PiterPy 2022

Функциональное программирование в Python - ежедневные рецепты

Arseny Sapelkin
KasperskyOS Tooling Development

KasperskyOS

Что будет в докладе

- Очень кратко о теории
- Применяем ФП принципы к Python
- Смотрим на примеры
- Делаем выводы

Чего *не будет* в докладе

- Сложной теории
- Громких утверждений

Что такое функциональное программирование?

- программирование через создание и композицию функций в их математическом понимании.

Основные концепции ФП

- Чистые функции
- Функции высшего порядка и функции первого класса
- Ленивые вычисления
- Рекурсия как основной способ имплементации алгоритмов
- Особый акцент на обработке последовательностей
- Декларативное программирование
- Думаем на уровне типов
- Иммутабельность

- **Чистые функции**
- Функции высшего порядка и функции первого класса
- Ленивые вычисления
- Рекурсия как основной способ имплементации алгоритмов
- Особый акцент на обработке последовательностей
- Декларативное программирование
- Думаем на уровне типов
- Иммутабельность

Чистые функции

- детерминированные функции, не имеющие побочных эффектов

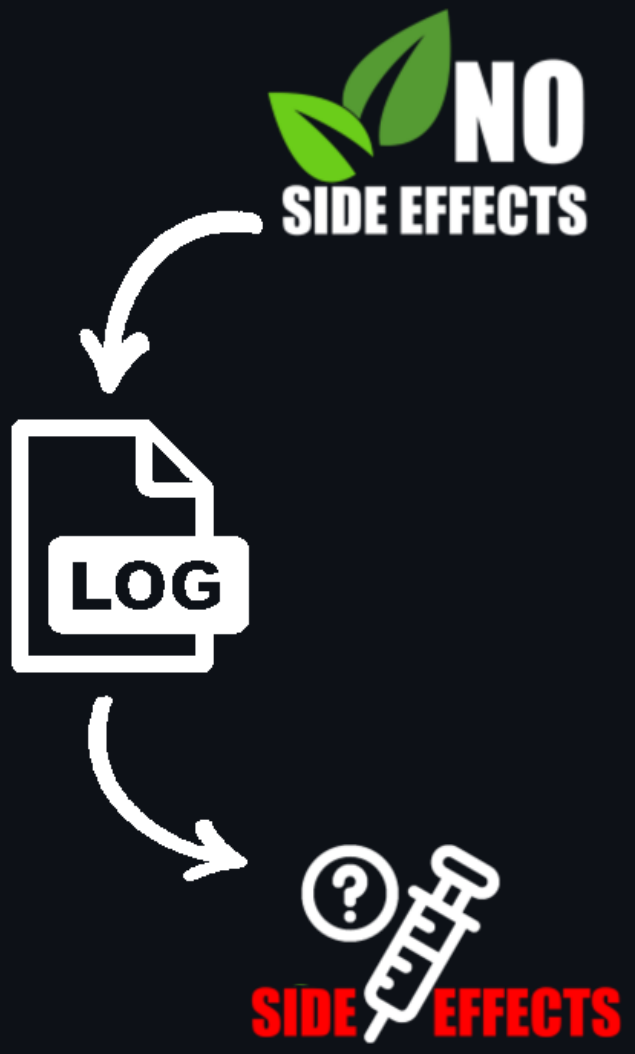
```
def distance(p1: Point, p2: Point) -> float:
    return ((p1.x - p2.x) ** 2 + (p1.y - p2.y) ** 2) ** 0.5
```

Примеры побочных эффектов:

- Изменение внешних переменных
- ...

