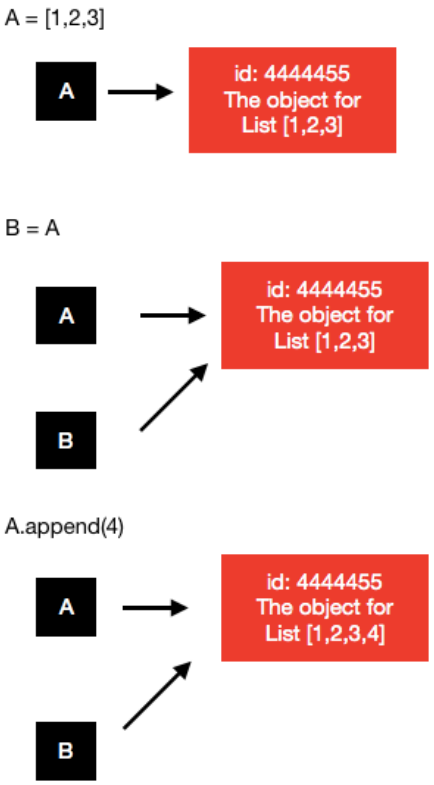


图 20: 可变对象的“就地修改”

```
>>> A = [1,2,3]
>>> B = A
>>> B
[1, 2, 3]
>>> A
[1, 2, 3]
>>> A.append(4)
>>> A
[1, 2, 3, 4]
>>> B
[1, 2, 3, 4]
```

图 21: 修改 A，B 也发生改变，因为 A 与 B 指向同一个对象

总结：调用带参数的函数时，每个实参的引用值被传递给函数的形参。如果实参是一个数字或一个字符串 (不可变对象)，则不管函数中的形参有没有变化，实参不受影响；但如果实参是可变对象，如列表，则函数中对形参的操作使其发生改变，同样会影响到实参。参见 Lecture 2 的可变对象与不可变对象。

1.4 返回值

函数的返回值紧跟在保留字 **return**后面。当函数被执行时，一旦遇到 **return**，将返回函数的执行结果，并跳出函数，执行函数外的其他代码；若没有返回值，函数会按顺序执行到函数体的最后一行。

但实际上，无论你的函数是否包含 **return**，都会返回一个值。如果没有 **return**，默认情况下，该函数将在最后返回 **None**。因此无返回值的函数，也被成为 None 函数。

返回多个值：**return a,b**

1.5 函数对象

Python 中一切皆对象。

```
>>> f = abs(-10)
>>> f
10
>>> f = abs
>>> f
<built-in function abs>
>>> f(-10)
10
>>> f is abs
True
>>>
```

图 22: 函数也是对象

上图 (图 22) 中，**abs(-10)**是**函数调用**，返回函数调用的结果；而 **abs**是函数名，指向函数对象本身；可以把函数名赋给其他变量，也就是把函数本身赋值给了其他变量，这样就可以通过该“其他变量”来调用这个函数，上图用 **f**调用了 **abs**函数。

1.6 lambda 函数

在 python 中，函数可以分为**具名函数**与**匿名函数**，其中**匿名函数**又称为 **lambda 函数 (Anonymous Function)**。

