保存到文件的函数

写好的函数，当然最好保存起来，以便将来随时调用。

模块

我们可以将以下内容保存到一个名为 mycode.py 的文件中 —— 这样可以被外部调用的 .py 文件，有个专门的称呼，**模块**（Module）—— 于是，其实任何一个 .py 文件都可以被称为*模块*：

In [3]:

```
# %load mycode.py
# 当前这个 Code Cell 中的代码，保存在当前文件夹中的 mycode.py 文件中
# 以下的代码，是使用 Jupyter 命令 %load mycode.py 导入到当前 Code Cell 中的：

def is_prime(n):
    """
    Return a boolean value based upon
    whether the argument n is a prime number.
    """
    if n < 2:
        return False
    if n == 2:
        return True
    for m in range(2, int(n**0.5)+1):
        if (n % m) == 0:
            return False
    else:
        return True

def say_hi(*names, greeting='Hello', capitalized=False):
    """
    Print a string, with a greeting to everyone.
    :param *names: tuple of names to be greeted.
    :param greeting: 'Hello' as default.
    :param capitalized: Whether name should be converted to capitalzed before print.
    :returns: None
    """
    for name in names:
        if capitalized:
            name = name.capitalize()
        print(f'{greeting}, {name}!')
```

而后，我们就可以在其它地方这样使用（以上代码现在已经保存在当前工作目录中的 mycode.py ）：

In [2]:

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

import mycode

help(mycode.is_prime)
help(mycode.say_hi)

mycode.__name__
mycode.is_prime(3)
mycode.say_hi('mike', 'zoe')
```

```
Help on function is_prime in module mycode:

is_prime(n)
    Return a boolean value based upon
    whether the argument n is a prime number.

Help on function say_hi in module mycode:

say_hi(*names, greeting='Hello', capitalized=False)
    Print a string, with a greeting to everyone.
    :param *names: tuple of names to be greeted.
    :param greeting: 'Hello' as default.
    :param capitalized: Whether name should be converted to capitalzed
before print. False as default.
    :returns: None

Hello, mike!
Hello, zoe!
```

以上这个**模块**（Module (https://docs.python.org/3/tutorial/modules.html)）的名称，就是 mycode 。

模块文件系统目录检索顺序

当你向 Python 说 import ... 的时候，它要去寻找你所指定的文件，那个文件应该是 import 语句后面引用的名称，再加上 .py 构成的名字的文件。Python 会按照以下顺序去寻找：

> - 先去看看内建模块里有没有你所指定的名称；
> - 如果没有，那么就按照 sys.path 所返回的目录列表顺序去找。

你可以通过以下代码查看你自己当前机器的 sys.path ：

In [1]:

```
import sys
sys.path
```

在 sys.path 所返回的目录列表中，你当前的工作目录排在第一位。

有时，你需要指定检索目录，因为你知道你要用的模块文件在什么位置，那么可以用 sys.path.append() 添加一个搜索位置：

```
import sys
sys.path.append("/My/Path/To/Module/Directory")
import my_module
```

系统内建的模块

你可以用以下代码获取系统内建模块的列表：

In [22]:

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

sys.builtin_module_names
"_sre" in sys.builtin_module_names # True
"math" in sys.builtin_module_names # True
```

Out[22]:

```
('_abc',
 '_ast',
 '_codecs',
 '_collections',
 '_functools',
 '_imp',
 '_io',
 '_locale',
 '_operator',
 '_signal',
 '_sre',
 '_stat',
 '_string',
 '_symtable',
 '_thread',
```

跟变量名、函数名，不能与关键字重名一样，你的模块名称也最好别与系统内建模块名称重合。

指定引入模块中特定函数

当你使用 import mycode 的时候，你向当前工作空间引入了 mycode 文件中定义的所有函数，相当于：

In [ ]:

```
from mycode import *
```

你其实可以只引入当前需要的函数，比如，只引入 is_prime()

