# JIT-компилятор Python в 300 строк

Может ли студент второго курса написать - компилятор Питона, конкурирующий по производительности с промышленным решением? С учётом того, что он это сделает за две недели за зачёт по программированию.

Как оказалось, может, но с нюансами.

## 1 Предисловие

Обучаясь в **РТУ МИРЭА** , на специальности программная инженерия я попал на семестровый курс программирования на Питоне. Питон я знал до этого, поэтому не хотелось много с ним возиться. Благо творчество студентов поощряется, иногда даже "автоматами". Собственно, стимулируемый этим "автоматом" и тягой к написанию системных модулей я написал JIT-компилятор, который назвал *MetaStruct* .

С кодом проекта можно ознакомиться в .

в написании низкоуровневых программ оказался нежизнеспособным и весьма поучительным. Но об этом сегодня речь не пойдёт.

Стандартная реализация Python - CPython - достаточно медленная. В сравнении с C++ называют замедление в . Но целое сообщество программистов на Питоне готовы заплатить эту цену ради удобства синтаксиса, быстроты написания, изящности и выразительности кода.

На этой почве появляются разнообразные способы оптимизации выполнения программ на Питоне. Такие диалекты как , пытаются решить проблему "разгона" Питона за счёт статической типизации и компиляции модулей.

В области JIT-компиляции промышленным решением является проект , спонсируемый такими технологическими гигантами как Intel, AMD и NVIDIA. Именно с этим пакетом мне предложили и посоревноваться, написав миниатюрный JIT-компилятор программ на Питоне.

В этой статье я хочу рассказать, с какими трудностями я, как программист достаточно прикладной, столкнулся при написании такой довольно низкоуровневой вещи, как миниатюрный JIT-компилятор.

## 2 Принцип работы

На схеме выше показано, какие этапы проходит функция на Питоне, становясь скомпилированным модулем на С++:

@jit  
def sum(x: int, y: int) -> int:  
 res: int = x + y  
 return res

extern "C" int sum(int x, int y) {  
 int res = (x + y);  
 return res;  
}

g++ -O2 -c source.cpp -o object.o  
g++ -shared object.o -o lib.dll

Процесс достаточно трудоёмкий для функции сложения из примера, но при частых вызовах и большом количестве вычислений внутри функции время компиляции окупается.

Если бы это был не Питон, а какой-нибудь предметно-ориентированный язык, то пришлось бы писать парсер и обход получившегося абстрактного дерева, и решение не было бы уже таким коротким. Но в моём случае, инфраструктура Питона и его гибкость сыграли мне на руку.

## 3 Впечатляющие результаты

Наверное, стоит от технической части переходить к части визуализации и маркетинга.

Созданный алгоритм JIT-компиляции был протестирован на нескольких простых алгоритмических задачах:

С расчётами и графиками можно подробнее ознакомиться в

Для оценки времени выполнения использованы функции timeit() и repeat() модуля timeit. Для отрисовки графиков - модуль matplotlib

В примерах будут сравниваться три реализации функций:

### 3.1 Сумма двух чисел

def py\_sum(x: int, y: int) -> int:  
 res: int = x + y  
 return res

На задаче сложения двух целых чисел никакой оптимизации не видно, даже наоборот. Накладные расходы на вызов функции из dll-файла и обработка результата занимает много времени по сравнению с самими расчётами. Numba обставила моего "питомца" в 3 раза на этом примере.

### 3.2 Хеш-функция для целых чисел

Обычно, для чисел из небольшого диапазона в качестве хеша используют их самих. Однако на просторах Интернета я нашёл такую :

def py\_hash(x: int) -> int:  
 x = ((x >> 16) ^ x) \* 0x45d9f3b  
 x = ((x >> 16) ^ x) \* 0x45d9f3b  
 x = (x >> 16) ^ x  
 return x

Автором сообщения утверждается, что значение параметра 0x45d9f3b позволяет достичь наибольшей "случайности" бит внутри числа. По крайней мере для хеш-функций такого вида.

Numba оказалась хорошо оптимизированной под битовые операции. Не совсем понятно, откуда она взялась. Оставим этот вопрос открытым, но мне кажется, спонсорство главных производителей процессоров и видеокарт не прошло даром. Мой же вариант оказался слегка быстрее простого Питона.