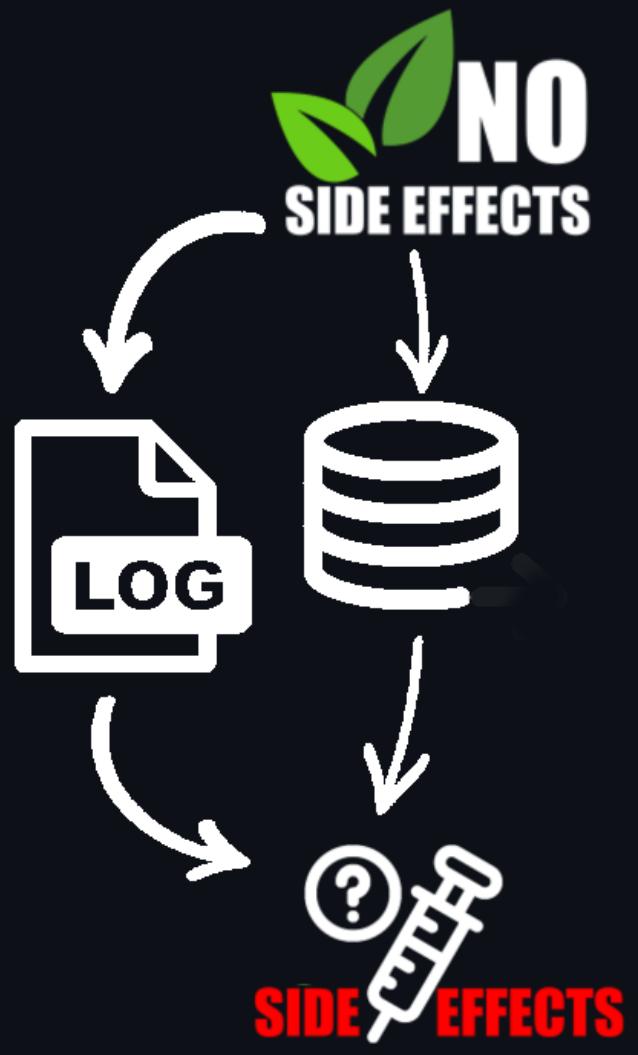
Примеры побочных эффектов:

- Изменение внешних переменных
- Логирование
- ...



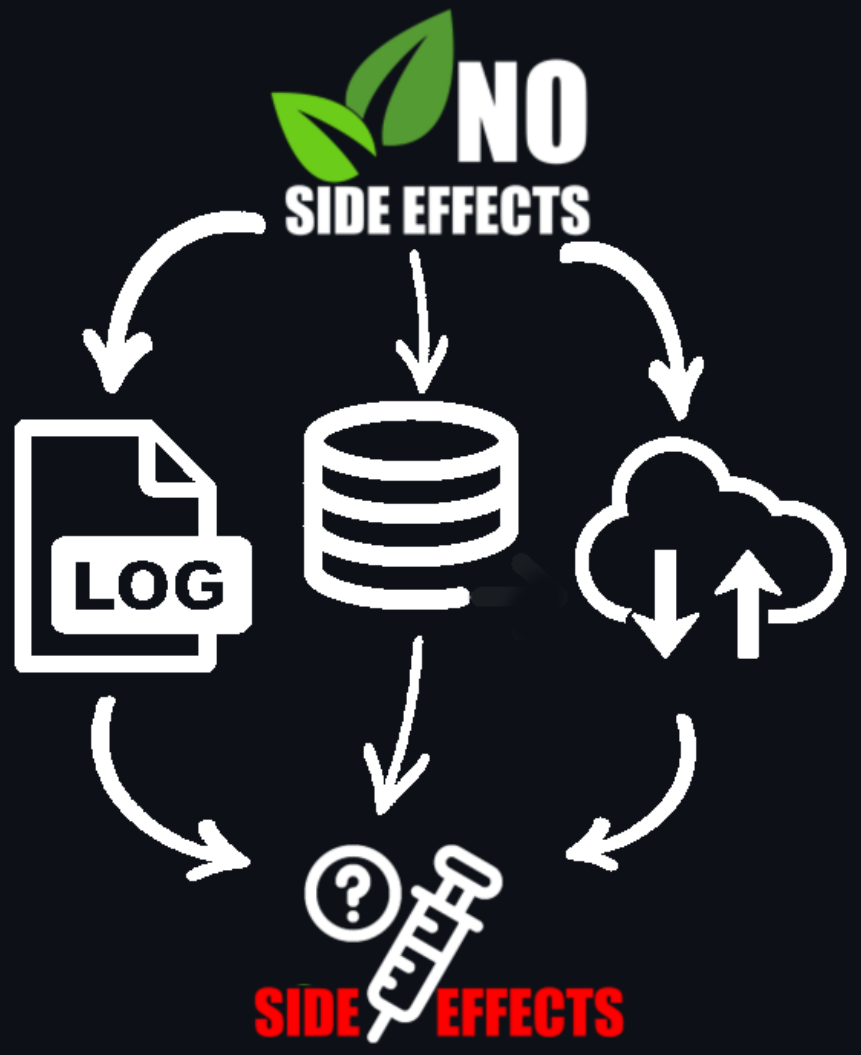
Примеры побочных эффектов:

- Изменение внешних переменных
- Логирование
- Обращения к базе данных
- ...



Примеры побочных эффектов:

- Изменение внешних переменных
- Логирование
- Обращения к базе данных
- Выполнение сетевых запросов
- ...



Примеры побочных эффектов:

- Изменение внешних переменных
- Логирование
- Обращения к базе данных
- Выполнение сетевых запросов
- Вывод текста (в консоль, в файл...)
- Получение пользовательского ввода
- Импорт модуля
- ...

Чистые функции это прекрасно, потому что:

- Их легче отлаживать
- Бесплатно параллелятся по определению

```
def do_something(x):
    ...

with Pool(N) as p:
    print(p.map(do_something, data_stream))
```

- А также бесплатно кэшируются

```
@functools.cache
def do_something(a, b):
    ...
```

- Один из инструментов борьбы с побочными эффектами

- Чистые функции
- **Функции высшего порядка и функции первого класса**
- Ленивые вычисления
- Рекурсия как основной способ имплементации алгоритмов
- Особый акцент на обработке последовательностей
- Декларативное программирование
- Думаем на уровне типов
- Иммутабельность

Функции высшего порядка и функции первого класса

- принимают другие функции как аргументы или возвращают их как результат.

Лучший и всем знакомый пример это декоратор:

```
@retry(3)
def upload(data):
    ...
```

```
upload_with_retries = retry(3)(upload)
```

Функции - такие-же объекты как и всё остальное

```
def retry(retries: int):
    def decorator(func: Callable):
        @wraps(func)
        def wrapper(*args, **kwargs):
            result = None
            for attempt in range(retries):
                try:
                    result = func(*args, **kwargs)
                    return result
                except Exception as e:
                    if attempt + 1 == retries:
                        raise e
            return result
        return wrapper
    return decorator
```

Partial

- частичное применение

```
import sys
from functools import partial

def some_func(logger: Callable):
    logger('Hello')
    logger('World')

def log(date, level, message): ...

warning_log = partial(log, now(), 'WARNING')
some_func(warning_log)
```

Curring

```
curry(f(a, b, c)) => f(a)(b)(c)
```

`curry()` - синтаксический сахар вокруг partial, доступна в `toolz` и `funcy`

```
from toolz import curry

def some_func(logger: Callable):
    logger('Hello')
    logger('World')

def log(date, level, message): ...

curried_log = curry(log)
warning_log = curried_log(some_date)('WARNING')

some_func(warning_log)
```

Композиция функций

```
from toolz import compose

def str_to_json(column): ...
def json_to_data(json): ...
def data_to_user(data): ...

parse_column = compose(str_to_json, json_to_data, data_to_user)
users = (parse_column(column) for column in columns)
```

- Чистые функции
- Функции высшего порядка и функции первого класса
- **Ленивые вычисления**
- Рекурсия как основной способ имплементации алгоритмов
- Особый акцент на обработке последовательностей
- Декларативное программирование
- Думаем на уровне типов
- Иммутабельность

Ленивые вычисления

В python ленивые вычисления доступны и широко используются

```
def decode(line: str):
    ...

def get_lines(filename):
    with open(filename) as f:
        for line in f:
            yield line
...

pattern = r"^Abnormal program \(.*\) termination\.\.\.$"
decoded = (decode(line) for line in get_lines("file.log"))

if any(line for line in decoded if re.search(regexp_line)):
    print("Found!")
```

