Метапрограммирование с **Python**

Об идеи
генерировать unittest-ы из кода

И почему это возможно (кажется)

Докладчик

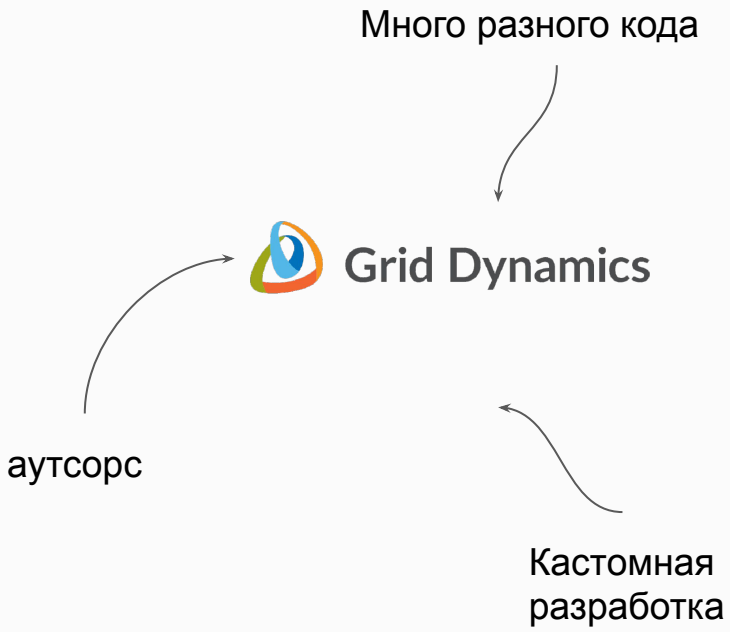
Юлия Волкова (Iuliia Volkova)