lambda 函数以关键字 **lambda**开头，冒号前为参数，冒号后为返回值；即使没有参数，冒号也不能省略。

例如，下面的 lambda 函数，冒号前的 **x**表示函数的参数，冒号后的 **x******2**表示返回值，即该匿名函数的作用是计算一个数的平方值：

lambda x:x****2**

也可以给 lambda 函数的参数设置默认值：

lambda x=1:x****2**

lambda 函数的优势在于，能够极大简化代码。但当函数的功能比较复杂，需要较多代码进行组织的时候，lambda 函数则不再适用，应使用具名函数。

二 类与对象

2.1 基本概念

类 (Class) 是蓝图 (Blue print)，而对象 (Object) 是基于蓝图的一个个实例 (Instance)。

例如 (图 23)，“狗” (Dog) 是一个抽象的、笼统的概念，因此可以将其看做是一个类 (Class)。而 Sam 家的狗狗、Bill 家的狗狗、Tom 家的狗狗，则是一只只具体的狗狗，我们就可以把它们看做“狗”类的一个个具体的实例 (Instance)，即一个个对象 (Object)。

```python
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print("Bow-wow!!!")

    def sayName(self):
        print("Hi, my name is %s." % self.name)


SamDog = Dog(name = 'Samer')
BillDog = Dog(name = 'Biller')
TomDog = Dog(name = 'Tomer')

SamDog.bark()           # Bow-wow!!!
BillDog.bark()          # Bow-wow!!!
TomDog.bark()           # Bow-wow!!!

SamDog.sayName()        # Hi, my name is Samer.
BillDog.sayName()       # Hi, my name is Biller.
TomDog.sayName()        # Hi, my name is Tomer.
```

```
Bow-wow!!!
Bow-wow!!!
Bow-wow!!!
Hi, my name is Samer.
Hi, my name is Biller.
Hi, my name is Tomer.
[Finished in 0.1s]
```

图 23: Dog 类与基于 Dog 类创建的三个实例

Python 中，所有数据都是对象——一个数字是一个对象，一个字符串是一个对象，一个列表也是一个对象。对象是一个个具体的实例，相同类型的对象源于一个共同的抽象的类。在 python 中，类 (class) 和类型 (type) 是一个概念，使用 type 函数可以查看对象的类型，也就是对象所抽象出的类的名称。

例如，**[1,2,3]**，**['a','b','c']**，**[True, False, True]**是三个对象，是三个具体的实例；而它们都属于列表，这里列表就是类的概念，即上述三个对象都属于“列表类”，都是“列表对象”。

使用 **id()**函数可以查看对象的内存地址，即此对象在计算机内存中的位置；**id()**函数将返回一个整数，代表对象的身份，如同对象在程序中的身份证号码。

```
>>> a = 3
>>> id(a)
4356273568
>>>
>>> A = [1,2,3]
>>> id(A)
4359736776
>>>
>>>
```

图 24: id 函数

2.2 定义类

类与对象关系的另一个比喻：**类像是菜谱，而对象是根据菜谱做出的一道道具体的菜**。

类 (class) 将**描述类的数据**以及**处理该数据的函数**集合在一起：

- 描述类的数据——**属性 (attribute)**，又叫做特征，或者数据域；
- 处理数据的函数——**方法 (method)**，又叫做函数，表示对象可以完成的某个动作。

例如，对于下图 (图 25) 的 **Circle**类，它规定了一个名为 **radius**的属性，与三个方法 (函数)——**getArea**, **getPerimeter**, **setRadius**，分别完成的动作为——返回面积，返回周长，设置半径。

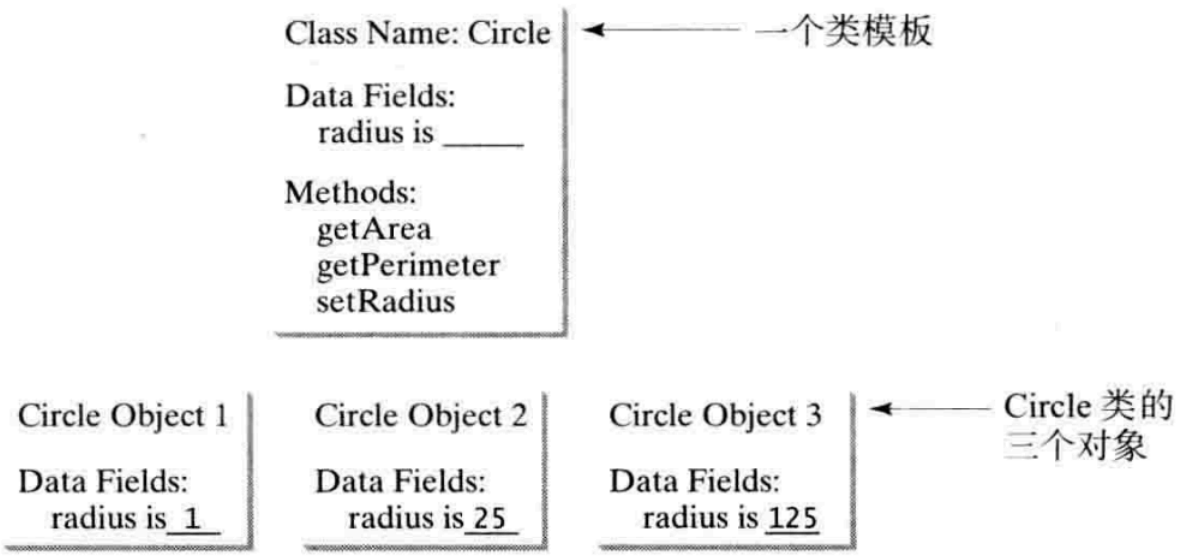


图 25: id 函数

此外，上图 (图 25) 的下半部分表示三个 **Circle**类的对象。因此，一个 Python 类使用变量来存储数据域，定义方法来完成相关动作。可以根据一个类，创建许许多多个实例，即对象，这个根据类创建对象的过程称为**实例化**。

