目录

1. 某科学的Python小册

> 人生は辛くて短くて、だからPythonを使ってください！ ——alphardex

本书是笔者知乎专栏Pythonが大好き的文集汇编，随着专栏的更新本文集也会同步进行修订。

离线PDF版本：猛戳这里

> 如果说优雅也有缺点的话，那就是你需要艰巨的工作才能得到它，需要良好的教育才能欣赏它。 —— Edsger Wybe Dijkstra

笔者精心整理了许多实用的Python tricks，欢迎各位Pythonistia参考。

1. 基本

1.1. f-string

```
name = 'alphardex'
f'Ore wa {name} desu, {4 * 6} sai, gakusei desu.'
# 'Ore wa alphardex desu, 24 sai, gakusei desu.'
```

1.2. 三元运算符

```
# if condition:
#     fuck
# else:
#     shit
fuck if condition else shit
```

1.3. 字符串的拼接，反转与分割

```
letters = ['h', 'e', 'l', 'l', 'o']
''.join(letters)
```

```
# 'hello'
letters.reverse()
# ["o", "l", "l", "e", "h"]
name = 'nameless god'
name.split(' ')
# ['nameless', 'god']
```

1.4. 判断元素的存在性

```
'fuck' in 'fuck you'
# True
'slut' in ['bitch', 'whore']
# False
'company' in {'title': 'SAO III', 'company': 'A1 Pictures'}
# True
```

2. 函数

2.1. 匿名函数

类似ES6的箭头函数，函数的简化写法，配合map、filter、sorted等高阶函数食用更佳

```
# def foo(parameters):
#     return expression
foo = lambda parameters: expression
```

2.1.1. map - 映射

```
numbers = [1, 2, 3, 4, 5]
```

```
list(map(lambda e: e ** 2, numbers))
# [1, 4, 9, 16, 25]
```

2.1.2. filter - 过滤

```
values = [None, 0, '', True, 'alphardex', 666]
list(filter(lambda e:e, values))
# [True, "alphardex", 666]
```

2.1.3. sort - 排序

```
tuples = [(1, 'kirito'), (2, 'asuna'), (4, 'alice'), (3, 'eugeo'
)]
sorted(tuples, key=lambda x: x[1])
# [(4, 'alice'), (2, 'asuna'), (3, 'eugeo'), (1, 'kirito')]
```

2.1.4. 其他骚操作

```
from functools import reduce
# 求1到100的积
reduce(lambda x, y: x * y, range(1, 101))
# 求和就更简单了
sum(range(101))
# 5050
```

扁平化列表

```
from functools import reduce
li = [[1,2,3],[4,5,6], [7], [8,9]]
flatten = lambda li: [item for sublist in li for item in sublist
]
```

```
flatten(li)
# [1, 2, 3, 4, 5, 6, 7, 8, 9]
# 或者直接用more_itertools这个第三方模块
# from more_itertools import flatten
# list(flatten(li))
```

2.2. 星号和双星号

2.2.1. 数据容器的合并

```
l1 = ['a', 'b']
l2 = [1, 2]
[*l1, *l2]
# ['a', 'b', 1, 2]
d1 = {'name': 'alphardex'}
d2 = {'age': 24}
{**d1, **d2}
# {'name': 'alphardex', 'age': 24}
```

2.2.2. 函数参数的打包

```
def foo(*args):
    print(args)
foo(1, 2)
# (1, 2)

def bar(**kwargs):
    print(kwargs)
bar(name='alphardex', age=24)
# {'name': 'alphardex', 'age': 24}
```

3. 数据容器

3.1. 列表

3.1.1. 推导式

笔者最爱的语法糖:)

```
even = [i for i in range(10) if not i % 2]
even
# [0, 2, 4, 6, 8]
```

3.1.2. 同时迭代元素与其索引

用enumerate即可

```
li = ['a', 'b', 'c']
print([f'{i+1}. {elem}' for i, elem in enumerate(li)])
# ['1. a', '2. b', '3. c']
```

3.1.3. 元素的追加与连接

append在末尾追加元素，extend在末尾连接元素

```
li = [1, 2, 3]
li.append([4, 5])
li
# [1, 2, 3, [4, 5]]
li.extend([4, 5])
li
# [1, 2, 3, [4, 5], 4, 5]
```

3.1.4. 测试是否整体/部分满足条件

all测试所有元素是否都满足于某条件，any则是测试部分元素是否满足于某条件

```
all([e<20 for e in [1, 2, 3, 4, 5]])
# True
any([e%2==0 for e in [1, 3, 4, 5]])
# False
```

3.1.5. 同时迭代2个以上的可迭代对象

用zip即可

```
subjects = ('nino', 'miku', 'itsuki')
predicates = ('saikou', 'ore no yome', 'is sky')
print([f'{s} {p}' for s, p in zip(subjects, predicates)])
# ['nino saikou', 'miku ore no yome', 'itsuki is sky']
```

3.1.6. 去重

利用集合的互异性

```
li = [3, 1, 2, 1, 3, 4, 5, 6]
list(set(li))
# [1, 2, 3, 4, 5, 6]
# 如果要保留原先顺序的话用如下方法即可
sorted(set(li), key=li.index)
# [3, 1, 2, 4, 5, 6]
```

3.1.7. 解包

此法亦适用于元组等可迭代对象

最典型的例子就是2数交换

```
a, b = b, a
# 等价于 a, b = (b, a)
```

用星号运算符解包可以获取剩余的元素

```
first, *rest = [1, 2, 3, 4]
first
# 1
rest
# [2, 3, 4]
```

3.2. 字典

3.2.1. 遍历

```
d = {'name': "alphardex", 'age': 24}
[key for key in d.keys()]
# ['name', 'age']
[value for value in d.values()]
# ['alphardex', 24]
[f'{key}: {value}' for key, value in d.items()]
# ['name: alphardex', 'age: 24']
```

3.2.2. 排序

```
import operator
data = [{'rank': 2, 'author': 'alphardex'}, {'rank': 1, 'author'
: 'alphardesu'}]
```

```
data_by_rank = sorted(data, key=operator.itemgetter('rank'))
data_by_rank
# [{'rank': 1, 'author': 'alphardesu'}, {'rank': 2, 'author': 'a
lphardex'}]
# 其实了解lambda的话也可以这么写：data_by_rank = sorted(data, key=
lambda x: x['rank'])
# sorted的排序默认是asc（升序），可以用reverse选项来实现desc（降序）
data_by_rank_desc = sorted(data, key=lambda x: x['rank'], revers
e=True)
# [{'rank': 2, 'author': 'alphardex'}, {'rank': 1, 'author': 'al
phardesu'}]
```

3.2.3. 反转

```
d = {'name': 'alphardex', 'age': 24}
{v: k for k, v in d.items()}
# {'alphardex': 'name', 24: 'age'}
```

4. 语言专属特性

4.1. 下划线_的几层含义

4.1.1. repl中暂存结果

```
>>> 1 + 1
# 2
>>> _
# 2
```

4.1.2. 忽略某个变量

```
filename, _ = 'eroge.exe'.split('.')
filename
# 'eroge'
for _ in range(2):
    print('wakarimasu')
# wakarimasu
# wakarimasu
```

4.1.3. i18n国际化

```
_("This sentence is going to be translated to other language.")
```