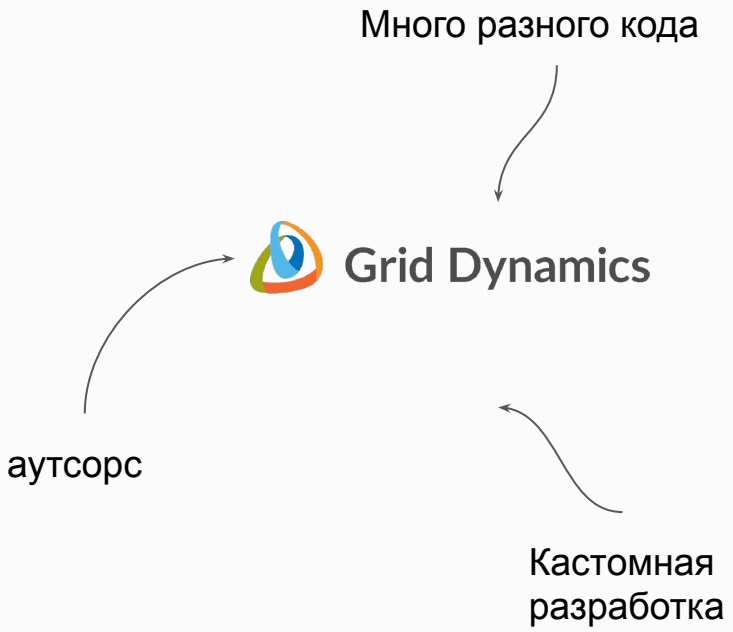
Python Developer

https://medium.com/@xnuinside

https://github.com/xnuinside

https://twitter.com/xnuinside

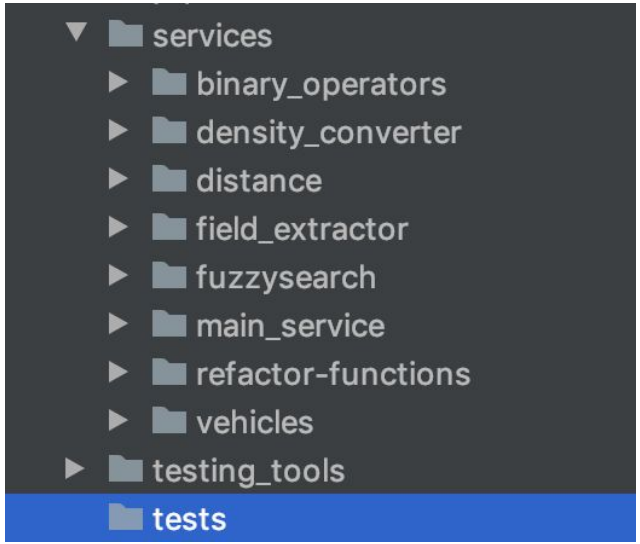



Много разного кода
Grid Dynamics
аутсорс
Кастомная
разработка

2 мысли

Новый проект

Как это обычно



Как это обычно

~ 8 сервисов

~ 20 тыс строк

50% кода вида

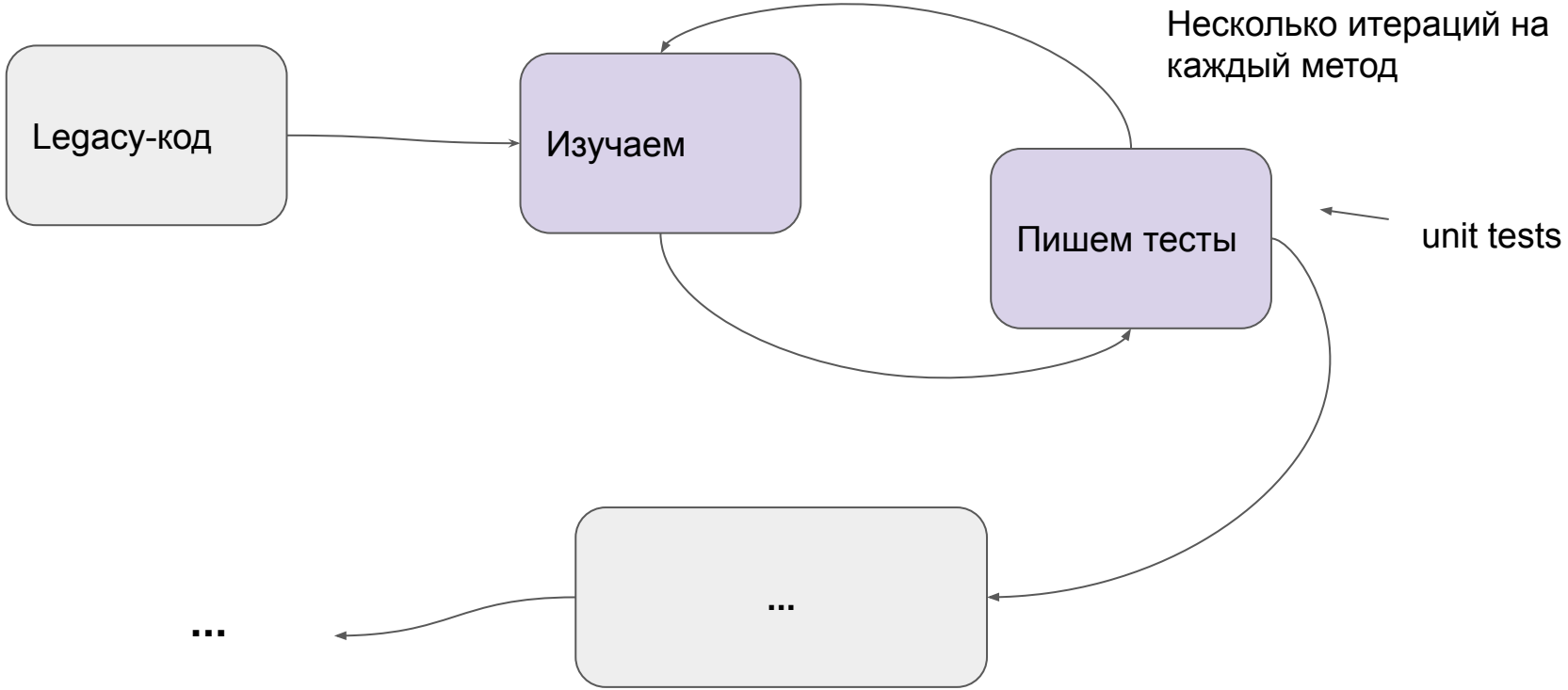
```
async def calculateExpectedMaintanceTime(vehichle,
station):
   return {'id': uuid.uuid4().hex, 'hours': 12}
```

```
def calculate_expected_time(rate,
product_quantity):
   hours = product_quantity['value']/rate['value']
   return {'id': uuid.uuid4().hex, 'days': 0,
'hours': hours}
```

Задачи

- Понять как код работает
- Гарантировать работоспособность кода при изменениях (покрыть тестами)
- Вносить изменения в существующий код

Когда проект без тестов



Хочется
автоматизации

Ну хотя бы
чуть-чуть

Генерация тестов обычно

- Автотесты
- Property-based тесты

Например,

Hypothesis:
https://hypothesis.readthedocs.io

```
from hypothesis import given, settings
from hypothesis.strategies  import lists,
integers

@given(
    list1=lists(integers(min_value=1)),
    list2=lists(integers(min_value=1)),
    depth=integers(min_value=1)
)
@settings(deadline=300)  # <- NEW CODE
def
test_average_agreement_properties(list1,
list2, depth):
      ...
```

Особенности генерации

Мы должны:

- добавить в **код** дополнительные декораторы/контракты

Особенности генерации

Мы должны:

- добавить в **код** дополнительные декораторы/контракты
- добавить в **существующие уже тесты** дополнительные декораторы/контракты **для получения большего количества тест-кейсов** в авто-режиме

А у нас

Тестов нет

Код трогать нельзя

Что хочется?

Код

code_module.py

```
class CustomException(Exception):
    pass


def func_condition(arg1):
    if arg1 == '15':
        raise CustomException('we hate 15')

    elif arg1 > 2:
        print(f'{arg1} more when 2')

    else:
        return arg1
```

Кнопочку жмяк

Тест

test_code_module.py

```
def test_func_condition(capsys):

    with pytest.raises(CustomException):
        # error message: we hate 15
        func_condition(arg1="15")

    func_condition(arg1=6)
    captured = capsys.readouterr()
    assert captured.out == '6 more when 2\n'

    assert func_condition(arg1=-238) == -238
```

Что хочется

code_module.py

```
def return_alias():
  dict_var = {'num': 'alias',
'value_two': 1}

  second_dir = {'str': 123}

  alias_var = dict_var

  result = (dict_var['num'] *
alias_var['value_two']) +
second_dir['str']

  return result
```

test_code_module.py

```
def test_return_alias():

   with pytest.raises(TypeError):
       # error message: can only
concatenate str (not "int") to str
       assert return_alias()
```

Результат - код, который я могу:

- прочитать глазами
- поправить
- дополнить
- **и закоммитить в git**

Перегенерировать тесты, которые были созданы ранее - **плохо**

А можно ли вообще?

```
class CustomException(Exception):
    pass


def func_condition(arg1):
    if arg1 == '15':
        raise CustomException('we hate 15')

    elif arg1 > 2:
        print(f'{arg1} more when 2')

    else:
        return arg1
```

Имя

Вид объекта (синхронная функция)

Параметры

Область видимости

Вид объекта (класс)