下图 (图 26) 的代码定义了上图 (图 25) 中的 **Circle**类，并分别创建了半径为 1 与半径为 25 的 **Circle**的对象：

```python
import math

class Circle:
    def __init__(self, radius = 1):
        self.radius = radius

    def getPerimeter(self):
        return 2 * self.radius * math.pi

    def getArea(self):
        return self.radius * self.radius * math.pi

    def setRadius(self, radius):
        self.radius = radius


circle1 = Circle(1)
circle2 = Circle(25)
```

图 26: Circle 类与 Circle 对象

总之，定义类的语法模式为 (图 27)：

```python
class ClassName:
    initializer     # 构造方法
    methods         # 其他的方法
```

图 27: 类的定义

2.3 构造函数

定义类的第一项工作往往是完成**构造方法**的编写，即上图 (图 27) 中的 **initializer**部分。

构造方法，又叫做**构造函数**，即 **___init___()**，是比较特殊的方法 (两个下划线)，是**对象实例化时自动被执行的函数**，常用作初始化对象，如**使用初始值创建对象的数据域**。构造方法会完成两个任务：

- 在内存中为类创建一个对象——这个对象还是空白的，数据域还没有任何内容；
- 调用类的 **___init___**方法来初始化对象——用具体的值填充数据域；

注意，包括构造方法在内的所有类方法的第一个参数都是 **self**，表示该对象自身，即调用该方法的对象；但在调用方法时，不需要传入这个参数。

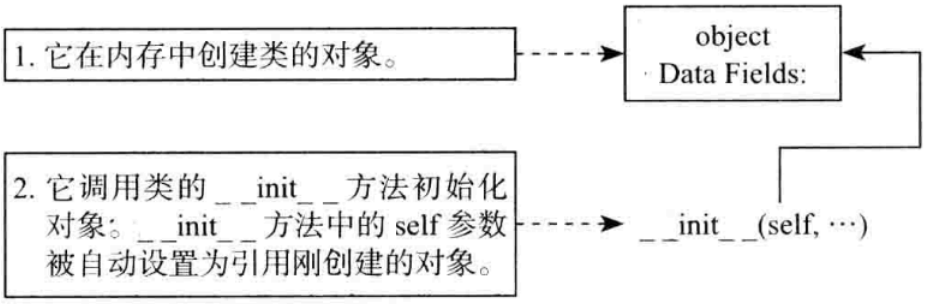


图 28: 构造方法的作用

创建对象的语法规则为：类名 (参数)。在创建对象，即实例化的同时，构造方法被自动调用，首先完成第一个任务，即在内存中创建一个还未初始化的对象；此后，**参数**被传递到**___init___**方法中 (这里忽略 self，即按位置参数规则传递，第一个形参是 self 后第一个参数而不是 self)。

例如，对于上面的 Circle 类 (图 26)，若想创建一个半径 **radius**为 10 的 Circle 对象，那么就 **Circle(10)**。这里的参数 10，按照位置规则传递到 **___init___(self, radius = 1)**中，忽略 self，第一个形参是 **radius**，因此 **radius = 10**，接着进入方法内部，执行 **self.radius = radius**，即把这个 Circle 对象的 radius 属性设置为 10。注意，这里 **self.radius**和 **radius**意义不同，**self.radius**是指**此对象的 radius 属性**，而 radius 是形式参数。

另外，因此该 Circle 类的构造函数设置 **radius**的默认值为 1，因此 **Circle()**将创建默认半径为 1 的 Circle 对象。

2.4 访问对象成员

对象成员指对象数据域和对象的方法。其中数据域又被称为实例变量，方法被称为实例方法。使用圆点运算符.来访问对象的数据域或调用其方法——**对象成员访问运算符**，例如：

```
circle1 = Circle(3)

print(circle1.radius)       # 访问circle1对象的radius数据域
print(circle1.getArea())    # 调用circle1对象的getArea方法
```

