Python3 模块 Python3 错误和异常

Python3 输入和输出

在前面几个章节中，我们其实已经接触了 Python 的输入输出的功能。本章节我们将具体介绍 Python 的输入输出。

输出格式美化

Python两种输出值的方式: 表达式语句和 print() 函数。

第三种方式是使用文件对象的 write() 方法，标准输出文件可以用 sys.stdout 引用。

如果你希望输出的形式更加多样，可以使用 str.format() 函数来格式化输出值。

如果你希望将输出的值转成字符串，可以使用 repr() 或 str() 函数来实现。

- **str()：** 函数返回一个用户易读的表达形式。
- **repr()：** 产生一个解释器易读的表达形式。

例如

```
>>> s = 'Hello, Runoob'
>>> str(s)
'Hello, Runoob'
>>> repr(s)
"'Hello, Runoob'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'x 的值为:  ' + repr(x) + ',  y 的值为: ' + repr(y) + '...'
>>> print(s)
x 的值为:  32.5,  y 的值为: 40000...
>>> #  repr() 函数可以转义字符串中的特殊字符
... hello = 'hello, runoob\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, runoob\n'
>>> # repr() 的参数可以是 Python 的任何对象
... repr((x, y, ('Google', 'Runoob')))
"(32.5, 40000, ('Google', 'Runoob'))"
```

这里有两种方式输出一个平方与立方的表:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # 注意前一行 'end' 的使用
...     print(repr(x*x*x).rjust(4))
```

```
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

注意：在第一个例子中, 每列间的空格由 print() 添加。

这个例子展示了字符串对象的 rjust() 方法, 它可以将字符串靠右, 并在左边填充空格。

还有类似的方法, 如 ljust() 和 center()。 这些方法并不会写任何东西, 它们仅仅返回新的字符串。

另一个方法 zfill(), 它会在数字的左边填充 0，如下所示：

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

str.format() 的基本使用如下:

```
>>> print('{}网址： "{}!"'.format('菜鸟教程', 'www.runoob.com'))
菜鸟教程网址： "www.runoob.com!"
```

括号及其里面的字符 (称作格式化字段) 将会被 format() 中的参数替换。

在括号中的数字用于指向传入对象在 format() 中的位置，如下所示：

```
>>> print('{0} 和 {1}'.format('Google', 'Runoob'))
Google 和 Runoob
>>> print('{1} 和 {0}'.format('Google', 'Runoob'))
Runoob 和 Google
```

如果在 format() 中使用了关键字参数, 那么它们的值会指向使用该名字的参数。

```
>>> print('{name}网址： {site}'.format(name='菜鸟教程', site='www.runoob.com'))
菜鸟教程网址： www.runoob.com
```

位置及关键字参数可以任意的结合:

```
>>> print('站点列表 {0}, {1}, 和 {other}。'.format('Google', 'Runoob',
                                                  other='Taobao'))
站点列表 Google, Runoob, 和 Taobao。
```

'!a' (使用 ascii()), '!s' (使用 str()) 和 '!r' (使用 repr()) 可以用于在格式化某个值之前对其进行转化:

```
>>> import math
>>> print('常量 PI 的值近似为： {}。'.format(math.pi))
常量 PI 的值近似为： 3.141592653589793。
>>> print('常量 PI 的值近似为： {!r}。'.format(math.pi))
常量 PI 的值近似为： 3.141592653589793。
```

可选项 ':' 和格式标识符可以跟着字段名。 这就允许对值进行更好的格式化。 下面的例子将 Pi 保留到小数点后三位：

```
>>> import math
>>> print('常量 PI 的值近似为 {0:.3f}。'.format(math.pi))
常量 PI 的值近似为 3.142。
```

在 ':' 后传入一个整数, 可以保证该域至少有这么多的宽度。 用于美化表格时很有用。

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}
>>> for name, number in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, number))
...
Runoob     ==>          2
Taobao     ==>          3
Google     ==>          1
```

如果你有一个很长的格式化字符串, 而你不想将它们分开, 那么在格式化时通过变量名而非位置会是很好的事情。

最简单的就是传入一个字典, 然后使用方括号 '[]' 来访问键值 :

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}
>>> print('Runoob: {0[Runoob]:d}; Google: {0[Google]:d}; Taobao: {0[Taobao]:d}'.format(table))
Runoob: 2; Google: 1; Taobao: 3
```

也可以通过在 table 变量前使用 '**' 来实现相同的功能：

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}
>>> print('Runoob: {Runoob:d}; Google: {Google:d}; Taobao: {Taobao:d}'.format(**table))
Runoob: 2; Google: 1; Taobao: 3
```

旧式字符串格式化

% 操作符也可以实现字符串格式化。 它将左边的参数作为类似 sprintf() 式的格式化字符串, 而将右边的代入, 然后返回格式化后的字符串. 例如:

```
>>> import math
>>> print('常量 PI 的值近似为：%5.3f。' % math.pi)
常量 PI 的值近似为：3.142。
```

因为 str.format() 比较新的函数， 大多数的 Python 代码仍然使用 % 操作符。但是因为这种旧式的格式化最终会从该语言中移除, 应该更多的使用 str.format().

读取键盘输入

Python提供了 input() 内置函数从标准输入读入一行文本，默认的标准输入是键盘。

input 可以接收一个Python表达式作为输入，并将运算结果返回。

```
#!/usr/bin/python3

str = input("请输入：");
print ("你输入的内容是：", str)
```

这会产生如下的对应着输入的结果：

```
请输入：菜鸟教程
你输入的内容是：  菜鸟教程
```

读和写文件

open() 将会返回一个 file 对象，基本语法格式如下:

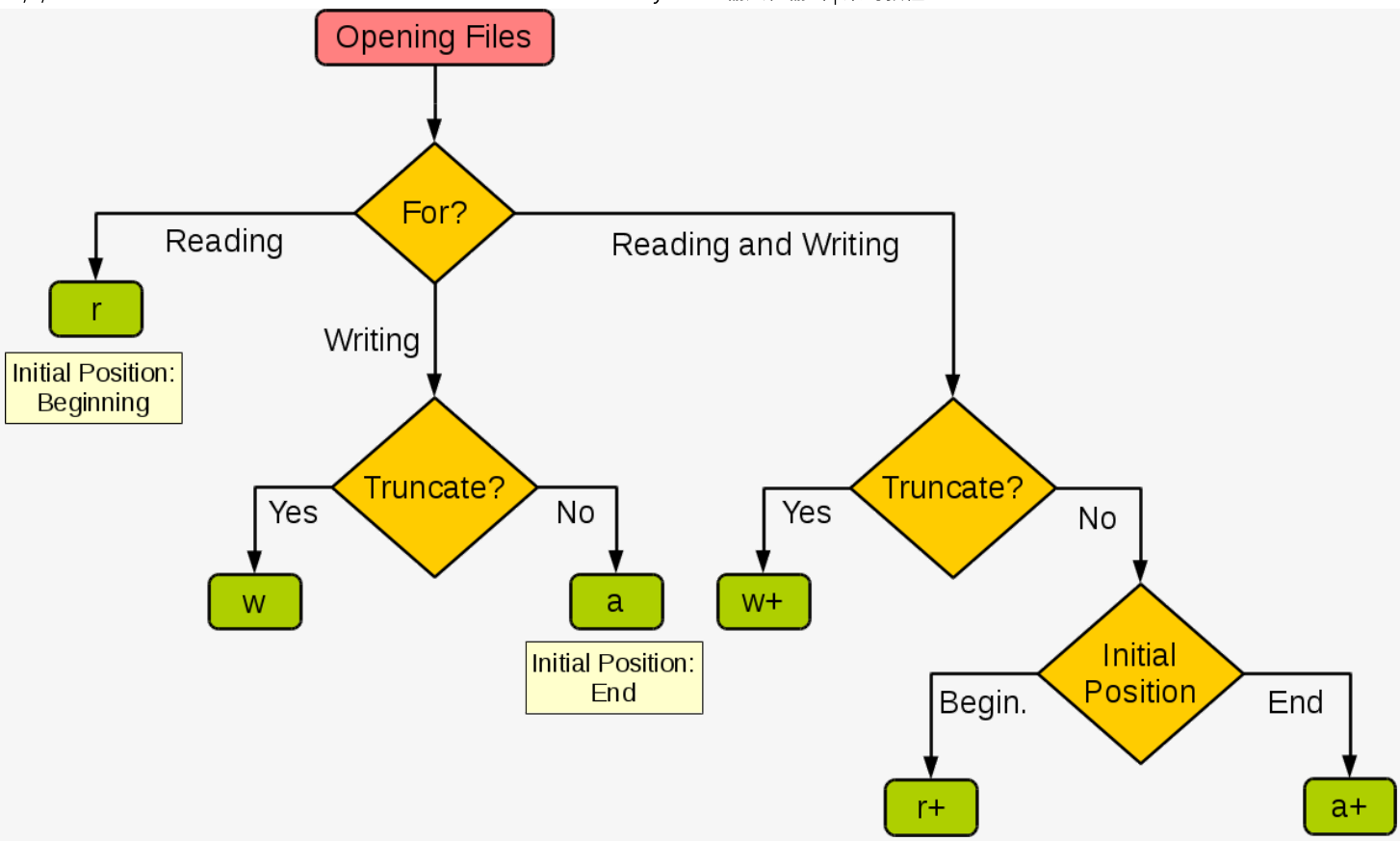
```
open(filename, mode)
```

- filename：包含了你要访问的文件名称的字符串值。
- mode：决定了打开文件的模式：只读，写入，追加等。所有可取值见如下的完全列表。这个参数是非强制的，默认文件访问模式为只读(r)。

不同模式打开文件的完全列表：

模式	描述
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。
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。
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。
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。
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。

下图很好的总结了这几种模式：



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

以下实例将字符串写入到文件 foo.txt 中：

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "w")

f.write( "Python 是一个非常好的语言。\n是的，的确非常好!!\n" )

# 关闭打开的文件
f.close()
```