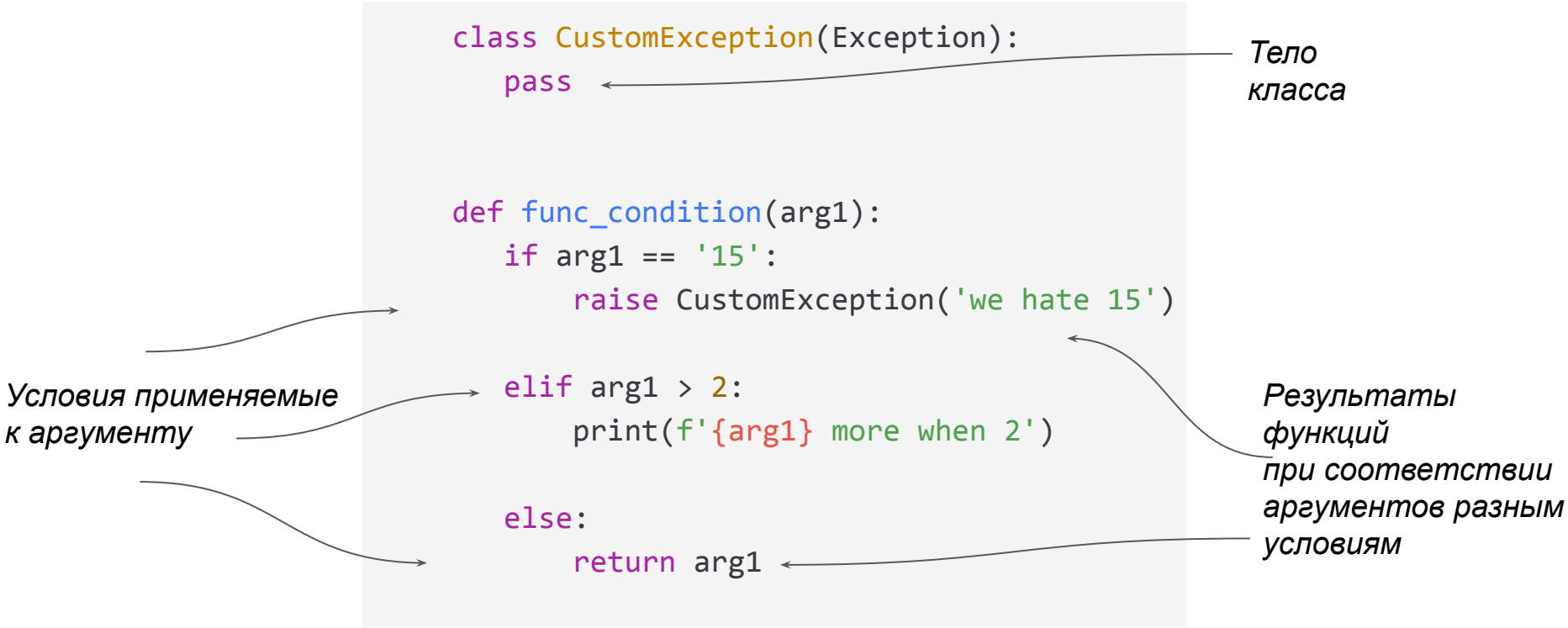
Родители
(для класса)

```
class CustomException(Exception):
    pass


def func_condition(arg1):
    if arg1 == '15':
        raise CustomException('we hate 15')

    elif arg1 > 2:
        print(f'{arg1} more when 2')

    else:
        return arg1
```


class CustomException(Exception):
pass
Тело класса
def func_condition(arg1):
if arg1 == '15':
raise CustomException('we hate 15')
elif arg1 > 2:
print(f'{arg1} more when 2')
else:
return arg1
Условия применяемые к аргументу
Результаты функций при соответствии аргументов разным условиям

И это всё только по синтаксису

о 100% генерации речи НЕ идет

сколько можно получить таким путем - ?

Есть ли что-то готовое?

Auger

https://github.com/laffra/auger

- Работает только с классами
- **Запускает код** (работает на базе **sys.trace**, ловит код в рантайме)
- Что-то с поддержкой Python 3
- **Только положительные сценарии**
- Сильно ограничен (надо передавать класс для которого нужна генерация тестов, к примеру)

Пример из документации:

To generate a unit test for this class, we run the code again, but this time in the context of Auger:

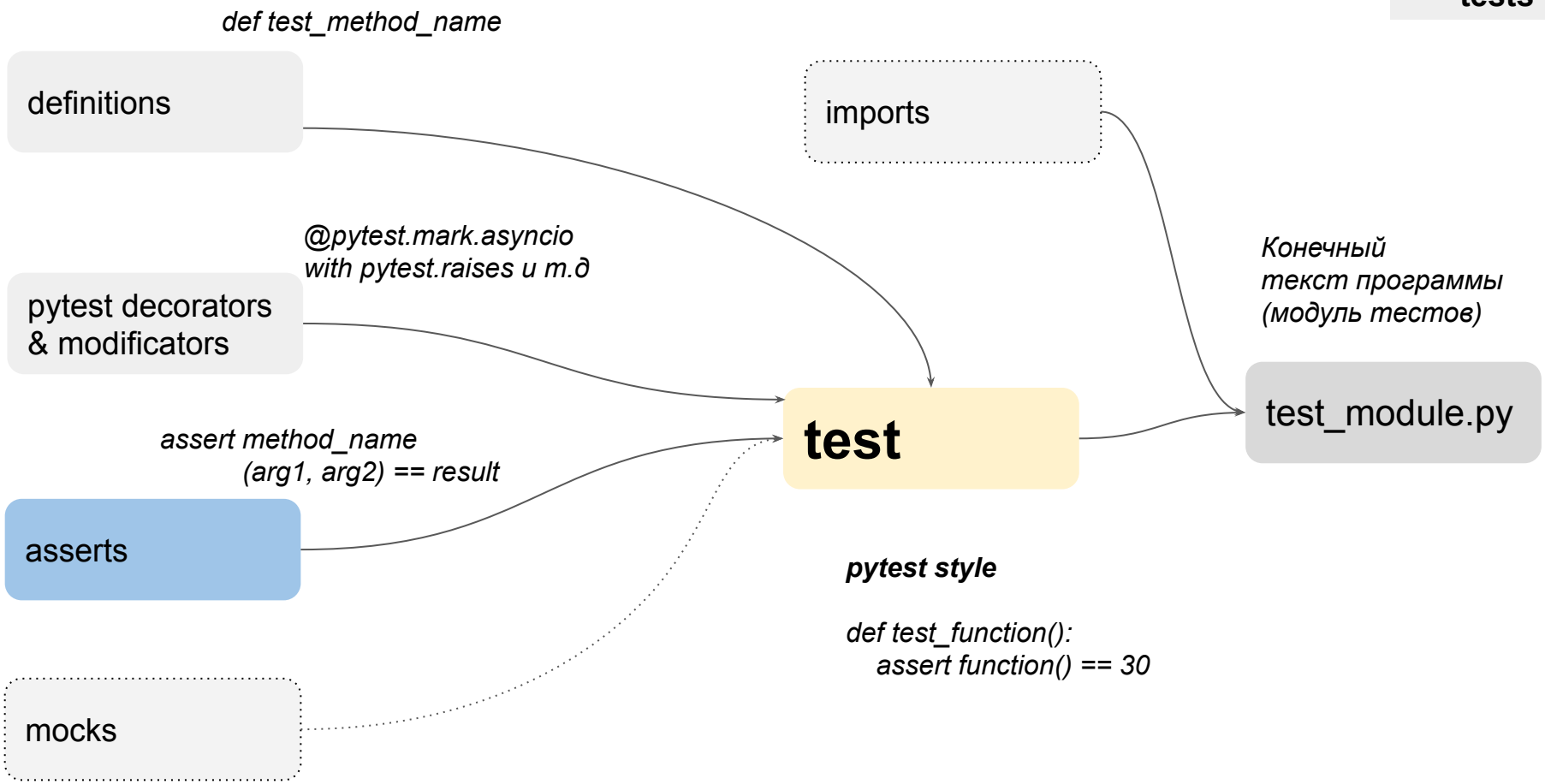
```
import auger

with auger.magic([Foo]):
      main()
```