4.1.4. 增强数字的可读性

```
1_000_000
# 1000000
```

4.2. 上下文管理器

用于资源的获取与释放，以代替try-except语句

常用于文件IO，锁的获取与释放，数据库的连接与断开等

```
# try:
#     f = open(input_path)
#     data = f.read()
# finally:
#     f.close()
with open(input_path) as f:
    data = f.read()
```

可以用@contextmanager来实现上下文管理器

```
from contextlib import contextmanager

@contextmanager
def open_write(filename):
    try:
        f = open(filename, 'w')
        yield f
    finally:
        f.close()

with open_write('onegai.txt') as f:
    f.write('Dagakotowaru!')
```

4.3. 静态类型注解

给函数参数添加类型，能提高代码的可读性和可靠性，大型项目的最佳实践之一

```
from typing import List

def greeting(name: str) -> str:
    return f'Hello {name}.'

def gathering(users: List[str]) -> str:
    return f"{', '.join(users)} are going to be raped."

print(greeting('alphardex'))
print(gathering(['Bitch', 'slut']))
```

4.4. 多重继承

在django中经常要处理类的多重继承的问题，这时就要用到super函数

如果单单认为super仅仅是“调用父类的方法”，那就错了

其实super指的是MRO中的下一个类，用来解决多重继承时父类的查找问题

MRO是啥？Method Resolution Order（方法解析顺序）

看完下面的例子，就会理解了

```
class A:
    def __init__(self):
        print('A')

class B(A):
    def __init__(self):
        print('enter B')
        super().__init__()
        print('leave B')

class C(A):
    def __init__(self):
        print('enter C')
        super().__init__()
        print('leave C')

class D(B, C):
    pass

d = D()
# enter B
# enter C
# A
# leave C
# leave B
print(d.__class__.__mro__)
# (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.
```

```
 C'>, <class '__main__.A'>, <class 'object'>)
```

首先，因为D继承了B类，所以调用B类的__init__，打印了 `enter B`

打印 `enter B` 后的super寻找MRO中的B的下一个类，也就是C类，并调用其__init__，打印 `enter C`

打印 `enter C` 后的super寻找MRO中的C的下一个类，也就是A类，并调用其__init__，打印 `A`

打印 `A` 后回到C的__init__，打印 `leave C`

打印 `leave C` 后回到B的__init__，打印 `leave B`

想掌握Python标准库，读它的官方文档很重要。本文并非此文档的复制版，而是对每一个库的一句话概括以及它的主要函数，由此用什么库心里就会有数了。

1.1. 文本处理

- string: 提供了字符集：ascii_lowercase, ascii_uppercase, digits, hexdigits, punctuation
- re: 正则表达式支持(pattern, string)：match, search, findall, sub, split, finditer

1.2. 数据结构

- datetime: 处理日期，建议用arrow代替
- calendar: 日历: Calendar
- collections: 其他数据结构: deque, Counter, defaultdict, namedtuple
- heapq: 堆排序实现: nlargest, nsmallest, merge
- bisect: 二分查找实现: bisect, insort
- array: 数列实现: array
- copy: 浅拷贝和深拷贝: copy, deepcopy
- pprint: 漂亮地输出文本: pprint
- enum: 枚举类的实现: Enum

1.3. 数学

- math: 数学函数库，函数太多故不一一列举
- fractions: 分数运算: Fraction as F
- random: 随机数: choice, randint, randrange, sample, shuffle, gauss
- statistics: 统计学函数: mean, median, mode, variance

1.4. 函数式编程

- itertools: 迭代器工具: permutations, combinations, product, chain, repeat, cycle, accumulate
- functools: 函数工具: @wraps, reduce, partial, @lrucache, @singledispatch
- operator: 基本运算符

1.5. 文件目录访问

- pathlib: 对路径对象进行操作，完美替代os.path: Path

- fileinput: 读取一个或多个文件并处理行: input
- filecmp: 比较两个文件是否相同: cmp
- tempfile：用来创建临时文件，一关闭就自动删除：TemporaryFile
- linecache: 读取文件的行，缓存优化: getline, getlines
- shutil: 文件操作: copy, copytree, rmtree, move, make_archive

1.6. 数据持久化

- pickle: 文件pickle序列化: dump, dumps, load, loads
- sqlite3: sqlite数据库接口

1.7. 文件格式

- csv: 处理csv文件: reader, writeheader, writerow
- configparser: 处理配置文件: ConfigParser, get, sections

1.8. 密码学

- hashlib: 哈希加密算法: sha256, hexdigest
- secrets：密钥生成：token_bytes, token_hex, token_urlsafe

1.9. 操作系统

- os: 操作系统，具体看文档
- io: 在内存中读写str和bytes: StringIO, BytesIO, write, get_value
- time: 计时器: time, sleep, strftime
- argparse, getopt: 命令行处理，建议用click或docopt代替
- logging: 打日志: debug, info, warning, error, critical，建议用loguru代替
- getpass: 获取用户输入的密码: getpass
- platform: 提供跨平台支持: uname, system

1.10. 并发执行

- threading: 多线程模型: Thread, start, join
- multiprocessing: 多进程模型: Pool, map, Process
- concurrent.futures: 异步执行模型: ThreadPoolExecutor, ProcessPoolExecutor
- subprocess: 子进程管理: run
- sched: 调度工具，建议用schedule代替
- queue: 同步队列: Queue

1.11. 进程间通信和网络

- asyncio: 异步IO, eventloop, 协程: get_event_loop, run_until_complete, wait, async和await关键字

1.12. 网络数据处理

- email: 处理email，建议用yagmail代替
- json: 处理json: dumps, loads
- base64: 处理base64编码: b64encode, b64decode

1.13. 结构化标记语言工具

- html: 转义html: escape, unescape
- html.parser: 解析html，建议用parsel代替

1.14. 网络协议支持

- webbrowser: 打开浏览器: open
- wsgiref: 实现WSGI接口
- uuid: 通用唯一识别码: uuid1, uuid3, uuid4, uuid5

- ftplib, poplib, imaplib, nntplib, smptlib, telnetlib: 实现各种网络协议

其余库用requests代替

1.15. 程序框架

- turtle: 画图工具
- cmd: 实现交互式shell
- shlex: 利用shell的语法分割字符串: split

1.16. GUI

- tkinter：可用pysimplegui代替，超好用

1.17. 开发工具

- typing: 类型注解，可配合mypy对项目进行静态类型检查
- pydoc: 查阅模块文档: python -m pydoc [name]
- doctest: 文档测试: python -m doctest [pyfile]
- unittest: 单元测试: python -m unittest [pyfile]

1.18. DEBUG和性能优化

- pdb: Python Debugger: python -m pdb [pyfile]
- cProfile: 分析程序性能: python -m cProfile [pyfile]
- timeit: 检测代码运行时间: python -m timeit [pyfile]

1.19. 软件打包发布

- setuptools: 编写setup.py专用: setup, find_packages
- venv: 创建虚拟环境，建议用pipenv代替

1.20. Python运行时服务

- sys: 系统环境交互: argv, path, exit, stderr, stdin, stdout
- builtins: 所有的内置函数和类，默认引进（还有一个boltons扩充了许多有用的函数和类）
- __main__: 顶层运行环境，使得python文件既可以独立运行，也可以当做模块导入到其他文件。
- warnings: 警告功能（代码过时等）: warn
- contextlib: 上下文管理器实现: @contextmanager
- inspect：用于获取对象的各种信息（自省）

