



БИБЛИОТЕКА
ПРОГРАММИСТА

Энтони Шоу



ВНУТРИ СРУТНОН

ГИД ПО ИНТЕРПРЕТАТОРУ PYTHON

 ПИТЕР®

C_Python Internals: Your Guide to the Python 3 Interpreter

Anthony Shaw



БИБЛИОТЕКА
ПРОГРАММИСТА

ВНУТРИ СРУТНОН

ГИД ПО ИНТЕРПРЕТАТОРУ PYTHON

Энтони Шоу



Санкт-Петербург · Москва · Минск

2023

ББК 32.973.2-018.1
УДК 004.43
Ш81

Шоу Энтони

- Ш81 Внутри CPYTHON: гид по интерпретатору Python. — СПб.: Питер, 2023. — 352 с.: ил. — (Серия «Библиотека программиста»).
ISBN 978-5-4461-1925-7

CPython, самая популярная реализация Python, абстрагируется от сложностей ОС и предоставляет платформу для создания масштабируемых и высокопроизводительных приложений. Каждому python-разработчику на какой-то стадии необходимо будет узнать, как работает CPython. Это позволит в полной мере использовать его мощь и оптимизировать приложения. Вы разберетесь с основными концепциями внутреннего устройства CPython и научитесь: читать исходный код интерпретатора CPython и свободно ориентироваться в нем; вносить изменения в синтаксис Python и компилировать их в вашу собственную версию CPython; понимать внутреннюю реализацию таких структур, как списки, словари и генераторы; управлять памятью CPython; масштабировать код Python за счет параллелизма и конкурентного выполнения; дополнять базовые типы новой функциональностью; выполнять наборы тестов; профилировать и проводить бенчмарк Python-кода и исполнительной среды; отлаживать код C и Python на профессиональном уровне; изменять или обновлять компоненты библиотеки CPython, чтобы они могли использоваться в будущих версиях.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с DevAcademy Media Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1775093343 англ.
ISBN 978-5-4461-1925-7

© Real Python (realpython.com)
© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Библиотека программиста», 2022

Оглавление

Об авторе	13
О группе редакторов	13
Предисловие	14
Введение	16
Как использовать эту книгу	17
Дополнительные материалы и учебные ресурсы	18
Загрузка исходного кода CPython.....	21
Что в исходном коде?	22
Настройка среды разработки	24
IDE или редактор?	24
Настройка Visual Studio	25
Настройка Visual Studio Code	28
Настройка Vim.....	36
Выводы	40
Компиляция CPython	41
Компиляция CPython на macOS	42
Компиляция CPython на Linux	44

Установка специализированной версии	46
Знакомство с Make	46
Make-цели CPython	48
Компиляция CPython на Windows.....	51
Профильная оптимизация.....	56
Выводы	58
Грамматика и язык Python.....	59
Почему CPython написан на C, а не на Python	60
Спецификация языка Python	62
Генератор парсеров.....	66
Повторное генерирование грамматики.....	67
Выводы	71
Конфигурация и ввод	73
Конфигурация состояния	75
Структура данных конфигурации среды выполнения	77
Конфигурация сборки.....	79
Сборка модуля из входных данных.....	80
Выводы	85
Лексический анализ и парсинг с использованием синтаксических деревьев.....	87
Генерирование конкретного синтаксического дерева.....	88
Парсер/токенизатор CPython	91
Абстрактные синтаксические деревья.....	96
Важные термины.....	104
Пример: добавление оператора «почти равно»	104
Выводы	109

Компилятор	110
Исходные файлы.....	111
Важные термины.....	111
Создание экземпляра компилятора	112
Флаги будущей функциональности и флаги компилятора.....	114
Таблицы символических имен	116
Основная компиляция.....	121
Ассемблер	128
Создание объекта кода.....	131
Использование Instaviz для вывода объекта кода.....	133
Пример: реализация оператора «почти равно»	134
Выходы	139
Цикл вычисления	140
Исходные файлы.....	141
Важные термины.....	141
Построение состояния потока	142
Построение объектов кадров	143
Выполнение кадра	150
Стек значений	153
Пример: добавление элемента в список.....	157
Выходы	161
Управление памятью	163
Выделение памяти в С	163
Проектирование системы управления памятью Python.....	167
Аллокаторы памяти CPython	169
Область выделения объектной памяти и PyMem	178

Область выделения сырой памяти	181
Нестандартные области выделения памяти.....	181
Санитайзеры выделенной памяти	183
Аренда памяти PyArena	185
Подсчет ссылок.....	186
Сборка мусора.....	192
Выводы	201
Параллелизм и конкурентность	203
Модели параллелизма и конкурентности	205
Структура процесса	205
Многопроцессорный параллелизм.....	208
Многопоточность.....	230
Асинхронное программирование	242
Генераторы.....	243
Сопрограммы	249
Асинхронные генераторы	254
Субинтерпретаторы	255
Выводы	259
Объекты и типы.....	260
Примеры этой главы	261
Встроенные типы	262
Типы объектов	263
Тип type	264
Типы bool и long	268
Тип строки Юникода	272
Словари	282
Выводы	287

Стандартная библиотека	288
Модули Python	288
Модули Python и C	290
Набор тестов	293
Запуск набора тестов в Windows	293
Запуск набора тестов в Linux или macOS	294
Флаги тестирования	295
Запуск конкретных тестов	295
Модули тестирования	296
Вспомогательные средства тестирования	297
Выводы	298
Отладка	299
Обработчик сбоев	299
Компиляция поддержки отладки	300
LLDB для macOS	301
GDB	305
Отладчик Visual Studio	307
Отладчик CLion	309
Выводы	314
Бенчмаркинг, профилирование и трассировка	315
Использование timeit для микробенчмарка	316
Использование набора тестов производительности Python	318
Профилирование кода Python с использованием cProfile	322
Профилирование кода C в DTrace	325
Выводы	330

Что дальше?.....	331
Создание расширений С для CPython	331
Улучшение приложений Python.....	332
Участие в проекте CPython	333
Дальнейшее обучение.....	336
Приложение. Введение в С для Python-программистов	338
Препроцессор С.....	338
Базовый синтаксис С.....	341
Выводы	348
Благодарности.....	349

ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения, вопросы отправляйте на *comp@piter.com* (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства *www.piter.com* вы найдете подробную информацию о наших книгах.

Что говорят читатели о книге «Внутри CPython»

Мне бы очень хотелось, чтобы такая книга была у меня много лет назад, когда я только изучала Python. После чтения этой книги ваша квалификация повысится, и вы сможете решать еще более сложные задачи, которые улучшат наш мир.

*Кэрол Уиллинг,
ключевой разработчик CPython
и участница управляющего совета CPython*

Глава «Параллелизм и конкурентность» — одна из моих любимых. Я уже давно старался глубже разобраться в теме, и ваша книга оказалась чрезвычайно полезной.

Конечно, после чтения этой главы я не смог устоять перед искушением прочитать остальное. С нетерпением жду свой печатный экземпляр, когда он выйдет! Ранее я прочитал вашу статью «Guide to the CPython Source Code», которая разожгла во мне желание больше узнать о внутреннем устройстве CPython.

О Python написаны десятки обучающих книг, но я еще не встречал ни одной, которая объясняла бы самым любознательным тонкости внутреннего устройства.

В настоящее время я сам обучаю языку Python свою дочь, и эта книга входит в ее список обязательного чтения. Сейчас она изучает информационные системы в Университете штата Джорджия.

*Милан Пател,
вице-президент (крупного инвестиционного банка)*

В книге Энтони меня больше всего впечатляет то, как все шаги по внесению изменений в кодовую базу CPython объединяются в доступную и понятную процедуру. Книга воспринимается как своего рода «недостающее руководство». Разобраться в языке C, основе Python, было очень интересно, и я нашел для себя ответы на некоторые давнишние вопросы. Особенно много полезного — в главе, посвященной управлению памятью в CPython. «Внутри

СPython» — превосходный (и уникальный) ресурс для всех, кто желает вывести свои знания Python на более высокий уровень.

Дэн Бейдер,
автор книги «*Python Tricks*»¹ и редактор сайта *Real Python*

Книга помогла мне лучше понять, как работает лексический анализ и парсинг в Python. Я рекомендую ее, если вы действительно хотите разобраться в теме.

Флориан Далиц, питонист

Искрывающее описание внутренних механизмов Python. Как ни странно, по этой теме почти нет хороших источников информации. Материал излагается доступно как для начинающих, так и для опытных пользователей Python.

Абхишек Шарма, специалист по Data Science

¹ Бейдер Д. Чистый Python. Тонкости программирования для профи. — СПб.: Питер.

ОБ АВТОРЕ

Энтони Шоу — заядлый питонист, участник Python Software Foundation.

Энтони занимался программированием с 12 лет. Любовь к Python он обрел спустя 15 лет, когда ему пришлось какое-то время просидеть в отеле в Сиэттле (штат Вашингтон). С тех пор Энтони исследует Python, пишет о нем и создает учебные курсы, забыв обо всех остальных языках, которые он прежде изучал.

Энтони также участвует в малых и больших проектах с открытым исходным кодом, включая CPython, и является участником Apache Software Foundation.

Страсть Энтони — разбираться в сложных системах, упрощать их и обучать других людей.

О ГРУППЕ РЕДАКТОРОВ

Джим Андерсон давно программирует на разных языках. Он работал над встроенными системами, разрабатывал распределенные системы сборки, занимался управлением сторонними исполнителями и участвовал во многих, многих собраниях.

Джоанна Яблонски — редактор сайта Real Python. Естественные языки интересуют ее не меньше, чем языки программирования. Ее любовь к головоломкам, поиску закономерностей и всевозможным мелочам привела к тому, что она выбрала карьеру переводчика. Прошло совсем немного времени, и она влюбилась в новый язык — Python! Джоанна присоединилась к команде Real Python в 2018 году и с тех пор помогает питонистам повышать их профессиональный уровень.

Джейкоб Шмитт уже много лет занимается редактированием академических и технологических образовательных материалов — как в печатном виде, так и в интернете. После присоединения к команде Real Python в 2020 году он редактирует учебники, статьи и книги, написанные разносторонней командой талантливых писателей и разработчиков.

Предисловие

Язык программирования, созданный сообществом, способствует счастью своих пользователей по всему миру.

Гвидо ван Россум, «Речь в День короля»¹

Мне нравится создавать инструменты, которые помогают нам учиться, наделяют нас возможностью творить и стимулируют обмениваться знаниями и идеями с другими людьми. Я испытываю трепет, благодарность и гордость, когда слышу, что эти инструменты и Python помогают в решении проблем реального мира — например, изменения климата или борьбы с болезнью Альцгеймера.

Я полюбила программирование 40 лет назад. Все эти годы я училась, писала много кода и делилась своими идеями с другими. Я видела кардинальные изменения в технологии, когда мир переходил от мейнфреймов к мобильным технологиям и повсеместному распространению чудес Всемирной паутины и облачных вычислений. У всех этих технологий, включая Python, есть нечто общее.

В какой-то момент все эти успешные нововведения были всего лишь идеей. Творцам — таким, как Гвидо, — приходилось идти на риск и действовать наугад, чтобы двигаться вперед. Преданность делу, изучение методом проб и ошибок и совместное преодоление многих неудач заложили прочную основу для успеха и дальнейшего роста.

«Внутри СPython» поведет вас по пути исследования невероятно успешного языка программирования Python. Эта книга поможет разобраться, как

¹ <http://neopythonic.blogspot.com/2016/04/>.

работают внутренние механизмы CPython. Она даст представление о том, как ключевые разработчики создавали язык.

К сильным сторонам Python относится удобочитаемость кода и дружелюбное сообщество, посвятившее себя распространению знаний. Энтони учитывает эти сильные стороны при объяснении CPython, подталкивая вас к чтению исходного кода и объясняя структурные элементы языка.

Почему я хочу поделиться с вами книгой «Внутри CPython»? Мне бы очень хотелось, чтобы такая книга была у меня много лет назад, когда я только изучала Python. Но что еще важнее, я верю, что у нас, участников сообщества Python, есть уникальная возможность применить наш опыт в деле и помочь в решении сложных задач реального мира.

Я уверена, что после чтения этой книги ваша квалификация вырастет и вы сможете решать еще более сложные задачи и улучшать наш мир.

Надеюсь, что вам захочется больше узнать о Python, создать нечто новое, и вы обретете уверенность, чтобы поделиться своими творениями с миром.

«Сейчас лучше, чем никогда».
Tim Peters, «Дзен Python»

Последуйте мудрому совету Тима и беритесь за дело.

С наилучшими пожеланиями,

*Кэрол Уиллинг,
ключевой разработчик CPython
и участница управляющего совета CPython*

Введение

Некоторые операции Python кажутся каким-то волшебством — почему поиск элементов по словарю выполняется намного быстрее, чем перебор по списку? Как генератор запоминает состояние переменных каждый раз, когда он возвращает значение командой `yield`? Почему нам не приходится выделять память, как в других языках?

Дело в том, что CPython, самая популярная реализация Python, написана на удобочитаемых языках С и Python.

CPython абстрагируется от сложностей операционной системы и лежащей в его основе платформы С. В CPython многопоточное выполнение становится прямолинейным и кроссплатформенным. Среда берет на себя все трудности управления памятью и упрощает его.

CPython предоставляет платформу для создания масштабируемых и высокопроизводительных приложений. На какой-то стадии вашего становления Python-разработчиком необходимо будет разобраться в том, как работает CPython. Абстракции не идеальны, и иногда приходится заглядывать внутрь.

А когда вы поймете, как работает CPython, это позволит вам в полной мере использовать его мощь и оптимизировать приложения. В книге объясняются понятия, идеи и технические детали CPython.

Вы разберетесь с основными концепциями внутреннего устройства CPython. В частности, научитесь:

- читать исходный код и ориентироваться в нем;
- компилировать CPython из исходного кода;
- вносить изменения в синтаксис Python и компилировать их в вашу версию CPython;
- понимать внутреннюю реализацию таких структур, как списки, словари и генераторы;

- управлять памятью CPython;
- масштабировать код Python за счет параллелизма и конкурентного выполнения;
- дополнять базовые типы новой функциональностью;
- выполнять наборы тестов;
- профилировать и проводить бенчмарк Python-кода и исполнительной среды;
- отлаживать код С и Python на профессиональном уровне;
- изменять или обновлять компоненты библиотеки CPython, чтобы они могли использоваться в будущих версиях.

Не торопитесь с чтением, опробуйте демоприложения и интерактивные элементы. А когда вы усвоите основные концепции, которые сделают вас более успешным программистом, вы будете вполне оправданно гордиться своими достижениями!

КАК ИСПОЛЬЗОВАТЬ ЭТУ КНИГУ

Вся суть книги — обучение в процессе работы, поэтому обязательно начните с настройки IDE: прочитайте инструкции, загрузите код и напишите примеры.

Чтобы достигнуть наилучших результатов, старайтесь избегать копирования примеров кода. Приведенные в книге примеры неоднократно перерабатывались, чтобы добиться их правильной работы, и они иногда могут содержать ошибки.

Ошибки и умение исправлять их — часть процесса обучения. Возможно, вы найдете более эффективные способы реализации примеров; попробуйте изменить их и посмотрите, как это повлияет на результат.

При достаточной практике вы освоите весь материал. Приятного путешествия!

Нужно ли быть квалифицированным Python-разработчиком для работы с книгой?

Книга предназначена для Python-разработчиков среднего и высокого уровня. Мы старались, чтобы примеры кода были доступными, но в них время от времени встречаются приемы, которые могут быть непонятны начинающим.

Нужно ли знать С для работы с книгой?

Хорошее знание С не обязательно для чтения. Если у вас нет опыта программирования на С, ознакомьтесь с кратким введением в язык в приложении «Введение в С для Python-программистов».

Сколько времени займет чтение книги?

Мы не рекомендуем торопиться с чтением. Попробуйте читать ее по одной главе, практикуйтесь на примерах после каждой главы и одновременно анализируйте код. А когда вы закончите читать книгу, она станет полезным справочным руководством, к которому вы сможете время от времени возвращаться.

Не устареет ли материал книги?

Python существует уже более 30 лет. Некоторые части кода Python не изменились с того момента, когда они были впервые написаны. Многие принципы, описанные в книге, оставались без изменений уже более 10 лет.

Более того, в ходе работы над книгой мы обнаружили, что многие строки кода были написаны Гвидо ван Россумом (автором Python) и остались неизменными с первой версии Python.

Некоторые концепции, представленные в книге, появились совсем недавно. Некоторые даже остаются в экспериментальном статусе. Работая над книгой, мы столкнулись с дефектами в исходном коде и ошибками в CPython, которые позднее были исправлены или доработаны¹. Это часть феномена CPython как энергично развивающегося проекта с открытым исходным кодом.

Навыки, которые вы получите при работе с книгой, помогут вам читать и понимать код текущих и будущих версий CPython. Изменения происходят постоянно, и вместе с ними приходит опыт.

ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ И УЧЕБНЫЕ РЕСУРСЫ

К книге прилагается ряд бесплатных дополнительных ресурсов, доступных по адресу realpython.com/cpython-internals/resources/.

¹ <https://realpython.com/cpython-fixes>.

На этой веб-странице вы также сможете скачать примеры исходного кода.

Там же вы найдете список опечаток с исправлениями, которые ведет группа Real Python.

Лицензия на примеры кода

Сценарии Python, имеющие отношение к книге, распространяются на условиях лицензии CC0 (Creative Commons Public Domain)¹. Это означает, что вы можете свободно использовать в своих программах любые части кода для любых целей.

CPython распространяется на условиях лицензии Python Software Foundation 2.0 (PSF 2.0)². Сниппеты и примеры исходного кода CPython, приведенные в книге, соответствуют PSF 2.0.

ПРИМЕЧАНИЕ

Код, приведенный в книге, был протестирован с Python 3.9 для Windows 10, macOS 10.15 и Linux.

Обратная связь и опечатки

Мы приветствуем идеи, предложения, обратную связь, а порой и критику. Тема показалась вам непонятной? Вы нашли ошибку в тексте или в коде? Мы пропустили тему, о которой вам хотелось бы узнать больше?

Мы всегда рады возможности улучшить свои учебные материалы. Пожалуйста, обращайтесь со своими замечаниями по адресу realpython.com/cpython-internals/feedback.

О Real Python

На сайте Real Python можно изучать реальные навыки программирования в сообществе профессиональных питонистов со всего мира. Веб-сайт

¹ <https://creativecommons.org/publicdomain/zero/1.0/>.

² <https://github.com/python/cpython/blob/master/LICENSE>.

realpython.com был запущен в 2012 году; в настоящее время он помогает более чем трем миллионам Python-разработчиков со всего мира своими книгами, учебниками по программированию и другими материалами.

Источники информации Real Python в интернете:

- *realpython.com*;
- Real Python Newsletter (<https://realpython.com/newsletter>);
- Real Python Podcast (<https://realpython.com/podcast>).

Загрузка исходного кода CPython

Когда вы вводите команду `python` в консоли или устанавливаете дистрибутив Python с сайта [Python.org](https://www.python.org), запускается **CPython**. CPython — одна из многих реализаций Python, разработкой и сопровождением которой занимаются разные команды разработчиков. Возможно, вам также доводилось слышать о таких альтернативах, как PyPy, Cython и Jython.

У CPython есть одна уникальная особенность: эта реализация содержит как среду выполнения, так и общую спецификацию языка, которая используется всеми остальными реализациями Python. CPython является официальной (или эталонной) реализацией Python.

Спецификация языка Python¹ представляет собой документ с описанием языка Python. Например, в ней сказано, что `assert` — ключевое слово, а `[]` используется для индексирования, срезов и создания пустых списков.

Подумайте, какую функциональность вы ожидаете от дистрибутива Python:

- Отображение интерактивного приглашения (REPL) при вводе команды `python` без указания файла или модуля.
- Импортирование встроенных модулей (таких, как `json`, `csv` и `collections`) из стандартной библиотеки.
- Установка пакетов из интернета командой `pip`.

¹ У Python нет спецификации в точном смысле слова, как для других языков программирования, но есть справочное руководство, которое описывает синтаксис и основную семантику языка: <https://docs.python.org/3/reference/>. — Примеч. ред.

- Тестирование приложений с использованием встроенной библиотеки `unittest`.

Все эти компоненты входят в дистрибутив CPython, который содержит намного больше, чем просто компилятор.

В этой книге будут рассмотрены разные элементы дистрибутива CPython:

- Спецификация языка.
- Компилятор.
- Модули стандартной библиотеки.
- Основные типы.
- Средства тестирования.

ЧТО В ИСХОДНОМ КОДЕ?

Дистрибутив с исходным кодом CPython включает обширный набор инструментов, библиотек и компонентов, которые будут рассмотрены в этой книге.

ПРИМЕЧАНИЕ

В этой книге описана версия 3.9¹ исходного кода CPython.

Чтобы загрузить копию последней версии исходного кода CPython, можно воспользоваться `git`:

```
$ git clone --branch 3.9 https://github.com/python/cpython  
$ cd cpython
```

Примеры этой книги основаны на Python версии 3.9.

ВАЖНО

Переход на ветку 3.9 – очень важный шаг. Master-ветка обновляется ежечасно. Многие примеры и упражнения, приведенные в книге, вряд ли будут работать на master-ветке.

¹ <https://github.com/python/cpython/tree/3.9>.

ПРИМЕЧАНИЕ

Если на вашем компьютере нет Git, установите его с сайта git-scm.com. Также можно загрузить ZIP-файл² исходного кода CPython прямо с веб-сайта GitHub.

Если исходный код будет загружен в виде ZIP-файла, то в нем не будет данных истории, тегов или веток.

В только что загруженному каталоге `cpython` находятся следующие подкаталоги:

 <code>cpython/</code>	
<code>Doc</code>	Документация
<code>Grammar</code>	Машиночитаемое определение языка
<code>Include</code>	Заголовочные файлы C
<code>Lib</code>	Модули стандартной библиотеки, написанные на Python
<code>Mac</code>	Файлы поддержки macOS
<code>Misc</code>	Разные файлы
<code>Modules</code>	Модули стандартной библиотеки, написанные на C
<code>Objects</code>	Основные типы и модель объекта
<code>Parser</code>	Исходный код парсера Python
<code>PC</code>	Файлы поддержки для старых версий Windows
<code>PCBuild</code>	Файлы поддержки для Windows
<code>Programs</code>	Исходный код для исполняемого файла <code>python</code> и других двоичных файлов
<code>Python</code>	Исходный код интерпретатора CPython
<code>Tools</code>	Автономные инструменты для создания или расширения CPython
<code>m4</code>	Специальные скрипты для автоматизации настройки make-файла (<code>makefile</code>)

На следующем шаге мы займемся настройкой среды разработки.

¹ <https://github.com/python/cpython/archive/3.9.zip>.

Настройка среды разработки

В этой главе мы будем работать как с кодом C, так и с кодом Python. Очень важно, чтобы среда разработки была настроена для обоих языков.

Исходный код CPython примерно на 65 % написан на Python (его значительную часть составляют тесты) и на 24 % — на языке C. Оставшуюся часть составляет смесь других языков.

IDE ИЛИ РЕДАКТОР?

Если вы еще не решили, какую среду разработки использовать, сначала необходимо принять принципиальное решение: выбрать интегрированную среду разработки (IDE) или редактор кода?

- **IDE** предназначены для конкретного языка и инструментария. Во многих IDE имеются интегрированные средства тестирования, проверки синтаксиса, контроля версий и компиляции.
- **Редактор кода** позволяет править файлы с программным кодом независимо от языка. Многие редакторы кода представляют собой простые текстовые редакторы с подсветкой синтаксиса.

Из-за своей полнофункциональной природы IDE часто потребляют больше аппаратных ресурсов. Таким образом, при ограниченном объеме памяти (менее 8 Гбайт) рекомендуется использовать редактор кода.

Кроме того, запуск IDE занимает больше времени. Если вам нужно быстро отредактировать файл, то, пожалуй, редактор кода для этого подойдет лучше.

Существуют сотни платных и бесплатных редакторов и IDE. Вот ряд примеров для работы с кодом CPython:

ПРИЛОЖЕНИЕ	НАЗНАЧЕНИЕ	ПОДДЕРЖИВАЕТСЯ
Microsoft Visual Studio Code	Редактор	Windows, macOS и Linux
Atom	Редактор	Windows, macOS и Linux
Sublime Text	Редактор	Windows, macOS и Linux
Vim	Редактор	Windows, macOS и Linux
Emacs	Редактор	Windows, macOS и Linux
Microsoft Visual Studio	IDE (C, Python и др.)	Windows
PyCharm by JetBrains	IDE (Python и др.)	Windows, macOS и Linux
CLion by JetBrains	IDE (C и др.)	Windows, macOS и Linux

Версия Microsoft Visual Studio также доступна для Mac, но она не поддерживает плагин Python Tools для Visual Studio и компиляцию C.

В следующих разделах будут рассмотрены подготовительные действия для нескольких редакторов и IDE:

- Microsoft Visual Studio
- Microsoft Visual Studio Code
- JetBrains CLion
- Vim

Перейдите к разделу для выбранного вами приложения или прочитайте все, чтобы сравнить разные варианты.

НАСТРОЙКА VISUAL STUDIO

Новейшая версия Visual Studio – Visual Studio 2019 – имеет встроенную поддержку Python и исходного кода C для Windows. Я рекомендую использовать ее для примеров и упражнений этой книги. Если на вашем компьютере уже установлена версия Visual Studio 2017, она тоже подойдет.

ПРИМЕЧАНИЕ

Никакие платные возможности Visual Studio не обязательны для компиляции CPython или чтения книги. Вы можете использовать бесплатное издание Community Edition.

Тем не менее для профильной оптимизации (profile-guided optimization) потребуется издание Professional Edition и выше.

Редактор Visual Studio можно бесплатно загрузить с веб-сайта Microsoft Visual Studio¹.

После того как вы загрузите программу установки Visual Studio, вам будет предложено выбрать устанавливаемые компоненты. Для книги понадобятся следующие компоненты:

- Рабочая конфигурация разработки Python (Python development).
- Необязательные средства нативной разработки Python (Python native development tools).
- Python 3, 64-разрядная версия (3.7.2).

Если у вас уже установлена версия Python 3.7, вы можете убрать галочку напротив Python 3 (3.7.2). Также можно отказаться от установки любых необязательных средств, если вы захотите сэкономить место на диске.

Программа установки загружает и устанавливает все необходимые компоненты. Установка может занять до часа; возможно, вам стоит продолжить чтение и вернуться к этому разделу после ее завершения.

Когда установка будет завершена, щелкните на кнопке Launch, чтобы запустить Visual Studio. Вам будет предложено войти в систему. Если у вас имеется учетная запись Microsoft, вы можете выполнить вход или пропустить этот шаг.

Затем будет предложено открыть проект. Также можно клонировать Git-репозиторий CPython прямо из Visual Studio — для этого выберите вариант Clone or check out code.

В поле расположения репозитория введите адрес <https://github.com/python/cpython>, выберите локальную папку для установки и нажмите Clone.

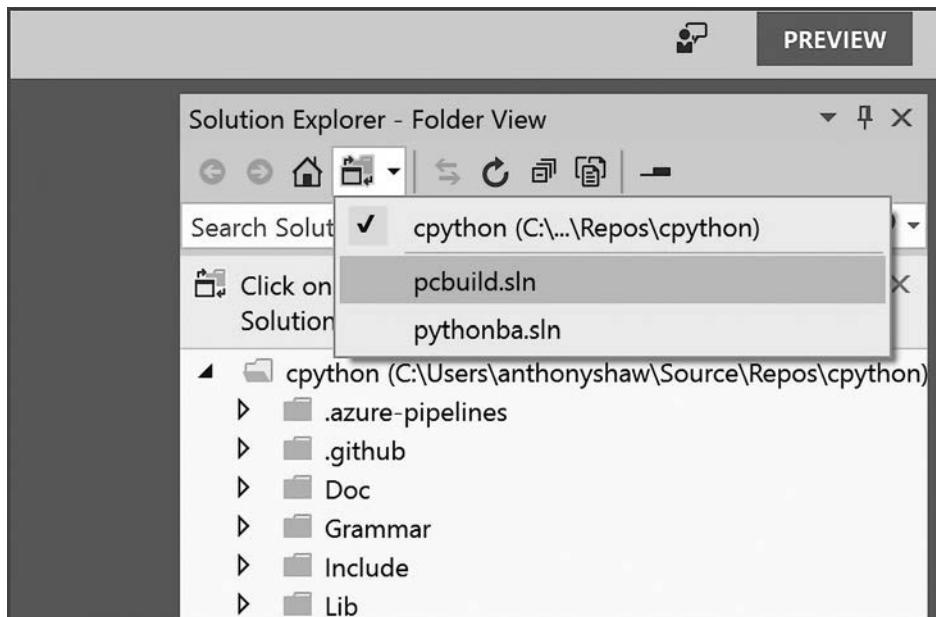
¹ <https://visualstudio.microsoft.com/vs/>.

Visual Studio загружает копию CPython с GitHub, для чего используется версия Git, входящая в поставку Visual Studio. Этот шаг также избавляет вас от хлопот с установкой Git в Windows. Загрузка может занять до десяти минут.

ВАЖНО

Visual Studio автоматически встает на master-ветку. Прежде чем переходить к компиляции, убедитесь в том, что вы переключились на ветку 3.9 в окне TeamExplorer. Переключение на ветку 3.9 – важный шаг. Master-ветка изменяется ежечасно, и многие примеры и упражнения в книге вряд ли будут работать на ней.

После того как проект будет загружен, необходимо передать Visual Studio информацию о файле решения PCBuild ▶ pcbuild.sln, выбрав команду Solutions and Projects ▶ pcbuild.sln:



Итак, среда Visual Studio настроена, а исходный код загружен. Можно переходить к компиляции CPython на Windows; эта процедура рассматривается в следующей главе.

НАСТРОЙКА VISUAL STUDIO CODE

Microsoft Visual Studio Code – расширяемый редактор кода с онлайн-магазином плагинов.

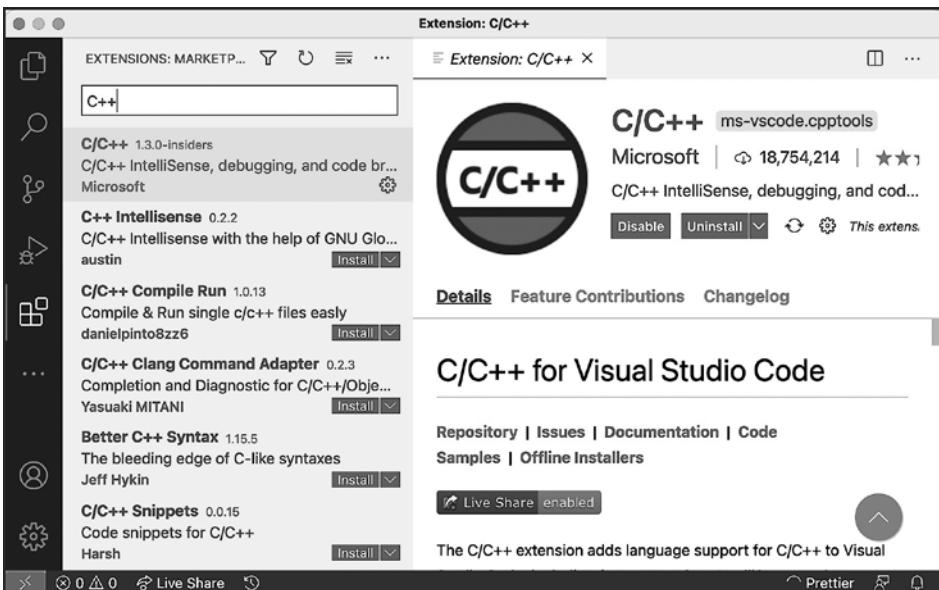
Он прекрасно подходит для работы с CPython, поскольку поддерживает как C, так и Python, а также имеет встроенный интерфейс Git.

Установка

Редактор Visual Studio Code, часто называемый VS Code, загружается с сайта code.visualstudio.com в виде простой программы установки.

В исходной конфигурации VS Code содержит все необходимое для редактирования кода, однако с установкой расширений (extensions) возможностей становится больше.

Панель **Extensions** находится в верхнем меню: View ▶ Extensions:



На панели **Extensions** возможен поиск расширений как по имени, так и по уникальному идентификатору (например, `ms-vscode.cpptools`). В отдельных

случаях может быть несколько плагинов с одинаковым именем; чтобы быть уверенным в том, что вы устанавливаете именно тот, который вам нужен, используйте уникальный идентификатор.

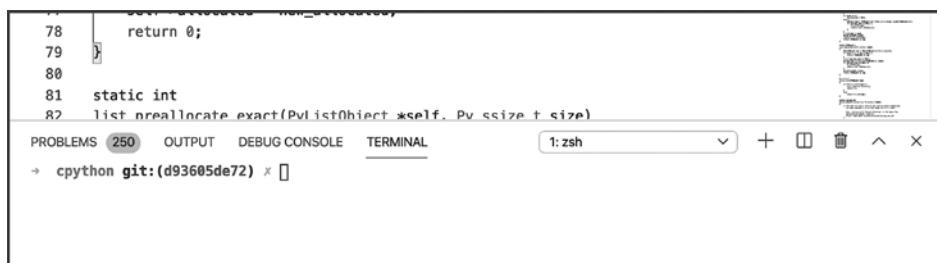
Рекомендуемые расширения для этой книги

Некоторые полезные расширения для работы с CPython:

- **C/C++** (`ms-vscode.cpptools`) обеспечивает поддержку C/C++, включая IntelliSense, отладку и подсветку синтаксиса.
- **Python** (`ms-python.python`) дает богатые возможности для редактирования, отладки и чтения кода Python.
- **reStructuredText** (`lextudio.restructuredtext`) предоставляет разностороннюю поддержку reStructuredText – формата, использованного в документации CPython.
- **Task Explorer** (`spmeesseman.vscode-taskexplorer`) добавляет на вкладку Explorer панель Task Explorer, упрощающую выполнение некоторых операций.

После установки этих расширений необходимо перезагрузить редактор.

Многие задачи, описанные в книге, требуют использования командной строки. Чтобы интегрировать терминал в VS Code, выберите команду **Terminal ▶ New Terminal**. Терминал открывается под редактором кода:



Расширенная навигация и развертывание кода

С установленными плагинами появляются новые возможности расширенной навигации в коде.

Например, если щелкнуть правой кнопкой мыши на вызове функции в файле C и выбрать команду Go to References, то VS Code найдет другие ссылки на эту функцию в кодовой базе:

The screenshot shows a code editor window for a file named `_bisectmodule.c`. The cursor is hovering over the macro definition `#define PyList_CheckExact(op) (Py_TYPE(op) == &PyList_Type)`. A context menu is open with the option "References (30)" selected. A tooltip displays a tree of references to this macro across various Python modules, including `_bisectmodule.c`, `PyList_CheckExact`, `_elementtree.c`, `textio.c`, `_pickle.c`, `statement.c`, `_testcapi.module.c`, and `acmodule.c`.

```

Include > C listobject.h > PyList_CheckExact(op)
50 #define PyList_CheckExact(op) (Py_TYPE(op) == &PyList_Type)

_bisectmodule.c ~/cpython/Modules - References (30)
101     index = internal_bisect_right(list, item, lo, hi);
102     if (index < 0)
103         return NULL;
104     if (PyList_CheckExact(list)) {
105         if (PyList_Insert(list, index, item) < 0)
106             return NULL;
107     } else {
108         result = _PyObject_CallMethodId(list, &PyId_insert,
109                                         if (result == NULL)
110                                         return NULL;
111                                         Py_DECREF(result);
112                                         Py_DECREF(result);

51 PyAPI_FUNC(PyObject *) PyList_New(Py_ssize_t size);
52 
```

Команда Go to References очень удобна для нахождения правильной формы вызова функции.

Если щелкнуть или навести указатель мыши на макрос C, то редактор развернет макрос в скомпилированный код:

The screenshot shows a code editor window for a file named `listobject.c`. The cursor is hovering over the macro definition `#define PY_SSIZE_T_MAX ((Py_ssize_t)((size_t)-1>>1))`. A tooltip displays the expanded form of the macro: `((Py_ssize_t)((size_t)-1>>1))`. The tooltip also includes the note "Largest positive value of type Py_ssize_t." and "Expands to:".

```

Objects > C listobject.c > list_resize(PyListObject *,Py_ssize_t)
50 }
51
52 /* This over-allocates proportional to the list size, making room
53 * for additional growth. The over-allocation is mild, but is
54 * enough to give linear-time amortized behavior over a long
55 * sequence of appends() in the presence of a poorly-performing
56 * system realloc().
57 * The growth pattern is: 0
58 * Note: new_allocated won't be larger than PY_SSIZE_T_MAX.
59 *       is PY_SSIZE_T_MAX *
60 */
61 new_allocated = (size_t)news((Py_ssize_t)((size_t)-1>>1))
62 if (new_allocated > (size_t)PY_SSIZE_T_MAX / sizeof(PyObject *)) {
63     PyErr_NoMemory();
64     return -1;
65 }
```

Чтобы перейти к определению функции, наведите указатель мыши на любой ее вызов, зажмите Cmd (macOS) или Ctrl (Linux и Windows) и кликните.

Конфигурирование файлов команд и запуска

VS Code использует папку `.vscode` в каталоге рабочего пространства. Если эта папка не существует, создайте ее. В ней можно сформировать следующие файлы:

- `tasks.json` для определения сокращенных команд, запускающих ваш проект;
- `launch.json` для настройки отладчика (см. главу «Отладка»);
- другие файлы, относящиеся к плагинам.

Создайте файл `tasks.json` в каталоге `.vscode`, если он еще не существует. Следующего описания команд в `tasks.json` будет достаточно для начала работы:

cpython-book-samples ▶ 11 ▶ tasks.json

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "build",
      "type": "shell",
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "windows": {
        "command": "PCBuild/build.bat",
        "args": ["-p", "x64", "-c", "Debug"]
      },
      "linux": {
        "command": "make -j2 -s"
      },
      "osx": {
        "command": "make -j2 -s"
      }
    }
  ]
}
```

При использовании плагина Task Explorer список настроенных команд будет отображаться в группе `vscode`:



В следующей главе процесс сборки при компиляции CPython будет рассмотрен более подробно.

Настройка JetBrains CLion

У компании JetBrains есть IDE как для Python (PyCharm), так и для C/C++ (CLion).

В CPython используется код и на C, и на Python. Встроить поддержку C/C++ в PyCharm не удастся, а CLion поддерживает Python.

ВАЖНО

Поддержка make-файлов (`makefile`) доступна только в CLion версий 2020.2 и выше.

ВАЖНО

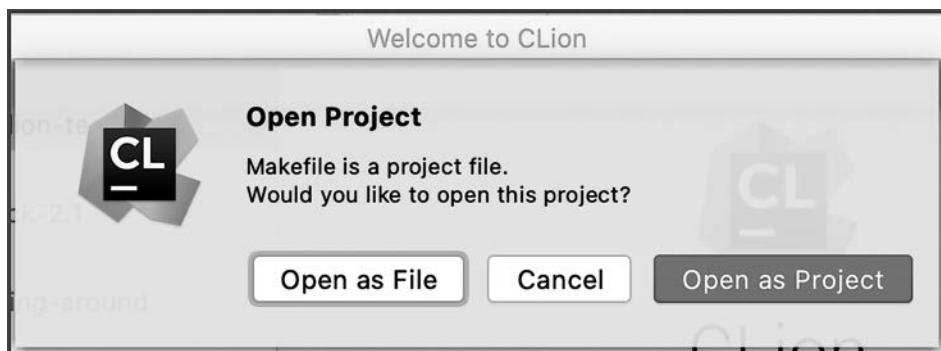
Для этого шага необходимо как сгенерировать `makefile`, запустив `configure`, так и скомпилировать CPython.

Прочтайте главу «Компиляция CPython» для вашей операционной системы, а потом вернитесь к этой главе.

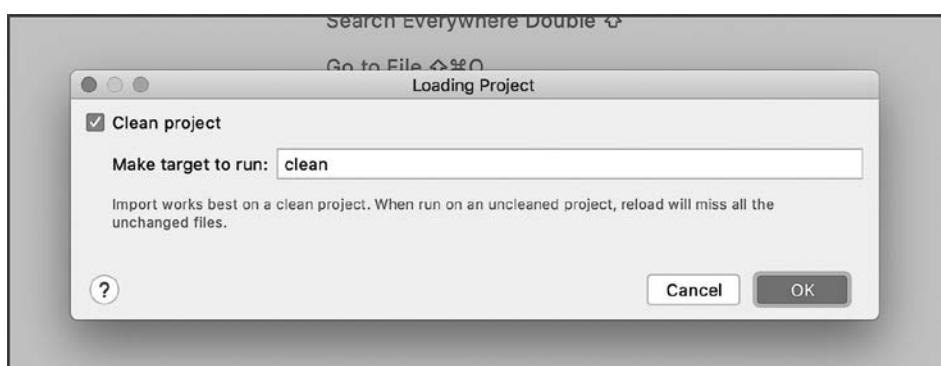
После первой компиляции CPython в корне каталога с исходным кодом появится `makefile`.

Откройте CLion и выберите на заставке вариант Open or Import. Перейдите в каталог с исходным кодом, выберите makefile и щелкните по кнопке Open.

CLion спросит, хотите ли вы открыть каталог или импортировать makefile как новый проект. Выберите вариант Open as Project, чтобы импортировать файл как проект:



Перед импортированием CLion спросит, какую make-цель следует запустить. Оставьте вариант по умолчанию clean и продолжите выполнение:



Затем убедитесь в том, что вы можете собрать исполняемый файл CPython из CLion. Выберите в верхнем меню команду Build ▶ Build Project.

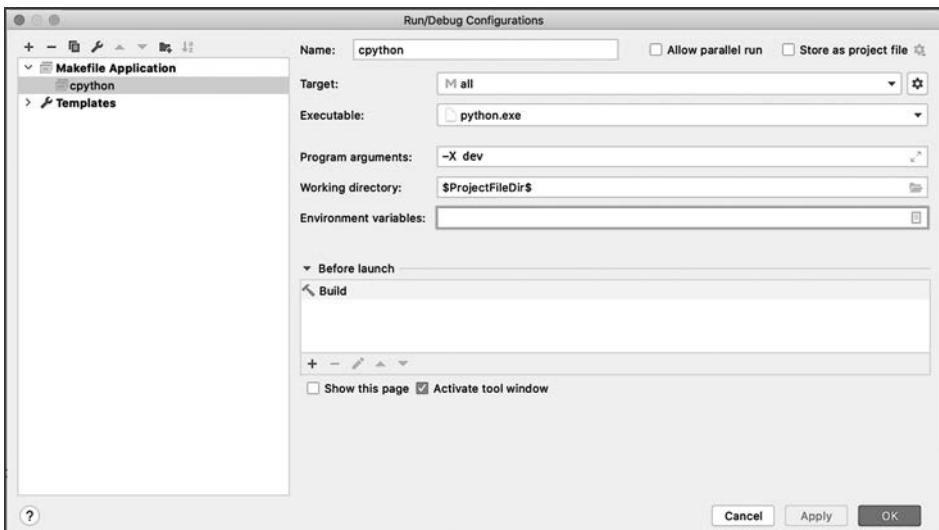
В строке состояния должен появиться индикатор прогресса сборки проекта.



Когда задача выполнена, скомпилированный двоичный файл можно будет выбрать в конфигурации запуска/отладки.

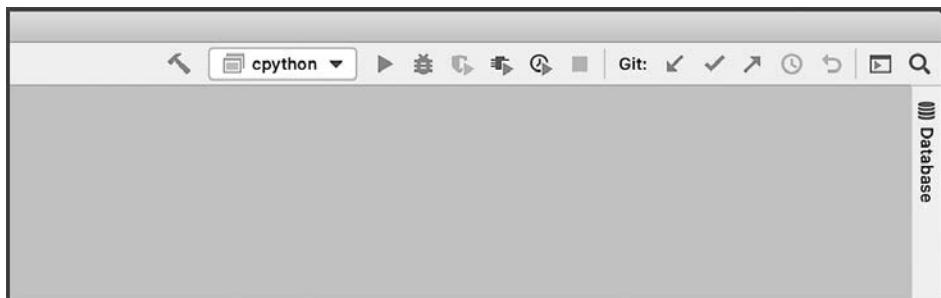
Выберите команду **Run ▶ Edit Configurations**, чтобы открыть окно конфигурации запуска/отладки (**Run/Debug Configurations**). В окне выберите команду **+ ▶ Makefile Application** и выполните следующие действия:

1. Введите в поле **Name** имя `cpython`.
2. Оставьте в поле **Target** цель `all`.
3. В поле **Executable** откройте выпадающий список и выберите **Select Other**, затем найдите скомпилированный двоичный файл CPython в каталоге с исходным кодом. Он будет называться `python` или `python.exe`.
4. В поле **Program Arguments** ведите аргументы программы, которые должны передаваться всегда (например, `-X dev` для включения режима разработки). Эти флаги будут рассмотрены позднее в разделе «Настройка конфигураций времени выполнения в командной строке».
5. В поле **Working Directory** введите макрос `CLion $ProjectFileDir$`:

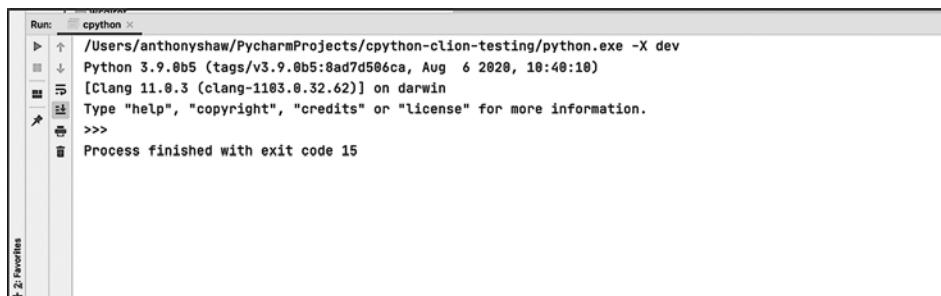


Щелкните по кнопке **OK**, чтобы добавить данную конфигурацию. Этот шаг можно повторить сколько угодно раз для всех make-целей CPython. За полной информацией обращайтесь к разделу «Make-цели CPython» в главе «Компиляция CPython».

После этого конфигурация сборки `cpython` будет доступна справа наверху в окне CLion:



Чтобы протестировать ее, щелкните по кнопке со стрелкой или выберите команду **Run ▶ Run 'cpython'** в верхнем меню. Снизу в окне CLion должна появиться панель REPL:



Отлично! Теперь вы сможете вносить изменения и быстро их применять командами **Build** и **Run**. Если в коде С установлены точки останова, вместо **Run** следует выбирать **Debug**.

В редакторе кода зажатие клавиши **Cmd** (macOS) или **Ctrl** (Windows и Linux) и щелчок мышью вызывает внутренние средства навигации:

```

708 static int do_raise(PyThreadState *tstate, PyObject *exc, PyObject *cause);
709 static int unpack_iterable(PyThreadState *, PyObject *, int, int, PyObject **);
710
711 #define _Py_TracingPossible(ceval) ((ceval)->tracing_possible)
712
713 PyObject *
714 PyEval_EvalCode(PyObject *code, PyObject *globals, PyObject *locals,
715                 PyObject **closure, PyObject **freevars, PyObject **cellvars,
716                 PyObject **dict, PyObject **name, PyObject **filename,
717                 PyObject **line, PyObject **lnotab, PyObject **stack,
718                 PyObject **argnames, PyObject **argvalues, PyObject **argdict,
719                 PyObject **argclosure, PyObject **argfreevars, PyObject **argcellvars,
720                 PyObject **argdict, PyObject **argname, PyObject **argfilename,
721                 PyObject **argline, PyObject **arglnotab, PyObject **argstack,
722                 PyObject **argargnames, PyObject **argargvalues, PyObject **argargdict,
723                 PyObject **argargclosure, PyObject **argargfreevars, PyObject **argargcellvars,
724                 PyObject **argargdict, PyObject **argargname, PyObject **argargfilename,
725                 PyObject **argargline, PyObject **argarglnotab, PyObject **argargstack);
726
727 /* Interpreter main loop */

```

НАСТРОЙКА VIM

Vim — мощный консольный текстовый редактор. Чтобы процесс разработки с Vim был особенно эффективным, разместите пальцы на базовых клавишиах (home keys¹) клавиатуры. Все необходимые сокращения и команды будут у вас прямо под рукой.

ПРИМЕЧАНИЕ

В большинстве дистрибутивов Linux и в терминале macOS для vim определен синоним vi. В книге будет использоваться команда vim, но команда vi тоже будет работать, если в вашей системе этот синоним существует.

В исходном состоянии Vim обладает базовой функциональностью и практически не превосходит такой текстовый редактор, как «Блокнот» (Notepad). Однако с некоторыми конфигурациями и расширениями Vim превращается в мощный инструмент для редактирования кода Python и C.

Расширения Vim хранятся в разных местах, включая GitHub. Для упрощения настройки и установки плагинов с GitHub можно воспользоваться менеджером плагинов, например Vundle.

Чтобы установить Vundle, выполните в терминале следующую команду:

```
$ git clone https://github.com/VundleVim/Vundle.vim.git \
~/vim/bundle/Vundle.vim
```

¹ Клавиши «A», «S», «D», «F» — левая рука и «J», «K», «L» — правая. — Примеч. ред.

После установки менеджера Vundle необходимо настроить Vim для загрузки ядра Vundle.

Мы установим два плагина:

1. Fugitive: строка состояния для Git с множеством сокращений для выполнения операций Git.
2. Tagbar: панель, упрощающая переход к функциям, методам и классам.

Чтобы установить эти плагины, сначала измените содержимое файла конфигурации Vim (обычно `HOME ▶ .vimrc`) и включите в него следующие строки:

cpython-book-samples ▶ 11 ▶ .vimrc

```
syntax on
set nocompatible          " отключить совместимость с Vi, обязательно
filetype off               " обязательно

" установка пути среды выполнения для включения Vundle и инициализация
set rtp+=~/vim/bundle/Vundle.vim
call vundle#begin()

" Разрешить Vundle управление Vundle, обязательно
Plugin 'VundleVim/Vundle.vim'

" Ниже приводятся примеры различных поддерживаемых форматов.
" Команды Plugin должны располагаться между vundle#begin/end.
" Плагин в репозитории GitHub
Plugin 'tpope/vim-fugitive'
Plugin 'majutsushi/tagbar'
" Все команды Plugin должны быть добавлены до этой строки
call vundle#end()           " обязательно
filetype plugin indent on   " обязательно

" Автоматически открывать панель tagbar в файлах C, не обязательно
autocmd FileType c call tagbar#autoopen(0)
" Автоматически открывать панель tagbar в файлах Python, не обязательно
autocmd FileType python call tagbar#autoopen(0)
" Вывести строку состояния, не обязательно
set laststatus=2
" Установить статус как git status (branch), не обязательно
set statusline=%{FugitiveStatusline()}
```

Чтобы загрузить и установить эти плагины, выполните следующие команды:

```
$ vim +PluginInstall +qall
```

На экране должны появиться сообщения о загрузке и установке плагинов, заданных в файле конфигурации.

В ходе редактирования и анализа исходного кода CPython обычно возникает необходимость в быстром переключении между методами, функциями и макросами. Простой поиск по тексту не отличит вызов или определение функции от ее реализации. Однако вы можете воспользоваться приложением ctags¹ и проиндексировать исходные файлы на разных языках в базу данных с обычным текстом. Чтобы проиндексировать заголовки CPython для всех файлов C и Python в стандартной библиотеке, выполните следующий код:

```
$ ./configure
$ make tags
```

Теперь откройте файл Python `ceval.c` в Vim:

```
$ vim Python/ceval.c
```

Статус Git должен отображаться в нижней строке, а справа в окне появляется список функций, макросов и переменных:

```
#include "setobject.h"
#include "structmember.h"

#include <ctype.h>

#ifndef Py_DEBUG
/* For debugging the interpreter: */
#define LLTRACE 1      /* Low-level trace feature */
#define CHECKEXC 1     /* Double-check exception checking */
#endif

#ifndef Py_BUILD_CORE
# error "ceval.c must be build with Py_BUILD_CORE define for best performance"
#endif

/* Private API for the LOAD_METHOD opcode. */
extern int _PyObject_GetMethod(PyObject *, PyObject *, PyObject **);

typedef PyObject *(*callproc)(PyObject *, PyObject *, PyObject *);

/* Forward declarations */
Py_LOCAL_INLINE(PyObject *) call_function(
    PyThreadState *tstate, PyObject ***pp_stack,
    Py_ssize_t oparg, PyObject *kwnames);
static PyObject * do_call_core(
    PyThreadState *tstate, PyObject *func,
    PyObject *callargs, PyObject *kwdict);

#ifndef LLTRACE
static int lltrace;
static int ptrace(PyObject *, const char *);
#endif
```

" Press <F1>, ? for help

- ▶ macros
- ▶ prototypes
- ▼ typedefs
 - callproc
- ▼ variables
 - _Py_CheckRecursionLimit
 - dpx
 - dpxpairs
 - lltrace
- ▼ functions
 - PyEval_AcquireLock(void)
 - PyEval_AcquireThread(PyThreadState *
 - PyEval_EvalCode(PyObject *co, PyObject *
 - PyEval_EvalCodeEx(PyObject *co, PyObject *
 - PyEval_EvalFrame(PyFrameObject *f)
 - PyEval_EvalFrameEx(PyFrameObject *f,
 - PyEval_GetBuiltins(void)
 - PyEval_GetFrame(void)
 - PyEval_GetFuncDesc(PyObject *func)
 - PyEval_GetFuncName(PyObject *func)
 - PyEval_GetGlobals(void)
 - PyEval_GetLocals(void)
 - PyEval_InitThreads(void)
 - PyEval_MergeCompilerFlags(PyCompiler
 - PyEval_ReleaseLock(void)
 - PyEval_ReleaseThread(PyThreadState *

[Name] ceval.c

¹ <http://ctags.sourceforge.net/>.

Откройте файл Python — например, Lib ▶ subprocess.py:

```
$ vim Lib/subprocess.py
```

На панели tagbar будет выведена информация об импортированных именах, классах, методах и функциях:

```
python — vi Lib/subprocess.py — vi — vi Lib/subprocess.py — 111x34
self.returncode = returncode
self.cmd = cmd
self.output = output
self.stderr = stderr

def __str__(self):
    if self.returncode and self.returncode < 0:
        try:
            return "Command '%s' died with %r." % (
                self.cmd, signal.Signals(-self.returncode))
        except ValueError:
            return "Command '%s' died with unknown signal %d." % (
                self.cmd, -self.returncode)
    else:
        return "Command '%s' returned non-zero exit status %d." % (
            self.cmd, self.returncode)

@property
def stdout(self):
    """Alias for output attribute, to match stderr"""
    return self.output

@stdout.setter
def stdout(self, value):
    # There's no obvious reason to set this, but allow it anyway so
    # .stdout is a transparent alias for .output
    self.output = value

class TimeoutExpired(SubprocessError):
[Git(master)]
```

" Press <F1>, ? for help
 ► imports
 ▾ CalledProcessError : class
 +__init__: function
 +__str__: function
 +_stdout : function
 +_stderr : function
 ▾ CompletedProcess : class
 +__init__: function
 +__repr__: function
 +check_returncode : function
 ▾ Handle : class
 +Close : function
 +Detach : function
 +__repr__: function
 [variables]
 +_PopenSelector : function
 +_PopenSelector : function
 +_del_ : function
 +closed : function
 ▾ Popen : class
 +__del__: function
 +__enter__: function
 +__exit__: function
 +__init__: function
 +check_timeout : function
 +_close_pipe_fds : function
 [Name] subprocess.py

В Vim можно переключаться между окнами клавишами **Ctrl+W**, перемещаться на правую панель клавишей **L** и использовать клавиши со стрелками для перемещения вверх и вниз между функциями.

Нажмите **Enter**, чтобы перейти к реализации любой функции. Чтобы вернуться к панели редактора, нажмите **Ctrl+W**, а затем **H**.

СМ. ТАКЖЕ

На сайте VIM Adventures¹ представлен занятный способ изучения и запоминания команд Vim.

¹ <https://vim-adventures.com/>.

ВЫВОДЫ

Если вы еще не решили, какую среду разработки выбрать, не обязательно принимать решение прямо сейчас. Мы использовали разные среды во время работы над книгой и при внесении изменений в CPython.

Отладка становится одним из критических факторов эффективной работы, так что надежный отладчик, в котором можно проверить состояние среды выполнения и понять причины ошибок, сэкономит немало времени. Если вы привыкли к отладке в Python с использованием команд `print()`, обратите внимание, что в С такой подход не работает. Более подробно отладка рассматривается далее в книге.

Компиляция CPython

После того как вы загрузили среду разработки и настроили ее, можно скомпилировать исходный код CPython в исполняемый интерпретатор.

В отличие от файлов Python, исходный код С необходимо заново компилировать при каждом изменении. Вероятно, вам стоит положить закладку на этой главе и запомнить некоторые шаги, потому что они будут неоднократно повторяться.

В предыдущей главе вы видели, как настроить среду разработки с возможностью запуска процесса сборки, который перекомпилирует CPython. Но чтобы операции сборки заработали, вам понадобится компилятор С и некоторые инструменты.

Выбор инструментов зависит от операционной системы, поэтому перейдите к разделу, в котором рассматривается ваша ОС.

ПРИМЕЧАНИЕ

Если вас беспокоит, что какие-либо из этих действий отразятся на уже установленных версиях CPython, не тревожьтесь. Каталог исходного кода CPython ведет себя как виртуальная среда.

При компиляции CPython или изменении исходного кода стандартной библиотеки все остается в изолированной среде («песочнице») каталога с исходным кодом.

Если вы захотите установить нестандартную версию, читайте дальше – этот шаг рассматривается в данной главе.

КОМПИЛЯЦИЯ CPYTHON НА MACOS

Компиляция CPython на macOS потребует дополнительных приложений и библиотек. Прежде всего вам понадобится основной тулкит компилятора C. Command Line Tools — приложение, которое можно обновлять в macOS через App Store. Исходная установка должна выполняться в терминале.

ПРИМЕЧАНИЕ

Чтобы открыть терминал в macOS, выберите команду Applications ▶ Other ▶ Terminal. Приложение лучше сохранить в Dock, поэтому зажмите Ctrl, кликните по иконке и выберите команду Keep in Dock.

В терминале установите компилятор C и тулкит следующей командой:

```
$ xcode-select --install
```

После выполнения команды вам будет предложено загрузить и установить набор инструментов, включая Git, Make и компилятор GNU C.

Кроме того, потребуется рабочая копия OpenSSL для загрузки пакетов с сайта PyPI. Если вы планируете использовать эту сборку для установки дополнительных библиотек, потребуется проверка SSL-сертификата.

Чтобы установить OpenSSL в macOS, проще всего воспользоваться менеджером пакетов Homebrew.

ПРИМЕЧАНИЕ

Если вы еще не установили программу Homebrew, загрузите и установите ее прямо с GitHub следующей командой:

```
$ /usr/bin/ruby -e "$(curl -fsSL \
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

С помощью Homebrew можно установить зависимости CPython командой `brew install`:

```
$ brew install openssl xz zlib gdbm sqlite
```

Все зависимости готовы, теперь можно запустить скрипт `configure`.

