程序在运行的过程中，所有的变量都是在内存中，比如定义一个dict:

```
>>> d = dict(name='Bob',age=20,score=88)
>>> d
{'name': 'Bob', 'age': 20, 'score': 88}
```

可以随时修改变量，比如把name改成'Bill'，但是一旦程序结束，变量所占用的内存就被操作系统全部收回。但是没有把修改后的'Bill'存储到磁盘上，下次重新运行程序，变量又被初始化为'Bob'。

我们把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，在其他语言中也被称为serialization,marshalling,flattening等等，都是一个意思。

序列化之后可以把序列化的东西写入磁盘，或者通过网络传输到别的机器上。

反过来，把内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

Python提供了pickle模块来实现序列化。

```
import pickle
>>> pickle.dumps(d)
b'\x80\x03}q\x00(X\x04\x00\x00\x00nameq\x01X\x03\x00\x00\x00Bobq\x02X\x03\x00\x00\x00ageq\x03K\x14X\x05\x00\x00\x00scoreq\x04KXu.'
```

pickle.dumps()方法把任意对象序列化为一个Bytes，然后，就可以把这个bytes写入文件。或者用另一个方法pickle.dump()直接把对象序列化后写入一个file-like Object:

```
>>> f = open('D:\\编程\\Python3_learn\\IO编程\\dump.txt','wb')
>>> pickle.dump(d,f)
>>> f.close()
```

当我们要把对象从磁盘读到内存时，可以先把内容读到一个bytes，然后使用pickle.loads()方法反序列化出对象，也可以直接用pickle.load()方法从一个file-like Object中直接反序列化出对象。

```
>>> f = open('D:\\编程\\Python3_learn\\IO编程\\dump.txt','rb')
>>> a = pickle.load(f)
>>> f.close()
>>> a
{'name': 'Bob', 'age': 20, 'score': 88}
```

Pickle的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于PYthon，并且可能不同版本的Python彼此不兼容。

JSON

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如xml,但更好的是序列化为JSON,因为JSON表示出来的就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON不仅是标准格式，并且比xml更快，而且可以直接在Web页面中读取，非常方便。

JSON表示的对象就是标准的JavaScript语言的对象，JSON和Python内置的数据类型的对应如下： | JSON类型 | Python类型 | | ---------- | ---------- | | {} | dict | | [] | list | | "string" | str | | 1234.56 | int 或float | | true/false | True/false | | null | None |

Python内置的json模块提供了非常完善的Python对象到JSON格式的转换。

```
>>> import json
>>> d = dict(name='Bob',age=20,score=88)
>>> json.dumps(d)
'{"name": "Bob", "age": 20, "score": 88}'
```

dumps()方法返回一个str，内容就是标准的JSON。类似的，dump()方法可以直接把JSON写入一个file-like Object。

要把JSON反序列化为Python对象，用loads()或者对应的load()方法，前者把JSON字符串反序列化。后者从file-like Object中读取字符串并序列化：

```
>>> json_str = '{"age":20,"score":88,"name":"Bob"}'
>>> import json
>>> json.loads(json_str)
{'age': 20, 'score': 88, 'name': 'Bob'}
```

由于JSON标准规定JSON编码是UTF-8，所以我们总是能正确地在Python的str和JSON的字符串之间转换。

JSON进阶

Python的dict对象可以直接序列化为JSON的{},不过很多时候我们更喜欢用class表示对象，比如定义Student类，然后序列化：

```
>>> class Student(object):
    def __init__(self,name,age,score):
        self.name = name
        self.age = age
        self.score = score
    >>> s = Student('Bob',20,88)
>>> print(json.dumps(s))
Traceback (most recent call last):...
```

毫无疑问，得到一个typeError: 错误的原因是Student对象不是一个可序列化的JSON的对象。 可选参数default就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为Student专门写一个转换函数，再把函数传进去即可：

```
>>> def student2dict(std):
    return{
        'name':std.name,
        'age':std.age,
        'score':std.score}
```

这样，Student实例首先被student2dict()转换成dict,然后被顺利序列化为JSON:

```
>>> print(json.dumps(s,default=student2dict))
{"name": "Bob", "age": 20, "score": 88}
```

不过，下次如果遇到一个Teacher类的实例照样无法序列化为JSON。我们可以偷个懒，把任意class的实例变为dict:

```
>>> print(json.dumps(s,default = lambda obj:obj.__dict__))
{"name": "Bob", "age": 20, "score": 88}
```

因为每个class的实例都有一个**dict**属性，它就是一个dict,用来存储实例变量。少数例外例如定义了**slots**的class。 同样的道理，如果我们要把JSON反序列化为一个Student对象实例，load()方法首先转换出一个dict对象，然后我们传入的object_hook函数负责把dict转换为Student实例：

```
>>> import json
>>> class Student(object):
    def __init__(self,name,age,score):
        self.name = name
        self.age = age
        self.score = score


>>> def dict2student(d):
    return Student(d['name'],d['age'],d['score'])

>>> json_str = '{"age":20,"score":88,"name":"Bob"}'
>>> print(json.loads(json_str,object_hook=dict2student))
```

```
<__main__.Student object at 0x02D532F0>
```

打印出的是反序列化的Student实例对象。