图 29: 圆点运算符

再次理解，上文中的 **self.radius**，指“对象自己的 radius 属性”。

2.5 隐藏数据域

在上述代码中，你可以直接访问对象的数据域，如 **circle1.radius**直接获得 **circle1**对象的 **radius**数据。但这样的直接访问有时却是灾难，它使得任何人可以随意篡改数据。例如，在 **circle1**被创建后，我可以使用 **circle1.radius = 10**将 **circle1**的半径改为 10：

为了避免上述不安全的情况发生，需要禁止客户端直接访问数据域——数据隐藏——定义私有数据域——在需要私有的数据域前加两个下划线，例如将 **radius**替换为 **___radius**：

```python
import math

class Circle:
    def __init__(self, radius = 1):
        self.radius = radius

    def getPerimeter(self):
        return 2 * self.radius * math.pi

    def getArea(self):
        return self.radius * self.radius * math.pi

    def setRadius(self, radius):
        self.radius = radius

circle1 = Circle(3)
print(circle1.radius)
circle1.radius = 10
print(circle1.radius)
```

```
3
10
[Finished in 0.4s]
```

图 30: 修改对象的数据域

```python
import math

class Circle:
    def __init__(self, radius = 1):
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def getPerimeter(self):
        return 2 * self.__radius * math.pi

    def getArea(self):
        return self.__radius * self.__radius * math.pi


circle1 = Circle(3)
print(circle1.__radius)
```

```
File "C:\Users\DraymondGao\Desktop\新建文本文档.py", line 18, in <module>
Traceback (most recent call last):
  File "C:\Users\DraymondGao\Desktop\新建文本文档.py", line 18, in <module>
    print(circle1.__radius)
AttributeError: 'Circle' object has no attribute '__radius'
[Finished in 0.3s with exit code 1]
```

图 31: 数据域被保护

同样，可以使用两个下划线开始定义私有方法，不允许对象外部调用，只能在对象内部调用。

2.6 父类、子类、多态 (直觉理解)

父类与**子类**的关系是**继承**——一个子类继承了父类所有的数据域和方法，同时可以有自己独有的数据域和方法。

继承——is a——“某个子类”is a “某个父类”。

例如，object 类是 Python 中所有类的父类，所以可以说 1 是整数类 (int) 的对象，也可以说 1 是 object 类的对象——多态——子类的对象可以传递给需要父类类型的参数。