Команда Homebrew `brew --prefix <пакет>` выдает каталог, в котором установлен *пакет*. Чтобы включить поддержку SSL, используйте путь, сгенерированный Homebrew.

Флаг `--with-pydebug` активизирует отладочные хуки (hooks). Добавьте этот флаг, если вы хотите отладить процесс разработки или тестирования. Отладка CPython подробно рассматривается в главе «Отладка».

Настроить конфигурации достаточно всего один раз, указав путь до пакета zlib:

```
$ CPPFLAGS="-I$(brew --prefix zlib)/include" \
  LDFLAGS="-L$(brew --prefix zlib)/lib" \
  ./configure --with-openssl=$(brew --prefix openssl) \
  --with-pydebug
```

Команда `./configure` генерирует makefile в корне репозитория. Ее можно использовать для автоматизации процесса сборки.

Теперь можно создать двоичный файл CPython следующей командой:

```
$ make -j2 -s
```

СМ. ТАКЖЕ

За дополнительной информацией о `make` обращайтесь к разделу «Знакомство с Make».

В процессе сборки могут появиться сообщения об ошибках. В сводной информации `make` оповестит вас о том, что не все пакеты были собраны. Например, с приведенными инструкциями не соберутся пакеты `ossaudiodev`, `spwd` и `_tkinter`. Это нормально, если вы не планируете их использовать. А если планируете — обращайтесь к руководству Python Developer's Guide¹ за дополнительной информацией.

Сборка займет несколько минут, и в результате будет сгенерирован двоичный файл с именем `python.exe`. Каждый раз, когда вы вносите изменения в исходный код, вам придется перезапустить `make` с теми же флагами.

¹ <https://devguide.python.org/>.

Двоичный файл `python.exe` является отладочной двоичной версией CPython. Запустите `python.exe`, чтобы увидеть рабочий интерпретатор REPL:

```
$ ./python.exe
Python 3.9 (tags/v3.9:9cf67522, Oct 5 2020, 10:00:00)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

ВАЖНО

Да, все верно, сборка macOS использует расширение `.exe`. Это расширение появилось вовсе не потому, что это двоичный файл Windows!

Так как в файловой системе macOS учитывается регистр символов, разработчики не хотели, чтобы при работе с двоичным файлом пользователи случайно обращались к каталогу `Python/`, поэтому они присоединили `.exe` для предотвращения неоднозначности.

Если позднее вы выполните команду `make install` или `make altinstall`, перед установкой в систему файл будет переименован в `python`.

КОМПИЛЯЦИЯ CPYTHON НА LINUX

Чтобы скомпилировать CPython на Linux, сначала загрузите и установите `make`, `gcc`, `configure` и `pkgconfig`.

Используйте следующую команду для Fedora Core, RHEL, CentOS или других систем на базе YUM:

```
$ sudo yum install yum-utils
```

Для Debian, Ubuntu или других систем на базе APT команда выглядит так:

```
$ sudo apt install build-essential
```

Затем установите дополнительные необходимые пакеты.

Команда для Fedora Core, RHEL, CentOS или других систем на базе YUM:

```
$ sudo yum-builddep python3
```

Команда для Debian, Ubuntu и других систем на базе APT:

```
$ sudo apt install libssl-dev zlib1g-dev libncurses5-dev \
libncursesw5-dev libreadline-dev libsqlite3-dev libgdbm-dev \
libdb5.3-dev libbz2-dev libexpat1-dev liblzma-dev libffi-dev
```

После подготовки зависимостей можно запустить скрипт `configure`, при желании включив отладочные хуки с ключом `--with-pydebug`:

```
$ ./configure --with-pydebug
```

Далее можно собрать двоичный файл CPython, запустив генерированный `makefile`:

```
$ make -j2 -s
```

СМ. ТАКЖЕ

За дополнительной информацией о параметрах `make` обращайтесь к разделу «Знакомство с Make».

Просмотрите вывод и убедитесь в том, что при компиляции модуля `_ssl` не возникло никаких проблем. Если они возникли, поищите в документации дистрибутива инструкции по установке заголовков для OpenSSL.

В процессе сборки могут появиться сообщения об ошибках. В сводной информации `make` оповестит вас о том, что не все пакеты были собраны. Это нормально, если вы не планируете их использовать. А если планируете — обращайтесь к описанию пакетов за информацией о необходимых библиотеках.

Сборка займет несколько минут, и в результате будет сгенерирован двоичный файл с именем `python`. Это отладочная двоичная версия CPython. Запустите `./python.exe`, чтобы увидеть рабочий интерпретатор REPL:

```
$ ./python
Python 3.9 (tags/v3.9:9cf67522, Oct 5 2020, 10:00:00)
[Clang 10.0.1 (clang-1001.0.46.4)] on Linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

УСТАНОВКА СПЕЦИАЛИЗИРОВАННОЙ ВЕРСИИ

Если ваши изменения вас устраивают и вы хотите использовать их в своей системе, установите двоичный файл Python из вашего репозитория с исходным кодом как специализированную версию.

Для macOS и Linux используйте команду `altinstall`, которая не создает символические ссылки для `python3` и устанавливает автономную версию:

```
$ make altinstall
```

Для Windows необходимо заменить конфигурацию сборки `Debug` на `Release`, а затем скопировать упакованные двоичные файлы в директорию вашего компьютера, входящую в системный путь.

ЗНАКОМСТВО С MAKE

Возможно, вы как Python-разработчик еще не сталкивались с программой `make`. А может, сталкивались, но ваш опыт работы с ней невелик.

Для C, C++ и других компилируемых языков список команд, которые необходимо выполнить для загрузки, компоновки и компиляции вашего кода в правильном порядке, может оказаться очень длинным. При компиляции приложений из исходного кода необходимо подключить все внешние библиотеки в системе.

Нереально ожидать, что разработчик знает местонахождение всех этих библиотек и сам вставляет их в командную строку, поэтому программы `make` и `configure` часто используются в проектах C/C++ для автоматизации создания скрипта сборки.

Когда вы выполняете `./configure`, утилита `autoconf` ищет в вашей системе библиотеки, необходимые для CPython, и копирует их пути в `makefile`. Сгенерированный `makefile` напоминает скрипт командной оболочки и делится на части, называемые **целями** (**targets**).

Для примера рассмотрим цель `docclean`. Она удаляет сгенерированные файлы документации командой `rm`:

```
docclean:  
-rm -rf Doc/build  
-rm -rf Doc/tools/sphinx Doc/tools/pygments Doc/tools/docutils
```

Чтобы реализовать эту цель, выполните команду `make docclean`. `docclean` — очень простая цель, поскольку запускает всего две команды.

При выполнении `make`-целей используется следующая конструкция:

```
$ make [параметры] [цель]
```

`make` без указания цели, команда запустит цель по умолчанию (первая цель, указанная в `makefile`). В CPython это цель `all`, компилирующая все части CPython.

`make` поддерживает большое количество параметров. Вот некоторые параметры, которые могут пригодиться вам по ходу чтения книги.

ПАРАМЕТР	ИСПОЛЬЗОВАНИЕ
<code>-d, --debug[=FLAGS]</code>	Вывод разнообразной отладочной информации
<code>-e, --environment-overrides</code>	Переопределение настроек <code>makefile</code> переменными среды
<code>-i, --ignore-errors</code>	Игнорирование ошибок команд
<code>-j [N], --jobs[=N]</code>	Допуск одновременного выполнения N задач (в противном случае количество задач не ограничивается)
<code>-k, --keep-going</code>	Продолжить выполнение, если некоторые цели не удалось выполнить
<code>-l [N], --load-average[=N], --max-load[=N]</code>	Запуск нескольких задач, только если <code>load < N</code>
<code>-n, --dry-run</code>	Вывод команд вместо их запуска
<code>-s, --silent</code>	Отключение команды <code>echo</code> ¹
<code>-S, --stop</code>	Остановка, если некоторые цели не удалось выполнить

В следующем разделе и далее в книге `make` будет выполняться со следующими параметрами:

```
$ make -j2 -s [target]
```

Флаг `-j2` позволяет выполнять две задачи одновременно. Если на вашем компьютере четыре ядра и более, вы можете задать значение 4 и выше; в этом случае компиляция будет быстрее.

¹ Вывод текста в терминал. — Примеч. ред.

Флаг `-s` блокирует вывод на консоль результатов каждой команды из `makefile`. Если вы хотите увидеть, что происходит при сборке, удалите флаг `-s`.

МАКЕ-ЦЕЛИ СРУТНОН

Как в Linux, так и в macOS периодически возникает необходимость удаления файлов, а также построения или обновления конфигурации. В приведенных ниже таблицах описаны полезные `make`-цели, встроенные в `makefile` CPython.

Цели сборки

Следующие цели используются для сборки двоичных файлов CPython.

ЦЕЛЬ	НАЗНАЧЕНИЕ
<code>all</code> (по умолчанию)	Сборка компилятора, библиотек и модулей
<code>clinic</code>	Запуск Argument Clinic во всех исходных файлах
<code>profile-opt</code>	Компиляция двоичного файла Python с профильной оптимизацией
<code>regen-all</code>	Повторное создание всех сгенерированных файлов
<code>sharedmods</code>	Сборка всех общих модулей

Цели тестирования

Следующие цели используются для тестирования скомпилированного двоичного файла.

ЦЕЛЬ	НАЗНАЧЕНИЕ
<code>coverage</code>	Компиляция и запуск тестов с использованием <code>gcov</code>
<code>coverage-lcov</code>	Создание HTML-отчетов о покрытии кода
<code>quicktest</code>	Выполнение набора более быстрых регрессионных тестов с исключением тестов, занимающих много времени
<code>test</code>	Выполнение основного набора регрессионных тестов

ЦЕЛЬ	НАЗНАЧЕНИЕ
testall	Двукратное выполнение полного набора тестов: один раз без файлов .rus и один – с ними
testuniversal	Выполнение набора тестов для обеих архитектур в универсальной сборке для OS X

Цели очистки

Основные цели очистки — `clean`, `clobber` и `distclean`. Цель `clean` просто удаляет скомпилированные и кэшированные библиотеки и файлы `.rus`.

Если вы обнаружили, что `clean` не делает этого, попробуйте использовать `clobber`. Цель `clobber` удаляет `makefile`, так что вам придется снова запустить `./configure`.

Чтобы полностью очистить среду и перевести ее в состояние до установки дистрибутива, выполните цель `distclean`.

В следующей таблице описаны эти три основные цели очистки, а также ряд дополнительных.

ЦЕЛЬ	НАЗНАЧЕНИЕ
<code>check-clean-src</code>	Проверка чистоты исходного кода при сборке
<code>clean</code>	Удаление файлов <code>.rus</code> , скомпилированных библиотек и профилей
<code>cleantest</code>	Удаление каталогов <code>test_python_*</code> предыдущих неудачных тестовых заданий
<code>clobber</code>	То же, что <code>clean</code> , но с удалением библиотек, тегов, конфигураций и сборок
<code>distclean</code>	То же, что <code>clobber</code> , но с удалением всего генерированного из исходных файлов (например, <code>make</code> -файлов)
<code>docclean</code>	Удаление собранной документации в <code>Doc/</code>
<code>profile-removal</code>	Удаление всех профилей оптимизации
<code>rusremoval</code>	Удаление файлов <code>.rus</code>

Цели установки

Существуют две разновидности целей установки: стандартные (например, `install`) и альтернативные (например, `altinstall`). Если вы хотите установить скомпилированную версию на ваш компьютер, но не хотите делать ее установкой Python 3 по умолчанию, используйте альтернативную версию команды.

ЦЕЛЬ	НАЗНАЧЕНИЕ
<code>altbininstall</code>	Установка интерпретатора <code>python</code> с присоединенной версией – например, <code>python3.9</code>
<code>altinstall</code>	Установка общих библиотек, двоичных файлов и документации с суффиксом версии
<code>altmaninstall</code>	Установка документации с версией
<code>bininstall</code>	Установка всех двоичных файлов, включая <code>python</code> , <code>idle</code> и <code>2to3</code>
<code>commoninstall</code>	Установка общих библиотек и модулей
<code>install</code>	Установка общих библиотек, двоичных файлов и документации (с запуском <code>commoninstall</code> , <code>bininstall</code> и <code>maninstall</code>)
<code>libinstall</code>	Установка общих библиотек
<code>maninstall</code>	Установка документации
<code>sharedinstall</code>	Динамическая загрузка модулей

После выполнения установки с помощью `make install` команда `python3` связывается со скомпилированным двоичным файлом. Однако при использовании `altinstall` установится только `python$(версия)`, а существующая ссылка на `python3` останется неизменной.

Другие цели

Ниже перечислены некоторые дополнительные `make`-цели, которые могут вам пригодиться.

ЦЕЛЬ	НАЗНАЧЕНИЕ
<code>autoconf</code>	Повторное генерирование <code>configure</code> и <code>pyconfig.h.in</code>
<code>python-config</code>	Генерирование скрипта <code>python-config</code>

ЦЕЛЬ	НАЗНАЧЕНИЕ
recheck	Повторное выполнение configure с теми же параметрами, которые применялись в последний раз
smelly	Проверка того, что имена экспортруемых символических имен начинаются с Py или _Py (см. PEP 7)
tags	Создание файла tags для vi
TAGS	Создание файла tags для Emacs

КОМПИЛЯЦИЯ СРУТНОН НА WINDOWS

Существуют два способа компиляции двоичных файлов и библиотек CPython на Windows:

1. Компиляция из командной строки. Для этого варианта потребуетсяся компилятор Microsoft Visual C++, входящий в поставку Visual Studio.
2. Прямая сборка через PCbuild ▶ pcbuild.sln в Visual Studio.

В следующих разделах рассматриваются оба варианта.

Установка зависимостей

Как при компиляции из командной строки, так и через Visual Studio необходимо установить ряд дополнительных инструментов, библиотек и заголовков С.

В папке PCbuild находится файл .bat, автоматизирующий этот процесс. Откройте окно командной строки в PCbuild и выполните команду PCbuild ▶ get_externals.bat:

```
> get_externals.bat
Using py -3.7 (found 3.7 with py.exe)
Fetching external libraries...
Fetching bzip2-1.0.6...
Fetching sqlite-3.28.0.0...
Fetching xz-5.2.2...
Fetching zlib-1.2.11...
Fetching external binaries...
Fetching openssl-bin-1.1.1d...
Fetching tcltk-8.6.9.0...
Finished.
```

Теперь можно выполнить компиляцию из командной строки или Visual Studio.

Компиляция из командной строки

Чтобы выполнить компиляцию из командной строки, необходимо выбрать архитектуру процессора, для которого компилируется программа. По умолчанию используется архитектура `win32`, но вполне вероятно, что вам понадобится 64-разрядный двоичный файл (`amd64`).

Если вы занимаетесь отладкой, то отладочная сборка включает возможность установки точек останова в исходном коде. Чтобы включить отладочную сборку, добавьте параметр `-c Debug` для определения отладочной конфигурации.

По умолчанию `build.bat` загружает внешние зависимости, но поскольку этот шаг уже выполнен, выведется сообщение о пропуске загрузки:

```
> build.bat -p x64 -c Debug
```

Команда создает двоичный файл Python PCbuild ▶ `amd64` ▶ `python_d.exe`. Запустите его прямо из командной строки:

```
> amd64\python_d.exe

Python 3.9 (tags/v3.9:9cf67522, Oct 5 2020, 10:00:00)
[MSC v.1922 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Теперь вы находитесь в REPL скомпилированного двоичного файла CPython.

Чтобы скомпилировать окончательный двоичный файл, используйте следующую команду:

```
> build.bat -p x64 -c Release
```

Команда создаст двоичный файл PCbuild ▶ `amd64` ▶ `python.exe`.

ПРИМЕЧАНИЕ

Суффикс `_d` указывает, что CPython собран в отладочной конфигурации.

Двоичные файлы на Python.org скомпилированы в конфигурации профильной оптимизации (PGO, Profile-Guided Optimization). За дополнительной информацией о PGO обращайтесь к разделу «Профильная оптимизация» в конце этой главы.

Аргументы

Для build.bat доступны следующие аргументы.

ФЛАГ	НАЗНАЧЕНИЕ	ОЖИДАЕМОЕ ЗНАЧЕНИЕ
-р	Архитектура процессора	x64, Win32 (по умолчанию), ARM, ARM64
-с	Конфигурация сборки	Release (по умолчанию), Debug, PGInstrument или PGUpdate
-т	Цель сборки	Build (по умолчанию), Rebuild, Clean, CleanAll

Флаги

Некоторые дополнительные флаги, которые могут использоваться с build.bat.

ФЛАГ	НАЗНАЧЕНИЕ
-v	Режим расширенного вывода: вывод информационных сообщений в процессе сборки
-vv	Режим подробного расширенного вывода: вывод подробных сообщений в процессе сборки
-q	Тихий режим: вывод только предупреждений и ошибок в процессе сборки
-e	Загрузка и установка внешних зависимостей (по умолчанию)
-E	Отказ от загрузки и установки внешних зависимостей
--rpo	Сборка с профильной оптимизацией
--regen	Повторное генерирование всей грамматики и токенов (используется при обновлении языка)

Чтобы просмотреть полный список, выполните команду build.bat -h.

Компиляция из Visual Studio

В папке PCbuild находится файл решения Visual Studio PCbuild ▶ pcbuild.sln, предназначенный для сборки и анализа исходного кода CPython.

При загрузке файла решения вам будет предложено переопределить цель проектов внутри данного решения в соответствии с установленной версией

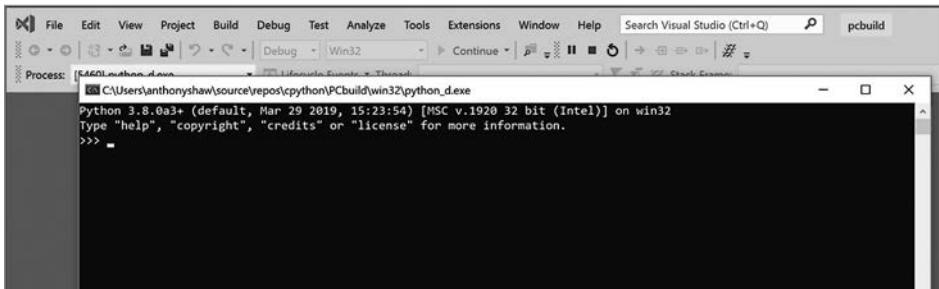
компилиатора C/C++. Visual Studio также ориентируется на установленную версию Windows SDK.

Не забудьте заменить версию Windows SDK последней установленной версией. Набор инструментов платформы тоже должен иметь новейшую версию. Если вы пропустили это окно, щелкните правой кнопкой мыши на файле решения в окне **Solutions and Projects** и выберите команду **Retarget Solution**.

Перейдите к окну **Build Configuration Manager**. Убедитесь в том, что в выпадающем списке **Active Solution Configuration** выбран вариант **Debug**, а в списке **Active Solution Platform** — вариант **x64** для 64-разрядной архитектуры процессора или **win32** для 32-разрядной.

Затем соберите CPython комбинацией клавиш **Ctrl+Shift+B** или командой **Build ▶ Build Solution**. Если вы получите ошибки, сообщающие об отсутствии Windows SDK, убедитесь в том, что вы выбрали правильные настройки цели в окне **Re-target Solution**. Также в меню **Пуск** должна быть папка **Windows Kits**, в которой находится **Windows Software Development Kit**.

В первый раз сборка может занять десять минут и более. После завершения сборки могут появиться предупреждения, на которые можно не обращать внимания. Чтобы запустить отладочную версию CPython, нажмите **F5**; CPython запустит REPL в отладочном режиме:



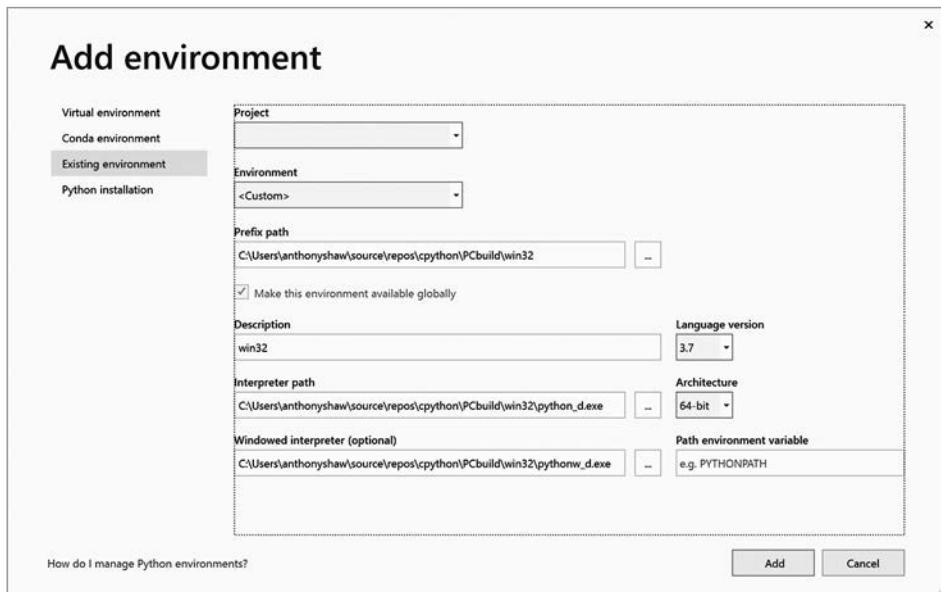
Чтобы запустить сборку в окончательной версии, замените конфигурацию сборки **Debug** на **Release** в верхней строке меню и снова выполните команду **Build ▶ Build Solution**. Теперь у вас имеются как отладочная, так и окончательная версии двоичного файла CPython в папке **PCbuild ▶ amd64**.

Чтобы настроить Visual Studio для открытия REPL с окончательной или отладочной версией, выберите команду **Tools ▶ Python ▶ Python Environments** в верхнем меню. На панели **Python Environments** щелкните по кнопке **Add Environment**, а затем

выберите двоичный файл в нужной версии. Отладочный двоичный файл завершается суффиксом _d.exe (например, python_d.exe или pythonw_d.exe).

Вам стоит использовать отладочный двоичный файл — он содержит поддержку отладки в Visual Studio и пригодится вам в ходе чтения книги.

В окне Add Environment выберите интерпретатор python_d.exe в поле Interpreter Path и оконный интерпретатор pythonw_d.exe в поле Windowed Interpreter (optional):

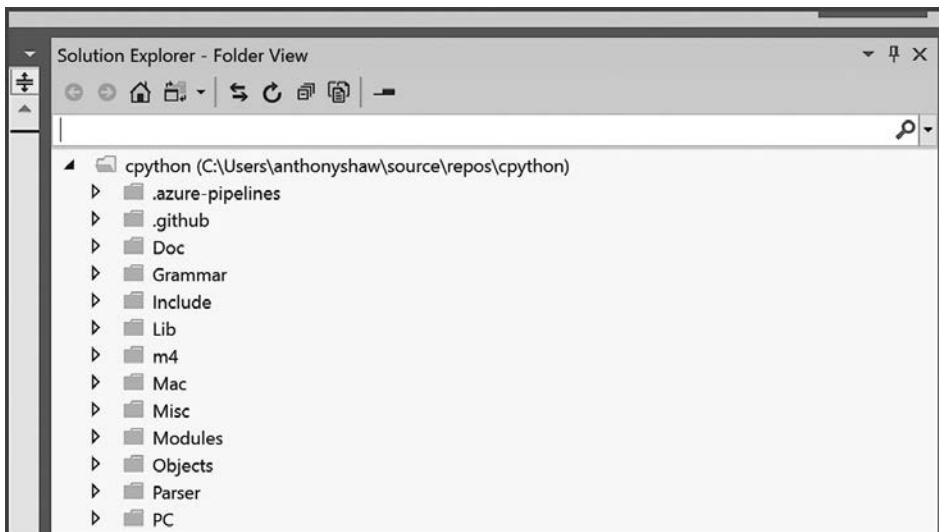


Запустите сеанс REPL при помощи кнопки Open Interactive Window в окне Python Environments. Откроется REPL скомпилированной версии Python:



В этой книге будут приводиться сессии REPL с примерами команд. Я рекомендую использовать отладочный двоичный файл для выполнения этих сессий REPL на случай, если вы захотите установить точки останова в коде.

Чтобы упростить навигацию в коде, в окне Solution переключитесь в режим Folder с помощью кнопки справа от значка Home:



ПРОФИЛЬНАЯ ОПТИМИЗАЦИЯ

Сборка на macOS, Linux и Windows поддерживает флаги **профильной оптимизации** (PGO). Режим PGO не был придуман командой Python; он встречается во многих компиляторах, включая те, что используются CPython.

PGO выполняет исходную компиляцию, после чего профилирует приложение, выполняя серию тестов. Затем профиль анализируется, и компилятор вносит в двоичный файл изменения для повышения производительности.

Для CPython на этапе профилирования запускается команда `python -m test --pgc`, которая выполняет регрессионные тесты, заданные в `Lib > test > libregtest > pgo.py`. Эти тесты были выбраны специально, потому что они используют часто встречающиеся модули расширений С и типы.

ПРИМЕЧАНИЕ

Процесс PGO занимает много времени, поэтому для ускорения компиляции я исключил его из списков рекомендуемых шагов, приводимых в книге.

Если вы хотите использовать специально скомпилированные версии CPython на продакшен, выполните команду `./configure` с флагом `--with-pgo` на Linux и macOS или добавьте флаг `--pgo` в `build.bat` на Windows.

Так как оптимизация привязана к платформе и архитектуре, для которой проводилось профилирование, профили PGO не могут совместно использоваться в разных операционных системах или архитектурах процессоров. Дистрибутивы CPython на Python.org уже прошли PGO, и если вы протестируете производительность скомпилированного в базовой конфигурации двоичного файла, он будет медленнее загруженного с Python.org.

Профильные оптимизации на Windows, macOS и Linux включают следующие проверки и улучшения:

- **Встроенные функции:** если функция часто вызывается из другой функции, она будет вложена (то есть скопирована в вызывающую функцию) для сокращения размера стека.
- **Прогнозирование виртуальных вызовов и вложенность:** если вызов виртуальной функции часто приходится на конкретную функцию, то PGO может вставить условно выполняемый прямой вызов этой функции. Это позволяет вложить непосредственный вызов.
- **Оптимизация распределения регистров:** на основании результатов профилирования PGO оптимизирует распределение регистров.
- **Оптимизация базовых блоков:** этот прием оптимизации позволяет разместить часто выполняемые базовые блоки (которые временно выполняются в заданных границах) в одной локальности или наборе страниц. Таким образом сводится к минимуму количество используемых страниц, что минимизирует лишние затраты памяти.
- **Оптимизация активных участков:** функции, на выполнение которых программа тратит большую часть времени, могут оптимизироваться для ускорения.

- **Оптимизация размещения функций:** после того, как PGO проанализирует граф вызовов, функции, которые обычно располагаются на одном пути выполнения, перемещаются в одну секцию скомпилированного приложения.
- **Оптимизация условных ветвлений:** PGO может проанализировать ветви принятия решений (такие, как `if ... else if` или `switch`) и выявить самый частый используемый путь. Например, если команда `switch` содержит 10 вариантов (путей) и один из них используется в 95 % случаев, то он перемещается в начало, чтобы выполняться раньше других в кодовом пути.
- **Отделение мертвых зон:** код, не вызываемый в процессе PGO, перемещается в отдельную секцию приложения.

ВЫВОДЫ

В этой главе вы узнали, как скомпилировать исходный код CPython в рабочий интерпретатор. Эта информация пригодится, когда мы займемся исследованием и адаптацией исходного кода.

Возможно, в ходе работы с CPython вам придется повторять действия по компиляции десятки и даже сотни раз. Если вы сможете адаптировать свою среду разработки и назначить сочетание клавиш для повторной компиляции, лучше сделать это прямо сейчас и сэкономить себе немало времени в будущем.

Грамматика и язык Python

Компилятор предназначен для преобразования одного языка в другой. Его можно сравнить с переводчиком: вы нанимаете переводчика, который слушает, как вы говорите на английском, а потом повторяет ваши слова на другом языке — скажем, на японском.

Для этого переводчик должен понимать грамматические структуры как исходного, так и целевого языка.

Некоторые компиляторы выполняют компиляцию в низкоуровневый машинный код, который может напрямую выполняться в системе. Другие компиляторы компилируют код в промежуточный язык, который выполняется виртуальной машиной.

Одним из факторов при выборе компилятора становятся требования к портируемости системы. Java и .NET CLR выполняют компиляцию в промежуточный язык, чтобы скомпилированный код мог переноситься между разными системными архитектурами. C, Go, C++ и Pascal компилируются в исполняемые двоичные файлы. Двоичный файл собирается для той платформы, на которой он компилировался.

Приложения Python обычно поставляются в виде исходного кода. Интерпретатор Python должен преобразовать исходный код Python и выполнить его в один этап. Среда выполнения CPython компилирует код при первом выполнении. Этот шаг остается незаметным для рядового пользователя.

Код Python не компилируется в машинный код. Он компилируется в низкоуровневый промежуточный язык, который называется **байт-кодом**. Байт-код хранится в файлах .рус и кэшируется для выполнения. Если одно приложение Python будет выполнять дважды без изменения исходного кода, то второй запуск будет проходить быстрее. Это связано с тем, что приложение запустит скомпилированный байт-код, вместо того чтобы каждый раз компилировать его заново.

ПОЧЕМУ СРУТНОН НАПИСАН НА С, А НЕ НА PYTHON

Буква С в СPython относится к языку программирования С — она означает, что этот дистрибутив Python написан на языке С.

В основном так и есть. Компилятор в СPython написан на чистом С. Тем не менее многие модули стандартной библиотеки написаны на чистом Python или комбинации С и Python.

Так почему же компилятор СPython написан на С, а не на Python?

Ответ основан на принципах работы компиляторов. Существуют две разновидности компиляторов:

1. **Автономные компиляторы** пишутся на том языке, который они компилируют (как компилятор Go). Для этого используется процесс, называемый **самозапуском** (bootstrapping).
2. **Компиляторы типа «исходный код в исходный код»** пишутся на другом языке, для которого уже существует компилятор.

Если вы создаете новый язык программирования с нуля, то вам понадобится исполняемое приложение для компиляции вашего компилятора! Для выполнения чего-либо нужен компилятор, поэтому при разработке новых языков они часто сначала пишутся на старых, более укоренившихся языках.

Также существуют инструменты, которые могут взять спецификацию языка и построить для него парсер; вы узнаете о них позднее в этой главе. Среди популярных «компиляторов компиляторов» можно выделить GNU Bison, Yacc и ANTLR.

СМ. ТАКЖЕ

Если вы захотите больше узнать о парсерах, ознакомьтесь с проектом Lark – парсером для контекстно-независимой грамматики, написанным на Python.

Отличным примером самозапуска компилятора служит язык программирования Go. Первый компилятор Go был написан на С; после того как код Go стал компилироваться, компилятор был переписан на Go.

¹ <https://github.com/lark-parser/lark>.

В отличие от этого, CPython сохраняет свое наследование С. Многие модули стандартной библиотеки (такие, как `sslmodule` или `socketsmodule`) переписаны на С для обращения к низкоуровневым API операционной системы.

API ядер Windows и Linux, предназначенные для создания сетевых сокетов¹, работы с файловой системой² или взаимодействия с экраном³, были написаны на С, поэтому логично, что уровень расширяемости был ориентирован на язык С. Стандартная библиотека Python и модули С будут рассмотрены далее.

Существует компилятор PyPy, написанный на Python, — он называется PyPy. На логотипе PyPy изображен уророс⁴, олицетворяющий природу самодостаточности компилятора.

ПРИМЕЧАНИЕ

В оставшейся части книги обозначение `./python` будет относиться к скомпилированной версии CPython. Тем не менее реальная команда будет зависеть от операционной системы.

В Windows:

```
> python.exe
```

В Linux:

```
$ ./python
```

В macOS:

```
$ ./python.exe
```

Другой пример кросс-компилятора для Python — Jython. Jython написан на Java и компилирует исходный код Python в байт-код Java. Подобно тому как CPython упрощает импортацию библиотек С и использование их из Python, Jython упрощает импортацию и использование модулей и классов Java.

Первым шагом создания компилятора становится определение языка. Например, следующий фрагмент не является валидным Python-кодом:

¹ <https://realpython.com/python-sockets/>.

² <https://realpython.com/working-with-files-in-python/>.

³ <https://realpython.com/python-gui-with-wxpython/>.

⁴ Змей, кусающий себя за хвост. — *Примеч. пер.*

```
def my_example() <str> :  
{  
    void* result = ;  
}
```

Чтобы компилятор мог обработать код языка, ему необходимы строгие правила грамматической структуры этого языка.

СПЕЦИФИКАЦИЯ ЯЗЫКА PYTHON

В исходном коде CPython содержится определение языка Python. Этот документ представляет собой эталонную спецификацию, используемую всеми интерпретаторами Python.

Спецификация содержит как формат, рассчитанный на чтение человеком, так и формат для машинного чтения. В документации содержится подробное объяснение языка Python с описанием разрешенных конструкций и поведения каждой команды.

Документация языка

Каталог Doc ▶ reference содержит разъяснение особенностей языка Python в формате reStructuredText. Из этих файлов составлено официальное справочное руководство Python на сайте docs.python.org/3/reference.

В каталоге Doc находятся файлы, необходимые для понимания всего языка, его структуры и ключевых слов:

 <i>cpython/Doc/reference</i>	
└── <i>compound_stmts.rst</i>	Составные операторы (if, while, for и т. д.) и определения функций
└── <i>datamodel.rst</i>	Объекты, значения и типы
└── <i>executionmodel.rst</i>	Структура программ Python
└── <i>expressions.rst</i>	Элементы выражений Python
└── <i>grammar.rst</i>	Базовая грамматика Python
└── <i>import.rst</i>	Система импортирования (import)
└── <i>index.rst</i>	Алфавитный указатель для справочника языка
└── <i>introduction.rst</i>	Введение в документацию языка
└── <i>lexical_analysis.rst</i>	Лексическая структура (строки, отступы, лексемы и ключевые слова)
└── <i>simple_stmts.rst</i>	Простые операторы (assert, import, return, yield и т. д.)
└── <i>toplevel_components.rst</i>	Описание способов выполнения кода Python (скрипты и модули)

Пример

В файле Doc ▶ reference ▶ compound_stmts.rst встречается простой пример определения оператора `with`.

Оператор `with` существует в нескольких формах; простейший вариант — реализация менеджера контекста¹ и вложенного блока кода:

```
with x():
    ...
```

Результат можно присвоить переменной при помощи ключевого слова `as`:

```
with x() as y:
    ...
```

Также можно объединять менеджеры контекстов в цепочку через запятую:

```
with x() as y, z() as jk:
    ...
```

Документация содержит спецификацию языка, предназначенную для чтения человеком. Спецификация, предназначенная для машинного чтения, расположается в одном файле Grammar ▶ python.gram.

Файл грамматики

Файл грамматики Python использует спецификацию в формате PEG (Parsing Expression Grammar). В файле грамматики могут использоваться следующие обозначения:

- * — повторение;
- + — минимум одно вхождение;
- [] — необязательные части;
- | — альтернативы;
- () — группировка.

¹ <https://dbader.org/blog/python-context-managers-and-with-statement>.

Для примера представим, как можно было бы определить чашку кофе:

- Необходима чашка.
- Чашка должна содержать как минимум одну порцию эспрессо, но может содержать несколько порций.
- В чашке может быть молоко, но необязательно.
- В чашке может быть вода, но необязательно.
- Если в чашке молоко, то оно может быть разного типа — жирное, обезжиренное, соевое и т. д.

В формате PEG заказ кофе может выглядеть так:

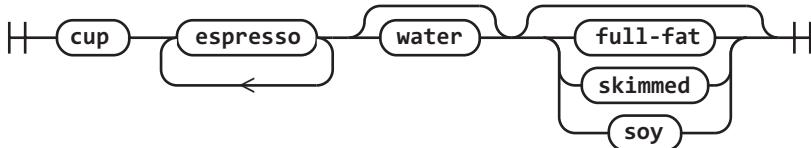
```
coffee: 'cup' ('espresso')+ ['water'] [milk]
milk: 'full-fat' | 'skimmed' | 'soy'
```

СМ. ТАКЖЕ

В CPython 3.9 исходный код CPython содержит два файла грамматики. Старая – контекстно-свободная грамматика, которая называется формой Бэкуса – Наура (BNF). В CPython 3.10 файл грамматики BNF (Grammar ▶ Grammar) был удален.

Форма BNF не привязана к Python и часто используется для записи грамматики во многих других языках.

В этой главе для наглядного представления грамматики будут использоваться *синтаксические диаграммы*. Синтаксическая диаграмма для команды coffee выглядит так:



На синтаксической диаграмме каждая возможная комбинация должна располагаться на одной линии слева направо. Необязательные компоненты можно обойти, а некоторые компоненты могут образовывать циклы.

Пример: оператор while

Существует несколько разновидностей оператора `while`. Простейший вариант использования — когда за завершающим двоеточием (`:`) следует блок кода:

```
while finished == True:
    do_things()
```

В альтернативном варианте используется оператор присваивания, которому в грамматике соответствует обозначение `named_expression`. Эта новая возможность появилась в Python 3.8:

```
while letters := read(document, 10):
    print(letters)
```

Также за оператором `while` может следовать оператор `else` и блок кода:

```
while item := next(iterable):
    print(item)
else:
    print("Iterable is empty")
```

Проведя поиск `while_stmt` в файле грамматики, вы увидите определение:

```
while_stmt[stmt_ty]:
| 'while' a=named_expression ':' b=block c=[else_block] ...
```

Символы в кавычках образуют строковый литерал, который называется *терминалом* (terminal). В частности, терминалы используются для распознавания ключевых слов.

В этих двух строках содержатся ссылки на два других определения:

1. `block` обозначает блок кода с одним или несколькими операторами.
2. `named_expression` обозначает простое выражение или выражение присваивания.

Если представить оператор `while` в виде синтаксической диаграммы, она будет выглядеть так:



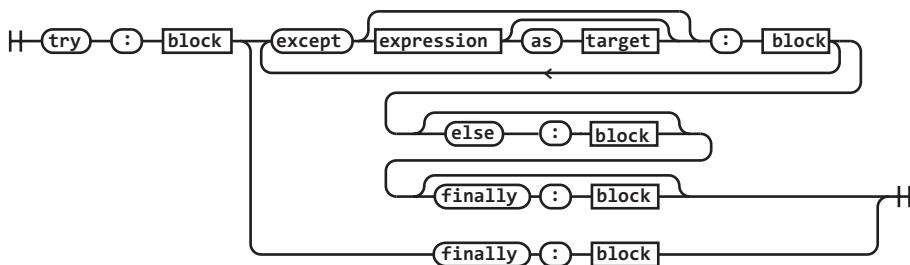
Рассмотрим более сложный пример. Оператор `try` определяется в грамматике так:

```
try_stmt[stmt_ty]:  
    | 'try' ':' b=block f=finally_block { _Py_Try(b, NULL, NULL, f, EXTRA) }  
    | 'try' ':' b=block ex=except_block+ el=[else_block] f=[finally_block]..  
except_block[excepthandler_ty]:  
    | 'except' e=expression t=['as' z=target { z }] ':' b=block {  
        _Py_ExceptHandler(e, (t) ? ((expr_ty) t)->v.Name.id : NULL, b, ...  
    }  
    | 'except' ':' b=block { _Py_ExceptHandler(NULL, NULL, b, EXTRA) }  
finally_block[asdl_seq*]: 'finally' ':' a=block { a }
```

У оператора `try` есть два варианта использования:

1. `try` только с оператором `finally`.
2. `try` с одним или несколькими блоками `except`, за которыми может следовать необязательный блок `else`, а после него необязательный `finally`.

Эти же варианты использования на синтаксической диаграмме:



Оператор `try` является хорошим примером более сложной структуры.

Если вы захотите понять язык Python на более глубоком уровне, прочитайте определение грамматики в [Grammar ▶ python.gram](#).

ГЕНЕРАТОР ПАРСЕРОВ

Сам файл грамматики никогда не используется компилятором Python. Вместо этого генератор парсеров читает файл и генерирует парсер. Если в файл грамматики будут внесены изменения, вам придется заново сгенерировать парсер и перекомпилировать CPython.

В Python 3.9 парсер CPython был переписан из автомата, заданного в табличной форме (модуль `pgen`), в контекстный парсер грамматики. В Python 3.9 старый парсер доступен в командной строке (флаг `-X oldparser`), а в Python 3.10 он полностью удален. В книге речь идет о новом парсере, реализованном в версии 3.9.

ПОВТОРНОЕ ГЕНЕРИРОВАНИЕ ГРАММАТИКИ

Чтобы увидеть в действии `pegen` — новый генератор PEG, появившийся в CPython 3.9, — можно изменить часть грамматики Python. Проведите в `Grammar ▶ python.gram` поиск `small_stmt`, чтобы увидеть определение простых операторов:

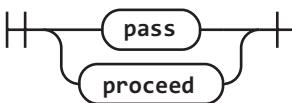
```
small_stmt[stmt_ty] (memo):
| assignment
| e=star_expressions { _Py_Expr(e, EXTRA) }
| &'return' return_stmt
| &('import' | 'from') import_stmt
| &'raise' raise_stmt
| 'pass' { _Py_Pass(EXTRA) }
| &'del' del_stmt
| &'yield' yield_stmt
| &'assert' assert_stmt
| 'break' { _Py_Break(EXTRA) }
| 'continue' { _Py_Continue(EXTRA) }
| &'global' global_stmt
| &'nonlocal' nonlocal_stmt
```

Строка `'pass' { _Py_Pass(EXTRA) }` относится к оператору `pass`:



Измените эту строку, чтобы в качестве ключевых слов принимались терминалы (ключевые слова) `'pass'` или `'proceed'`; для этого добавьте конструкцию выбора `|` и литерал `'proceed'`:

```
| ('pass' | 'proceed') { _Py_Pass(EXTRA) }
```



Соберите заново файлы грамматики. В поставку СPython включаются скрипты для автоматизации повторного генерирования грамматики.

В macOS и Linux выполните цель `make regen-pegen`:

```
$ make regen-pegen
```

В Windows откройте командную строку из каталога PCBuild и выполните `build.bat` с флагом `--regen`:

```
> build.bat --regen
```

Должно появиться сообщение о том, что новый файл `Parser ▶ pegen ▶ parse.c` был сгенерирован заново.

С заново сгенерированной таблицей парсера при перекомпиляции Python будет использоваться новый синтаксис. Выполните тот же алгоритм компиляции, который был приведен для вашей операционной системы в предыдущей главе.

Если код был скомпилирован успешно, вы можете выполнить новый двоичный файл СPython и запустить REPL.

Теперь попробуйте определить функцию в REPL. Вместо команды `pass` используйте альтернативное ключевое слово `proceed`, которое было скомпилировано в грамматике Python:

```
$ ./python
Python 3.9 (tags/v3.9:9cf67522, Oct 5 2020, 10:00:00)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def example():
... proceed
...
>>> example()
```

Поздравляю — вы изменили синтаксис СPython и скомпилировали собственную версию СPython!

На следующем этапе будут рассмотрены лексемы и их отношение к грамматике.

Лексемы

Наряду с файлом грамматики в папке `Grammar` содержится файл `Grammar ▶ Tokens`, в котором хранятся все уникальные типы, присутствующие в листовых узлах (*leaf node*) в дереве синтаксического разбора. Каждая лексема обладает именем и сгенерированным уникальным идентификатором. Имена упрощают обращения к лексемам в `tokenizer`.

ПРИМЕЧАНИЕ

Файл `Grammar ▶ Tokens` – одна из новых возможностей Python 3.8.

Например, левая круглая скобка называется `LPAR`, а символ «точка с запятой» — `SEMI`. Эти лексемы будут разбираться далее в книге:

```
LPAR      '('
RPAR      ')'
LSQB      '['
RSQB      ']'
COLON    ':'
COMMA    ','
SEMI      ';'
```

Как и в случае с файлом `Grammar`, при изменении файла `Grammar ▶ Tokens` необходимо заново запустить `pegen`.

Чтобы увидеть лексемы в действии, можно воспользоваться модулем `tokenizer` в CPython.

ПРИМЕЧАНИЕ

Модуль `tokenizer`, написанный на Python, является служебным модулем. Реальный парсер Python использует другой способ распознавания лексем.

Создайте простой Python-скрипт с именем `test_tokens.py`:

cpython-book-samples ▶ 13 ▶ test_tokens.py

```
# Demo application
def my_function():
    proceed
```

Передайте файл `test_tokens.py` модулю стандартной библиотеки с именем `tokenize`. На экран выводится список лексем с указанием их позиции (строк и столбцов). Используйте флаг `-e` для вывода имен конкретных лексем:

```
$ ./python -m tokenize -e test_tokens.py
```

0,0-0,0:	ENCODING	'utf-8'
1,0-1,14:	COMMENT	'# Demo application'
1,14-1,15:	NL	'\n'
2,0-2,3:	NAME	'def'
2,4-2,15:	NAME	'my_function'
2,15-2,16:	LPAR	'('
2,16-2,17:	RPAR	')'
2,17-2,18:	COLON	::
2,18-2,19:	NEWLINE	'\n'
3,0-3,3:	INDENT	' '
3,3-3,7:	NAME	'proceed'
3,7-3,8:	NEWLINE	'\n'
4,0-4,0:	DEDENT	' '
4,0-4,0:	ENDMARKER	''

В первой колонке выводится интервал с номерами строк и столбцов. Вторая содержит имя лексемы, а в последней выводится значение лексемы.

В выводе модуль `tokenize` подставил ряд подразумеваемых лексем:

- `ENCODING` для `utf-8`;
- `DEDENT` для закрытия объявления функции;
- `ENDMARKER` для завершения файла;
- пустую строку в конце.

В конце исходных файлов Python рекомендуется оставлять пустую строку. Если не сделать этого, то CPython добавит ее за вас.

Модуль `tokenize` написан на чистом Python и находится в файле `Lib ▶ tokenize.py`.

Чтобы увидеть подробный вывод парсера С, можно запустить отладочную версию Python с флагом `-d`. Запустите скрипт `test_tokens.py`, созданный ранее, следующей командой:

```
$ ./python -d test_tokens.py  
> file[0-0]: statements? $  
  > statements[0-0]: statement+  
    > _loop1_11[0-0]: statement  
      > statement[0-0]: compound_stmt  
...  
  + statements[0-10]: statement+ succeeded!  
+ file[0-11]: statements? $ succeeded!
```

Как видите, `proceed` выделяется как ключевое слово. В следующей главе вы увидите, как при выполнении двоичного файла Python используется `tokenizer` и что происходит в дальнейшем для выполнения вашего кода.

Чтобы очистить код, отмените изменения в `Grammar ▶ python.gram`, снова сгенерируйте грамматику, а затем проведите очистку сборки и повторную компиляцию.

В macOS и Linux это делается так:

```
$ git checkout -- Grammar/python.gram  
$ make regen-pegen  
$ make -j2 -s
```

В Windows используются следующие команды:

```
> git checkout -- Grammar/python.gram  
> build.bat --regen  
> build.bat -t CleanAll  
> build.bat -t Build
```

ВЫВОДЫ

В этой главе вы познакомились с определениями грамматики Python и генератором парсеров. В следующей главе на основе этих знаний будет построен более сложный элемент синтаксиса — оператор «почти равно».

На практике любые изменения в грамматике Python необходимо тщательно продумывать и обсуждать. Для этого есть две причины:

1. Избыток языковых средств или сложная грамматика будут противоречить кредо Python как простого и удобочитаемого языка.
2. Изменения грамматики создают обратные несовместимости, которые усложняют работу всех разработчиков.

Если ключевой Python-разработчик предлагает изменения в грамматике, они должны быть оформлены в виде документа **PEP** (Python Enhancement Proposal). Все PEP нумеруются и включаются в индекс PEP. PEP 5 документирует рекомендации для развития языка и указывает, что изменения должны предлагаться в виде PEP.

Предлагаемые, отклоненные и принятые PEP для будущих версий CPython можно найти в индексе PEP¹. Участники, не входящие в группу ключевых разработчиков, также могут предлагать изменения в языке через список рассылки `python-ideas`².

Когда по поводу PEP будет достигнут консенсус, а черновая версия примет окончательную форму, руководящий совет должен принять или отклонить предложение. Мандат руководящего совета, определенный в PEP 13, утверждает, что члены совета должны работать над «поддержанием качества и стабильности языка Python и интерпретатора CPython».

¹ <https://www.python.org/dev/peps/>.

² <https://www.python.org/community/lists/>.

Конфигурация и ввод

После рассмотрения грамматики Python давайте разберемся, как код переходит в исполняемое состояние.

Существует много способов выполнения Python-кода в CPython. Несколько наиболее популярных:

1. Выполнение команды `python -c` со строкой Python.
2. Выполнение команды `python -m` с именем модуля.
3. Выполнение команды `python <файл>` с путем к файлу, содержащему Python-код.
4. Передача Python-кода исполняемому файлу `python` через `stdin` — например, `cat <файл> | python`.
5. Запуск REPL и выполнение команд по очереди.
6. Использование С API и Python как встроенной среды.

СМ. ТАКЖЕ

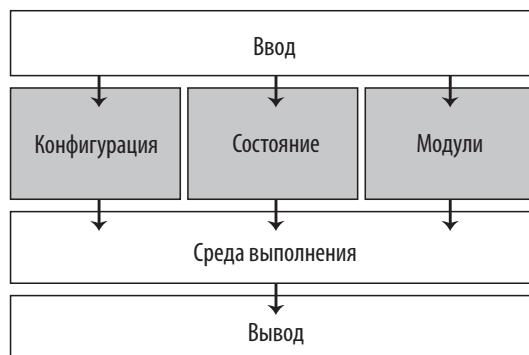
Обилие способов выполнения скриптов Python ошеломляет. За дополнительной информацией о выполнении скриптов Python обратитесь к статье «How to Run Your Python Scripts»¹ на сайте Real Python.

¹ <https://realpython.com/run-python-scripts/>.

Для выполнения любого Python-кода интерпретатору понадобятся три элемента:

1. Модуль, который должен быть выполнен.
2. Состояние для хранения информации (например, переменных).
3. Конфигурация (активные настройки).

С этими тремя компонентами интерпретатор может выполнить код и предоставить вывод:



ПРИМЕЧАНИЕ

По аналогии с руководством по стилю PEP 8 для Python-кода существует руководство PEP 7 для кода C в CPython. Оно включает следующие стандарты выбора имен для исходного кода C:

- Префикс `Py` для общедоступных функций (и никогда для статических).
- Префикс `Py_` для глобальных служебных процедур – таких, как `Py_FatalError`. Конкретные группы функций (например, API конкретных объектных типов) должны использовать более длинный префикс – например, `PyString_` для строковых функций.
- Имена общедоступных функций и переменных должны записываться в смешанном регистре с разделением слов нижним подчеркиванием – например, `PyObject_GetAttr()`, `Py_BuildValue()` и `PyExc_TypeError()`.

- Префикс `_Py` должен быть зарезервирован для внутренних функций, которые должны быть видимыми для загрузчика, например `_PyObject_Dump()`.
- Имена макросов должны иметь префикс в смешанном регистре, все остальные слова должны быть в верхнем регистре и разделены нижним подчеркиванием, например `PyString_AS_STRING` и `Py_PRINT_RAW`.

Существует не так много программных инструментов для проверки соблюдения руководства PEP 7 (в отличие от PEP 8). Эта задача должна решаться разработчиками в ходе код-ревью. Как и любой процесс, управляемый человеком, такой код-ревью не избавлен от ошибок, поэтому вам, скорее всего, попадется код, не соответствующий PEP 7. Единственный инструмент автоматизации этого процесса, входящий в дистрибутив, – скрипт `smelly.py`, который можно выполнить командой `make smelly` в Linux или macOS либо запустить из командной строки:

```
$ ./python Tools/scripts/smelly.py
```

Скрипт выдает сообщения об ошибках для всех символических имен в `libpython` (общей библиотеке CPython), не начинающихся с `Py` или `_Py`.

КОНФИГУРАЦИЯ СОСТОЯНИЯ

Прежде чем выполнять какой-либо Python-код, среда выполнения CPython сначала устанавливает конфигурацию среды выполнения Python с учетом всех параметров, переданных пользователем.

Согласно определению из PEP 587, данные конфигурации среды выполнения хранятся в трех местах:

1. `PyPreConfig` – данные конфигурации для предварительной инициализации.
2. `PyConfig` – данные конфигурации среды выполнения.
3. Скомпилированная конфигурация интерпретатора CPython.

Обе структуры данных, `PyPreConfig` и `PyConfig`, определяются в `Include > cpython > initconfig.h`.

Конфигурация предварительной инициализации

Конфигурация предварительной инициализации существует отдельно от конфигурации среды выполнения, так как ее свойства относятся к операционной системе или окружению пользователя.

PyPreConfig имеет три основные функции:

1. Настройка распределителя памяти Python.
2. Настройка локали LC_CTYPE системной или предпочтаемой пользователем локалью.
3. Настройка режима UTF-8 (PEP 540).

Тип PyPreConfig содержит следующие поля (все они относятся к типу `int`):

- `allocator`: выбор распределителя памяти — например, `PYMEM_ALLOCATOR_MALLOC`. Чтобы получить дополнительную информацию о распределителе памяти, выполните команду `./configure --help`.
- `configure_locale`: `LC_CTYPE` присваивается локаль, выбранная пользователем. Если значение равно 0, то `coerce_c_locale` и `coerce_c_locale_warn` также присваивается 0.
- `coerce_c_locale`: если поле содержит 2, принудительно ввести локаль C. Если значение равно 1, прочитать локальный контекст `LC_CTYPE`, чтобы принять решение о его принудительном использовании.
- `coerce_c_locale_warn`: если значение отлично от нуля, при принудительном применении локали C выдается предупреждение.
- `dev_mode`: включение режима разработки.
- `isolated`: включение изолированного режима. `sys.path` не содержит ни каталог скрипта, ни каталог веб-пакетов пользователя.
- `legacy_windows_fs_encoding` (только для Windows): если значение отлично от нуля, режим UTF-8 отключается, а для кодировки файловой системы Python устанавливается значение `mbcS`.
- `parse_argv`: если значение отлично от нуля, использовать аргументы командной строки.
- `use_environment`: если значение больше нуля, использовать переменные среды.
- `utf8_mode`: если значение отлично от нуля, включить режим UTF-8.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к PyPreConfig.

ФАЙЛ	НАЗНАЧЕНИЕ
Python ▶ initconfig.c	Загружает данные конфигурации из системных параметров среды и объединяет их с флагами командной строки
Include ▶ cpython ▶ initconfig.h	Определяет структуру данных конфигурации инициализации

СТРУКТУРА ДАННЫХ КОНФИГУРАЦИИ СРЕДЫ ВЫПОЛНЕНИЯ

Вторая фаза конфигурации — конфигурация среды выполнения. Структура данных конфигурации среды выполнения в PyConfig содержит несколько значений, включая следующие:

- Флаги среды выполнения для таких режимов, как отладка и оптимизация.
- Режим выполнения (например, файл сценария, `stdin` или модуль).
- Расширенные параметры, задаваемые с ключом `-X <параметр>`.
- Переменные среды для настроек среды выполнения.

Данные конфигурации используются средой выполнения CPython для включения и отключения отдельных возможностей.

Настройка конфигурации среды выполнения в командной строке

Python также включает ряд параметров интерфейса командной строки¹. Например, в CPython существует режим **подробного вывода**, предназначенный прежде всего для отладки CPython разработчиками.

Режим подробного вывода включается флагом `-v`. Python выводит сообщения на экран при загрузке модулей:

¹ <https://docs.python.org/3/using/cmdline.html>.