1.21. 自定义Python解释器

- code: 实现自定义的Python解释器（比如Scrapy的shell）: interact

参考repo：https://github.com/hemanth/functional-programming-jargon

1.1. Arity - 函数参数个数

```
import inspect
add = lambda a, b: a + b
len(inspect.getfullargspec(add).args)
# 2
```

1.2. Higher-Order Function - 高阶函数

以函数为参数或返回值

```
is_type = lambda type_: lambda x: isinstance(x, type_)
li = [0, '1', 2, None]
[l for l in li if is_type(int)(l)]
# [0, 2]
```

1.3. Closure - 闭包

闭包是一种在变量作用域之外访问变量的方法。是一种将函数存储在环境中的方法。

闭包是一个作用域，它捕获函数的局部变量以便访问，即使在执行已经移出定义它的块之后也是如此。

```
add_to = lambda x: lambda y: x + y
add_to_five = add_to(5)
add_to_five(3)
# 8
```

函数addTo()返回一个函数（内部称为add()），将它存储在名为addToFive的变量中，它带有参数为5的柯里化调用。

理想情况下，当函数addTo完成执行时，其作用域与局部变量add，x，y就变得不可访问。 但是，它在调用addToFive()时返回8。 这意味着即使在代码块完成执行后也会保存函数addTo的状态，否则无法知道addTo被调用为addTo(5)并且x的值被设置为5。

词法作用域范围是它能够找到x和add的值的原因 - 已经完成执行的父项的私有变量。该值称为闭包。

堆栈以及函数的词法范围以对父项的引用形式存储。这可以防止关闭和底层变量被垃圾收集（因为至少有一个对它的实时引用）。

闭包是一种通过引用其主体外部的字段来包围其周围状态的函数。封闭状态保持在闭包的调用之间。

1.4. Partial Function - 偏函数

通过对原始函数预设参数来创建一个新的函数

```
from functools import partial
add3 = lambda a, b, c: a + b + c
five_plus = partial(add3, 2, 3)
five_plus(4)
# 9
```

1.5. Currying - 柯里化

将一个多元函数转变为一元函数的过程

```
add = lambda a, b: a + b
curried_add = lambda a: lambda b: a + b
curried_add(3)(4)
# 7
add2 = curried_add(2)
add2(10)
```

```
# 12
```

1.6. Auto Currying - 自动柯里化

```
from toolz import curry
add = lambda a, b: a + b
curried_add = curry(add)
curried_add(1, 2)
# 3
curried_add(1)(2)
# 3
curried_add(1)
# <function <lambda> at 0x000002088BBD5E18>
```

1.7. Function Composition - 函数组合

接收多个函数作为参数，从右到左，一个函数的输入为另一个函数的输出

```
import math
from functools import reduce
# 组合2个函数
compose = lambda f, g: lambda a: f(g(a))
# 组合多个函数
compose = lambda *funcs: reduce(lambda f, g: lambda *args: f(g(*
args)), funcs)
floor_and_to_string = compose(str, math.floor)
floor_and_to_string(12.12)
# '12'
```

1.8. Purity - 纯函数

输出仅由输入决定，且不产生副作用

```
greet = lambda name: f'hello, {name}'
greet('world')
'hello, world'
```

以下代码不是纯函数

```
# 情况1：函数依赖全局变量
NAME = 'alphardex'
greet = lambda: f'hi, {NAME}'
greet()
# 'hi, alphardex'

# 情况2：函数修改了全局变量
greeting = None
def greet(name):
    global greeting
    greeting = f'hi, {name}'
greet('alphardex')
greeting
# 'hi, alphardex'
```

1.9. Side effects - 副作用

如果函数与外部可变状态进行交互，则它是有副作用的

最典型的例子是创建日期和IO

```
from datetime import datetime
different_every_time = datetime.now()
different_every_time
# datetime.datetime(2019, 4, 20, 17, 30, 24, 824876)
different_every_time = datetime.now()
```

```
different_every_time
# datetime.datetime(2019, 4, 20, 17, 31, 41, 204302)
```

1.10. Idempotent - 幂等性

如果一个函数执行多次皆返回相同的结果，则它是幂等性的

```
abs(abs(abs(10)))
# 10
```

1.11. Point-Free Style - Point-Free 风格

定义函数时，不显式地指出函数所带参数，这种风格通常需要柯里化或者高阶函数

Point-Free风格的函数就像平常的赋值，不使用def或者lambda关键词

```
map_ = lambda func: lambda li: [func(l) for l in li]
add = lambda a: lambda b: a + b
increment_all = map_(add(1))

numbers = [1, 2, 3]
increment_all(numbers)
# [2, 3, 4]
```

1.12. Predicate - 谓词

根据输入返回 True 或 False。常用于filter函数中

filter函数亦可以用列表推导式的if判断实现

```
above_two =  lambda a: a > 2
```

```
li = [1, 2, 3, 4]
[l for l in li if above_two(l)]
# [3, 4]
```

1.13. Contracts - 契约

契约保证了函数或者表达式在运行时的行为。当违反契约时，将抛出一个错误。

```
def contract(input):
    if isinstance(input, int):
        return True
    raise Exception('Contract Violated: expected int -> int')

add_one = lambda num: contract(num) and num + 1
add_one(2)
# 3
add_one('hello')
# Exception Traceback
```

1.14. Functor - 函子

一个实现了map函数的对象，map会遍历对象中的每个值并生成一个新的对象。

Python中最具代表性的函子就是list, 因为它遵守因子的两个准则

在Python中可以用列表推导式来代表map操作

1.14.1. Preserves identity - 一致性

```
li = [1, 2, 3]
[l for l in li] == li
```

```
# True
```

1.14.2. Composable - 组合性

```
li = [1, 2, 3]
compose = lambda f, g: lambda a: f(g(a))
[compose(str, lambda x: x+1)(l) for l in li]
# ['2', '3', '4']
[str(l+1) for l in li]
# ['2', '3', '4']
```

1.15. Referential Transparency - 引用透明性

一个表达式能够被它的值替代而不改变程序的行为成为引用透明

```
greet = lambda: 'hello, world.'
```

1.16. Lazy evaluation - 惰性求值

按需求值机制，只有当需要计算所得值时才会计算

Python中可用生成器实现

```
import random
def rand():
    while True:
        yield random.random()
rand_iter = rand()
next(rand())
# 0.16066473752585098
```

1.17. Monoid - 单位半群

一个对象拥有一个函数用来连接相同类型的对象

数值加法是一个简单的Monoid

```
1 + 1
# 2
```

以上例子中，数值是对象，而+是函数

以下能更清晰地说明它

```
from operator import add
type(1)
# <class 'int'>
add(1, 1)
# 2
```

数值是int类的实例对象，add是实现了加法的函数

与另一个值结合而不会改变它的值必须存在，称为 `identity` 。

加法的identity值为 0:

```
1 + 0
# 1
```

需要满足结合律

```
1 + (2 + 3) == (1 + 2) + 3
# True
```

list的结合也是Monoid

```
[1, 2].extend([3, 4])
```

identity值为空数组

```
[1, 2].extend([])
```

identity与compose函数能够组成monoid

```
identity = lambda a: a
compose = lambda f, g: lambda a: f(g(a))
foo = lambda bar: bar + 1
compose(foo, identity)(1) == compose(identity, foo)(1) == foo(1)
# True
```