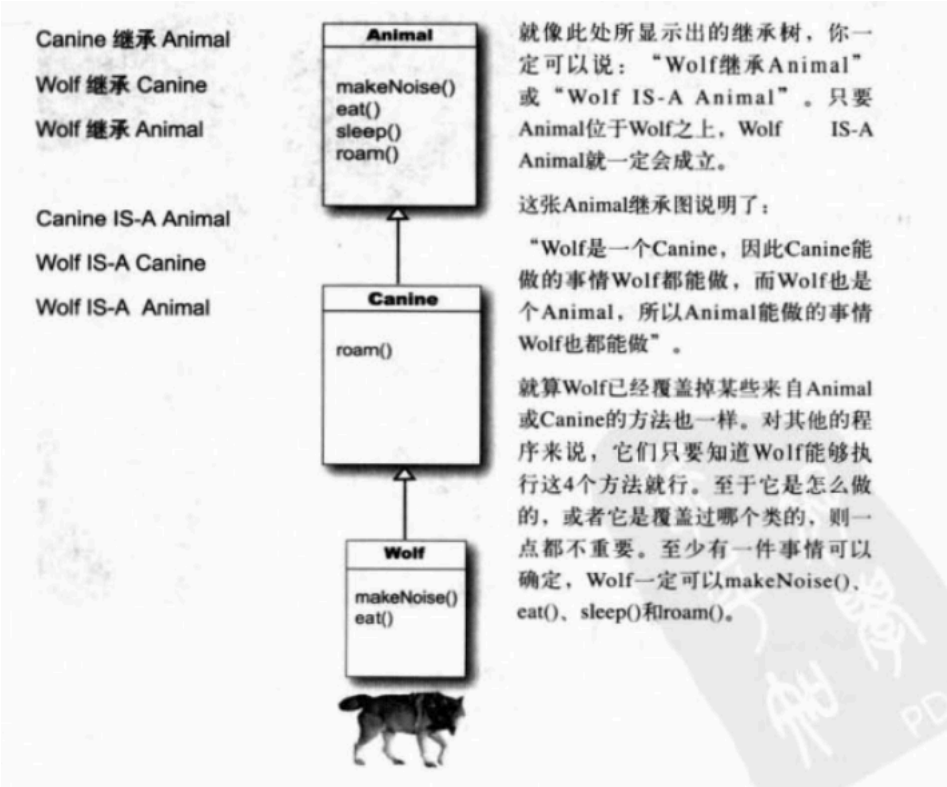


图 32: 直觉理解

三 异常处理

异常 (exception)：程序运行时出现的错误。每个异常都是一个 python 对象。当 python 代码运行时发生错误后，默认情况下，python 会**抛出**相应的异常对象，然后**终止程序**，除非用户手动编写代码捕获处理该异常对象。

```
1  import time
2
3  print("Hello")
4  time.sleep(1)        # 让程序休眠1秒
5  print("Hello")
6  time.sleep(1)        # 让程序休眠1秒
7  print(Hello)         # 忘记引号，会报错
8  print("Hello")
9  print("Hello")
```

```
Hello
Hello
Traceback (most recent call last):
  File "C:\Users\DraymondGao\Desktop\L3.py", line 7, in <module>
    print(Hello)         # 忘记引号，会报错
NameError: name 'Hello' is not defined
```

图 33: 程序在异常抛出后终止

堆栈回溯 (Traceback)：错误信息中包含的内容，追溯到导致这条语句的函数调用来给出导致错误的这条语句的信息，行号。

3.1 常见的异常类型

- **ValueError**：传入无效的参数
- **TypeError**：对类型无效的操作
- **IndentationError**：缩进错误
- **SyntaxError**：Python 语法错误
- **NameError**：未声明/初始化对象 (没有属性)
- **KeyError**：映射中没有这个键
- **IndexError**：序列中没有此索引 (index)
- **ImportError**：导入模块/对象失败
- **ZeroDivisionError**：除零

3.2 异常处理

使用 **try/except**语句捕获并处理异常 (不能只有 try，有 try 必配 except)：

```
try:
    <body>     # 可能出现异常的语句
except <ExceptionType>:
    <handler>
```

图 34: 程序在异常抛出后终止

上述形式为异常处理的最简单形式。将可能出现异常的语句放在 **try**中，如果无异常出现，正常执行完 **try**中的所有语句，忽略 **except**的所有语句；如果出现异常，**<body>**中的剩余代码被跳过，直接转到 **except**中，执行 **<handler>**中的代码。

注意：如果省略 **<ExceptionType>**，则 **except**块能够捕获所有类型的异常；如果设置特定的 **<ExceptionType>**，则只能捕获相对应类型的异常，如 **except IOError**只能捕获 **IO**异常，其他的异常如 **ValueError**不能被捕获。

3.3 多 except 子句

```
try:
    <body>     # 可能出现异常的语句
except <ExceptionType1>:
    <handler1>
...
except <ExceptionTypeN>:
    <handlerN>
except:
    <handlerExcept>
else:
    <process_else>
finally:
    <process_finally>
```

图 35: 多 except 子句

当 **try**中出现异常时，会**按顺序**检查是否匹配该 **try**子句后的 **except**子句中的异常；如果找到匹配的 **except**子句，则该 **except**下的处理器被执行，其他 **except**全部被忽略；如果没有匹配到任何一个捕捉特定异常的 **except**子句，就被最后的那个 **except**子句捕获并执行 **<handlerExcept>**。

注意：python 是按照顺序寻找异常的处理器的，如果父类类型异常的 **except**块在子类的前面，则子类异常的 **except**永远不会被执行，这也是为何上面代码中将能够捕获一切的 **except**块放在最后。

else 子句：可选。如果 **try**块中没有异常抛出，则执行 **else**块中的内容；有异常抛出，被忽略。

finally 块：可选。定义收尾动作，无论有没有异常抛出，都会执行这个块。

3.4 抛出异常

异常是对象，使用 **raise**抛出异常。

```
1  myException = IOError("这是我创建的IO异常")
2  raise myException
   File "C:\Users\DraymondGao\Desktop\新建文本文档.py", line 2, in <module>
3

Traceback (most recent call last):
  File "C:\Users\DraymondGao\Desktop\新建文本文档.py", line 2, in <module>
    raise myException
OSError: 这是我创建的异常
[Finished in 0.4s with exit code 1]
```