```
$ ./python -v -c "print('hello world')"

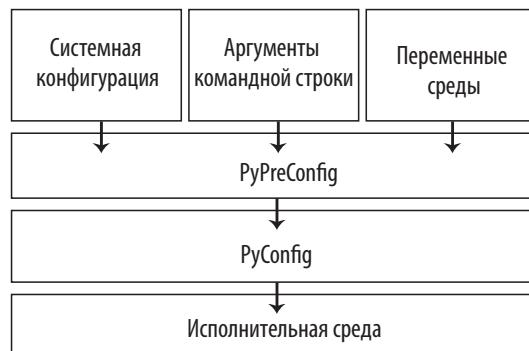
# installing zipimport hook
import zipimport # builtin
# installed zipimport hook
...
```

Вы увидите сотню и более строк с сообщениями об импортировании веб-пакетов пользователя и всех остальных событий, происходящих в системной среде.

Так как конфигурацию среды выполнения можно задать несколькими способами, у настроек конфигурации существуют разные приоритеты. Порядок приоритетов для режима подробного вывода:

1. Для `config->verbose` в исходном коде жестко фиксируется значение `-1`.
2. Переменная среды `PYTHONVERBOSE` используется для задания значения `config->verbose`.
3. Если переменная среды не существует, используется значение по умолчанию `-1`.
4. В функции `config_parse_cmdline()` из файла Python `initconfig.c` флаг командной строки (если он указан) используется для присваивания значения.
5. Значение копируется в глобальную переменную `Py_VerboseFlag` при вызове `_Py_GetGlobalVariablesAsDict()`.

Для всех значений `PyConfig` существует один порядок действий и последовательность приоритетов:



Просмотр флагов времени выполнения

У интерпретаторов CPython имеется набор **флагов времени выполнения**. Эти флаги управляют расширенными возможностями по включению/отключению поведения, специфического для CPython. В сеансе Python можно обращаться к флагам времени выполнения (например, режиму подробного вывода и тихому режиму) через кортеж с именем `sys.flags`.

Все флаги -X доступны в словаре `sys._xoptions`:

```
$ ./python -X dev -q

>>> import sys
>>> sys.flags
sys.flags(debug=0, inspect=0, interactive=0, optimize=0,
dont_write_bytecode=0, no_user_site=0, no_site=0,
ignore_environment=0, verbose=0, bytes_warning=0,
quiet=1, hash_randomization=1, isolated=0,
dev_mode=True, utf8_mode=0)
>>> sys._xoptions
{'dev': True}
```

КОНФИГУРАЦИЯ СБОРКИ

Наряду с конфигурацией среды выполнения в файле `Include` ▶ `cpython` ▶ `initconfig.h` также существует конфигурация сборки в файле `pyconfig.h` корневой папки. Этот файл создается динамически на шаге `./configure` процесса сборки в macOS и Linux либо `build.bat` в Windows.

Чтобы просмотреть конфигурацию сборки, выполните следующую команду:

```
$ ./python -m sysconfig

Platform: "macosx-10.15-x86_64"
Python version: "3.9"
Current installation scheme: "posix_prefix"

Paths:
    data = "/usr/local"
    include = "/Users/anthonyshaw/CLionProjects/cpython/Include"
    platinclude = "/Users/anthonyshaw/CLionProjects/cpython"
...
...
```

Свойства конфигурации сборки представляют собой значения времени компиляции, используемые для выбора дополнительных модулей, которые компонуются с двоичным файлом. Например, отладчики, инструментальные библиотеки и распределители памяти задаются во время компиляции.

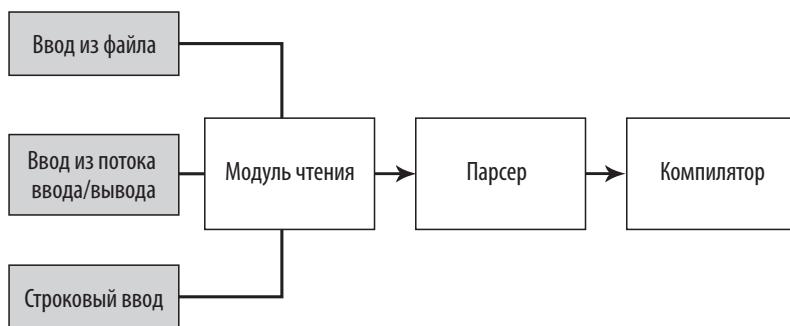
Когда вы выполните три этапа конфигурации, интерпретатор CPython сможет получить входные данные и преобразовать текст в исполняемый код.

СБОРКА МОДУЛЯ ИЗ ВХОДНЫХ ДАННЫХ

Любой код перед выполнением необходимо скомпилировать в модуль из входных данных. Как упоминалось ранее, входные данные могут существовать в разных формах:

- локальные файлы и пакеты;
- потоки ввода/вывода (например, `stdin` или канал в памяти);
- строки.

Входные данные читаются, передаются парсеру, а затем компилятору:



Из-за этой гибкости большая часть исходного кода CPython относится к обработке входных данных парсера CPython.

Исходные файлы

Существуют четыре основных файла для взаимодействия с интерфейсом командной строки.

ФАЙЛ	НАЗНАЧЕНИЕ
Lib ▶ runpy.py	Модуль стандартной библиотеки для импортирования и выполнения модулей Python
Modules ▶ main.c	Функции-обертки для выполнения внешнего кода (например, из файла, модуля или входного потока)
Programs ▶ python.c	Точка входа для исполняемого файла python в Windows, Linux и macOS. Служит оберткой для Modules/main.c
Python ▶ pythonrun.c	Функции-обертки внутренних API, написанные на C и предназначенные для обработки ввода из командной строки

Чтение файлов и ввод

Как только для CPython настроены среда выполнения и аргументы командной строки, он может загрузить код, который требуется выполнить. Эту задачу решает функция `PyMain_Main()` из `Modules ▶ main.c`.

Теперь CPython выполняет заданный код с любыми параметрами, указанными во вновь созданном экземпляре `PyConfig`.

Получение входных данных из командной строки

CPython может запустить небольшое приложение Python из командной строки с ключом `-c`. Например, рассмотрим, что произойдет при выполнении `print(2 ** 2)`:

```
$ ./python -c "print(2 ** 2)"
```

4

Сначала `PyMain_Run_Command()` выполняется внутри `Modules ▶ main.c`; при этом команда из `-c` передается в аргументе с типом `C wchar_t*`.

Затем `PyMain_Run_Command()` передает объект Python с набором байтов функции `PyRun_SimpleStringFlags()` для выполнения.

`PyRun_SimpleStringFlags()` является частью файла `Python ▶ pythonrun.c`. Его цель — преобразование строки в модуль Python и его последующая передача для выполнения.

ПРИМЕЧАНИЕ

Тип `wchar_t*` часто используется в CPython как низкоуровневый тип для хранения данных в Юникоде, так как размер типа позволяет хранить символы UTF-8.

При преобразовании `wchar_t*` в строку Python файл `Objects` ▶ `unicodeobject.c` содержит вспомогательную функцию `PyUnicode_FromWideChar()`, которая возвращает строку в Юникоде. Затем она кодируется в UTF-8 функцией `PyUnicode_AsUTF8String()`.

Строки в кодировке Юникод в Python подробно рассматриваются в разделе «Тип строк в Юникоде» главы «Объекты и типы».

Модуль Python должен иметь точку входа `__main__` для возможности его автономного выполнения, и `PyRun_SimpleStringFlags()` неявно создает эту точку входа.

После того как функция `PyRun_SimpleStringFlags()` создаст модуль и словарь, она вызывает `PyRun_StringFlags()`. `PyRun_SimpleStringFlags()` создает фиктивное имя файла, после чего вызывает парсер Python для создания дерева абстрактного синтаксиса¹ (AST, abstract syntax tree) из строки и возвращает модуль. Вы узнаете больше об AST в следующей главе.

ПРИМЕЧАНИЕ

Модули Python – структуры данных, используемые для передачи синтаксически разобранного кода компилятору. Структура данных С для модулей Python – `mod_ty` – определяется в `Include` ▶ `Python-ast.h`.

Получение входных данных из локального модуля

Другой способ выполнения команд Python основан на использовании параметра `-m` с именем модуля. Типичный пример – команда `python -m unittest`, которая выполняет модуль `unittest` из стандартной библиотеки.

¹ Иначе – абстрактное синтаксическое дерево. – *Примеч. ред.*

Возможность выполнения модулей как скриптов была изначально предложена в PEP 338, а стандарт явного относительного импортирования был определен в PEP 366.

Флаг `-m` означает, что внутри пакета модуля должно быть выполнено все, что находится внутри точки входа (`__main__`)¹. Он также подразумевает, что вы хотите найти `sys.path` для модуля с заданным именем.

Именно благодаря механизму поиска в библиотеке импортирования (`importlib`) вам не приходится запоминать, где в вашей файловой системе хранится модуль `unittest`.

CPython импортирует модуль стандартной библиотеки `rumpy` и выполняет его с использованием `PyObject_Call()`. Импортирование осуществляется с использованием API-функции языка C — `PyImport_ImportModule()`, находящейся в файле Python `import.c`.

ПРИМЕЧАНИЕ

Если в Python вы захотите получить атрибут для имеющегося объекта, следует вызвать функцию `getattr()`. В C API этому вызову соответствует функция `PyObject_GetAttrString()`, находящаяся в файле `Objects ▶ object.c`.

Если вы захотите использовать вызываемый (`callable`) объект, поставьте круглые скобки либо вызовите свойство `__call__()` объекта Python. `__call__()` реализуется внутри `Objects ▶ object.c`:

```
>>> my_str = "hello, world"
>>> my_str.upper()
'HELLO, WORLD'
>>> my_str.upper.__call__()
'HELLO, WORLD'
```

Модуль `rumpy` написан на чистом Python и находится в файле `Lib ▶ rumpy.py`.

Выполнение команды `python -m <модуль>` эквивалентно выполнению команды `python -m rumpy <модуль>`. Модуль `rumpy` был создан для абстрагирования процесса поиска и выполнения модулей в операционной системе.

¹ <https://realpython.com/python-main-function/>.

Для запуска целевого модуля `rumpy` выполняет три операции:

1. Для предоставленного имени модуля вызывается `__import__()`.
2. Переменной `__name__` (имя модуля) присваивается пространство имен `__main__`.
3. Модуль выполняется в пространстве имен `__main__`.

Модуль `rumpy` также поддерживает выполнение каталогов и ZIP-файлов.

Входные данные из скрипта или стандартного ввода

Если первым аргументом `python` является имя файла (например, `python test.py`), то CPython открывает дескриптор файла и передает его функции `PyRun_SimpleFileExFlags()` из Python ► `pythonrun.c`.

Существуют три разных способа выполнения этой функции в зависимости от пути:

1. Если указан путь до файла `.pyc`, вызывается `run_pyc_file()`.
2. Если указан путь до файла со скриптом (`.py`), выполняется `PyRun_FileExFlags()`.
3. Если указан путь до потока ввода `stdin` (потому что пользователь выполнил перенаправление `<команда> | python`), то `stdin` рассматривается как дескриптор файла и выполняется `PyRun_FileExFlags()`.

Для `stdin` и базовых файлов со скриптами CPython передает дескриптор файла функции `PyRun_FileExFlags()`, находящейся в файле Python ► `pythonrun.c` file.

Функция `PyRun_FileExFlags()` делает примерно то же, что и `PyRun_SimpleStringFlags()`. CPython загружает дескриптор файла в `PyParser_ASTFromPyObject()`.

Как и в случае с `PyRun_SimpleStringFlags()`, после того как `PyRun_FileExFlags()` создает модуль Python из файла, он передает его в `run_mod()` для выполнения.

Входные данные из скомпилированного байт-кода

Если пользователь выполняет `python` с путем к файлу `.pyc`, то вместо того, чтобы загрузить его как простой текстовый файл и распарсить, CPython предполагает, что файл `.pyc` содержит программный объект, записанный на диск.

В `PyRun_SimpleFileExFlags()` есть условие для передачи пользователем пути к файлу `.pyc`.

Функция `run_pyc_file()` из файла `Python ▶ pythonrun.c` осуществляет **маршалинг** объектного кода из файла `.pyc` с использованием дескриптора файла.

Структура данных объектного кода на диске используется компилятором CPython для кэширования скомпилированного кода, чтобы его не приходилось заново парсить при каждом вызове скрипта.

ПРИМЕЧАНИЕ

Термином «маршалинг» обозначается копирование содержимого файла в память и преобразование его в конкретную структуру данных.

Когда объектный код маршилизован в память, он передается функции `run_eval_code_obj()`, которая вызывает `Python ▶ ceval.c` для выполнения.

ВЫВОДЫ

В этой главе вы узнали, как загружаются многочисленные параметры конфигурации Python и как код передается на вход интерпретатора.

Благодаря возможности выбора ввода Python отлично подходит для разнообразных приложений, например:

- утилит командной строки;
- постоянно работающих сетевых приложений (например, веб-серверов);
- коротких скриптов компоновки.

Возможность разнообразной настройки свойств конфигурации в Python создает некоторую сложность. Например, если вы протестирували Python-приложение в среде с Python 3.8 и оно было выполнено правильно, а в другой среде произошла ошибка, вам придется выяснить, какие настройки отличались.

Это означает, что придется проверить переменные среды, флаги среды выполнения и даже свойства системной конфигурации.

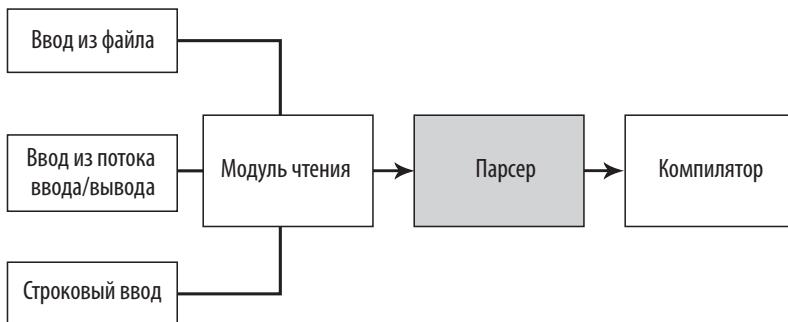
Свойства времени компиляции, находящиеся в системной конфигурации, могут различаться между дистрибутивами Python. Например, дистрибутив Python 3.8 для macOS, загруженный с [Python.org](https://www.python.org), по умолчанию использует значения, отличные от тех, которые использует Python 3.8 из Homebrew или Anaconda.

При любом способе ввода на выходе вы получаете модуль Python. В следующей главе вы увидите, как это происходит.

Лексический анализ и парсинг с использованием синтаксических деревьев

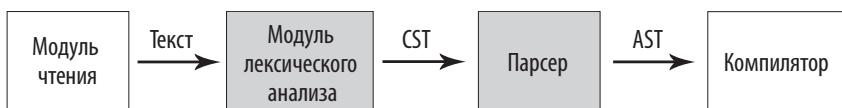
В предыдущей главе вы узнали, как происходит чтение Python-кода из разных источников. Затем его необходимо преобразовать в структуру, которая может использоваться компилятором.

Эта стадия называется **парсингом** (parsing):



В этой главе вы узнаете, как текст разбирается в логическую структуру, которая может быть обработана компилятором.

При парсинге кода в CPython используются две структуры: конкретное синтаксическое дерево (CST) и абстрактное синтаксическое дерево (AST):



Процесс парсинга состоит из двух частей:

1. Создание конкретного синтаксического дерева (CST, concrete syntax tree) с использованием парсера/токенизатора (parser-tokenizer) или лексического анализатора (lexer).
2. Создание абстрактного синтаксического дерева (AST, abstract syntax tree) из конкретного синтаксического дерева с использованием парсера.

Эти два шага — распространенные парадигмы, используемые во многих языках программирования.

ГЕНЕРИРОВАНИЕ КОНКРЕТНОГО СИНТАКСИЧЕСКОГО ДЕРЕВА

CST, иногда называемое **деревом разбора**, — упорядоченная древовидная структура с корнем, представляющая код в контекстно-свободной грамматике.

CST строится **токенизатором** и **парсером**. Парсер рассматривался в главе «Язык Python и грамматика». Вывод парсера представляет собой таблицу разбора в форме детерминированного конечного автомата (ДКА), описывающую возможные состояния контекстно-свободной грамматики.

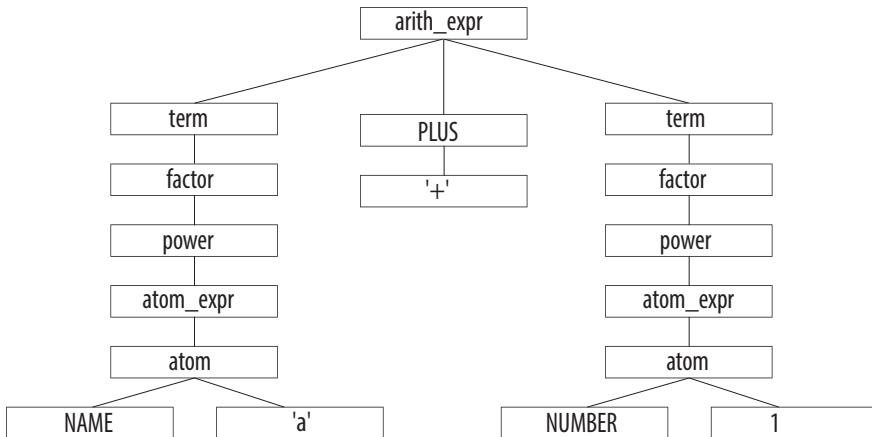
СМ. ТАКЖЕ

Создатель Python Гвидо ван Россум разработал контекстную грамматику для использования в CPython 3.9 как альтернативу для LL(1) – грамматики, использованной в предыдущих версиях CPython. Новая грамматика называется PEG (Parser Expression Grammar).

Парсер PEG стал доступным в Python 3.9. В Python 3.10 старая грамматика LL(1) была полностью исключена.

В главе «Язык Python и грамматика» рассматривались некоторые типы выражений, такие как `if_stmt` и `with_stmt`. CST представляет грамматические обозначения вида `if_stmt` как ветви, а лексемы и терминалы — как листовые узлы.

Например, арифметическое выражение `a+1` преобразуется в следующее дерево CST:



Арифметическое выражение представляется тремя основными ветвями: левой ветвью, ветвью оператора и правой ветвью. Парсер перебирает лексемы из входного потока и сопоставляет их с возможными состояниями и лексемами в грамматике для построения CST.

Все символические имена в CST определяются в Grammar ▶ Grammar:

```

arith_expr: term (( '+' | '-' ) term)*
term: factor (( '*' | '@' | '/' | '%' | '//' ) factor)*
factor: ('+' | '-' | '~') factor | power
power: atom_expr [ '**' factor]
atom_expr: [ AWAIT ] atom trailer*
atom: ('(' [yield_expr|testlist_comp] ')'
      '[' [testlist_comp] ']'
      '{' [dictorsetmaker] '}'
      NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
  
```

Лексемы определяются в Grammar ▶ Tokens:

```

ENDMARKER
NAME
NUMBER
STRING
NEWLINE
INDENT
DEDENT
  
```

```
LPAR      '('
RPAR      ')'
LSQB      '['
RSQB      ']'
COLON     ':'
COMMA     ','
SEMI      ';'
PLUS      '+'
MINUS    '-'
STAR      '*'
...
...
```

Лексема `NAME` представляет имя переменной, функции, класса или модуля. Синтаксис Python не допускает, чтобы `NAME` было одним из зарезервированных ключевых слов (например, `await` или `async`), а также числовым или другим литеральным типом. Например, если вы попытаетесь определить функцию с именем `1`, Python выдаст ошибку синтаксиса (`SyntaxError`):

```
>>> def 1():
File "<stdin>", line 1
  def 1():
^
SyntaxError: invalid syntax
```

`NUMBER` — особая разновидность лексем для представления одного из числовых значений Python. В Python существует специальная грамматика для чисел:

- **Восьмеричные значения**, например `0o20`
- **Шестнадцатеричные значения**, например `0x10`
- **Двоичные значения**, например `0b10000`
- **Комплексные числа**, например `10j`
- **Числа с плавающей точкой**, например `1.01`
- **Подчеркивания в качестве разделителя разрядов**, например `1_000_000`

Для просмотра скомпилированных символических имен и лексем в Python можно воспользоваться модулями `symbol` и `token`:

```
$ ./python
>>> import symbol
>>> dir(symbol)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
 __name__, '__package__', '__spec__', '_abc_registry']
```

```
'__name__', '__package__', '__spec__', '__main', '__name', '__value',
'and_expr', 'and_test', 'annassign', 'arglist', 'argument',
'arith_expr', 'assert_stmt', 'async_funcdef', 'async_stmt',
'atom', 'atom_expr',
...
>>> import token
>>> dir(token)
['AMPER', 'AMPEREQUAL', 'AT', 'ATEQUAL', 'CIRCUMFLEX',
'CIRCUMFLEXEQUAL', 'COLON', 'COMMA', 'COMMENT', 'DEDENT', 'DOT',
'DOUBLESLASH', 'DOUBLESLASHEQUAL', 'DOUBLESTAR', 'DOUBLESTAREQUAL',
...
...
```

ПАРСЕР/ТОКЕНИЗАТОР CPYTHON

В языках программирования применяются разные реализации лексического анализатора. Некоторые языки используют лексический анализатор/генератор как дополнение к парсеру. В CPython существует модуль парсера/токенизатора, написанный на С.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к парсеру/выделителю лексем.

ФАЙЛ	НАЗНАЧЕНИЕ
Python ▶ pythonrun.c	Запускает парсер и компилятор для входных данных
Parser ▶ parsetok.c	Реализация парсера и токенизатора
Parser ▶ tokenizer.c	Реализация токенизатора
Parser ▶ tokenizer.h	Заголовочный файл для реализации токенизатора, описывающий модели данных (например, состояние лексем)
Include ▶ token.h	Объявление типов лексем, генерируемых Tools ▶ scripts ▶ generate_token.py
Include ▶ node.h	Интерфейс разбора узлов дерева и макросы для токенизатора

Ввод данных в парсер из файла

Точка входа парсера/токенизатора, `PyParser_ASTFromFileObject()`, получает дескриптор файла, флаги компилятора и экземпляр `PyArena` и преобразует объект файла в модуль.

Процедура состоит из двух шагов:

1. Преобразование в CST с использованием `PyParser_ParseFileObject()`.
2. Преобразование в AST или модуль функцией AST — `PyAST_FromNodeObject()`.

Функция `PyParser_ParseFileObject()` выполняет две важные задачи:

1. Создание экземпляра состояния токенизатора `tok_state` при помощи `PyTokenizer_FromFile()`.
2. Преобразование лексем в CST (список узлов) функцией `parsetok()`.

Логика парсера/токенизатора

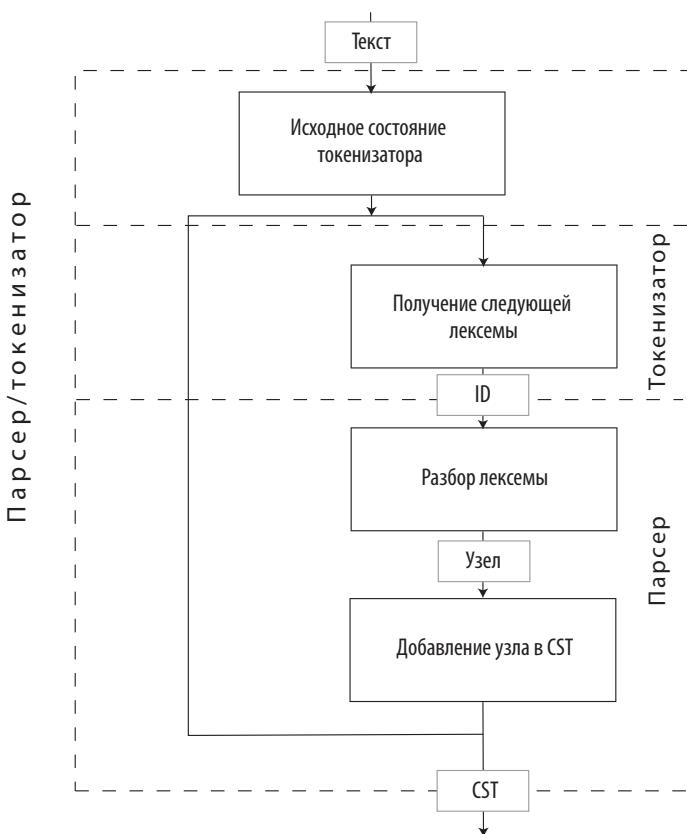
Парсер/токенизатор получает текстовый ввод и запускает токенизатор и парсер в цикле, пока курсор не достигнет конца текста (или не произойдет ошибка синтаксиса).

Перед этим парсер/токенизатор создает `tok_state` — временную структуру данных для хранения всех состояний, используемых токенизатором. Состояние содержит такую информацию, как текущая позиция курсора и строка.

Парсер/токенизатор вызывает `tok_get()` для получения следующей лексемы. Затем он передает полученный идентификатор лексемы парсеру, который использует ДКА парсера/генератора для создания узла конкретного синтаксического дерева.

`tok_get()` — одна из самых сложных функций во всей кодовой базе CPython. Она содержит свыше 640 строк, и в ней отражены десятилетия унаследованных граничных случаев, новые языковые возможности и синтаксис.

Процесс вызова токенизатора и парсера в цикле может быть наглядно представлен следующим образом:



Корневой узел CST, возвращаемый `PyParser_ParseFileObject()`, критичен для следующей стадии — преобразования CST в абстрактное синтаксическое дерево (AST).

Тип узла определяется в файле `Include > node.h`:

```

typedef struct _node {
    short          n_type;
    char           *n_str;
    int            n_lineno;
    int            n_col_offset;
    int            n_nchildren;
    struct _node  *n_child;
    int            n_end_lineno;
    int            n_end_col_offset;
} node;

```

Так как CST является деревом синтаксиса, идентификаторов лексем и символьических имен, компилятору трудно принимать быстрые решения, основанные на коде Python.

Прежде чем переходить к AST, отметим, что существует возможность просмотра вывода стадии разбора. В CPython имеется модуль стандартной библиотеки `parser`, который предоставляет функции С с Python API.

Вывод будет числовым, в нем используются номера лексем и символовических имен, сгенерированных на стадии `make regen-grammar` и хранящихся в `Include ▶ token.h`:

Чтобы вам было проще понять вывод, можно взять все числа из модулей `symbol` и `token`, поместить их в словарь и рекурсивно заменить значения в выводе `parser.st2list()` именами лексем:

[cpython-book-samples](#) ▶ 21 ▶ `lex.py`

```
import symbol
import token
import parser

def lex(expression):
    symbols = {v: k for k, v in symbol._dict_.items()
               if isinstance(v, int)}
```

```
tokens = {v: k for k, v in token.__dict__.items()
          if isinstance(v, int)}
lexicon = {**symbols, **tokens}
st = parser.expr(expression)
st_list = parser.st2list(st)

def replace(l: list):
    r = []
    for i in l:
        if isinstance(i, list):
            r.append(replace(i))
        else:
            if i in lexicon:
                r.append(lexicon[i])
            else:
                r.append(i)
    return r

return replace(st_list)
```

Выполните `lex()` с простым выражением (например, `a + 1`), чтобы увидеть, как оно представляется в виде дерева парсера:

В выводе встречаются символические имена в нижнем регистре (например, 'arith_expr') и лексемы в верхнем регистре (например, 'NUMBER').

АБСТРАКТНЫЕ СИНТАКСИЧЕСКИЕ ДЕРЕВЬЯ

На следующем этапе интерпретатор CPython преобразует дерево CST, сгенерированное парсером, в нечто более логичное и подходящее для выполнения.

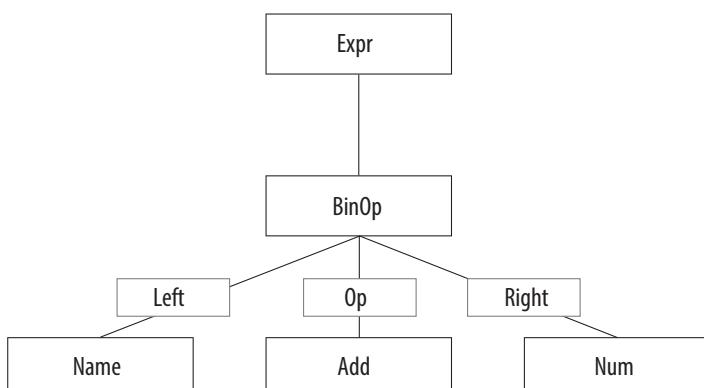
Конкретные синтаксические деревья являются буквальным представлением текста из кодового файла. Базовая грамматическая структура Python была интерпретирована, но CST не может использоваться для представления функций, областей видимости, циклов или других базовых средств языка Python.

Прежде чем код будет скомпилирован, CST необходимо преобразовать в структуру более высокого уровня, представляющую фактические конструкции Python. Эта структура является представлением CST и называется абстрактным синтаксическим деревом (AST).

Например, бинарная операция в AST называется `BinOp` и определяется как разновидность выражения. Она имеет три компонента:

1. `left`: левая часть операции.
2. `op`: оператор (например, +, - или *).
3. `right`: правая часть выражения.

Представление дерева AST для выражения `a + 1` может выглядеть так:



Деревья AST строятся парсером CPython, но их также можно генерировать из кода Python с использованием модуля `ast` в стандартной библиотеке.

Прежде чем погружаться в реализацию AST, будет полезно понять, как выглядит AST для несложного фрагмента Python-кода.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к абстрактным синтаксическим деревьям.

ФАЙЛ	НАЗНАЧЕНИЕ
Include ➤ Python-ast.h	Объявление типов узлов AST, генерируемых Parser ➤ asdl_c.py
Parser ➤ Python.asdl	Список типов узлов и свойств AST на предметно-ориентированном языке ASDL 5
Python ➤ ast.c	Реализация AST

Использование Instaviz для просмотра абстрактных синтаксических деревьев

Instaviz — пакет Python, написанный специально для этой книги. Он выводит AST и скомпилированный код в веб-интерфейсе.

Чтобы установить Instaviz, загрузите пакет `instaviz` с помощью `pip`:

```
$ pip install instaviz
```

Затем откройте REPL, выполнив команду `python` в командной строке без аргументов.

Функция `instaviz.show()` принимает единственный аргумент — объект кода. Такие объекты будут рассматриваться в следующей главе. В данном примере определяется функция и используется ее имя как значение аргумента:

```
$ python
>>> import instaviz
>>> def example():
    a = 1
```

```
b = a + 1
return b

>>> instaviz.show(example)
```

В командной строке появляется уведомление о том, что веб-сервер был запущен на порте 8080. Если порт используется для других целей, его можно изменить вызовом `instaviz.show(example, port=9090)` или с другим номером порта.

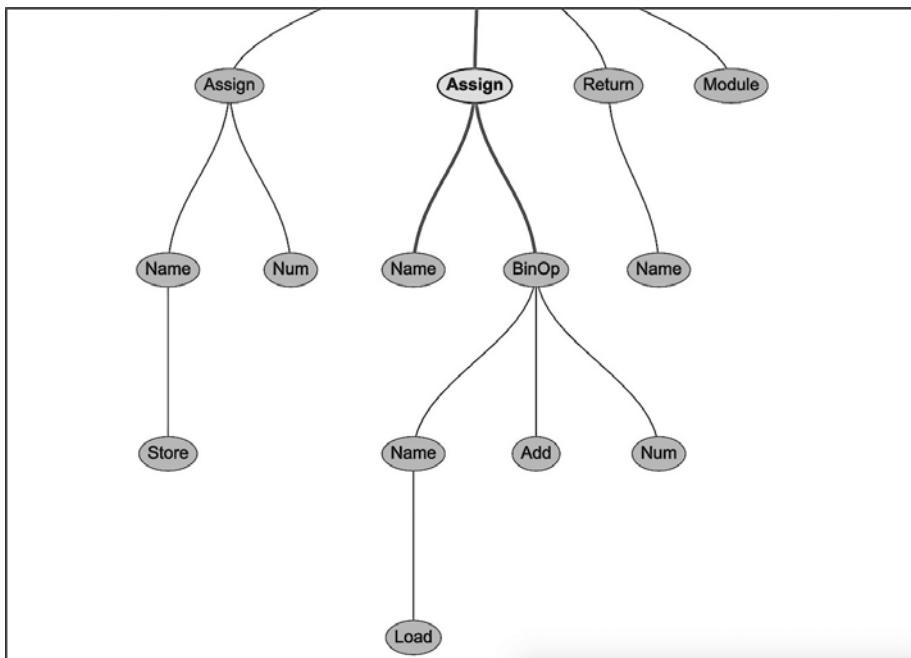
В браузере выводятся подробные сведения о функции:

Code Object Properties	
Field	Value
<code>co_argcount</code>	0
<code>co_cellvars</code>	0
<code>co_code</code>	64017d007c00640117007d017c015300
<code>co_consts</code>	(None, 1)
<code>co_filename</code>	test.py
<code>co_firstlineno</code>	4
<code>co_freevars</code>	0
<code>co_kwonlyargcount</code>	0
<code>co_lnotab</code>	b'\x00\x01\x04\x01\x08\x01'
<code>co_name</code>	foo
<code>co_names</code>	0
<code>co_nlocals</code>	2
<code>co_stacksize</code>	2
<code>co_varnames</code>	('a', 'b')
<code>4 def foo():</code>	
<code>5 a = 1</code>	
<code>6 b = a + 1</code>	
<code>7 return b</code>	
Graph direction: Up-Down Down-Up Left-Right Right-Left	

Граф внизу слева изображает функцию, объявленную в REPL и представленную в виде абстрактного синтаксического дерева. Каждый узел в дереве представляется типом AST. Эти типы находятся в модуле `ast` и наследуются от `_ast.AST`.

Некоторые узлы обладают свойствами, связывающими их с дочерними узлами, в отличие от деревьев CST, которые содержат обобщенное свойство дочернего узла.

Например, если кликнуть по узлу `Assign` в центре, он связывается со строкой `b = a + 1:`

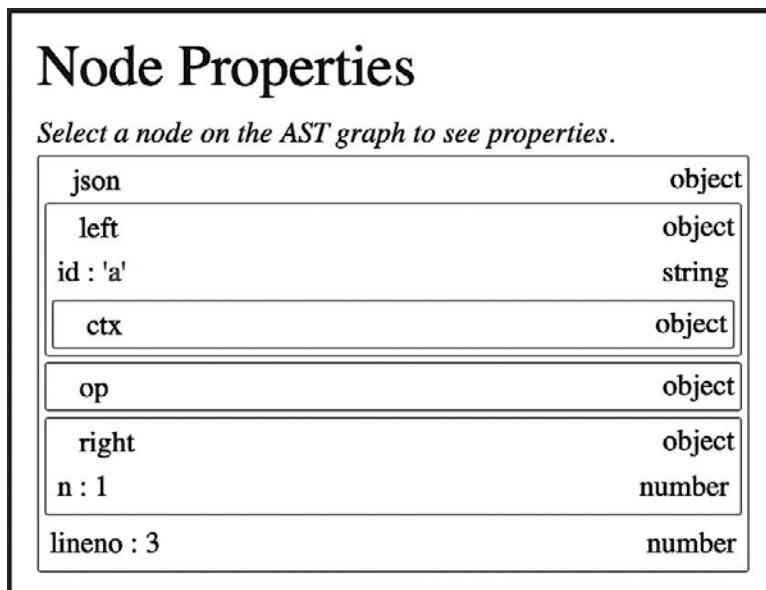


Узел `Assign` содержит два свойства:

1. `targets` — список присваиваемых имен. Это список, потому что синтаксис распаковки позволяет присвоить значения сразу нескольким переменным одним выражением.
2. `value` — присваиваемое значение, которым в данном случае является команда `BinOp` — `a + 1`.

Если кликнуть по команде `BinOp`, выводятся соответствующие свойства:

- `left`: узел слева от оператора.
- `op`: оператор, в данном случае — узел `Add` (+) для сложения.
- `right`: узел справа от оператора.



Компиляция AST

Компиляция AST в C — дело нетривиальное. Модуль Python ► `ast.c` содержит более 5000 строк кода.

Есть несколько точек входа, образующих часть общедоступного API для работы с AST. AST API получает узел дерева (CST), имя файла, флаги компилятора и область для хранения данных в памяти.

Результат имеет тип `mod_ty`, который представляет модуль Python, определенный в `Include ▶ Python-ast.h`.

`mod_ty` — контейнер для одного из четырех типов модулей в Python:

1. `Module`
2. `Interactive`
3. `Expression`
4. `FunctionType`

Все типы модулей перечислены в файле `Parser ▶ Python.asdl`. Все типы модулей, типы команд, типы выражений, операторы и включения (`comprehensions`) определяются в этом файле.

Названия типов в Parser ▶ Python.asdl соотносятся с классами, генерируемыми AST, и с именами для этих же классов, которые приводятся в модуле стандартной библиотеки `ast`:

```
-- ASDL's 4 builtin types are:  
-- identifier, int, string, constant  
  
module Python  
{  
    mod = Module(stmt* body, type_ignore *type_ignores)  
        | Interactive(stmt* body)  
        | Expression(expr body)  
        | FunctionType(expr* argtypes, expr returns)
```

Модуль `ast` импортирует Include ▶ Python-ast.h — файл, созданный автоматически из Parser ▶ Python.asdl при повторном генерировании грамматики. Параметры и имена в Include ▶ Python-ast.h напрямую соответствуют заданным в Parser ▶ Python.asdl.

Тип `mod_ty` генерируется в Include ▶ Python-ast.h из определения `Module` в Parser ▶ Python.asdl:

```
enum _mod_kind {Module_kind=1, Interactive_kind=2, Expression_kind=3,  
                FunctionType_kind=4};  
struct _mod {  
    enum _mod_kind kind;  
    union {  
        struct {  
            asdl_seq *body;  
            asdl_seq *type_ignores;  
        } Module;  
  
        struct {  
            asdl_seq *body;  
        } Interactive;  
  
        struct {  
            expr_ty body;  
        } Expression;  
  
        struct {  
            asdl_seq *argtypes;  
            expr_ty returns;  
        } FunctionType;  
    } v;  
};
```

Заголовочный файл и структуры С позволяют программе Python ► `ast.c` быстро генерировать структуры с указателями на соответствующие данные.

Точка входа AST, `PyAST_FromNodeObject()`, по сути представляет собой оператор `switch` для результата `TYPE(n)`. `TYPE()` — макрос, используемый AST для определения типа узлов в конкретном синтаксическом дереве.

Результатом `TYPE()` является тип лексемы или символического имени.

Начиная с корневого узла, тип может относиться только к одному из определенных типов модулей — `Module`, `Interactive`, `Expression` или `FunctionType`:

- Для `file_input` должен использоваться тип `Module`.
- Для `eval_input` (например, для ввода из REPL) должен использоваться тип `Expression`.

Для каждого типа оператора в Python ► `ast.c` есть соответствующая функция C `ast_for_xxx`, которая просматривает узлы CST для заполнения свойств этой команды.

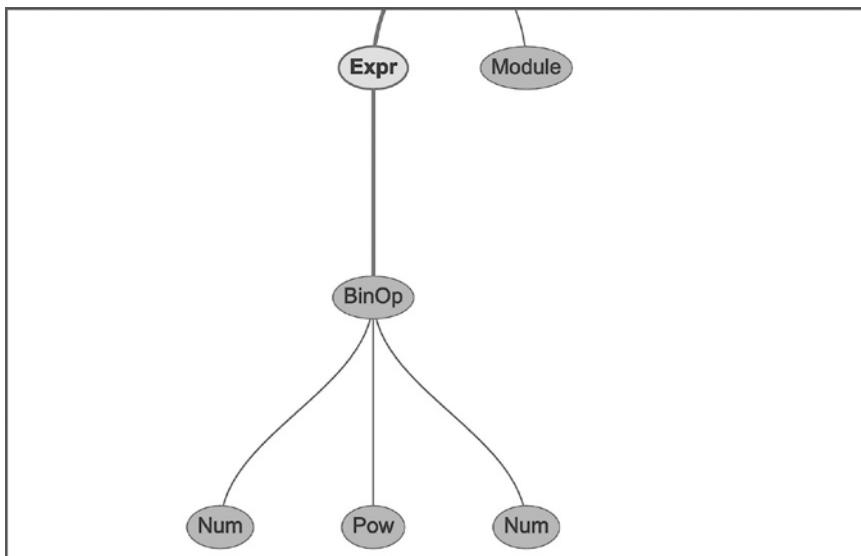
Одним из простых примеров служит выражение возведения в степень — например, `2 ** 4` (то есть 2 в степени 4). `ast_for_power()` возвращает `BinOp` с оператором `Pow` (Power), левой частью `e` (2) и правой частью `f` (4):

Python ► `ast.c`, строка 2717

```
static expr_ty
ast_for_power(struct compiling *c, const node *n)
{
    /* power: atom trailer* ('**' factor)* */
    expr_ty e;
    REQ(n, power);
    e = ast_for_atom_expr(c, CHILD(n, 0));
    if (!e)
        return NULL;
    if (NCH(n) == 1)
        return e;
    if (TYPE(CHILD(n, NCH(n) - 1)) == factor) {
        expr_ty f = ast_for_expr(c, CHILD(n, NCH(n) - 1));
        if (!f)
            return NULL;
        e = BinOp(e, Pow, f, LINENO(n), n->n_col_offset,
                  n->n_end_lineno, n->n_end_col_offset, c->c_arena);
    }
    return e;
}
```

Чтобы увидеть результат, отправьте короткую функцию модулю `instaviz`:

```
>>> def foo():
...     2**4
...>>> import instaviz
...>>> instaviz.show(foo)
```



Соответствующие свойства отображаются в пользовательском интерфейсе:

Node Properties

Select a node on the AST graph to see properties.

json	object
value	object
left	object
n : 2	number
op	object
right	object
n : 4	number
lineno : 2	number

Короче говоря, у каждого типа оператора и выражения имеется соответствующая функция `ast_for_*`() для его создания. Аргументы определяются

в Parser ► `Python.asdl`, а доступ к ним предоставляется через модуль `ast` в стандартной библиотеке.

Если у выражения или команды есть дочерние узлы, то соответствующая дочерняя функция `ast_for_*`() будет вызываться в порядке обхода в глубину.

ВАЖНЫЕ ТЕРМИНЫ

Ниже перечислены некоторые ключевые термины, встретившиеся в этой главе:

- **Абстрактное синтаксическое дерево (AST)**: представление грамматики и команд Python в виде контекстного дерева.
- **Конкретное синтаксическое дерево (CST)**: представление лексем и символьических имен в виде неконтекстного дерева.
- **Дерево разбора (parse tree)**: другой термин для обозначения конкретного синтаксического дерева.
- **Лексема (token)**: тип символьического имени (например, +).
- **Токенизация (tokenization)**: процесс преобразования текста в лексемы.
- **Парсинг (parsing)**: процесс преобразования текста в CST или AST.

ПРИМЕР: ДОБАВЛЕНИЕ ОПЕРАТОРА «ПОЧТИ РАВНО»

Чтобы объединить все сказанное, мы добавим в язык Python новый элемент синтаксиса и перекомпилируем CPython, чтобы он этот синтаксис поддерживал.

Оператор сравнения сравнивает два и более значений:

```
>>> a = 1
>>> b = 2
>>> a == b
False
```

Операторы, используемые в выражениях сравнения, называются **операторами сравнения**. Вероятно, вам знакомы следующие операторы сравнения:

- **Меньше:** <
- **Больше:** >
- **Равно:** ==
- **Не равно:** !=

СМ. ТАКЖЕ

Расширенные сравнения в модели данных были предложены для Python 2.1 в PEP 207. PEP содержит контекст, историю и обоснование для реализации методов сравнения в нестандартных типах Python.

Добавим еще один оператор сравнения «почти равно», который будет представляться символами ~=. Оператор будет обладать следующим поведением:

- Число с плавающей точкой при сравнении с целым числом сначала будет приводиться к целочисленному типу.
- При сравнении двух целых чисел будут использоваться обычные операторы проверки равенства.

Новый оператор должен возвращать следующий результат в REPL:

```
>>> 1 ~= 1
True
>>> 1 ~= 1.0
True
>>> 1 ~= 1.01
True
>>> 1 ~= 1.9
False
```

Чтобы добавить новый оператор, необходимо сначала обновить грамматику CPython. В файле `Grammar ▶ python.gram` операторы сравнения определяются символическим именем `comp_op`:

```
comparison[expr_ty]:
    | a=bitwise_or b=compare_op_bitwise_or_pair+ ...
    | bitwise_or
compare_op_bitwise_or_pair[CmpopExprPair*]:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
    | notin_bitwise_or
    | in_bitwise_or
    | isnot_bitwise_or
    | is_bitwise_or
eq_bitwise_or[CmpopExprPair*]: '==' a=bitwise_or ...
noteq_bitwise_or[CmpopExprPair*]:
    | (tok!=! {_PyPegen_check_barry_as_flufl(p) ? NULL : tok}) ...
lte_bitwise_or[CmpopExprPair*]: '<=' a=bitwise_or ...
lt_bitwise_or[CmpopExprPair*]: '<' a=bitwise_or ...
gte_bitwise_or[CmpopExprPair*]: '>=' a=bitwise_or ...
gt_bitwise_or[CmpopExprPair*]: '>' a=bitwise_or ...
notin_bitwise_or[CmpopExprPair*]: 'not' 'in' a=bitwise_or ...
in_bitwise_or[CmpopExprPair*]: 'in' a=bitwise_or ...
isnot_bitwise_or[CmpopExprPair*]: 'is' 'not' a=bitwise_or ...
is_bitwise_or[CmpopExprPair*]: 'is' a=bitwise_or ...
```

Измените выражение `compare_op_bitwise_or_pair`, чтобы оно также допускало новую пару `ale_bitwise_or`:

```
compare_op_bitwise_or_pair[CmpopExprPair*]:
    | eq_bitwise_or
...
    | ale_bitwise_or
```

Определите новое выражение `ale_bitwise_or` под существующим выражением `is_bitwise_or`:

```
...
is_bitwise_or[CmpopExprPair*]: 'is' a=bitwise_or ...
ale_bitwise_or[CmpopExprPair*]: '~=' a=bitwise_or
{ _PyPegen_cmpop_expr_pair(p, Ale, a) }
```

Новый тип определяет именованное выражение `ale_bitwise_or`, которое содержит терминал `'~='`.

Вызов функции `_PyPegen_cmpror_expr_pair(p, A1E, a)` является выражением для получения узла `cmpror` из AST. Название типа `A1E` происходит от слов «almost equal» («почти равно»).

Затем добавьте лексему в `Grammar ▶ Tokens`:

```
ATEQUAL          '@='
RARROW           '->'
ELLIPSIS         '...'
COLONEQUAL      ':='
# Добавьте эту строку
ALMOSTEQUAL     '~='
```

Чтобы обновить грамматику и лексемы в C, необходимо заново сгенерировать заголовки.

В macOS и Linux используется следующая команда:

```
$ make regen-token regen-pegen
```

В Windows выполните следующую команду из каталога `PCBuild`:

```
> build.bat --regen
```

Эти действия автоматически обновляют токенизатор. Например, откройте файл `Parser/token.c` и посмотрите, как изменилась секция `case` в функции `PyToken_TwoChars()`:

```
case '~':
    switch (c2) {
        case '=': return ALMOSTEQUAL;
    }
    break;
}
```

Если перекомпилировать CPython на этом этапе и открыть REPL, вы увидите, что токенизатор успешно распознает лексему, но AST не знает, как обработать ее:

```
$ ./python
>>> 1 ~= 2
SystemError: invalid comp_op: ~=
```

Иключение поднимается функцией `ast_for_comp_op()` из файла `Python ▶ ast.c`, потому что `ALMOSTEQUAL` не распознается как допустимый оператор для команды сравнения.

`Compare` — тип выражения, определенный в `Parser ▶ Python.asdl`. Он содержит свойства для левого выражения, список операторов `ops` и список сравниваемых выражений `comparators`:

```
| Compare(expr left, cmpop* ops, expr* comparators)
```

Внутри определения `Compare` находится ссылка на перечисление `cmpop`:

```
cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn
```

Это список возможных листовых узлов AST, которые могут использоваться как операторы сравнения. Наш оператор в списке отсутствует, его необходимо добавить. Включите в этот список новый тип `A1E`:

```
cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn | A1E
```

Затем снова сгенерируйте AST, чтобы обновить заголовочные файлы C AST:

```
$ make regen-ast
```

Команда обновляет список операторов сравнения (`_cmpop`) в файле `Include/Python-ast.h` и включает в него вариант `A1E`:

```
typedef enum _cmpop { Eq=1, NotEq=2, Lt=3, LtE=4, Gt=5, GtE=6, Is=7,
                      IsNot=8, In=9, NotIn=10, A1E=11 } cmpop_ty;
```

AST не знает, что лексема `ALMOSTEQUAL` эквивалентна оператору сравнения `A1E`. А значит, необходимо обновить код C для AST.

Перейдите к функции `ast_for_comp_op()` в `Python ▶ ast.c`. Найдите команду `switch` для лексем операторов. Она возвращает одно из списка значений `_cmpop`.

Добавьте две строки для обнаружения лексемы `ALMOSTEQUAL` и возвращения оператора сравнения `A1E`:

Python ▶ ast.c, строка 1222

```
static cmpop_ty
ast_for_comp_op(struct compiling *c, const node *n)
{
    /* comp_op: '<' | '>' | '==' | '>=' | '<=' | '!='
     | 'in' | 'not' | 'in' | 'is'
     | 'is' | 'not'
```

```
*/  
REQ(n, comp_op);  
if (NCH(n) == 1) {  
    n = CHILD(n, 0);  
    switch (TYPE(n)) {  
        case LESS:  
            return Lt;  
        case GREATER:  
            return Gt;  
        case ALMOSTEQUAL: // Добавьте эту строку для обнаружения лексемы  
            return A1E; // Эта строка возвращает узел AST
```

На этой стадии токенизатор и AST могут разобрать код, но компилятор не будет знать, как обработать оператор. Чтобы протестировать представление AST, вызовите `ast.parse()` и проанализируйте первый оператор в выражении:

```
>>> import ast  
>>> m = ast.parse('1 ~= 2')  
>>> m.body[0].value.ops[0]  
<_ast.A1E object at 0x10a8d7ee0>
```

Это экземпляр нашего типа оператора сравнения `A1E`, и AST правильно разбирает код.

В следующей главе вы узнаете, как работает компилятор CPython, и мы вернемся к оператору «почти равно», чтобы определить его поведение.

ВЫВОДЫ

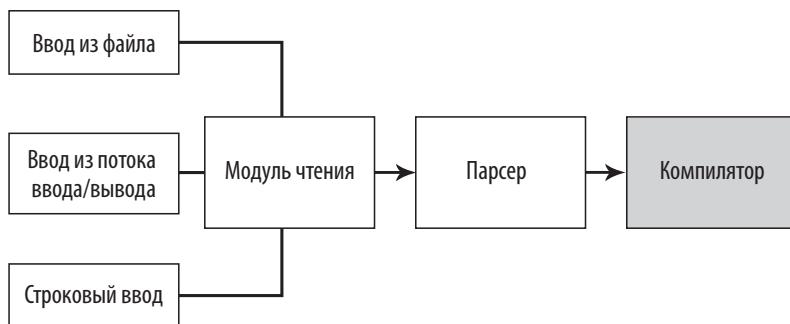
Гибкость CPython и низкоуровневый API выполнения делают его идеальным кандидатом на роль встроенного скриптового движка. CPython используется во многих UI-приложениях, включая компьютерные игры, 3D-графику и системы автоматизации.

Процесс работы интерпретатора гибок и эффективен. Теперь вы получили представление о том, как он функционирует, и мы можем перейти к изучению компилятора.

Компилятор

После завершения задачи парсинга у интерпретатора имеется дерево AST с операциями, функциями, классами и пространствами имен Python-кода.

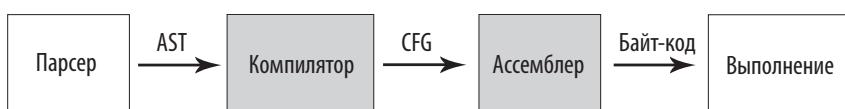
Задача компилятора – преобразование AST в инструкции, понятные для процессора:



В задаче компиляции участвуют два компонента:

1. **Компилятор**: обходит AST и создает **граф потока управления** (CFG, Control Flow Graph), представляющий логическую последовательность выполнения.
2. **Ассемблер**: преобразует узлы CFG в последовательность выполняемых команд, называемую **байт-кодом**.

Наглядное представление процесса компиляции:



ВАЖНО

Важно помнить, что единицей компиляции для CPython является **модуль**. Шаги компиляции и процесс, описанный в этой главе, выполняются однократно для каждого модуля в вашем проекте.

В этой главе мы сосредоточимся на компиляции модуля AST в объект кода.

Функция `PyAST_CompileObject()` — главная точка входа компилятора CPython. Она получает в качестве основного аргумента модуль Python AST; также передается имя файла, глобальные и локальные переменные и PyArena — все эти данные должны быть созданы ранее в процессе работы интерпретатора.

ПРИМЕЧАНИЕ

Теперь вы начинаете углубляться во внутреннее устройство компилятора CPython, за которым лежат десятилетия разработки и развития computer science. Не пугайтесь его размеров и сложности. Если разбить компилятор на логические шаги, понять его будет не так сложно.

ИСХОДНЫЕ ФАЙЛЫ

Ниже перечислены исходные файлы, относящиеся к компилятору:

ФАЙЛ	НАЗНАЧЕНИЕ
Python ▶ <code>compile.c</code>	Реализация компилятора
Include ▶ <code>compile.h</code>	API и определения типов компилятора

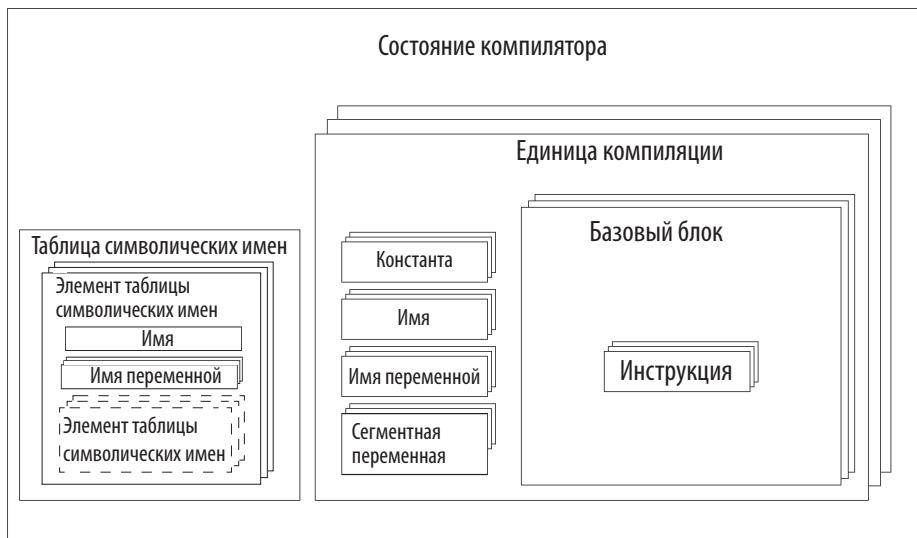
ВАЖНЫЕ ТЕРМИНЫ

Ниже перечислены некоторые ключевые термины, которые вам могут быть пока незнакомы:

- **Состояние компилятора** реализуется как тип контейнера, содержащий одну таблицу символических имен.

- Таблица символьических имен содержит **имена переменных**, также в ней могут дополнительно находиться дочерние таблицы символьических имен.
- Состояние компилятора включает в себя несколько **единиц компиляции**.
- Каждая единица компиляции может содержать множество имен, имен переменных, констант и сегментных переменных.
- Единица компиляции содержит несколько **базовых блоков**.
- Базовые блоки содержат множество **инструкций байт-кода**.

Контейнер состояния компилятора и его компоненты можно наглядно представить в следующем виде:



СОЗДАНИЕ ЭКЗЕМПЛЯРА КОМПИЛЯТОРА

Перед запуском компилятора создается глобальное состояние компилятора. Структура состояния компилятора (тип `compiler`) содержит свойства, используемые компилятором: флаги компилятора, стек и `PyArena`. Также она содержит ссылки на другие структуры данных, такие как таблица символьических имен.

Поля состояния компилятора:

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
c_arena	PyArena *	Указатель на арену выделения памяти
c_const_cache	PyObject * (dict)	Словарь Python (dict) со всеми константами, включая кортеж names
c_do_not_emit_bytecode	int	Флаг отключения компиляции байт-кода
c_filename	PyObject * (str)	Имя компилируемого файла
c_flags	PyCompilerFlags *	Унаследованные флаги компилятора (см. раздел «Флаги компилятора»)
c_future	PyFutureFeatures *	Указатель на __future__ модуля
c_interactive	int	Флаг интерактивного режима
c_nestlevel	int	Текущий уровень вложенности
c_optimize	int	Уровень оптимизации
c_st	symtable *	Таблица символических имен компилятора
c_stack	PyObject * (list)	Список Python (list), содержащий указатели compiler_unit
u	compiler_unit*	Состояние компилятора для текущего блока

Экземпляр состояния компилятора создается внутри `PyAST_CompileObject()`:

- Если модуль не содержит строку документации (`__doc__`), в этой точке создается пустая строка, как и в свойстве `__annotations__`.
- `PyAST_CompileObject()` задает переданное значение как имя файла состояния компилятора, которое используется для трассировки стека и обработки исключений.

- В качестве арены выделения памяти компилятора присваивается та, которая использовалась интерпретатором. За дополнительной информацией о распределителях памяти обращайтесь к разделу «Специальные распределители памяти».
- Все флаги будущей функциональности задаются перед компиляцией кода.

ФЛАГИ БУДУЩЕЙ ФУНКЦИОНАЛЬНОСТИ И ФЛАГИ КОМПИЛЯТОРА

Для включения функциональности внутри компилятора используются флаги двух типов: **флаги будущей функциональности** и **флаги компилятора**. Эти флаги могут задаваться в двух местах:

1. В конфигурации состояния, содержащей переменные среды и флаги командной строки.
2. В исходном коде модуля с использованием команд `__future__`.

За дополнительной информацией о конфигурации состояния обращайтесь к разделу «Конфигурация состояния» главы «Конфигурация и ввод».

Флаги будущей функциональности

Необходимость в флагах будущей функциональности возникает из-за синтаксиса или возможностей конкретного модуля. Например, в Python 3.7 появилась отложенная обработка аннотаций типов с использованием флага будущей функциональности `annotations`:

```
from __future__ import annotations
```

В коде, следующем за этой командой, могут использоваться неразрешенные аннотации типов, поэтому команда `__future__` необходима. В противном случае модуль не будет импортироваться.

Флаги будущей функциональности в 3.9

В версии 3.9 все флаги будущей функциональности, кроме двух (`annotations` и `barry_as_FLUFL`), являются обязательными и активизируются автоматически:

ФЛАГ	НАЗНАЧЕНИЕ
absolute_import	Включение абсолютного импортирования (PEP 328)
annotations	Отложенная обработка аннотаций типов (PEP 563)
barry_as_FLUFL	Пасхальное яйцо (PEP 401)
division	Использование оператора истинного деления (PEP 238)
generator_stop	Возможность использования StopIteration внутри генераторов (PEP 479)
generators	Включение простых генераторов (PEP 255)
nested_scopes	Добавление статических вложенных областей видимости (PEP 227)
print_function	Использование print как функции (3105)
unicode_literals	Хранение литералов str в Юникоде вместо байтов (PEP 3112)
with_statement	Включение оператора with (PEP 343)

ПРИМЕЧАНИЕ

Большинство флагов `__future__` использовалось для обеспечения портируемости между Python 2 и 3. Вероятно, с приближением выхода Python 4.0 появится больше флагов будущей функциональности.

Флаги компилятора

Флаги компилятора привязаны к установленному окружению, что позволяет им изменять способ выполнения кода или работы компилятора, но они не должны связываться с исходным кодом, как команды `__future__`.

Один из примеров флагов компилятора — флаг `-O`¹ для оптимизации команд `assert`. Этот флаг отключает все команды `assert`, которые могли быть вставлены в код в целях отладки². Режим также можно включить установкой переменной среды `PYTHONOPTIMIZE=1`.

¹ <https://docs.python.org/3/using/cmdline.html#cmdoption-O>.

² <https://realpython.com/python-debugging-pdb/>.

ТАБЛИЦЫ СИМВОЛИЧЕСКИХ ИМЕН

Перед компиляцией кода создается **таблица символических имен**, для чего используется функция API `PySymtable_BuildObject()`.

Таблица символических имен предоставляет список пространств имен, глобальных и локальных имен, которые должны использоваться компилятором для ссылок и разрешения областей видимости.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к таблице символических имен:

ФАЙЛ	НАЗНАЧЕНИЕ
Python ► symtable.c	Реализация таблицы символических имен
Include ► symtable.h	Определение API таблицы символических имен и типов
Lib ► symtable.py	Модуль стандартной библиотеки symtable

Структура данных таблицы символических имен

Структура `symtable` должна быть единственным экземпляром `symtable` для компилятора, из-за чего пространства имен начинают играть особенно важную роль.

Например, если вы создали метод с именем `resolve_names()` в одном классе, а потом объявили еще один метод с тем же именем в другом классе, необходимо точно указать, какой из методов будет вызываться внутри модуля.

Структура `symtable` служит этой цели, а также гарантирует, что переменные, объявленные в узкой области видимости, не будут автоматически становиться глобальными.

Структура таблицы символических имен `symtable` содержит следующие поля.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>recursion_depth</code>	<code>int</code>	Текущая глубина рекурсии
<code>recursion_limit</code>	<code>int</code>	Предельная глубина рекурсии до выдачи ошибки <code>RecursionError</code>

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
st_blocks	PyObject * (dict)	Соответствие адресов узлов AST и элементов таблицы символьических имен
st_cur	_symtable_entry	Текущий элемент таблицы символьических имен
st_filename	PyObject * (str)	Имя компилируемого файла
st_future	PyFutureFeatures	Возможности будущей функциональности модуля, влияющие на таблицу символьических имен
st_global	PyObject * (dict)	Ссылки на символические имена в st_top
st_nblocks	int	Количество использованных блоков
st_private	PyObject * (str)	Имя текущего класса (не обязательно)
st_stack	PyObject * (list)	Стек с информацией о пространствах имен
st_top	_symtable_entry	Элемент таблицы символьических имен для модуля

Использование модуля стандартной библиотеки `symtable`

Доступ к некоторым функциям С API таблицы символьических имен предоставляется в Python через модуль `symtable` стандартной библиотеки.

При помощи другого модуля `tabulate` (доступного в PyPI) можно создать скрипт для вывода таблицы символьических имен.

Таблицы символьических имен могут быть вложенными. Если модуль содержит функцию или класс, они тоже будут иметь таблицу символьических имен.

Создайте скрипт `symviz.py` с рекурсивной функцией `show()`:

cpython-book-samples ▶ 30 ▶ symviz.py

```
import tabulate
import symtable

code = """
def calc_pow(a, b):
    return a ** b
a = 1
```

```
b = 2
c = calc_pow(a,b)
"""

_st = symtable.symtable(code, "example.py", "exec")

def show(table):
    print("Symtable {0} ({1})".format(table.get_name(),
                                      table.get_type()))
    print(
        tabulate.tabulate(
            [
                (
                    symbol.get_name(),
                    symbol.is_global(),
                    symbol.is_local(),
                    symbol.get_namespaces(),
                )
                for symbol in table.get_symbols()
            ],
            headers=["name", "global", "local", "namespaces"],
            tablefmt="grid",
        )
    )
    if table.has_children():
        [show(child) for child in table.get_children()]
    show(_st)
```

Запустите `symviz.py` в командной строке, чтобы просмотреть таблицу символьических имен для примера:

(venv) → instaviz git:(master) ✘ python symviz.py			
Symtable top (module)			
name	global	local	namespaces
calc_pow	False	True	[<Function SymbolTable for calc_pow in example.py>]
a	False	True	()
b	False	True	()
c	False	True	()

Symtable calc_pow (function)			
name	global	local	namespaces
a	False	True	()
b	False	True	()

Реализация таблицы символьических имен

Реализация таблицы символьических имен находится в файле Python ▶ `symtable.c`, а первичным интерфейсом является функция `PySymtable_BuildObject()`.

По аналогии с компиляцией AST, рассмотренной в предыдущей главе, `PySymtable_BuildObject()` переключается между возможными типами `mod_ty` (`Module`, `Interactive`, `Expression` и `FunctionType`) и обходит все содержащиеся в них команды.

Таблица символьических имен в рекурсивном режиме анализирует узлы и ветви AST (типа `mod_ty`) и добавляет элементы в `symtable`:

Python ▶ `symtable.c`, строка 261

```
struct symtable *
PySymtable_BuildObject(mod_ty mod, PyObject *filename,
                      PyFutureFeatures *future)
{
    struct symtable *st = symtable_new();
    asdl_seq *seq;
    int i;
    PyThreadState *tstate;
    int recursion_limit = Py_GetRecursionLimit();
    ...
    st->st_top = st->st_cur;
    switch (mod->kind) {
        case Module_kind:
            seq = mod->v.Module.body;
            for (i = 0; i < asdl_seq_LEN(seq); i++)
                if (!symtable_visit_stmt(st,
                                         (stmt_ty)asdl_seq_GET(seq, i)))
                    goto error;
            break;
        case Expression_kind:
            ...
        case Interactive_kind:
            ...
        case FunctionType_kind:
            ...
    }
    ...
}
```

`PySymtable_BuildObject()` перебирает все команды модуля и вызывает `symtable_visit_stmt()` — по сути, огромную команду `switch` с условием `case` для каждого типа команд (определенного в Parser ▶ `Python.asdl`).

У каждого типа команд имеется соответствующая функция для разрешения символьических имен. Например, для команды типа определения функции (`FunctionDef_kind`) реализована конкретная логика для следующих действий:

- проверка текущей глубины рекурсии и сравнение ее с предельной глубиной;
- добавление имени функции в таблицу символьических имен, чтобы ее можно было вызывать или передавать как объект функции;
- разрешение не литеральных аргументов по умолчанию по таблице символьических имен;
- разрешение аннотаций типа;
- разрешение декораторов функций.

Наконец, `symtable_enter_block()` посещает блок с содержимым функции. Далее происходит посещение и преобразование аргументов, после чего посещается и обрабатывается тело функции.

ВАЖНО

У вас когда-либо возникал вопрос, почему аргументы по умолчанию в Python изменяются? Все дело в `symtable_visit_stmt()`. Аргумент по умолчанию представляет собой ссылку на переменную из `symtable`.

Такой подход позволяет избежать лишней работы по копированию значений в неизменяемый тип.

Чтобы вы лучше поняли суть происходящего, ниже приведен код C для построения таблицы символьических имен для функции из `symtable_visit_stmt()`.

Python ▶ `symtable.c`, строка 1171

```
static int
symtable_visit_stmt(struct symtable *st, stmt_ty s)
{
    if (++st->recursion_depth > st->recursion_limit) {
        PyErr_SetString(PyExc_RecursionError,
                       "maximum recursion depth exceeded during compilation");
        VISIT_QUIT(st, 0);
    }
    switch (s->kind) {
    case FunctionDef_kind:
        if (!symtable_add_def(st, s->v.FunctionDef.name, DEF_LOCAL))
```

```
    VISIT_QUIT(st, 0);
if (s->v.FunctionDef.args->defaults)
    VISIT_SEQ(st, expr, s->v.FunctionDef.args->defaults);
if (s->v.FunctionDef.args->kw_defaults)
    VISIT_SEQ_WITH_NULL(st, expr,
        s->v.FunctionDef.args->kw_defaults);
if (!symtable_visit_annotations(st, s, s->v.FunctionDef.args,
    s->v.FunctionDef.returns))
    VISIT_QUIT(st, 0);
if (s->v.FunctionDef.decorator_list)
    VISIT_SEQ(st, expr, s->v.FunctionDef.decorator_list);
if (!symtable_enter_block(st, s->v.FunctionDef.name,
    FunctionBlock, (void *)s, s->lineno,
    s->col_offset))
    VISIT_QUIT(st, 0);
VISIT(st, arguments, s->v.FunctionDef.args);
VISIT_SEQ(st, stmt, s->v.FunctionDef.body);
if (!symtable_exit_block(st, s))
    VISIT_QUIT(st, 0);
break;
case ClassDef_kind: {
    ...
}
case Return_kind:
    ...
case Delete_kind:
    ...
case Assign_kind:
    ...
case AnnAssign_kind:
    ...
...
```

После того как таблица символических имен будет создана, она передается компилятору.

ОСНОВНАЯ КОМПИЛЯЦИЯ

Итак, теперь, когда у `PyAST_CompileObject()` имеется состояние компилятора, таблица символических имен и модуль в форме AST, можно начинать фактическую компиляцию.

Основной компилятор решает две задачи:

1. Преобразование состояния, таблицы символических имен и AST в граф потока управления (CFG).

- Предупреждение возникновения исключений на стадии исполнения посредством перехвата логических ошибок и ошибок кода.

Обращение к компилятору из Python

Компилятор можно вызвать из кода Python при помощи встроенной функции `compile()`. Функция возвращает объект кода (`code object`):

```
>>> co = compile("b+1", "test.py", mode="eval")
>>> co
<code object <module> at 0x10f222780, file "test.py", line 1>
```

Как и в случае с `symtable()` API, простое выражение должно использовать режим (`mode`) "eval", а модуль, функция или класс — режим "exec".

Скомпилированный код находится в свойстве `co_code` объекта кода:

```
>>> co.co_code
b'e\x00d\x00\x17\x005\x00'
```

Стандартная библиотека также включает модуль `dis`, дизассемблирующий инструкции байт-кода. Их можно вывести на экран или получить список экземпляров `Instruction`.

ПРИМЕЧАНИЕ

Тип `Instruction` в модуле `dis` является отражением типа `instr` из C API.

Если импортировать модуль `dis` и передать `dis()` свойство `co_code` объекта кода, то функция дизассемблирует его и выведет инструкции в REPL:

```
>>> import dis
>>> dis.dis(co.co_code)
  0 LOAD_NAME 0 (0)
  2 LOAD_CONST 0 (0)
  4 BINARY_ADD
  6 RETURN_VALUE
```

`LOAD_NAME`, `LOAD_CONST`, `BINARY_ADD` и `RETURN_VALUE` — инструкции байт-кода. Термин «байт-код» объясняется тем, что в двоичной форме длина

инструкций составляет 1 байт. Впрочем, начиная с Python 3.6, формат хранения был расширен до машинного слова¹, так что формально это «слово-код», а не «байт-код».

Полный список инструкций байт-кода² доступен для каждой версии Python, причем он изменяется между версиями. Например, в Python 3.7 появились новые инструкции байт-кода, ускоряющие вызов некоторых методов.

В предшествующих главах мы рассмотрели пакет `instaviz`. Его функциональность включает визуализацию типа объекта кода посредством запуска компилятора. Кроме того, `instaviz` выводит операции байт-кода внутри объектов кода.

Снова выполните `instaviz`, чтобы просмотреть объект кода и байт-код для функции, определенной в REPL:

```
>>> import instaviz
>>> def example():
...     a = 1
...     b = a + 1
...     return b
>>> instaviz.show(example)
```

API компилятора на C

Точка входа для компиляции модулей AST, `compiler_mod()`, переключается на разные функции компилятора в зависимости от типа модуля. Если в `mod` установлен тип `Module`, модуль компилируется в качестве единиц компиляции в свойство `c_stack`. Затем запускается функция `assemble()` для создания объекта `PyCodeObject` из стека единиц компиляции.

Возвращается новый объект кода, который передается на выполнение интерпретатором или кэшируется и сохраняется на диске в файле `.pyc`:

Python ▶ compile.c, строка 1820

```
static PyCodeObject *
compiler_mod(struct compiler *c, mod_ty mod)
{
```

¹ 2 байта. — Примеч. ред.

² <https://docs.python.org/3/library/dis.html#python-bytecode-instructions>.

```
PyCodeObject *co;
int addNone = 1;
static PyObject *module;
...
switch (mod->kind) {
    case Module_kind:
        if (!compiler_body(c, mod->v.Module.body)) {
            compiler_exit_scope(c);
            return 0;
        }
        break;
    case Interactive_kind:
        ...
    case Expression_kind:
        ...
    ...
    co = assemble(c, addNone);
    compiler_exit_scope(c);
    return co;
}
```

`compiler_body()` перебирает все команды в модуле и посещает их:

Python ▶ compile.c, строка 1782

```
static int
compiler_body(struct compiler *c, asdl_seq *stmts)
{
    int i = 0;
    stmt_ty st;
    PyObject *docstring;
    ...
    for (; i < asdl_seq_LEN(stmts); i++)
        VISIT(c, stmt, (stmt_ty)asdl_seq_GET(stmts, i));
    return 1;
}
```

Тип команды определяется вызовом `asdl_seq_GET()`, который проверяет тип узла AST.

Через макрос `VISIT` вызывает функцию для каждого типа команды из Python ▶ compile.c:

```
#define VISIT(C, TYPE, V) \
    if (!compiler_visit_##TYPE((C), (V))) \
        return 0; \
}
```

Затем для `stmt` (общий тип команды) компилятор вызывает `compiler_visit_stmt()`, после чего происходит переключение по всем возможным типам команд из `Parser ▶ Python.asdl`:

Python ▶ compile.c, строка 3375

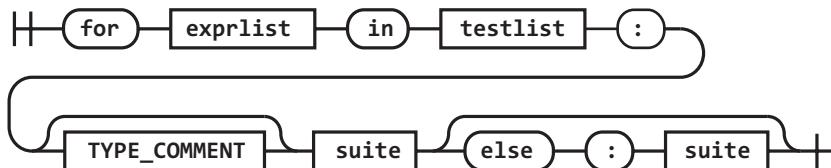
```
static int
compiler_visit_stmt(struct compiler *c, stmt_ty s)
{
    Py_ssize_t i, n;
    /* Всегда присваивайте номер строки следующей инструкции для stmt */
    SET_LOC(c, s);
    switch (s->kind) {
        case FunctionDef_kind:
            return compiler_function(c, s, 0);
        case ClassDef_kind:
            return compiler_class(c, s);
        ...
        case For_kind:
            return compiler_for(c, s);
        ...
    }

    return 1;
}
```

Пример команды с `for` в Python:

```
for i in iterable:
    # блок
else: # не обязательно, если iterable – False
    # блок
```

Команду `for` можно наглядно представить на синтаксической диаграмме:



Если команда относится к типу `for`, то `compiler_visit_stmt()` вызывает `compiler_for()`. Эквивалентная функция `compiler_*`() существует для всех типов выражений и команд. Более тривиальные типы встраивают строки с инструкциями байт-кода, тогда как некоторые более сложные типы команд вызывают другие функции.

Инструкции

У многих команд есть подкоманды. Цикл `for` имеет тело, но при присваивании и в итераторе также могут использоваться составные выражения.

Компилятор передает блоки с последовательностью инструкций состоянию компилятора. Структура данных инструкций содержит код операции, аргументы, целевой блок (для инструкции перехода; об этом ниже), а также номер строки команды.

Тип инструкции

Тип инструкции `instr` содержит следующие поля.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>i_jabs</code>	<code>unsigned</code>	Флаг инструкции абсолютного перехода
<code>i_jrel</code>	<code>unsigned</code>	Флаг инструкции относительного перехода
<code>i_lineno</code>	<code>int</code>	Номер строки, для которой создается эта инструкция
<code>i_opcode</code>	<code>unsigned char</code>	Код операции, представляемой инструкцией (см. Include ▶ Opcode.h)
<code>i_oparg</code>	<code>int</code>	Аргумент кода операции
<code>i_target</code>	<code>basicblock*</code>	Указатель на блок перехода из основного блока (<code>basicblock</code>), если значение <code>i_jrel</code> истинно

Инструкции перехода

Инструкции перехода позволяют «перепрыгнуть» из одной инструкции в другую. Они делятся на абсолютные (безусловные) и относительные (условные).

Инструкции абсолютного перехода задают точный номер инструкции в скомпилированном объекте кода, а инструкции относительного перехода — позицию перехода относительно другой инструкции.

Базовые блоки

Базовый блок (тип `basicblock`) содержит следующие поля.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>b_ialloc</code>	<code>int</code>	Длина массива инструкций (<code>b_instr</code>)
<code>b_instr</code>	<code>instr *</code>	Указатель на массив инструкций
<code>b_iused</code>	<code>int</code>	Количество использованных инструкций (<code>b_instr</code>)
<code>b_list</code>	<code>basicblock *</code>	Список блоков в данной единице компиляции (в обратном порядке)
<code>b_next</code>	<code>basicblock*</code>	Указатель на следующий блок при обычном потоке управления
<code>b_offset</code>	<code>int</code>	Смещение инструкции для блока, вычисляемое <code>assemble_jump_offsets()</code>
<code>b_return</code>	<code>unsigned</code>	Истинно, если вставлен код операции <code>RETURN_VALUE</code>
<code>b_seen</code>	<code>unsigned</code>	Используется для выполнения поиска в глубину по базовым блокам (см. «Ассемблер»)
<code>b_startdepth</code>	<code>int</code>	Глубина стека при входе в блок, вычисляемая <code>stackdepth()</code>

Операции и аргументы

Разным типам операций требуются разные аргументы. Например, `ADDOP_JREL` и `ADDOP_JABS` представляют «операцию сложения с переходом к относительной позиции» и «операцию сложения с переходом к абсолютной позиции» соответственно.

Существуют и другие макросы: `ADDOP_I` вызывает функцию `compiler_addop_i()`, которая добавляет операцию с целочисленным аргументом. `ADDOP_O` вызывает функцию `compiler_addop_o()`, которая добавляет операцию с аргументом `PyObject`.

АССЕМБЛЕР

После завершения этих стадий компиляции у компилятора имеется список блоков (frame blocks), каждый из которых содержит список инструкций и указатель на следующий блок. Ассемблер выполняет поиск в глубину (DFS, Depth-First Search) по базовым блокам и объединяет инструкции в одну последовательность байт-кода.

Структура данных ассемблера

Структура данных ассемблера `assembler` объявляется в файле Python ▶ `compile.c` и содержит следующие поля.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>a_bytecode</code>	<code>PyObject * (str)</code>	Строка, содержащая байт-код
<code>a_lineno</code>	<code>int</code>	Последнее значение <code>lineno</code> сгенерированной инструкции
<code>a_lineno_off</code>	<code>int</code>	Смещение последней <code>lineno</code> в байт-коде
<code>a_lnotab</code>	<code>PyObject * (str)</code>	Строка, содержащая <code>lnotab</code>
<code>a_lnotab_off</code>	<code>int</code>	Смещение в <code>lnotab</code>
<code>a_nblocks</code>	<code>int</code>	Количество достижимых блоков ¹
<code>a_offset</code>	<code>int</code>	Смещение в байт-коде
<code>a_postorder</code>	<code>basicblock **</code>	Список блоков в обратном порядке DFS

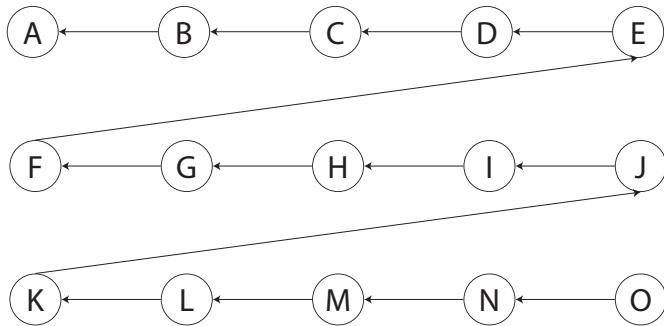
Алгоритм поиска в глубину

Ассемблер использует поиск в глубину (DFS) для обхода узлов в графе базовых блоков. Алгоритм DFS не является специфическим для CPython, но часто используется при обходе графов.

Если CST и AST являются структурами деревьев, то состояние компилятора является структурой графа, в которой узлам соответствуют базовые блоки, содержащие инструкции.

¹ Блоки, для которых существуют пути между парами вершин. — Примеч. ред.

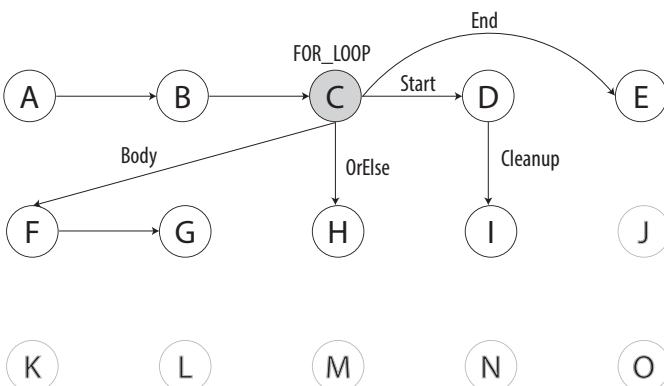
Базовые блоки связываются двумя графами. В одном графе используется обратный порядок создания, основанный на свойстве `b_list` каждого блока. Серия базовых блоков с именами от A до O будет выглядеть так:



Граф, созданный на `b_list`, используется для последовательного обхода всех блоков в единице компиляции.

Второй граф использует свойство `b_next` каждого блока. Этот список представляет поток управления. Вершины графа создаются вызовами `compiler_use_next_block(c, next)`, где `next` — следующий блок, для которого будет создана вершина, связанная ребром с текущим блоком (`c->u->u_curblock`).

Граф узла цикла `for` может выглядеть примерно так:



Используются оба графа — как последовательный, так и потока управления, — но реализация DFS использует граф потока управления.

API ассемблера на C

API ассемблера содержит точку входа `assemble()`, которая решает следующие задачи:

- Вычисляет количество блоков для выделения памяти.
- Гарантирует, что каждый блок, выходящий за границу, возвращает `None`.
- Выполняет разрешение всех смещений команд перехода, помеченных как относительные.
- Вызывает `dfs()` для выполнения обхода блоков в глубину.
- Генерирует все инструкции для компилятора.
- Вызывает `makecode()` с состоянием компилятора для генерирования `PyCodeObject`.

Python ▶ compile.c, строка 6010

```
static PyCodeObject *
assemble(struct compiler *c, int addNone)
{
    ...
    if (!c->u->u_curblock->b_return) {
        NEXT_BLOCK(c);
        if (addNone)
            ADDOP_LOAD_CONST(c, Py_None);
        ADDOP(c, RETURN_VALUE);
    }
    ...
    dfs(c, entryblock, &a, nblocks);

    /* Байт-код не может изменяться после вычисления смещений */
    assemble_jump_offsets(&a, c);

    /* Сгенерировать код в обратном порядке */
    for (i = a.a_nblocks - 1; i >= 0; i--) {
        b = a.a_postorder[i];
        for (j = 0; j < b->b_iused; j++)
            if (!assemble_emit(&a, &b->b_instr[j]))
                goto error;
    }
    ...

    co = makecode(c, &a);
error:
    assemble_free(&a);
    return co;
}
```

Поиск в глубину

Поиск в глубину выполняется функцией `dfs()` в Python ▶ `compile.c`. Эта функция переходит по указателям `b_next` в каждом из блоков, помечает их как просмотренные, изменяя значение `b_seen`, после чего добавляет их в список `a_postorder` ассемблера в обратном порядке.

Функция в цикле перебирает обратный список ассемблера и для каждого блока, если он содержит операцию перехода, рекурсивно вызывает `dfs()` для этого перехода:

Python ▶ `compile.c`, строка 5441

```
static void
dfs(struct compiler *c, basicblock *b, struct assembler *a, int end)
{
    int i, j;

    /* Исключение рекурсии для обычного потока управления.
       Так как количество блоков ограничено, неиспользуемое пространство
       в a_postorder (от a_nbblocks до end) может использоваться
       как стек для еще не упорядоченных блоков */
    for (j = end; b && !b->b_seen; b = b->b_next) {
        b->b_seen = 1;
        assert(a->a_nbblocks < j);
        a->a_postorder[--j] = b;
    }
    while (j < end) {
        b = a->a_postorder[j++];
        for (i = 0; i < b->b_iused; i++) {
            struct instr *instr = &b->b_instr[i];
            if (instr->i_jrel || instr->i_jabs)
                dfs(c, instr->i_target, a, j);
        }
        assert(a->a_nblocks < j);
        a->a_postorder[a->a_nbblocks++] = b;
    }
}
```

После того как ассемблер построит граф в CFG при помощи поиска в глубину, можно переходить к созданию объекта кода.

СОЗДАНИЕ ОБЪЕКТА КОДА

Задача функции `makecode()` — перебрать состояние компилятора и некоторых свойств ассемблера и поместить их в `PyCodeObject` вызовом `PyCode_New()`.

Имена переменных и констант помещаются в объект кода как свойства:

Python ▶ compile.c, строка 5893

```
static PyCodeObject *
makecode(struct compiler *c, struct assembler *a)
{
...
consts = consts_dict_keys_inorder(c->u->u_consts);
names = dict_keys_inorder(c->u->u_names, 0);
varnames = dict_keys_inorder(c->u->u_varnames, 0);
...
cellvars = dict_keys_inorder(c->u->u_cellvars, 0);
...
freevars = dict_keys_inorder(c->u->u_freevars,
                           PyTuple_GET_SIZE(cellvars));
...
flags = compute_code_flags(c);
if (flags < 0)
    goto error;

bytecode = PyCode_Optimize(a->a_bytecode, consts,
                           names, a->a_lnotab);
...
co = PyCode_NewWithPosOnlyArgs(
    posonlyargcount+posorkeywordargcount,
    posonlyargcount, kwonlyargcount, nlocals_int,
    maxdepth, flags, bytecode, consts, names,
    varnames, freevars, cellvars, c->c_filename,
    c->u->u_name, c->u->u_firstlineno, a->a_lnotab);
...
return co;
}
```

Возможно, вы заметите, что байт-код передается `PyCode_Optimize()` перед отправкой `PyCode_NewWithPosOnlyArgs()`. Эта функция является частью процесса оптимизации байт-кода в Python ▶ peephole.c.

Оптимизатор глазка перебирает инструкции байт-кода и в некоторых случаях заменяет их другими инструкциями. Например, существует оптимизатор, который удаляет все недостижимые инструкции после команды `return`.

ИСПОЛЬЗОВАНИЕ INSTAVIZ ДЛЯ ВЫВОДА ОБЪЕКТА КОДА

Все стадии работы компилятора можно собрать воедино с помощью модуля `instaviz`:

```
import instaviz

def foo():
    a = 2**4
    b = 1 + 5
    c = [1, 4, 6]
    for i in c:
        print(i)
    else:
        print(a)
    return c

instaviz.show(foo)
```

Этот фрагмент построит большое и сложное дерево графа AST. Инструкции байт-кода можно просмотреть по порядку:

Disassembled Code				
OpCode	Operation Name	Numeric Arg	Resolved Arg Value	Argument description
100	LOAD_CONST	1	16	16
125	STORE_FAST	0	a	a
100	LOAD_CONST	2	6	6
125	STORE_FAST	1	b	b
100	LOAD_CONST	3	1	1
100	LOAD_CONST	4	4	4
100	LOAD_CONST	2	6	6
103	BUILD_LIST	3	3	

Объект кода с именами переменных, константами и двоичным значением `co_code`:

Code Object Properties

Field	Value
co_argcount	0
co_cellvars	0
co_code	64017d0064027d0164036404640267037d02781c7c0244005d0c7d
co_consts	(None, 16, 6, 1, 4)
co_filename	test.py
co_firstlineno	4
co_freevars	0
co_kwonlyargcount	0
co_lnotab	b'\x00\x01\x04\x01\x04\x01\x01\n\x01\n\x01\x0c\x02\x08\x01'

Опробуйте эту возможность с другим, более сложным кодом, чтобы больше узнать о компиляторе CPython и объектах кода.

ПРИМЕР: РЕАЛИЗАЦИЯ ОПЕРАТОРА «ПОЧТИ РАВНО»

Теперь, после рассмотрения компилятора, инструкций байт-кода и ассемблера, мы сможем изменить CPython и добавить поддержку оператора «почти равно», который был скомпилирован в грамматику в предыдущей главе.

Сначала необходимо добавить внутреннее определение `#define` для оператора `Py_EQ`, чтобы на него можно было ссылаться внутри расширенных функций сравнения для `PyObject`.

Откройте файл `Include > object.h` и найдите следующие команды `#define`:

```
/* Коды операций расширенного сравнения */
#define Py_LT 0
#define Py_LE 1
#define Py_EQ 2
#define Py_NE 3
#define Py_GT 4
#define Py_GE 5
```

Добавим дополнительное значение Py_EQ со значением 6:

```
/* Новый оператор сравнения "почти равно" */
#define Py_EQ 6
```

Прямо под этим выражением располагается макрос Py_RETURN_RICHCOMPARE. Обновите этот макрос и добавьте в него условие `case` для Py_EQ:

```
/*
 * Макрос для реализации расширенного сравнения
 *
 * Необходим макрос, потому что может использоваться
 * любой С-совместимый тип
 */
#define Py_RETURN_RICHCOMPARE(val1, val2, op) \
do { \
    switch (op) { \
        case Py_EQ: if ((val1) == (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE; \
        case Py_NE: if ((val1) != (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE; \
        case Py_LT: if ((val1) < (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE; \
        case Py_GT: if ((val1) > (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE; \
        case Py_LE: if ((val1) <= (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE; \
        case Py_GE: if ((val1) >= (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE; \
        /* + */ case Py_EQ: if ((val1) == (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE; \
        default: \
            Py_UNREACHABLE(); \
    } \
} while (0)
```

В файле `Objects ▶ object.c` присутствует проверка того, что оператор лежит в диапазоне от 0 до 5. Так как вы добавили значение 6, проверку необходимо обновить:

Objects ▶ object.c, строка 709

```
PyObject *
PyObject_RichCompare(PyObject *v, PyObject *w, int op)
{
    PyThreadState *tstate = _PyThreadState_GET();

    assert(Py_LT <= op && op <= Py_GE);
```

Замените последнюю строку следующей:

```
assert(Py_LT <= op && op <= Py_EQ);
```

Затем необходимо обновить код операции `COMPARE_OP` для поддержки `Py_ALE` как значения для типа оператора.

Сначала отредактируйте файл `Objects ▶ object.c` и добавьте `Py_ALE` в список `_Py_SwappedOp`. Этот список используется для определения того, содержит ли пользовательский класс только один магический (dunder) метод оператора, но не содержит другого.

Например, если вы определили класс `Coordinate`, оператор равенства можно определить реализацией магического метода `__eq__`:

```
class Coordinate:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __eq__(self, other):  
        if isinstance(other, Coordinate):  
            return (self.x == other.x and self.y == other.y)  
        return super(self, other).__eq__(other)
```

И хотя вы не реализовали `__ne__` (не равно, not equal) для `Coordinate`, CPython предполагает, что может применяться оператор, обратный `__eq__`:

```
>>> Coordinate(1, 100) != Coordinate(2, 400)  
True
```

В файле `Objects ▶ object.c` найдите список `_Py_SwappedOp` и добавьте `Py_ALE` в его конец. Затем добавьте `"~="` в конец списка `opstrings`:

```
int _Py_SwappedOp[] = {Py_GT, Py_GE, Py_EQ, Py_NE, Py_LT, Py_LE, Py_ALE};  
  
static const char * const opstrings[]  
= {"<", "<=", "==", "!=" , ">", ">=", "~="};
```

Откройте файл `Lib/opcode.py` и отредактируйте кортеж операторов расширенного сравнения:

```
cmp_op = ('<', '<=', '==', '!=', '>', '>=')
```

Добавьте новый оператор в конец кортежа:

```
cmp_op = ('<', '<=', '==', '!=', '>', '>=', '~=')
```

Список `opstrings` используется для сообщений об ошибках, если операторы расширенного сравнения не реализованы для класса.

Теперь можно обновить компилятор для обработки условия в свойстве `PyCmp_A1E` в узле `BinOp`. Откройте файл `Python > compile.c` и найдите `compiler_addcompare()`:

Python > compile.c, строка 2479

```
static int compiler_addcompare(struct compiler *c, cmpop_ty op)
{
    int cmp;
    switch (op) {
        case Eq:
            cmp = Py_EQ;
            break;
        case NotEq:
            cmp = Py_NE;
            break;
        case Lt:
            cmp = Py_LT;
            break;
        case LtE:
            cmp = Py_LE;
            break;
        case Gt:
            cmp = Py_GT;
            break;
        case GtE:
            cmp = Py_GE;
            break;
    }
}
```

Добавьте еще один оператор `case` в команду `switch`, чтобы сопоставить `comp_op_A1E` дерева абстрактного синтаксиса с кодом операции сравнения `PyCmp_A1E`:

```
...
    case A1E:
        cmp = Py_A1E;
        break;
```

Теперь можно запрограммировать поведение оператора «почти равно» по следующему сценарию:

- `1 ~= 2` — `False`;
- `1 ~= 1.01` — `True` с округлением вниз.

Чтобы добиться нужного результата, придется написать дополнительный код. Пока можно привести оба числа с плавающей точкой к целым числам и сравнить их.

API CPython содержит много функций для работы с типами `PyLong` (`int`) и `PyFloat` (`float`). Эта тема будет рассмотрена в главе «Объекты и типы».

Найдите функцию `float_richcompare()` в `Objects` ► `floatobject.c` и добавьте в определение `Compare`: `goto` следующее условие `case`:

Objects ► `floatobject.c`, строка 358

```
static PyObject*
float_richcompare(PyObject *v, PyObject *w, int op)
{
    ...
    case Py_GT:
        r = i > j;
        break;
    /* НАЧАЛО нового кода */
    case Py_Ale: {
        double diff = fabs(i - j);
        double rel_tol = 1e-9; // Относительная погрешность
        double abs_tol = 0.1; // Абсолютная погрешность
        r = (((diff <= fabs(rel_tol * j)) ||
              (diff <= fabs(rel_tol * i))) ||
              (diff <= abs_tol));
    }
    break;
}
/* КОНЕЦ нового кода */
return PyBool_FromLong(r);
```

Этот код обрабатывает сравнение чисел с плавающей точкой при использовании оператора «почти равно». В нем используется логика, сходная с логикой `math.isclose()`, определенной в PEP 485, но с жестко закодированной абсолютной погрешностью `0.1`.

Также необходимо внести изменения еще в одну защитную проверку, которая находится в цикле вычисления Python `ceval.c`. Цикл вычисления рассматривается в следующей главе.

Найдите следующий фрагмент кода:

```
...
case TARGET(COMPARE_OP): {
    assert(oparg <= Py_GE);
```

Замените проверку следующей:

```
assert(oparg <= Py_Ale);
```

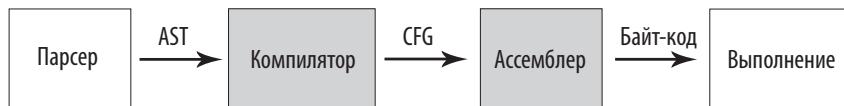
После перекомпиляции CPython откройте REPL и протестируйте оператор:

```
$ ./python
>>> 1.0 ~= 1.01
True
>>> 1.02 ~= 1.01
True
>>> 1.02 ~= 2.01
False
>>> 1 ~= 1.01
True
>>> 1 ~= 1
True
>>> 1 ~= 2
False
>>> 1 ~= 1.9
False
>>> 1 ~= 2.0
False
>>> 1.1 ~= 1.101
True
```

В последующих главах эта реализация будет расширена для других типов.

ВЫВОДЫ

В этой главе вы узнали, как синтаксически разобранный модуль Python преобразуется в таблицу символических имен, состояние компиляции, а затем в серию операций байт-кода:



Теперь наступает следующая фаза — интерпретатор CPython в основном цикле вычисления должен выполнить эти модули. В следующей главе вы узнаете, как выполняются объекты кода.

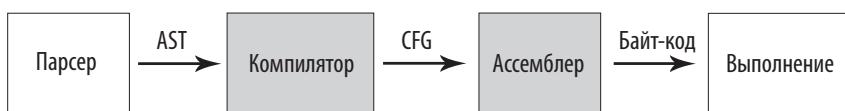
Цикл вычисления

Мы увидели, как Python-код превращается в дерево абстрактного синтаксиса и компилируется в объекты кода. Эти объекты кода содержат списки отдельных операций в форме байт-кода.

Но чтобы объекты кода заработали, им кое-чего не хватает. Им необходимы входные данные. В Python входные данные принимают вид локальных и глобальных переменных.

В этой главе вы познакомитесь с концепцией **стека значений**, в котором переменные создаются, изменяются и используются операциями байт-кода скомпилированных объектов кода.

Код в CPython выполняется в центральном цикле, который называется **циклом вычисления**. Интерпретатор CPython вычисляет и выполняет объект кода, полученный из файла .рус или от компилятора:



В цикле вычисления берется каждая инструкция байт-кода и выполняется в кадровой системе стека.

ПРИМЕЧАНИЕ

Кадры стека – тип данных, используемый во многих средах выполнения, не только в Python. Стековые кадры позволяют вызывать функции и возвращать переменные из вызовов. Также они содержат аргументы, локальные переменные и другую информацию с состоянием.

Кадр стека существует для каждого вызова функции, причем кадры объединяются в стек последовательно. Содержимое стека CPython выводится каждый раз при выдаче необработанного исключения:

```
Traceback (most recent call last):
  File "example_stack.py", line 8, in <module> <--- Кадр
    function1()
  File "example_stack.py", line 5, in function1 <--- Кадр
    function2()
  File "example_stack.py", line 2, in function2 <--- Кадр
    raise RuntimeError
RuntimeError
```

ИСХОДНЫЕ ФАЙЛЫ

Ниже перечислены исходные файлы, относящиеся к циклу вычисления.

ФАЙЛ	НАЗНАЧЕНИЕ
Python ► ceval.c	Базовая реализация цикла вычисления
Python ► ceval-gil.h	Определение GIL ¹ и управляющий алгоритм

ВАЖНЫЕ ТЕРМИНЫ

Ниже перечислены некоторые ключевые термины, встречающиеся в этой главе:

- **Цикл вычисления** берет объект кода и преобразует его в серию объектов кадров.
- Интерпретатор имеет хотя бы один **поток** (thread).
- Каждый поток характеризуется **состоянием потока**.
- Объекты кадров выполняются в **стеке кадров**.
- Для ссылок на переменные используется **стек значений**.

¹ Глобальная блокировка интерпретатора. — Примеч. ред.

ПОСТРОЕНИЕ СОСТОЯНИЯ ПОТОКА

Прежде чем кадр может быть выполнен, он должен связаться с потоком. В CPython сразу несколько потоков могут выполняться одновременно внутри одного интерпретатора. Состояние интерпретатора включает связанный список таких потоков.

CPython всегда имеет хотя бы один поток, и каждый поток обладает собственным состоянием.

СМ. ТАКЖЕ

Потоки более подробно рассматриваются в главе «Параллелизм и конкурентность».

Тип состояния потока

Тип состояния потока `PyThreadState` содержит свыше тридцати свойств, среди которых:

- Уникальный идентификатор.
- Связанный список с состояниями других потоков.
- Состояние интерпретатора, которым был порожден поток.
- Кадр, выполняемый в настоящее время.
- Текущая глубина рекурсии.
- Необязательные функции трассировки.
- Исключение, обрабатываемое в настоящий момент.
- Асинхронное исключение, обрабатываемое в настоящий момент.
- Стек поднятых исключений, если их несколько (например, внутри блока `except`).
- Счетчик GIL.
- Счетчики асинхронного генератора.

Исходные файлы

Исходники, относящиеся к состоянию потоков, распределены на несколько файлов:

ФАЙЛ	НАЗНАЧЕНИЕ
Python ► thread.c	Реализация API потоков
Include ► threadstate.h	Некоторые API состояния потока и определения типов
Include ► pystate.h	API состояния интерпретатора и определения типов
Include ► pythread.h	API потоков
Include ► cpython pystate.h	Некоторые API потоков и состояния интерпретатора

ПОСТРОЕНИЕ ОБЪЕКТОВ КАДРОВ

Скомпилированные объекты кода вставляются в объекты кадров. Объекты кадров являются типом Python, поэтому к ним можно обращаться как из C, так и из Python.

Объекты кадров также содержат другие данные среды выполнения, необходимые для исполнения инструкций в объектах кода. Эти данные включают локальные переменные, глобальные переменные и встроенные модули.

Тип объекта кадра

Тип объекта кадра — `PyObject` со следующими дополнительными свойствами.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>f_back</code>	<code>PyFrameObject *</code>	Указатель на предыдущий кадр в стеке или <code>NULL</code> для первого кадра
<code>f_blockstack</code>	<code>PyTryBlock[]</code>	Последовательность блоков <code>for</code> , <code>try</code> и <code>loop</code>
<code>f_builtins</code>	<code>PyObject * (dict)</code>	Таблица символических имен для модуля <code>builtin</code>

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
f_code	PyCodeObject *	Объект кода, который нужно выполнить
f_executing	char	Флаг, указывающий, что кадр еще выполняется
f_gen	PyObject *	Ссылка на генератор или NULL
f_globals	PyObject *(dict)	Таблица глобальных символических имен (PyDictObject)
f_iblock	int	Индекс кадра в f_blockstack
f_lasti	int	Последняя инструкция
f_lineno	int	Номер текущей строки
f_locals	PyObject *	Таблица локальных символических имен (произвольное отображение)
f_localsplus	PyObject *[]	Объединение locals и stack
f_stacktop	PyObject **	Следующий свободный слот в f_valuestack
f_trace	PyObject *	Указатель на пользовательскую функцию трассировки (см. «Трассировка выполнения кадров»)
f_trace_lines	char	Переключение пользовательской функции трассировки на трассировку на уровне строки
f_trace_opcodes	char	Переключение пользовательской функции трассировки на трассировку на уровне кода операции
f_valuestack	PyObject **	Указатель на последнее локальное значение

Исходные файлы

Исходные файлы, относящиеся к объектам кадров:

ФАЙЛ	НАЗНАЧЕНИЕ
Objects ▶ frameobject.c	Реализация объекта кадра и Python API
Include ▶ frameobject.h	API объекта кадра и определение типа

API инициализации объекта кадра

API инициализации объекта кадра, `PyEval_EvalCode()`, является точкой входа для вычисления объекта кода и оберткой для внутренней функции `_PyEval_EvalCode()`.

ПРИМЕЧАНИЕ

`_PyEval_EvalCode()` – сложная функция, которая определяет многие особенности поведения как объектов кадров, так и цикла интерпретатора. Очень важно понимать эту функцию, так как она демонстрирует некоторые принципы архитектуры интерпретатора CPython.

В этом разделе мы последовательно разберем логику `_PyEval_EvalCode()`.

`_PyEval_EvalCode()` определяет группу аргументов:

- `tstate`: аргумент `PyThreadState*`, указывающий на состояние потока, в котором будет вычисляться этот код;
- `co`: аргумент `PyCodeObject*` с кодом, который должен быть помещен в объект кадра;
- `globals`: `PyObject* (dict)` с именами переменных (ключи) и их значениями;
- `locals`: `PyObject* (dict)` с именами переменных (ключи) и их значениями.

ПРИМЕЧАНИЕ

В Python локальные и глобальные переменные хранятся в виде словаря.

Для обращения к словарю можно воспользоваться встроенными функциями `locals()` и `globals()`:

```
>>> a = 1
>>> print(locals()["a"])
1
```

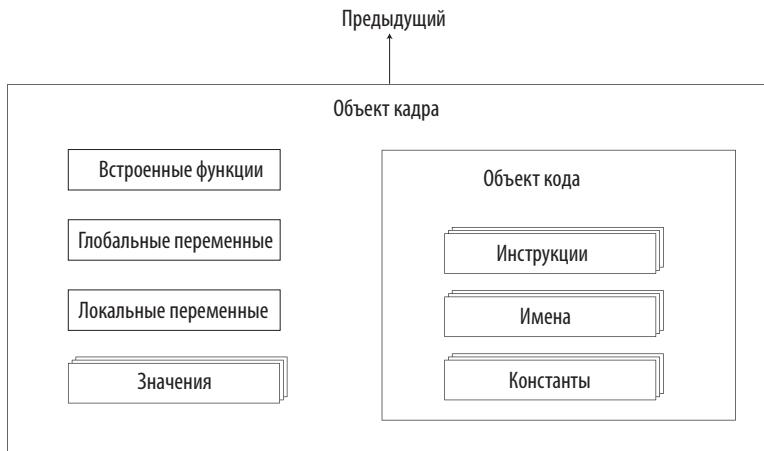
Другие аргументы не являются обязательными и не используются в базовом API:

- `argcount`: количество позиционных аргументов;
- `args`: `PyObject*` (`tuple`) со значениями позиционных аргументов;
- `closure`: кортеж со строками, которые объединяются в поле `co_freevars` объекта кода;
- `defcount`: длина списка значений по умолчанию для позиционных аргументов;
- `defs`: список значений по умолчанию для позиционных аргументов;
- `kwargs`: список значений именованных аргументов;
- `kwcount`: количество именованных аргументов;
- `kwdefs`: словарь со значениями по умолчанию для именованных аргументов;
- `kwnames`: список имён именованных аргументов;
- `name`: имя команды вычисления (строка);
- `qualname`: уточненное имя команды вычисления (строка).

Вызов `_PyFrame_New_NoTrack()` создает новый кадр. Этот API также доступен из С API с использованием `PyFrame_New()`. `_PyFrame_New_NoTrack()` создает новый объект `PyFrameObject` по следующей схеме:

1. Свойству `f_back` присваивается последний кадр состояния потока.
2. С помощью установки значения свойства `f_builtins` и загрузки модуля `builtins` вызовом `PyModule_GetDict()` загружаются текущие встроенные функции.
3. Свойству `f_code` задается вычисляемый объект кода.
4. Свойству `f_valuestack` задается пустой стек значений.
5. Указателю `f_stacktop` присваивается `f_valuestack`.
6. Свойству `f_globals` присваивается аргумент `globals`.
7. Свойству `f_locals` присваивается новый словарь.
8. Свойству `co_firstlineno` присваивается значение `f_lineno`, чтобы трассировка содержала номера строк.
9. Всем остальным свойствам значение задается по умолчанию.

Теперь, когда создан новый экземпляр `PyFrameObject`, можно построить аргументы объекта кадра:



Преобразование именованных параметров в словарь

Определения функций могут содержать универсальное обозначение `**kwargs` для именованных аргументов:

```
def example(arg, arg2=None, **kwargs):
    print(kwargs["x"], kwargs["y"]) # Преобразуется в ключ словаря
example(1, x=2, y=3) # 2 3
```

В этом скрипте создается новый словарь, а непреобразованные аргументы копируются. Затем в локальной области видимости кадра создается переменная с именем `kwargs`.

Преобразование позиционных аргументов в переменные

Каждый из позиционных аргументов (если они предоставлены) задается как локальная переменная. В Python аргументы функций уже являются локальными переменными в теле функции. Когда позиционный аргумент определяется со значением, он становится доступным в области видимости функции:

```
def example(arg1, arg2):
    print(arg1, arg2)
example(1, 2) # 1 2
```

Счетчик ссылок для этих переменных увеличивается, поэтому сборщик мусора не удаляет их до вычисления кадра, например при завершении функции и возврате результата.

Упаковка позиционных аргументов в `*args`

Как и в случае с `**kwargs`, аргумент функции с символом `*` может использоваться для перехвата всех оставшихся позиционных аргументов. Создается локальная переменная типа кортеж с именем `*args`:

```
def example(arg, *args):
    print(arg, args[0], args[1])

example(1, 2, 3) # 1 2 3
```

Загрузка именованных аргументов

Если функция вызывается с именованными аргументами и значениями, то все оставшиеся именованные аргументы, переданные при вызове, которые не преобразуются ни в именованные, ни в позиционные аргументы, попадают в словарь.

Например, аргумент `e` не является ни позиционным, ни именованным, поэтому он добавляется в `**remaining`:

```
>>> def my_function(a, b, c=None, d=None, **remaining):
    print(a, b, c, d, remaining)

>>> my_function(a=1, b=2, c=3, d=4, e=5)
(1, 2, 3, 4, {"e": 5})
```

Преобразование значений из словаря именованных аргументов происходит после распаковки всех остальных аргументов. Для исключительно позиционных аргументов PEP 570 цикл именованных аргументов начинается с `co_posonlyargcount`. Если символ `/` использовался в третьем аргументе, то значение `co_posonlyargcount` будет равно 2.

`PyDict_SetItem()` вызывается для каждого оставшегося аргумента, чтобы он был добавлен в словарь `locals`. При выполнении каждый из именованных аргументов становится локальной переменной с соответствующей областью видимости.

ПРИМЕЧАНИЕ

Исключительно позиционные аргументы (position-only arguments) появились в Python 3.8. Представленные в PEP 570, исключительно позиционные аргументы не позволяют пользователям вашего API использовать позиционные аргументы по синтаксису именованных.

Например, следующая простая функция преобразует температуру по Фаренгейту в температуру по шкале Цельсия. Обратите внимание на косую черту (/) – это специальный аргумент, отделяющий исключительно позиционные аргументы от других аргументов:

```
def to_celsius(fahrenheit, /, options=None):
    return (fahrenheit-32)*5/9
```

Все аргументы слева от / должны передаваться только как позиционные. Аргументы справа могут передаваться как позиционные или как именованные:

```
>>> to_celsius(110)
```

Если в вызов функции, объявленной с исключительно позиционным аргументом, передать именованный аргумент, возникнет ошибка типа `TypeError`:

```
>>> to_celsius(fahrenheit=110)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: to_celsius() got some positional-only arguments
passed as keyword arguments: 'fahrenheit'
```

Если именованный аргумент определяется со значением, то оно будет доступно в этой области видимости:

```
def example(arg1, arg2, example_kwarg=None):
    print(example_kwarg) # example_kwarg уже является локальной переменной.
```

Добавление отсутствующих позиционных аргументов

Любые дополнительные позиционные аргументы, переданные при вызове функции, добавляются в кортеж `*args`. Если кортеж не существует, выдается исключение.

Добавление отсутствующих именованных аргументов

Любые дополнительные именованные аргументы, переданные при вызове функции, добавляются в словарь `**kwargs`.

Если этот словарь не существует, выдается исключение.

Свертка замыканий

Все имена замыканий добавляются в список свободных имен переменных объекта кода.

Создание генераторов, сопрограмм и асинхронных генераторов

Если вычисляемый объект кода содержит флаг, указывающий на то, что это генератор, сопрограмма или асинхронный генератор, то создается новый кадр с использованием одного из уникальных методов библиотек генераторов, сопрограмм или асинхронных библиотек, а текущий кадр добавляется как свойство.

СМ. ТАКЖЕ

API и реализация генераторов, сопрограмм и асинхронных кадров рассматриваются в главе «Параллелизм и конкурентность».

Возвращается новый кадр, а исходный кадр не вычисляется. Кадр будет вычисляться только тогда, когда генератор, сопрограмма или асинхронный метод вызываются для выполнения его цели.

Наконец, с новым кадром вызывается `_PyEval_EvalFrame()`.

ВЫПОЛНЕНИЕ КАДРА

Как упоминалось ранее в главах «Лексический анализ и парсинг с использованием синтаксических деревьев» и «Компилятор», объект кода содержит выполняемый байт-код, закодированный в двоичном формате. Также он содержит список переменных и таблицу символических имен.

Локальные и глобальные переменные определяются во время выполнения в зависимости от того, как была вызвана функция, модуль или блок. Эта информация добавляется в кадр функцией `_PyEval_EvalCode()`.

Существуют и другие варианты использования кадров — например, декоратор сопрограмм, который динамически генерирует кадр с целью в качестве переменной.

Функция открытого API `PyEval_EvalFrameEx()` вызывает функцию вычисления кадра, настроенную для интерпретатора, из свойства `eval_frame`. Предложение PEP 523 сделало вычисление кадра подключаемым (Python 3.7 и выше).

`_PyEval_EvalFrameDefault()` — установленная по умолчанию функция вычисления кадра, и это единственный вариант, входящий в комплект поставки CPython.

ПРИМЕЧАНИЕ

При чтении файла Python `ceval.c` можно заметить, сколько раз в нем используются макросы С.

Макросы С открывают возможность повторного использования кода без лишних затрат на вызов функций. Компилятор преобразует макросы в код С, а затем компилирует его.

После установки официального расширения C/C++ в Visual Studio Code будет выводиться подстановка макроса:

```

1122     dtrace_function_entry(f);
1123
1124     co = f->f_code;
1125     names = co->co_names;
1126     consts = co->co_consts;
1127     fastlocals = f->f_localsplus;
1128     freevars = f->f_localsplus + co->co_nlocals;
1129     assert(#define _Py_IS_ALIGNED(p,a) (((uintptr_t)(p) & (uintptr_t)((a) - 1)))
1130     assert();
1131     assert( Check if pointer "p" is aligned to "a"-bytes boundary.
1132     assert(_Py_IS_ALIGNED(PyBytes_AS_STRING(co->co_code), sizeof(_Py_CODEUNIT)));
1133     first_instr = (_Py_CODEUNIT *) PyBytes_AS_STRING(co->co_code);
1134     /*
1135      f->f_lasti refers to the index of the last instruction,
1136      unless it's -1 in which case next_instr should be first_instr.
1137
1138      YIELD_FROM sets f_lasti to itself, in order to repeatedly yield
1139      multiple values.
1140
1141 When the PREDICT() macros are enabled, some oncode pairs follow in

```

В CLion выделите макрос и нажмите Alt+пробел, чтобы просмотреть его определение.

Эта центральная функция собирает все воедино и приводит в действие ваш код. В ней воплощены целые десятилетия оптимизации, так что даже одна строка кода может значительно повлиять на производительность CPython в целом.

Все, что выполняется в CPython, проходит через функцию вычисления кадра.

Трассировка выполнения кадра

В Python 3.7 и выше можно пошагово отслеживать выполнение кадров, установив атрибут трассировки для текущего потока. Тип `PyFrameObject` содержит свойство `f_trace` типа `PyObject *`. Предполагается, что значение указывает на функцию Python.

Следующий пример кода выбирает в качестве глобальной функции трассировки функцию с именем `my_trace()`, которая получает стек из текущего кадра, выводит дизассемблированные коды операций на экран и добавляет информацию для отладки:

cpython-book-samples › 31 › my_trace.py

```
import sys
import dis
import traceback
import io

def my_trace(frame, event, args):
    frame.f_trace_opcodes = True
    stack = traceback.extract_stack(frame)
    pad = " "*len(stack) + "|"
    if event == "opcode":
        with io.StringIO() as out:
            dis.disco(frame.f_code, frame.f_lasti, file=out)
            lines = out.getvalue().split("\n")
            [print(f"{pad}{l}") for l in lines]
    elif event == "call":
        print(f"{pad}Calling {frame.f_code}")
    elif event == "return":
        print(f"{pad}Returning {args}")
    elif event == "line":
        print(f"{pad}Changing line to {frame.f_lineno}")
    else:
        print(f"{pad}{frame} ({event} - {args})")
    print(f"{pad}-----")
    return my_trace
sys.settrace(my_trace)

# Выполнение кода для демонстрации
eval('"-'.join([letter for letter in "hello"]))'
```

Функция `sys.settrace()` назначает переданную функцию функцией трассировки по умолчанию для текущего состояния потока. У всех новых кадров, созданных после этого вызова, поле `f_trace` инициализируется этой функцией.

Следующий фрагмент выводит код из каждого стека и указатель на следующую операцию перед ее выполнением. Если кадр возвращает значение, выводится команда `return`:

```
→ cpython git:(master) ✘ ./python.exe my_trace.py
| Calling <code object <module> at 0x104cdcc110, file "<string>", line 1>
| |
| Changing line to 1
| |
| 1 -->     0 LOAD_CONST          0 ('-')
|   2 LOAD_METHOD           0 (join)
|   4 LOAD_CONST            1 (<<code object <listcomp> at 0x104cdce0, file "<string>", line 1>)
|   6 LOAD_CONST            2 ('<listcomp>')
|   8 MAKE_FUNCTION         0
| 10 LOAD_CONST           3 ('hello')
| 12 GET_ITER
| 14 CALL_FUNCTION         1
| 16 CALL_METHOD           1
| 18 RETURN_VALUE

→ | 1     0 LOAD_CONST          0 ('-')
|   2 LOAD_METHOD           0 (join)
|   4 LOAD_CONST            1 (<<code object <listcomp> at 0x104cdce0, file "<string>", line 1>)
|   6 LOAD_CONST            2 ('<listcomp>')
|   8 MAKE_FUNCTION         0
| 10 LOAD_CONST           3 ('hello')
| 12 GET_ITER
| 14 CALL_FUNCTION         1
| 16 CALL_METHOD           1
| 18 RETURN_VALUE
```

Полный список возможных инструкций байт-кода приведен в документации модуля `dis`¹.

СТЕК ЗНАЧЕНИЙ

Внутри основного цикла вычисления создается стек значений. Стек содержит список указателей на экземпляры `PyObject`. Это могут быть переменные, ссылки на функции (которые в Python являются объектами) или любые другие объекты Python.

Инструкции байт-кода в цикле вычисления получают входные данные из стека значений.

Пример операции байт-кода: `BINARY_OR`

Бинарные операции, которые рассматривались в предшествующих главах, компилируются в одну инструкцию.

¹ <https://docs.python.org/3/library/dis.html#python-bytecode-instructions>.

Допустим, вы вставили в Python оператор `or`:

```
if left or right:  
    pass
```

Компилятор преобразует оператор `or` в инструкцию `BINARY_OR`:

```
static int  
binop(struct compiler *c, operator_ty op)  
{  
    switch (op) {  
        case Add:  
            return BINARY_ADD;  
        ...  
        case BitOr:  
            return BINARY_OR;
```

В цикле вычисления секция `case` для `BINARY_OR` получает два значения из стека значений (левый и правый operandы), после чего вызывает `PyNumber_Or` для этих двух объектов:

```
...  
case TARGET(BINARY_OR): {  
    PyObject *right = POP();  
    PyObject *left = TOP();  
    PyObject *res = PyNumber_Or(left, right);  
    Py_DECREF(left);  
    Py_DECREF(right);  
    SET_TOP(res);  
    if (res == NULL)  
        goto error;  
    DISPATCH();  
}
```

Результат (`res`) назначается вершиной стека, переопределяя текущее верхнее значение.

Моделирование стека значений

Чтобы понять, как работает цикл вычислений, необходимо понимать смысл стека значений.

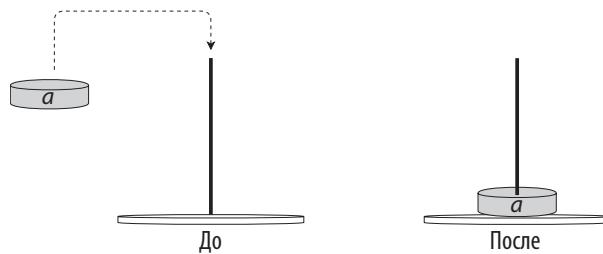
Стек значений можно представить себе в виде деревянного колышка, на который нанизываются цилиндры. При этом цилиндры можно добавлять или снимать только по одному, и всегда только на вершину стека или с нее.

В CPython для добавления объектов в стек значений используется макрос `PUSH(a)`, где `a` — указатель на `PyObject`.

Предположим, вы создали `PyLong` со значением 10 и занесли его в стек значений:

```
PyObject *a = PyLong_FromLong(10);
PUSH(a);
```

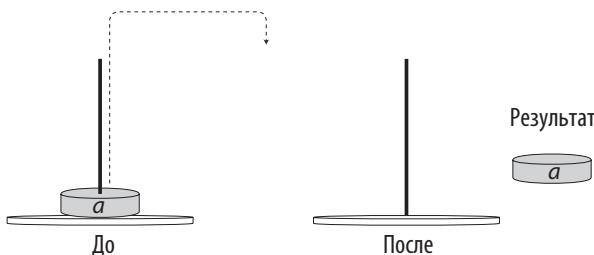
Это действие приводит к следующему эффекту:



В следующей операции для получения этого значения используется макрос `POP()`(получение верхнего значения стека):

```
PyObject *a = POP(); // a - PyLongObject со значением 10
```

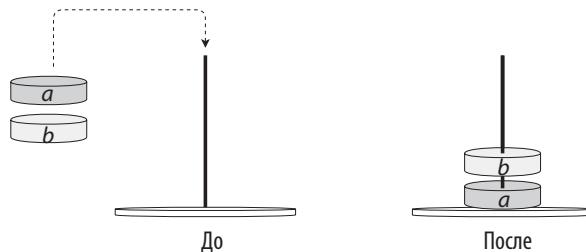
Это действие возвращает верхнее значение, а стек значений остается пустым:



Допустим, в стек были добавлены два значения:

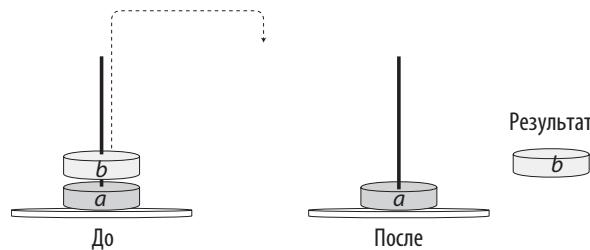
```
PyObject *a = PyLong_FromLong(10);
PyObject *b = PyLong_FromLong(20);
PUSH(a);
PUSH(b);
```

Они сохраняются в стеке в порядке добавления, поэтому `a` попадет на вторую позицию в стеке:



При получении верхнего значения из стека вы получите указатель на `b`, потому что именно эта переменная сейчас находится на вершине:

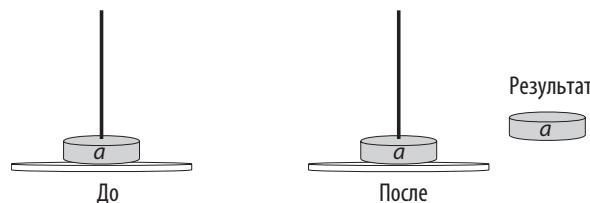
```
PyObject *val = POP(); // Возвращает указатель на b
```



Если вам понадобится получить указатель на верхнее значение в стеке без его извлечения, можно воспользоваться операцией `PEEK(v)`, где `v` — позиция стека:

```
PyObject *first = PEEK(0);
```

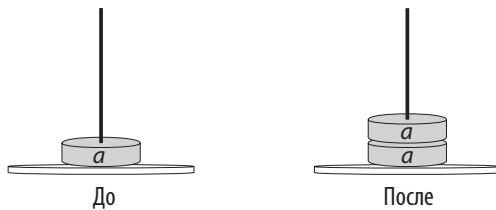
0 соответствует вершине стека, а 1 — второй позиции:



Для клонирования значения с вершины стека используется макрос `DUP_TOP()`:

```
DUP_TOP();
```

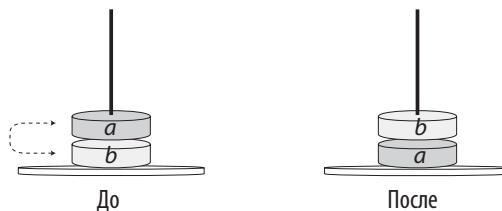
Операция копирует значение с вершины стека, создавая два указателя на один объект:



Макрос ROT_TWO меняет местами первое и второе значения:

```
ROT_TWO();
```

Операция переставляет первое значение на место второго и наоборот:



Эффект стека

Каждый код операции имеет заранее определенный **эффект стека**, вычисляемый вызовом `stack_effect()` из файла `Python ▶ compile.c`. Функция возвращает дельту в количестве значений в стеке для каждого кода операции.

Эффект стека может иметь положительное, отрицательное или нулевое значение. Если после выполнения операции эффект стека (например, +1) не равен дельте в стеке значений, выдается исключение.

ПРИМЕР: ДОБАВЛЕНИЕ ЭЛЕМЕНТА В СПИСОК

В Python доступен метод `append()` на созданном объекте списка:

```
my_list = []
my_list.append(obj)
```

В этом примере `obj` — объект, который добавляется в конец списка.

В этой операции задействованы две другие операции:

1. `LOAD_FAST` для загрузки `obj` на вершину стека значений из списка `locals` в кадре.
2. `LIST_APPEND` для добавления объекта.

`LOAD_FAST` состоит из пяти шагов:

1. Указатель на `obj` загружается из `GETLOCAL()`, при этом аргументом операции является загружаемая переменная. Список указателей на переменные хранится в `fastlocals` — копии атрибута `PyFrame` из `f_localsplus`. Аргумент операции — число, ссылающееся на индекс в массиве указателей `fastlocals`. Это означает, что Python загружает локальную переменную как копию указателя (вместо того, чтобы проводить поиск по имени переменной).
2. Если переменная более не существует, выдается исключение несвязанной локальной переменной (`UnboundLocalError`).
3. Счетчик ссылок для значения (в данном случае `obj`) увеличивается на 1.
4. Указатель на `obj` помещается на вершину стека значений.
5. Вызывается макрос `FAST_DISPATCH`. Если трассировка включена, то цикл выполняется снова с полной трассировкой. Если трассировка отключена, для `fast_next_opcode` вызывается `goto`. Операция `goto` осуществляет переход к началу цикла за следующей инструкцией.

Пять шагов `LOAD_FAST`:

```
...
    case TARGET(LOAD_FAST): {
        PyObject *value = GETLOCAL(oparg); // 1.
        if (value == NULL) {
            format_exc_check_arg(
                PyExc_UnboundLocalError,
                UNBOUNDLOCAL_ERROR_MSG,
                PyTuple_GetItem(co->co_varnames, oparg));
            goto error; // 2.
        }
        Py_INCREF(value); // 3.
        PUSH(value); // 4.
        FAST_DISPATCH(); // 5.
    }
...

```

Указатель на `obj` теперь находится на вершине стека значений, и выполняется следующая инструкция `LIST_APPEND`.

Многие операции байт-кода ссылаются на базовые типы, такие как `PyUnicode` или `PyNumber`. Например, `LIST_APPEND` добавляет объект в конец списка. Для этого он извлекает указатель из стека значений и возвращает указатель на последний объект в стеке.

Макрос является сокращением для следующего кода:

```
PyObject *v = (*--stack_pointer);
```

Теперь указатель на `obj` хранится как `v`. Указатель списка загружается из `PEEK(oparg)`.

Затем вызывается функция С API для списков Python для `list` и `v`. Код находится в файле `Objects` ► `listobject.c`. Он будет рассматриваться в главе «Объекты и типы».

Далее происходит вызов `PREDICT`, который прогнозирует, что следующей операцией будет `JUMP_ABSOLUTE`. Макрос `PREDICT` содержит генерированные компилятором команды `goto` для секций `case` всех потенциальных операций.

Это означает, что процессор может перейти к этой инструкции без повторного прохождения цикла:

```
...
    case TARGET(LIST_APPEND): {
        PyObject *v = POP();
        PyObject *list = PEEK(oparg);
        int err;
        err = PyList_Append(list, v);
        Py_DECREF(v);
        if (err != 0)
            goto error;
        PREDICT(JUMP_ABSOLUTE);
        DISPATCH();
    }
...
```

Некоторые операции, такие как `CALL_FUNCTION` и `CALL_METHOD`, содержат аргумент операции, который ссылается на другую скомпилированную функцию. В этом случае другой кадр помещается в стек кадров в потоке, и цикл вычисления выполняется для этой функции до ее завершения.

ПРИМЕЧАНИЕ

Некоторые коды операций образуют пары, что позволяет прогнозировать второй код при выполнении первого. Например, заCOMPARE_OP часто следуетPOP_JUMP_IF_FALSEилиPOP_JUMP_IF_TRUE.

Если вы ведете статистику кодов операций, у вас два варианта:

1. Оставить прогнозирование включенным и интерпретировать результаты так, словно некоторые коды операций были объединены.
2. Отключить прогнозирование, чтобы счетчик частоты кодов операций обновлялся для обоих кодов.

Прогнозирование кодов операций отключается в потоковом коде, так как последний позволяет ЦП регистрировать предсказания ветвлений раздельно для каждого кода операции.

Каждый раз, когда новый кадр создается и заносится в стек, в поле `f_back` кадра устанавливается текущий кадр, прежде чем будет создан новый. Такое вложение кадров становится очевидным, если посмотреть на трассировку стека:

cpython-book-samples › 31 › example_stack.py

```
def function2():
    raise RuntimeError

def function1():
    function2()

if __name__ == "__main__":
    function1()
```

При выполнении этого кода в командной строке вы получите следующий результат:

```
$ ./python example_stack.py
```

```
Traceback (most recent call last):
  File "example_stack.py", line 8, in <module>
    function1()
  File "example_stack.py", line 5, in function1
    function2()
  File "example_stack.py", line 2, in function2
    raise RuntimeError
RuntimeError
```

В файле `Lib\traceback.py` можно использовать функцию `walk_stack()` для получения трассировки:

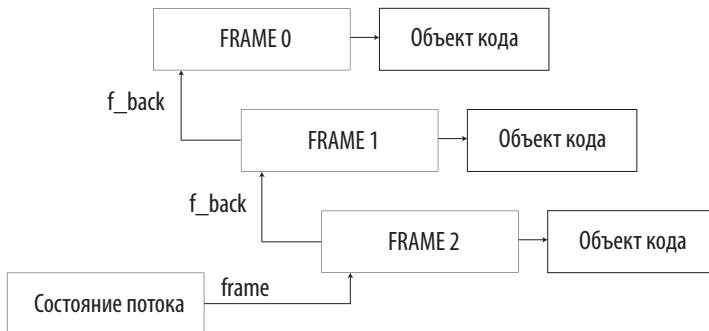
```
def walk_stack(f):
    """Обход стека с получением кадра и номера строки для каждого кадра.

    Переход осуществляется по ссылке f.f_back из заданного кадра.
    Если кадр не задан, используется текущий стек.
    Обычно используется с StackSummary.extract
    """
    if f is None:
        f = sys._getframe().f_back.f_back
    while f is not None:
        yield f, f.f_lineno
        f = f.f_back
```

В качестве кадра выбирается родитель родителя (`sys._getframe().f_back.f_back`), потому что в трассировке не должны присутствовать вызовы `walk_stack()` или `print_trace()`. Указатель `f_back` отслеживается до вершины стека вызовов.

`sys._getframe()` — функция Python API для получения атрибута `frame` текущего потока.

А вот как будет выглядеть стек с тремя кадрами, каждый из которых связан со своим объектом кода и состоянием потока, ссылающимся на текущий кадр:



ВЫВОДЫ

В этой главе вы познакомились с «мозгом» CPython. Основной цикл вычисления обеспечивает взаимодействие между скомпилированным кодом

Python и базовыми модулями расширения C, библиотеками и системными вызовами.

Некоторые темы этой главы были изложены предельно кратко, так как мы вернемся к ним далее в книге. Например, у интерпретатора CPython есть основной цикл вычисления, но при этом могут работать сразу несколько циклов — параллельно или конкурентно.

В CPython может работать множество циклов вычисления, выполняющих несколько кадров в системе. В одной из следующих глав «Параллелизм и конкурентность» вы увидите, как система стека кадров используется CPython для выполнения на разных ядрах процессора. Кроме того, API объекта кадра CPython позволяет приостанавливать и возобновлять выполнение кадров в форме асинхронного программирования.

Загрузка переменных с использованием стека значений требует выделения памяти и управления ею. Чтобы CPython мог работать эффективно, он должен иметь надежный механизм управления памятью. В следующей главе рассматривается процесс управления памятью и его связь с указателями `PyObject`, используемыми циклом вычисления.

Управление памятью

Два важнейших компонента вашего компьютера — память и процессор. Один компонент не может работать без другого. Они должны использоваться правильно и эффективно.

При разработке языка программирования его создатели решают, как пользователь будет управлять памятью компьютера. Это зависит от множества факторов: насколько простым должен быть интерфейс в представлении создателей, должен ли язык быть кроссплатформенным и ставят ли его создатели производительность выше стабильности.

Авторы Python приняли эти решения за вас, но оставили некоторые дополнительные варианты, которые вы сможете выбрать самостоятельно.

В этой главе вы узнаете, как в С организовано управление памятью, так как СPython написан на С. Мы рассмотрим два важнейших аспекта управления памятью в Python:

1. Подсчет ссылок.
2. Сборка мусора.

К концу этой главы вы поймете, как СPython выделяет память в операционной системе, как выделяется и освобождается память для объектов и как СPython разбирается с утечкой памяти.

ВЫДЕЛЕНИЕ ПАМЯТИ В С

Чтобы переменные можно было использовать в С, сначала необходимо получить для них память от операционной системы. В С существуют три механизма выделения памяти:

1. **Статическое выделение памяти:** требования к памяти вычисляются во время компиляции, а память выделяется исполняемым файлом при запуске.
2. **Автоматическое выделение памяти:** память для области видимости выделяется из стека вызовов при входе в кадр и освобождается при завершении кадра.
3. **Динамическое выделение памяти:** память запрашивается и выделяется динамически во время выполнения через вызов API выделения памяти.

Статическое выделение памяти в С

Типы в языке С имеют фиксированный размер. Компилятор вычисляет требования к памяти для всех статических и глобальных переменных, а затем компилирует эти требования в приложение:

```
static int number = 0;
```

Для проверки размера типа в С используется функция `sizeof()`. В моей системе — 64-разрядной macOS с GCC — размер `int` составляет 4 байта. Базовые типы в С могут иметь разные размеры в зависимости от архитектуры и компилятора.

Массивы определяются статически. Возьмем массив из 10 целых чисел:

```
static int numbers[10] = {0,1,2,3,4,5,6,7,8,9};
```

Компилятор С преобразует эту команду в выделение `sizeof(int) * 10` байт памяти.

Компилятор С использует системные функции для выделения памяти. Речь идет о системных сервисах, зависящих от операционной системы; это низкоуровневые функции ядра, выделяющие страницы системной памяти.

Автоматическое выделение памяти в С

По аналогии со статическим выделением памяти, механизм автоматического выделения памяти вычисляет требования к памяти во время компиляции.

Приложение из следующего примера преобразует 100 градусов по Фаренгейту к значению по шкале Цельсия:

cpython-book-samples ▶ 32 ▶ automatic.c

```
#include <stdio.h>

static const double five_ninths = 5.0/9.0;

double celsius(double fahrenheit) {
    double c = (fahrenheit - 32) * five_ninths;
    return c;
}

int main() {
    double f = 100;
    printf("%f F is %f C\n", f, celsius(f));
    return 0;
}
```

В этом примере используется как статическое, так и автоматическое выделение памяти:

- Память для константы `five_ninths` выделяется статически, потому что она объявлена с ключевым словом `static`.
- Память для переменной `c` в `celsius()` выделяется автоматически при вызове `celsius()` и освобождается при завершении `celsius()`.
- Память для переменной `f` в `main()` выделяется автоматически при вызове `main()` и освобождается при завершении `main()`.
- Память для результата `celsius(f)` неявно выделяется автоматически.
- Память, автоматически выделяемая для `main()`, освобождается при завершении функции.

Динамическое выделение памяти в С

В многих случаях как статического, так и автоматического выделения памяти оказывается недостаточно. Например, в некоторых случаях программа не может вычислить требования к памяти во время компиляции, потому что они определяются пользовательским вводом.

В таких случаях память выделяется **динамически**. Динамическое выделение памяти основано на вызове API выделения памяти языка С. Операционные системы резервируют часть системной памяти для динамического выделения процессам. Эта часть памяти называется **кучей** (heap).

В следующем примере память выделяется динамически для массива со значениями температуры по Фаренгейту и Цельсию. Приложение вычисляет значения по шкале Цельсия, соответствующие заданным пользователем значениям по Фаренгейту:

cpython-book-samples › 32 › dynamic.c

```
#include <stdio.h>
#include <stdlib.h>

static const double five_ninths = 5.0/9.0;

double celsius(double fahrenheit) {
    double c = (fahrenheit - 32) * five_ninths;
    return c;
}

int main(int argc, char** argv) {
    if (argc != 2)
        return -1;
    int number = atoi(argv[1]);
    double* f_values = (double*)calloc(number, sizeof(double));
    double* c_values = (double*)calloc(number, sizeof(double));
    for (int i = 0 ; i < number ; i++) {
        f_values[i] = (i + 10) * 10.0 ;
        c_values[i] = celsius((double)f_values[i]);
    }
    for (int i = 0 ; i < number ; i++) {
        printf("%f F is %f C\n", f_values[i], c_values[i]);
    }
    free(c_values);
    free(f_values);

    return 0;
}
```

Если выполнить эту программу с аргументом 4, она выведет следующие значения:

```
100.000000 F is 37.777778 C
110.000000 F is 43.333334 C
120.000000 F is 48.888888 C
130.000000 F is 54.444444 C
```

Пример использует механизм динамического выделения блока памяти из кучи. Затем, когда надобность в блоке памяти отпадет, он возвращается в кучу. Если динамически выделенная память не будет освобождена, произойдет **утечка памяти**.

ПРОЕКТИРОВАНИЕ СИСТЕМЫ УПРАВЛЕНИЯ ПАМЯТЬЮ PYTHON

Система CPython была построена на базе С, поэтому ей приходится использовать ограничения статического, динамического и автоматического выделения памяти. Из-за некоторых особенностей проектирования языка Python эти ограничения создают еще больше проблем:

1. Python является языком с динамической типизацией. Размер переменных не может быть вычислен во время компиляции.
2. Размеры большинства базовых типов Python определяются динамически. Тип `list` может иметь произвольный размер, `dict` может содержать любое число ключей, и даже `int` является динамическим. Пользователю никогда не приходится задавать размер этих типов.
3. Имена в Python могут повторно использоваться для значений разных типов:

```
>>> a_value = 1
>>> a_value = "Now I'm a string"
>>> a_value = ["Now" , "I'm", "a", "list"]
```

Чтобы преодолеть эти ограничения, CPython в значительной мере полагается на динамическое выделение памяти, но добавляет страховку для автоматизации ее освобождения, используя алгоритмы сборки мусора и подсчета ссылок.

Вместо того чтобы заставлять Python-разработчика заниматься выделением памяти, память объектов в Python выделяется автоматически через один унифицированный API. Архитектура требует, чтобы вся стандартная библиотека CPython и базовые модули (написанные на С) использовали этот API.

Области выделения памяти

CPython поддерживает три области динамического выделения памяти:

1. **Область сырой (raw) памяти** — используется для выделения памяти из системной кучи и больших объемов памяти, а также если она выделяется не для объектов Python.
2. **Область объектной (object) памяти** — используется для выделения памяти для всех объектов Python.

3. **Область PyMem** — то же, что PYMEM_DOMAIN_OBJ. Этот тип существует для обеспечения совместимости со старыми API.

Все эти типы реализуют один и тот же интерфейс функций:

- `_Alloc(size_t size)` выделяет память размером `bytes` и возвращает указатель.
- `_Calloc(size_t nelem, size_t elsize)` выделяет память для `nelem` элементов, каждый из которых имеет размер `elsize`, и возвращает указатель.
- `_Realloc(void *ptr, size_t new_size)` выделяет память размером `new_size`.
- `_Free(void *ptr)` освобождает память по указателю `ptr` и возвращает ее в кучу.

`PyMemAllocatorDomain` представляет три области выделения памяти в CPython именами PYMEM_DOMAIN_RAW, PYMEM_DOMAIN_OBJ и PYMEM_DOMAIN_MEM.

Аллокаторы

CPython использует два аллокатора¹:

1. `malloc`: аллокатор операционной системы для выделения сырой памяти.
2. `pymalloc`: аллокатор CPython для выделения объектной памяти и PyMem.

ПРИМЕЧАНИЕ

В CPython по умолчанию компилируется аллокатор CPython, `pymalloc`. Его можно удалить, для этого следует перекомпилировать CPython после установки параметра `WITH_PYMalloc = 0` в `pyconfig.h`. При его удалении API Object и PyMem будут использовать системный аллокатор.

Если вы скомпилировали CPython с поддержкой отладки (`--with-pydebug` в macOS или Linux, или цель `Debug` в Windows), то каждая функция-аллокатор будет входить в отладочную реализацию. Например, с включенной

¹ Механизм распределения памяти. — Примеч. ред.

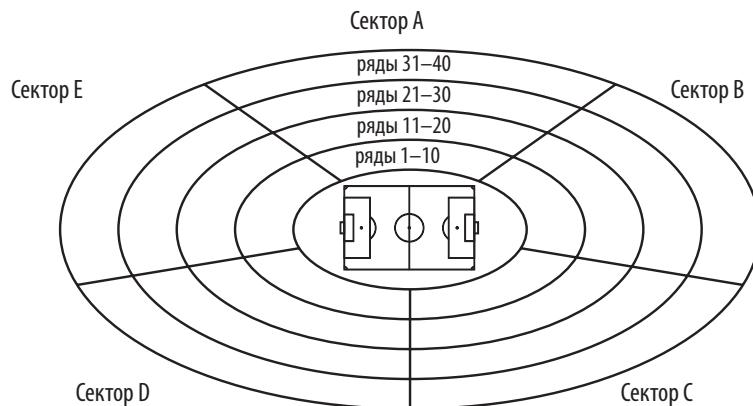
отладкой вызовы выделения памяти будут выполнять `_PyMem_DebugAlloc()` вместо `_PyMem_Alloc()`.

АЛЛОКАТОРЫ ПАМЯТИ СPYTHON

Аллокатор памяти CPython работает поверх системного аллокатора и использует собственный алгоритм для этого. Алгоритм похож на использующийся в системе механизм, но он адаптирован для CPython:

- В большинстве запросов выделения памяти используется небольшой фиксированный размер: для `PyObject` 16 байт, для `PyASCIIObject` 42 байта, для `PyCompactUnicodeObject` 72 байта и для `PyLongObject` 32 байта.
- Аллокатор `rumalloc` выделяет блоки памяти размером не более 256 Кбайт. Все, что больше, передается для распределения системе.
- `rumalloc` использует GIL вместо системной проверки безопасности потока.

Чтобы вам было проще понять этот процесс, представьте стадион. Для управления потоками людей была реализована система, разбивающая стадион на сектора от А до Е, при этом сектора разбиты на ряды. В каждом секторе имеется 40 мест:



В рядах с 1 по 10 располагаются более просторные премиальные места, по 80 мест в каждом ряду. В рядах с 31 по 40 находятся места эконом-класса, по 150 мест в каждом ряду.

Алгоритм выделения памяти Python обладает сходными характеристиками:

- На стадионе зрители имеют места, а алгоритм `pymalloc` имеет **блоки** памяти.
- На стадионе места могут быть премиальными, стандартными и дешевыми; блоки памяти принадлежат к одному из диапазонов фиксированного размера. Вам не удастся принести на стадион свою табуретку!
- Все места одного размера объединяются в ряды, а блоки одного размера — в **пулы**.

Центральный реестр хранит информацию о том, где хранятся блоки, и о количестве доступных блоков в пуле — так же, как на стадионе распределяются места. Когда ряд на стадионе заполняется, используется следующий ряд. Когда пул блоков заполняется, используется следующий пул. Пулы группируются в **арены**, подобно тому как на стадионе ряды группируются в сектора.

Такая стратегия обладает рядом достоинств:

1. Алгоритм обладает большей производительностью для основного сценария использования CPython: небольшие объекты с коротким сроком жизни.
2. Алгоритм использует GIL вместо системного обнаружения блокировок потоков.
3. Алгоритм использует перераспределение памяти (`mmap()`) вместо выделения памяти из кучи.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к аллокатору памяти.

ФАЙЛ	НАЗНАЧЕНИЕ
Include ▶ <code>pymem.h</code>	API аллокатора PyMem
Include ▶ <code>cpython</code> ▶ <code>pymem.h</code>	API конфигурации аллокатора памяти PyMem
Include ▶ <code>internal</code> ▶ <code>pycore_mem.h</code>	Структура данных сборщика мусора и внутренний API
Objects ▶ <code>obmalloc.c</code>	Реализации областей распределения памяти и <code>pymalloc</code>

Важные термины

Ниже перечислены некоторые ключевые термины, встречающиеся в этой главе:

- Запрашиваемая память сопоставляется с размером **блока**.
- Блоки одного размера входят в один **пул** памяти.
- Пулы группируются в **арены**.

Блоки, пулы и арены

Наибольшая группа ячеек памяти называется **ареной** (arena). CPython создает арены размером 256 Кбайт, чтобы они совпадали по размеру с системной страницей. Граница системной страницы — непрерывный участок памяти фиксированной длины.

Даже с современной скоростной памятью непрерывные участки памяти загружаются быстрее, чем фрагментированные, поэтому использовать непрерывную память выгоднее.

Арены

Арены выделяются из системной кучи, а в системах, поддерживающих анонимное распределение памяти, используется функция `mmap()`¹. Распределение памяти помогает сократить уровень фрагментации кучи для аренд.

Наглядное представление четырех аренд в системной куче:



¹ <http://man7.org/linux/man-pages/man2/mmap.2.html>.

Арена имеет структуру данных arenaobject:

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
address	uintptr_t	Адрес аренды в памяти
pool_address	block *	Указатель на следующий пул, из которого будет выделяться память
nfreepools	uint	Количество доступных пулов в арене (свободные + не выделявшиеся пулы)
ntotalpools	uint	Общее количество пулов в арене (доступных и недоступных)
freepools	pool_header*	Односвязный список доступных пулов
nextarena	arena_object*	Следующая арена (см. примечание)
prevarena	arena_object*	Предыдущая арена (см. примечание)

ПРИМЕЧАНИЕ

Арены объединяются в двусвязный список внутри структуры данных арены при помощи указателей nextarena (следующая арена) и prevarena (предыдущая арена).

Если память для арены не выделена, то используется поле nextarena. Оно связывает все несвязанные арены в односвязный глобальный список unused_arena_objects.

Когда арена связывается с ареной, для которой выделена память хотя бы с одним доступным пулом, оба указателя, nextarena и prevarena, используются в двусвязном списке usable Arenas. Этот список поддерживается по возрастанию значений nfreepools.

Пулы

Внутри арены создаются пулы, состоящие из блоков размером до 512 байт. В 32-разрядных системах шаг увеличения размера блока составляет 8 байт, поэтому определены 64 класса размера:

ЗАПРОС В БАЙТАХ	РАЗМЕР ВЫДЕЛЕННОГО БЛОКА	ИНДЕКС КЛАССА РАЗМЕРА
1-8	8	0
9-16	16	1
17-24	24	2
25-32	32	3
...
497-504	504	62
505-512	512	63

В 64-разрядных системах шаг составляет 16 байт, поэтому определены 32 класса размера:

ЗАПРОС В БАЙТАХ	РАЗМЕР ВЫДЕЛЕННОГО БЛОКА	ИНДЕКС КЛАССА РАЗМЕРА
1-16	16	0
17-32	32	1
33-48	48	2
49-64	64	3
...
480-496	496	30
496-512	512	31

Все пулы имеют размер 4096 байт (4 Кбайт), так что арена всегда содержит 64 пула:

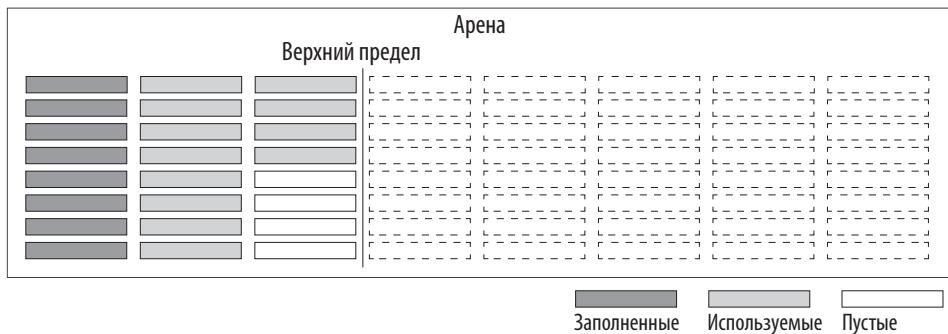


Пулы выделяются в памяти по требованию. Если для запрашиваемого индекса класса размера доступные пулы отсутствуют, предоставляется новый. У арен имеется **верхний предел** для индексирования количества введенных в действие пулов.

Пулы имеют три состояния:

1. **Заполненный**: все доступные блоки в пульке уже выделены.
2. **Используемый**: пул выделен, некоторые блоки заняты, но в пульке все еще осталось место.
3. **Пустой**: пул выделен, но блоки еще не заняты.

Внутри арены верхний предел устанавливается на последний выделенный пул:



Пулы имеют структуру данных `poolp`, которая является статическим распределением структуры `pool_header`. Тип `pool_header` содержит следующие свойства:

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
ref	uint	Текущее количество выделенных блоков в пульке
freeblock	block *	Указатель на начало списка свободных блоков в пульке
nextpool	pool_header*	Указатель на следующий пул текущего класса размера
prevpool	pool_header*	Указатель на предыдущий пул текущего класса размера
arenaindex	uint	Односвязный список доступных пулов

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
szidx	uint	Индекс класса размера текущего пула
nextoffset	uint	Количество байтов до свободного блока
maxnextoffset	uint	Максимальное значение, которое может принять nextoffset до заполнения пула

Каждый пул некоторого класса размера хранит двусвязный список на следующий и предыдущий пул этого класса. Когда происходит операция выделения памяти, список позволяет легко перемещаться между пулами одного класса размера в арене.

Таблицы пулов

Реестр пулов в арене называется **таблицей пулов**. Таблица пулов представляет собой циклический, имеющий начало двусвязный список частично используемых пулов.

Таблица пулов сегментируется по индексу класса размера (*i*). Для индекса *i* `usedpools[i + i]` указывает на начало списка всех частично используемых пулов с таким же индексом класса размера.

Основные характеристики таблицы пулов:

- Когда пул заполняется, он отсоединяется от своего списка `usedpools[]`.
- Если в заполненном пуле освобождается блок, то он возвращается в состояние «используемый». Вновь освобожденный пул вставляется в начало соответствующего списка `usedpools[]`, так что следующее выделение памяти для его класса размера использует освобожденный блок.
- При переходе в состояние «пустой» пул отсоединяется от списка `usedpools[]` и добавляется в начало односвязного списка `freepools` своей арены.

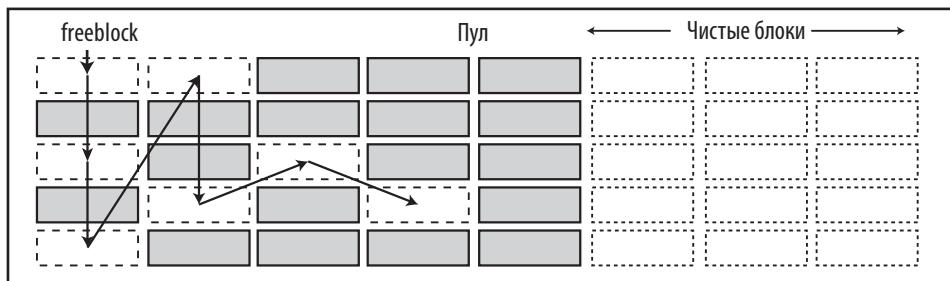
Блоки

Внутри пула память делится на блоки. Блоки обладают следующими характеристиками:

- Внутри пула выделяются и освобождаются блоки фиксированного размера.

- Доступные блоки внутри пула объединяются в односвязный список `freeblock`.
- Когда блок освобождается, он вставляется в начало списка `freeblock`.
- Когда пул инициализируется, только первые два блока связываются в списке `freeblock`.
- Пока пул в состоянии «используемый», в нем есть хотя бы один доступный для выделения блок.

Вот как выглядит частично выделенный пул, содержащий используемые, свободные и доступные блоки:



API выделения блоков

Когда блок памяти запрашивается областью памяти, использующей алlocator `rumalloc`, вызывается функция `rumalloc_alloc()`. В этой функции можно поставить точку останова и выполнить код поэтапно, чтобы проверить ваше понимание блоков, пулов и арен:

Objects > obmalloc.c, строка 1590

```
static inline void*
rumalloc_alloc(void *ctx, size_t nbytes)
{
    ...
}
```

В запросе `nbytes = 30` размер должен быть не равен нулю и не превышать значение `SMALL_REQUEST_THRESHOLD`, равное 512:

```
if (UNLIKELY(nbytes == 0)) {
    return NULL;
}
if (UNLIKELY(nbytes > SMALL_REQUEST_THRESHOLD)) {
    return NULL;
}
```

В 64-разрядных системах при вычислении индекса класса размера будет получен результат 1. Это соответствует второму индексу класса размера (17–32 байта).

Следовательно, целевым пулом будет `usedpools[1 + 1]` (`usedpools[2]`):

```
uint size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT;
poolp pool = usedpools[size + size];
block *bp;
```

Затем проверяется наличие доступного используемого ('used') пула для индекса класса размера. Если список `freeblock` находится в конце пула, значит, в пуле еще остаются чистые блоки.

Функция `pymalloc_pool_extend()` вызывается для расширения списка `freeblock`:

```
if (LIKELY(pool != pool->nextpool)) {
    /*
     * Для этого класса размера имеется используемый пул.
     * Получить первый блок его свободного списка
     */
    ++pool->ref.count;
    bp = pool->freeblock;
    assert(bp != NULL);

    if (UNLIKELY((pool->freeblock = *(block ***)bp) == NULL)) {
        // Достигнут конец свободного списка. Попробовать расширить его
        pymalloc_pool_extend(pool, size);
    }
}
```

Если доступные пулы отсутствуют, то создается новый пул и возвращается первый блок. `allocate_from_new_pool()` автоматически добавляет новый пул в список `usedpools`:

```
else {
    /*
     * Нет пула нужного класса размера.
     * Использовать свободный пул
     */
    bp = allocate_from_new_pool(size);
}

return (void *)bp;
}
```

Наконец, возвращается адрес нового блока.

Использование отладочного API Python

Модуль `sys` содержит внутреннюю функцию `_debugmallocstats()` для получения количества используемых блоков для пулов каждого класса размеров. Он также выводит количество выделенных и освобожденных арен с общим количеством используемых блоков.

При помощи этой функции можно просмотреть информацию об использовании памяти во время работы:

```
$ ./python -c "import sys; sys._debugmallocstats()"

Small block threshold = 512, in 32 size classes.

class    size    num pools   blocks in use   avail blocks
-----  -----  -----
 0        16          1            181            72
 1        32          6            675            81
 2        48         18           1441            71
...
2 free 18-sized PyTupleObjects * 168 bytes each =      336
3 free 19-sized PyTupleObjects * 176 bytes each =      528
```

В выходных данных отображается таблица классов размеров, выделенная память и дополнительная статистика.

ОБЛАСТЬ ВЫДЕЛЕНИЯ ОБЪЕКТНОЙ ПАМЯТИ И РУМЕМ

Объектный аллокатор CPython — первый из трех механизмов, которые мы рассмотрим. Задача объектного аллокатора памяти — выделение памяти, относящейся к объектам Python (например, заголовки новых объектов и данные об объектах, такие как ключи и значения словарей или элементы списка).

Аллокатор также используется для компилятора, AST, парсера и цикла вычисления. Отличным примером использования объектного аллокатора памяти служит конструктор типа `PyLongObject` (`int`), `PyLong_New()`:

- При конструировании нового значения типа `int` память выделяется объектным аллокатором.
- Размер запрашиваемой памяти равен сумме размера структуры `PyLongObject` и объема памяти, необходимого для хранения цифр.

Тип `long` в Python не эквивалентен типу `long` языка С. Они представляют собой *список* цифр. Число 12378562834 в Python будет представлено в виде списка цифр [1, 2, 3, 7, 8, 5, 6, 2, 8, 3, 4]. Именно эта структура памяти позволяет Python работать с очень большими числами, не беспокоясь об ограничениях 32- или 64-разрядных целых чисел.

Чтобы увидеть пример выделения памяти, возьмем конструктор `PyLong`:

```
PyLongObject *
_PyLong_New(Py_ssize_t size)
{
    PyLongObject *result;
    ...
    if (size > (Py_ssize_t)MAX_LONG_DIGITS) {
        PyErr_SetString(PyExc_OverflowError,
                       "too many digits in integer");
        return NULL;
    }
    result = PyObject_MALLOC(offsetof(PyLongObject, ob_digit) +
                           size*sizeof(digit));
    if (!result) {
        PyErr_NoMemory();
        return NULL;
    }
    return (PyLongObject*)PyObject_INIT_VAR(result, &PyLong_Type, size);
}
```

Если вызвать `_PyLong_New(2)`, функция вычислит значение `size_t` следующим образом:

ЗНАЧЕНИЕ	БАЙТЫ
<code>sizeof(digit)</code>	4
<code>size</code>	2
<code>header offset</code>	26
Итого	32

При вызове `PyObject_MALLOC()` будет использоваться значение `size_t`, равное 32.

В моей системе максимальное количество цифр в типе `long`, `MAX_LONG_DIGITS`, равно 2305843009213693945 (очень, очень большое число). Если выполнить

_PyLong_New(2305843009213693945), он вызовет PyObject_MALLOC() с size_t, равным 9223372036854775804 байта, или 8 589 934 592 Гбайт (что превышает объем оперативной памяти на моем компьютере).

Использование модуля tracemalloc

Модуль `tracemalloc` стандартной библиотеки может использоваться для отладки выделения памяти через объектный аллокатор. Он предоставляет информацию о том, где была выделена память для объекта, и количество выделенных блоков. Как отладочный инструмент, `tracemalloc` поможет вычислить объем памяти, затраченный при выполнении вашего кода, и обнаружить утечку памяти.

Чтобы включить трассировку памяти, можно запустить Python с `-X tracemalloc=1`, где 1 — глубина отслеживания (в кадрах). Также можно включить трассировку памяти при помощи переменной среды `PYTHONTRACEMALLOC=1`. Чтобы указать, на сколько кадров в глубину будет осуществляться трассировка, замените 1 любым целым числом.

Используйте `take_snapshot()` для создания экземпляра снимка, а затем сравните снимки вызовом `compare_to()`. Чтобы увидеть это в действии, создайте файл `tracedemo.py`:

cpython-book-samples › 32 › tracedemo.py