- Чистые функции
- Функции высшего порядка и функции первого класса
- Ленивые вычисления
- **Рекурсия как основной способ имплементации алгоритмов**
- Особый акцент на обработке последовательностей
- Декларативное программирование
- Думаем на уровне типов
- Иммутабельность

Рекурсия

```
def run_connection(client):
    ...

def main_loop(server_socket):
    client, _ = server_socket.accept()
    run_connection(client)
    main_loop(server_socket)

def main():
    ...
    main_loop(server_socket)
```

Почему рекурсия?

1. Изолированные итерации без изменений состояния
2. Инструмент математического языка

```python
def run_connection(client):
    ...

def main_loop(server_socket):
    client, _ = server_socket.accept()
    run_connection(client)
    main_loop(server_socket)  # Boom!  RecursionError

def main():
    ...
    main_loop(server_socket)
```

TCO оптимизация

```
from fn import recur


@recur.tco
def main_loop(server_socket):
    client, _ = server_socket.accept()
    run_connection(client)
    return True [server_socket]
```

Реализация `@recur.tco`

```
class tco(object):
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        action = self
        while True:
            result = action.func(*args, **kwargs)
            if not result[0]: return result[1]
            act, args = result[:2]
            if callable(act): action = act
            kwargs = result[2] if len(result) > 2 else {}
```

* Существует и более экстремальный tco декоратор.

* Не стоит ждать настоящей tco в python - Final Words on Tail Calls

```python
def main_loop(server_socket):
    while True:
        client, _ = server_socket.accept()
        run_connection(client)
```

И всё же без рекурсии никуда

Поиск в глубину через цикл

```
def iterative_dfs(visited, graph, node):
    stack = [node]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            for neighbour in graph[vertex]:
                stack.append(neighbour)
```

Рекурсивный вариант

```
def recursive_dfs(visited, graph, node):
    if node not in visited:
        visited.add(node)
        for neighbour in graph[node]:
            recursive_dfs(visited, graph, neighbour)
```

- Чистые функции
- Функции высшего порядка и функции первого класса
- Ленивые вычисления
- Рекурсия как основной способ имплементации алгоритмов
- **Особый акцент на обработке последовательностей**
- **Декларативное программирование**
- Думаем на уровне типов
- Иммутабельность

Работа с последовательностями

Всё вокруг это списки:

- Парсинг файла - список строк ➔ список токенов ➔ список символов
- Сервер - список соединений ➔ список запросов ➔ список ответов.
- Игра - список игровых объектов ➔ список действий ➔ список событий

Что можно взять из ФП?

- Избавляемся от циклов
- Декларативное описание результата
- Ленивое исполнение

Comprehensions VS map & filter

```
map(lambda x: int(is_prime(x)), range(2, 10)) -> 1, 1, 0, 1, 0, 1, 0, 0
filter(lambda x: is_prime(x), range(2, 10))   -> 2, 3, 5, 7
```

```
(is_prime(x) for x in range(2, 10))      -> 1, 1, 0, 1, 0, 1, 0, 0
(x for x in range(2, 10) if is_prime(x)) -> 2, 3, 5, 7
```

Подробнее в статье The fate of reduce() in Python 3000 - by Guido van van Rossum

Композиция в работе с последовательностями

Императивный подход

```
file_rows = get_file_rows('filght.csv')
parsed_filtered = []
for row in file_rows:
    pos = PlanePos.from_row(row)
    if pos.in_polygon(restrictedArea):
        parsed_filtered.append(pos)
sorted_data = sorted(parsed_filtered, key=lambda pos: (pos.lat, pos.time))
violations = []
for pos in sorted_data:
    violations.append(f"{pos.time}: {pos.lat}-{pos.lon}")
```

Композиция в работе с последовательностями

Сложное преобразование одним выражением

```
file_rows = get_file_rows('flight.csv')

violations = list(
                map(lambda pos: f"{pos.time}: {pos.lat}-{pos.lon}",
                 sorted(
                        filter(lambda pos: pos.in_polygon(restrictedArea),
                            map(PlanePos.from_row, file_rows)),
                        key=lambda pos: (pos.lat, pos.time)
                    )
                )
            )
```