图 36: 创建异常对象后使用 raise 抛出异常

3.5 访问异常对象

```
try:
    <body>
except ExceptionType as ex:
    <handler>
```

图 37: 将异常捕获后赋给变量 ex

上述代码中，当 **except**子句捕获到异常时，将该异常对象赋值给变量 **ex**，因此在 **<handler>**中可以用 **ex**来访问该异常。

3.6 Example

下例 (图 38) 要求用户进行两次输入，并打印出两次输入相除的结果。**try**块中可能出现的错误有：**除零错误**——用户的确输入了两个有效的数值，但除数为 0；**未声明错误**——用户给 input 非数值型输入，如”Hello”。

```
1  try:
2      number1 = eval(input("Enter the first number: "))
3      number2 = eval(input("Enter the second number: "))
4      result = number1 / number2
5      print("The answer is %s" % str(result))
6  except ZeroDivisionError as e1:
7      print("成功捕获除零异常: %s" % e1)
8  except Exception as e2:
9      print("警告！该异常并非除零异常: %s" % e2)
10 else:
11     print("没有异常我才会显示")
12 finally:
13     print("不论有没有异常我都会显示")
```

图 38: Example

第一种情况 (图 39)，用户键入两个数值型输入，且除数不为 0，**try**块中无异常抛出，程序正常运行。因此，各 **except**块无“响应”，**else**块中的语句因为没有异常出现所以执行，**finally**块因为无论如何都会执行所以执行，结果如下图 (图 39)：

```
C:\Users\DraymondGao\Desktop\Lecture 3
$ python 异常.py
Enter the first number: 10
Enter the second number: 2
The answer is 5.0
没有异常我才会显示
不论有没有异常我都会显示
```

图 39: 正常，无异常抛出

第二种情况 (图 40)，用户键入两个数值型输入，但除数为 0，**try**块中抛出除零异常 (line 4)，按照顺序，被 **except ZeroDivisionError**先捕获 (虽然 **except Exception**也能捕获，但是其在 **except ZeroDivisionError**的后面，轮不到)。由于 **try**块中出现异常，**else**块中的语句不会执行，**finally**块因为无论如何都会执行所以执行，结果如下图 (图 40)：

```
C:\Users\DraymondGao\Desktop\Lecture 3
$ python 异常.py
Enter the first number: 10
Enter the second number: 0
成功捕获除零异常: division by zero
不论有没有异常我都会显示
```

图 40: 抛出 ZeroDivisionError

第三种情况 (图 41)，用户键入的输入中有非数值型输入，如在第一个 input 时键入”Hello”，**try**块中将抛出 **NameError**。该异常不能被 **except ZeroDivisionError**捕获，但可以被 **except Exception**捕获，因为 **Exception**是所有异常的父类，是 NameError 当然也是 Exception(多态的概念)。另外，由于 **try**块中出现异常，**else**块中的语句不会执行，**finally**块因为无论如何都会执行所以执行，结果如下图 (图 41)：

```
C:\Users\DraymondGao\Desktop\Lecture 3
$ python 异常.py
Enter the first number: 10
Enter the second number: Hello
警告！该异常并非除零异常: name 'Hello' is not defined
不论有没有异常我都会显示
```

图 41: 抛出 NameError

四 文件操作

4.1 绝对路径与相对路径

注意，Win 使用 \ 作为**路径分隔符**，但在 python 中 \ 是转义字符，所以常常出现 \\ ；Mac 使用 / 作为路径分隔符。

绝对路径：文件或文件夹在电脑硬盘中的真正路径，如 Mac 上的：/Users/draymondgao/Desktop/data.txt，Win 上的 $C:\\Users\\DraymondGao\\Desktop\\data.txt$。绝对路径以根目录为起始点，即对 Win 来说以盘符开始，对 Mac 来说以 / 开始。