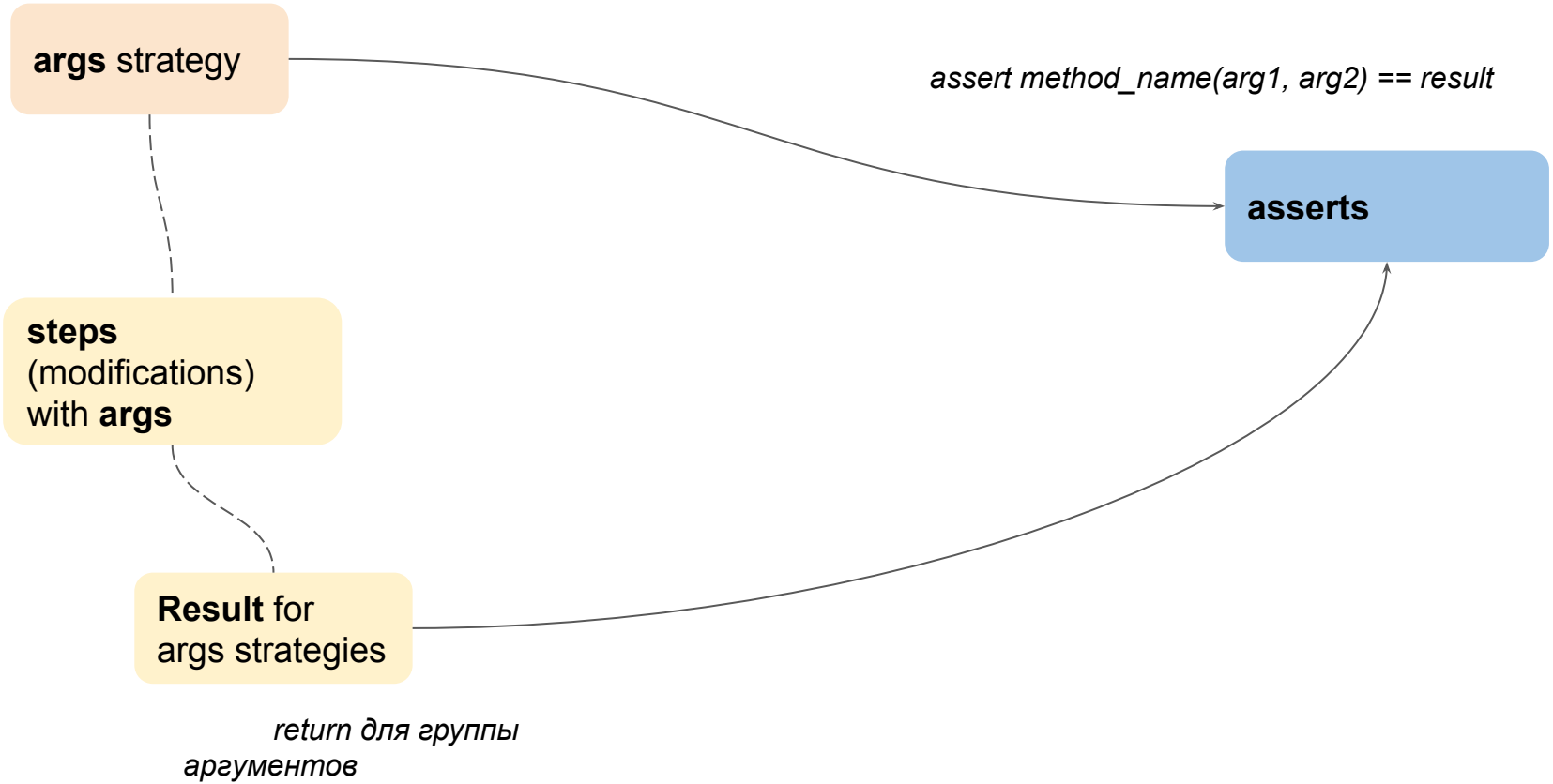

?
input
output
code
metaprogramm
code with tests

Насколько сложно?

Как получить тест?


def test_method_name
definitions
imports
@pytest.mark.asyncio
with pytest.raises и т.д
pytest decorators
& modificators
Конечный
текст программы
(модуль тестов)
test_module.py
assert method_name
(arg1, arg2) == result
test
asserts
pytest style
def test_function():
assert function() == 30
mocks

Стратегия для получения выраженной группы аргументов



args strategy

Стратегии

```
def condition_func(arg1, arg2, arg3):

    if arg1 == '15':  1
        raise CustomException('we hate 15')

    elif arg2[3] > 2:  2
        print(f'{arg2[3]} more when 2')
        return

    var = 1
    alias = var  3
    return arg1 * arg2[3] + arg3['number'],
var * arg1 * alias - 2
```

Стратегии - правила генерации групп аргументов включающие в себя описание **типа данных** аргумента, а также **правила для генерации значений**

Примеры **правил:**

arg1 > 2 and arg1 != ‘15’

dict(‘need_this_key’=randint(0,2))

str(‘*@mail.ru’, in)

args strategy

Стратегии

```
def condition_func(arg1, arg2, arg3):

    if arg1 == '15':  1
        raise CustomException('we hate 15')

    elif arg2[3] > 2:  2
        print(f'{arg2[3]} more when 2')
        return

    var = 1
    alias = var  3
    return arg1 * arg2[3] + arg3['number'],
var * arg1 * alias - 2
```

f(x)

condition_func

Аргумент (x)

arg1, arg2, arg3

значения функции

- raise Exception('we hate 15')
- return arg1 * arg2[3] + arg3['number'], var * arg1 * alias - 2
- None print(f'{arg2[3]} more when 2')

args strategy

Стратегии

```
def condition_func(arg1, arg2, arg3):

    if arg1 == '15':  1
        raise CustomException('we hate 15')

    elif arg2[3] > 2:  2
        print(f'{arg2[3]} more when 2')
        return

    var = 1
    alias = var  3
    return arg1 * arg2[3] + arg3['number'],
var * arg1 * alias - 2
```