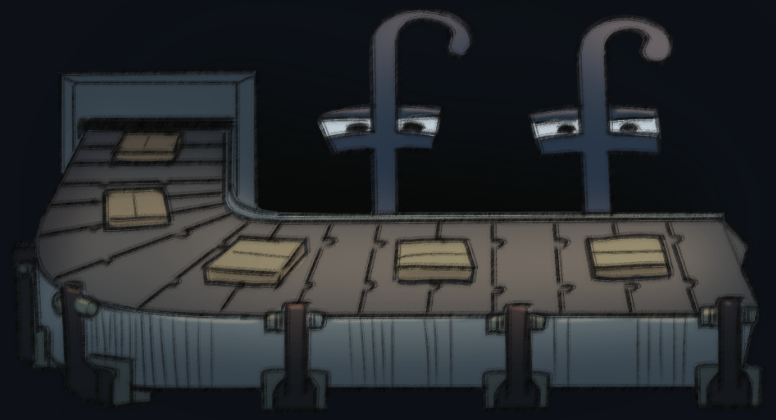
С comprehensions как-то не легче не стало...

```
file_rows = get_file_rows('flight.csv')
violations = [f"{pos.time}: {pos.lat}-{pos.lon}" for pos in
             sorted((pos for pos in (PlanePos.from_row(row) for row in file_rows)
                    if pos.in_polygon(restrictedArea)),
             key=lambda pos: (pos.lat, pos.time))]
```

```python
file_rows = get_file_rows('filght.csv')
parsed = (PlanePos.from_row(row) for row in file_rows)
in_polygon = (pos for pos in parsed if pos.in_polygon(restrictedArea))
sorted_by_lat_time = sorted(in_polygon, key=lambda pos: (pos.lat, pos.time))
violations = [f"{pos.time}: {pos.lat}-{pos.lon}" for pos in sorted_by_lat_time]
```

Композиция и конвейер

```
any(map(filter(...,),)        # Композиция

.filter(..).map(..).any(..)   # Конвейер
```



Конвейер - а как у других?

C# LINQ

```
file_rows.Select(PlanePos.FromRow)
         .Where(pos => pos.InPolygon(restrictedArea))
         .OrderBy(pos => pos.Lat)
         .ThenBy(pos => pos.Time)
         .Select(pos => $"{pos.Time}: {pos.Lat}-{pos.Lon}")
```

Конвейер в python - библиотека Pipe

```
from pipe import select, where, sort

violations = (file_rows |
              select(PlanePos.from_row) |
              where(lambda pos: pos.in_polygon(restrictedArea)) |
              sort(key=lambda pos: (pos.lat, pos.time)) |
              select(lambda pos: f"{pos.time}: {pos.lat}-{pos.lon}"))
```

Было

```
parsed_filtered = []

for row in file_rows:
    pos = PlanePos.from_row(row)
    if pos.in_polygon(restrictedArea):
        parsed_filtered.append(pos)

sorted_data = sorted(parsed_filtered, key=lambda pos: (pos.lat, pos.time))

violations = []
for pos in sorted_data:
    violations.append(f"{pos.time}: {pos.lat}-{pos.lon}")
```

Стало

```
violations = (file_rows |
              select(PlanePos.from_row) |
              where(lambda pos: pos.in_polygon(restrictedArea)) |
              sort(key=lambda pos: (pos.lat, pos.time)) |
              select(lambda pos: f"{pos.time}: {pos.lat}-{pos.lon}"))
```