1.18. Monad - 单子

拥有 `of` 和 `chain` 函数的对象。 `chain` 很像 `map` ，除了用来铺平嵌套数据。

```
flatten = lambda li: sum(li, [])
of = lambda *args: list(args)
chain = lambda func: lambda li: list(flatten([func(l) for l in l
i]))

[s.split(',') for s in of('cat,dog', 'fish,bird')]
# [['cat', 'dog'], ['fish', 'bird']]

chain(lambda s: s.split(','))(of('cat,dog', 'fish,bird'))
# ['cat', 'dog', 'fish', 'bird']
```

1.19. Comonad - 余单子

拥有 `extract` 与 `extend` 函数的对象。

```
class CoIdentity:
    def __init__(self, v):
        self.val = v
    def extract(self):
        return self.val
    def extend(self, func):
        return CoIdentity(func(self))

CoIdentity(1).extract()
1
from beeprint import pp
pp(CoIdentity(1).extend(lambda x: x.extract() + 1))
# instance(CoIdentity):
#   val: 2
```

1.20. Morphism - 态射

一个变形的函数

1.20.1. Endomorphism - 自同态

输入输出是相同类型的函数

```
uppercase = lambda string: string.upper()
uppercase('hello')
# 'HELLO'

decrement = lambda number: number - 1
decrement(2)
# 1
```

1.20.2. Isomorphism - 同构

不同类型对象的变形，保持结构并且不丢失数据。

例如，一个二维坐标既可以表示为列表 `[2, 3]` ，也可以表示为字典 `{'x': 2, 'y': 3}` 。

```
pair_to_coords = lambda pair: {'x': pair[0], 'y': pair[1]}
coords_to_pair = lambda coords: [coords['x'], coords['y']]
pair_to_coords(coords_to_pair({'x': 1, 'y': 2}))
#{'x': 1, 'y': 2}
```

1.21. Setoid

拥有 `equals` 函数的对象。 `equals` 可以用来和其它对象比较。

Python里的 `==` 就是 `equals` 函数

```
[1, 2] == [1, 2]
# True

[1, 2] == [3, 4]
# False
```

1.22. Semigroup - 半群

拥有 `concat` 函数的对象。 `concat` 可以连接相同类型的两个对象。

Python里列表的 `extend` 就是 `concat` 函数

```
li = [1]
li.extend([2])
li
```

```
# [1, 2]
```

1.23. Foldable

一个拥有 `reduce` 函数的对象。 `reduce` 可以把一种类型的对象转化为另一种类型。

```
from functools import reduce
sum_ = lambda li: reduce(lambda acc, val: acc + val, li, 0)
sum_([1, 2, 3])
6
```

1.24. Type Signatures - 类型签名

通常可以在注释中指出参数与返回值的类型

```
# add :: int -> int -> int
add = lambda x: lambda y: x + y

# increment :: int -> int
increment = lambda x: x + 1
```

如果函数的参数也是函数，那么这个函数需要用括号括起来

```
# call :: (a -> b) -> a -> b
call = lambda func: lambda x: func(x)
```

字符a, b, c, d表明参数可以是任意类型。以下版本的 `map` 的参数func，把一种类型a的数组转化为另一种类型b的数组

```
# map :: (a -> b) -> [a] -> [b]
```

```
map_ = lambda func: lambda li: [func(l) for l in li]
```

1.25. 常用库

- functools
- itertools
- operator
- more-itertools
- toolz

网络爬虫是一种按照一定的规则，自动地抓取网站信息的程序或者脚本。

本文以Python语言为例简要谈谈爬虫是如何编写的。

1.1. 如何爬取网站信息

写爬虫之前，我们必须确保能够爬取目标网站的信息。

不过在此之前必须弄清以下三个问题:

1. 网站是否已经提供了api
2. 网站是静态的还是动态的
3. 网站是否有反爬的对策

1.1.1. 情形1：开放api的网站

一个网站倘若开放了api，那你就可以直接GET到它的json数据。

比如xkcd的about页就提供了api供你下载

```
import requests
requests.get('https://xkcd.com/614/info.0.json').json()
```

那么如何判断一个网站是否开放api呢？有3种方法：

1. 在站内寻找api入口
2. 用搜索引擎搜索“某网站 api”
3. 抓包。有的网站虽然用到了ajax（比如果壳网的瀑布流文章，亦或

是unsplash的瀑布流图片），但是通过抓包还是能够获取XHR里的json数据的，不要傻乎乎地去用selenium，反而会降低效率。

> 怎么抓包：F12 - Network - F5刷新

实际上，app的数据也可以通过抓包来获取。

app抓包

安装fiddler并启动，打开Tools-Options-Connections，将Allow remote computers to connect打上勾并重启fiddler。

命令行上输入ipconfig，查看自己网络的ipv4地址，在手机的网络上设置HTTP代理，端口为8888。

这时虽说能抓到数据，但都是HTTP的，而app的大部分数据都是HTTPS的。

在Options-HTTPS中将Decrypt HTTPS traffic打上勾。

以ios系统为例，在Safari浏览器中输入http://ipv4:8888，下载证书并安装。

这样就能抓到HTTPS数据啦！

1.1.2. 情形2：不开放api的网站

如果此网站是静态页面，那么你就可以解析它的HTML。

解析库强烈推荐parsel，不仅语法和css选择器类似，而且速度也挺快，Scrapy用的就是它。

> 你需要了解一下css选择器的语法（xpath也行），并且学会看网页的审查元素

比如获取konachan的所有原图链接

```
from parsel import Selector
```

```
res = requests.get('https://konachan.com/post')
tree = Selector(text=res.text)
imgs = tree.css('a.directlink::attr(href)').extract()
```

如果此网站是动态页面，先用selenium来渲染JS，再用HTML解析库来解析driver的page_source。

比如获取hitomi.la的数据（这里把chrome设置成了无头模式）

```
from selenium import webdriver
options = webdriver.ChromeOptions()
options.add_argument('--headless')
driver = webdriver.Chrome(options=options)
driver.get('https://hitomi.la/type/gamecg-all-1.html')
tree = Selector(text=driver.page_source)
gallery_content = tree.css('.gallery-content > div')
```

1.1.3. 情形3：反爬的网站

目前的反爬策略常见的有：验证码、登录、封ip等。

验证码：利用打码平台破解（如果硬上的话用opencv或keras训练图）

登录：利用requests的post或者selenium模拟用户进行模拟登陆

封ip：买些代理ip（免费ip一般都不管用），requests中传入proxies参数即可

其他防反爬方法：伪装User-Agent，禁用cookies等

1.2. 如何编写结构化的爬虫

爬虫的结构很简单，无非就是创造出一个tasklist，对tasklist里的每一个task调用crawl函数。

> 大多数网页的url构造都是有规律的，你只需根据它用列表推倒式来构造出tasklist
>
> 对于那些url不变的动态网页，先考虑抓包，不行再用selenium点击下一页

