



PEP 289 – Generator Expressions

Python List Comprehension

```
[expression(x) for x  
in existing_list if  
condition(x)]
```



WTMatter





Списковые включения

Списковые включения или генераторы списков – это способ построения нового списка за счет применения **выражения** к каждому элементу в последовательности, который связан с циклом **for** а также инструкции **if-else** для определения того, что в итоге окажется в финальном списке.



Пример

Do this

For this collection

In this situation

[x2 for x in range(0, 50) if x % 3 == 0]**



Способы формирования списков

- 1) при помощи циклов
- 2) при помощи функции `map()`
- 3) при помощи `list comprehension`



1. При помощи цикла `for`

```
s = []
```

```
for i in range(10):
```

```
    s.append(i ** 3) # Добавляем к списку куб каждого  
числа
```

```
print(s)
```

```
# [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

2. При помощи функции `map()`

```
list(map(lambda x: x ** 3, range(0,10)))
```

```
# [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Лаконично!

3. При помощи конструкции **list comprehension**

```
[x**3 for x in range(10)]
```

```
# [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```



Условие в конце включения

[«возв. значение» for «элемент списка» in «список» **if** «условие»]

#Получить все нечетные цифры в диапазоне от 0 до 9

```
[x for x in range(10) if x%2 == 1]
```

```
#[1, 3, 5, 7, 9]
```

Возведение в квадрат

```
[x**2 for x in range(10)]
```



Условие в начале включения

```
# Замена отрицательного диапазона нулем
>>> prices = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
>>> prices_ = [i if i > 0 else 0 for i in prices]
>>> prices_
[1.25, 0, 10.22, 3.78, 0, 1.16]
```




Условие в начале включения

```
from string import ascii_letters
```

```
letters = 'hыtфtrцзqp' # набор букв из разных  
алфавитов
```

```
# Разграничиваем буквы на английские и не английские
```

```
is_eng = [f'{letter}-ДА' if letter in ascii_letters  
else f'{letter}-НЕТ' for letter in letters]
```

```
# ['h-ДА', 'ы-НЕТ', 't-ДА', 'ф-НЕТ', 'т-НЕТ', 'r-ДА',  
'ц-НЕТ', 'з-НЕТ', 'q-ДА', 'п-НЕТ']
```



Вызов функции в выражении генераторов

```
# Замена отрицательного диапазона нулем
```

```
prices = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
```

```
def get_price(price_):  
    return price_ if price > 0 else 0
```

```
prices = [get_price(i) for i in prices]
```



Вложенная генерация

Представим список из слов, который мы хотим привести к сплошному списку из букв. Двойная итерация: по словам и по буквам

```
words = ['Я', 'изучаю', 'Python']
```

```
res = [letter for word in words for letter in word]  
print(res)
```

```
>>>res
```

```
['Я', 'и', 'з', 'у', 'ч', 'а', 'ю', 'Р', 'у', 'т', 'h', 'о', 'n']
```



Вложенная генерация

```
key = ["name", "age", "weight"]
```

```
value = ["Lilu", 25, 100 ]
```

```
[{x, y}  for x in key  for y in value ]
```

```
[  
{ 'Lilu', 'name' }, { 25, 'name' }, { 100, 'name' },  
{ 'Lilu', 'age' }, { 25, 'age' }, { 100, 'age' },  
{ 'weight', 'Lilu' }, { 'weight', 25 }, { 'weight', 100 }  
]
```



Вложенная генерация

```
>>> matrix = [[i for i in range(5)] for _ in range(6)]  
>>> matrix
```

```
[  
    [0, 1, 2, 3, 4],  
    [0, 1, 2, 3, 4],  
    [0, 1, 2, 3, 4],  
    [0, 1, 2, 3, 4],  
    [0, 1, 2, 3, 4],  
    [0, 1, 2, 3, 4]  
]
```

Внешний генератор [... for _ in range(6)] создает 6 строк в то время как внутренний генератор[i for i in range(5)]заполняет каждую строку значениями.



Вложенная генерация

Преобразование матрицы в плоский вид

```
matrix = [  
...     [0, 0, 0],  
...     [1, 1, 1],  
...     [2, 2, 2],  
... ]  
>>> flat = [col for row in matrix for col in row]  
>>> flat  
[0, 0, 0, 1, 1, 1, 2, 2, 2]
```



Вложенная генерация

Генерация таблицы умножения от 1 до 5

```
T = [[x*y for x in range(1, 6)] for y in range(1, 6)]
```

```
print(T)
```

```
[[1, 2, 3, 4, 5],  
 [2, 4, 6, 8, 10],  
 [3, 6, 9, 12, 15],  
 [4, 8, 12, 16, 20],  
 [5, 10, 15, 20, 25]]
```



Когда использовать генератор списков ?

- Использовать для выполнения простых фильтраций, модификаций или форматирования итерируемых объектов.
- Для увеличение производительности. Для компактности
- Для компактности
- Следует избегать использования генератора списков, если вам нужно добавить слишком много условий это делает код трудным для чтения.



Python

Dictionary Comprehension



Генераторы словарей

Генерация словаря похожа на генерацию списка и предназначена для создания словаря.

