Python. Подходы, приемы, интересные факты.

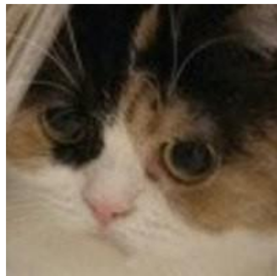
Думбай А.Д., гр. 617

ММП ВМК МГУ

30 сентября 2019 г.

Зачем?

- Ноутбуки — не панацея
- Иногда нужно катить в прод
- Лучше пайплайн — легче эксперименты



Содержание

1 Прикладная часть
- Принципы и подходы

2 Дополнительная часть

Игры памяти

- Возвращение памяти (2 vs 3)
- Пустые контейнеры

Type	sys.getsizeof()
dict	240
list	64
tuple	48
set	224
str	49

- __slots__

Code

```
class Foo:
    __slots__ = ('a', )
    ...
```

Структуры данных

- Список — не список!
- dict — хэш-таблица
- set — хэш-таблица (!)
- https://wiki.python.org/moin/TimeComplexity

Пример

- Задача — случайное семплирование из списка
- Элемент, который выдается — должен быть удален
- Хранится в list, высоканагруженное
- Как удалить?

Пример

- Задача — случайное семплирование из списка
- Элемент, который выдается — должен быть удален
- Хранится в list, высоканагруженное
- Как удалить?

Code

```
...
lst[i], lst[sz - 1] = lst[sz - 1], lst[i]
sz -= 1
...
```

Пример 2

- Задача — использование dict как хранилище текущих сессий

Code

```
d[id_session] = get_info(id_session)
...
```

Какие вообще могут возникнуть проблемы с таким подходом?

Модуль collections

- defaultdict

Code

```
id_to_samples = defaultdict(list)
id_to_samples[0].append(1)
id_to_name = defaultdict(lambda: 'Ivanov Sergey')
id_to_name[0]
# 'Ivanov Sergey'
```

- namedtuple

Code

```
Cat = namedtuple('Tree', ['color', 'age'])
barsik = Cat('black', 10)
```

- Counter
- OrderedDict

Simple is better than complex

- >20 миллионов поисковых запросов по документам
- Уникальных среди всех столбцов после последнего — миллион
- csv > pandas

Threading

- Только один поток исполняется в один момент времени на процессоре(!)
- Накладные расходы на переключение процессов => медленнее
- Зачем можно применять:
  - Запросы в виде интерфейса
  - Не CPU-нагруженные команды (e.g. работа с файлами)
  - Вызовы других процессов
- multiprocessing, asyncio, etc

GIL

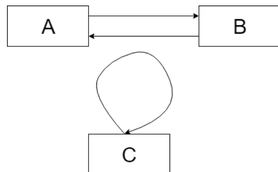
- Только один поток исполняется в один момент времени на процессоре(!)
- Python < 3.2 — переключение зависит от «тиков»
- Python >= 3.2 — от секунд. Тык
- Переключение:
  - время
  - I \O операции
- 3.7 -> time.thread_time()

threading vs multiprocessing

- Накладные расходы на переключение процессов => медленнее
- Зачем можно применять:
  - Запросы в виде интерфейса
  - Не CPU-нагруженные команды (e.g. работа с файлами)
  - Вызовы других процессов
- multiprocessing, joblib, asyncio, etc

Garbage Collector

- Счетчик ссылок
- del x — уменьшает счетчик ссылок
- Есть три поколения. Элемент выживает - переходит в следующее поколение.
- В каждом поколении есть счетчик аллокаций и деаллокаций. Когда первый минус второй превышает некоторый порог - вызывается сборщик.
- Циклы?



- Затратно по памяти — постоянная ручная чистка вредна!

Конструкции

for else

```
for i in some_iterator:
    # check & break
else:
    print('Bad. :(')
```

try except else finally

```
try:
    do_something()
except RuntimeError as e:
    fix_something()
else:
    log_something()
finally:
    do_something_another()
```

Декораторы

Декоратор

```
def time_decorator(func):
    def g(*args, **kwargs):
        t_start = time.time()
        res = func(*args, **kwargs)
        t_work = time.time() - t_start
        logger.info(f'{t_work}')
        return res
    return g
```

Декораторы

Применение

```
@time_decorator
def some_func(x):
    do_things(x)

# equal to
def some_func(x):
    do_things(x)

some_func = time_decorator(some_func)
```

hints

Замыкание

```
def count_call(func):
    count = 0
    def g(*args, **kwargs):
        nonlocal count
        count += 1
        print(count)
        return func(*args, **kwargs)
    return g
```

Декоратор с параметрами

Отсутсвует. Но можно сделать «фабрику» декораторов!

Декоратор с параметром

```
def adder(x=1):
    def decorator(func):
        def g(*args, **kwargs):
            return func(*args, **kwargs) + x
        return g
    return decorator
```

Что выдает help?

