测试驱动的开发

写一个函数，或者写一个程序，换一种说法，其实就是“实现一个算法”—— 而所谓的“算法”，Wikipedia 上的定义是这样的：

> In mathematics and computer science, an **algorithm** is an unambiguous specification of how to solve a class of problems. Algorithms can perform calculation, data processing, and automated reasoning tasks.

“算法”，其实没多神秘，就是“解决问题的步骤”而已。

在第二部分的第一章里，我们看过一个判断是否为闰年的函数：

> 让我们写个判断闰年年份的函数，取名为 is_leap()，它接收一个年份为参数，若是闰年，则返回 True，否则返回 False。
>
> 根据闰年的定义：
>
> > - 年份应该是 4 的倍数；
> > - 年份能被 100 整除但不能被 400 整除的，不是闰年。
> > - 所以，相当于要在能被 4 整除的年份中，排除那些能被 100 整除却不能被 400 整除的年份。

不要往回翻！现在自己动手尝试着写出这个函数？你会发现其实并不容易的……

```
def is_leap(year):
    pass
```

第一步，跟很多人想象得不一样，第一步不是上来就开始写……

第一步是先假定这个函数写完了，我们需要验证它返回的结果对不对……

这种“通过先想办法验证结果而后从结果倒推”的开发方式，是一种很有效的方法论，叫做“Test Driven Development”，以测试为驱动的开发。

如果我写的 `is_leap(year)` 是正确的，那么：

- `is_leap(4)` 的返回值应该是 `True`
- `is_leap(200)` 的返回值应该是 `False`
- `is_leap(220)` 的返回值应该是 `True`
- `is_leap(400)` 的返回值应该是 `True`

能够罗列出以上四种情况，其实只不过是根据算法“考虑全面”之后的结果 —— 但你自己试试就知道了，无论多简单的事，想要“考虑全面”好像并不容易……

所以，在写 `def is_leap(year)` 中的内容之前，我只是用 `pass` 先把位置占上，而后在后面添加了四个用来测试结果的语句 —— 它们的值，现在当然都是 `False`…… 等我把整个函数写完了，写正确了，那么它们的值就都应该变成 `True`。

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

def is_leap(year):
    pass

is_leap(4) is True
is_leap(200) is False
is_leap(220) is True
is_leap(400) is True
```

```
False

False

False

False
```

考虑到更多的年份不是闰年，所以，排除顺序大抵上应该是这样：

- 先假定都不是闰年；

> - 再看看是否能被 `4` 整除；
> - 再剔除那些能被 `100` 整除但不能被 `400` 整除的年份……

于是，先实现第一句：“先假定都不是闰年”：

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

def is_leap(year):
    r = False
    return r

is_leap(4) is True
is_leap(200) is False
is_leap(220) is True
is_leap(400) is True
```

```
False
True
False
False
```

然后再实现这部分：“年份应该是 4 的倍数”：

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

def is_leap(year):
    r = False
    if year % 4 == 0:
        r = True
    return r

is_leap(4) is True
is_leap(200) is False
is_leap(220) is True
is_leap(400) is True
```

```
True
False
True
True
```

现在剩下最后一条了：“剔除那些能被 `100` 整除但不能被 `400` 整除的年份”…… 拿一个参数值，比如，`200` 为例：

> - 因为它能被 `4` 整除，所以，使 `r = True`，

- 然后再看它是否能被 `100` 整除 —— 能 —— 既然如此再看它能不能被 `400` 整除，
  - 如果不能，那就让 `r = False`；
  - 如果能，就保留 `r` 的值..... 如此这般，`200` 肯定使得 `r = False`。

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

def is_leap(year):
    r = False
    if year % 4 == 0:
        r = True
        if year % 100 == 0:
            if year % 400 !=0:
                r = False
    return r

is_leap(4) is True
is_leap(200) is False
is_leap(220) is True
is_leap(400) is True
```

```
True
True
True
True
```

尽管整个过程读起来很直观，但真的要自己从头到尾操作，就可能四处出错，不信你就试试 —— 这一页最下面添加一个单元格，自己动手从头写到尾试试.....