```
d = {}
```

```
for num in range(1, 10):
```

```
    d[num] = num**2
```

```
print(d)
```

```
{1:1, 2:4, 3:9, 4:16, 5:25, 6:36, 7:49, 8:64, 9: 81}
```

```
D = { num: num**2 for num in range(1, 10) }
```

```
>>>d
```

```
{1:1, 2:4, 3:9, 4:16, 5:25, 6:36, 7:49, 8:64, 9: 81}
```



Генераторы словарей

#Создадим словарь по списку кортежей

```
items = [('c', 3), ('d', 4), ('a', 1), ('b', 2)]
```

```
dict_variable = { key:value for (key,value) in items }
```

```
print(dict_variable)
```

Что если не будет значения `:value` ?

Set comprehensions!



Условие if

Добавим в конструкцию генератора условие фильтрации

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

Проверка, больше ли элемент, чем 2

```
filtered = {k:v for (k,v) in dict1.items() if v>2}
```

```
print(filtered)
```

```
# {'e': 5, 'c': 3, 'd': 4}
```



Условие if

Фильтрация по возрасту

```
ages = {  
    'kevin': 12,  
    'marcus': 9,  
    'evan': 31,  
    'nik': 31  
}  
  
f = {k:val for (k, val) in ages.items() if val > 25}  
print(new_ages)
```



Несколько условий if

#Последовательные операторы if работают так, как если бы между ними были логические **and**.

```
dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
r = {k:v for (k,v) in dict.items() if v>2 if v%2 == 0}
print(r)
# {'d': 4}
```



Вложенные генераторы словарей

```
dict = {'first':{'a':1}, 'second':{'b':2}}  
fd = {o_key: {float(i_val) for (i_key, i_val) in o_val.items()}  
for (o_key, o_val) in dict.items()}  
print(fd)  
# {'first': {1.0}, 'second': {2.0}}
```

Код имеет вложенный генератор словаря, то есть один генератор внутри другого. Как видите, вложенный генератор словаря может быть довольно трудным как для чтения, так и для понимания. Использование генераторов при этом теряет смысл (ведь мы их применяем для улучшения читабельности кода).



Использование enumerate функции

```
names = ['Harry', 'Hermione', 'Ron', 'Neville', 'Luna']  
index = {k:v for (k, v) in enumerate(names)}  
print(index)  
{'Harry':0, 'Hermione':1, 'Ron':2, 'Neville':3, 'Luna':4}
```



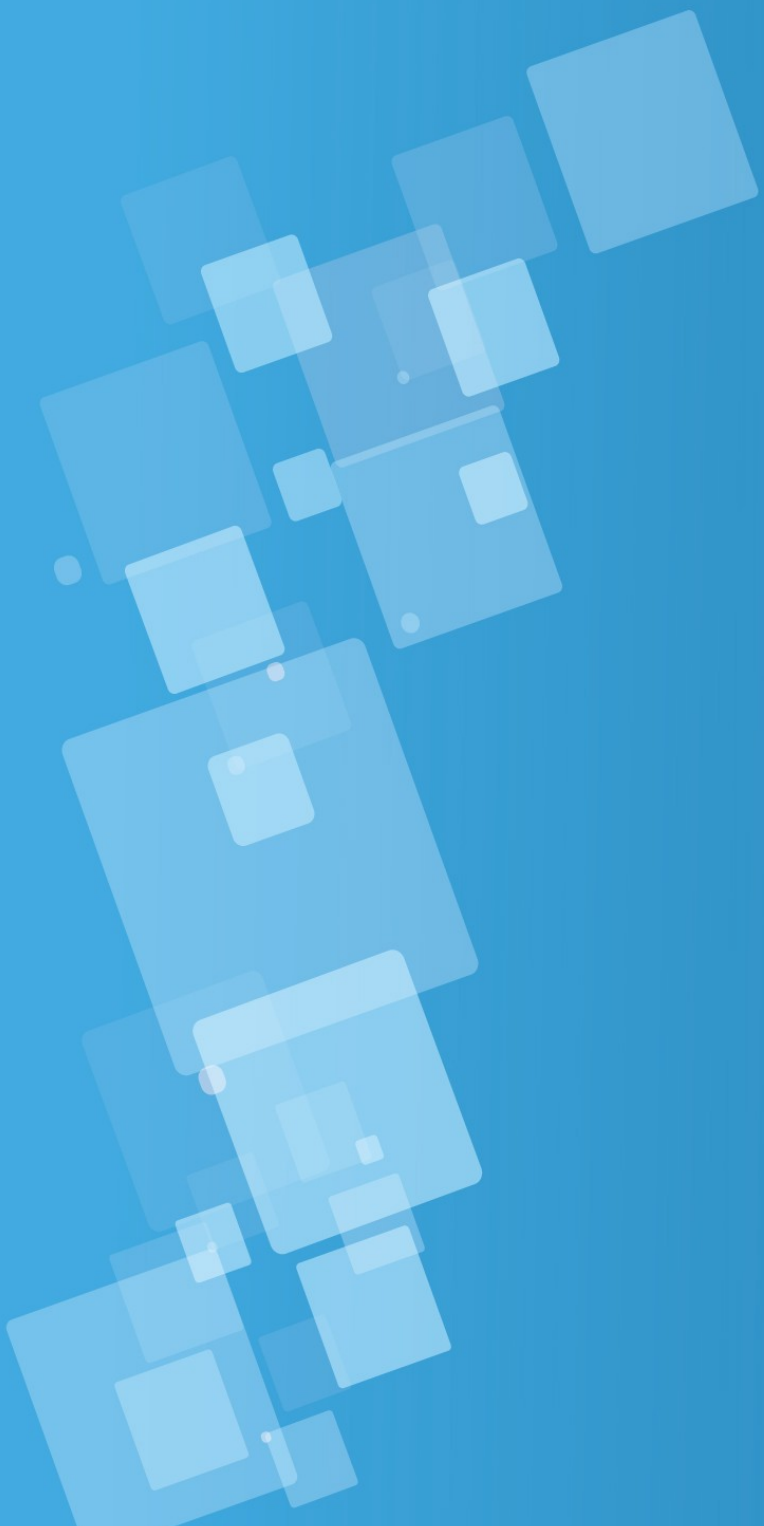

Когда использовать генераторы словарей?

Во всех случаях что и при генерации списков.



Резюме

Подобные конструкции позволяют создавать не только списки (list comprehension) и словари (dictionary comprehension), генераторы (generator expression – при помощи «()»), а также множества (set comprehension – при помощи «{ }») и кортежи «tuple()»). Принцип везде один и тот же.



Изолированное окружение. Пакеты, модули





Модули и пакеты

Система модулей позволяет вам логически организовать ваш код на Python. Группирование кода в модули значительно облегчает процесс написания и понимания программы. Говоря простым языком, модуль в Python это **просто файл**, содержащий код на Python. Каждый модуль в Python может содержать **переменные, объявления классов и функций**. Кроме того, в модуле может находиться **исполняемый код**.



Команда **import**

Вы можете использовать любой питоновский файл как модуль в другом файле, выполнив в нем команду `import`. Команда `import` в Python обладает следующим синтаксисом:

```
import module_1[, module_2[, ... module_N]
```

Когда интерпретатор Python встречает команду `import`, он импортирует этот модуль, если он присутствует в пути поиска Python. Путь поиска Python это список директорий, в которых интерпретатор производит поиск перед попыткой загрузить модуль.



Например, чтобы использовать модуль `math` следует написать:

```
import math
```

```
# Используем функцию sqrt из модуля math
```

```
print(math.sqrt(9))
```

```
# Печатаем значение переменной pi, определенной в math
```

```
print(math.pi)
```

Важно знать! Модуль загружается лишь однажды, независимо от того, сколько раз он был импортирован. Это препятствует циклическому выполнению содержимого модуля.



`from module import var`

Выражение `from ... import ...` не импортирует весь модуль, а только предоставляет доступ к конкретным объектам, которые мы указали.

```
# Импортируем из модуля math функцию sqrt
```

```
from math import sqrt
```

```
# Выводим результат выполнения функции sqrt.
```

```
# Обратите внимание, что нам больше не зачем указывать  
имя модуля
```

```
print (sqrt(144))
```

```
# Но мы уже не можем получить из модуля то, что не  
импортировали !!!
```

```
print (pi) # Выдаст ошибку
```




from *module* **import** *var, func, class*

Импортировать из модуля объекты можно через запятую.

```
from math import pi, sqrt
print(sqrt(121))
print(pi)
print(e)
```



from *module* import *

В Python так же возможно импортировать всё (переменные, функции, классы) за раз из модуля, для этого используется конструкция **from ... import ***

```
from math import *
```

```
# Теперь у нас есть доступ ко всем функция и  
переменным, определенным в модуле math
```

```
print(sqrt(121))
```

```
print(pi)
```

```
print(e)
```

Не импортируются объекты с ***_var***

Повторяющиеся названия перезаписываются. Такое поведение нужно отслеживать при импорте нескольких модулей.