### 3.3 Вычисление экспоненты через ряд Тейлора

Питон явно показал себя неважно, поэтому посмотрим на двух оптимизаторов отдельно.

Странное большое время для маленького x, выяснилось, обосновано тем, что Numba делает какие-то отложенные шаги компиляции при первом запуске. На общей её производительности это почти никак не сказывается.

def py\_exp(x: float) -> float:  
 res: float = 0  
 threshold: float = 1e-30  
 delta: float = 1  
 elements: int = 0  
 while delta > threshold:  
 elements = elements + 1  
 delta = delta \* x / elements  
 while elements >= 0:  
 res += delta  
 delta = delta \* elements / x  
 elements -= 1  
 return res

Кому интересен матан, экспонента считается по формуле соответствующего ряда Тейлора:

Алгоритм прекращается, когда разница между дельтами двух итераций становится меньше порога, либо превращается в машинный ноль. Суммирование происходит от меньших членов к большим для уменьшения потерь точности.

Наконец-то моё творение начало соперничать с Numba. На больших объёмах вычислений однозначного лидера нет.

### 3.4 Числа Фибоначчи

def fib(n: int) -> int:  
 if n < 2:  
 return 1  
 return fib(n - 1) + fib(n - 2)

Несмотря на то, что аннотация позволяет компилировать функции по одной, в ней всё ещё можно использовать рекурсию.

На рекурсии Питон вообще перестал за себя отвечать. Что там с оптимизаторами?

Внезапно, реализованная в проекте компиляция начала работать в 4 раза быстрее, чем Numba. Получается, что с задачами разветлённой рекурсии мой JIT-компилятор неплохо справляется.

Это одно из самых интересных мест всего исследования, которе можно было бы продолжить.

## 4 Мысли сходятся

На самом деле, такой подход к оптимизации не нов в мире программирования. Чем-то похожим Владимир Макаров, оптимизируя Ruby до уровня языка передачи регистров в своём проекте MJIT.

Существуют даже оптимизации сделанные поверх решения Макарова, о которых можно почитать .

В частности, в исследованиях отмечается, что выбор компилятора, будь то GCC или LLVM, существенно не сказывается на производительности. В моём решении использован g++ из-за большей портируемости скомпилированного кода.

Для ускорения вычислений в проекте Ruby используются также предскомплированные заголовки. Однако, для студенческой работы такой уровень оптимизации не требуется.

## 5 Непредвиденные трудности

Конечно же, всё заработало не с первого раза. Вероятно, даже не с десятого. Поэтому хотелось бы привести здесь небольшую "работу над ошибками"

### 5.1 Типы данных

Питон медленный во многом из-за динамической типизации, так как довольно много времени уходит на определение типа переменной перед её использованием. Также, идеология "всё есть объект" раздувает примитивные типы данных до размера остальных объектов и классов. Чтобы ускорить вычисления, нужно использовать именно примитивы, а не объекты.

Проблема в том, что из кода на Питоне не всегда очевидно, какого типа будет переменная. Продвинутые оптимизаторы умеют определять тип переменной "на лету" из контекста. Я решил не усложнять жизнь и использовать аннотации типов.

Про использование аннотаций типов есть хорошие статьи в или на .

Пусть нужно скомпилировать ту самую функцию сложения:

def sum(x, y):  
 res = x + y  
 return res

Какой-нибудь компилируемый язык, C++ например, за такую программу спасибо не скажет. Добавим аннотации:

def sum(x: int, y: int) -> int:  
 res: int = x + y  
 return res

В этом примере явного объявления типов требуют три вещи:

В базовой реализации будет только три типа данных:

Использование более сложных типов данных выходит за задачу миниатюрности компилятора. Но стоит отметить, что строки, коллекции и объекты Питона могут быть поддержаны с использованием

### 5.2 Кодирование имён функций

С++ видоизменяет названия функций согласно их сигнатуре и аргументам. Например, функцию int f(int x) компилятор может преобразовать в \_Z1fi. Подробнее о соглашении именования функции при компиляции можно узнать, например,

После переименования к функциям уже нельзя обратиться по первоначальному названию. Конечно, можно было бы написать свой алгоритм, который делает те же преобразования, что и компилятор. Но на самом деле, существует более простое решение, к которому мне пришлось в итоге прийти.