当然，Python 内建库中的 `datetime.py` 模块里的代码更简洁，之前给你看过：

```
# cpython/Lib/datetime.py
def _is_leap(year):
    return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
_is_leap(300)
```

```
False
```

你自己动手，从写测试开始，逐步把它实现出来试试？—— 肯定不能允许你拷贝粘贴，哈哈。

在 Python 语言中，有专门用来“试错”的流程控制 —— 今天的绝大多数编程语言都有这种“试错语句”。

当一个程序开始执行的时候，有两种错误可能会导致程序执行失败：

- 语法错误（Syntax Errors）
- 意外（Exceptions）

比如，在 Python3 中，你写 `print i`，而没有写 `print(i)`，那么你犯的是语法错误，于是，解析器会直接提醒你，你在第几行犯了什么样的语法错误。语法错误存在的时候，程序无法启动执行。

但是，有时会出现这种情况：语法上完全正确，但出现了**意外**。这种错误，都是程序已经执行之后才发生的（Runtime Errors）—— 因为只要没有语法错误，程序就可以启动。比如，你写的是 `print(11/0)`：

```
print(11/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-2-5544d98276be> in <module>
----> 1 print(11/0)

ZeroDivisionError: division by zero
```

虽然这个语句本身没有语法错误，但这个表达式是不能被处理的。于是，它触发了 `ZeroDivisionError`，这个“意外”使得程序不可能继续执行下去。

在 Python 中，定义了大量的常见“意外”，并且按层级分类：

> 在第三部分阅读完毕之后，可以回来重新查看以下官方文档：
> https://docs.python.org/3/library/exceptions.html

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
```

```
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

拿 `FileNotFoundError` 为例 —— 当我们想要打开一个文件之前，其实应该有个办法提前验证一下那个文件是否存在。如果那个文件并不存在，就会引发“意外”。

```
f = open('test_file.txt', 'r')
```

```
---------------------------------------------------------------------------
----

FileNotFoundError                         Traceback (most recent call
last)

<ipython-input-3-5fac19176fe6> in <module>
----> 1 f = open('test_file.txt', 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'test_file.txt'
```

在 Python 中，我们可以用 `try` 语句块去执行那些可能出现“意外”的语句，`try` 也可以配合 `except`、`else`、`finally` 使用。从另外一个角度看，`try` 语句块也是一种特殊的流程控制，专注于“当意外发生时应该怎么办？”

```
try:
    f = open('test_file.txt', 'r')
except FileNotFoundError as fnf_error:
    print(fnf_error)
```

```
[Errno 2] No such file or directory: 'test_file.txt'
```

如此这般的结果是：

> 当程序中的语句 `f = open('test_file.txt', 'r')` 因为 `test_file.txt` 不存在而引发意外之时，`except` 语句块会接管流程；而后，又因为在 `except` 语句块中我们指定了 `FileNotFoundError`，所以，若是 `FileNotFoundError` 真的发生了，那么，`except` 语句块中的代码，即，`print(fnf_error)` 会被执行……

你可以用的试错流程还有以下变种：

```
try:
    do_something()
except built_in_error as name_of_error:
    do_something()
else:
    do_something()
```

或者：

```
try:
    do_something()
except built_in_error as name_of_error:
    do_something()
```

```
else:
    do_something()
finally:
    do_something()
```

甚至可以嵌套：

```
try:
    do_something()
except built_in_error as name_of_error:
    do_something()
else:
    try:
        do_something()
    except built_in_error as name_of_error:
        do_something()
...
```

更多关于错误处理的内容，请在阅读完第三部分中与 Class 相关的内容之后，再去详细阅读以下官方文档：

> - Errors and Exceptions
> - Built-in Exceptions
> - Handling Exceptions

理论上，这一章不应该套上这么大的标题：《测试驱动开发》，因为在实际开发过程中，所谓测试驱动开发要使用更为强大更为复杂的模块、框架和工具，比如，起码使用 Python 内建库中的 unittest 模块。

在写程序的过程中，为别人（和将来的自己）写注释、写 Docstring；在写程序的过程中，为了保障程序的结果全面正确而写测试；或者干脆在最初写的时候就考虑到各种意外所以使用试错语句块 —— 这些明明是天经地义的事情，却是绝大多数人不做的……因为感觉有点麻烦。

这里是“聪明反被聪明误”的最好示例长期堆积的地方。很多人真的是因为自己很聪明，所以才觉得“没必要麻烦”—— 这就好像当年苏格拉底仗着自己记忆力无比强大甚至干脆过目不忘于是鄙视一切记笔记的人一样。

但是，随着时间的推移，随着工程量的放大，到最后，那些“聪明人”都被自己坑死了 —— 聪明本身搞不定工程，能搞定工程的是智慧。苏格拉底自己并没完成任何工程，是他的学生柏拉图不顾他的嘲笑用纸笔记录了一切；而后柏拉图的学生亚里士多德才有机会受到苏格拉底的启发，写了《前分析篇》，提出对人类影响至今的“三段论”……

千万不要因为这第二部分中所举的例子太容易而把自己迷惑了。刻意选择简单的例子放在这里，是为了让读者更容易集中精力去理解关于自己动手写函数的方方面面 —— 可将来你自己真的动手去做，哪怕真的去阅读真实的工程代码，你就会发现，难度还是很高的。现在的轻敌，会造成以后的溃败。

现在还不是时候，等你把整本书都完成之后，记得回来再看这个链接：

> - doctest —— Test interactive Python examples
> - unittest —— Unit testing framework