如果追求速度的话，可以考虑用concurrent.futures或者asyncio等库。

```
import requests
from parsel import Selector
from concurrent import futures

domain = 'https://www.doutula.com'

def crawl(url):
    res = requests.get(url)
    tree = Selector(text=res.text)
    imgs = tree.css('img.lazy::attr(data-original)').extract()
    # save the imgs ...


if __name__ == '__main__':
    tasklist = [f'{domain}/article/list/?page={i}' for i in rang
e(1, 551)]
    with futures.ThreadPoolExecutor(50) as executor:
        executor.map(crawl, tasklist)
```

数据存储的话，看需求，存到数据库中的话只需熟悉对应的驱动即可。

常用的数据库驱动有：pymysql(MySQL), pymongo(MongoDB)

1.3. 框架

读到这里，相信你已经对网络爬虫的结构有了个清晰的认识，可以去上手框架了。

looter是本人写的一个轻量级框架，适合中小型项目；比较大型的项目建议用scrapy。

如今，网上的爬虫教程可谓是泛滥成灾了，从urllib开始讲，最后才讲到requests和selenium这类高级库，实际上，根本就不必这么费心地去了解这么多无谓的东西的。只需记住爬虫总共就三大步骤：发起请求——解析数据——存储数据，这样就足以写出最基本的爬虫了。诸如像Scrapy这样的框架，可以说是集成了爬虫的一切，但是新人可能会用的不怎么顺手，看教程可能还会踩各种各样的坑，而且Scrapy本身体积也有点大。因此，本人决定亲手写一个轻量级的爬虫框架————looter，里面集成了调试和爬虫模板这两个核心功能，利用looter，你就能迅速地写出一个高效的爬虫。另外，本项目的函数文档也相当完整，如果有不明白的地方可以自行阅读源码（一般都是按Ctrl+左键或者F12）。

1.1. 安装

```
$ pip install looter
```

仅支持Python3.6及以上版本。

1.2. 快速开始

让我们先来撸一个非常简单的图片爬虫：首先，用shell获取网站

```
$ looter shell https://konachan.com/post
```

然后用1行代码将图片的url提取出来

```
>>> imgs = tree.css('a.directlink::attr(href)').extract()
```

或者用另一种方式提取

```
>>> imgs = links(res, pattern=r'.*/(jpeg|image)/.*')
```

将url保存到本地

```
>>> Path('konachan.txt').write_text('\n'.join(imgs))
```

可以通过wget等下载工具将图片下载下来

```
$ wget -i konachan.txt
```

如果想要看更多的爬虫例子，猛戳这里

1.3. 工作流

如果你想迅速撸出一个爬虫，那么你可以用looter提供的模板来自动生成一个

```
$ looter genspider <name> [--async]
```

async是一个备用的选项，它使得生成的爬虫核心用asyncio而非线程池。

在生成的模板中，你可以自定义domain和tasklist这两个变量。

什么是tasklist？实际上它就是你想要抓取的页面的所有链接。

以konachan.com为例，你可以使用列表推导式来创建自己的tasklist：

```
domain = 'https://konachan.com'
tasklist = [f'{domain}/post?page={i}' for i in range(1, 9777)]
```

然后你就要定制你的crawl函数，这是爬虫的核心部分。

```
def crawl(url):
    tree = lt.fetch(url)
    items = tree.css('ul li')
    for item in items:
        data = {}
        # data[...] = item.css(...)
        pprint(data)
```

在大多数情况下，你所要抓取的内容是一个列表（也就是HTML中的ul或ol标签），可以用css选择器将它们保存为items变量。

然后，你只需使用for循环来迭代它们，并抽取你想要的数据，将它们存储到dict中。

注意：目前looter使用了parsel来解析网页，和Scrapy的解析工具一样。如果想用以前的cssselect的话，把fetch的use_parsel设为False就可以了。

但是，在你写完这个爬虫之前，最好用looter提供的shell来调试一下你的css代码是否正确。（目前已集成ptpython，一个支持自动补全的REPL）

```
>>> items = tree.css('ul li')
>>> item = items[0]
>>> item.css(anything you want to crawl)
# 注意代码的输出是否正确！
```

调试完成后，你的爬虫自然也就完成了。怎么样，是不是很简单:)

1.4. 函数

looter为用户提供了一些比较实用的函数。

1.4.1. view

在爬取页面前，你最好确认一下页面的渲染是否是你想要的

```
>>> view(url)
```

1.4.2. links

获取网页的所有链接

```
>>> links(res)                    # 获取所有链接
>>> links(res, search='...')      # 查找指定链接
>>> links(res, pattern=r'...')    # 正则查找链接
```

1.4.3. save_as_json

将所得结果保存为json文件，支持按键值排序（sort_by)和去重(no_duplicate)

```
>>> total = [...]
>>> save_as_json(total, sort_by='key', no_duplicate=True)
```

如果想保存为别的格式（csv、xls等），用pandas转化即可

```
>>> import pandas as pd
>>> data = pd.read_json('xxx.json')
>>> data.to_csv()
```

1.5. 套路总结

1. 通过抓包，确认网站是否开放了api，如果有，直接抓取api；如果没有，进入下一步
2. 确认网站是静态的还是动态的（有无JS加载，是否需要登录等），方法有：肉眼观察、抓包、looter的view函数
3. 若网站是静态网页，直接用looter genspider生成爬虫模板，再配合looter shell写出爬虫即可
4. 若网站是动态网页，先抓包试试，尝试获取所有ajax生成的api链接；如果没有api，则进入下一步
5. 有的网站并不会直接暴露ajax的api链接，这时就需要你自行根据规律，构造出api链接
6. 如果上一步无法成功，那么就只好用requestium来渲染JS，抓取页面
7. 至于模拟登录、代理IP、验证码、分布式等问题，由于范围太广，请自行解决
8. 如果你的爬虫项目被要求用Scrapy，那么你也可以将looter的解析代码无痛地复制到Scrapy上（毕竟都用了parsel）

掌握了以上的套路，再难爬的网站也难不倒你。

flask是一个Python语言开发的web“微框架”，和django不同的是，它既没有数据库、也没有表单验证等工具它本身仅仅提供了一个WSGI的桥梁，其他东西统统靠你来定制，具有很大的灵活性

1.1. 脚手架

为了迅速搭建一个像样的flask网站，我们可以使用脚手架

之前在Github上看到cookiecutter-flask，是个不错的选择，但是新手可能会看不懂里面代码是如何封装的

于是本人做出了一个更user-friendly的脚手架——cookiecutter-flask-bootstrap

这个脚手架的功能大致和上个脚手架差不多，不过更加轻量化，而且结构更加清晰明了，best practice也基本都做到了，希望大家用的开心:d

最后还要感谢李辉大大的狼书，给了我很大的帮助

1.2. 路由

路由是将特定的业务代码（即视图函数）绑定到某个url上，以实现某个功能

1.2.1. 注册

flask中使用装饰器来注册路由

```
@app.route('/')
def index():
    return 'Hello world.'
```

可以为路由传递变量，变量左边可以带转换器用来转换变量的类型

```
@app.route('/user/<string:username>')
def user_profile(username):
    return f'User {username}'
```

常用的转换器有6种：string, int, float, path, any, uuid

比较特殊的是any，格式如下（var变量只接受a, b其中的任意一值）

```
@app.route('/<any(a, b):var>/')
```

如果想通过路由直接访问文件呢？用path转换器和send_from_directory就行了

```
@app.route('/uploads/<path:filename>')
def get_image(filename):
    return send_from_directory(current_app.config['UPLOAD_PATH']
, filename)
```

