2025-10-15

Элементы декларативного программирования в Python

Декларативное программирование — парадигма, при которой определяется ожидаемый результат выполнения программы, а не последовательность отдельных конкретных шагов, которые ведут к нему (в отличие от императивной парадигмы).

Функциональное программирование — подвид декларативного, при котором процесс выполнения программы интерпретируется концептуально как вычисление значений функций (с позиции математического определения функции)

К функциональным ЯП традиционно относят:

- LISP
- F#
- Erlang → Elixir
- Haskell
- Scala
- Эксель-функции*

Элементы функциональной парадигмы присутствуют во многих ЯП, включая Python.

```
(+ (fibonacci (- N 1)) (fibonacci (- N 2)))))
```

```
lambda n:reduce(lambda x,n:[x[1],x[0]+x[1]], range(n),[0,1])[0]
```

Включения (comprehensions)

Списковые и словарные включения как часть более общей категории выражений-генераторов (PEP 289 – Generator Expressions | peps.python.org)

```
list_comp = [letter*2 for letter in 'abcdefg']
dict_comp = {f'id_{x}': x**2 for x in range(1000)}
```

Генераторы (generators)

Часть более общей категории, описанной в PEP-289

Однострочный генератор:

```
single_line_gen = (letter*2 for letter in 'abcdefg')
```

Многострочные генераторы:

```
for prog_idx, prog in enumerate(set_1):
    for mach_idx, mach in enumerate(prog):
        for work_idx, work in enumerate(mach):
            yield (prog_idx, mach_idx, work_idx)
```

Генераторы не хранят в памяти все свои возможные значения, а только текущее значение и правила вывода следующего.

Такой подход обеспечивает экономию памяти (и поэтому в Питоне 3 возможно `range(100_000_000_000_000_000_000_000_000)`), но генератор при этом работает только в 1 сторону. Чтобы вернуться к уже пройденным значениям, генератор необходимо пересоздать.

Следующее значение получается за счет передачи генератора во встроенный метод `next()` или в ходе итерации по аналогии с традиционными итерируемыми коллекциями (но такая итерация возможна лишь 1 раз).

При исчерпании генератора (правило не позволяет получить следующее значение), т.е. при вызове `next()` на генератор с максимальным допустимым значением или при попытке повторной итерации по генератору, будет выброшено исключение `StopIteration`.

Анонимные функции (лямбда-функции, lambda)

```
import math

lambda_sqrt = lambda x: math.sqrt(x+1 if x > 0 else x)

lambda_sqrt(50)
lambda_sqrt(100)
```

```
[skill in map(lambda x: x[0].lower(), cluster) for skill in context_skills]
```

Функции высшего порядка (Higher-Order Functions, HOF)

Функция высшего порядка — функция, которая принимает в качестве аргумента другую функцию и/или возвращает функцию как результат своего вычисления.

А т.к. в Питоне любая функция всегда является одновременно объектом, никто не запрещает передавать этот объект в качестве аргумента в другие функции аналогично другим переменным или возвращать его при вызове функции.

```
import sys

def display(fun, arg):
    print(f'{type(fun)} : {fun}')
    print(f'arg={arg} => fun(arg)={fun(arg)}'

if len(sys.argv) > 1:
    n = float(sys.argv[ 1 ])
```

```
else:
    n = float( input( "число?: " ) )

def pow3(n):  # 1-е определение функции
    return n * n * n
display(pow3, n)

pow3 = lambda n: n * n * n # 2-е определение функции
display(pow3, n)

display((lambda n: n * n * n), n)  # 3-е определение функции, ad-hoc lambda

'''
[out]
<class 'function'> : <function pow3 at 0xb74542ac>
arg=1.3 => fun( arg )=2.1970000000000005
<class 'function'> : <function <lambda> at 0xb745432c>
arg=1.3 => fun( arg )=2.1970000000000005
<class 'function'> : <function <lambda> at 0xb74542ec>
arg=1.3 => fun( arg )=2.1970000000000005
'''
```

Декораторы

Синтаксический сахар для модификации функций другой функцией высшего порядка

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner


def ordinary():
    print("I am ordinary")

# Output: I am ordinary

@make_pretty
def ordinary():
    print("I am ordinary")

# I got decorated
# I am ordinary

decorated_func = make_pretty(ordinary)
```

map()

Применяет переданную функцию к каждому элементу в переданной последовательности и возвращает последовательность результатов той же размерности, что и входная.