Одной строкой еще лучше

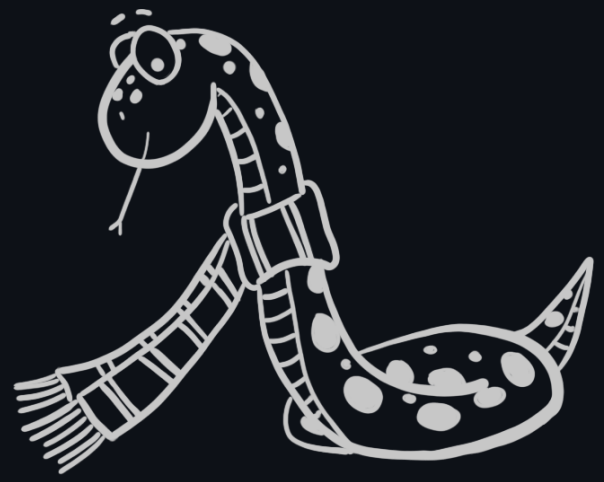
```
from pipe import select, take_while

result = flight | take_while(lambda p: p.lat < 80) | select(lambda p: p.time)
```

Лямбды

Некоторые минусы Python лябмд:

- Нельзя писать многострочные лямбды
- Многословный синтаксис, даже в C++ получилось короче

```
auto fun = []{};  // валидная c++ лямбда
fun();
```



```
{
    "min": lambda x: x.min(),
    "25%": lambda x: x.quantile(0.25),
    "50%": lambda x: x.quantile(0.50),
    "median": lambda x: x.median,
    "# > 0": lambda x: (x > 0)
}
```

```python
def call(name, *args, **kwargs):
    return lambda obj: getattr(obj, name)(*args, **kwargs)


def get(item):
    return lambda obj: getattr(obj, item)


def gt(number):
    return lambda x: x > number


{
    "min": call("min"),
    "25%": call("quantile", 0.25),
    "50%": call("quantile", 0.50),
    "median": get("median"),
    "# > 0": gt(0)
}
```

```python
from functools import partial
from operator import methodcaller as call, itemgetter as get, lt

{
    "min": call("min"),
    "25%": call("quantile", 0.25),
    "50%": call("quantile", 0.50),
    "median": get("median"),
    "# > 0": partial(lt, 0),
}
```

В некоторых случаях может помочь магический объект

whatever

```
from pipe import select, take_while

result = flight | take_while(lambda p: p.lat < 80) | select(lambda p: p.time)
```

```
from whatever import _
from pipe import select, take_while

result = flight | take_while(_.lat < 80) | select(_.time)
```

whatever позволяет реже писать слово `lambda`

```
return {
    "title": try_get(video, lambda x: x['title']),
    "category": try_get(video, lambda x: x['category']),
    "age_limit": try_get(video, lambda x: x['is_adult']),
    "view_count": try_get(video, lambda x: x['hits']),
    "uploader": try_get(video, lambda x: x['author']),
    "duration": try_get(video, lambda x: x['duration']),
}
```

```
return {
    "title": try_get(video, _['title']),
    "category": try_get(video, _['category']),
    "age_limit": try_get(video, _['is_adult']),
    "view_count": try_get(video, _['hits']),
    "uploader": try_get(video, _['author']),
    "duration": try_get(video, _['duration']),
}
```

Разгрузочный пример - RLE

Рассмотрим RLE (run-length encoding) - простой алгоритм сжатия строки

```
"AAAABBBCCXYZDDDDEEEFFFAAAAAABB" -> "4A3B2C1X1Y1Z4D3E3F6A2B"
```

Стандартный подход

```python
def rle_encode(data: str) -> str:
    encoded = []
    i = 0
    while i <= len(data)-1:
        count = 1
        ch = data[i]
        j = i
        while j < len(data) - 1:
            if data[j] == data[j+1]:
                count = count+1
                j = j+1
            else:
                break
        encoded.append(str(count) + ch)
        i = j+1
    return "".join(encoded)
```

- Избавимся от цикла
- Найдем подходящую функцию из itertools
- Пишем лениво (впрочем в данном коде не используем это)

```
from itertools import groupby

def rle_encode(data: str) -> str:
    return "".join(str(len(list(group_items))) + key
                   for key, group_items in groupby(data))
```

- Чистые функции
- Функции высшего порядка и функции первого класса
- Ленивые вычисления
- Рекурсия как основной способ имплементации алгоритмов
- Особый акцент на обработке последовательностей
- Декларативное программирование
- **Думаем на уровне типов**
- Иммутабельность

Думаем на уровне типов

Рассмотрим:

- Фантомные типы
- Рекурсивные типы

Зачем нужны фантомные типы?