Явные правила

1 **arg1 = ‘15’**

$f(x_1)$ = `raise Exception('we hate 15')`

2 **arg2[3] > 2 and arg1 != ‘15’**

$f(x_2)$ = `None`

3 **arg2[3] <= 2 and arg1 != ‘15’**

int *str*

$f(x_3)$ = arg1

args strategy

Стратегии

```
def condition_func(arg1, arg2, arg3):

    if arg1 == '15':  1
        raise CustomException('we hate 15')

    elif arg2[3] > 2:  2
        print(f'{arg2[3]} more when 2')
        return

    var = 1
    alias = var  3
    return arg1 * arg2[3] + arg3['number'],
var * arg1 * alias - 2
```

Неявные правила

condition_func(12, [0,1, 4, ['привет, Вася!'], {'number': 2})

TypeError: '>' not supported between instances of 'str' and 'int'

condition_func(12, [0,1,3, 2], {'number': 'привет, Вася!'})

TypeError: can only concatenate str (not "int") to str

args strategy

Таблица стратегий

function / args	**Funct1 - [name, args_names]**		
arg1	value strategy 1.1	value strategy 1.2	value strategy 1.N
arg2	value strategy **2.1**	value strategy **2.1**	value strategy 2.N
arg3	value strategy 3.1	value strategy 3.2	value strategy 3.N
argN	value strategy N.1	value strategy N.2	value strategy N.N
result	result 1	result 2	result N

От стратегий аргументов
к **результатам**

Result for args strategies

Результаты

```
def condition_func(arg1, arg2, arg3):

    if arg1 == '15':
        raise CustomException('we hate 15')

    elif arg2[3] > 2:
        print(f'{arg2[3]} more when 2')
        return

    var = 1
    alias = var
    return arg1 * arg2[3] + arg3['number'],
var * arg1 * alias - 2
```

Для получения результата нам нужны:

- **Аргументы** (уже созданные, уже полученные)
- **Шаги** преобразований аргументов
- Собственно сам **return** (стратегии результатов)

Result for
args strategies

Шаги

```
def function_with_args_modifications(arg_1):
    arg_1 *= 10
    arg_1 = str(arg_1) + ' was incremented'
    return arg_1
```

1. `arg_1 * 10`
2. `str(arg_1)`
3. `arg_1 + ' was incremented'`

Результат `arg_1`

Result for args strategies

Результаты

```
def calculate_expected_time(rate,
product_quantity):

    hours = product_quantity[1]/rate[0]
    return {'id': uuid.uuid4().hex,
'days': 0, 'hours': hours}
```

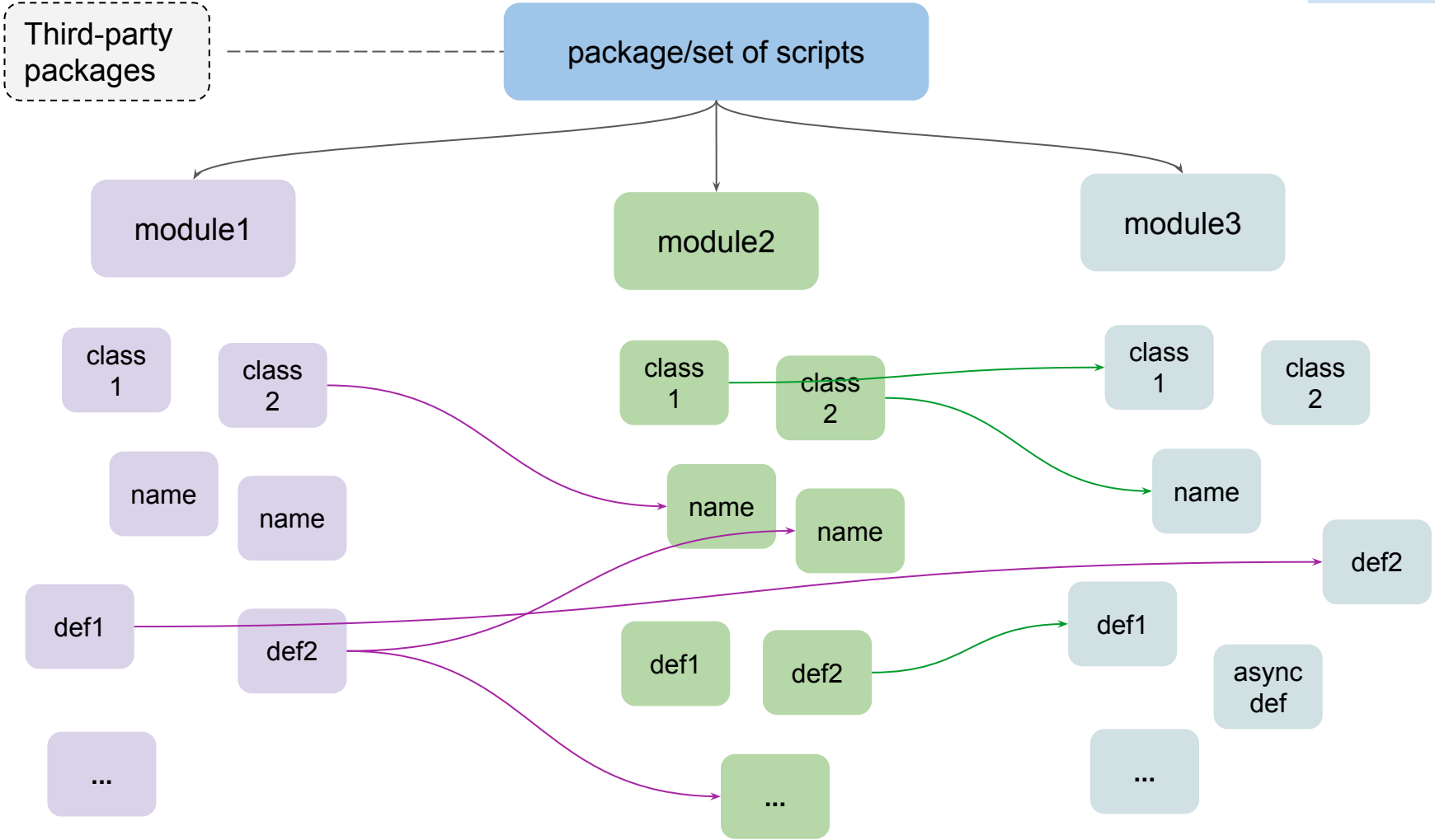
А ещё

Результат может быть:

- Постоянным
- полностью или частично **случайным** (datetime, random, uuid и тд)

А ещё mock-и и импорты

Быстро про код


Third-party packages
package/set of scripts
module1
module2
module3
class 1
class 2
name
name
def1
def2
...
class 1
class 2
name
name
def1
def2
...
class 1
class 2
name
def2
def1
async def
...

Свойства кода

Код изначально обезличен

Доменная область, бизнес-ориентация кода - это исключительно человеческая составляющая

Изменчивость

Код модифицируется/ дописывается/ удаляется/ подвергается рефакторингу

Вариативность

Один и тот же результат может быть достигнут разным набором операций

Вариативность

```
def validate_package_weight(weight):
    if weight <= 0:
        raise Exception("Weight of
package cannot be 0 or below")
    else:
        if weight > 200:
            return False
        elif weight < 200:
            return True
```

Вариативность

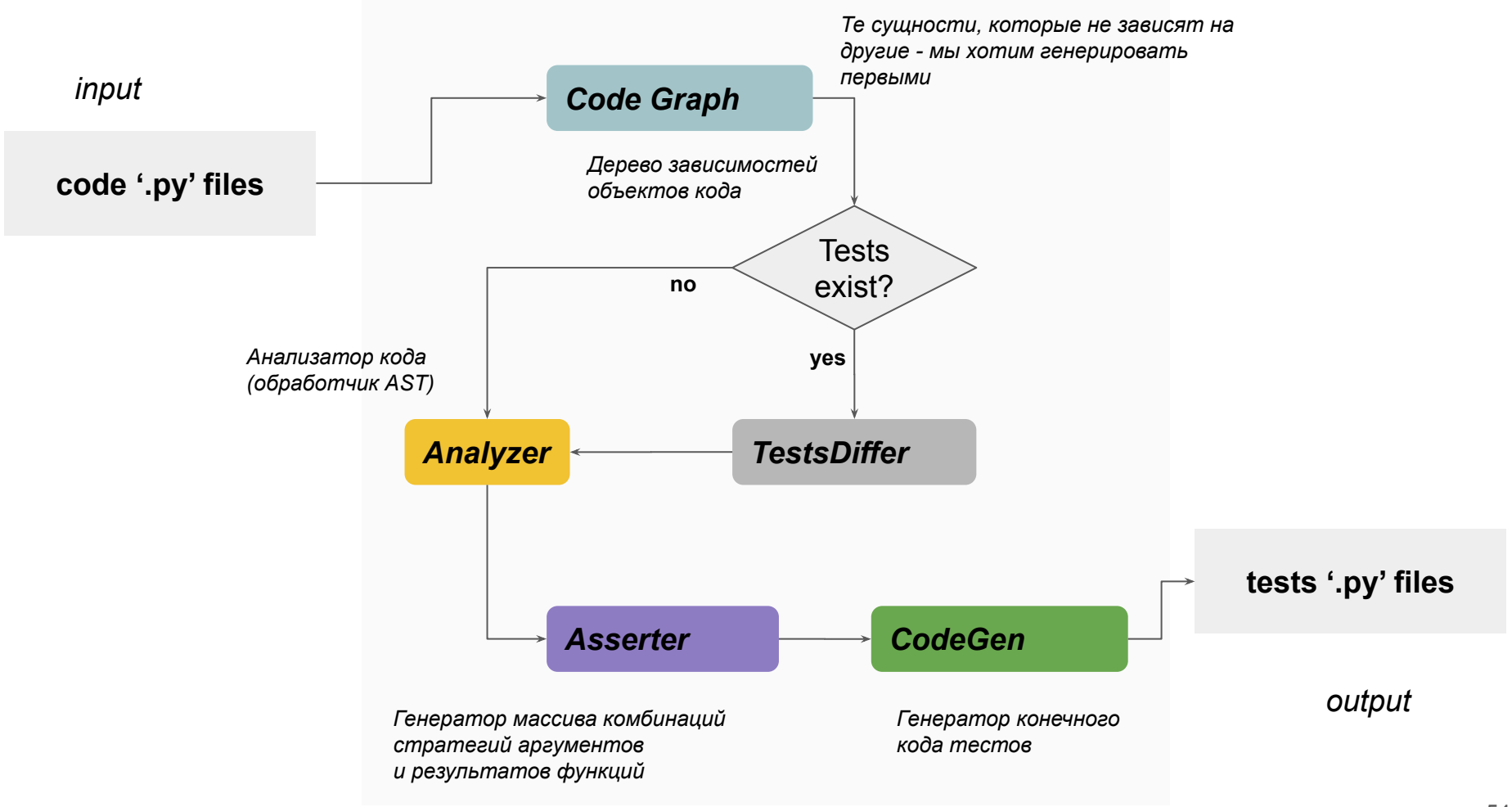
```
def validate_package_weight(weight):
    if weight <= 0:
        raise CustomException("Weight of
package cannot be 0 or below")
    return not (weight > 200)
```