```
import tracemalloc

tracemalloc.start()

def to_celsius(fahrenheit, /, options=None):
    return (fahrenheit-32)*5/9

values = range(0, 100, 10) # values 0, 10, 20, ... 90

for v in values:
    c = to_celsius(v)

after = tracemalloc.take_snapshot()

tracemalloc.stop()
after = after.filter_traces([tracemalloc.Filter(True, '**/tracedemo.py')])
stats = after.statistics('lineno')

for stat in stats:
    print(stat)
```

При выполнении будет выведен список значений памяти, используемой для каждой строки программы (по убыванию):

```
$ ./python -X tracemalloc=2 tracedemo.py  
/Users/.../tracedemo.py:5: size=712 B, count=2, average=356 B  
/Users/.../tracedemo.py:13: size=512 B, count=1, average=512 B  
/Users/.../tracedemo.py:11: size=480 B, count=1, average=480 B  
/Users/.../tracedemo.py:8: size=112 B, count=2, average=56 B  
/Users/.../tracedemo.py:6: size=24 B, count=1, average=24 B
```

Строка с наибольшим потреблением памяти была строка `return (fahrenheit-32)*5/9`, в которой выполняется фактическое вычисление.

ОБЛАСТЬ ВЫДЕЛЕНИЯ СЫРОЙ ПАМЯТИ

Область выделения сырой памяти используется либо напрямую, либо при вызове двух других областей с запросом памяти, превышающим 512 Кбайт. Она получает размер запрашиваемой памяти в байтах и вызывает `malloc(size)`. Если аргумент размера равен 0, то некоторые системы возвращают `NULL` для `malloc(0)`, что будет рассматриваться как ошибка. Некоторые платформы возвращают указатель, который не ссылается на выделенную память, — это приведет к нарушению работы `rumalloc`.

Для решения проблем такого рода `_PyMem_RawMalloc()` добавляет лишний байт перед вызовом `malloc()`.

ПРИМЕЧАНИЕ

По умолчанию аллокаторы памяти PyMem используют объектные аллокаторы. `PyMem_Malloc()` и `PyObject_Malloc()` следуют по одной ветви исполнения.

НЕСТАНДАРТНЫЕ ОБЛАСТИ ВЫДЕЛЕНИЯ ПАМЯТИ

CPython также позволяет переопределить реализацию выделения памяти для любой из трех областей памяти. Если ваша системная среда требует специальных проверок памяти или алгоритмов ее выделения, можно подключить новый набор функций выделения памяти в среде выполнения.

PyMemAllocatorEx представляет собой `typedef struct` с полями для всех методов, которые необходимо реализовать для переопределения аллокатора:

```
typedef struct {
    /* Пользовательский контекст передается в качестве первого аргумента
       четырем функциям */
    void *ctx;

    /* Выделение блока памяти */
    void* (*malloc) (void *ctx, size_t size);

    /* Выделение блока памяти, инициализированного нулями */
    void* (*calloc) (void *ctx, size_t nelem, size_t elsize);

    /* Выделение блока памяти или изменение размера */
    void* (*realloc) (void *ctx, void *ptr, size_t new_size);

    /* Освобождение блока памяти */
    void (*free) (void *ctx, void *ptr);
} PyMemAllocatorEx;
```

Для получения существующей реализации можно воспользоваться API методом `PyMem_GetAllocator()`:

```
PyMemAllocatorEx * existing_obj;
PyMem_GetAllocator(PYMEM_DOMAIN_OBJ, existing_obj);
```

ВАЖНО

Несколько важных тестовых критериев для нестандартных аллокаторов памяти:

- Новый аллокатор должен возвращать осмысленный указатель, отличный от `NULL`, при запросе нуля байтов.
- Для области памяти `PYMEM_DOMAIN_RAW` аллокатор должен быть потокобезопасным.

Реализовав функции `My_Malloc()`, `My_Calloc()`, `My_Realloc()` и `My_Free()` по сигнатурам из `PyMemAllocatorEx`, вы сможете переопределить аллокатор для любой области памяти, например для `PYMEM_DOMAIN_OBJ`:

```
PyMemAllocatorEx my_allocators =
    {NULL, My_Malloc, My_Calloc, My_Realloc, My_Free};
PyMem_SetAllocator(PYMEM_DOMAIN_OBJ, &my_allocators);
```

САНИТАЙЗЕРЫ ВЫДЕЛЕННОЙ ПАМЯТИ

Санитайзеры выделенной памяти (memory allocation sanitizers) — дополнительные алгоритмы, находящиеся между системным вызовом для выделения памяти и функцией ядра для распределения памяти в системе. Они используются в средах, требующих особых ограничений для стабильной работы, высокой безопасности или отладки ошибок выделения памяти.

CPython может компилироваться с различными средствами устранения ошибок памяти, которые являются частью библиотек компилятора. Как правило, они значительно замедляют работу CPython, не могут комбинироваться друг с другом и чаще всего применяются в ситуациях отладки или в системах, в которых крайне важно предотвратить некорректные обращения к памяти.

AddressSanitizer

AddressSanitizer быстро обнаруживает ошибки, связанные с работой с памятью на стадии выполнения:

- Выход за пределы границ кучи, стека и глобальных переменных.
- Использование памяти после ее освобождения.
- Двойное освобождение и некорректное освобождение.

AddressSanitizer включается следующей командой:

```
$ ./configure --with-address-sanitizer ...
```

ВАЖНО

Использование AddressSanitizer может замедлить приложения вдвое и увеличить затраты памяти до трех раз.

AddressSanitizer поддерживается в следующих операционных системах:

- Linux
- macOS
- NetBSD
- FreeBSD

За дополнительной информацией обращайтесь к официальной документации¹.

MemorySanitizer

MemorySanitizer обнаруживает попытки чтения из неинициализированной памяти. Если программа обращается к адресному пространству до того, как оно было инициализировано (при выделении), процесс останавливается, прежде чем содержимое памяти будет прочитано.

MemorySanitizer включается следующей командой:

```
$ ./configure --with-memory-sanitizer ...
```

ВАЖНО

Использование MemorySanitizer может замедлить приложения вдвое и увеличить затраты памяти до двух раз.

MemorySanitizer поддерживается в следующих операционных системах:

- Linux
- NetBSD
- FreeBSD

За дополнительной информацией обращайтесь к официальной документации².

UndefinedBehaviorSanitizer

UndefinedBehaviorSanitizer (UBSan) быстро обнаруживает ситуации неопределенного поведения в ходе выполнения:

- Неправильно выровненный или неопределенный (нулевой) указатель.
- Переполнение целочисленного типа со знаком.

¹ <https://clang.llvm.org/docs/AddressSanitizer.html>.

² <https://clang.llvm.org/docs/MemorySanitizer.html>.

- Преобразование к типу с плавающей точкой, из него или между значениями этого типа.

UBSan включается следующей командой:

```
$ ./configure --with-undefined-behavior-sanitizer ...
```

UBSan поддерживается в следующих операционных системах:

- Linux
- macOS
- NetBSD
- FreeBSD

За дополнительной информацией обращайтесь к официальной документации¹.

UBSan поддерживает множество параметров конфигурации. Параметр `--with-undefined-behavior-sanitizer` выбирает профиль `undefined`. Чтобы использовать другой профиль (например, `nullability`), выполните `./configure` с переменной `CFLAGS`:

```
$ ./configure CFLAGS="-fsanitize=nullability" \
LDFLAGS="-fsanitize=nullability"
```

После перекомпиляции CPython эта конфигурация создаст двоичный файл CPython с использованием `UndefinedBehaviorSanitizer`.

АРЕНА ПАМЯТИ PYARENA

В книге неоднократно упоминается объект `PyArena`. Это отдельный API выделения арен, используемый компилятором, системой вычисления кадров и другими частями системы, которые не запускаются из API распределения памяти для объектов Python.

`PyArena` также поддерживает собственный список созданных объектов в структуре данных арены. Память, выделенная `PyArena`, не обрабатывается сборщиком мусора.

¹ <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>.

Когда экземпляр PyArena выделяет память, он сохраняет текущее количество выделенных блоков, а затем вызывает `PyMem_Alloc`. Запросы на выделение памяти к PyArena используют объектный аллокатор для блоков размером до 512 Кбайт (включительно) и низкоуровневый аллокатор для больших блоков.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к PyArena.

ФАЙЛ	НАЗНАЧЕНИЕ
Include ▶ pyarena.h	API PyArena и определения типов
Python ▶ pyarena.c	Реализация PyArena

ПОДСЧЕТ ССЫЛОК

Как упоминалось в этой главе, CPython использует систему динамического выделения памяти С. Требования к памяти определяются во время выполнения, а память выделяется в системе средствами `PyMem` API.

Для Python-разработчиков эта система была абстрагирована и упрощена. Им не придется беспокоиться о выделении и освобождении памяти.

Для простоты управления выделенной памятью Python применяет две стратегии:

1. Подсчет ссылок.
2. Сборка мусора.

Эти две стратегии более подробно рассматриваются ниже.

Создание переменных в Python

Чтобы создать переменную в Python, необходимо присвоить значение переменной с *的独特им именем*:

```
my_variable = ["a", "b", "c"]
```

Когда переменной присваивается значение, Python проверяет, существует ли имя этой переменной в области видимости локальных и глобальных имен.

В приведенном примере `my_variable` отсутствует в словарях `locals()` или `globals()`. Создается новый объект типа `list`, и указатель сохраняется в словаре `locals()`.

Теперь есть одна ссылка на `my_variable`. Память объекта списка не должна освобождаться, пока на нее существуют действительные ссылки. Если ее память будет освобождена, то указатель `my_variable` будет ссылаться на недействительную область памяти, и в CPython произойдет сбой.

В исходном коде CPython на языке С встречаются вызовы `Py_INCREF()` и `Py_DECREF()`. Эти макросы образуют основной API для увеличения и уменьшения значения счетчика ссылок на объекты Python. Каждый раз, когда появляется ссылка на объект, счетчик увеличивается. Когда ссылка на объект пропадает, счетчик уменьшается.

Если счетчик ссылок достигает нуля, предполагается, что память больше не нужна, и она автоматически освобождается.

Увеличение счетчика ссылок

Каждый экземпляр `PyObject` содержит свойство `ob_refcnt`. В этом свойстве хранится счетчик ссылок на данный объект.

Счетчики ссылок на объект увеличиваются во многих ситуациях. В кодовой базе CPython присутствует более 3000 вызовов `Py_INCREF()`. Чаще всего вызовы происходят тогда, когда объект:

- присваивается имени переменной;
- используется как аргумент функции или метода;
- возвращается из функции через `return` или `yield`.

Логика макроса `Py_INCREF` состоит из одного шага: значение `ob_refcnt` увеличивается на 1:

```
static inline void _Py_INCREF(PyObject *op)
{
    _Py_INC_REFTOTAL;
    op->ob_refcnt++;
}
```

Если CPython скомпилирован в отладочном режиме, то `_Py_INC_REFTOTAL` будет увеличивать глобальный счетчик ссылок `_Py_RefTotal`.

ПРИМЕЧАНИЕ

Чтобы увидеть глобальный счетчик ссылок, добавьте флаг -X showrefcount при запуске отладочной сборки CPython:

```
$ ./python -X showrefcount -c "x=1; x+=1; print(f'x is {x}')"
x is 2
[18497 refs, 6470 blocks]
```

Первое число в квадратных скобках – количество ссылок, созданных в процессе, а второе – количество выделенных блоков.

Уменьшение счетчика ссылок

Счетчики ссылок на объект уменьшаются при выходе переменной за пределы области видимости, в которой она была объявлена. Областью видимости в Python могут быть функция или метод, включение (comprehension) или лямбда-выражение. Некоторые области видимости выражаются явно, но также существует много неявных областей видимости, например при передаче переменной вызываемой функции.

Макрос `Py_DECREF()` сложнее `Py_INCREF()`, потому что он должен обрабатывать событие, когда счетчик ссылок дошел до 0 и нужно освободить память:

```
static inline void _Py_DECREF(
    #ifdef Py_REF_DEBUG
        const char *filename, int lineno,
    #endif
        PyObject *op)
{
    _Py_DEC_REFTOTAL;
    if (--op->ob_refcnt != 0) {
    #ifdef Py_REF_DEBUG
        if (op->ob_refcnt < 0) {
            _Py_NegativeRefCount(filename, lineno, op);
        }
    #endif
        }
    else {
        _Py_Dealloc(op);
    }
}
```

Внутри `Py_DECREF()` при обнулении счетчика ссылок (`ob_refcnt`) вызывается деструктор объекта с помощью `_Py_Dealloc(op)`, и вся выделенная память освобождается. Как и в случае с `Py_INCREF()`, существуют дополнительные функции при компиляции CPython в отладочном режиме.

Для операции увеличения должна существовать эквивалентная операция уменьшения. Если счетчик ссылок становится отрицательным, это указывает на разбалансировку кода С. При попытке уменьшения ссылок на объект, на который не осталось ни одной ссылки, выдается сообщение об ошибке:

```
<file>:<line>: _Py_NegativeRefCount: Assertion failed:  
    object has negative ref count  
Enable tracemalloc to get the memory block allocation traceback  
  
object address  : 0x109eaac50  
object refcount : -1  
object type    : 0x109cadf60  
object type name: <type>  
object repr    : <refcnt -1 at 0x109eaac50>
```

При внесении изменений в поведение отдельных операций языка Python или компилятора необходимо тщательно учитывать последствия таких изменений для счетчиков ссылок.

Подсчет ссылок в операциях байт-кода

Большая часть подсчета ссылок в Python происходит в операциях байт-кода в Python ► `ceval.c`.

Посчитайте ссылки на переменную `y` в этом примере:

```
y = "hello"  
  
def greet(message=y):  
    print(message.capitalize() + " " + y)  
  
messages = [y]  
  
greet(*messages)
```

На первый взгляд здесь существуют четыре ссылки на `y`:

1. Переменная в области видимости верхнего уровня.
2. Значение по умолчанию для именного аргумента `message`.

3. Внутри `greet()`.
4. Элемент списка `messages`.

Выполните этот код со следующим дополнительным фрагментом:

```
import sys
print(sys.getrefcount(y))
```

Всего существует шесть ссылок на `y`.

Вместо размещения в центральной функции, которая должна обрабатывать эти (и другие) случаи, логика увеличения и уменьшения счетчиков ссылок разбивается на несколько небольших частей.

Операция байт-кода должна иметь определяющее влияние на счетчики ссылок на объект, которые она получает как аргументы.

Например, в цикле вычисления кадра операция `LOAD_FAST` загружает объект с заданным именем и помещает его на вершину стека значений. После того как имя переменной, предоставленное в `oparg`, будет обработано `GETLOCAL()`, счетчик ссылок увеличится:

```
...
case TARGET(LOAD_FAST): {
    PyObject *value = GETLOCAL(oparg);
    if (value == NULL) {
        format_exc_check_arg(tstate, PyExc_UnboundLocalError,
                             UNBOUNDLOCAL_ERROR_MSG,
                             PyTuple_GetItem(co->co_varnames, oparg));
        goto error;
    }
    Py_INCREF(value);
    PUSH(value);
    FAST_DISPATCH();
}
```

Операция `LOAD_FAST` компилируется многими узлами AST с операциями.

Допустим, вы присваиваете значения двум переменным, `a` и `b`, а затем создаете третью переменную, `c`, которая равна произведению `a` и `b`:

```
a = 10
b = 20
c = a * b
```

В третьей строке (`c = a * b`) выражение справа (`a * b`) будет разбито на три операции:

1. `LOAD_FAST` обрабатывает переменную `a`, помещает ее в стек значений и увеличивает счетчик ссылок на `a` на 1.
2. `LOAD_FAST` обрабатывает переменную `b`, помещает ее в стек значений и увеличивает счетчик ссылок на `b` на 1.
3. `BINARY_MULTIPLY` умножает переменные в левой и правой части, после чего помещает результат в стек значений.

Бинарный оператор умножения `BINARY_MULTIPLY` знает, что ссылки на левую и правую переменную были загружены в первую и вторую позиции стека значений и что метод `LOAD_FAST` увеличил счетчики ссылок.

Внутри операции `BINARY_MULTIPLY` счетчики ссылок на `a` (левый operand) и `b` (правый operand) уменьшаются после вычисления результата:

```
case TARGET(BINARY_MULTIPLY): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *res = PyNumber_Multiply(left, right);
    Py_DECREF(left);
    Py_DECREF(right);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

У полученного числа `res` счетчик ссылок будет равен 1, когда оно будет помещено на вершину стека значений.

Преимущества подсчета ссылок в CPython

Подсчет ссылок в CPython прост, работает быстро и эффективно. Наиболее серьезный недостаток заключается в том, что он должен учитывать эффект каждой операции (и тщательно выдерживать баланс производительности).

Как вы уже видели, операция байт-кода увеличивает счетчик; предполагается, что эквивалентная операция соответствующим образом уменьшит его. А что произойдет при возникновении непредвиденной ошибки? Были ли проверены все возможные сценарии?

Все, о чем говорилось до настоящего момента, лежит в области среды выполнения CPython. Python-разработчик практически не может управлять этим поведением. У механизма подсчета ссылок также имеется серьезный недостаток: **циклические ссылки**.

Рассмотрим следующий пример Python:

```
x = []
x.append(x)
del x
```

Счетчик ссылок для `x` остается равным 1, потому что `x` ссылается на себя.

Для преодоления этих затруднений и устранения подобных утечек памяти в CPython существует второй механизм управления памятью, называемый **сборкой мусора**.

СБОРКА МУСОРА

Насколько часто у вас вывозят мусор? Каждую неделю, каждый день? Когда вы перестаете чем-то пользоваться и предмет становится ненужным, вы выбрасываете его. Но мусор не вывозится немедленно. Приходится ждать, когда приедет мусоровоз и заберет его.

CPython использует аналогичный принцип в алгоритме сборки мусора. Сборщик мусора CPython старается освободить память, которая используется для несуществующих объектов. Он включен по умолчанию и работает в фоновом режиме. Так как алгоритм сборки мусора немного сложнее подсчета ссылок, он не работает постоянно — так он расходовал бы слишком много ресурсов процессора. Сборка мусора выполняется периодически после заданного количества операций.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к сборщику мусора:

ФАЙЛ	НАЗНАЧЕНИЕ
Modules ▶ gcmodule.c	Модуль сборки мусора и реализация алгоритма
Include ▶ internal ▶ pycore_mem.h	Структура данных сборки мусора и внутренние API

Архитектура сборщика мусора

Как вы узнали в предыдущем разделе, каждый объект Python хранит количество ссылок на него. Когда счетчик достигает нуля, объект перестает существовать и память освобождается.

Многие объекты Python контейнерного типа — списки, кортежи, словари, множества — могут привести к появлению циклических ссылок. Счетчики ссылок сами по себе не могут гарантировать освобождение памяти, занимаемой неиспользуемыми объектами.

Хотя появления циклических ссылок в контейнерах следует избегать, примеры такого рода часто встречаются в стандартной библиотеке и основном интерпретаторе. Еще один распространенный пример, в котором контейнерный тип (`class`) может ссылаться на себя:

cpython-book-samples ▶ 32 ▶ user.py

```
__all__ = ["User"]

class User(BaseUser):
    name: 'str' = ""
    login: 'str' = ""

    def __init__(self, name, login):
        self.name = name
        self.login = login
        super(User).__init__()

    def __repr__(self):
        return ""

class BaseUser:
    def __repr__(self):
        # Создает циклическую ссылку
        return User.__repr__(self)
```

В этом примере экземпляр `User` связывается с типом `BaseUser`, который содержит обратную ссылку на экземпляр `User`. Цель сборки мусора — найти **недостижимые** объекты и пометить их как мусор.

Некоторые алгоритмы сборки мусора, такие как алгоритм **маркировки и очистки** (**mark and sweep**) или алгоритм **остановки с копированием** (**stop and copy**), начинают с корневого узла системы и анализируют все *достижимые* объекты. Это трудно сделать в CPython, потому что модули расширения С могут определять и сохранять собственные объекты. Вы не можете определить все объекты, просто просмотрев `locals()` и `globals()`.

Для продолжительных процессов или крупных задач обработки данных нехватка памяти создаст серьезную проблему.

Вместо этого сборщик мусора CPython использует существующий счетчик ссылок и специализированный алгоритм сборки мусора для нахождения всех недостижимых объектов. Так как счетчики ссылок уже существуют, ролью сборщика мусора CPython становится поиск циклических ссылок в некоторых контейнерных типах данных.

Типы контейнеров, включенные в сборщик мусора

Сборщик мусора ищет объекты, у которых в определении типа установлен флаг `Py_TPFLAGS_HAVE_GC`. Определения типов рассматриваются в главе «Объекты и типы».

Типы, помеченные для сборки мусора:

- Классы, методы и функции.
- Объекты ячеек (cell).
- Байтовые массивы, байты и строки Юникода.
- Словари.
- Объекты дескрипторов, используемые в атрибутах.
- Объекты перечисления.
- Исключения.
- Объекты кадров.
- Списки, кортежи, именованные кортежи и множества.
- Объекты памяти.
- Модули и пространства имен.
- Типы и объекты слабых ссылок.
- Итераторы и генераторы.
- Буферы `pickle`.

Чего же не хватает в списке? Чисел с плавающей точкой, целых чисел, логических значений и `NoneType`. Они не помечаются для сборки мусора.

Пользовательские типы, написанные с модулями расширения C, можно пометить как требующие сборки мусора с использованием С API¹ сборщика мусора.

Неотслеживаемые объекты и изменяемость

Сборщик мусора отслеживает некоторые типы и обнаруживает изменения в их свойствах, чтобы определить, являются ли они недостижимыми.

Некоторые экземпляры контейнерного типа никогда не изменяются, поэтому API предоставляет механизм для **отмены отслеживания**. Чем меньше объектов отслеживается сборщиком мусора, тем быстрее и эффективнее выполняется сборка мусора.

Отличным примером неотслеживаемых объектов являются кортежи. Кортежи неизменяемы; после того как они были созданы, изменить их не удастся. Тем не менее кортежи могут содержать изменяемые типы, например списки и словари.

Такая архитектура Python создает много побочных эффектов, одним из которых является алгоритм сборки мусора. Когда создается кортеж (не пустой), он помечается для отслеживания.

При запуске сборщика мусора каждый кортеж обращается к своему содержимому и проверяет, действительно ли в нем только неизменяемые (неотслеживаемые) элементы. Этот шаг выполняется в `_PyTuple_MaybeUntrack()`. Если кортеж определяет, что он содержит только неизменяемые типы (например, логические (булевы) или целочисленные значения), то он исключает себя из списка отслеживания сборщика мусора вызовом `_PyObject_GC_UNTRACK()`.

Вновь созданный пустой словарь не отслеживается сборщиком мусора, но при добавлении в него элемента, который является отслеживаемым объектом, словарь подает запрос на отслеживание.

Чтобы узнать, отслеживается ли тот или иной объект, используйте функцию `gc.is_tracked(obj)`.

¹ <https://docs.python.org/3.8/c-api/gcsupport.html>.

Алгоритм сборки мусора

Перейдем к алгоритму сборки мусора. Команда ключевых разработчиков CPython написала подробное руководство¹, к которому вы можете обратиться за дополнительной информацией.

Инициализация

Точка входа `PyGC_Collect()` выполняет пять шагов от запуска до остановки сборщика мусора:

1. Получить состояние сборщика мусора `GCState` от интерпретатора.
2. Проверить, включен ли сборщик мусора.
3. Проверить, работает ли сборщик мусора в настоящее время.
4. Выполнить функцию сборки мусора `collect()` с обратным вызовом уведомления о прогрессе.
5. Пометить сборку мусора как завершенную.

Вы можете назначить методы, срабатывающие при завершении фаз сборки мусора, добавив их в список `gc.callbacks`. Методы обратного вызова должны иметь сигнатуру `f(stage: str, info: dict)`:

```
Python 3.9 (tags/v3.9:9cf67522, Oct 5 2020, 10:00:00)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import gc
>>> def gc_callback(phase, info):
...     print(f"GC phase:{phase} with info:{info}")
...
>>> gc.callbacks.append(gc_callback)
>>> x = []
>>> x.append(x)
>>> del x
>>> gc.collect()
GC phase:start with info:{'generation': 2,'collected': 0,'uncollectable': 0}
GC phase:stop with info:{'generation': 2,'collected': 1,'uncollectable': 0}
1
```

¹ https://devguide.python.org/garbage_collector/.

Стадия сборки мусора

Главная функция сборки мусора `collect()` выбирает целью одно из трех поколений в CPython. Но прежде чем рассматривать поколения, важно понять, как работает алгоритм сборки мусора.

Для каждого поколения сборщик мусора использует двусвязный список типа `PyGC_HEAD`. Чтобы сборщику не выполнять поиск по всем контейнерным типам, те из них, которые являются целями для него, имеют дополнительный заголовок, который связывает их в двусвязный список.

При создании объект контейнерного типа добавляет сам себя в список, а при уничтожении — удаляет. Пример можно посмотреть в `cellobj.c`:

Objects > `cellobj.c`, строка 7

```
PyObject *
PyCell_New(PyObject *obj)
{
    PyObject *op;

    op = (PyCellObject *)PyObject_GC_New(PyCellObject, &PyCell_Type);
    if (op == NULL)
        return NULL;
    op->ob_ref = obj;
    Py_XINCREF(obj);

    >> _PyObject_GC_TRACK(op);
    return (PyObject *)op;
}
```

Так как ячейки являются изменяемыми, объект помечается как отслеживаемый вызовом `_PyObject_GC_TRACK()`.

При удалении объектов ячеек вызывается функция `cell_dealloc()`. Работа этой функции состоит из трех шагов:

1. Деструктор приказывает сборщику мусора прекратить отслеживание этого экземпляра вызовом `_PyObject_GC_UNTRACK()`. Так как экземпляр был уничтожен, его содержимое не обязательно проверять на изменения при последующей сборке мусора.
2. `Py_XDECREF` — стандартный вызов в любом деструкторе для уменьшения счетчика ссылок. Счетчик ссылок для объекта инициализируется со значением 1, уменьшение компенсирует эту операцию.

3. `PyObject_GC_Del()` удаляет объект из связанного списка сборки мусора вызовом `gc_list_remove()`, после чего освобождает память вызовом `PyObject_FREE()`.

Исходный код `cell_dealloc()`:

Objects > `cellobject.c`, строка 79

```
static void
cell_dealloc(PyCellObject *op)
{
    _PyObject_GC_UNTRACK(op);
    Py_XDECREF(op->ob_ref);
    PyObject_GC_Del(op);
}
```

Когда система сборки мусора начинает работу, она объединяет более молодые поколения в текущее поколение. Например, если сборке подвергается второе поколение, то в начале работы сборщик мусора объединяет объекты первого поколения в список сборки вызовом `gc_list_merge()`. Затем сборщик мусора определяет недостижимые объекты в более молодом поколении (которое в настоящее время является целевым).

Логика определения недостижимых объектов находится в `deduce_unreachable()`. Она состоит из следующих этапов:

1. Для каждого объекта в поколении значение счетчика ссылок `ob->ob_refcnt` копируется в `ob->gc_ref`.
2. Для каждого объекта внутренние (циклические) ссылки вычитаются из `gc_refs` для определения того, сколько объектов может быть уничтожено сборщиком мусора. Если значение `gc_refs` достигает 0, значит, объект недостижим.
3. Создается список недостижимых объектов, и в него добавляются все объекты, удовлетворяющие критериям шага 2.
4. Все объекты, удовлетворяющие критериям шага 2, удаляются из списка поколений.

Единственно правильного способа определения циклических ссылок не существует. Каждый тип должен определить собственную функцию с сигнатурой `traverseproc` в слоте `tp_traverse`.

Чтобы завершить шаг 2 из приведенной выше схемы, `deduce_unreachable()` вызывает функцию обхода для каждого объекта в `subtract_refs()`. Функция

обхода должна активизировать обратный вызов `visit_decref()` для каждого элемента:

Modules › `gcmodule.c`, строка 462

```
static void
subtract_refs(PyGC_Head *containers)
{
    traverseproc traverse;
    PyGC_Head *gc = GC_NEXT(containers);
    for (; gc != containers; gc = GC_NEXT(gc)) {
        PyObject *op = FROM_GC(gc);
        traverse = Py_TYPE(op)->tp_traverse;
        (void) traverse(FROM_GC(gc),
                        (visitproc)visit_decref,
                        op);
    }
}
```

Функции обхода хранятся в исходном коде каждого объекта в `Objects`. Например, функция обхода для типа кортеж, `tupletraverse()`, вызывает `visit_decref()` для всех своих элементов. Тип словарь вызывает `visit_decref()` для всех ключей и значений.

Любой объект, который в конечном итоге не будет перемещен в список недостижимых объектов `unreachable`, переходит в следующее поколение.

Освобождение объектов

После того как недостижимые объекты будут определены, они могут быть (осторожно) освобождены по приведенной ниже схеме. Способ зависит от того, где тип реализует финализатор (в старом или новом слоте):

1. Если объект определил финализатор в наследуемом слоте `tp_del`, то он не может быть безопасно удален и помечается как несобираемый. Такие объекты добавляются в список `gc.garbage`, чтобы разработчик мог уничтожить их вручную.
2. Если объект определил финализатор в слоте `tp_finalize`, то он помечается как финализированный для предотвращения повторного вызова.
3. Если объект на шаге 2 был воскрешен повторной инициализацией, то сборщик мусора перезапускает цикл сборки.
4. Для всех объектов вызывается слот `tp_clear`. Этот слот обнуляет счетчик ссылок `ob_refcnt`, инициируя освобождение памяти.

Сборка мусора по поколениям

Метод сборки мусора по поколениям основан на данных наблюдений, согласно которым большинство объектов (80 % и более) уничтожается вскоре после создания.

Сборка мусора в CPython использует три поколения, ее запуск инициируется пороговыми значениями. Самое молодое поколение (0) имеет высокий порог для предотвращения слишком частого выполнения цикла сборки. Если объект остался жив после сборки мусора, он переходит во второе поколение, а потом в третье.

В функции сборки мусора целью становится одно поколение, в которое перед выполнением объединяются более молодые поколения. По этой причине при выполнении `collect()` для поколения 1 сборка будет проведена в поколении 0. Аналогичным образом выполнение `collect()` для поколения 2 приведет к сборке мусора в поколениях 0 и 1.

При создании экземпляров объектов счетчики поколений увеличиваются. Когда счетчик достигает порога, определенного пользователем, `collect()` выполняется автоматически.

Использование API сборки мусора из Python

Стандартная библиотека CPython включает модуль Python `gc` для взаимодействия с аренами и сборщиком мусора. Пример использования модуля `gc` в отладочном режиме:

```
>>> import gc  
>>> gc.set_debug(gc.DEBUG_STATS)
```

Следующий фрагмент выводит статистику при запуске сборщика мусора:

```
gc: collecting generation 2...  
gc: objects in each generation: 3 0 4477  
gc: objects in permanent generation: 0  
gc: done, 0 unreachable, 0 uncollectable, 0.0008s elapsed
```

Используйте флаг `gc.DEBUG_COLLECTABLE` для получения информации об элементах, которые уничтожаются при сборке мусора. А если объединить его с отладочным флагом `gc.DEBUG_SAVEALL`, он переместит элементы в список `gc.garbage` после сборки мусора:

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_COLLECTABLE | gc.DEBUG_SAVEALL)
>>> z = [0, 1, 2, 3]
>>> z.append(z)
>>> del z
>>> gc.collect()
gc: collectable <list 0x10d594a00>
>>> gc.garbage
[[0, 1, 2, 3, [...]]]
```

Чтобы получить порог, после которого запускается сборщик мусора, вызовите `get_threshold()`:

```
>>> gc.get_threshold()
(700, 10, 10)
```

Также можно получить текущие пороговые счетчики:

```
>>> gc.get_count()
(688, 1, 1)
```

Наконец, алгоритм сборки мусора можно запустить вручную для конкретного поколения; функция вернет размер освобожденной памяти:

```
>>> gc.collect(0)
24
```

Если поколение не задано, по умолчанию используется значение 2, которое объединяет поколения 0 и 1:

```
>>> gc.collect()
20
```

ВЫВОДЫ

В этой главе вы узнали, как CPython выделяет память, управляет ею и освобождает ее. Эти операции выполняются тысячи раз на протяжении жизненного цикла даже простейших Python-скриптов. Надежность и масштабируемость системы управления памятью CPython позволяет масштабировать систему от двухстрочного скрипта до самого популярного веб-сайта в мире.

Система выделения сырой и объектной памяти, представленная в этой главе, пригодится вам при разработке модулей расширения. Модули расширения С

требуют досконального знания системы управления памятью CPython. Даже один пропущенный вызов `Py_INCREF()` может привести к утечке памяти или сбою системы.

При работе с чистым кодом Python хорошее знание сборщика мусора позволит вам писать код длительного исполнения. Например, если вы написали одну функцию, выполнение которой занимает часы, дни и более, эта функция должна тщательно управлять своей памятью в ограничениях системы, в которой она запущена.

Теперь вы сможете использовать некоторые методы, представленные в этой главе, для управления и настройки поколений сборки мусора для оптимизации вашего кода и его затрат памяти.

Параллелизм и конкурентность

Первые компьютеры проектировались с расчетом на то, что в определенный момент времени они будут делать что-то одно. Значительная часть их работы была связана с вычислительной математикой. Но с течением времени возникла необходимость в обработке ввода, получаемого из самых разнообразных источников — вплоть до дальних галактик.

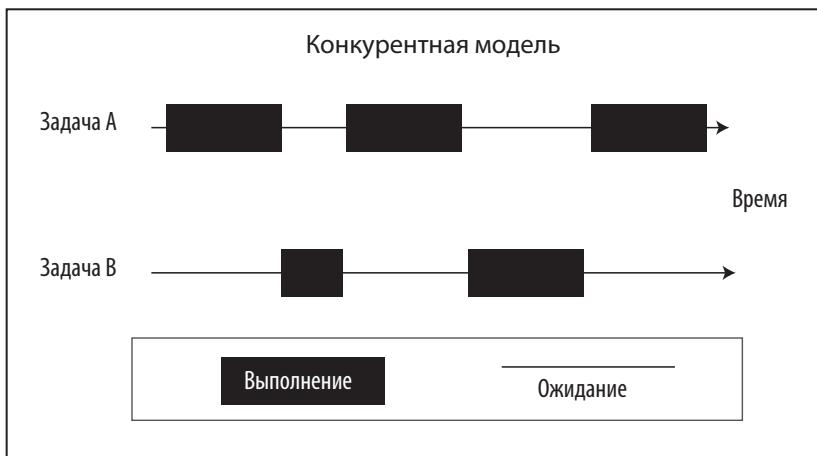
Как следствие, компьютерным программам приходилось проводить много времени, простоявая в ожидании ответа — от шины, устройства ввода, ячейки памяти, вычислений, API или удаленного источника.

Достижением в области вычислений стал отход от однопользовательских терминалов к многозадачным операционным системам. Приложения должны были работать в фоновом режиме — прослушивать сетевые порты, реагировать на данные сети и обрабатывать ввод (например, от мыши).

Многозадачность стала востребованной задолго до появления современных многоядерных процессоров, так что операционные системы уже давно умеют разделять ресурсы между несколькими процессами.

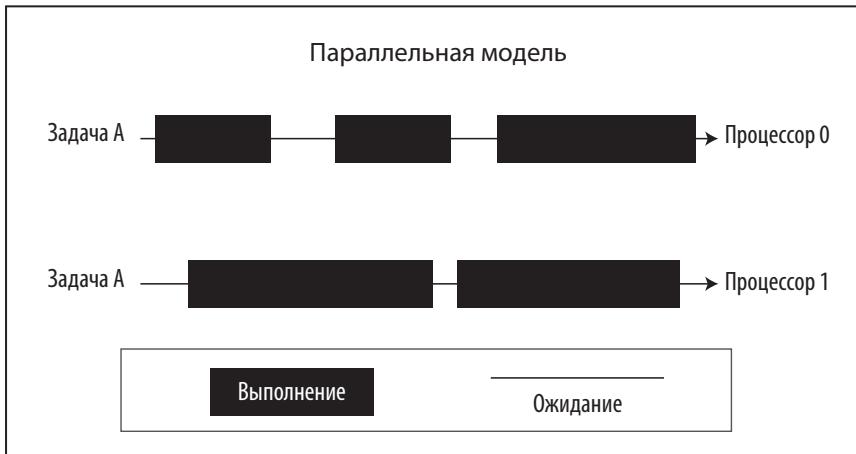
В любой операционной системе центральное место занимает реестр запущенных процессов. У каждого процесса есть владелец, и он может запрашивать ресурсы, например память или процессорное время. В предыдущей главе рассматривалось выделение памяти.

Процессы запрашивают **процессорное время** в форме выполняемых операций. Операционная система управляет тем, какой процесс использует ЦП. Для этого она выделяет процессорное время и планирует процессы с учетом приоритетов:



Даже одному процессу может понадобиться выполнять несколько операций одновременно. Например, если вы работаете в текстовом редакторе, он должен проверять орфографию в процессе ввода. Современные приложения решают эту проблему одновременным выполнением сразу нескольких потоков и обработкой своих ресурсов.

Конкурентность — превосходное решение для реализации многозадачности, но возможности процессоров конечны. Некоторые высокопроизводительные компьютеры оснащаются несколькими процессорами или ядрами для распределения задач. Операционные системы предоставляют механизм планирования процессов по нескольким ЦП:



Итак, компьютеры используют параллелизм и конкурентность для решения проблемы многозадачного выполнения:

- Параллелизм требует нескольких вычислительных устройств. Вычислительными устройствами могут быть процессоры или ядра.
- Конкурентность требует механизма планирования задач, чтобы проприетарные задачи не блокировали ресурсы.

Многие компоненты архитектуры CPython абстрагируют сложность операционных систем, чтобы предоставить простой API для разработчиков. Подход CPython к параллелизму и конкурентности не является исключением.

МОДЕЛИ ПАРАЛЛЕЛИЗМА И КОНКУРЕНТНОСТИ

CPython предлагает множество подходов к **параллелизму и конкурентности**. Выбор зависит от ряда факторов. Также в CPython встречаются варианты использования, когда модели пересекаются.

Может оказаться, что для конкретной задачи можно выбирать между разными реализациями конкурентного выполнения, каждая из которых обладает своими достоинствами и недостатками.

В поставку CPython включены четыре модели в соответствующих модулях:

МОДУЛЬ	КОНКУРЕНТНОСТЬ	ПАРАЛЛЕЛИЗМ
threading	Да	Нет
multiprocessing	Да	Да
asyncio	Да	Нет
subinterpreters	Да	Да

СТРУКТУРА ПРОЦЕССА

Одной из задач операционной системы, такой как Windows, macOS или Linux, является управление запущенными процессами. Такими процессами

могут быть UI-приложения вроде браузеров или IDE. Это также могут быть фоновые процессы — сетевые службы, службы операционной системы и т. д.

Для управления этими процессами ОС предоставляет API для запуска нового процесса. При создании процесс регистрируется операционной системой, чтобы она знала, какие процессы работают. Каждому процессу назначается уникальный идентификатор (PID). В зависимости от ОС процессы могут обладать другими свойствами.

POSIX-процессы обладают минимальным набором свойств, которые регистрируются в ОС:

- Управляющий терминал.
- Текущий рабочий каталог.
- Действующий идентификатор группы и действующий идентификатор пользователя.
- Дескрипторы файлов и маска режимов создания файла.
- Идентификатор группы процессов и идентификатор процесса.
- Реальный идентификатор группы и реальный идентификатор пользователя.
- Корневой каталог.

Чтобы просмотреть эти атрибуты для работающих процессов в macOS или Linux, выполните команду `ps`.

СМ. ТАКЖЕ

Стандарт IEEE POSIX (1003.1-2017) определяет интерфейс и стандартное поведение процессов и потоков.

Windows содержит похожий набор свойств, но устанавливает собственный стандарт. Файловые разрешения, структуры каталогов и реестр процессов в Windows сильно отличаются от POSIX.

Для получения информации о процессах Windows, представленных структурой `Win32_Process`, можно обратиться с запросом к среде WMI (Windows Management Instrumentation) или воспользоваться Диспетчером задач.

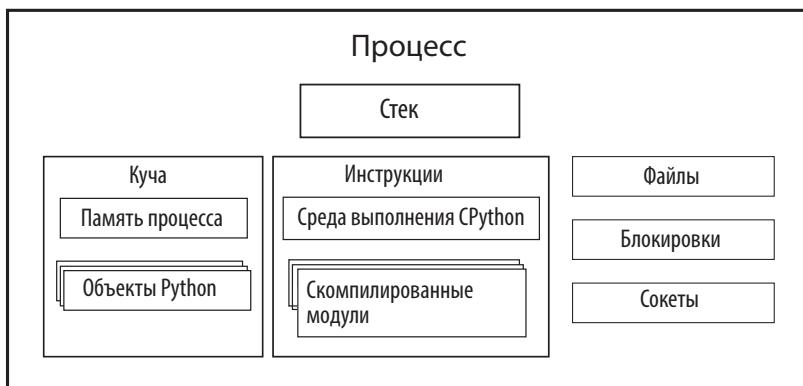
При запуске процесса в операционной системе ему предоставляется:

- Стек в памяти для вызова подпрограмм.
- Куча (см. «Динамическое выделение памяти в C»).
- Доступ к файлам, блокировкам и сокетам операционной системы.

При выполнении процесса ЦП вашего компьютера также хранит дополнительные данные:

- **Регистр** с текущей выполняемой инструкцией или любыми другими данными, необходимыми процессу для этой инструкции.
- **Указатель на инструкцию**, или программный счетчик, указывающий, какая инструкция выполняется в настоящее время.

Процесс CPython включает скомпилированный интерпретатор Python и скомпилированные модули. Модули загружаются во время выполнения и преобразуются в инструкции циклом вычисления CPython:



Регистр программы и программный счетчик указывают на одну инструкцию в процессе. Это означает, что в определенный момент времени выполняется

только одна инструкция. Для CPython это означает, что только одна инструкция байт-кода Python может выполняться одновременно.

Существуют два основных подхода к параллельному выполнению инструкций в процессе:

1. Ответвление (fork) другого процесса.
2. Порождение (spawn) потока.

Итак, мы разобрались в том, из чего состоит процесс, и можем переходить к ответвлению и порождению дочерних процессов.

МНОГОПРОЦЕССОРНЫЙ ПАРАЛЛЕЛИЗМ

POSIX-системы предоставляют API, при помощи которого любой процесс может ответвить (fork) дочерний процесс. Ответвление процессов осуществляется низкоуровневым вызовом API к операционной системе. Такой вызов может осуществляться любым выполняемым процессом.

При этом операционная система клонирует все атрибуты текущего процесса и создает новый процесс. Клонируется куча, регистр и значение счетчика родительского процесса. Во время ветвления дочерний процесс может прочитать любые переменные из родительского.

Ответвление процесса в POSIX

Для примера возьмем программу с преобразованием температур, описанную в начале раздела «Динамическое выделение памяти в С». Ее можно адаптировать так, чтобы вызовом `fork()` порождался дочерний процесс для каждого значения по шкале Фаренгейта вместо их последовательного вычисления. Каждый дочерний процесс продолжит выполнение с этой точки:

cpython-book-samples › 33 › thread_celsius.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static const double five_ninths = 5.0/9.0;

double celsius(double fahrenheit){
```

```
        return (fahrenheit - 32) * five_ninths;
    }

int main(int argc, char** argv) {
    if (argc != 2)
        return -1;
    int number = atoi(argv[1]);
    for (int i = 1 ; i <= number ; i++ ) {
        double f_value = 100 + (i*10);
        pid_t child = fork();
        if (child == 0) { // Is child process
            double c_value = celsius(f_value);
            printf("%f F is %f C (pid %d)\n", f_value, c_value, getpid());
            exit(0);
        }
    }
    printf("Spawned %d processes from %d\n", number, getpid());
    return 0;
}
```

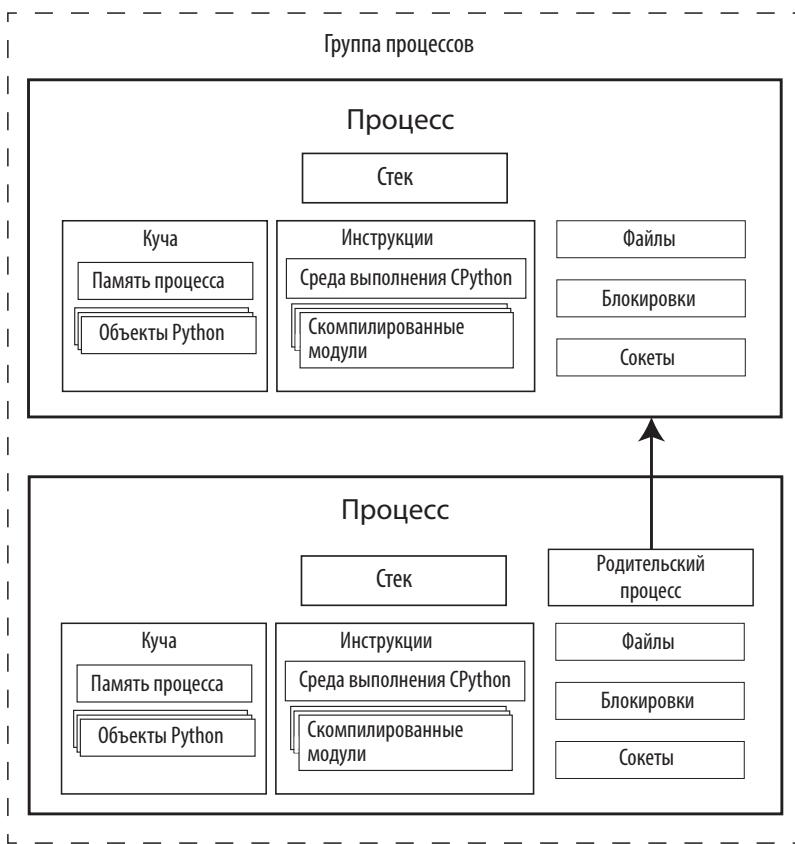
Результат выполнения этой программы в командной строке будет выглядеть примерно так:

```
$ ./thread_celsius 4
110.000000 F is 43.333333 C (pid 57179)
120.000000 F is 48.888889 C (pid 57180)
Spawned 4 processes from 57178
130.000000 F is 54.444444 C (pid 57181)
140.000000 F is 60.000000 C (pid 57182)
```

Родительский процесс (57178) породил четыре процесса. Для каждого дочернего процесса программа продолжается в строке `child = fork()`, в которой итоговое значение `child` равно 0. Затем вычисление завершается, значение выводится, а процесс завершается. Наконец, родительский процесс выводит количество порожденных процессов и свой идентификатор PID.

Третьему и четвертому дочернему процессу понадобилось больше времени для завершения, чем родительскому. Это объясняет, почему родительский процесс выводит свой результат раньше, чем третий и четвертый.

Родительский процесс может завершиться (со своим кодом выхода) раньше дочерних. Дочерние процессы добавляются операционной системой в группу процессов, что упрощает управление всеми взаимосвязанными процессами.



Основной недостаток такого подхода к реализации параллелизма заключается в том, что дочерний процесс является полной копией родительского.

В нашем случае это означает, что будут работать два интерпретатора CPython, и обоим придется загружать модули и все библиотеки. Это приводит к значительным затратам ресурсов. Использование нескольких процессов имеет смысл в тех случаях, когда размер выполняемой задачи превышает затраты ресурсов на ответвление.

У ответвленных процессов есть и другой важный недостаток: они используют отдельную от родительского процесса кучу. Это означает, что дочерний процесс не может сделать запись в пространстве памяти родительского процесса.

При создании дочернего процесса куча родителя становится доступной для дочернего процесса. Для передачи информации родителю должна

использоваться та или иная разновидность межпроцессных коммуникаций (IPC).

ПРИМЕЧАНИЕ

Модуль `os` предоставляет обертку для `fork()`.

Многопроцессная обработка в Windows

До настоящего момента речь шла о модели POSIX. У Windows нет эквивалента `fork()`, а Python должен (по мере возможности) предоставлять одинаковый API в Linux, macOS и Windows.

Для решения проблемы порождения другого процесса `python.exe` с аргументом командной строки `-c` используется API `CreateProcessW()`. Эта операция, называемая *порождением* (*spawning*) процесса, также доступна в POSIX. Она неоднократно встречается вам в данной главе.

Пакет `multiprocessing`

CPython предоставляет API поверх системного API ветвления процессов. Это упрощает создание многопроцессного параллелизма в Python.

API доступен в пакете `multiprocessing`, который предоставляет обширные возможности для создания пулов процессов, очередей, ветвлений, общих куч памяти, объединения процессов и т. д.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к многопроцессной обработке.

ФАЙЛ	НАЗНАЧЕНИЕ
Lib ▶ <code>multiprocessing</code>	Исходный код Python для пакета <code>multiprocessing</code>
Modules ▶ <code>_posixsubprocess.c</code>	Модуль расширения C, являющийся оберткой для системной функции POSIX <code>fork()</code>
Modules ▶ <code>_winapi.c</code>	Модуль расширения C, являющийся оберткой для API ядра Windows

ФАЙЛ	НАЗНАЧЕНИЕ
Modules ► _multiprocessing	Модуль расширения C, используемый пакетом multiprocessing
PC ► msvcrtmodule.c	Интерфейс Python для библиотеки среды выполнения Microsoft Visual C

Порождение и ответвление процессов

Пакет `multiprocessing` предоставляет три метода для запуска нового параллельного процесса:

1. Ответвление интерпретатора (только POSIX).
2. Порождение нового процесса интерпретатора (POSIX и Windows).
3. Запуск fork-сервера, в котором создается новый процесс, способный ответить любое количество процессов (только POSIX).

ПРИМЕЧАНИЕ

В Windows и macOS по умолчанию используется порождение, а в Linux – ответвление. Метод по умолчанию можно переопределить методом `multiprocessing.set_start_method()`.

Python API для запуска нового процесса получает вызываемый объект `target` и кортеж аргументов `args`.

Возьмем пример порождения нового процесса для преобразования температур:

cpython-book-samples ► 33 ► spawn_process_celsius.py

```
import multiprocessing as mp
import os

def to_celsius(f):
    c = (f - 32) * (5/9)
    pid = os.getpid()
    print(f"{f}F is {c}C (pid {pid})")

if __name__ == '__main__':
    mp.set_start_method('spawn')
    p = mp.Process(target=to_celsius, args=(110,))
    p.start()
```

Хотя вы можете запустить один процесс, многопроцессорный API предполагает, что вы хотите запустить несколько. Существуют вспомогательные механизмы для порождения нескольких процессов и передачи им наборов данных. Один из них находится в классе Pool.

Пример можно расширить так, чтобы диапазон значений вычислялся в отдельных интерпретаторах Python:

cpython-book-samples ▶ 33 ▶ pool_process_celsius.py

```
import multiprocessing as mp
import os

def to_celsius(f):
    c = (f - 32) * (5/9)
    pid = os.getpid()
    print(f"{f}F is {c}C (pid {pid})"

if __name__ == '__main__':
    mp.set_start_method('spawn')
    with mp.Pool(4) as pool:
        pool.map(to_celsius, range(110, 150, 10))
```

Обратите внимание: в выводе один и тот же идентификатор PID. Так как процесс интерпретатора CPython требует значительных затрат ресурсов, Pool рассматривает каждый процесс в пуле как **воркер** (worker). Если воркер завершается, он будет использоваться повторно.

Это поведение можно изменить, для чего строка:

```
with mp.Pool(4) as pool:
```

заменяется следующим кодом:

```
with mp.Pool(4, maxtasksperchild=1) as pool:
```

Теперь результат выполнения этого примера многопроцессной обработки будет выглядеть примерно так:

```
$ python pool_process_celsius.py
110F is 43.333333333333336C (pid 5654)
120F is 48.8888888888889C (pid 5653)
130F is 54.44444444444445C (pid 5652)
140F is 60.0C (pid 5655)
```

В выводе приводятся идентификаторы порожденных процессов и вычисленные значения.

Создание дочерних процессов

В обоих скриптах создается новый процесс интерпретатора Python, которому передаются данные средствами pickle.

СМ. ТАКЖЕ

Модуль pickle – пакет, используемый для сериализации объектов Python. За дополнительной информацией обращайтесь к статье «[The Python pickle Module: How to Persist Objects in Python](#)» на сайте Real Python.

Для систем POSIX создание подпроцесса модулем `multiprocessing` эквивалентно следующей команде, где `<i>` – дескриптор файла, а `<j>` – дескриптор канала (pipe):

```
$ python -c 'from multiprocessing.spawn import spawn_main; \
spawn_main(tracker_fd=<i>, pipe_handle=<j>)' --multiprocessing-fork
```

Для систем Windows вместо дескриптора файла используется PID родителя, как в следующей команде, где `<k>` – PID родителя, а `<j>` – дескриптор канала:

```
> python.exe -c 'from multiprocessing.spawn import spawn_main; \
spawn_main(parent_pid=<k>, pipe_handle=<j>)' --multiprocessing-fork
```

Передача данных дочернему процессу

Когда в операционной системе создается новый процесс, он ожидает данных инициализации от родительского процесса. Родительский процесс записывает два объекта в поток файлового канала. Поток файлового канала представляет собой специализированный поток ввода/вывода, используемый для передачи данных между процессами в командной строке.