Вспомним наши координаты

```
@dataclass
class PlanePos:
    lat: float
    lon: float
```

Допустимо перепутать аргументы местами

```
lat = 50
lon = 120
p = PlanePos(lon, lat)
```

Допустимы потенциально бесмысленные операции

```
p3.lat = p1.lat + p2.lon
```

Фантомные типы

- типы, не использующие как минимум один из своих параметров типа

```
T = TypeVar('T')
class MyType(Generic[T]):
    value: int
```

```python
class Latitude: pass
class Longitude: pass

T = TypeVar('T')
class Coordinate(Generic[T]):
    def __init__(self, value: float): ...
    def __add__(self, second: Coordinate[T]): ...

@dataclass
class PlanePos:
    lat: Coordinate[Latitude]
    lon: Coordinate[Longitude]
```

Стало меньше простора для ошибок

```python
latitude = Coordinate[Latitude](10)
longitude = Coordinate[Longitude](50)

PlanePos(longitude, latitude)  # Mypy error
nonsense = latitude + longitude  # Mypy error
```

По прежнему можем себе навредить

```
latitude = Coordinate[Latitude](-100)
longitude = Coordinate[Longitude](190)
```

```python
from phantom import Phantom
from phantom.predicates.interval import inclusive

class Latitude(float, Phantom, predicate=inclusive(-90, 90)):
    ...

class Longitude(float, Phantom, predicate=inclusive(-180, 180)):
    ...

@dataclass
class PlanePos:
    lat: Latitude
    lon: Longitude

PlanePos(500, -500)                          # mypy error
PlanePos(Latitude(500), Longitude(-500))  # runtime error
PlanePos(Longitude(120), Latitude(80))    # mypy error
PlanePos(Latitude(80), Longitude(120))    # ok
```

Принцип "Parse, don't validate", библиотека phantom-types

Рекурсивные типы

```
from __future__ import annotations

class Tree:
    def __init__(self, left: Tree, right: Tree):
        self.left = left
        self.right = right
```

Рекурсивные типы в жизни

```
from __future__ import annotations
from typing import Union, Dict, List

JSON = Union[None, bool, str, float, int, List[JSON], Dict[str, JSON]]

good: JSON = {'a': ['b', {"c": [1, 2]}]}
bad: JSON = {'a': ['b', {3: [1, 2]}]}
ugly: JSON = {'a': ['b', {"c": [set(), 2]}]}
```

Муру:

```
from __future__ import annotations
from typing import Union, Dict, List

JSON = Union[None, bool, str, float, int, List[JSON], Dict[str, JSON]]

good: JSON = {'a': ['b', {"c": [1, 2]}]}      # ok
bad: JSON = {'a': ['b', {3: [1, 2]}]}         # Dict entry 0 has incompatible type "int"
ugly: JSON = {'a': ['b', {"c": [set(), 2]}]}  # List item 0 has incompatible type "Set[<nothing>]"
```

* нужен флаг --enable-recursive-aliases

* на момент 04.09.2022 фича эксперементальная

- Чистые функции
- Функции высшего порядка и функции первого класса
- Ленивые вычисления
- Рекурсия как основной способ имплементации алгоритмов
- Особый акцент на обработке последовательностей
- Декларативное программирование
- Думаем на уровне типов
- **Иммутабельность**

Иммутабельность

```
@dataclass
class PlanePos:
    lat: Latitude
    lon: Longitude

flight = [
    PlanePos(Latitude(10), Longitude(20)),
    PlanePos(Latitude(40), Longitude(40))
    ...
]
```

Много строк спустя, где-то в далёкой далекой функции...

```
flight[0].lat = Latitude(0)
del flight[1]
```