1.2.2. 构造url

使用url_for函数可以反向构建访问路由的url

```
url_for('index')  # '/'
url_for('user_profile', username='alphardex')  # '/user/alphardex'
url_for('index', _external=True)  # 'http://localhost:5000/' 绝对路径
```

1.2.3. HTTP

路由默认支持GET方法，如果需要POST方法则需在route的methods中传入

HTTP methods的用处如下：

- GET：获取资源
- POST：创建资源
- PUT：更新资源
- DELETE：删除资源

```
@app.route('/login', methods=['GET', 'POST'])
```

如果想更改HTTP请求头内容，那就要用到make_response

比如制作网站的RSS，就需要把响应的mimetype设置为application/xml

```
@app.route('/rss')
def rss():
    articles = Article.query.order_by(Article.date.desc).limit(10)
    rss = render_template('rss.xml', articles=articles)
    response = make_response(rss)
    response.mimetype = 'application/xml'
```

```
    return response
```

1.3. 模板

渲染一个模板，简言之就是通过上下文变量来生成HTML

```
from flask import render_template

@app.route('/')
def index():
    greetings = 'Hello world.'
    return render_template('index.html', greetings=greetings)
```

render_template中第一个参数是要渲染的模板文件名，**其余参数则是上下文变量**

```
<h1>{{ greetings }}</h1>
```

通过mustache语法将上下文变量传入模板并渲染，同时也支持if、for等控制流语句语法，更高级的有过滤器、模板继承、宏等

过滤器的添加格式如下所示

```
{{ var|filter }}
```

提几个常用的过滤器：

- safe: 避免HTML的自动转义，本质上是个Markup对象
- length: 获取变量长度
- default: 为变量设置默认值
- trim: 去除变量前后的空格
- tojson: 将变量转化为json

- truncate: 截断字符串，常用于显示文章摘要

网站的静态文件放在static文件夹中，通过反向构造url访问

```
url_for('static', filename='style.css')
```

1.4. 上下文全局变量

- current_app：指向处理请求的app实例
- g：global的简写，以object的方式存储信息（比如用户登录后的用户对象 g.user）
- request：以dict形式存储HTTP请求相关变量
- session：以dict的方式存储会话信息（比如用户登录后的用户id session['user_id']）

以下是request所封装的几个最常用的参数，全部参数请点这里

```
request.args          # GET请求的查询字符串
request.cookies       # cookies信息
request.files         # 请求上传的文件
request.form          # POST请求的表单数据
request.headers       # 请求头
request.method        # 请求类型
request.path          # 请求路径
request.referrer      # 请求发源地址
request.remote_addr # 用户的ip地址
request.get_json()  # 获取api的json数据
```

1.5. 工具函数

- abort：放弃请求
- flash：闪现信息，可以附带类别
- jsonify：将数据序列化为json，常用于设计restful api

- redirect：重定向

1.6. 工厂模式

工厂模式使得app在创建之时能同时完成以下步骤：加载配置，初始化扩展，注册蓝本，注册shell上下文，以及注册错误处理函数等

```
def create_app(config_name=None):
    if config_name is None:
        config_name = os.getenv('FLASK_CONFIG', 'development')

    app = Flask(__name__)
    app.config.from_object(config[config_name])

    register_blueprints(app)
    register_extensions(app)
    register_shell_context(app)
    register_errors(app)

    return app

def register_extensions(app):
    debugtoolbar.init_app(app)
    ...

def register_blueprints(app):
    app.register_blueprint(main_bp)
    ...

def register_shell_context(app):
    @app.shell_context_processor
    def make_shell_context():
        return {'db': db, ...}

def register_errors(app):
```

```
    @app.errorhandler(400)
    def bad_request(e):
        return render_template('errors/400.html'), 400
    ...
```

1.7. 蓝本

用来实现模块化编程

例如一个app（名字叫flasky）通常会有以下的文件夹结构

```
├─flasky
│  ├─blueprints
│  ├─static
│  │  └─css
│  └─templates
│      ├─auth
│      ├─errors
│      ├─main
│      └─user
└─tests
```

其中blueprints就是蓝本文件夹，里面存放着和templates对应的4个蓝本

```
├─blueprints
│      auth.py
│      main.py
│      user.py
│      __init__.py
```

这4个蓝本中__init__.py负责Python的包结构，其余三个则是app的3个功能模块：认证、主页、用户

以auth.py为例

```
from flask import Blueprint, ...
...(此处省略了一堆import)

bp = Blueprint('auth', __name__)

@bp.route('/login', methods=['GET', 'POST'])
def login():
    ...

@bp.route('/register', methods=['GET', 'POST'])
def register():
    ...

@bp.route('/logout')
def log_out():
    ...
```

蓝本也可以注册路由，如果反向构造url的话就得加上蓝本名，比如url_for('auth.login')

蓝本的注册应在工厂函数中执行，并且每个蓝本可以通过url_prefix来给url添加前缀

1.8. 常用插件

- flask-admin：提供admin管理后台
- flask-avatars：生成用户头像
- flask-ckeditor：集成富文本编辑器ekeditor
- flask-cors：提供跨域支持
- flask-dropzone：集成文件上传插件dropzone
- flask-login：处理用户登陆认证逻辑
- flask-mail：邮箱服务
- flask-migrate：提供数据库迁移支持

- flask-moment：提供时间规范化支持
- flask-mongoengine：集成mongoengine——面向mongodb的ORM
- flask-restful：RESTful API支持
- flask-socketio：集成socketio，常用于编写聊天室
- flask-sqlalchemy：集成sqlalchemy——面向标准SQL的ORM
- flask-weasyprint：提供PDF打印功能
- flask-whooshee：集成whooshee——全文搜索引擎
- flask-wtf：集成wtforms——表单支持
- bootstrap-flask：集成bootstrap，并提供一些有用的宏
- faker：能生成假数据，用于测试

1.9. 高级玩法

1.9.1. 强制响应格式

API返回的一般都是json，故在每个视图函数中调用jsonify将dict序列化为json

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/')
def index():
    return jsonify({'message': 'Hello World!'})

@app.route('/foo')
def foo():
    return jsonify({'message': 'Hello foo!'})
```

但其实没必要这么做，因为**flask的Response是可以定制的**

flask的app实例提供了response_class属性，默认是Response

继续查照定义，发现Response其实继承了werkzeug里的BaseResponse

通过查阅BaseResponse，我们可以重载Response的force_type类方法，将类型为dict的response直接jsonify，并且无需在每个视图函数中都显式调用jsonify函数了

```
from flask import Flask, jsonify, Response

class JSONResponse(Response):
    @classmethod
    def force_type(cls, response, environ=None):
        if isinstance(response, dict):
            response = jsonify(response)
        return super().force_type(response, environ)

app = Flask(__name__)
app.response_class = JSONResponse

@app.route('/')
def index():
    return {'message': 'Hello World!'}

@app.route('/foo')
def foo():
    return {'message': 'Hello foo!'}
```

当然，你也可以类似地强制响应格式为xml，RSSHub就是这么实现的

```
from flask import Flask, Response

class XMLResponse(Response):
    def __init__(self, response, **kwargs):
        if 'mimetype' not in kwargs and 'contenttype' not in kwa
rgs:
            if response.startswith('<?xml'):
```

