Парадигмы программирования, функциональная парадигма

Якупов Павел

Парадигмы программирования

Парадигмы программирования

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером

Начнем с цитаты

“Когда я размышляю о функциональных языках программирования, я в основном думаю о языках с невероятно мощным компилятором, таким как у HASKELL. Для таких компиляторов функциональная парадигма очень полезна, так как предоставляет огромный массив возможностей по трансформации, включая параллелизацию. Но компилятор Python понятия не имеет о том, что означает ваш код. И это тоже полезно. Поэтому я не считаю, что есть смысл в добавлении «функциональных» примитивов в Python, потому что причины, по которым эти примитивы хорошо показали себя в функциональных языках, не подходят к языку Python. А ещё код из-за них становится крайне нечитабельным для людей не привыкших к функциональным языкам (а таких большинство среди программистов)»

Гвидо ван Россум

Основные парадигмы программирования

Всего в программировании выделяют две крупные парадигмы программирования: **императивная** и **декларативная**.

Императивное программирование предполагает ответ на вопрос “Как?”. В рамках этой парадигмы вы задаете последовательность действий, которые нужно выполнить, для того чтобы получить результат. Результат выполнения сохраняется в ячейках памяти, к которым можно обратиться в последствии.

Декларативное программирование предполагает ответ на вопрос “Что?”. Здесь вы описываете задачу, даете спецификацию, говорите, что вы хотите получить в результате выполнения программы, но не определяете, как этот ответ будет получен.

Основные парадигмы программирования

В коммерческом программировании наибольшее распространение получили **структурное** и **объектно-ориентированное** программирование из группы “императивное программирование” и **функциональное** программирование из группы “декларативное программирование”.

Основные парадигмы программирования

В рамках структурного подхода к программированию основное внимание сосредоточено на декомпозиции — разбиении программы/задачи на отдельные блоки / подзадачи. Разработка ведётся пошагово, методом “сверху вниз”. Наиболее распространенным языком, который предполагает использование структурного подхода к программирования является язык C (и его сынуля Golang), в нем, как правило, основными строительными блоками являются функции или отдельные структуры данных, широко используются циклы.


C
Golang

Основные парадигмы программирования

В рамках объектно-ориентированного (ООП) подхода программа представляется в виде совокупности объектов, каждый из которых является экземпляром определенного класса, и классы могут образовывать иерархию наследования. ООП базируется на следующих принципах (столпах): инкапсуляция, наследование, полиморфизм, абстракция. В качестве примера классов, в которых ООП парадигма развита особенно сильно, приводят C#, Java и C++. В Python ООП парадигма тоже развита достаточно сильно, но часто подпадает под огонь критики и все-таки часть стандартных ООП практик в Python не реализовано.

Основные парадигмы программирования

ООП мы обсудим с вами несколько позже, а пока можно поговорить про функциональное программирование. В
рамках функционального программирования выполнение программы — это процесс вычисления, который трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). Языки, которые были созданы для полноценной реализации этой парадигмы— это Haskell, Lisp, ~~JavaScript,~~ Java(только начиная в основном с 8 версии), и многие другие.

Основные парадигмы программирования

Хорошо, я собираюсь создавать мультипарадигмальный язык программирования, т.е. он должен поддерживать сразу несколько парадигм, если собирается стать известным и популярным коммерческим языком программирования. Какими качествами должен обладать язык, чтобы быть признанным как функциональный? В нем просто должны быть функции? Вовсе нет. Функции есть практически в любых высокоуровневых языках программирования. Но в языках с развитой функциональной парадигмой есть ряд особенностей, которые и позволяют носить им это гордое звание:

Основы функционального программирования

- Функции являются объектами первого класса (First Class Object). Это означает, что с функциями вы можете работать, также как и с данными: передавать их в качестве аргументов другим функциям, присваивать переменным и т.п.
- Рекурсия является основным подходом для управления вычислениями, а не циклы и условные операторы.
- Используются функции высшего порядка (High Order Functions). Функции высшего порядка — это функций, которые могут в качестве аргументов принимать другие функции.
- Функции являются “чистыми” (Pure Functions) — т.е. не имеют побочных эффектов (иногда говорят: не имеют сайд-эффектов).
- Акцент на том, что должно быть вычислено, а не на том, как вычислять.