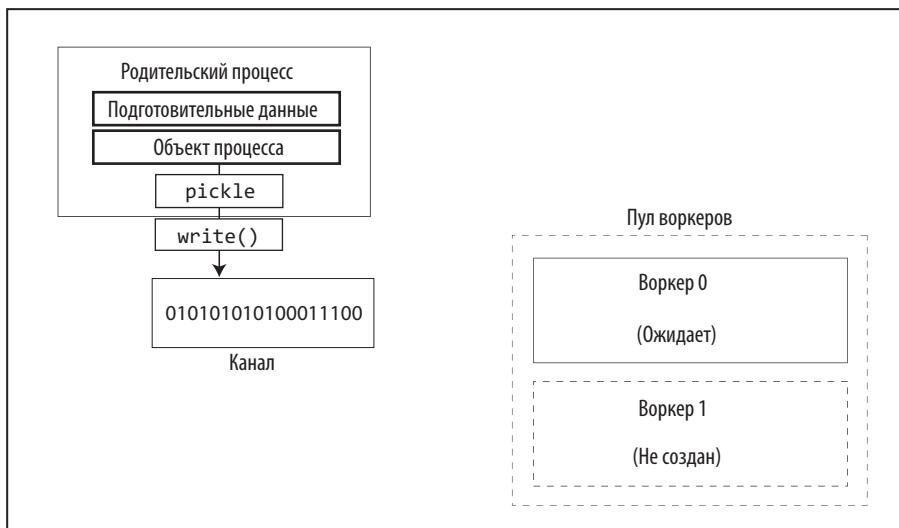
Первый объект, записанный в родительский процесс, – подготовительный объект данных. Он представляет собой словарь с информацией о родителе: каталог исполнения, стартовый метод, специальные аргументы командной строки и `sys.path`. Чтобы просмотреть пример генерируемой информации, выполните `multiprocessing.spawn.get_preparation_data(name)`:

```
>>> import multiprocessing.spawn
>>> import pprint
>>> pprint.pprint(multiprocessing.spawn.get_preparation_data("example"))
{'authkey': b'\x90\xaa\x22[\x18\ri\xbcag]\x93\xfe\xf5\xe5@[wJ\x99p#\x00'
 b'\xce\xd4)1j.\xc3c',
```

```
'dir': '/Users/anthonyshaw',
'log_to_stderr': False,
'name': 'example',
'orig_dir': '/Users/anthonyshaw',
'start_method': 'spawn',
'sys_argv': [''],
'sys_path': [
    '/Users/anthonyshaw',
]}
```

Второй записываемый объект — экземпляр дочернего класса `BaseProcess`. В зависимости от того, как вызывалась многопроцессная обработка и какая операционная система используется, будет сериализоваться один из дочерних классов `BaseProcess`.

Как подготовительные данные, так и объект процесса сериализуются с использованием модуля `pickle` и записываются в поток канала родительского процесса:



ПРИМЕЧАНИЕ

Реализация порождения дочерних процессов POSIX и процесса сериализации находится в файле Lib ▶ multiprocessing ▶ popen_spawn_posix.py.

Реализация для Windows находится в файле Lib ▶ multiprocessing ▶ popen_spawn_win32.py.

Выполнение дочернего процесса

Точка входа дочернего процесса, `multiprocessing.spawn.spawn_main()`, получает аргумент `pipe_handle` и либо `parent_pid` (для Windows), либо `tracked_fd` (для POSIX):

```
def spawn_main(pipe_handle, parent_pid=None, tracker_fd=None):
    ...
    Выполнение кода, заданного данными, полученными по каналу
    ...
    assert is_forking(sys.argv), "Not forking"
```

В Windows функция вызовет функцию API `OpenProcess` для родительского PID. Она создает дескриптор файла `fd` для канала родительского процесса:

```
if sys.platform == 'win32':
    import msvcrt
    import _winapi

    if parent_pid is not None:
        source_process = _winapi.OpenProcess(
            _winapi.SYNCHRONIZE | _winapi.PROCESS_DUP_HANDLE,
            False, parent_pid)
    else:
        source_process = None
    new_handle = reduction.duplicate(pipe_handle,
                                      source_process=source_process)
    fd = msvcrt.open_osfhandle(new_handle, os.O_RDONLY)
    parent_sentinel = source_process
```

В POSIX `pipe_handle` становится дескриптором файла `fd` и дублируется для того, чтобы стать значением `parent_sentinel`:

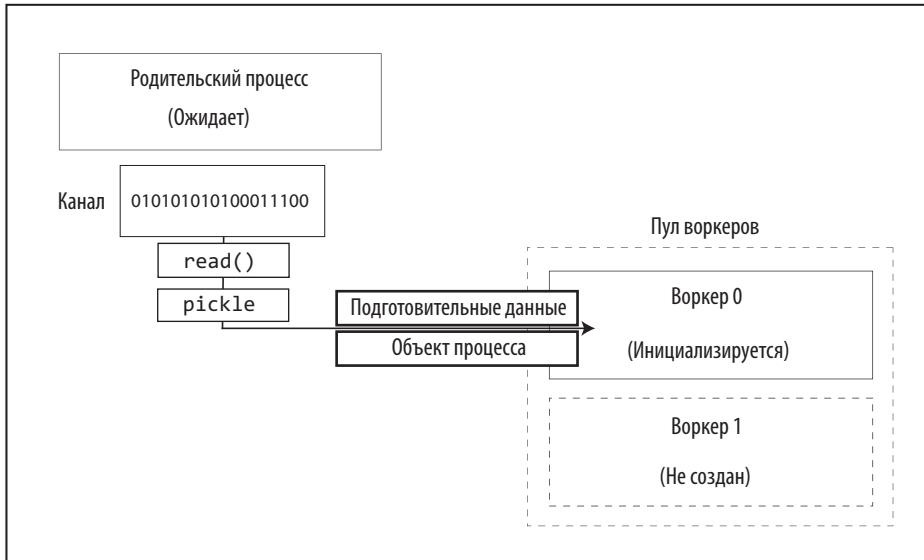
```
else:
    from . import resource_tracker
    resource_tracker._resource_tracker._fd = tracker_fd
    fd = pipe_handle
    parent_sentinel = os.dup(pipe_handle)
```

Теперь `_main()` вызывается с файловым дескриптором родительского канала `fd` и сигнальной меткой родительского процесса `parent_sentinel`. Возвращаемое значение `_main()` станет кодом выхода для процесса, а интерпретатор завершается:

```
exitcode = _main(fd, parent_sentinel)
sys.exit(exitcode)
```

`_main()` вызывается с `fd` и `parent_sentinel` для проверки того, произошел ли выход из родительского процесса во время выполнения дочернего.

`_main()` десериализует двоичные данные в потоке байтов `fd`. Не забудьте, что это файловый дескриптор канала. Десериализация использует ту же библиотеку `pickle`, которая используется родительским процессом:



Первое значение — словарь, содержащий подготовительные данные. Вторым значением является экземпляр `SpawnProcess`, для которого затем вызывается `_bootstrap()`:

```

def _main(fd, parent_sentinel):
    with os.fdopen(fd, 'rb', closefd=True) as from_parent:
        process.current_process()._inheriting = True
        try:
            preparation_data = reduction.pickle.load(from_parent)
            prepare(preparation_data)
            self = reduction.pickle.load(from_parent)
        finally:
            del process.current_process()._inheriting
    return self._bootstrap(parent_sentinel)
    
```

`_bootstrap()` обеспечивает создание экземпляра `BaseProcess` по десериализованным данным, после чего вызывается целевая функция с переданными

аргументами, включая именованные. Последняя задача выполняется вызовом `BaseProcess.run()`:

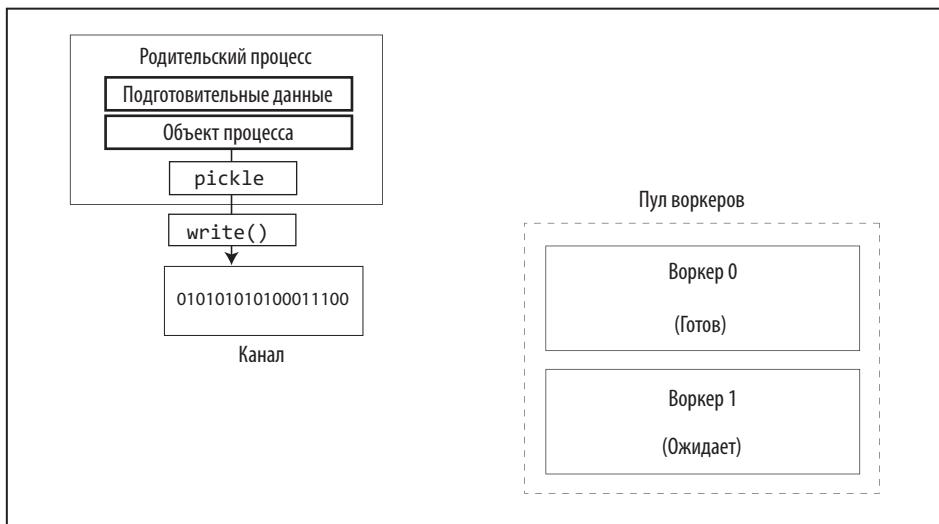
```
def run(self):
    ...
    Метод, выполняемый в подпроцессе; может переопределяться в подклассе
    ...
    if self._target:
        self._target(*self._args, **self._kwargs)
```

Код выхода `self._bootstrap()` устанавливается как общий код выхода, и дочерний процесс прекращает работу.

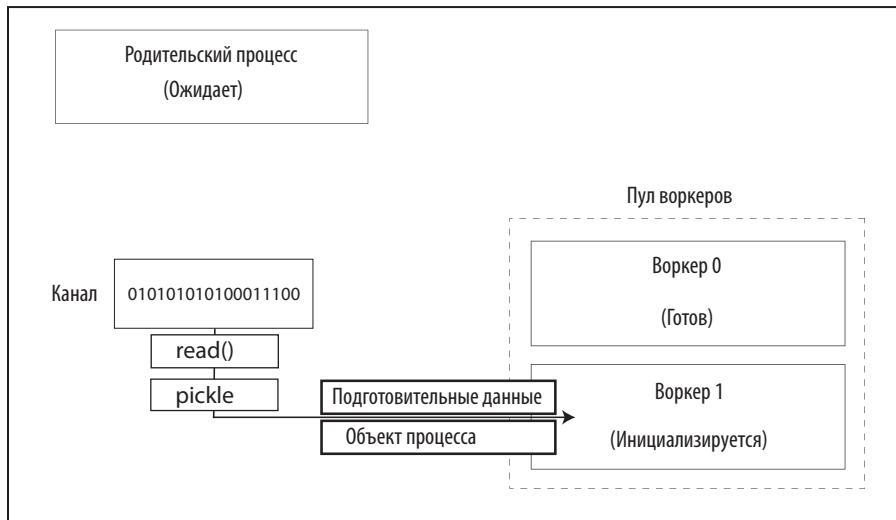
Эта схема позволяет родительскому процессу сериализовать модуль и исполняемую функцию. Также она позволяет дочернему процессу десериализовать этот экземпляр, выполнить функцию с аргументами и вернуть управление.

Она не дает возможности обмениваться данными после запуска дочернего процесса. Для решения этой задачи используется расширение объектов `Queue` и `Pipe`.

Если процессы создаются в пуле, то первый процесс будет готовым и находится в состоянии ожидания. Родительский процесс повторяет схему и отправляет данные следующему воркеру (worker):



Следующий воркер получает данные, инициализирует свое состояние и выполняет целевую функцию:



Для обмена любыми данными за пределами инициализации необходимо использовать **очереди и каналы**.

Обмен данными с помощью очередей и каналов

В предыдущем разделе было показано, как порождаются дочерние процессы, а каналы используются как поток сериализации, чтобы сообщить дочернему процессу, какая функция вызывается с аргументами.

Существуют два типа взаимодействий между процессами в зависимости от природы задачи: **очереди и каналы**. Но прежде чем рассматривать их, необходимо разобраться в том, как операционные системы защищают доступ к ресурсам, используя специальные переменные, называемые **семафорами**.

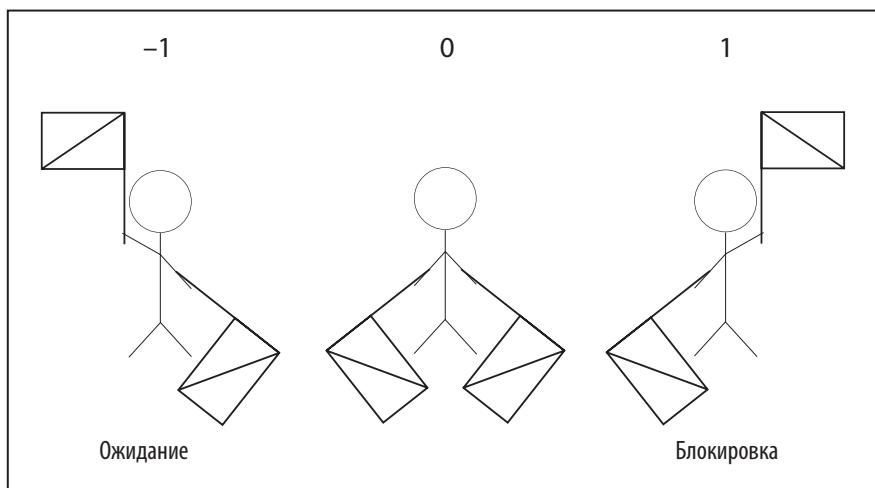
Семафоры

Многие механизмы многопроцессорной обработки применяют семафоры как сигналы о том, какие ресурсы заблокированы, не используются или находятся в ожидании. Операционные системы применяют двоичные семафоры как простой тип переменной для блокировки таких ресурсов, как файлы, сокеты и т. д.

Если один процесс осуществляет запись в файл или сетевой сокет, другой процесс не должен неожиданно начать запись в тот же файл. Это мгновенно приведет к повреждению данных.

Вместо этого операционные системы устанавливают блокировку для ресурсов при помощи семафора. Процессы также могут сигнализировать, что они ожидают разблокировки, чтобы, когда это произойдет, получить сообщение о том, что ресурс готов и им можно начать пользоваться.

В реальном мире семафоры представляют собой сигнальный механизм, который передает сообщения с помощью флагов. Вот так можно представить, как семафор подает сигнал о том, что ресурс находится в ожидании, заблокирован или не используется:



В разных операционных системах используются разные API семафоров, поэтому существует класс абстракции `multiprocessing.synchronize.Semaphore`.

Семафоры используются в CPython для многопроцессной обработки, потому что они одновременно обеспечивают безопасность и потока, и процесса. Операционная система решает проблемы с потенциальными взаимными блокировками, возникающими при чтении или записи с одним семафором.

Реализация API функций семафоров находится в модуле расширения С `Modules ▶ _multiprocessing ▶ semaphore.c`. Этот модуль расширения предоставляет единый метод для создания, блокировки и освобождения семафоров наряду с другими операциями.

Вызов к операционной системе осуществляется через серию макросов, которые компилируются в разные реализации в зависимости от платформы ОС.

В Windows макросы используют для семафоров функции API <winbase.h>:

```
#define SEM_CREATE(name, val, max) CreateSemaphore(NULL, val, max, NULL)
#define SEM_CLOSE(sem) (CloseHandle(sem) ? 0 : -1)
#define SEM_GETVALUE(sem, pval) _GetSemaphoreValue(sem, pval)
#define SEM_UNLINK(name) 0
```

В POSIX макросы используют API <semaphore.h>:

```
#define SEM_CREATE(name, val, max) sem_open(name, O_CREAT | O_EXCL, 0600, ...
#define SEM_CLOSE(sem) sem_close(sem)
#define SEM_GETVALUE(sem, pval) sem_getvalue(sem, pval)
#define SEM_UNLINK(name) sem_unlink(name)
```

Очереди

Очереди отлично подходят для отправки небольших объемов данных нескольким процессам и от них.

Приведенный выше пример многопроцессной обработки можно адаптировать для использования экземпляра `multiprocessing Manager()` и создания двух очередей:

1. `inputs` для хранения входных значений по шкале Фаренгейта.
2. `outputs` для хранения полученных значений по шкале Цельсия.

Измените размер пула на 2, чтобы в нем использовались два воркера:

cpython-book-samples ▶ 33 ▶ pool_queue_celsius.py

```
import multiprocessing as mp

def to_celsius(input: mp.Queue, output: mp.Queue):
    f = input.get()
    # Продолжительная задача...
    c = (f - 32) * (5/9)
    output.put(c)

if __name__ == '__main__':
    mp.set_start_method('spawn')
    pool_manager = mp.Manager()
    with mp.Pool(2) as pool:
        inputs = pool_manager.Queue()
        outputs = pool_manager.Queue()
```

```

        input_values = list(range(110, 150, 10))
        for i in input_values:
            inputs.put(i)
        pool.apply(to_celsius, (inputs, outputs))

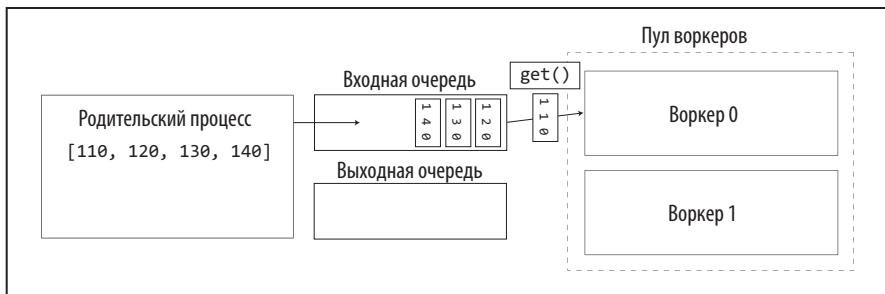
    for f in input_values:
        print(outputs.get(block=False))
    
```

Программа выводит в очередь `outputs` возвращаемый список кортежей:

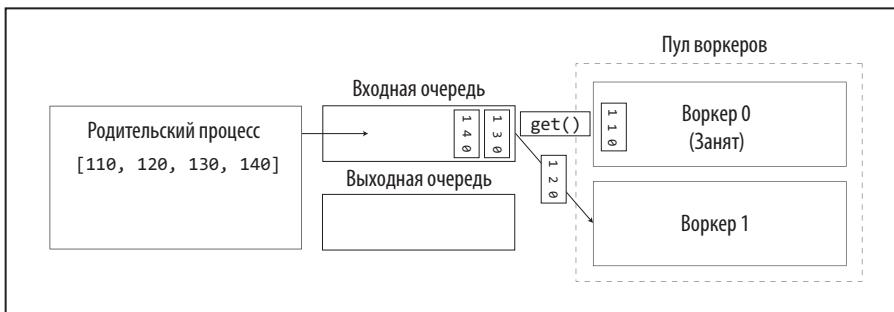
```

$ python pool_queue_celsius.py
43.33333333333336
48.88888888888889
54.44444444444445
60.0
    
```

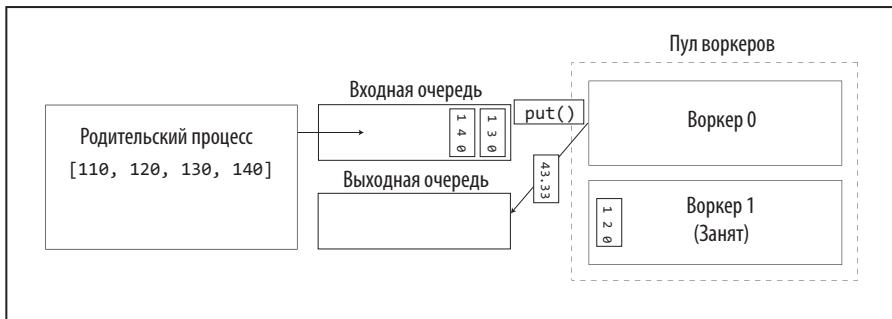
Родительский процесс сначала помещает входные значения в очередь `inputs`. Первый воркер получает элемент из очереди. Каждый раз, когда из очереди извлекается элемент вызовом `.get()`, для объекта очереди используется семафорная блокировка:



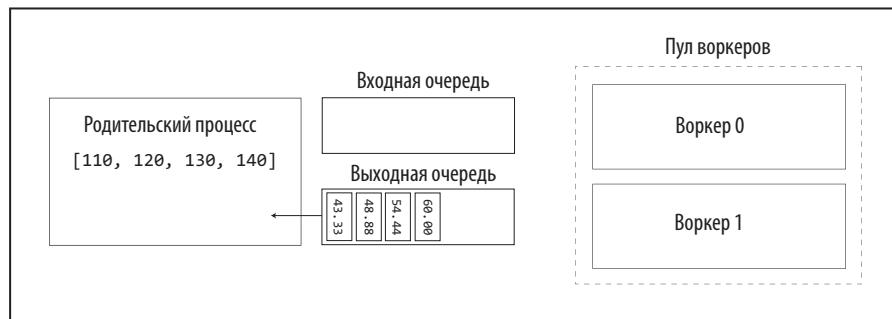
Пока этот воркер занят, второй воркер получает другое значение из очереди:



Первый воркер завершает вычисления и помещает полученное значение в очередь `outputs`:



Две очереди используются для разделения входных и выходных значений. В конечном итоге все входные значения будут обработаны, а очередь `outputs` будет заполнена. После этого значения выводятся родительским процессом:



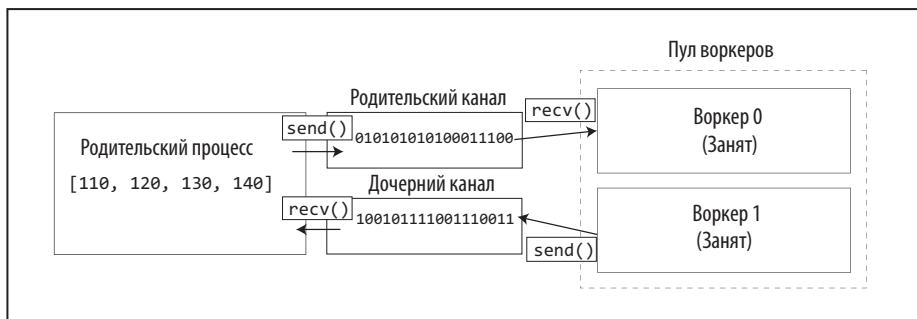
Этот пример показывает, что пул воркеров может получать очередь с набором небольших дискретных значений и обрабатывать их параллельно, чтобы затем передать полученные данные обратно управляющему процессу.

На практике преобразование температур — слишком мелкое и тривиальное вычисление, не подходящее для параллельного выполнения. Если бы рабочий процесс выполнял другие вычисления, создающие значительную нагрузку на процессор, это бы обеспечило значительный выигрыш в производительности на многопроцессорных или многоядерных компьютерах.

Для потоковых данных вместо дискретных очередей можно использовать каналы.

Каналы

В пакете `multiprocessing` определен тип `Pipe`, представляющий коммуникационный канал. При создании экземпляра канала необходимо указать две конечные точки: родителя и потомка. Каждая из них может отправлять и получать данные:



В примере с очередью блокировка неявно устанавливается при отправке и получении данных. Каналы не обладают таким поведением, так что вы должны внимательно следить за тем, чтобы два процесса не пытались что-то записывать в один канал одновременно.

Чтобы адаптировать последний пример для работы с каналом, следует изменить `pool.apply()` на `pool.apply_async()`. Тем самым выполнение следующего процесса заменяется на неблокирующую операцию:

cpython-book-samples › 33 › pool_pipe_celsius.py

```
import multiprocessing as mp

def to_celsius(child_pipe: mp.Pipe):
    f = child_pipe.recv()
    # Продолжительная задача...
    c = (f - 32) * (5/9)
    child_pipe.send(c)

if __name__ == '__main__':
    mp.set_start_method('spawn')
    pool_manager = mp.Manager()
    with mp.Pool(2) as pool:
        parent_pipe, child_pipe = mp.Pipe()
        results = []
        for input in range(110, 150, 10):
```

```

parent_pipe.send(input)
results.append(pool.apply_async(to_celsius, args=(child_pipe,)))
print("Got {0:}".format(parent_pipe.recv()))
parent_pipe.close()
child_pipe.close()

```

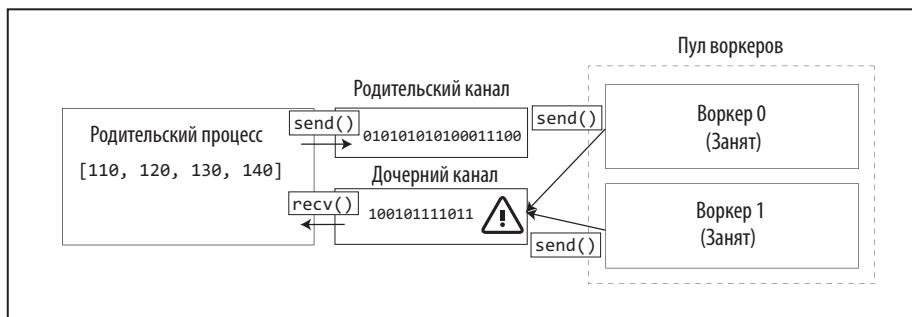
Возникает риск того, что два и более процесса попытаются одновременно прочитать данные из родительского канала в следующей строке:

```
f = child_pipe.recv()
```

Также существует риск того, что два и более процесса попытаются одновременно записать данные в дочерний канал:

```
child_pipe.send(c)
```

В такой ситуации данные окажутся поврежденными как при операции получения, так и при операции отправки:



Чтобы этого не происходило, можно реализовать семафорную блокировку в операционной системе. Тогда все дочерние процессы будут проверять наличие блокировки, прежде чем пытаться читать или записывать данные в тот же канал.

Потребуются две блокировки: для получателя родительского канала и отправителя дочернего канала:

cpython-book-samples ▶ 33 ▶ pool_pipe_locks_celsius.py

```

import multiprocessing as mp

def to_celsius(child_pipe: mp.Pipe, child_lock: mp.Lock):
    child_lock.acquire(blocking=False)

```

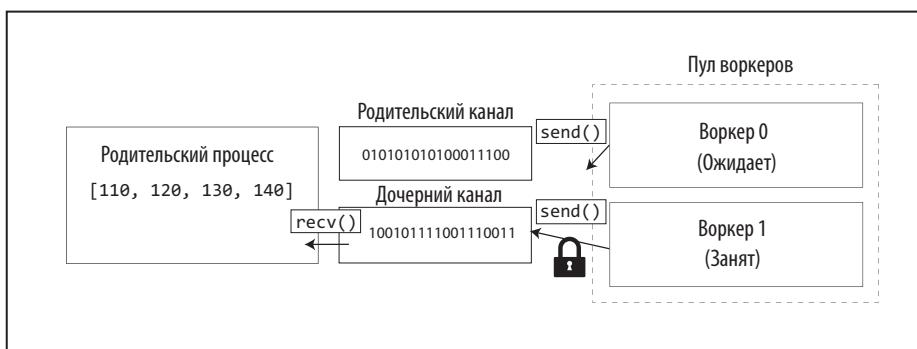
```

try:
    f = child_pipe.recv()
finally:
    child_lock.release()
# Продолжительная задача... снять блокировку перед продолжением
c = (f - 32) * (5/9)
# Повторно установить блокировку после завершения
child_lock.acquire(blocking=False)
try:
    child_pipe.send(c)
finally:
    child_lock.release()

if __name__ == '__main__':
    mp.set_start_method('spawn')
    pool_manager = mp.Manager()
    with mp.Pool(2) as pool:
        parent_pipe, child_pipe = mp.Pipe()
        child_lock = pool_manager.Lock()
        results = []
        for i in range(110, 150, 10):
            parent_pipe.send(i)
            results.append(pool.apply_async(
                to_celsius, args=(child_pipe, child_lock)))
            print(parent_pipe.recv())
        parent_pipe.close()
        child_pipe.close()

```

Теперь рабочие процессы будут ожидать блокировки, прежде чем получить данные, а затем, перед отправкой данных, будут ожидать еще одну:



Этот пример подходит для ситуаций, в которых по каналу передаются большие объемы данных, потому что вероятность коллизии повышается.

Общее состояние процессов

Ранее было показано, как организовать совместное использование данных родительским и дочерним процессом.

Возможны сценарии, в которых требуется обеспечить обмен данными между дочерними процессами. В такой ситуации пакет `multiprocessing` предоставляет два решения:

1. Производительный API общей памяти на базе общего распределения памяти и общих типов С.
2. Гибкий API серверного процесса, поддерживающий сложные типы через класс `Manager`.

Пример приложения

В оставшейся части этой главы мы проведем рефакторинг сканера TCP-портов, чтобы проверить разные методы параллелизма и конкурентности.

По сети связь с хостом устанавливается через порты с номерами от 1 до 65 535. Стандартным сервисам назначаются стандартные порты. Например, HTTP operates работает на порте 80, а HTTPS — на порте 443. Сканер TCP-портов — стандартный инструмент тестирования сети, который проверяет возможность передачи пакетов по сети.

В этом примере используется интерфейс `Queue` — потокобезопасная реализация очереди, сходная с той, которая использовалась в примерах многопроцессорной обработки. Пакет `socket` обеспечивает возможность подключения к удаленному порту с коротким секундным тайм-аутом.

Функция `check_port()` проверяет, ответил ли хост на заданном порте. Если он ответил, `check_port()` добавляет номер порта в очередь `results`.

При выполнении скрипта функция `check_port()` последовательно вызывается для портов 80–100. После завершения очередь `results` очищается, а результаты выводятся в командной строке. Чтобы вы могли увидеть различия, в конце выводится время выполнения:

cpython-book-samples ▶ 33 ▶ portscanner.py

```
from queue import Queue
import socket
import time
```

```
timeout = 1.0

def check_port(host: str, port: int, results: Queue):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.settimeout(timeout)
    result = sock.connect_ex((host, port))
    if result == 0:
        results.put(port)
    sock.close()

if __name__ == '__main__':
    start = time.time()
    host = "localhost" # Замените вашим хостом
    results = Queue()
    for port in range(80, 100):
        check_port(host, port, results)
    while not results.empty():
        print("Port {} is open".format(results.get()))
    print("Completed scan in {} seconds".format(time.time() - start))
```

Программа выводит информацию об открытых портах и затраченное время:

```
$ python portscanner.py
Port 80 is open
Completed scan in 19.623435020446777 seconds
```

Можно провести рефакторинг кода, чтобы в нем использовалась много-процессная обработка. Замените интерфейс `Queue` на `multiprocessing.Queue` и просканируйте порты с использованием исполнителя пула (`pool executor`):

cpython-book-samples ▶ 33 ▶ portscanner_mp_queue.py

```
import multiprocessing as mp
import time
import socket

timeout = 1

def check_port(host: str, port: int, results: mp.Queue):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.settimeout(timeout)
    result = sock.connect_ex((host, port))
    if result == 0:
        results.put(port)
    sock.close()

if __name__ == '__main__':
```

```
start = time.time()
processes = []
scan_range = range(80, 100)
host = "localhost" # Замените вашим хостом
mp.set_start_method('spawn')
pool_manager = mp.Manager()
with mp.Pool(len(scan_range)) as pool:
    outputs = pool_manager.Queue()
for port in scan_range:
    processes.append(pool.apply_async(check_port,
                                       (host, port, outputs)))
for process in processes:
    process.get()
while not outputs.empty():
    print("Port {} is open".format(outputs.get()))
print("Completed scan in {} seconds".format(time.time() - start))
```

Как и следовало ожидать, приложение работает намного быстрее, потому что порты тестируются параллельно:

```
$ python portscanner_mp_queue.py
Port 80 is open
Completed scan in 1.556523084640503 seconds
```

Многопроцессная обработка: выводы

Многопроцессная обработка предоставляет масштабируемый API параллельного выполнения для Python. Данные могут совместно использоваться процессами, а интенсивная вычислительная нагрузка может быть разбита на параллельные задачи для использования возможностей многоядерных или многопроцессорных компьютеров.

Многопроцессная обработка плохо подходит для задач, связанных с вводом/выводом (в отличие от ресурсоемких задач). Например, если вы породите четыре рабочих процесса для чтения и записи в одни и те же файлы, то один будет выполнять всю работу, а остальные три — ожидать освобождения блокировки.

Многопроцессная обработка также не подходит для задач с коротким сроком жизни из-за затрат времени и вычислительных ресурсов, необходимых для запуска нового интерпретатора Python.

В обоих сценариях один из способов, описанных ниже, может оказаться предпочтительным.

МНОГОПОТОЧНОСТЬ

CPython предоставляет как высокоуровневый, так и низкоуровневый API для создания, порождения и управления потоками из Python.

Чтобы понять, как работают потоки Python, необходимо сначала разобраться в потоках операционной системы. В CPython существуют две реализации потоков:

1. `pthreads`: потоки POSIX для Linux и macOS.
2. `nt threads`: потоки NT для Windows.

В разделе «Структура процесса» показано, что процесс обладает следующими характеристиками:

- **Стек** подпрограмм.
- **Куча** памяти.
- Доступ к **файлам, блокировкам и сокетам** операционной системы.

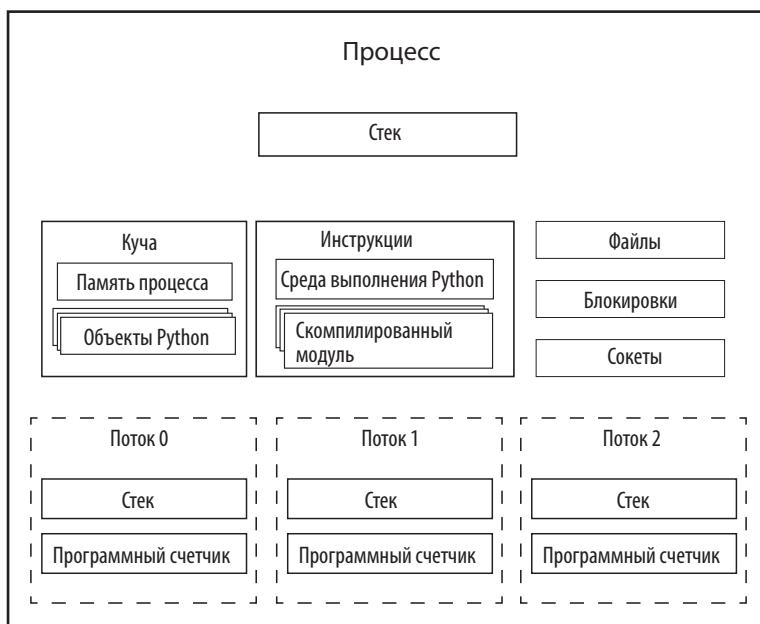
Главное ограничение масштабирования одного процесса заключается в том, что операционная система поддерживает один **программный счетчик** для исполняемого файла.

Для преодоления этого ограничения современные ОС позволяют процессам отправить сигнал операционной системе о необходимости разветвления выполнения на несколько потоков.

Каждый поток имеет собственный программный счетчик, но использует те же ресурсы, что и управляющий процесс. Также поток имеет собственный стек вызовов, поэтому в них могут выполняться разные функции.

Так как сразу несколько потоков могут выполнять чтение и запись в одно пространство памяти, возможны коллизии. Решение проблемы – обеспечение **потокобезопасности**; для этого перед обращением к пространству памяти необходимо убедиться, что пространство памяти заблокировано только одним потоком.

Структура одного процесса с тремя потоками выглядит так:



СМ. ТАКЖЕ

В качестве вводного урока по потоковому API в Python рекомендуем статью «Intro to Python Threading»¹ на сайте Real Python.

GIL

Если вы знакомы с потоками NT или POSIX из С либо если работали на другом высоконивневом языке, то, вероятно, предполагаете, что многопоточность будет параллельной.

¹ <https://realpython.com/intro-to-python-threading/>.

В CPython потоки базируются на C API, но являются потоками Python. Это означает, что каждый поток Python должен выполнять байт-код Python через цикл вычисления.

Цикл вычисления Python не является потокобезопасным. Многие компоненты состояния интерпретатора, такие как сборщик мусора, являются глобальными и используются совместно. Чтобы обойти это препятствие, разработчики CPython реализовали мега-блокировку, которая называется **глобальной блокировкой интерпретатора (GIL, Global Interpreter Lock)**. Перед выполнением любого кода операции в цикле вычисления кадра поток получает блокировку GIL. И хотя такое решение обеспечивает потокобезопасность каждой операции в Python, у него есть один серьезный недостаток. Любые операции, выполнение которых занимает много времени, заставят другие потоки ожидать освобождения GIL, чтобы продолжить работу. Это означает, что в любой конкретный момент времени только один поток может выполнить операцию байт-кода Python.

Для установки GIL вызывается функция `take_gil()`, а для сбросывания — функция `drop_gil()`. Установка GIL осуществляется в основном цикле вычисления кадра, `_PyEval_EvalFrameDefault()`.

Чтобы избежать долговременного удержания GIL при выполнении одного кадра, состояние цикла вычисления содержит флаг `gil_drop_request`. После завершения каждой операции байт-кода в кадре этот флаг проверяется, и GIL временно сбрасывается перед повторной установкой:

```
if (_Py_atomic_load_relaxed(&ceval->gil_drop_request)) {
    /* Предоставить возможность другому потоку */
    if (_PyThreadState_Swap(&runtime->gilstate, NULL) != tstate) {
        Py_FatalError("ceval: tstate mix-up");
    }
    drop_gil(ceval, tstate);

    /* Теперь другие потоки смогут выполняться */

    take_gil(ceval, tstate);

    /* Проверить быстрый выход */
    exit_thread_if_finalizing(tstate);

    if (_PyThreadState_Swap(&runtime->gilstate, tstate) != NULL) {
        Py_FatalError("ceval: orphan tstate");
    }
}
...
...
```

Несмотря на ограничения, которые блокировка GIL накладывает на параллельное выполнение, она делает многопоточность в Python чрезвычайно безопасной и идеально подходящей для одновременного выполнения нескольких задач с интенсивным вводом/выводом.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к циклу вычисления.

ФАЙЛ	НАЗНАЧЕНИЕ
Include ▶ pythread.h	API и определение PyThread
Lib ▶ threading.py	Высокоуровневый API потоков и модуль стандартной библиотеки
Modules ▶ _threadmodule.c	Низкоуровневый API потоков и модуль стандартной библиотеки
Python ▶ thread.c	Расширение С для модуля thread
Python ▶ thread_nt.h	API потоков для Windows
Python ▶ thread_pthread.h	API потоков для POSIX
Python ▶ ceval_gil.h	Реализация блокировок GIL

Запуск потоков в Python

Чтобы продемонстрировать прирост производительности от использования многопоточного кода (несмотря на GIL), мы реализуем простой сетевой сканер портов на Python.

Начнем с клонирования предыдущего скрипта, но изменим логику так, чтобы для каждого порта порождался поток вызовом `threading.Thread()`. Происходит примерно то же, что в API `multiprocessing`, где при вызове передается вызываемый объект `target` и кортеж `args`.

Потоки будут запускаться в цикле, но мы не будем ожидать их завершения. Вместо этого присоединим экземпляр потока к списку `threads`:

```
for port in range(80, 100):
    t = Thread(target=check_port, args=(host, port, results))
    t.start()
    threads.append(t)
```

После того как потоки будут созданы, мы перебираем список `threads` и вызываем `.join()`, чтобы дождаться их завершения:

```
for t in threads:  
    t.join()
```

Затем все элементы в очереди `results` извлекаются и выводятся на экран:

```
while not results.empty():  
    print("Port {0} is open".format(results.get()))
```

Полный скрипт:

cpython-book-samples ▶ 33 ▶ portscanner_threads.py

```
from threading import Thread  
from queue import Queue  
import socket  
import time  
  
timeout = 1.0  
  
def check_port(host: str, port: int, results: Queue):  
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
    sock.settimeout(timeout)  
    result = sock.connect_ex((host, port))  
    if result == 0:  
        results.put(port)  
    sock.close()  
  
def main():  
    start = time.time()  
    host = "localhost" # Замените вашим хостом  
    threads = []  
    results = Queue()  
    for port in range(80, 100):  
        t = Thread(target=check_port, args=(host, port, results))  
        t.start()  
        threads.append(t)  
    for t in threads:  
        t.join()  
    while not results.empty():  
        print("Port {0} is open".format(results.get()))  
    print("Completed scan in {0} seconds".format(time.time() - start))  
  
if __name__ == '__main__':  
    main()
```

В командной строке этот многопоточный скрипт выполняется в десять раз быстрее однопоточного:

```
$ python portscanner_threads.py
Port 80 is open
Completed scan in 1.0101029872894287 seconds
```

Он также выполняется на 50–60 % быстрее многопроцессного скрипта. Не стоит забывать, что многопроцессная обработка сопряжена с определенными затратами на запуск нового процесса. У многопоточных решений затраты тоже есть, но они намного меньше.

Возникает вопрос — если GIL означает, что в любой момент времени выполняться может только одна операция, то почему такое решение работает быстрее?

Следующая команда занимает 1–1000 мс:

```
result = sock.connect_ex((host, port))
```

В модуле расширения C **Modules ▶ socketmodule.c** подключение реализуется следующей функцией:

Modules ▶ socketmodule.c, строка 3245

```
static int
internal_connect(PySocketSockObject *s, struct sockaddr *addr, int addrlen,
                 int raise)
{
    int res, err, wait_connect;

    Py_BEGIN_ALLOW_THREADS
    res = connect(s->sock_fd, addr, addrlen);
    Py_END_ALLOW_THREADS
```

Системный вызов `connect()` заключен между макросами `Py_BEGIN_ALLOW_THREADS` и `Py_END_ALLOW_THREADS`. Макросы определяются в **Include ▶ ceval.h**:

```
#define Py_BEGIN_ALLOW_THREADS { \
    PyThreadState *_save; \
    _save = PyEval_SaveThread(); \
}
#define Py_BLOCK_THREADS      PyEval_RestoreThread(_save);
#define Py_UNBLOCK_THREADS    _save = PyEval_SaveThread();
#define Py_END_ALLOW_THREADS   PyEval_RestoreThread(_save); \
}
```

Таким образом, при выполнении `Py_BEGIN_ALLOW_THREADS` вызывается `PyEval_SaveThread()`. Эта функция изменяет состояние потока на `NULL` и сбрасывает GIL:

Python ▶ ceval.c, строка 444

```
PyThreadState *
PyEval_SaveThread(void)
{
    PyThreadState *tstate = PyThreadState_Swap(NULL);
    if (tstate == NULL)
        Py_FatalError("PyEval_SaveThread: NULL tstate");
    assert(gil_created());
    drop_gil(tstate);
    return tstate;
}
```

Так как GIL сброшен, любой другой выполняемый поток может продолжить выполнение. Этот поток просто будет ожидать системного вызова, не блокируя цикл вычисления.

После того как `connect()` завершится успешно или по тайм-ауту, макрос `Py_END_ALLOW_THREADS` запустит `PyEval_RestoreThread()` с исходным состоянием потока. Состояние потока восстанавливается, и GIL снова устанавливается. Вызов `take_gil()` является блокирующим с ожиданием по семафору:

Python ▶ ceval.c, строка 458

```
void
PyEval_RestoreThread(PyThreadState *tstate)
{
    if (tstate == NULL)
        Py_FatalError("PyEval_RestoreThread: NULL tstate");
    assert(gil_created());

    int err = errno;
    take_gil(tstate);
    /* _Py_Finalizing защищается GIL */
    if (_Py_IsFinalizing() && !_Py_CURRENTLY_FINALIZING(tstate)) {
        drop_gil(tstate);
        PyThread_exit_thread();
        Py_UNREACHABLE();
    }
    errno = err;

    PyThreadState_Swap(tstate);
}
```

Это не единственный системный вызов, который заключается между макросами `Py_BEGIN_ALLOW_THREADS` и `Py_END_ALLOW_THREADS`, управляющими блокированием потоков. В стандартной библиотеке эти макросы используются более трехсот раз, включая следующие области:

- Отправка HTTP-запросов.
- Взаимодействие с локальным оборудованием.
- Шифрование данных.
- Чтение и запись файлов.

Состояние потока

В CPython есть собственная реализация управления потоками. Так как потоки должны выполнять байт-код Python в цикле вычисления, выполнение потока в CPython не сводится к простому порождению потока операционной системы.

Потоки Python называются PyThread, они были кратко рассмотрены в главе «Цикл вычислений CPython».

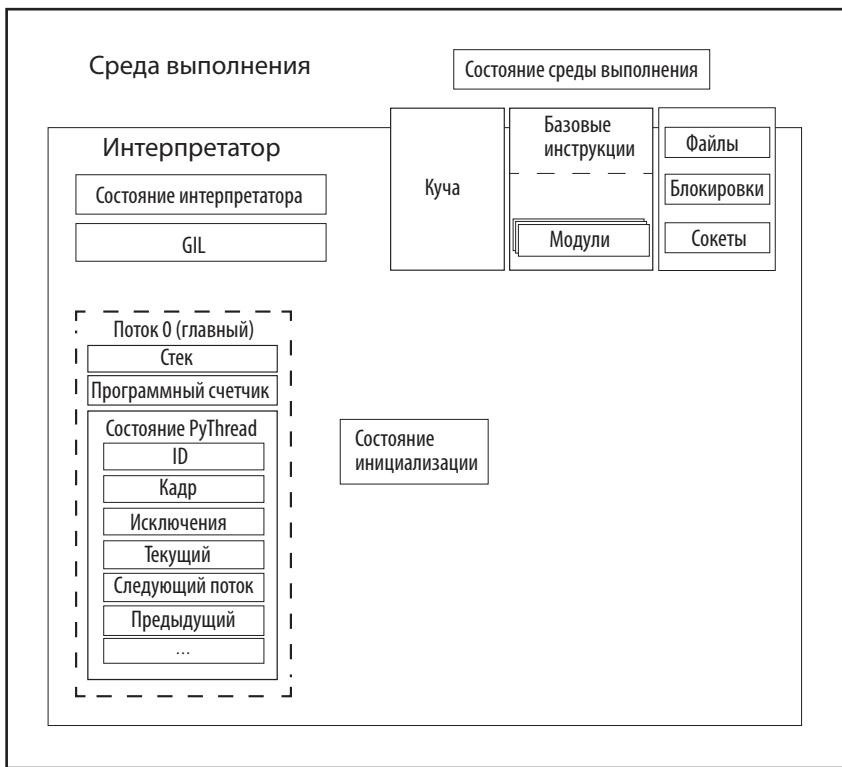
Потоки Python выполняют объекты кода и порождаются интерпретатором.

Краткое напоминание:

- CPython использует единую среду выполнения, которая обладает собственным **состоянием выполнения**.
- CPython может иметь один или несколько интерпретаторов.
- Интерпретатор обладает состоянием, которое называется **состоянием интерпретатора**.
- Интерпретатор получает **объект кода** и преобразует его в серию **объектов кадров**.
- Интерпретатор содержит хотя бы один **поток**, и каждый поток обладает **состоянием потока**.
- Объекты кадров выполняются в стеке, который называется **стеком кадров**.

- СPython обращается к переменным в **стеке значений**.
- **Состояние интерпретатора** включает связанный список всех его потоков.

Однопоточная среда выполнения с единственным интерпретатором будет обладать следующими состояниями:



Структура данных состояния потока `PyThreadState` содержит более тридцати свойств, включая следующие:

- уникальный идентификатор;
- связанный список с другими состояниями;

- состояние интерпретатора, которым был порожден поток;
- выполняемый кадр;
- текущая глубина рекурсии;
- необязательные функции трассировки;
- обрабатываемое исключение;
- обрабатываемое асинхронное исключение;
- стек выданных исключений;
- счетчик GIL;
- счетчики асинхронного генератора.

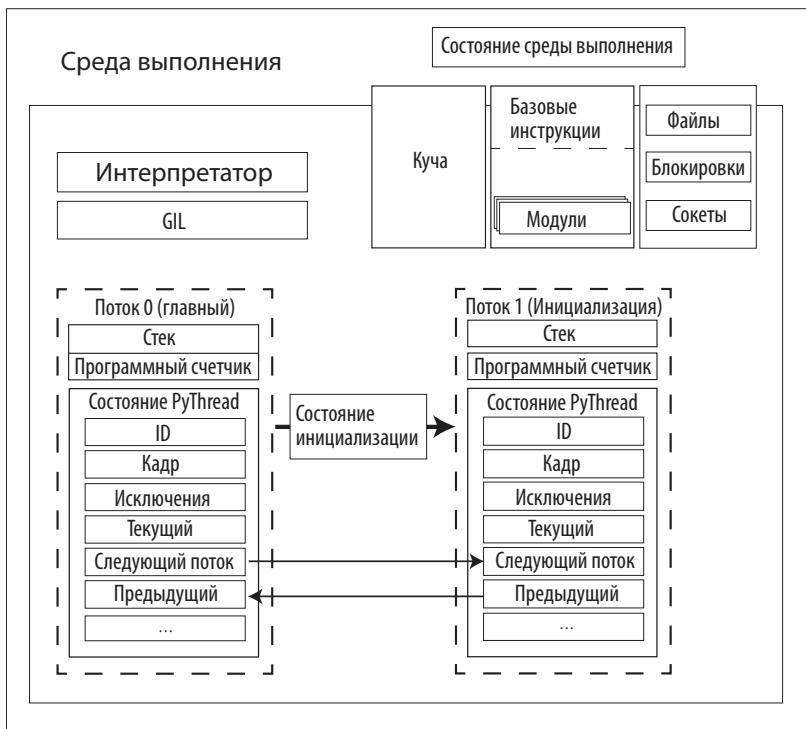
По аналогии с многопроцессными подготовительными данными потоки обладают состоянием инициализации. Однако поскольку они используют общее пространство памяти, необходимость в сериализации и передаче данных через файловый поток отсутствует.

Потоки создаются как экземпляры типа `threading.Thread`. Это высокоуровневый модуль, абстрагирующий тип данных `PyThread`. Экземплярами `PyThread` управляет модуль расширения `C_thread`.

Модуль `_thread` содержит точку входа для запуска нового потока, `thread_PyThread_start_new_thread(). start_new_thread()` — метод для экземпляра типа `Thread`.

Новые экземпляры потоков создаются по следующей схеме:

1. Создается состояние инициализации `bootstate`, связанное с `target`, с аргументами `args` и `kwargs`.
2. `bootstate` связывается с состоянием интерпретатора.
3. Создается новый экземпляр `PyThreadState`, связанный с текущим интерпретатором.
4. Вызывается `PyEval_InitThreads()` для включения блокировки GIL (если она не была включена ранее).
5. Запускается новый поток на основе подходящей для ОС реализации `PyThread_start_new_thread`.



Экземпляр `bootstate` потока содержит следующие свойства.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>interp</code>	<code>PyInterpreterState*</code>	Ссылка на интерпретатор, управляющий этим потоком
<code>func</code>	<code>PyObject * (callable)</code>	Ссылка на объект (<code>callable</code>), вызываемый при запуске потока
<code>args</code>	<code>PyObject * (tuple)</code>	Аргументы для вызова <code>func</code>
<code>keyw</code>	<code>PyObject * (dict)</code>	Именованные аргументы для вызова <code>func</code>
<code>tstate</code>	<code>PyThreadState *</code>	Состояние потока для нового потока

С `bootstate` используются две реализации `PyThread`:

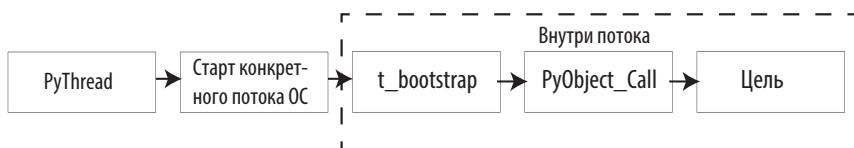
1. Потоки POSIX для Linux и macOS.
2. Потоки NT для Windows.

Обе реализации создают поток операционной системы, устанавливают его атрибут, после чего делают обратный вызов `t_bootstrap()` из нового потока.

Функция вызывается с одним аргументом `boot_raw`, которому присваивается экземпляр `bootstate`, сконструированный в `_PyThread_start_new_thread()`.

Функция `t_bootstrap()` становится интерфейсом между низкоуровневым потоком и средой выполнения Python. Она инициализирует поток, а затем выполняет цель (`target`) с помощью вызова `PyObject_Call()`.

После выполнения цели происходит выход из потока:



Потоки POSIX

Реализация потоков POSIX (`pthread`) находится в файле Python `thread_pthread.h`. Эта реализация абстрагирует C API `<pthread.h>` с дополнительными защитными мерами и оптимизациями.

Размер стека потоков может настраиваться. Python содержит собственную конструкцию кадра стека, как было показано в главе о цикле вычисления. Если в цикле возникнет рекурсия и выполнение кадра достигнет предельной глубины, Python выдаст ошибку `RecursionError`, которую можно обработать блоком `try...except` в Python-коде.

Так как потоки `pthread` имеют собственный размер стека, возможен конфликт между максимальной глубиной рекурсии Python и размером стека `pthread`. Если размер стека потока меньше максимальной глубины кадра в Python, то весь процесс Python аварийно завершится до того, как будет выдана ошибка `RecursionError`.

Максимальная глубина в Python может настраиваться во время выполнения вызовом `sys.setrecursionlimit()`. Для предотвращения сбоев реализация `pthread` в CPython задает в качестве размера стека значение `pythread_stacksize` из состояния интерпретатора.

Многие современные POSIX-совместимые операционные системы поддерживают системное планирование потоков `pthread`. Если в `pyconfig.h` определено

значение `PTHREAD_SYSTEM_SCHED_SUPPORTED`, то поток `pthread` переводится в режим `PTHREAD_SCOPE_SYSTEM`; это означает, что приоритет потока в планировщике ОС определяется относительно других потоков в системе, а не только внутри процесса Python.

После того как свойства потоков будут настроены, создается поток вызовом функции `pthread_create()`. При этом из нового потока будет запущена функция инициализации.

Наконец, дескриптор потока `pthread_t` преобразуется в `unsigned long` и возвращается, превращаясь в идентификатор потока.

Потоки Windows

Потоки Windows, реализованные в `Python ▶ thread_nt.h`, используют похожую, но более простую схему.

Размер стека нового потока настраивается в соответствии со значением интерпретатора `pythread_stacksize` (если оно задано). Затем создается поток функцией Windows API `beginthreadex()`, которая использует функцию загрузки (`bootstrap`) в качестве обратного вызова. Схема завершается возвращением идентификатора потока.

Многопоточность: выводы

Этот раздел не является исчерпывающим руководством по использованию потоков Python. Реализация потоков в Python весьма обширна, и она предоставляет множество механизмов для совместного использования данных потоками, блокировки объектов и ресурсов.

Потоки – эффективный способ повышения производительности приложений Python с интенсивным вводом/выводом. В этом разделе вы узнали, что такая блокировка GIL, зачем она нужна и какие части стандартной библиотеки могут быть выведены из-под ее ограничений.

АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ

Python предоставляет разные способы реализации параллельного программирования без использования потоков и многопроцессной обработки. Эти

возможности неоднократно дополнялись, расширялись и часто заменялись более совершенными альтернативами.