```
                kwargs['mimetype'] = 'application/xml'
        return super().__init__(response, **kwargs)

app = Flask(__name__)
app.response_class = XMLResponse

@bp.route('/feed')
def rss_feed():
    from rsshub.spiders.feed import ctx
    return render_template('main/atom.xml', ctx())
```

1.9.2. 全局模板函数

有的时候你希望把某种逻辑抽象成一个函数，使得多个页面能共用，那么就要定义全局模板函数了

有的网站的排序功能是这么实现的：通过在url的查询字符串中追加sort（要排序的字段）和order（升序还是降序）参数，在视图函数中获取这2个参数并进行相应的排序处理

要实现这个功能，可以编写2个全局函数：1个负责修改url的查询字符串，另一个负责处理给查询排序

```
@bp.app_template_global()
def modify_querystring(**new_values):
    args = request.args.copy()
    for k, v in new_values.items():
        args[k] = v
    return f'{request.path}?{url_encode(args)}'

@bp.app_template_global()
def get_article_query():
    sort_key = request.args.get('s', 'date')
    order = request.args.get('o', 'desc')
    article_query = Article.query
```

```
    if sort_key:
        if order == 'asc':
            article_query = article_query.order_by(db.asc(sort_k
ey))
        else:
            article_query = article_query.order_by(db.desc(sort_
key))
    return article_query
```

此外还要定义一个模板宏，这样当用户点击页面上的排序链接时，就能够进行排序了

```
{% macro sort_column(sort_key) %}
    {% set order = request.args.get('o', 'asc') %}
    <a href="{% if order == 'desc' %}{{ modify_querystring(s=sor
t_key, o='asc') }}{% else %}{{ modify_querystring(s=sort_key, o=
'desc') }}{% endif %}">
    {{ caller() }}
    </a>
{% endmacro %}
```

```
<th>{% call sort_column('date') %}<div>日期</div>{% endcall %}</
th>
```

1.9.3. 自定义路由转换器

我们都知道flask的路由映射是存储在app.url_map中的，因此查阅官方文档有关url_map的部分，就能轻松实现

```
from flask import Flask
from urllib.parse import unquote
from werkzeug.routing import BaseConverter
```

```
class ListConverter(BaseConverter):
    def __init__(self, url_map, separator='+'):
        super().__init__(url_map)
        self.separator = unquote(separator)

    def to_python(self, value): # 把路径转换成一个Python对象，比如['python', 'javascript', 'sql']
        return value.split(self.separator)

    def to_url(self, values): # 把参数转换成符合URL的形式，比如/python+javascript+sql/
        return self.separator.join(BaseConverter.to_url(self, value) for value in values)

app = Flask(__name__)
app.url_map.converters['list'] = ListConverter

@app.route('/list1/<list:var>/')
def list1(var):
    return f'Separator: + {var}'


@app.route('/list2/<list(separator=u"|"):var>/')
def list2(var):
    return f'Separator: | {var}'
```

官方文档仅仅用了逗号分隔符，而在这里我们通过添加了separator属性来实现了自定义分隔符 访问如下链接体验下效果

```
http://localhost:5000/list1/python+javascript+sql
http://localhost:5000/list2/python|javascript|sql
```

地址：http://techattic.herokuapp.com

源码：https://github.com/alphardex/techattic

这是我的第一个用flask和looter搭建的项目，爬取了约7家博客平台的大部分技术文章，经过笔者的一番筛选把那些高质量的文章给挑了出来。

本站点支持排序和搜索，想要技术干货的万万别错过哦。

RSSHub是一个RSS生成器，其实它的本质就是爬虫的集合。

由于爬虫是Python的强项，因此笔者就尝试用Python的flask框架来实现它。

项目地址：https://github.com/alphardex/RSSHub-python

1.1. RSS入门

注：以下把RSS信息统称为信息源。

RSS（简易信息聚合）是目前常见的一种信息订阅方式。用户只需有一个RSS阅读器，就能订阅各种各样的信息源了。

1.2. RSSHub

当你有了RSS阅读器以后，就得寻找自己的信息源了。

笔者逛了逛Github，发现RSSHub真是个好东西，里面有各种社区贡献的信息源，但笔者一看全是JS写的爬虫脚本。

由于笔者JS的功力没Python深厚，因此就开始考虑用Python来实现一个RSSHub，这样一来爬虫脚本就也能用Python写了XD。

1.3. RSS获取途径

想获取RSS源其实很容易，只要按照以下顺序就行：

1. 找网站自带的RSS（通常是“订阅”、“RSS”字样等）
2. 在RSSHub上按Ctrl+F搜索

3. 在RSS阅读器（例如inoreader）上搜索RSS源
4. 利用搜索引擎搜索RSS源
5. 尝试在想订阅的网站url后面追加rss和atom.xml
6. 自己写一个RSS源并给RSSHub提PR

1.4. 技术RSS源

用以上的办法，笔者挑选出了一些技术RSS源：

- 博客园精华区
- infoq推荐内容
- segmentfault
- 开发者头条
- 掘金后端
- 开源中国资讯

最近把《django by example》的项目差不多都撸完了，是时候该写个小结了。

个人体会：django的api可以说是很多很全，这辈子都不可能全记住的。

其实，懂得速查文档就没有什么大问题。

大量的封装虽然牺牲了一定的灵活性，但大大提高了开发效率，或许这就是django的哲学吧。

1.1. 准备

首先，你可以用django-admin来生成你的项目。

不过笔者更建议用一个脚手架来快速生成一个启动模板。

1.2. 核心工作流

Django的核心是MVC，更准确来说是MVT（Model-View-Template）

首先创建app，并在settings中的INSTALLED_APPS添加其配置，在全局urls中通过include引入app的所有url

接下来才是最关键的3步：

1. 在models.py中定义好数据模型并迁移它们
2. 在views.py中编写视图函数，并在urls.py中为其创建相应的映射
3. 在templates文件夹中编写要渲染的模板HTML

数据模型的定义其实就是定义各种各样的字段，还有个Meta类可以定义一些元数据（比如字段的排序等），此外你也可以为模型封装一些method来简化视图的编写。

视图函数的编写主要涉及以下方面：模型的CRUD、表单的处理、模板的上下文渲染（必须熟悉HTTP）

视图函数分2种：函数和类（即FBV和CBV）。两者各有利弊，根据需求自行权衡。目前来说后者在django中比较流行。

给视图添加额外功能：FBV用装饰器，CBV用Mixin。

模板的编写主要涉及：上下文的渲染、if、for、with语句、过滤器、继承等

1.3. 后台管理

其实在定义完数据模型后就已经可以从admin开始玩起了（初次进入需要创建一个超级用户）。

把数据模型通通注册上去，以便进行CRUD。

1.4. 其他杂项

- 数据模型的Meta元信息（ordering、abstract等）
- 利用Manager来简化查询
- 熟悉QuerySet的各种接口和查询方法（比如Q、select_related等）
- 懂得如何优化查询
- 利用Library实现自定义标签
- 创建sitemaps来优化SEO
- 创建feeds来提供RSS订阅功能
- 利用Postgresql来实现全文搜索
- 利用Ajax来优化用户体验
- 利用contenttypes来追踪models（比如实现用户活动流）
- 利用signals来反规范化计数

- 利用Redis实现各种功能（缓存、计数、排行榜等）
- 利用sessions在服务端存储数据（比如购物车）
- 利用celery实现异步任务（比如邮件发送）
- 定制admin
- 数据模型的继承（abstract、multi-table、proxy）
- 自定义数据模型字段
- 用Mixin为类视图添加额外功能
- 用formset处理多张表单
- 缓存的使用
- 用drf创建RESTful API
- 自定义中间件
- 部署上线

如果对以上内容都了然于胸的话，可以说是掌握django了。

当然，以上的所有内容用flask也都可以实现，只是方式不同罢了:)

1.5. 常用第三方库

- django-debug-toolbar：提供DEBUG信息，必备
- django-crispy-forms：美化表单
- django-extensions：各种扩展（shell、server等）
- django-taggit：提供简单的打标签功能
- django-braces：为类视图提供一系列Mixin
- django-embed-video：为页面嵌入视频
- djangorestframework：大名鼎鼎的drf，为django提供RESTful API支持
- django-xadmin：一个更强大的admin后台
- django-ckeditor：为表单提供富文本编辑器
- markdown：提供Markdown渲染支持
- pillow：图像处理
- sorl-thumbnail：缩略图生成
- redis：redis数据库的接口

- celery：分布式任务队列，用来任务调度
- flower：监控celery
- weasyprint：用HTML生成PDF文件

Python和JavaScript在笔者看来是很相似的语言，本文归纳了JavaScript的各种tricks，相对于之前的Python版。

两篇文章都读完，有没有发现它们的目录结构是一个样的呢XD

1. 基本

1.1. 模板字符串