При добавлении к объявлению функции префикса extend &quot;C&quot; имена не будут кодироваться:

extern "C" int sum(int x, int y) {  
 int res = (x + y);  
 return res;  
}

Так происходит, потому что мы явно указываем, что имена функций должны кодироваться по соглашению языка C, то есть, никак.

### 5.3 Запуск DLL

Как программисту, плохо знакомому с чем-то ниже C++, мне было трудно понять, как подключить собранную dll-библиотеку к Питону. Изначально была идея использовать rundll32.exe для запуска. Почитав и я немного разочаровался в прикладной применимости и портируемости этого решения.

Потом я нашёл статью с говорящим названием:

Только после этого я был направлен на путь истинный, а точнее, на использование модуля ctypes. В этот момент мои скомпилированные функции впервые начали возвращать мне значения прямо в Питоне.

### 5.4 Вызов функции из DLL

Всё шло гладко, пока я оперировал целыми числами. При попытке считать дробные... возвращались целые очень странного формата.

# просто питон  
print(exp(0.1))   
# >>> 1.1051709180756475  
  
# моё чудо  
print(jit\_exp(0.1))   
# >>> -1285947181

Как оказалось, по умолчанию, все функции, импортируемые из dll работают с аргументами как с int и возвращают тоже int. Тут до меня дошло, что возвращать просто строку программы на C++ из транслятора недостаточно. Из функций необходимо ещё получить сигнатуру, чтобы потом применить нужные типы из модуля ctypes. Метод посещения объявления функции начал выглядеть вот так:

def visit\_FunctionDef(self, node: FunctionDef) -> Tuple[str, dict]:  
 ret\_type = self.visit(node.returns)  
 name = node.name  
 args, args\_signature = [], []  
 for arg in node.args.args:  
 arg, arg\_type = self.visit(arg)  
 args.append(f"{arg\_type} {arg}")  
 args\_signature.append(ctype\_convert(arg\_type))  
 args = ", ".join(args)  
 res = f"extern \"C\" {ret\_type} {name}({args}) {{\n"  
 res += self.dump\_body(node.body) + "}"  
 signature = {  
 "argtypes": args\_signature,  
 "restype": ctype\_convert(ret\_type)  
 }  
 return res, signature

Появился метод встраивания этих типов в сигнатуру dll:

def jit(func: Callable) -> Callable:  
 exec\_module, signatures = compile\_dll(func)  
 name = func.\_\_name\_\_  
 jit\_func = exec\_module[name]  
 jit\_func.argtypes = signatures[name]["argtypes"]  
 jit\_func.restype = signatures[name]["restype"]  
 return jit\_func

Для того чтобы функции вызывались так же хорошо, как с целыми числами, достаточно было поменять у функций внутри загруженного dll атрибуты argtypes и restype. Сами типы для подстановки были взяты из модуля ctypes:

def ctype\_convert(type\_str: str):  
 match type\_str:  
 case "int":  
 return ctypes.c\_int  
 case "double":  
 return ctypes.c\_double  
 case "bool":  
 return ctypes.c\_bool  
 case \_:  
 raise Exception(f"unsupported type str {type\_str}")

После всех этих махинаций ответ экспоненты начал совпадать с Питоном.

Подробнее про использование функций, загруженных из DLL, можно почитать в этой

## 6 Немного магии

Хороший код не пишется сразу. Например, в моём проекте основные блоки пришлось переписать два раза. В крупных проектах борьба с техдолгом вообще может уходить в бесконечность, но я пока что, к сожалению или к счастью, с этим не сталкивался.

Я хотел показать на примере функции синтаксического разбора бинарной операции ast.BinOp как можно по-разному писать код, который будет в разной степени сложно поддерживать.

### 6.1 Словарь с типами

Это самая первая реализация:

def dump\_bin\_op(module: ast.BinOp) -> str:  
 res = ""  
 left = dump\_expr(module.left)  
 right = dump\_expr(module.right)  
 op = module.op  
 bin\_op\_signs = {  
 ast.Add: "+",  
 ast.Sub: "-",  
 ast.Div: "/",  
 ast.Mult: "\*",  
 # и ещё 6-8 бинарных операций  
 }  
 op\_sign = bin\_op\_signs[type(op)]  
 return f"({left} {op\_sign} {right})"

По сравнению с блоком if-elif-elif-elif-...-else такой код кажется проще. Но тут происходит явное обращение к типу через type(), что не очень хорошо.