```
import module_1 [ ,module_2 ]
```

За один раз можно импортировать сразу несколько модулей, для этого их нужно перечислить через запятую после слова `import`

```
import math, os  
print (math.sqrt (121))  
print (os.env)
```



import *module* **as** *my_alias*

Если вы хотите задать псевдоним для модуля в вашей программе, можно воспользоваться вот таким синтаксисом

```
import math as matan  
print (matan.sqrt (121))
```




Местонахождение модулей в Python

Когда вы импортируете модуль, интерпретатор Python ищет этот модуль в следующих местах:

- Директория, в которой находится файл, в котором вызывается команда импорта
- Если модуль не найден, Python ищет в каждой директории, определенной в консольной переменной **PYTHONPATH**.
- Если и там модуль не найден, Python проверяет путь заданный по умолчанию

Путь поиска модулей сохранен в системном модуле `sys` в переменной `path`. Переменная **`sys.path`** содержит все три вышеописанных места поиска модулей.



Получение списка всех модулей Python установленных на компьютере

Для того, чтобы получить список всех модулей, установленных на вашем компьютере достаточно выполнить команду:

```
>>>help("modules")
```

Через несколько секунд вы получите список всех доступных модулей.



Создание своего модуля в Python

Чтобы создать свой модуль в Python достаточно сохранить ваш скрипт с расширением `.py` Теперь он доступен в любом другом файле. Например, создадим два файла: `module_1.py` и `module_2.py` и сохраним их в одной директории. В первом запишем:

```
# module_1.py
def hello():
    print("Hello from module_1")
```

А во втором вызовем эту функцию:

```
# module_2.py
from module_1 import hello
hello()
```



Пакеты модулей в Python

Отдельные файлы-модули с кодом на Python могут объединяться в пакеты модулей. Пакет это директория (папка), содержащая несколько отдельных файлов-скриптов.

Например, имеем следующую структуру:

my_file.py

my_package

__init__.py

inside_file.py

В файле `inside_file.py` определена некая функция `foo`. Тогда чтобы получить доступ к функции `foo`, в файле `my_file` следует выполнить следующий код:

```
from my_package.inside_file import foo
```




Функция **dir()**

Встроенная функция `dir()` возвращает отсортированный список строк, содержащих все имена, определенные в модуле.

на данный момент нам доступны лишь встроенные функции

dir()

импортируем модуль `math`

import `math`

теперь модуль `math` в списке доступных имен

dir()

получим имена, определенные в модуле `math`

dir(math)



Инструкция `if __name__ == '__main__':`

`__name__` хранит название программы.

Если запустить файл напрямую, то значением `__name__` будет `__main__`.

Если запустить файл как модуль то значением `__name__`, будет название модуля.



Пример:

```
# Suppose this is foo.py.
```

```
import math
```

```
def function_a():  
    print("Function A")
```

```
def function_b():  
    print("Function B {}".format(math.sqrt(100)))
```

```
if __name__ == '__main__':  
    function_a()  
    function_b()
```

Менеджер пакетов **pip** (Python Package Index)





Установка **pip**

Начиная с Python версии 3.4, pip поставляется вместе с интерпретатором Python. Метод универсален и подходит для любой операционной системы, если в ней уже установлена какая-либо версия Python

Открыть консоль (терминал)

Скачать файл `get-pip.py`:

```
wget https://bootstrap.pypa.io/get-pip.py
```

Установить pip:

```
python3 get-pip.py
```



Использование **pip**

Самый распространённый способ использования **pip** - это через консоль (терминал). Чтобы использовать **pip**, в консоли нужно вызвать команду **pip** для Python2 или **pip3** для Python3. Для того, чтобы узнать какие команды есть в **pip** нужно вызвать **pip3 -help**:

Usage:

pip3 <command> [options]

Commands:

install	Install packages.
download	Download packages.
uninstall	Uninstall packages.
freeze	Output installed packages in requirements format.
list	List installed packages.
show	Show information about installed packages.
check	Verify installed packages have compatible dependencies.
config	Manage local and global configuration.
search	Search PyPI for packages.
wheel	Build wheels from your requirements.
hash	Compute hashes of package archives.
completion	A helper command used for command completion.
help	Show help for commands.



install

Команда `install` позволяет установить какой-либо пакет.

```
pip3 install Flask==2.1
```

После Flask мы также указали версию пакета, которую мы хотим установить. Это необязательно, если мы не укажем версию, то установится самая последняя версия пакета, которая присутствует в репозитории.

```
pip3 install Flask
```

Также можно указывать ограничения на версии, к примеру, что хотим установить Django не старше версии

```
pip3 install Flask > 2.1
```



pip install -r *reqfile.txt*

Установка пакетов перечисленных в файле

```
pip3 install -r requirements.txt
```

Файл requirements.txt

```
Flask==2.0.2
```

```
Flask-JWT-Extended==4.3.1
```

```
Flask-RESTful==0.3.9
```

```
Flask-SQLAlchemy==2.5.1
```

```
passlib==1.7.4
```

```
pymongo==4.0.1
```

```
Werkzeug==2.0.2
```

Установленные пакеты будут храниться в папке `/python3.X/site-packages`



Импортирование в скрипте

После установки пакета его можно импортировать в скрипт.

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```



Понижение версии `--force-reinstall`

А если пакет уже установлен и вы хотите понизить его версию добавьте `--force-reinstall` вот так:

```
pip install 'stevedore>=1.3.0,<1.4.0' --force-reinstall
```



Проблемы **--no-cache-dir**

Иногда ранее установленная версия кэшируется.

При установке новой указанной версии пакета

```
pip install pillow==5.2.0
```

pip возвращает следующее:

```
Требование уже выполнено: pillow==5.2.0 in  
/home/ubuntu/anaconda3/lib/python3.6/site-packages (5.2.0)
```

Мы можем использовать `--no-cache-dir` вместе с `-I`, чтобы перезаписать это

```
pip install --no-cache-dir -I pillow==5.2.0
```



uninstall - удаление пакета

Удаление установленного пакета

```
pip uninstall Flask
```

Удаление пакетов перечисленных в файле

```
pip uninstall -r requirements.txt
```

Удаление всех установленных пактов

```
pip freeze | xargs pip uninstall -y
```



download – *закачка без установки*

Позволяет скачать пакеты без установки.

```
pip3 download Flask
```

Пакеты скачиваются с зависимостями и имеют расширения .whl
Установить их в проект можно через install так:

```
pip install --find-links=/download Flask-2.1.1-py3-  
none-any.whl
```



pip list

Позволяет просмотреть список всех установленных в системе пакетов.

Пример:

```
pip3 list
```



pip show

Позволяет просмотреть информацию об установленном в системе пакете.

Пример:

```
pip3 show Flask
```

В дополнение к pip show есть пакет



Спасибо за внимание !



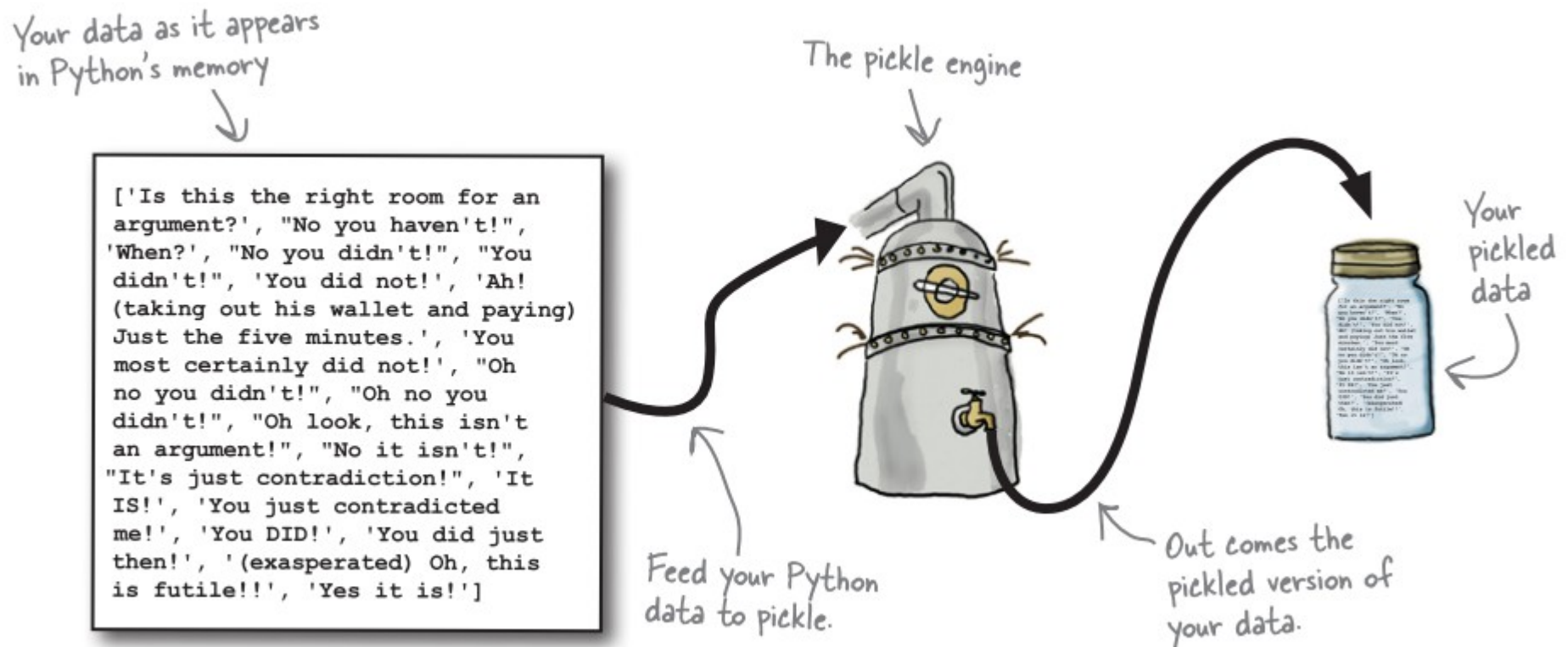
Сериализация

десериализация



PYTHON SERIALIZATION WITH PICKLE

Pickle - алгоритм сериализации и десериализации объектов Python.





Сериализация/десериализация

Сериализация (англ. **serialization**) — процесс перевода какой-либо структуры данных в любой другой, более удобный для хранения формат. Обратной к операции сериализации является операция десериализации (англ. **deserialization**) — восстановление начального состояния структуры данных из битовой последовательности.

Самой основной структурой данных в языке программирования Python является объект. Сериализация и десериализация объектов используется в том случае, если нам надо передавать информацию между запусками одной программы или между несколькими программами.



Способы сохранить/восстановить объект.

1. Pickle
2. JSON
3. YAML



pickle

Модуль pickle реализует мощный алгоритм сериализации и десериализации объектов Python. "Pickling" - процесс преобразования объекта Python в поток байтов, а "unpickling" - обратная операция, в результате которой поток байтов преобразуется обратно в Python-объект. Так как поток байтов легко можно записать в файл, модуль pickle широко применяется для сохранения и загрузки сложных объектов в Python.

Часто сериализация используется для сохранения пользовательских данных между разными сессиями работы приложения, обычно игры.



Что можно консервировать ?

- None, True или False
- Числа , вещественные и комплексные числа
- Строки , байт строки.
- Списки, наборы , картежи и словари
- Функции определенные def в голобальной области
- Классы определенные в глобальной области
- Экземпляры классов у которых можно вызвать `__dict__`

Интерфейс модуля `dir(pickle)`