```
let name = 'alphardex'
`Ore wa ${name} desu, ${4 * 6} sai, gakusei desu.`
// "Ore wa  desu, 24 sai, gakusei desu."
```

1.2. 三元运算符

```
// if(condition){
//     fuck
// } else {
//     shit
// }
(condition)? fuck: shit
```

1.3. 字符串的拼接，反转与分割

```
let letters = ['h', 'e', 'l', 'l', 'o']
letters.join('')
// "hello"
letters.reverse()
// ["o", "l", "l", "e", "h"]
let name = 'nameless god'
name.split(' ')
// ['nameless', 'god']
```

1.4. 判断元素的存在性

```
'fuck you'.includes('fuck')
// true
['bitch', 'whore'].includes('slut')
// false
'company' in {'title': 'SAO III', 'company': 'A1 Pictures'}
// true
```

2. 函数

2.1. 箭头函数

函数的简化写法，配合map、filter、sort等实现函数式编程

```
// function foo(parameters){
//     return expression
// }
let foo = (parameters) => expression
```

2.1.1. map - 映射

```
let numbers = [1, 2, 3, 4, 5];
numbers.map(e=>e ** 2)
// [1, 4, 9, 16, 25]
```

2.1.2. filter - 过滤

```
let values = [null, undefined, NaN, 0, '', true, 'alphardex', 666
]
values.filter(e=>e)
// [true, "alphardex", 666]
```

2.1.3. sort - 排序

```
let numbers = [4, 2, 5, 1, 3];
numbers.sort((a, b)=>b-a)
// [5, 4, 3, 2, 1]
```

2.1.4. 其他骚操作

求1到100的和

```
[...Array(101).keys()].reduce((a, b)=>a+b)
// 5050
// 或者用lodash实现，写法简直跟Python一模一样
// _.sum(_.range(101))
```

扁平化数组

```
const flatten = (arr, depth=1) => arr.reduce((a, v)=>a.concat(de
pth>1 && Array.isArray(v)?flatten(v, depth-1):v), [])
let arr = [1, [2, 3, ['a', 'b', 4], 5], 6]
flatten(arr, 2)
// [1, 2, 3, "a", "b", 4, 5, 6]
// 或者用ES10新增的flat
// arr.flat(2)
```

2.2. 扩展运算符

2.2.1. 数据结构的合并

```
let arr1 = ['a', 'b']
let arr2 = [1, 2]
[...arr1, ...arr2]
// ['a', 'b', 1, 2]
let obj1 = {'name': 'alphardex'}
let obj2 = {'age': 24}
{...obj1, ...obj2}
// {name: 'alphardex', age: 24}
```

2.2.2. 函数参数的打包

```
let foo = (...args) => console.log(args)
foo(1, 2)
// [1, 2]
```

3. 数据结构

3.1. 数组

3.1.1. 推导式

由于推导式暂时不在标准规范内，因此用高阶函数配合箭头函数代替

```
let even = [...Array(10).keys()].filter(e=>e%2!==1)
even
// [0, 2, 4, 6, 8]
```

3.1.2. 同时迭代元素与其索引

相当于Python的enumerate

```
let li = ['a', 'b', 'c']
li.map((e, i)=>`${i+1}. ${e}`)
// ["1. a", "2. b", "3. c"]
```

3.1.3. 元素的追加与连接

push在末尾追加元素，concat在末尾连接元素

```
let li = [1, 2, 3]
li.push([4, 5])
li
// [1, 2, 3, [4, 5]]
li.concat([4, 5])
li
// [1, 2, 3, [4, 5], 4, 5]
```

3.1.4. 测试是否整体/部分满足条件

every测试所有元素是否都满足于某条件，some则是测试部分元素是否满足于某条件

```
[1, 2, 3, 4, 5].every(e=>e<20)
// true
[1, 3, 4, 5].some(e=>e%2===0)
// true
```

3.1.5. 同时迭代2个以上的数组

相当于Python的zip

```
let subjects = ['nino', 'miku', 'itsuki']
let predicates = ['saikou', 'ore no yome', 'is sky']
```

```
subjects.map((e,i)=>`${e} ${predicates[i]}`)
// ["nino saikou", "miku ore no yome", "itsuki is sky"]
```

3.1.6. 去重

利用集合的互异性，同时此法还保留了原先的顺序

```
let li = [3, 1, 2, 1, 3, 4, 5, 6]
[...new Set(li)]
// [3, 1, 2, 4, 5, 6]
```

3.1.7. 解构赋值

最典型的例子就是2数交换

```
let [a, b] = [b, a]
```

用rest运算符可以获取剩余的元素

```
let [first, ...rest] = [1, 2, 3, 4]
first
// 1
rest
// [2, 3, 4]
```

3.2. 对象

3.2.1. 遍历

```
let obj = {name: "alphardex", age: 24}
Object.keys(obj)
```

```
// ["name", "age"]
Object.values(obj)
// ["alphardex", 24]
Object.entries(obj).map(([key, value])=>`${key}: ${value}`)
// ["name: alphardex", "age: 24"]
```

3.2.2. 排序

```
let data = [{'rank': 2, 'author': 'beta'}, {'rank': 1, 'author': 'alpha'}]
data.sort((a, b)=>a.rank - b.rank)
// [{'rank': 1, 'author': 'alpha'}, {'rank': 2, 'author': 'beta'}]
```

3.2.3. 反转

```
let obj = {name: 'alphardex', age: 24}
Object.fromEntries(Object.entries(obj).map(([key, value])=>[value, key]))
// {24: "age", alphardex: "name"}
// 或者用lodash实现
// _.invert(obj)
```

4. 语言专属特性

待整理