相对路径：相对当前用户所编写的 py 文件所在文件夹的路径。例如 **open("data.txt")**中，**"data.txt"**就是一个相对路径的表示方法，表示和该 py 文件处于同一文件夹下的名为**data**的 txt 文件；**open("../data.txt")**则表示打开位于该 py 文件上一层文件夹中的一个名为**data.txt**的文件。

4.2 文件对象

文件对象是 python 代码对电脑上外部文件的主要接口。python 可以通过文件对象实现对电脑上文件的**读取或写入**。创建文件对象主要用到内置函数 **open()**，它有两个主要的参数：

- **外部的文件名 (字符串)**：如/Users/draymondgao/Desktop/data.txt
- **处理模式 (字符串)**：如 **w**，这个模式为写入模式，**r**为读取模式

另外，为了防止编码混乱，常常写入另一个关键字参数——**encoding = 'utf8'**，将编码规则确定为 UTF8。

4.3 文件的写入

```
>>> filename = r"C:\Users\DraymondGao\Desktop\data.txt"    # 绝对路径
>>> f = open(filename, 'w', encoding = 'utf8')
>>> f.write("Hello world")
11
>>> f.close()
>>>
>>>
```

图 42: 文件的写入

进行上述操作 (图 42) 后 (注意 **write**后，最后一定要 close 文件对象，否则不成功)，桌面出现 **data.txt**文件，打开 (图 43)：

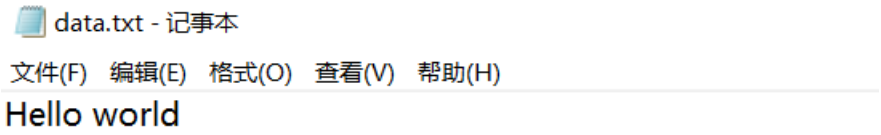


图 43: data.txt

总的看，在指定路径下创建了一个指定名字的文件，并向它写入文本。

需要注意，**write()**只能将 **str**类型的数据写入文件，如果要写入其他类型的数据，必须进行类型转换，如下图 (图 44)：

```
>>> f = open(filename, 'w', encoding = 'utf8')
>>> f.write(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: write() argument must be str, not int
>>>
>>> f.write(str(1))
1
>>>
```

图 44: 不能直接写入非字符串类型

需要将 **int**类型的 1 转化成 **str** 类型的 1，才能写入 **data.txt**中。另外，打开 **data.txt**发现 (图 45)：

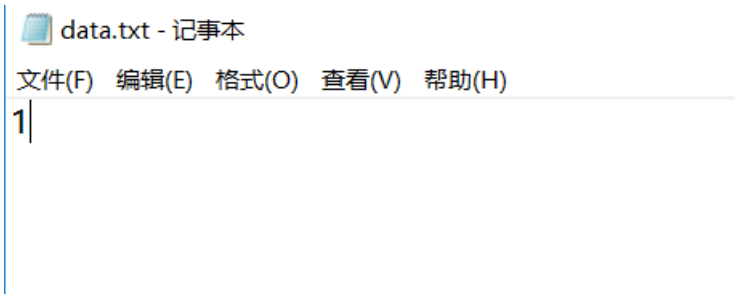


图 45: 覆盖

原有内容 **Hello world**消失，只有刚刚写入的 1——这说明，用 **open**函数以 **w**模式打开一个文件写入，是**完全覆盖**的，而不是在**后面追加**的。

4.4 文件的读取

首先在桌面创建文件 **demo.txt**，内容如下图 (图 46) 所示：

图 46: demo.txt

读出刚才写的内容 (图 47)，则将模式改成 **r**，或者不写，因为默认模式就是 **r**：

```
>>> filename = r"C:\Users\DraymondGao\Desktop\demo.txt"    # 绝对路径
>>> f = open(filename, encoding = 'utf8')
>>> text = f.read()
>>> text
'one\ntwo\nthree\nfour\nfive\n'
>>> f.close()
>>>
```

图 47: 使用 read 函数一次性读取所有内容

read()函数将文件中的内容一次全部读入内存。如果文件很大，超过内存容量，可以选择使用逐行读入——直接对文件对象使用 for 循环，例如下图的逐行读入 (图 48)：

```
>>> filename = r"C:\Users\DraymondGao\Desktop\demo.txt"    # 绝对路径
>>> f = open(filename, encoding = 'utf8')
>>> for line in f:
...     print(line)
...
one

two

three

four

five

>>>
```