Всё испортили

Иммутабельность - Tuple + NamedTuple

```
from typing import NamedTuple

class PlanePos(NamedTuple):
    lat: Latitude
    lon: Longitude

flight = (
    PlanePos(Latitude(10), Longitude(20)),
    PlanePos(Latitude(40), Longitude(40))
    ...
)
```

Много строк спустя, где-то в далёкой далекой функции ...

```
flight[0].lat = Latitude(0)  # AttributeError: can't set attribute
del flight[1]                # TypeError: 'tuple' object doesn't support item deletion
```

... в этот раз ничего сломать нельзя

Иммутабельность - Tuple + Frozen dataclass

```
@dataclass(frozen=True)
class PlanePos:
    lat: Latitude
    lon: Longitude

flight = (
    PlanePos(Latitude(10), Longitude(20)),
    PlanePos(Latitude(40), Longitude(40))
    ...
)
```

Много строк спустя, где-то в далёкой далекой функции ...

```
flight[0].lat = Latitude(0)   # dataclasses.FrozenInstanceError: cannot assign to field 'lat'
del flight[1]                 # TypeError: 'tuple' object doesn't support item deletion
```

... в этот раз ничего сломать нельзя

Иммутабельность - а если нужен dict?

Mutable Version	Immutable Version
list	tuple
set	frozenset
dict	???

Библиотека `pyrsistent`

Mutable Version	Immutable Version
list	tuple, PVector, PList
set	frozenset, PSet
dict	PMap, PRecord

И многое другое

```python
from pyrsistent import pmap

@dataclass(frozen=True)
class PlanePos:
    lat: Latitude
    lon: Longitude

flight = pmap({0: PlanePos(Latitude(10), Longitude(20), []),
               1: PlanePos(Latitude(30), Longitude(40), []),
               ...
})
```

Много строк спустя, где-то в далёкой далекой функции ...

```python
del flight[0]  # TypeError: 'PMap' object doesn't support item deletion
```

... ничего испортить нельзя

Изменяемые объекты внутри неизменяемых всё ещё могут быть изменены

```
@dataclass(frozen=True)
class PlanePos:
    lat: Latitude
    lon: Longitude
    some_list: list


flight = pmap({0: PlanePos(Latitude(10), Longitude(20), []),
               1: PlanePos(Latitude(30), Longitude(40), []),
               ...
})
```

```
object.__setattr__(flight[0], 'lat', Latitude(0))  # Ok
flight[1].some_list.append(1)                      # Ok too
```

* <extra fanatic> существует возможность заменить все стандартные коллекции на иммутабельные при помощи Pyrthon </extra fanatic>

```python
from typing import Final

class PlanePos:
    lat: Final[Latitude]
    lon: Final[Longitude]

    def __init__(self, lat: Latitude, lon: Longitude):
        self.lat = lat
        self.lon = lon
```

```python
pos = PlanePos(Latitude(0), Longitude(0))
pos.lat = Latitude(10)  # Mypy error: Cannot assign to final attribute "lat"
```

Выводы

- ФП позволяет писать более красивый и выразительный код
- Получилось применить многие концепции в python

Мастер должен знать инструменты на своем станке. Но иногда не помешает знать инструменты и на чужом, может получится чего свиснуть

Остаётся за кадром

- Hy - Lisp диалект для Python

```
=> (print f"Hy, world! {(+ 1 41)}")
Hy, world! 42
```

- coconut - Функциональный язык, который компилируется в Python

```
>>> range(10) |> map$(.**2) |> list
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- PyMonad, returns - обработка ошибок в функциональном стиле

Материалы

Книги:

Functional Python Programming - Steven F. Lott

Functional Programming in Python - David Mertz

Статьи:

The fate of reduce() in Python 3000 - by Guido van van Rossum

Itertools: Itertools tricks, More itertools, Tour of Python Itertools

Functional Programming HOWTO

Some History of Functional Programming Languages. D. A. Turner

Python Type Hints are Turing Complete - Github

Рекурсия Tail Recursion Elimination, Final Words on Tail Calls, Реализация TCO

Доклады:

Berlin 2016 - Daniel Kirsch - Functional Programming in Python

Joel Grus: Learning Data Science Using Functional Python