```
['ADDITEMS', 'APPEND', 'APPENDS', 'BINBYTES', 'BINBYTES8', 'BINFLOAT',
'BINGET', 'BININT', 'BININT1', 'BININT2', 'BINPERSID', 'BINPUT',
'BINSTRING', 'BINUNICODE', 'BINUNICODE8', 'BUILD', 'BYTEARRAY8',
'DEFAULT_PROTOCOL', 'DICT', 'DUP', 'EMPTY_DICT', 'EMPTY_LIST',
'EMPTY_SET', 'EMPTY_TUPLE', 'EXT1', 'EXT2', 'EXT4', 'FALSE', 'FLOAT',
'FRAME', 'FROZENSET', 'FunctionType', 'GET', 'GLOBAL', 'HIGHEST_PROTOCOL',
'INST', 'INT', 'LIST', 'LONG', 'LONG1', 'LONG4', 'LONG_BINGET',
'LONG_BINPUT', 'MARK', 'MEMOIZE', 'NEWFALSE', 'NEWOBJ', 'NEWOBJ_EX',
'NEWTTRUE', 'NEXT_BUFFER', 'NONE', 'OBJ', 'PERSID', 'POP', 'POP_MARK',
'PROTO', 'PUT', 'PickleBuffer', 'PickleError', 'Pickler', 'PicklingError',
'PyStringMap', 'READONLY_BUFFER', 'REDUCE', 'SETITEM', 'SETITEMS',
'SHORT_BINBYTES', 'SHORT_BINSTRING', 'SHORT_BINUNICODE', 'STACK_GLOBAL',
'STOP', 'STRING', 'TRUE', 'TUPLE', 'TUPLE1', 'TUPLE2', 'TUPLE3',
'UNICODE', 'Unpickler', 'UnpicklingError', '_Framer',
'_HAVE_PICKLE_BUFFER', '_Pickler', '_Stop', '_Unframer', '_Unpickler',
'__all__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', '__compat_pickle',
'_dump', '_dumps', '_extension_cache', '_extension_registry',
'_getattribute', '_inverted_registry', '_load', '_loads', '_test',
'_tuplesize2code', 'bytes_types', 'codecs', 'compatible_formats',
'decode_long', 'dispatch_table', 'dump', 'dumps', 'encode_long',
'format_version', 'io', 'islice', 'load', 'loads', 'maxsize', 'pack',
'partial', 're', 'sys', 'unpack', 'whichmodule']
```




Методы модуля

Модуль `pickle` предоставляет следующие методы, чтобы сделать процесс `pickling` (пиклинг) более удобным:

Синтаксис:

```
pickle.dump(obj, file, protocol=None, *,  
fix_imports=True, buffer_callback=None)
```

Записывает сериализованный объект в файл. Дополнительный аргумент `protocol` указывает используемый протокол. По умолчанию равен 3 и именно он рекомендован для использования в Python 3 (несмотря на то, что в Python 3.4 добавили протокол версии 4 с некоторыми оптимизациями). В любом случае, записывать и загружать надо с одним и тем же протоколом.