In [6]:

```
from mycode import is_prime
```

这种情况下，你就不必使用 mycode.is_prime() 了；而是就好像这个函数就写在当前工作空间一样，直接写 is_prime() ：

In [7]:

```
from mycode import is_prime
is_prime(3)
```

Out[7]:

```
True
```

注意，如果当前目录中并没有 mycode.py 这个文件，那么， mycode 会被当作目录名再被尝试一次，如果当前目录内，有个叫做 mycode 的目录（或称文件夹），那么， from mycode import * 的作用就是把 mycode 这个文件夹中的所有 .py 文件全部导入……

如果我们想要导入 foo 这个目录中的 bar.py 这个模块文件，那么，可以这么写：

```
import foo.bar
```

或者

```
from foo import bar
```

引入并使用化名

有的时候，或者为了避免混淆，或者为了避免输入太多字符，我们可以为引入的函数设定 **化名**（alias），而后使用化名调用函数。比如：

In [8]:

```
from mycode import is_prime as isp
isp(3)
```

Out[8]:

```
True
```

甚至干脆给整个模块取个化名：

In [13]:

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

import mycode as m

m.is_prime(3)
m.say_hi('mike', 'zoe')
```

Out[13]:

```
True
```

```
Hello, mike!
Hello, zoe!
```

模块中不一定只有函数

一个模块文件中，不一定只包含函数；它也可以包含函数之外的可执行代码。只不过，在 import 语句执行的时候，模块中的非函数部分的可执行代码，只执行一次。

有一个 Python 的彩蛋，恰好是可以用在此处的最佳例子 —— 这个模块是 this ，它的文件名是 this.py (https://github.com/python/cpython/blob/master/Lib/this.py)：

In [16]:

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

这个 this 模块中的代码如下：

In [ ]:

```
s = """Gur Mra bs Clguba, ol Gvz Crgref
Ornhgvshy vf orggre guna htyl.
Rkcyvpvg vf orggre guna vzcyvpvg.
Fvzcyr vf orggre guna pbzcyrk.
Pbzcyrk vf orggre guna pbzcyvpngrq.
Syng vf orggre guna arfgrq.
Fcnefr vf orggre guna qrafr.
Ernqnovyvgl pbhagf.
Fcrpvny pnfrf nera'g fcrpvny rabhtu gb oernx gur ehyrf.
Nygubhtu cenpgvpnyvgl orngf chevgl.
Reebef fubhyq arire cnff fvyragyl.
Hayrff rkcyvpvgyl fvyraprq.
Va gur snpr bs nzovthvgl, ershfr gur grzcgngvba gb thrff.
Gurer fubhyq or bar-- naq cersrenoyl bayl bar --boivbhf jnl gb qb vg.
Nygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er Qhgpu.
Abj vf orggre guna arire.
Nygubhtu arire vf bsgra orggre guna *evtug* abj.
Vs gur vzcyrzragngvba vf uneq gb rkcynva, vg'f n onq vqrn.
Vs gur vzcyrzragngvba vf rnfl gb rkcynva, vg znl or n tbbq vqrn.
Anzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!"""

d = {}
for c in (65, 97):
    for i in range(26):
        d[chr(i+c)] = chr((i+13) % 26 + c)

print("".join([d.get(c, c) for c in s]))
```

这个 this.py 文件中也没有什么函数，但，这个文件里所定义的变量，我们都可以在 import this 之后触达：

In [19]:

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

import this
this.d
this.s
```

Out[19]:

```
{'A': 'N',
 'B': 'O',
 'C': 'P',
 'D': 'Q',
 'E': 'R',
 'F': 'S',
 'G': 'T',
 'H': 'U',
 'I': 'V',
 'J': 'W',
 'K': 'X',
 'L': 'Y',
 'M': 'Z',
 'N': 'A',
 'O': 'B',
```

试试吧，试试能否独立读懂这个文件里的代码 —— 对初学者来说，还是挺练脑子的呢！

它先是通过一个规则生成了一个密码表，保存在 d 这个字典中；而后，将 s 这个变量中保存的"密文"翻译成了英文……

或许，你可以试试，看看怎样能写个函数出来，给你一段英文，你可以把它加密成跟它一样的"密文"？

dir() 函数

你的函数，保存在模块里之后，这个函数的用户（当然也包括你），可以用 dir() 函数查看模块中可触达的变量名称和函数名称：

In [23]:

```
import mycode
dir(mycode)
```

Out[23]:

```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'is_prime',
 'say_hi']
```