```
[skill in map(lambda x: x[0].lower(), cluster) for skill in context_skills]
```

reduce()

Применяет переданную функцию к каждому значению в списке и ко внутреннему накопителю результата (на примере обозначен как `aggr`) — агрегатору.

> **Пример: вычисление факториала 10!**
>
> ```
> from functools import reduce
>
> reduce(lambda aggr, m: aggr*m, range(1, 11))
> ```

```
from functools import reduce

reduce(lambda aggr, b: aggr+b[1][0], [skill for skill in cluster if skill[1]
[1].lower() in context_skills], 0.0)
```

В JavaScript оно даже более наглядно:

```
console.log([1, 2, 3, 4, 5].reduce((a, b) => {return a+b}, 0)) // на выходе 15
```

filter()

Применяет переданную функцию к каждому элементу коллекции и возвращает коллекцию тех элементов исходной коллекции, для которых переданная функция вернула значение истинности.

```
list(filter(lambda x: len(x) >= 3, ready_clusters))
```

> **Important**
>
> Функции `map()` и `filter()` возвращают не список, а особые объекты-итераторы, которые сами по себе являются «ленивыми» (lazy) — вычисления происходят только в момент запроса следующего элемента для экономии памяти.
> Это роднит их с генераторными выражениями.

Поэтому `filter()` всегда можно заменить генератором:

```
(x for x in ready_clusters if len(x) >= 3)
```

Корутины (сопрограммы)

Функции, выполнение которых многократно может быть остановлено, а затем продолжено с той же точки.

Корутины представляют собой более обобщенную форму функций (подпрограмм). Обычно у функций есть одна точка входа и одна точка выхода. У корутин может быть несколько точек входа, выхода и продолжения выполнения.

```
async def my_coroutine():
    print("Starting")
    await asyncio.sleep(1)
    print("Finished")

async def main():
    await my_coroutine()
    print("Main finished")

asyncio.run(main())
```

Note

В Python корутины изначально произошли от генераторов, т.к. генераторы по своему строгому определению ничем от них не отличаются.
До введения обязательного синтаксиса `async/await` в **PEP 492** корутина могла выглядеть так:

```
import re

def grep(pattern):
   while True:
       line = yield
       if re.search(pattern, line):
           print(line)

search = grep("coroutine")
next(search)
search.send("Capybara")
search.send("Avocado")
search.send("Capybara eats an avocado and does coroutines")
```

После для новых проектов допускается только `async/await`, но под капотом в CPython они все еще мало чем отличаются.

```
import asyncio

async def grep(pattern):
   while True:
       line = await asyncio.to_thread(input)
       if re.search(pattern, line):
           print(line)
```

```
asyncio.run(grep("coroutine"))
```

Замыкания (Closures)

Функция, которая возвращает другую функцию с «замороженным» на момент определения набором данных

```
def multiplier(n):
    def mul(k):
        return n * k
    return mul

mul3 = multiplier(3) # функция умножения на 3

print(mul3(3), mul3(5))
# 9, 15
```

Частичное применение функции (partial)

Частичное применение функции предполагает на основе функции N переменных определение новой функции с меньшим числом переменных $M < N$, при этом остальные $N—M$ переменных получают фиксированные «замороженные» значения.

```
from functools import partial

def mulPart(a, b):
    return a * b

par3 = partial(mulPart, 3)  # заморозка функции вокруг значения 3

print(par3(3), par3(5)) # частичное применение функции
# 9, 15
```

Функторы

Функтор — объект некоторого класса, который может быть вызван как функция.

В Python любой объект с определенным методом `__call__` является функтором.