```
import pickle
```

```
obj = {"one": 123, "two": [1, 2, 3]}
```

```
fd = open("data.pkl", "wb")
```

```
pickle.dump(obj, fd, pickle.HIGHEST_PROTOCOL)
```



`pickle.dumps`

Возвращает обработанное представление объекта `obj` в виде байтового объекта, без записи его в файл.

Синтаксис:

```
pickle.dumps(obj, protocol=None, *, fix_imports=True,  
               buffer_callback=None)
```

Параметры:

`obj` – объект Python, подлежащий сериализации,

`file` – файловый объект

`protocol=None` – протокол сериализации

`fix_imports=True` – сопоставление данных Python2 и Python3

`buffer_callback=None` – сериализация буфера в файл как часть потока `pickle`.

Пример:

```
import pickle  
obj = {"one": 123, "two": [1, 2, 3]}  
output = pickle.dumps(obj, 2)
```



`pickle.load`

Восстанавливает сериализованный объект из файла

Синтаксис:

```
pickle.load(file, *, fix_imports=True,  
encoding="ASCII", errors="strict")
```

Версия протокола определяется автоматически

Считайте байтовое представление объекта из открытого файла и возвращает сериализованный объект.

Пример:

```
import pickle  
  
with open("data.pkl", "rb") as f:  
    obj = pickle.load(f)
```



`pickle.loads`

Восстанавливает сериализованный объект в обычное представление из байтового.

Синтаксис:

```
pickle.loads(bytes_object, *, fix_imports=True,  
encoding="ASCII", errors="strict")
```

```
import pickle
```


```
obj = {"one": 123, "two": [1, 2, 3]}
```

```
output = pickle.dumps(obj)
```

```
new_obj = pickle.loads(output)
```



JSON Serialization



Пользоваться pickle лучше только в рамках python-приложения. При обмене данными между разными языками программирования обычно используют модуль json. Также pickle никак не решает вопрос безопасности данных. Поэтому не следует десериализовать данные из неизвестных источников.

Для работы с форматом JSON в Python используется модуль json

```
import json
```

Интерфейс модуля `dir(json)`

```
['JSONDecodeError', 'JSONDecoder', 'JSONEncoder',  
'__all__', '__author__', '__builtins__', '__cached__',  
'__doc__', '__file__', '__loader__', '__name__',  
'__package__', '__path__', '__spec__', '__version__',  
'_default_decoder', '_default_encoder', 'codecs',  
'decoder', 'detect_encoding', 'dump', 'dumps',  
'encoder', 'load', 'loads', 'scanner']
```



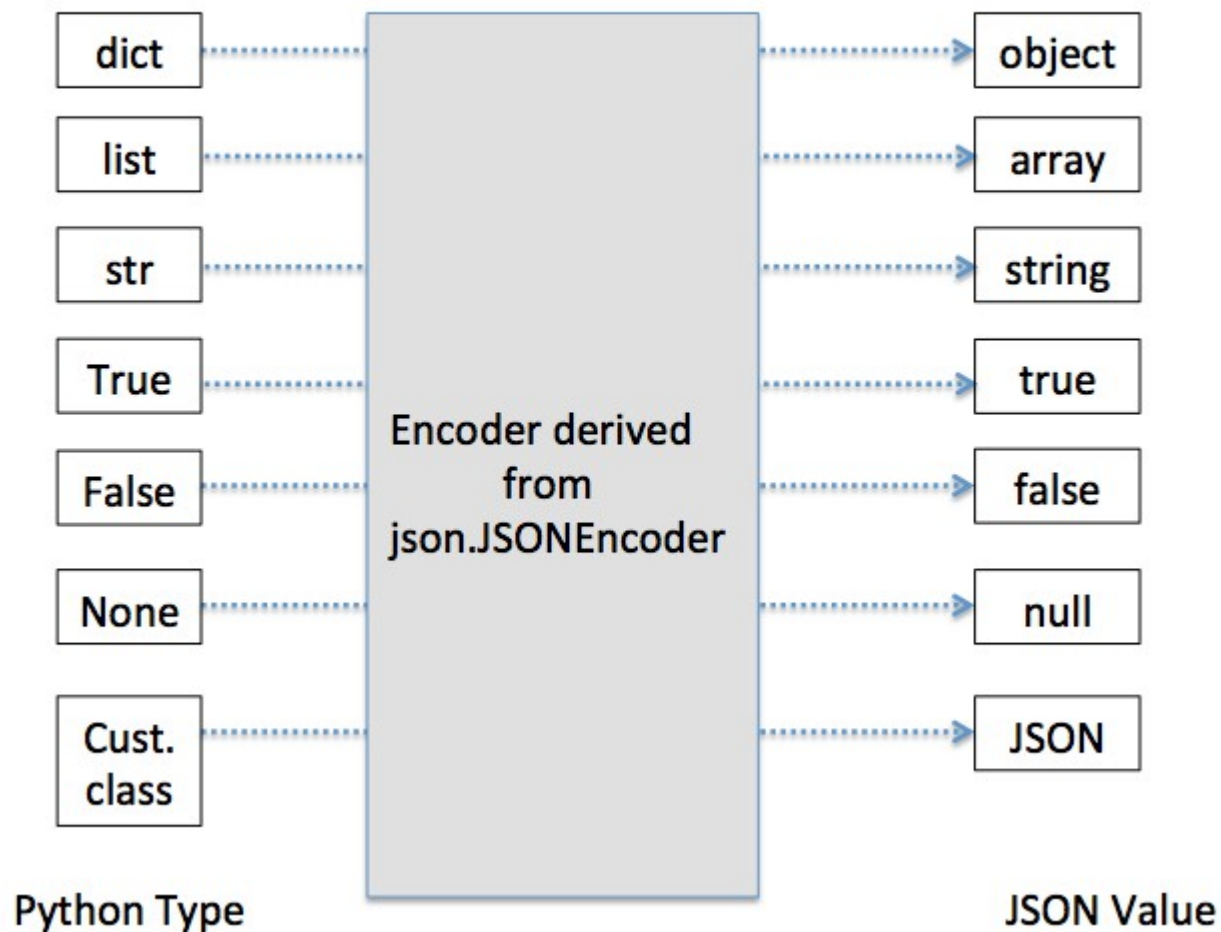
Сериализация и десериализация

Сериализация JSON

Модуль `json` предоставляет удобный метод `dump()` для записи данных в файл. Существует также метод `dumps()` для записи данных в обычную строку. Типы данных Python кодируются в формат JSON в соответствии с интуитивно понятными правилами преобразования, представленными в виде таблицы ниже.

Кодировщики и декодировщики

Serialization using specialized classes of `json.JSONEncoder`





Сериализация и десериализация в строку

json.dumps (*object*) – сериализует obj в строку JSON-формата

json.loads (*output*) – десериализует строку содержащую документ JSON в объект Python.