Help

```python
def decor(func):
    def g(*args, **kwargs):
        return func(*args, **kwargs)
    return g

@decor
def adder(x: int, y: int) -> int:
    """
    Add two arguments for glory of Odin.
    """
    return x + y

help(adder)
```

Как решить

```
Help on function g in module __main__:

g(*args, **kwargs)
```

Мы встроим вам декоратор в декоратор

```python
from functools import wraps

def decor(func):
    @wraps(func)
    def g(*args, **kwargs):
        return func(*args, **kwargs)
    return g
...
```

Стандартные ошибки\просто факты

- lst[:] — копирует
- ... — ellipsis
- d[1, 2] == d[(1,2)]
- np.unique работает через сортировку
- LabelEncoder из sklearn — сортировка + бинпоиск
- Исключения замедляют

ООП

- Полиморфизм
- Наследование (интерфейсы меньше используются)
- Инкапсуляция?
  - @property
  - __getattr__
  - __getattribute__

ООП

- Полиморфизм
- Наследование (интерфейсы меньше используются)
- Инкапсуляция?
  - @property
  - __getattr__
  - __getattribute__

Nope

```
object.__getattribute__(s, attr)
```

- SOLID, GRASP, GoF

Метаклассы

- type — не просто type

type

```
def bar(self):
    print('bar')
dynamic_class = type('Foo',
                     tuple(),
                     {
                         'bar': bar
                     })
sample = dynamic_class()
sample.bar()
# bar
```

Тестирование

Зачем юниттестирование?

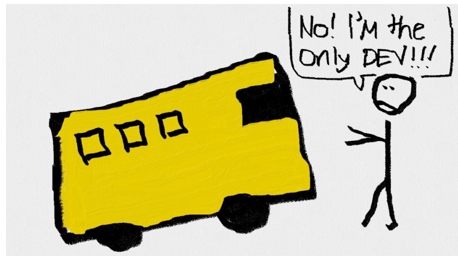
- Код от результата
- Проще при изменении кода увидеть ошибки
- Проще поддерживать и понимать, как работать (сценарии)
- Вариантов много:
  - unittest
  - pytest
  - doctest
  - ...

Моки, патчи

```
@patch('some_lib.process')
@patch('os.path.isfile', side_effect=[True, False])
def test_something(self, is_file_mock, process_mock):
        tested_object = ImportantObject(some_args)
        tested_object.read_state('some.data', 'another.data')
        self.assertTrue(is_file_mock.call_count == 6)
        self.assertTrue(process_mock.call_count == 1)
```

Чистота кода

- PEP8, auto pep8\среды типа pycharm
- Итог задачи = Результат + затраты + **Тех.долг**
- Bus-фактор. Как уменьшить (или увеличить)?
    - Дублирование ролей или областей ответственности
    - Код ревью
    - Тесты
    - Документация



Чистота кода

- В ноутбуках эксперименты, логика — в .py
- %% autoreload, importlib
- Тесты — не только для тестов.
- logging — очень полезная штука.

Логи

```
import logging

F_STR = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
logging.basicConfig(filename="sample.log",
                    format=F_STR,
                    level=logging.INFO)

logging.info('Info message')
logging.error('Error Message')
```

Содержание

1 Прикладная часть
- Принципы и подходы

2 Дополнительная часть

Проверка типов

Что хочется

```
@check(list, int)
def list_multiply(lst, num):
    return type(lst)([x * num for x in lst])
```

- *args
- **kwargs
- x, y, *, z

Проверка типов

Что хочется

```
@check(list, int)
def list_multiply(lst, num):
    return type(lst)([x * num for x in lst])
```

- *args **kwargs
- x, y, *, z

Модуль inspect!

Как развернуть аргументы?

```
def to_args(f, *args, **kwargs):
    signature_f = inspect.signature(f)
    bind_arguments = signature_f.bind(*args, **kwargs)
    return bind_arguments.arguments.values()
```

Уменьшение dict

- del d[key] — помещает вместо ключа специальную метку
- При подсчете ресайза они учитываются для переполнения
- На разеных платформах разные поведения!
- d[1] = 1
- 21, 42

Модуль inspect!

Как развернуть аргументы?

```
def to_args(f, *args, **kwargs):
    signature_f = inspect.signature(f)
    bind_arguments = signature_f.bind(*args, **kwargs)
    return bind_arguments.arguments.values()
```

Полезные библиотеки

- glob

Пример

```
import glob

glob.glob('./*/*.py')

# ['./dir_1/1.py', './dir_1/2.py', './dir_2/3.py']
```

- pprint
- shutil
- difflib

Синглтон

itemize

Декоратор

Декоратор, возвращающий класс

Наследование и реализация __new__

(+) Метакласс реализующий метод __call__

Пример

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*
        return cls._instances[cls]
```

(+) Отдельный модуль