Основные парадигмы программирования

Python не является языком, который был задуман только для для функционального программирования. Однако его возможностей вполне хватает для того, чтобы писать программы в функциональном стиле. ~~Лучше только в JavaScript, который в обновленной версии все забирает у старшего брата.~~

Функции в Python

Области видимости

При создании пользовательских функций необходимо понимать принцип доступности переменных в программе (область видимости переменных).

К переменным, которые созданы вне функции, можно обращаться из инструкций внутри функций, они являются глобальными.

К переменным, которые создаются внутри функции, нельзя обращаться извне — они имеют локальную область видимости.

Такая логика есть практически во всех современных языках программирования. Не иметь ее это нумеровать массив с 1 .

Локальные и глобальные области видимости

Ограниченная доступность локальных переменных означает, что переменные с одним и тем же именем без каких-либо последствий могут появляться в различных функциях.

Если вы хотите, чтобы к локальной функции был доступ из любого места, ее нужно сначала объявить с использованием ключевого слова **global**, после которого следует имя переменной. После этого ей можно присваивать значение, сколько угодно раз, и оно будет доступно из любого места программы. В тех случаях, когда две переменные — глобальная и локальная — имеют одно имя, функция будет использовать локальную версию.

Простой вариант

```
x = 10


def print_x():
    x = 12
    print(x)


print(x)
print_x()
#сначала у нас появится 10, затем 12
#здесь все довольно просто
```

Работа с global

```
y = 12


def print_y():
    global y
    y = 14
    print(y)
print(y)  # получим 12
print_y() #получим 14
#попробуйте теперь закомментировваться перменную y на 12 строчке
#и поменять функцию print_y и просто print местами
```

Проблема глобальных переменных

Практически во всех языках программирования считается, что глобальные переменные — ~~чистое~~ зло. Глобальные переменные несут в себе множество проблем, кроме того, что через 10000 строк вы начинаете путаться в именах переменных, и без дробления на модули программа может начинаться схлопываться (это переменная была инициализирована? или нет) И пускай Python был спроектирован таким образом, чтобы не допустить этого(модульная структура функций, namespace) и прочее, глобальные переменные все равно занимают область оперативной памяти, потому что не удаляются по окончанию работы. За это отвечают автоматический сборщик мусора.

Сборщик мусора Python

Мы не будем здесь подробно останавливаться на том, что такое сборщик мусора. Простыми словами- это определенный конструкт в интерпретаторе языка, который удаляет данные из тех функций, которые уже завершили свою работу. В языке С, например, его надо вызывать вручную. Несмотря на то, что проблема глобальных переменных остро стоит в менее совершенных с точки синтаксиса и рантайма языках (C++, JavaScript), в Python она тоже поднимается. Полезные ссылочки:

https://www.rupython.com/465-465.html – о проблеме глобальных переменных вообще.

https://habr.com/ru/company/wunderfund/blog/328404/ - статья как Инстаграмма пробки из щитка Python доставал.

Присвоение функции в качестве переменной

```
def mul5(value):
    return value ** 5


first_number = 15
func_1 = mul5
print(func_1(first_number))
```

Рекурсия

Рекурсия – это вызов функции внутри ее же самой. Приведем пример рекурсивной функции, которая считает факториал. Если вы забыли, что такое факториал, то факториал натурального числа это произведение всех натуральных чисел от 1 до n включительно.

К примеру, 5! = 1 * 2 * 3 * 4 * 5 = 120

Рекурсия

Программа вычисления факториала с помощью рекурсии:

```
def fact_rec(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * fact_rec(n - 1)
print(fact_rec(5)) #120
```

Рекурсия

Однако в большинстве языков рекурсия так сконструирована, что циклы чаще всего куда быстрее ее и меньше прожирают память. Однако это не означает, что рекурсию вообще не стоит использовать, однако лучше использовать такую ее реализацию, как хвостовая рекурсия:

Хвостовая рекурсия

Иногда используют ее итеративную реализацию (она несколько проще), однако мы сразу посмотрим на ее ее хвостовой рекурсий. В ней не сохраняется состояние, и работает она несколько быстрее, чем ее более простой аналог:

```
def fact_line_iter_proc(n):
    return fact_iter(1, 1, n)

def fact_iter(prod, counter, n):
    if counter > n:
        return prod
    else:
        return fact_iter(counter * prod, counter + 1, n)

print(fact_line_iter_proc(5))#выдаст все те же 120
```

Бенчмарки

Бенчмарки

Бенмаркинг - контрольная задача, которая определяет производительность операционной системы или любых выполняемых программ (т.к. как они являются частью операционной системы в прикладной ее части). Бенчмаркинг широко используется в highload проектах, в которых очень понимать скорость работы программы, потому что каждая милисекундочка на счету.