图 48: 使用 for 循环遍历文件对象，实现逐行读入

注：上图 (图 48) 中词与词间存在空行 (也就是连续两个换行) 的原因是：第一个换行是文本中原有的换行，第二个换行是 **print**函数的默认结束符，可以使用 **print**函数的 **end**参数来修改：如 **end** = ”表示 print 完后不换行，**end** = ’ ’表示 print 完后光标再向后移动一个空格。

另外还有 **readline()**：一次读一行，光标自动移动到下一行。若到达文件末尾，则返回 False，因此文件还可以这样读 (图 49)：

```
>>> filename = r"C:\Users\DraymondGao\Desktop\demo.txt"    # 绝对路径
>>> f = open(filename, encoding = 'utf8')
>>> line = f.readline()
>>> while line:
...     print(line, end = '')
...     line = f.readline()
...
one
two
three
four
five
>>>
```

图 49: readline()

但更加常用的是 **readlines()**方法。**readlines()**也是**一次性全部读入**，但返回的不是字符串，而是由一行一行内容组成的列表 (图 50)：

```
>>> filename = r"C:\Users\DraymondGao\Desktop\demo.txt"    # 绝对路径
>>> f = open(filename, encoding = 'utf8')
>>> lines = f.readlines()
>>> lines
['one\n', 'two\n', 'three\n', 'four\n', 'five\n']
>>> lines = [line.strip() for line in lines]
>>> lines
['one', 'two', 'three', 'four', 'five']
>>>
```

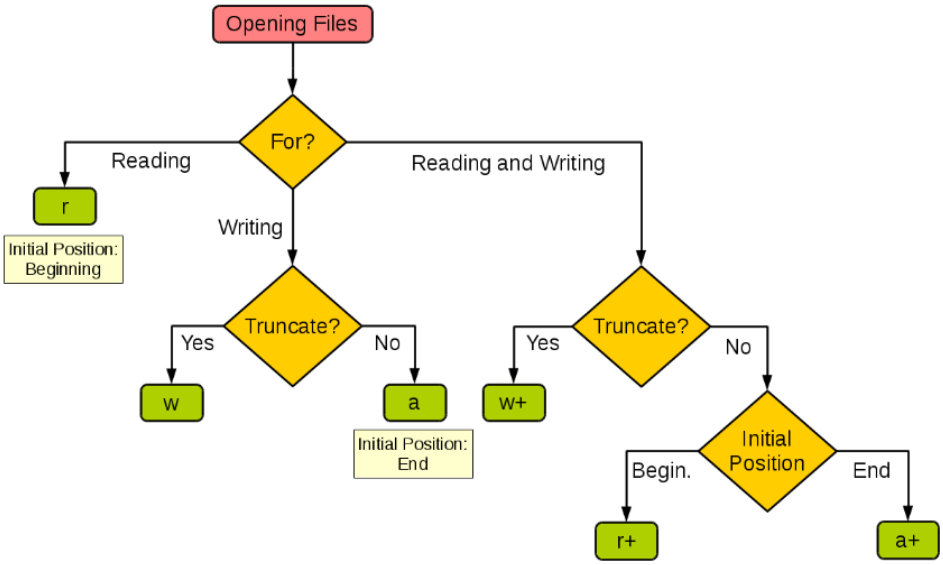
图 50: readlines()

关于 **open**函数的第二个参数——**处理模式**，除了 **w**写入和 **r**读取，还有其他一些，之后还会仔细谈到，先放一张网上的图 (图 51，From：菜鸟教程)：

模式	描述
t	文本模式 (默认)。
x	写模式，新建一个文件，如果该文件已存在则会报错。
b	二进制模式。
+	打开一个文件进行更新(可读可写)。
U	通用换行模式（不推荐）。
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。一般用于非文本文件如图片等。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。一般用于非文本文件如图片等。
w	打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等。
w+	打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

图 51: 各种处理模式

处理模式的判断方式 (图 52，From：菜鸟教程)：



模式	r	r+	w	w+	a	a+
读	+	+		+		+
写		+	+	+	+	+
创建			+	+	+	+
覆盖			+	+		
指针在开始	+	+	+	+		
指针在结尾					+	+

图 52: 常用处理模式

By Draymond