### 6.2 Match/case

Тут мне посоветовали начать уже использовать плюшки версии Питона 3.10 на полную катушку, а именно, применить сравнение по шаблону и оператор match/case

def dump\_bin\_op(module: ast.BinOp) -> str:  
 match module:  
 case ast.BinOp(op=ast.Add()):  
 op\_sign = "+"  
 case ast.BinOp(op=ast.Sub()):  
 op\_sign = "-"  
 case ast.BinOp(op=ast.Div()):  
 op\_sign = "/"  
 case ast.BinOp(op=ast.Mult()):  
 op\_sign = "\*"  
 # и ещё 6-8 бинарных операций  
 case \_:  
 raise Exception(f"unsupported bin op type {op\_type}")  
 left = dump\_expr(module.left)  
 right = dump\_expr(module.right)  
 return f"({left} {op\_sign} {right})"

Код стал чуточку короче и выразительнее. Шаблоны после слова case можно всячески усложнять, выбирая всё более специфичные случаи. Какой-нибудь switch/case в другом языке программирования такого себе не может позволить. Подробнее про возможности оператора match/case можно узнать

Вот ещё маленький пример, решающий проблему занижения регистра для констант True и False:

def visit\_Constant(self, node: Constant) -> str:  
 match node:  
 case Constant(value=True):  
 return "true"  
 case Constant(value=False):  
 return "false"  
 case \_:  
 return str(node.value)

Выглядит покрасивее, чем if node.value == True:. Но тут уже на вкус и цвет.

### 6.3 ast.NodeVisitor

Вот я и пришёл к тому, что давно написали за меня, но я об этом никогда не слышал, поэтому ещё не применил.

Как ни странно, модуль, описывающий абстрактное дерево в виде структуры, предоставляет также и методы для его обхода. Этот метод называется ast.NodeVisitor.visit()

Как следует из названия, NodeVisitor реализует шаблон проектирования , позволяющий создавать новую внешнюю функциональность с минимальным изменением уже написанного кода.

Для написания своего посетителя необходимо объявить класс-наследник класса ast.NodeVisitor:

class DumpVisitor(ast.NodeVisitor):  
 ...  
 def visit\_BinOp(self, node: ast.BinOp) -> str:  
 return f"({self.visit(node.left)} {self.visit(node.op)} {self.visit(node.right)})"  
  
 def visit\_Add(self, node: ast.Add) -> str:  
 return "+"  
  
 def visit\_Sub(self, node: ast.Sub) -> str:  
 return "-"  
  
 def visit\_Div(self, node: ast.Div) -> str:  
 return "/"  
  
 def visit\_Mult(self, node: ast.Mult) -> str:  
 return "\*"  
  
 # ещё 6-8 методов обхода бинарных операций  
 ... # и не только

Код остальных методов посетителя можно изучить в .

Код стал более модульным, названия и сигнатуры методов уже придуманы за нас, выбор нужного метода происходит без нашего участия. Вот на этом варианте я и решил остановиться.

Во время написания кода на Питоне у меня возникает чувство ощущения красоты, краткости и мощи собственного кода. В этом есть какая-то магия.

## 7 Подсчёт строк

Дабы не потерять доверие читателя, я провёл подсчёт строк кода транслятора:

Остальные файлы, как оказалось, к процессу исполнения напрямую не причастны. Я даже не стал исключать того большого числа пустых строк, которого требует PEP8. Получается, что даже немного наврал читателю насчёт числа строк. Пусть он меня простит.

Думаю, такой небольшой проект можно было бы легко поддерживать, если бы в этом была бы необходимость.

## 8 Итог

Хочу сказать спасибо за наставничество при написании этого проекта и этой статьи. Как оказывается, писать статьи труднее, чем писать код в ящик.

Проекту в плане функционала есть куда расти. Поддержка строк, коллекций, объектов, классов. Правда с учётом полученного зачёта, предлагаю энтузиастам, взяв моё решение за основу, добиться большей производительности и функциональности.

Предел у этого совершенства всё равно есть. На поддержке библиотек такие решения оптимизации обычно отказываются работать либо поддерживают самые популярные и базовые, такие как Numpy или PIL.

И всё же, если очень захотеть, можно заставить Питон работать быстрее, уничтожая один из извечных аргументов программистов на C++ и Java против использования Python.