Бенчмарки

Для бенчмаркинга мы используем простой пакет на Python, который посвящен именно бенчмаркингу. Он называется simple-benchmark. Поставить его можно с помощью следующей команды:

```
pip install simple_benchmark
```

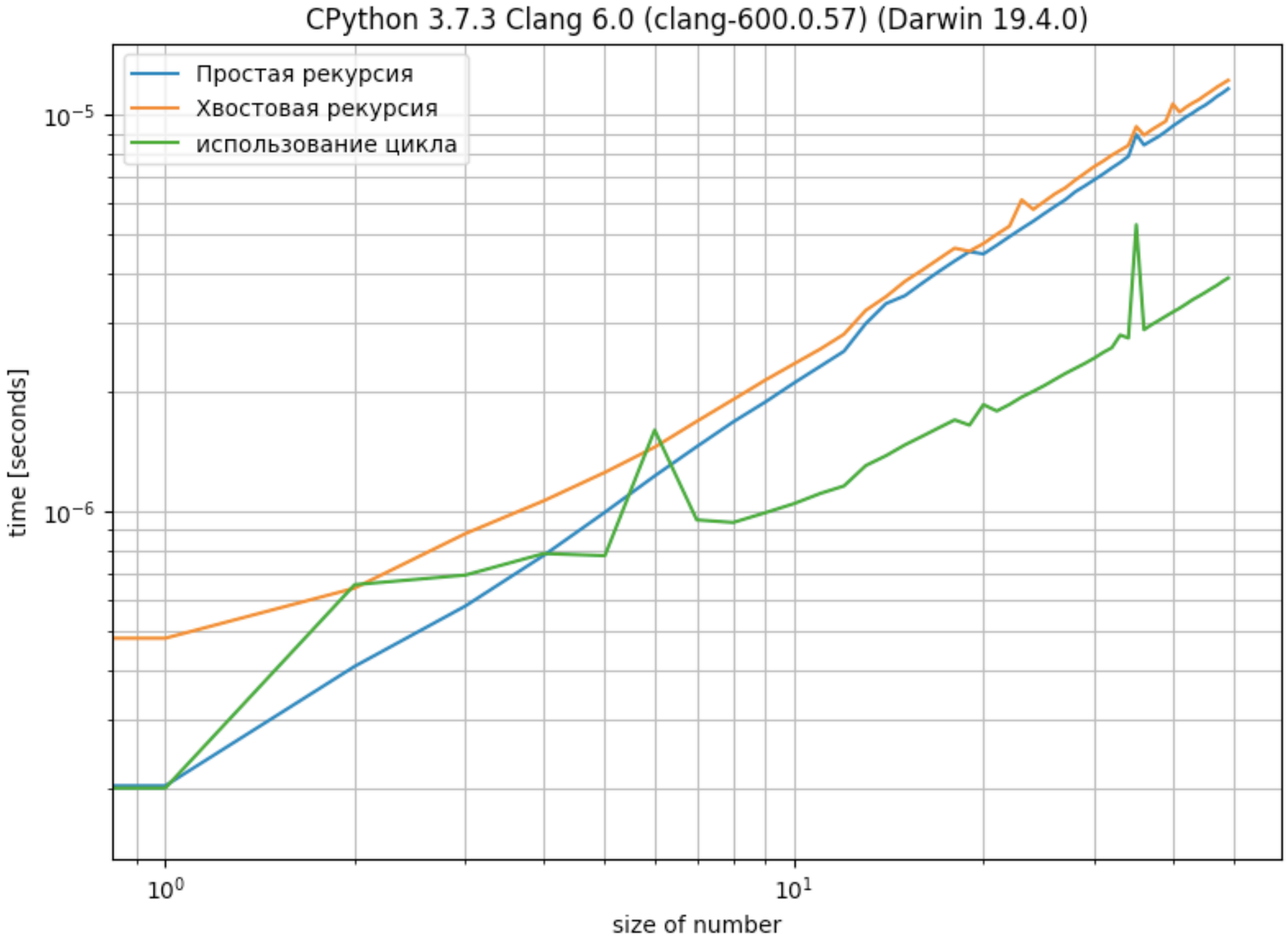
Страница самой библиотеки:

https://pypi.org/project/simple-benchmark/

Бенчмарки

Мы создадим словарь наших аргументов, которые могут быть использованы для теста. После этого мы создадим наши тесты. Только будьте готовы к неожиданным результатам…

Результаты наших бенчмарков



Лямбда функции

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью ключевого слова lambda. Кроме этого, их не обязательно присваивать переменной для того, чтобы передать в качестве аргумента, достаточно просто передать их в качестве переменной. Еще лямба-функции широко используются для обработки целых массив или многомерных массивов, к примеру, в numpy:

Лямбда функции

Лямбда функции

```
func = lambda x, y: x + y
print(func(1, 2))
#3
print(func('a', 'b'))
# 'ab'
print((lambda x, y: x + y)(1, 2))
# 3
print((lambda x, y: x + y)('a', 'b'))
# 'ab'
```

*args и **kwargs спешат на помощь

В Python можно передать неопределенное количество аргументов двумя способами:

- *args для неименованных аргументов;
- **kwargs для именованных аргументов

*args

*args нужен, когда мы хотим передать неизвестное количество неименованных аргументов. Если поставить * перед именем, это имя будет принимать не один аргумент, а несколько. Аргументы передаются как кортеж и доступны внутри функции под тем же именем, что и имя параметра, только без *.

*args

```
def adder(*nums):
    sum = 0

    for n in nums:
        sum += n

    print("Sum: ", sum)


adder(3, 5)
adder(4, 5, 6, 7)
adder(1, 2, 3, 5, 6)
```

**kwargs

По аналогии с *args мы используем **kwargs для передачи переменного количества именованных аргументов. Схоже с *args, если поставить ** перед именем, это имя будет принимать любое количество именованных аргументов. Кортеж/словарь из нескольких переданных аргументов будет доступен под этим именем. Например:

**kwargs

```
def intro(**data):
    print("\nData type of argument: ",type(data))

    for key, value in data.items():
        print("{} is {}".format(key, value))

intro(Firstname="Petr", Lastname="Ivanov", Age=22,
Phone=1234567890)
intro(Firstname="Semen", Lastname="Semenov",
Email="semen@example.com", Country="Pussia",
Age=25, Phone=9876543210)
```

Map

Принимает функцию и набор данных.
Создаёт новую коллекцию, выполняет функцию на каждой позиции данных и добавляет возвращаемое значение в новую коллекцию.
Возвращает новую коллекцию.

```
main_data = ['Маша', 'Ян', 'Вася']

name_lengths = map(len, main_data)

print(list(name_lengths))
```

Filter

В языке программирования Python есть встроенная функция filter(), которая принимает два параметра и возвращает объект-итератор. Первый аргумент этой функции - какая-либо другая функция, а второй - последовательность (строки, списки и кортежи), итератор или объект, поддерживающий итерацию.
Функция filter() возвращает итератор, состоящий из тех элементов последовательности, для которых переданная в качестве первого аргумента функция вернула истину (true) или ее аналог (не ноль, не пустую строку, не None).
В примере ниже создается функция func(), которая возвращает 1, если ей передан аргумент больше нуля, и 0 во всех остальных случаях. Когда эту функцию применяют для списка a, то получается объект-итератор из положительных элементов a.

Забавные истории

Почитать немного про биографию Гвидо ван Россума и его суждения о Python можно по следующей ссылке:

https://www.academia.edu/29612806/%D0%93%D0%B2%D0%B8%D0%B4%D0%BE_%D0%B2%D0%B0%D0%BD_%D0%A0%D0%BE%D1%81%D1%81%D1%83%D0%BC