```
import json
```

```
obj = {"one": 123, "two": [1, 2, 3]}
```

```
output = json.dumps (obj)
```

```
new_obj = json.loads (output)
```

Во что превратится tupl – ?



Сериализация в файл

json.dump(obj, ff, indent=" ") – сериализует объект в файл.

```
import json
```

```
obj = {"one": 123, "two": [1, 2, 3]}
```

```
with open("data.json", "wt") as f:
```

```
    json.dump(obj, f, indent=4)
```

indent – количество отступов при записи



Десериализация из файла

json.load(*file_descriptor*) – десериализует содержимое файла

```
import json  
with open("data.json", "rt") as f:  
    obj = json.load(f)
```



Десериализация из файла


```
import json
```

```
with open("data.json", "rt") as f:
```

```
    obj = json.load(f)
```



YAML Serialization



YAML — это язык для сериализации данных, который отличается простым синтаксисом и позволяет хранить сложноорганизованные данные в компактном и читаемом формате.

Для работы с форматом YAML в Python используется модуль `yaml`

Установка

```
$ pip install pyyaml
```

Подключение

```
>>> import yaml
```

Интерфейс модуля `dir(yaml)`

```
[ 'StreamEndToken', 'StreamStartEvent', 'StreamStartToken',  
'TagToken', 'Token', 'UnsafeLoader', 'ValueToken',  
'YAMLError', 'YAMLObject', 'YAMLObjectMetaclass',  
'__builtins__', '__cached__', '__doc__', '__file__',  
'__loader__', '__name__', '__package__', '__path__',  
'__spec__', '__version__', '__with_libyaml__', '_yaml',  
'add_constructor', 'add_implicit_resolver',  
'add_multi_constructor', 'add_multi_representer',  
'add_path_resolver', 'add_representer', 'compose',  
'compose_all', 'composer', 'constructor', 'cyaml', 'dump',  
'dump_all', 'dumper', 'emit', 'emitter', 'error', 'events',  
'full_load', 'full_load_all', 'io', 'load', 'load_all',  
'loader', 'nodes', 'parse', 'parser', 'reader',  
'representer', 'resolver', 'safe_dump', 'safe_dump_all',  
'safe_load', 'safe_load_all', 'scan', 'scanner',  
'serialize', 'serialize_all', 'serializer', 'tokens',  
'unsafe_load', 'unsafe_load_all', 'warnings']
```




Сериализация и десериализация

yaml.dump(obj) – сериализация объекта в строку

yaml.load(new_obj) – десериализация объекта в строку

```
import yaml
```

```
from yaml.loader import SafeLoader
```

```
obj = {"one": 123, "two": [1, 2, 3]}
```

```
output = yaml.dump(obj)
```

```
new_obj = yaml.load(output, Loader=SafeLoader)
```



Сериализация в файл

```
import yaml

obj = {"one": 123, "two": [1, 2, 3]}
with open("data.yaml", "wt") as f:
    yaml.dump(obj, f)
```



Десериализация из файла

```
import yaml

with open("data.yml", "wt") as f:
    obj = yaml.load(f)
```



Заключение

Сериализация и десериализация объектов Python является важным аспектом распределенных систем. Вам часто приходится взаимодействовать с другими системами, реализованными на других языках, а иногда просто нужно сохранить состояние своей программы в постоянном хранилище.

Python поставляется с несколькими схемами сериализации в своей стандартной библиотеке, а многие другие доступны в качестве сторонних модулей.