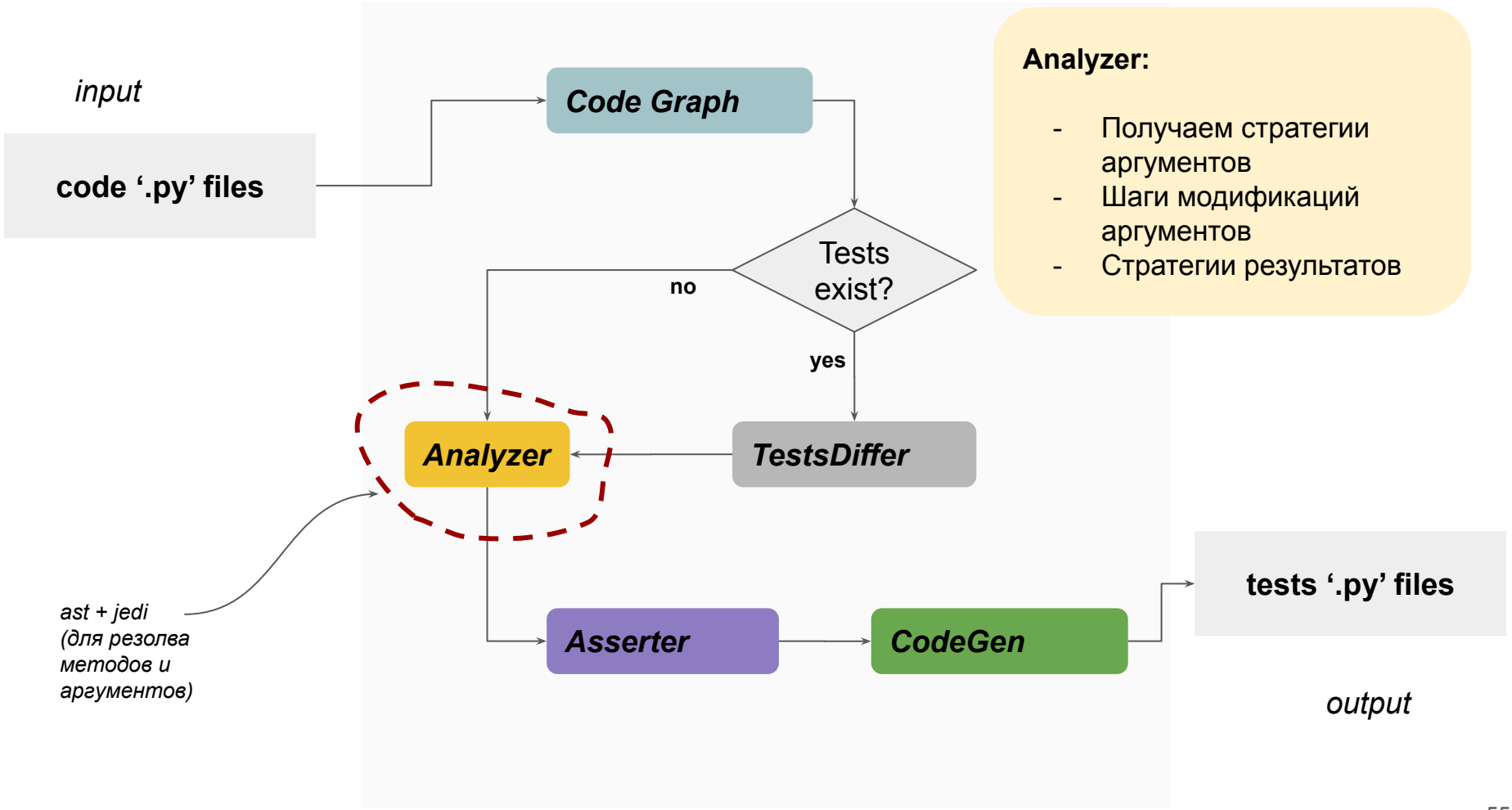
Итого

- Всё что мы проговорили про код - нам нужно учитывать
- Чтобы создать тест нам нужно:
  - Сначала получить все возможные **стратегии** аргументов и **шаги** для получения **результата** под эти стратегии
  - Сгенерировать значения аргументов по стратегиям *(очень рассчитываю на **готовые генераторы**)*
  - **Прогнать** для каждого набора аргументов **все шаги для получения результатов**

И вот ассерт готов

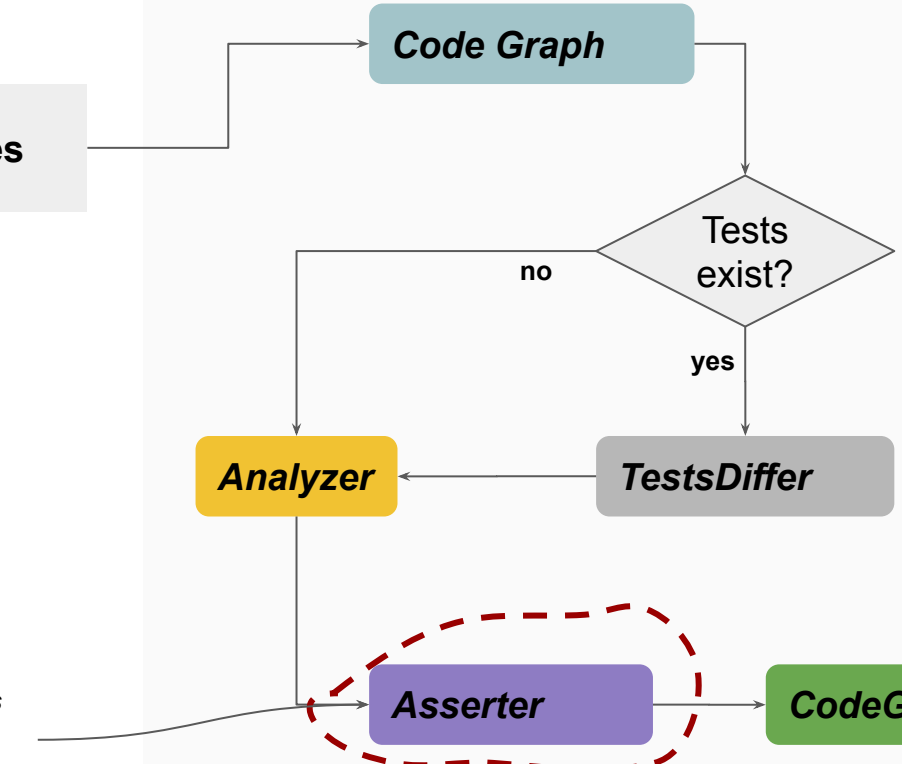
Некоторый план


input
code '.py' files
Code Graph
Те сущности, которые не зависят на другие - мы хотим генерировать первыми
Дерево зависимостей объектов кода
Tests exist?
no
yes
Анализатор кода (обработчик AST)
Analyzer
TestsDiffer
Asserter
CodeGen
tests '.py' files
output
Генератор массива комбинаций стратегий аргументов и результатов функций
Генератор конечного кода тестов


input
code '.py' files
Code Graph
Tests exist?
no
yes
Analyzer
TestsDiffer
ast + jedi
(для резолва
методов и
аргументов)
Asserter
CodeGen
tests '.py' files
output
Analyzer:
- Получаем стратегии аргументов
- Шаги модификаций аргументов
- Стратегии результатов

input

code ‘.py’ files



Asserter:

- Генерируем таблицу аргументов
- Выполняем все шаги с аргументами
- Получаем результат

*Вот сюда должен прийти **hypotesis** или что-то аналогичное*

(для задачи генерации)

tests ‘.py’ files

output

Очень хочется:
максимально использовать уже существующие пакеты

Инструменты

Стадии, которые я прошла

1. **inspect**

   Почему нет: потому что (inspect live objects) запуск кода/недостаток информации для написания полноценных тестов

Стадии, которые я прошла

1. **inspect**

   Почему нет: потому что (inspect live objects) запуск кода/недостаток информации для написания полноценных тестов

2. **Только синтаксический/лексический** анализ

   Почему нет: необходимость обработки большого количества синтаксических конструкций, отслеживание их очередности и т.д, по факту приходим к тому же AST

Стадии, которые я прошла

1. **inspect**

   Почему нет: потому что (inspect live objects) запуск кода/недостаток информации для написания полноценных тестов

2. **Только синтаксический/лексический** анализ

   Почему нет: необходимость обработки большого количества синтаксических конструкций, отслеживание их очередности и т.д, по факту приходим к тому же AST

3. **Микс** из **лексического анализа** и **AST**

   То что есть сейчас (tokens only используется там, где AST излишне) + скоро будет **Jedi** (тот самый, который в основе language сервера)

Ещё сложности

Количество операций для обработки

И их комбинации, они **конечны**, но их очень **большое количество**

Сколько всего типов нод

https://docs.python.org/3/library/ast.html#abstract-grammar

```
module Python
{
    mod = Module(stmt* body, type_ignore *type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
                       string? type_comment)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? returns,
                             string? type_comment)

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value, string? type_comment)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)
```

- Около 60 нод типа expr, stmt и mod

- А ещё около 30 операторов: Eq | NotEq | Lt и т.д.
- Контексты действий - expr_context = Load | Store | Del | AugLoad | AugStore | Param
- И тд

Всего **100+** различных нод/операций

А ещё поведение части из них зависит от **типа** операнда

Количество операций для обработки

И их комбинации, они **конечны**, но их очень **большое количество**

Невозможность точно спланировать структуру на микроуровне*

Постоянно приходится нагромождать **Analyzer**, а затем рефакторить это.

Непрерывный цикл рефакторинга.

* для меня как для человек, у которого не было подобных проектов связанных с AST

Работа с Analyzer

```
def function_with_binary_op(arg1, arg2, arg3):
  var = 'one'
  return arg1 * arg2 + arg3, var
```

Работа с Analyzer

```
def function_with_binary_op(arg1, arg2, arg3):
  var = 'one'
  return arg1 * arg2 + arg3, var
```

FunctionDef(name='function_with_binary...
args=[arg(arg='arg1', annotation=None), ...],...),

body=[**Assign**(targets=[**Name**(id='var')], value=**Str**(s='one')),

Return(value=**Return**(elts=[
BinOp(left=BinOp(left=**Name**(id='arg1'), op=**Mult**, right=**Name**(id='arg2')),
op=**Add**,
right=**Name**(id='arg3')),
Name(id='var')]))],
decorator_list=[], returns=None)])

Работа с Analyzer

```
def function_with_binary_op(arg1, arg2, arg3):
  var = 'one'
  return arg1 * arg2 + arg3, var
```

Чтобы обработать эту функцию нужно обработать:

- **BinOp** ноду с 2-мя **Mult** и **Add** операторами
- **Return** ноду
- **Tuple** ноду
- **Assign**
- **Name**
- **FunctionDef**
- **Str**

Количество операций для обработки

И их комбинации, они **конечны**, но их очень **большое количество**

Невозможность точно спланировать структуру на микроуровне*

Постоянно приходится нагромождать **Analyzer**, а затем рефакторить это.

Непрерывный цикл рефакторинга.

* для меня как для человек, у которого не было проектов подобной сложности, связанных с AST

Работа с AST может быть внезапной

Если ты до этого не работал с определенными операциями, не знаешь как они бьются на нодах и что внутри этой ноды - нужно ожидать что угодно

Но это очень интересно

Немного демо

Если вдруг захочется принять участие

https://github.com/xnuinside/laziest

- Код, который покрывается тестами текущим функционалом:

  tests/code_sample/done

- Кейсы в процессе работы

  tests/code_sample/in_process

- ToDo-кейсы здесь (их тоже надо наполнять и в этом тоже нужна помощь)

  tests/code_sample/todo

Welcome!

Вопросы?

Форматирование строк

AST

1

```
print(f'more {arg1} when 2')
```

JoinedStr:

- Str(s='more '),
- FormattedValue(value=Name(id='arg1'), conversion=-1, format_spec=None)
- Str(s=' when 2')

AST

2

```
print('more {arg1} when 2'.format(
                arg1=arg1))
```

Attribute

- `Str(s='more {arg1} when 2'), attr='format')`
- `keywords=[keyword(arg='arg1', value=Name(id='arg1'))]`

AST

3

```
print('more % when 2' % arg1)
```

BinOp

- left=Str(s='more %s when 2'),
- op=Mod,
- right=Name(id='arg1')

Модули / пакеты

tabnany

tokens & tokenize

ast & _ast

Python Standard Library

- Python Language Services
  - `parser` — Access Python parse trees
  - `ast` — Abstract Syntax Trees
  - `symtable` — Access to the compiler's symbol tables
  - `symbol` — Constants used with Python parse trees
  - `token` — Constants used with Python parse trees
  - `keyword` — Testing for Python keywords
  - `tokenize` — Tokenizer for Python source
  - `tabnanny` — Detection of ambiguous indentation
  - `pyclbr` — Python class browser support
  - `py_compile` — Compile Python source files
  - `compileall` — Byte-compile Python libraries
  - `dis` — Disassembler for Python bytecode
  - `pickletools` — Tools for pickle developers

\+ os, glob и т.д.