```
class mulFunctor: # эквивалентный функтор
    def __init__(self, val1):
        self.val1 = val1
    def __call__(self, val2):
        return self.val1 * val2
fun3 = mulFunctor(3)
```

```
print(fun3(3), fun3(5)) # применение функтора
# 9, 15
```

Функторами являются и встроенные объекты, например, `staticmethod` и `classmethod`.

Каррирование (Currying)

Каррирование (или транслитерацией карринг) — преобразование функции от многих переменных в функцию, принимающую 1 аргумент за 1 вызов (т.е. берущую их по одному).

> **Note**
>
> Такое преобразование было названо в честь Хаскелла Карри (того же, в честь кого назвали Haskell).
> Впервые каррирование ввели М. Шейнфинкель и Г. Фреге

В некоторых «изначально» функциональных ЯП (Haskell, ML) существует отдельный оператор каррирования. В иных случаях любой язык, поддерживающий замыкания, включая Python, Perl, C++, поддерживает и каррирование.

```
def mul(x, y):
    return x*y

cur_mul = lambda x: lambda y: mul(x, y)

print(cur_mul(2)(3)) # 6

def mul2(x):
    return x*2
print(mul2(3))  # 6

# сравним с замыканием

def closure_mul(x):
    def mul(y):
        return x * y
    return mul

print(closure_mul(2)(3))  # 6
```

Визуализация концептов функционального программирования

The purest coding style, where bugs are near impossible - YouTube

Лабораторная работа №3: Элементы декларативного и функционального программирования в Python

Общие требования ко всем программам:

1. **Функциональный стиль:** максимально использовать функции высшего порядка (`map` , `filter` , `reduce`), генераторы и выражения вместо императивных конструкций.
2. **Чистота функций:** стремиться к созданию чистых функций без побочных эффектов.
3. **Обработка ошибок:** корректно обрабатывать исключения при работе с файлами и некорректными данными.
4. **Декомпозиция:** разделять решение на небольшие, переиспользуемые функции.

Задание 1: Обработка данных о странах в функциональном стиле

Требования к реализации:

1. Загрузите список стран из файла `countries.json`
2. Используя `map()` , создайте новый список, преобразовав названия всех стран к верхнему регистру
3. Используя `filter()` , отфильтруйте страны:
   - Содержащие подстроку `'land'`
   - Имеющие ровно 6 символов в названии
   - Содержащие 6 и более букв
   - Начинающиеся с буквы `'E'`
4. Используя `reduce()` , объедините страны Северной Европы в строку: "Финляндия, Швеция, Дания, Норвегия и Исландия являются странами Северной Европы"
5. Реализуйте те же задачи без использования встроенных функций высшего порядка (только генераторы и циклы)
6. Используя каррирование (через `lambda`), создайте функцию `categorize_countries()` , которая возвращает список стран по заданному шаблону (`'land'` , `'ia'` , `'island'` , `'stan'`)
7. Реализуйте ту же функциональность через замыкания

```
# Пример каррирования
categorize_curried = lambda pattern: lambda countries: [c for c in countries if
pattern in c]
land_countries = categorize_curried('land')(countries_list)
```

8. Используя файл `countries-data.json` , реализуйте в функциональном стиле:

- Сортировку стран по названию, столице и населению
- Поиск 10 самых распространенных языков и стран, где на них говорят
- Вывод 10 самых населенных стран

```
[
  "Afghanistan",
  "Albania",
  "Algeria",
  "Andorra",
  "Angola",
```

```
  "Antigua and Barbuda",
  "Argentina",
  "Armenia",
  "Australia",
  "Austria",
]
```


```
[
    {
        "name": "Afghanistan",
        "capital": "Kabul",
        "languages": [
            "Pashto",
            "Uzbek",
            "Turkmen"
        ],
        "population": 27657145,
        "flag": "https://restcountries.eu/data/afg.svg",
        "currency": "Afghan afghani"
    },
    {
        "name": "Åland Islands",
        "capital": "Mariehamn",
        "languages": [
            "Swedish"
        ],
        "population": 28875,
        "flag": "https://restcountries.eu/data/ala.svg",
        "currency": "Euro"
    },
    {
        "name": "Albania",
        "capital": "Tirana",
        "languages": [
            "Albanian"
        ],
        "population": 2886026,
        "flag": "https://restcountries.eu/data/alb.svg",
        "currency": "Albanian lek"
    },
]
```


Задание 2: Генератор координат и графов для задачи коммивояжера и ее решение в 3D

Требования к реализации:

1. Реализуйте генератор случайных точек в трехмерном евклидовом пространстве:

```
def generate_points(n: int, bounds: tuple = (0, 100)) -> Generator[tuple,
None, None]:
```

```
        pass
```

2. Реализуйте генератор "дорог" между точками:
    - Поддерживайте как однонаправленные, так и двунаправленные связи
    - Генерируйте расстояния между точками на основе евклидовой метрики
3. Используйте генераторы для создания наборов данных разного размера (200, 500, 1000 точек)

```
# Пример использования
points = list(generate_points(100))
roads = list(generate_roads(points, bidirectional_ratio=0.7))
```

4. Реализуйте основные компоненты муравьиного алгоритма (Муравьиный алгоритм — Википедия, Муравьиный алгоритм | Задача коммивояжёра / Хабр, Муравьиный алгоритм. Решение задачи коммивояжера / Хабр):
    - Функцию расчета вероятности перехода между точками
    - Функцию обновления феромонов
    - Функцию испарения феромонов
5. Используйте `functools.partial` для создания специализированных версий функций
6. Примените `map()` и `filter()` для обработки путей муравьев
7. Реализуйте основную логику алгоритма с использованием генераторов:

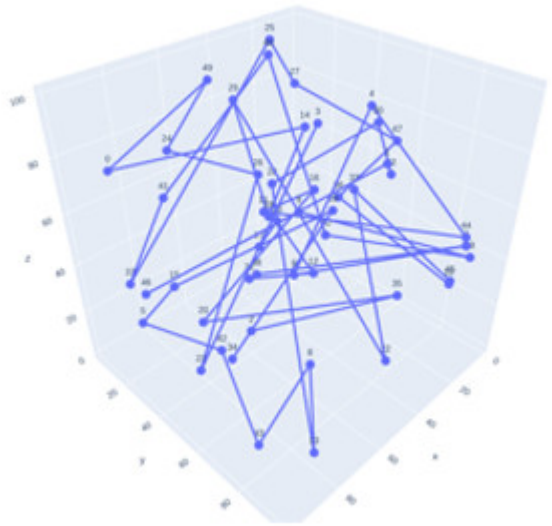
```
def ant_colony_optimization(points, iterations=100):
    pheromone = initialize_pheromones(points)

    for iteration in range(iterations):
        # Генератор путей для каждого муравья
        paths = (construct_path(pheromone, points) for _ in range(len(points)))
        best_path = min(paths, key=calculate_path_length)
        pheromone = update_pheromones(pheromone, best_path)
        yield best_path, calculate_path_length(best_path)
```

8. Реализуйте асинхронную корутину для визуализации процесса работы алгоритма:

```
async def visualize_optimization(algorithm_generator):
    fig = setup_plot()
    async for path, length in algorithm_generator:
        await update_plot(fig, path, length)
        await asyncio.sleep(0.1)  # Контроль скорости анимации
```

9. Используйте библиотеку `matplotlib` или `plotly` для отображения:
    - Трехмерного графа точек
    - Текущего лучшего маршрута
    - Динамики изменения длины маршрута
10. Реализуйте интерактивные элементы управления с помощью `asyncio`*

Что примерно должно получиться:



Задание 3: Сравнительный анализ и метрики для алгоритма

Требования к реализации:

1. Создайте декоратор для измерения времени выполнения функций:

```
def timing_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} executed in {end-start:.4f} seconds")
        return result
    return wrapper
```

2. Используйте `functools.lru_cache` для мемоизации тяжелых вычислений
3. Реализуйте pipeline обработки данных с помощью композиции функций:

```
def compose(*functions):
    return reduce(lambda f, g: lambda x: f(g(x)), functions)

analysis_pipeline = compose(
    normalize_data,
    calculate_metrics,
    generate_report
)
```

4. Проведите сравнительный анализ алгоритма на наборах данных разного размера (200, 500, 1000)