- 第一个参数为要打开的文件名。
- 第二个参数描述文件如何使用的字符。 mode 可以是 'r' 如果文件只读, 'w' 只用于写 (如果存在同名文件则将被删除), 和 'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾. 'r+' 同时用于读写。 mode 参数是可选的; 'r' 将是默认值。

此时打开文件 foo.txt,显示如下：

```
$ cat /tmp/foo.txt
Python 是一个非常好的语言。
是的，的确非常好!!
```

文件对象的方法

本节中剩下的例子假设已经创建了一个称为 f 的文件对象。

f.read()

为了读取一个文件的内容，调用 f.read(size), 这将读取一定数目的数据, 然后作为字符串或字节对象返回。

size 是一个可选的数字类型的参数。 当 size 被忽略了或者为负, 那么该文件的所有内容都将被读取并且返回。

以下实例假定文件 foo.txt 已存在（上面实例中已创建）：

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.read()
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
Python 是一个非常好的语言。
是的，的确非常好!!
```

f.readline()

f.readline() 会从文件中读取单独的一行。换行符为 '\n'。f.readline() 如果返回一个空字符串, 说明已经已经读取到最后一行。

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.readline()
```

```
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
Python 是一个非常好的语言。
```

f.readlines()

f.readlines() 将返回该文件中包含的所有行。

如果设置可选参数 sizehint, 则读取指定长度的字节, 并且将这些字节按行分割。

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.readlines()
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
['Python 是一个非常好的语言。\n', '是的，的确非常好!!\n']
```

另一种方式是迭代一个文件对象然后读取每行:

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

for line in f:
    print(line, end='')

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
Python 是一个非常好的语言。
是的，的确非常好!!
```

这个方法很简单, 但是并没有提供一个很好的控制。 因为两者的处理机制不同, 最好不要混用。

f.write()

f.write(string) 将 string 写入到文件中, 然后返回写入的字符数。

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "w")

num = f.write( "Python 是一个非常好的语言。\n是的，的确非常好!!\n" )
print(num)
# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
29
```

如果要写入一些不是字符串的东西, 那么将需要先进行转换:

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo1.txt", "w")

value = ('www.runoob.com', 14)
s = str(value)
f.write(s)

# 关闭打开的文件
f.close()
```

执行以上程序，打开 foo1.txt 文件：

```
$ cat /tmp/foo1.txt
('www.runoob.com', 14)
```

f.tell()

f.tell() 返回文件对象当前所处的位置, 它是从文件开头开始算起的字节数。

f.seek()

如果要改变文件当前的位置, 可以使用 f.seek(offset, from_what) 函数。

from_what 的值, 如果是 0 表示开头, 如果是 1 表示当前位置, 2 表示文件的结尾，例如：

- seek(x,0) ： 从起始位置即文件首行首字符开始移动 x 个字符
- seek(x,1) ： 表示从当前位置往后移动x个字符
- seek(-x,2)：表示从文件的结尾往前移动x个字符

from_what 值为默认为0，即文件开头。下面给出一个完整的例子：

```
>>> f = open('/tmp/foo.txt', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)     # 移动到文件的第六个字节
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # 移动到文件的倒数第三字节
13
>>> f.read(1)
b'd'
```

f.close()

在文本文件中 (那些打开文件的模式下没有 b 的), 只会相对于文件起始位置进行定位。

当你处理完一个文件后, 调用 f.close() 来关闭文件并释放系统的资源，如果尝试再调用该文件，则会抛出异常。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

当处理一个文件对象时, 使用 with 关键字是非常好的方式。在结束后, 它会帮你正确的关闭文件。 而且写起来也比 try - finally 语句块要简短:

```
>>> with open('/tmp/foo.txt', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

文件对象还有其他方法, 如 isatty() 和 trucate(), 但这些通常比较少用。

---

pickle 模块

python的pickle模块实现了基本的数据序列和反序列化。

通过pickle模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。

通过pickle模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

基本接口：

```
pickle.dump(obj, file, [,protocol])
```

有了 pickle 这个对象, 就能对 file 以读取的形式打开:

```
x = pickle.load(file)
```

注解：从 file 中读取一个字符串，并将它重构为原来的python对象。

file: 类文件对象，有read()和readline()接口。

实例1：

```
#!/usr/bin/python3
import pickle

# 使用pickle模块将数据对象保存到文件
data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

实例2：

```
#!/usr/bin/python3
import pprint, pickle

#使用pickle模块从文件中重构python对象
pkl_file = open('data.pkl', 'rb')
```

```
data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

← Python3 模块 Python3 错误和异常 →