Для использующейся в этой книге версии (3.9) декоратор `@coroutine` считается устаревшим.

Все еще доступно:

- Создание объектов `future` (`futures`) с ключевыми словами `async`.
- Выполнение сопрограмм с использованием конструкции `yield from`.

ГЕНЕРАТОРЫ

Генераторы Python представляют собой функции, которые возвращают значение с помощью оператора `yield` и при последующих обращениях генерируют новые значения.

Генераторы часто используются как механизм перебора значений в большом блоке данных (файле, базе данных, по сети), более эффективно использующий память. Когда используется `yield`, а не `return`, вместо значений возвращаются объекты генераторов. Объект генератора создается оператором `yield` и возвращается на сторону вызова.

Простая функция-генератор, перебирающая буквы от a до z:

cpython-book-samples ▶ 33 ▶ letter_generator.py

```
def letters():
    i = 97 # Letter 'a' in ASCII
    end = 97 + 26 # Буква 'z' в ASCII
    while i < end:
        yield chr(i)
        i += 1
```

Если вызвать функцию `letters()`, она не вернет значения. Вместо этого она вернет объект генератора:

```
>>> from letter_generator import letters
>>> letters()
<generator object letters at 0x1004d39b0>
```

В синтаксис оператора `for` встроена возможность перебора через объект генератора до тех пор, пока он не перестанет возвращать значения:

```
>>> for letter in letters():
...     print(letter)
a
b
c
d
...
```

В этой реализации используется протокол итераторов. Объекты, содержащие метод `__next__()`, можно перебирать в циклах `for` и `while` или с использованием встроенного метода `next()`.

Все контейнерные типы (списки, множества, кортежи) в Python реализуют протокол итераторов. Генераторы уникальны тем, что выполнение метода `__next__()` заново вызывает функцию-генератор из ее последнего состояния.

Генераторы не выполняются в фоновом режиме — они приостанавливаются. Когда вы запрашиваете следующее значение, они возобновляют выполнение. В структуре объекта генератора хранится объект кадра в том виде, в котором он находился при выполнении последней команды `yield`.

Структура генератора

Объекты генераторов создаются шаблонным макросом `_PyGenObject_HEAD(prefix)`.

Этот макрос используется следующими типами и префиксами:

- **Объекты генераторов:** `PyGenObject (gi_)`
- **Объекты сопрограмм:** `PyCoroObject (cr_)`
- **Асинхронные объекты генераторов:** `PyAsyncGenObject (ag_)`

Объекты сопрограмм и асинхронных генераторов будут рассмотрены далее.

Тип `PyGenObject` содержит следующие базовые свойства.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>[x]_code</code>	<code>PyObject * (PyCodeObject*)</code>	Скомпилированная функция, которая возвращает генератор оператором <code>yield</code>
<code>[x]_exc_state</code>	<code>_PyErr_StackItem</code>	Данные исключения (если оно поднимается при вызове генератора)

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
[x]_frame	PyFrameObject*	Текущий объект кадра для генератора
[x]_name	PyObject * (str)	Имя генератора
[x]_qualname	PyObject * (str)	Уточненное имя генератора
[x]_running	char	0 или 1, в зависимости от того, работает ли генератор в настоящее время
[x]_weakreflist	PyObject * (list)	Список слабых ссылок на объекты внутри функции-генератора

Кроме базовых свойств, тип `PyCoroObject` содержит следующее свойство:

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
cr_origin	PyObject * (tuple)	Кортеж, содержащий кадр, в котором создан генератор, и сторону вызова

Кроме базовых свойств, тип `PyAsyncGenObject` содержит следующие свойства.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
ag_closed	int	Флаг для пометки того, что генератор закрыт
ag_finalizer	PyObject *	Ссылка на метод-финализатор
ag_hooks_initied	int	Флаг для пометки того, что хуки инициализированы
ag_running_asyn	int	Флаг для пометки того, что генератор выполняется

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к генераторам.

ФАЙЛ	НАЗНАЧЕНИЕ
Include ▶ genobject.h	API генераторов и определение <code>PyGenObject</code>
Objects ▶ genobject.c	Реализация объекта генератора

Создание генераторов

При компиляции функции, содержащей оператор `yield`, в итоговый объект кода включается дополнительный флаг `CO_GENERATOR`.

В разделе «Построение объектов кадров» главы «Цикл вычисления» было показано, как скомпилированный объект кода при выполнении преобразуется в объект кадра. В этом процессе выделяется особый вариант для генераторов, сопрограмм и асинхронных генераторов.

`_PyEval_EvalCode()` проверяет объект кода на наличие флагов `CO_GENERATOR`, `CO_COROUTINE` и `CO_ASYNC_GENERATOR`. Если один из этих флагов будет найден, то вместо встроенного вычисления объекта кода функция создает кадр и преобразует его в объект генератора, сопрограммы или асинхронного генератора вызовом `PyGen_NewWithQualName()`, `PyCoro_New()` или `PyAsyncGen_New()` соответственно:

```
PyObject *
_PyEval_EvalCode(PyObject *_co, PyObject *globals, PyObject *locals, ...
...
/* Обработка генератора/сопрограммы/асинхронного генератора */
if (co->co_flags & (CO_GENERATOR | CO_COROUTINE | CO_ASYNC_GENERATOR)) {
    PyObject *gen;
    PyObject *coro_wrapper = tstate->coroutine_wrapper;
    int is_coro = co->co_flags & CO_COROUTINE;
    ...
    /* Создает новый генератор, который является владельцем кадра,
     * готового к выполнению, и возвращает его как значение */
    if (is_coro) {
        >>>         gen = PyCoro_New(f, name, qualname);
        >>>     } else if (co->co_flags & CO_ASYNC_GENERATOR) {
        >>>         gen = PyAsyncGen_New(f, name, qualname);
        >>>     } else {
        >>>         gen = PyGen_NewWithQualName(f, name, qualname);
        >>>     }
        ...
        return gen;
    }
...
}
```

Фабрика генераторов `PyGen_NewWithQualName()` получает кадр и для заполнения полей объекта генератора выполняет следующие действия:

1. Свойству `gi_code` присваивается скомпилированный объект кода.
2. Генератор переводится в нерабочее состояние (`gi_running = 0`).
3. Спискам исключений и слабых ссылок присваивается `NULL`.

Чтобы убедиться в том, что `gi_code` является скомпилированным объектом кода для функций-генераторов, импортируйте модуль `dis` и дизассемблируйте его байт-код:

```
>>> from letter_generator import letters
>>> gen = letters()
>>> import dis
>>> dis.disco(gen.gi_code)
2           0 LOAD_CONST 1 (97)
2           2 STORE_FAST 0 (i)
...
...
```

В главе, посвященной циклу вычисления, рассматривался тип объекта кадра. Объекты кадров содержат локальные и глобальные объекты, последние выполненные инструкции и код, который должен быть запущен.

Встроенное поведение и состояние объектов кадров позволяет генераторам приостанавливать и возобновлять выполнение по требованию.

Выполнение генераторов

Каждый раз, когда для объекта генератора вызывается `__next__()`, с экземпляром генератора вызывается функция `gen_iteernext()`, что немедленно приводит к вызову `gen_send_ex()` из `Objects ▶ genobject.c`.

Функция `gen_send_ex()` преобразует объект генератора в следующий результат, возвращаемый `yield`. Можно заметить много общего с построением кадров из объекта кода, так как эти функции выполняют сходные задачи. `gen_send_ex()` используется для стандартных генераторов, сопрограмм и асинхронных генераторов и работает по следующей схеме:

1. Читается текущее состояние потока.
2. Читается объект кадра из объекта генератора.
3. Если генератор запускается при вызове `__next__()`, выдается исключение `ValueError`.
4. Если кадр в генераторе находится на вершине стека, возможны три варианта:
 - Если это сопрограмма, еще не помеченная как закрытая, выдается `RuntimeError`.
 - Если это асинхронный генератор, выдается исключение `StopAsyncIteration`.

- Если это стандартный генератор, поднимается исключение `StopIteration`.
5. Если последняя инструкция в кадре (`f->f_lasti`) все еще содержит `-1`, потому что кадр только что запустился, и если это сопрограмма или асинхронный генератор, то в аргументе не может передаваться никакое другое значение, кроме `None`, и выдается исключение.
 6. В противном случае это первый вызов, и аргументы разрешены. Значение аргумента заносится в стек значений кадра.
 7. Поле `f_back` кадра содержит сторону вызова, которой передаются возвращаемые значения, поэтому полю присваивается текущий кадр в потоке. Это означает, что возвращаемое значение будет отправляться на сторону вызова, а не туда, где генератор создан.
 8. Генератор помечается как работающий.
 9. Последнее исключение в информации об исключениях генератора копируется из последнего исключения состояния потока.
 10. Информация об исключении состояния потока записывается в адрес информации об исключениях генератора. Это означает, что если сторона вызова попадет в точку останова в момент выполнения генератора, трассировка стека пройдет через генератор, и мы увидим код, вызвавший ошибку.
 11. Кадр внутри генератора выполняется в главном цикле `Python ▶ ceval.c`, после чего возвращается значение.
 12. Информация последнего исключения состояния потока сбрасывается до значения, предшествующего вызову кадра.
 13. Генератор помечается как не работающий.
 14. В зависимости от возвращаемого значения может быть несколько вариантов исключений при вызове генератора. Помните, что генераторы должны выдавать исключение `StopIteration` при исчерпании данных — либо вручную, либо не возвращая значение:
 - Если кадр ничего не возвращает, то для стандартных генераторов выдается исключение `StopIteration`, а для асинхронных — `StopAsyncIteration`.
 - Если исключение `StopIteration` было выдано явно, но это сопрограмма или асинхронный генератор, то выдается `RuntimeError`, так как такая ситуация недопустима.

- Если `StopAsyncIteration` было выдано явно, и это асинхронный генератор, то выдается `RuntimeError`, так как такая ситуация недопустима.

15. Наконец, результат возвращается вызывающей стороне `__next__()`.

Объединяя все сказанное, можно заключить, что генераторы — мощная синтаксическая конструкция, в которой одно ключевое слово `yield` запускает цепочку действий для создания уникального объекта, копирования объекта скомпилированного кода в качестве свойства, назначения кадра и сохранения списка переменных в локальной области видимости.

СОПРОГРАММЫ

У сопрограмм есть одно серьезное ограничение: они могут возвращать значения вызовом `yield` только своей непосредственной стороне вызова.

Для преодоления этого ограничения в Python был добавлен дополнительный синтаксис — конструкция `yield from`. С этим синтаксисом можно преобразовать генераторы в служебные функции, содержащие `yield from`.

Например, генератор букв из предыдущего примера можно преобразовать в служебную функцию, у которой начальная буква передается в аргументе. С `yield from` можно выбрать, какой объект генератора будет возвращаться функцией:

cpython-book-samples ▶ 33 ▶ letter_coroutines.py

```
def gen_letters(start, x):
    i = start
    end = start + x
    while i < end:
        yield chr(i)
        i += 1

def letters(upper):
    if upper:
        yield from gen_letters(65, 26) # A--Z
    else:
        yield from gen_letters(97, 26) # a--z

for letter in letters(False):
    # Нижний регистр a--z
    print(letter)
```

```
for letter in letters(True):
    # Верхний регистр A--Z
    print(letter)
```

Генераторы также отлично подходят для ленивых последовательностей (lazy sequences¹), в которых они могут вызываться многократно.

Концепция **сопрограммы** (coroutine), развивающая поведение генераторов (например, возможность приостановки и возобновления выполнения), была применена в Python во многих API.

Генераторы представляют собой ограниченную форму сопрограммы, потому что генераторам можно отправлять данные методом `.send()`. Возможна двусторонняя отправка сообщений между стороной вызова и целью. Сопрограммы также хранят сторону вызова в атрибуте `cr_origin`.

Изначально сопрограммы включались декоратором, но этот вариант считается устаревшим и был заменен «полноценными» сопрограммами с использованием ключевых слов `async` и `await`.

Чтобы пометить, что функция возвращает сопрограмму, необходимо поставить перед ней ключевое слово `async`. Оно явно показывает, что в отличие от генераторов эта функция возвращает сопрограмму, а не значение.

Для создания сопрограммы следует определить функцию с ключевыми словами `async def`. В этом примере добавляется таймер с использованием функции `asyncio.sleep()` и возвращается строка активизации:

```
>>> import asyncio
>>> async def sleepy_alarm(time):
...     await asyncio.sleep(time)
...     return "wake up!"
>>> alarm = sleepy_alarm(10)
>>> alarm
<coroutine object sleepy_alarm at 0x1041de340>
```

При вызове функция возвращает объект сопрограммы.

Существует много способов выполнения сопрограммы. Самый простой — использование функции `asyncio.run(coro)`. Выполните `asyncio.run()` со своим объектом сопрограммы, и через 10 секунд будет выдан сигнал:

¹ Вычисление элемента последовательности откладывается, пока не потребуется результат. — Примеч. ред.

```
>>> asyncio.run(alarm)
'wake up'
```

Среди преимуществ сопрограмм можно выделить возможность их параллельного выполнения. Так как объект сопрограммы представляет собой переменную, которую можно передать функции, эти объекты можно объединить в цепочку или создать из них последовательность.

Например, если вы хотите установить десять сигналов с разными интервалами и таймеры должны запускаться одновременно, эти объекты сопрограмм можно преобразовать в задачи.

API задач используется для планирования и выполнения нескольких сопрограмм одновременно. Перед планированием задач должен быть запущен цикл событий. Цель цикла событий — планирование одновременно выполняемых задач и связывание событий (завершение, отмена, исключения и т. д.) с обратными вызовами.

Когда мы вызывали `asyncio.run()` (из `Lib ▶ asyncio/runners.py`), функция выполняла за нас следующие задачи:

1. Запуск нового цикла событий.
2. Упаковка объекта сопрограммы в задачу.
3. Назначение обратного вызова для завершения задачи.
4. Цикл по задаче до ее завершения.
5. Возвращение результата.

Исходные файлы

Ниже указан исходный файл, относящийся к сопрограммам.

ФАЙЛ	НАЗНАЧЕНИЕ
<code>Lib ▶ asyncio</code>	Реализация <code>asyncio</code> из стандартной библиотеки Python

Циклы событий

Циклы событий связывают асинхронный код воедино. Они пишутся на чистом Python и представляют собой объекты, содержащие задачи.

Все задачи в цикле могут иметь обратные вызовы. Цикл выполняет обратные вызовы при завершении задачи или сбое:

```
loop = asyncio.new_event_loop()
```

Внутри цикла находится последовательность задач, представленная типом `asyncio.Task`. Задачи планируются в цикле, а затем, когда этот цикл запущен, он перебирает все задачи, пока они не будут завершены.

Вы можете преобразовать один таймер в цикл задач:

cpython-book-samples › 33 › sleepy_alarm.py

```
import asyncio

async def sleepy_alarm(person, time):
    await asyncio.sleep(time)
    print(f"{person} -- wake up!")

async def wake_up_gang():
    tasks = [
        asyncio.create_task(sleepy_alarm("Bob", 3), name="wake up Bob"),
        asyncio.create_task(sleepy_alarm("Yudi", 4), name="wake up Yudi"),
        asyncio.create_task(sleepy_alarm("Doris", 2), name="wake up Doris"),
        asyncio.create_task(sleepy_alarm("Kim", 5), name="wake up Kim")
    ]
    await asyncio.gather(*tasks)

asyncio.run(wake_up_gang())
```

Программа выводит следующий результат:

```
Doris -- wake up!
Bob -- wake up!
Yudi -- wake up!
Kim -- wake up!
```

Цикл событий выполняет каждую из сопрограмм и проверяет, завершились ли они. По аналогии с тем, как ключевое слово `yield` может возвращать несколько значений из одного кадра, ключевое слово `await` может вернуть несколько состояний.

Цикл событий выполняет объекты сопрограмм `sleepy_alarm()` снова и снова, пока `await asyncio.sleep()` не вернет конечный результат, и функция `print()` сможет выполниться.

Чтобы эта схема работала, необходимо использовать `asyncio.sleep()` вместо блокирующего (и не `async`-совместимого) вызова `time.sleep()`.

Пример

Многопоточный сканер портов можно преобразовать для `asyncio` по следующей схеме:

- Измените `check_port()` для подключения через сокет, а не через функцию `asyncio.open_connection()`, создающую future-объект вместо немедленного подключения.
- Используйте future-подключение через сокет в событии таймера с `asyncio.wait_for()`.
- Добавьте порт в список результатов (`results`), если проверка была успешной.
- Добавьте новую функцию `scan()` для создания сопрограмм `check_port()` для каждого порта. Добавьте их в список задач `tasks`.
- Объедините все задачи в новую сопрограмму вызовом `asyncio.gather()`.
- Проведите сканирование вызовом `asyncio.run()`.

Код выглядит так:

cpython-book-samples › 33 › portscanner_async.py

```
import time
import asyncio

timeout = 1.0
async def check_port(host: str, port: int, results: list):
    try:
        future = asyncio.open_connection(host=host, port=port)
        r, w = await asyncio.wait_for(future, timeout=timeout)
        results.append(port)
        w.close()
    except OSError: # Ничего не делаем при закрытии порта
        pass
    except asyncio.TimeoutError:
        pass # Порт закрыт, пропустить и продолжить

async def scan(start, end, host):
    tasks = []
    results = []
    for port in range(start, end):
        tasks.append(check_port(host, port, results))
    await asyncio.gather(*tasks)
    return results

if __name__ == '__main__':
```

```
start = time.time()
host = "localhost" # Замените вашим хостом
results = asyncio.run(scan(80, 100, host))
for result in results:
    print("Port {} is open".format(result))
print("Completed scan in {} seconds".format(time.time() - start))
```

Сканирование занимает чуть более секунды:

```
$ python portscanner_async.py
Port 80 is open
Completed scan in 1.0058400630950928 seconds
```

АСИНХРОННЫЕ ГЕНЕРАТОРЫ

Концепции генераторов и сопрограмм, с которыми вы познакомились к настоящему моменту, можно объединить в **асинхронные генераторы**.

Если функция объявляется с ключевым словом `async` и содержит оператор `yield`, при вызове она преобразуется в объект асинхронного генератора.

Асинхронные генераторы, как и обычные, должны выполняться конструкцией, поддерживающей протокол. Вместо `__next__()` асинхронные генераторы содержат метод `__anext__()`.

Обычный цикл `for` не поймет асинхронный генератор, поэтому вместо него необходимо использовать команду `async for`.

Функцию `check_port()` можно преобразовать в асинхронный генератор, который через `yield` возвращает следующий открытый порт, пока не достигнет последнего порта или не найдет заданное количество открытых портов:

```
async def check_ports(host: str, start: int, end: int, max=10):
    found = 0
    for port in range(start, end):
        try:
            future = asyncio.open_connection(host=host, port=port)
            r, w = await asyncio.wait_for(future, timeout=timeout)
            yield port
            found += 1
            w.close()
            if found >= max:
                return
        except asyncio.TimeoutError:
            pass # Закрыт
```

Для выполнения кода используется команда `async for`:

```
async def scan(start, end, host):
    results = []
    async for port in check_ports(host, start, end, max=1):
        results.append(port)
    return results
```

Полный код примера находится в файле `cpython-book-samples` ▶ 33 ▶ `portscanner_async_generators.py`.

СУБИНТЕРПРЕТАТОРЫ

К настоящему моменту мы рассмотрели:

- Параллельное выполнение с многопроцессной обработкой.
- Конкурентное выполнение с потоками и `async`.

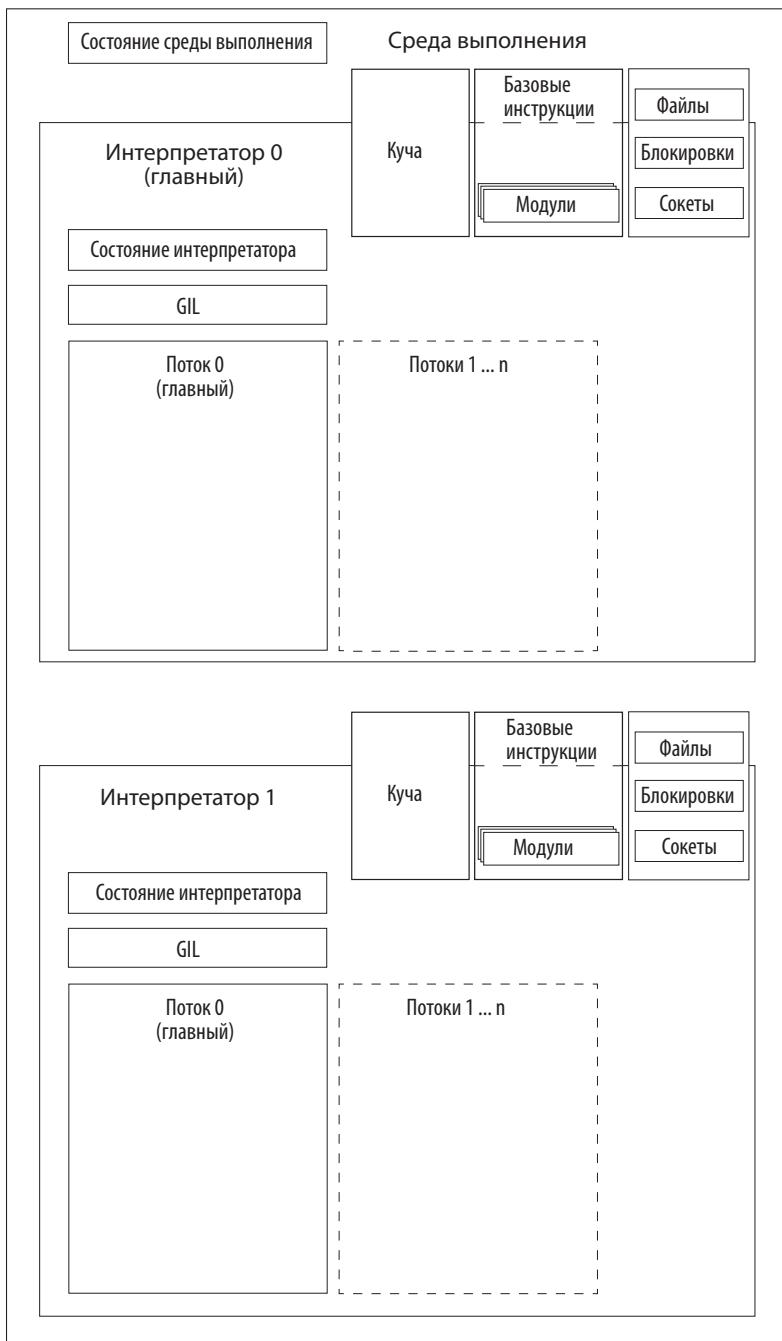
У многопроцессной обработки есть недостатки: межпроцессные коммуникации, использующие каналы и очереди, работают медленнее общей памяти, а дополнительные затраты на запуск нового процесса весьма значительны.

Дополнительные затраты у многопоточных решений и `async` невелики, но они не обеспечивают полноценного параллельного выполнения из-за гарантий потоковой безопасности в GIL.

Еще один вариант — создание субинтерпретаторов (`subinterpreters`); он имеет более низкие дополнительные затраты, чем у многопроцессных решений, и позволяет иметь отдельную блокировку GIL для каждого субинтерпретатора. В конце концов, GIL — глобальная блокировка **интерпретатора**.

В среде выполнения CPython всегда существует только один интерпретатор. Он содержит информацию о состоянии, а внутри интерпретатора может быть один или несколько потоков Python. Интерпретатор является контейнером для цикла вычисления. Он также управляет своей памятью, обеспечивает подсчет ссылок и сборку мусора.

В CPython поддерживаются низкоуровневые С API для создания интерпретаторов, например `Py_NewInterpreter()`:



ПРИМЕЧАНИЕ

Модуль `subinterpreters` остается экспериментальным в версии 3.9, так что API еще может изменяться, а реализация содержит ошибки.

Так как состояние интерпретатора содержит арену выделения памяти — коллекцию всех указателей на объекты Python (локальные и глобальные), — субинтерпретаторы не могут обращаться к глобальным переменным других интерпретаторов.

Как и в случае с многопроцессной обработкой, для совместного использования объектов интерпретаторами необходимо сериализовать их или использовать `ctypes` и некоторую разновидность IPC¹ (сеть, диск, общая память).

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к субинтерпретаторам.

ФАЙЛ	НАЗНАЧЕНИЕ
Lib ▶ <code>_xxsubinterpreters.c</code>	Реализация модуля <code>subinterpreters</code> на C
Python ▶ <code>pylifecycle.c</code>	Реализация API управления интерпретатором на C

Пример

В последнем примере код подключения будет храниться в строке. В версии 3.9 субинтерпретаторы могут выполняться только кодом в строке (`string`).

Для запуска каждого субинтерпретатора запускается список потоков с обратным вызовом функции `run()`.

Эта функция:

- создает канал взаимодействия;
- запускает новый субинтерпретатор;
- отправляет субинтерпретатору выполняемый код;
- получает данные по каналу взаимодействия;

¹ Inter Process Communications; межпроцессное взаимодействие. — Примеч. ред.

- если подключение к порту проходит успешно, он добавляется в потокобезопасную очередь.

cpython-book-samples › 33 › portscanner_subinterpreters.py

```
import time
import _xxsubinterpreters as subinterpreters
from threading import Thread
import textwrap as tw
from queue import Queue

timeout = 1 # В секундах

def run(host: str, port: int, results: Queue):
    # Создание канала взаимодействия
    channel_id = subinterpreters.channel_create()
    interpid = subinterpreters.create()
    subinterpreters.run_string(
        interpid,
        tw.dedent(
"""
import socket; import _xxsubinterpreters as subinterpreters
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.settimeout(timeout)
result = sock.connect_ex((host, port))
subinterpreters.channel_send(channel_id, result)
sock.close()
"""),
        shared=dict(
            channel_id=channel_id,
            host=host,
            port=port,
            timeout=timeout
        )
    )
    output = subinterpreters.channel_recv(channel_id)
    subinterpreters.channel_release(channel_id)
    if output == 0:
        results.put(port)

if __name__ == '__main__':
    start = time.time()
    host = "127.0.0.1" # Выберите ваш хост
    threads = []
    results = Queue()
    for port in range(80, 100):
        t = Thread(target=run, args=(host, port, results))
        t.start()
        threads.append(t)
    for t in threads:
```

```
t.join()
while not results.empty():
    print("Port {0} is open".format(results.get()))
print("Completed scan in {0} seconds".format(time.time() - start))
```

Из-за снижения затрат по сравнению с многопроцессной обработкой этот пример должен выполняться на 30–40 % быстрее и с меньшими затратами памяти:

```
$ python portscanner_subinterpreters.py
Port 80 is open
Completed scan in 1.3474230766296387 seconds
```

ВЫВОДЫ

Поздравляю, вы осилили самую большую главу в книге! В ней был рассмотрен нешуточный объем материала. Давайте вспомним некоторые концепции и их применения.

Для полноценного **параллельного выполнения** понадобится несколько процессоров или ядер. Также необходимо использовать пакет `multiprocessing` или `subinterpreters`, чтобы интерпретатор Python мог выполнять параллельно.

Не забывайте, что время запуска весьма значительно, а каждый интерпретатор создает существенные затраты памяти. Если задачи, которые нужно выполнить, имеют короткий срок жизни, используйте пул воркеров и очередь задач.

Если у вас несколько задач с интенсивным вводом/выводом, которые должны выполняться **одновременно** (конкурентно), используйте многопоточное решение или сопрограммы с пакетом `asyncio`.

При всех четырех подходах необходимо понимать, как безопасно и эффективно передавать данные между процессами или потоками. Если вы захотите закрепить материал этой главы, рассмотрите написанное вами приложение и подумайте, как провести его рефакторинг с применением этих методов.

Объекты и типы

В поставку СPython включен набор базовых типов: строки, списки, кортежи, словари, объекты и т. д. Все эти типы являются встроеннымными. Вам не придется импортировать никакие пакеты, даже из стандартной библиотеки. Например, для создания нового списка можно вызвать функцию `list()`:

```
lst = list()
```

А можно воспользоваться квадратными скобками:

```
lst = []
```

Строки могут создаваться на основе строковых литералов при помощи одинарных или двойных кавычек. В главе «Грамматика и язык Python» были представлены грамматические определения, которые заставляют компилятор интерпретировать двойные кавычки как строковый литерал.

Все типы в Python наследуются от встроенного базового типа `object`, даже строки, кортежи и списки.

В файле `Objects` ► `object.c` находится базовая реализация типа `object`, написанная на чистом С. В ней содержатся некоторые конкретные реализации базовой логики (например, поверхностное сравнение¹).

Можно считать, что объект Python состоит из двух частей:

1. Базовая модель данных с указателями на скомпилированные функции.
2. Словарь с нестандартными атрибутами и методами.

¹ Поверхностное сравнение — сравнение объектов без проверки того, являются ли они одним и тем же объектом и указывают ли на один и тот же адрес памяти. Проще говоря, сравнение значений объектов. — *Примеч. ред.*

Большая часть базового API объекта объявляется в `Objects ▶ object.c`, как и реализация встроенной функции `repr()`, `PyObject_Repr`. Также здесь можно найти `PyObject_Hash()` и другие API.

Все эти функции могут переопределяться в пользовательских объектах с помощью реализации соответствующих **dunder-методов** для объектов Python:

```
class MyObject(object):
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def __repr__(self):
        return "<{0} id={1}>".format(self.name, self.id)
```

Эти встроенные функции в совокупности называются **моделью данных** Python¹. Не все методы в объекте Python являются частью модели данных, что позволяет объектам Python содержать как атрибуты классов и экземпляров, так и методы.

СМ. ТАКЖЕ

Один из лучших источников информации о модели данных Python – книга Лучано Рамальо (Luciano Ramalho) «*Fluent Python*».

ПРИМЕРЫ ЭТОЙ ГЛАВЫ

В этой главе объяснение каждого типа будет сопровождаться примером. В примере мы реализуем оператор «почти равно», созданный в предшествующих главах.

Если вы еще не внесли изменения, которые были описаны в главах, посвященных грамматике CPython и компилятору, обязательно вернитесь и сделайте это, прежде чем читать дальше. Это необходимо для реализации рассматриваемых примеров.

¹ <https://docs.python.org/3/reference/datamodel.html>.

ВСТРОЕННЫЕ ТИПЫ

Базовая модель данных определяется в `PyTypeObject`, а функции определяются в `Objects > typeobject.c`.

Каждый из исходных файлов имеет соответствующий заголовок в `Include`. Например, `Objects/rangeobject.c` имеет заголовочный файл `Include > rangeobject.h`.

Ниже приведен список исходных файлов и соответствующих им типов.

ИСХОДНЫЙ ФАЙЛ	ТИП
<code>Objects object.c</code>	Встроенные методы и базовый объект
<code>Objects boolobject.c</code>	Тип <code>bool</code>
<code>Objects bytearrayobject.c</code>	Тип <code>byte[]</code>
<code>Objects bytesobject.c</code>	Тип <code>bytes</code>
<code>Objects cellobject.c</code>	Тип <code>cell</code>
<code>Objects classobject.c</code>	Абстрактный тип <code>class</code> , используемый в метапрограммировании
<code>Objects codeobject.c</code>	Встроенный тип объекта <code>code</code>
<code>Objects complexobject.c</code>	Тип комплексного числа
<code>Objects iterobject.c</code>	Тип итератора
<code>Objects listobject.c</code>	Тип <code>list</code>
<code>Objects longobject.c</code>	Числовой тип <code>long</code>
<code>Objects memoryobject.c</code>	Базовый тип памяти (объект <code>memoryview</code>)
<code>Objects methodobject.c</code>	Тип метода класса
<code>Objects moduleobject.c</code>	Тип модуля
<code>Objects namespaceobject.c</code>	Тип пространства имен
<code>Objects odictobject.c</code>	Тип упорядоченного словаря
<code>Objects rangeobject.c</code>	Тип генератора диапазона (<code>range</code>)
<code>Objects setobject.c</code>	Тип <code>set</code>
<code>Objects sliceobject.c</code>	Тип ссылки на срез

ИСХОДНЫЙ ФАЙЛ	ТИП
Objects structseq.c	Тип struct.Struct
Objects tupleobject.c	Тип tuple
Objects typeobject.c	Тип type
Objects unicodeobject.c	Тип str
Objects weakrefobject.c	Тип weakref

Некоторые типы будут рассмотрены в этой главе.

ТИПЫ ОБЪЕКТОВ

Так как C в отличие от Python не является объектно-ориентированным языком¹, объекты C не наследуются друг от друга. `PyObject` определяет исходный сегмент данных для любого объекта Python, а `PyObject *` представляет ссылку на него.

При определении типов Python `typedef` использует один из двух макросов:

1. `PyObject_HEAD` (`PyObject`) для простых типов.
2. `PyObject_VAR_HEAD` (`PyVarObject`) для контейнерных типов.

Для простого типа `PyObject` содержит следующие поля:

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>ob_refcnt</code>	<code>Py_ssize_t</code>	Счетчик ссылок на экземпляр
<code>ob_type</code>	<code>_typeobject*</code>	Тип объекта

Например, `cellobj` объявляет одно поле `ob_ref` в дополнение к базовым:

```
typedef struct {
    PyObject_HEAD
    PyObject *ob_ref; /* Содержимое ячейки или NULL для пустой ячейки */
} PyCellObject;
```

¹ <https://realpython.com/python3-object-oriented-programming/>.

Тип `PyVarObject` расширяет `PyObject`, добавляя следующие поля:

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>ob_base</code>	<code>PyObject</code>	Базовый тип
<code>ob_size</code>	<code>Py_ssize_t</code>	Количество содержащихся элементов

Например, объявление типа `int` — `PyLongObject` — выглядит так:

```
struct _longobject {
    PyObject_VAR_HEAD
    digit ob_digit[1];
}; /* PyLongObject */
```

ТИП TYPE

В Python объекты содержат свойство `ob_type`. Для получения его значения можно воспользоваться встроенной функцией `type()`:

```
>>> t = type("hello")
>>> t
<class 'str'>
```

Результатом `type()` является экземпляр `PyTypeObject`:

```
>>> type(t)
<class 'type'>
```

Объекты типов используются для определения реализации абстрактных базовых классов.

Например, объекты всегда реализуют метод `__repr__()`:

```
>>> class example:
...     x = 1
>>> i = example()
>>> repr(i)
'<__main__.example object at 0x10b418100>'
```

Реализация `__repr__()` всегда располагается по одному адресу в определении типа каждого объекта. Эта позиция называется **слотом типа**.

Слоты типов

Все слоты типов определяются в `Include ▶ cpython ▶ object.h`.

Каждый слот типа характеризуется именем свойства и сигнатурой функции. Например, функция `__repr__()` называется `tp_repr` и имеет сигнатуру `reprfunc`:

```
struct PyTypeObject
---
typedef struct _typeobject {
    ...
    reprfunc tp_repr;
    ...
} PyTypeObject;
```

Сигнатура `reprfunc` определяется в `Include ▶ cpython ▶ object.h` как имеющая один аргумент `PyObject* (self)`:

```
typedef PyObject *(*reprfunc)(PyObject *);
```

А `cellobject` реализует слот `tp_repr` при помощи функции `cell_repr`:

```
PyTypeObject PyCell_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "cell",
    sizeof(PyCellObject),
    0,
    (destructor)cell_dealloc,           /* tp_dealloc */
    0,                                /* tp_vectorcall_offset */
    0,                                /* tp_getattr */
    0,                                /* tp_setattr */
    0,                                /* tp_as_async */
    (reprfunc)cell_repr,               /* tp_repr */
    ...
};
```

Кроме слотов `PyTypeObject`, обозначаемых префиксом `tp_`, существуют и другие определения слотов типов:

СЛОТ ТИПА	ПРЕФИКС
PyNumberMethods	nb_
PySequenceMethods	sq_

СЛОТ ТИПА	ПРЕФИКС
PyMappingMethods	mp_
PyAsyncMethods	am_
PyBufferProcs	bf_

Каждому слоту типа назначается уникальный номер, определяемый в `Include > typeslots.h`. При обращении к слоту типа в объекте следует использовать эти константы.

Например, `tp_repr` имеет постоянную позицию 66, а константа `Py_tp_repr` всегда соответствует позиции слота типа. Эти константы полезны для проверки того, реализует ли объект конкретную функцию слота типа.

Работа с типами в С

В модулях расширения С и коде CPython вам придется часто работать с типом `PyObject*`.

Например, при выполнении `x[n]` для объекта, поддерживающего индексирование (такого, как список или строка), будет вызвана функция `PyObject_GetItem()`, которая обратится к объекту `x` для определения того, как он должен индексироваться:

Objects > abstract.c, строка 146

```
PyObject *
PyObject_GetItem(PyObject *o, PyObject *key)
{
    PyMappingMethods *m;
    PySequenceMethods *ms;
    ...
```

`PyObject_GetItem()` подходит как для типов отображений (например, словарей), так и для типов последовательностей (например, списков и кортежей).

Если экземпляр `o` содержит методы последовательностей, то `o->ob_type->tp_as_sequence` дает значение `true`. Кроме того, если в экземпляре определена функция слота `sq_item`, предполагается, что он правильно реализует протокол последовательности.

Значение `key` вычисляется для проверки того, является ли оно целым числом, и элемент запрашивается из объекта последовательности вызовом `PySequence_GetItem()`:

```
ms = o->ob_type->tp_as_sequence;
if (ms && ms->sq_item) {
    if (PyIndex_Check(key)) {
        Py_ssize_t key_value;
        key_value = PyNumber_AsSsize_t(key, PyExc_IndexError);
        if (key_value == -1 && PyErr_Occurred())
            return NULL;
        return PySequence_GetItem(o, key_value);
    }
    else {
        return type_error("sequence index must "
                           "be integer, not '%.200s'", key);
    }
}
```

Словари свойств типов

Python поддерживает определение новых типов с ключевым словом `class`. Типы, определяемые пользователем, создаются вызовом `type_new()` в модуле объекта типа.

Типы, определяемые пользователем, содержат словарь свойств, для обращения к которому используется вызов `__dict__()`. Каждый раз, когда нестандартный класс обращается к свойству, реализация `__getattr__()` по умолчанию обращается к этому словарю. Методы класса, экземпляры метода, свойства класса, свойства экземпляра — все это находится в словаре.

СМ. ТАКЖЕ

У нестандартных типов есть много уровней, и они подробно документированы. О метаклассах можно написать целую книгу, но здесь мы ограничимся реализацией.

Если вы захотите больше узнать о метaprogramмировании, обращайтесь к статье «[Python Metaclasses](#)»¹ на сайте Real Python.

¹ <https://realpython.com/python-metaclasses/>.

`PyObject_GenericGetDict()` реализует логику получения экземпляра словаря для заданного объекта. `PyObject_GetAttr()` выполняет реализацию `__getattr__()` по умолчанию, а `PyObject_SetAttr()` реализует `__setattr__()`.

ТИПЫ BOOL И LONG

Тип `bool` — самая прямолинейная реализация встроенных типов. Он наследуется от `long` и содержит предопределенные константы `Py_True` и `Py_False`. Эти константы являются неизменяемыми экземплярами, создаваемыми при инстанцировании интерпретатора Python.

Внутри `Objects ▶ boolobject.c` находится вспомогательная функция для создания экземпляра `bool` из числа:

Objects ▶ boolobject.c, строка 28

```
PyObject *PyBool_FromLong(long ok)
{
    PyObject *result;

    if (ok)
        result = Py_True;
    else
        result = Py_False;
    Py_INCREF(result);
    return result;
}
```

Эта функция использует C для вычисления числового типа, чтобы присвоить `result` значение `Py_True` или `Py_False` и увеличить счетчик ссылок.

Числовые функции для `and`, `xor` и `or` реализованы, но операции сложения, вычитания и деления заблокированы из базового типа `long`, так как деление двух логических значений не имеет смысла.

Реализация `and` для значения `bool` построена так, что сначала проверяет, являются ли `a` и `b` логическими типами. Если условие не выполняется, значения преобразуются в числа и операция выполняется с двумя числами:

Objects ▶ boolobject.c, строка 61

```
static PyObject *
bool_and(PyObject *a, PyObject *b)
{
```

```
if (!PyBool_Check(a) || !PyBool_Check(b))
    return PyLong_Type.tp_as_number->nb_and(a, b);
return PyBool_FromLong((a == Py_True) & (b == Py_True));
}
```

Тип long

Тип `long` немного сложнее `bool`. При переходе от Python 2 к Python 3 CPython перестал поддерживать тип `int`, и основным целочисленным типом стал `long`.

Тип `long` в Python отличается тем, что он способен хранить значение вариативной (изменяемой) длины. Максимальная длина задается в скомпилированном двоичном файле.

Структура данных `long` в Python состоит из заголовка переменной `PyObject` и списка цифр. Список цифр `ob_digit` изначально состоит из одной цифры, но при инициализации расширяется:

Include > longintrepr.h, строка 85

```
struct _longobject {
    PyObject_VAR_HEAD
    digit ob_digit[1];
};
```

Например, числу 1 будет соответствовать `ob_digit [1]`, а числу 24 601 — `ob_digit [2, 4, 6, 0, 1]`.

Память для нового значения `long` выделяется вызовом `_PyLong_New()`. Эта функция получает фиксированную длину и проверяет, что она меньше `MAX_LONG_DIGITS`. Затем память для `ob_digit` выделяется заново в соответствии с длиной.

Чтобы преобразовать тип `long` языка C в тип `long` языка Python, C `long` преобразуется в список цифр, выделяется память для Python `long`, после чего задается значение каждой цифры.

Для чисел из одной цифры объект `long` инициализируется с `ob_digit`, уже имеющим длину 1. Затем значение задается без выделения памяти:

Objects > longobject.c, строка 297

```
PyObject *
PyLong_FromLong(long ival)
{
```

```
PyLongObject *v;
unsigned long abs_ival;
unsigned long t; /* unsigned, поэтому >> не распространяет бит знака */
int ndigits = 0;
int sign;

CHECK_SMALL_INT(ival);
...
/* Быстрая ветвь для целых чисел из одной цифры */
if (!(abs_ival >> PyLong_SHIFT)) {
    v = _PyLong_New(1);
    if (v) {
        Py_SIZE(v) = sign;
        v->ob_digit[0] = Py_SAFE_DOWNCASE(
            abs_ival, unsigned long, digit);
    }
    return (PyObject*)v;
}
...
/* Большие числа: цикл для определения количества цифр */
t = abs_ival;
while (t) {
    ++ndigits;
    t >>= PyLong_SHIFT;
}
v = _PyLong_New(ndigits);
if (v != NULL) {
    digit *p = v->ob_digit;
    Py_SIZE(v) = ndigits*sign;
    t = abs_ival;
    while (t) {
        *p++ = Py_SAFE_DOWNCASE(
            t & PyLong_MASK, unsigned long, digit);
        t >>= PyLong_SHIFT;
    }
}
return (PyObject *)v;
}
```

Преобразование числа с плавающей точкой `double` в Python `long` осуществляется вызовом `PyLong_FromDouble()`.

Другие функции реализации в `Objects ► longobject.c` содержат вспомогательные функции, например функцию `PyLong_FromUnicodeObject()` для преобразования строки Юникода в число.

Пример

Слот типа расширенного сравнения для long заполняется `long_richcompare()`. Эта функция является оберткой для `long_compare()`:

Objects › `longobject.c`, строка 3031

```
static PyObject *
long_richcompare(PyObject *self, PyObject *other, int op)
{
    Py_ssize_t result;
    CHECK_BINOP(self, other);
    if (self == other)
        result = 0;
    else
        result = long_compare((PyLongObject*)self, (PyLongObject*)other);
    Py_RETURN_RICHCOMPARE(result, 0, op);
}
```

Функция `long_compare()` сначала проверяет длину (количество цифр) двух переменных: `a` и `b`. Если длины совпадают, то она перебирает в цикле все цифры и проверяет, равны ли они друг другу.

`long_compare()` возвращает один из трех видов значений:

1. Если `a < b`, возвращается отрицательное число.
2. Если `a == b`, возвращается 0.
3. Если `a > b`, возвращается положительное число.

Например, при выполнении условия `1 == 5` результат будет равен `-4`. Для `5 == 1` результат равен `4`.

Можно добавить следующий блок кода перед макросом `Py_RETURN_RICHCOMPARE`, чтобы он возвращал `True`, если абсолютное значение результата ≤ 1 . Макрос `Py_ABS()` возвращает абсолютное целое значение со знаком:

```
if (op == Py_EQ) {
    if (Py_ABS(result) <= 1)
        Py_RETURN_TRUE;
    else
        Py_RETURN_FALSE;
}
Py_RETURN_RICHCOMPARE(result, 0, op);
```

После повторной компиляции Python вы увидите последствия изменений:

```
>>> 2 == 1
False
>>> 2 ~= 1
True
>>> 2 ~= 10
False
```

ТИП СТРОКИ ЮНИКОДА

Строки Юникода в Python устроены достаточно сложно. Впрочем, кроссплатформенные типы Юникода сложны на любой платформе.

Это связано с количеством поддерживаемых кодировок и разных конфигураций по умолчанию на платформах, поддерживаемых Python.

В Python 2 строковый тип хранился с использованием типа `char` языка C. Однобайтовый тип `char` позволял хранить любые символы ASCII (American Standard Code for Information Interchange) и использовался в программировании с 1970-х годов.

ASCII поддерживает не все языки и алфавиты мира. Кроме того, он не поддерживает многие расширенные наборы глифов (например, эмодзи).

Для решения подобных проблем в 1991 году Консорциумом Юникода была представлена стандартная система кодирования и база данных символов, известная как «Юникод». Современный стандарт Юникода включает символы всех письменных языков, а также расширенные наборы глифов и символов.

База данных символов Юникода (**UCD**, Unicode Character Database) версии 13.0 содержит 143 859 именованных символов (тогда как в ASCII их всего 128). Стандарт Юникода определяет эти символы в виде таблицы, называемой универсальным набором символов, или **UCS** (Universal Character Set). Каждый символ имеет уникальный идентификатор, называемый **кодовой точкой**.

Существует много разных кодировок, которые используют стандарт Юникода и преобразуют кодовую точку в двоичное значение.

Строки Юникода в Python поддерживают три варианта длины кодирования:

- 1-байтовая (8 бит)
- 2-байтовая (16 бит)
- 4-байтовая (32 бита)

В реализации кодировкам изменяемой длины соответствуют следующие обозначения:

- 1-байтовый Py_UCS1, хранится как 8-битный тип `int` без знака `uint8_t`.
- 2-байтовый Py_UCS2, хранится как 16-битный тип `int` без знака `uint16_t`.
- 4-байтовый Py_UCS4, хранится как 32-битный тип `int` без знака `uint32_t`.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к строкам.

ФАЙЛ	НАЗНАЧЕНИЕ
Include ▶ unicodeobject.h	Определение объекта строки Юникода
Include ▶ cpython/unicodeobject.h	Определение объекта строки Юникода
Objects ▶ unicodeobject.c	Реализация объекта строки Юникода
Lib ▶ encodings	Пакет encodings со всеми возможными кодировками
Lib ▶ codecs.py	Модуль codecs
Modules ▶ _codecsmodule.c	Расширения С модуля codecs; реализация кодировок с привязкой к ОС
Modules ▶ _codecs	Реализация для многих альтернативных кодировок

Обработка кодовых точек Юникода

CPython не содержит копии UCD и не обновляется при добавлении новых алфавитов и символов в стандарт Юникода.

Для строк Юникода в CPython важны только кодировки. Операционная система берет на себя задачу представления кодовых точек в правильных наборах символов.

Стандарт Юникода включает UCD и регулярно обновляется новыми наборами символов, эмодзи и отдельными символами. Операционные системы получают обновления в Юникоде и обновляют свое программное обеспечение. Обновления включают новые кодовые точки UCD и поддерживают различные кодировки Юникода. UCD разбивается на разделы, называемые **кодовыми блоками**.

Таблицы символов Юникода публикуются на официальном веб-сайте¹.

Другим центром поддержки Юникода является веб-браузер. Браузеры декодируют двоичные данные HTML, основываясь на кодировке, указанной в HTTP-заголовках. Если вы используете CPython как веб-сервер, то ваши кодировки Юникода должны совпадать с заголовками HTTP, отправляемыми пользователям.

UTF-8 и UTF-16

Две самые популярные кодировки:

- **UTF-8** – 8-битная кодировка символов, поддерживающая все возможные символы UCD с кодовыми точками от 1 до 4 байтов.
- **UTF-16** – 16-битная кодировка символов, похожая на UTF-8, но несочетимая с 7- и 8-битовыми кодировками (такими, как ASCII).

Среди всех кодировок Юникода самой популярной является UTF-8.

Во всех кодировках Юникода кодовые точки могут представляться в шестнадцатеричной сокращенной записи. Вот несколько примеров:

- U+00F7 для знака деления ('÷');
- U+0107 для латинской строчной буквы с с диакритическим знаком ('ć').

В Python кодовые пункты Юникода могут кодироваться непосредственно в программном коде с префиксом \u и шестнадцатеричным значением кодовой точки:

```
>>> print("\u0107")
ć
```

¹ <https://unicode.org/charts/>.

CPython не пытается преобразовать такие данные в полную форму, так что если вы попробуете использовать запись \u107, будет выдано следующее исключение:

```
print("\u107")
  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode
  bytes in position 0-4: truncated \uXXXX escape
```

Как XML, так и HTML поддерживают кодовые точки Юникода в специальной записи &#val;, где val — десятичное значение кодовой точки. Если вы хотите закодировать кодовые точки Юникода в XML или HTML, используйте обработчик ошибок `xmlcharrefreplace` в методе `.encode()`:

```
>>> "\u0107".encode('ascii', 'xmlcharrefreplace')
b'&#263;'
```

Вывод будет содержать кодовые точки в экранированном представлении HTML или XML. Все современные браузеры декодируют эту последовательность в правильный символ.

Совместимость с ASCII

Если вы работаете с текстом, закодированным в ASCII, важно понимать, чем UTF-8 отличается от UTF-16. Главным преимуществом UTF-8 является совместимость с текстом, закодированным в 7-битной кодировке ASCII.

Первые 128 кодовых точек стандарта Юникод представляют существующие 128 символов стандарта ASCII. Например, латинская буква "а" является 97-м символом ASCII и 97-м символом Юникода. Десятичное значение 97 эквивалентно 61 в шестнадцатеричной системе, так что букве "а" будет соответствовать кодовая точка Юникода U+0061.

В REPL можно создать двоичный код для буквы "а":

```
>>> letter_a = b'a'
>>> letter_a.decode('utf8')
'a'
```

Он правильно декодируется в UTF-8.

UTF-16 работает с кодовыми пунктами, содержащими от 2 до 4 байт. 1-байтовое представление буквы "а" декодироваться не будет:

```
>>> letter_a.decode('utf16')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-16-le' codec can't decode
byte 0x61 in position 0: truncated data
```

Важно учитывать этот факт при выборе механизма кодирования. Если вам понадобится импортировать данные, закодированные в ASCII, UTF-8 будет более надежным вариантом.

Широкие символы

Если вы работаете с входной строкой Юникода в неизвестной кодировке из исходного кода CPython, используйте тип C `wchar_t`.

`wchar_t` — стандарт C для строк в многобайтовой кодировке, хорошо подходящий для хранения строк Юникода в памяти. После PEP 393 тип `wchar_t` был выбран как формат хранения Юникода. Строковый объект Юникода предоставляет `PyUnicode_FromWideChar()` — вспомогательную функцию, которая преобразует константу `wchar_t` в объект строки.

Например, функция `pymain_run_command()`, используемая `python -c`, преобразует аргумент `-c` в строку Юникода:

Modules › main.c, строка 226

```
static int
 pymain_run_command(wchar_t *command, PyCompilerFlags *cf)
{
    PyObject *unicode, *bytes;
    int ret;

    unicode = PyUnicode_FromWideChar(command, -1);
```

Маркеры последовательности байтов

При декодировании ввода (например, файла) CPython может определить порядок байтов по маркеру последовательности байтов (BOM, byte order mark). BOM — специальные символы, располагающиеся в начале потока байтов Юникода. Они сообщают получателю порядок байтов, с помощью которого хранятся данные.

Разные компьютерные системы могут кодировать данные с разным порядком байтов. Если вы используете неправильный порядок (даже с правильной

кодировкой), данные будут повреждены. При **прямом порядке** (Big-Endian) сначала передается старший, наиболее значимый байт. При **обратном порядке** (Little-Endian) на первое место ставится младший, наименее значимый байт.

Спецификация UTF-8 поддерживает маркеры ВОМ, но они ни на что не влияют. UTF-8 ВОМ может находиться в начале закодированной последовательности данных, представленной в виде `b'\xef\xbb\xbf'`; он сообщает Python, что поток данных с большой вероятностью закодирован в UTF-8. UTF-16 и UTF-32 поддерживают ВОМ для прямого и обратного порядка.

Порядок байтов по умолчанию в CPython задается глобальным значением `sys.byteorder`:

```
>>> import sys; print(sys.byteorder)
little
```

Пакет encodings

Пакет `encodings` в Lib ▶ `encodings` включает более сотни встроенных поддерживаемых кодировок для CPython. Каждый раз, когда метод `.encode()` или `.decode()` вызывается для обычной или байтовой строки, кодировка ищется в этом пакете.

Каждая кодировка определяется как отдельный модуль. Например, `ISO2022_JP` — популярная кодировка для японских систем электронной почты — объявляется в Lib ▶ `encodings iso2022_jp.py`.

Каждый модуль кодировки определяет функцию `getregentry()` и регистрирует следующие характеристики:

- уникальное имя;
- функции кодирования и декодирования из модуля кодека;
- классы инкрементального кодера и декодера;
- классы чтения и записи потоков данных.

Многие модули кодировок совместно используют одни и те же кодеки из модуля `codecs` или `_multibytecodec`. Некоторые модули кодировок используют отдельный модуль кодеков на С из файла Modules ▶ `cjckodecs`.

Например, модуль кодировки `ISO2022_JP` импортирует модуль расширения C, `_codecs_iso2022`, из файла Modules ▶ `cjckodecs _codecs_iso2022.c`:

```
import _codecs_iso2022, codecs
import _multibytecodec as mbc

codec = _codecs_iso2022.getcodec('iso2022_jp')

class Codec(codecs.Codec):
    encode = codec.encode
    decode = codec.decode

class IncrementalEncoder(mbc.MultibyteIncrementalEncoder,
                        codecs.IncrementalEncoder):
    codec = codec

class IncrementalDecoder(mbc.MultibyteIncrementalDecoder,
                        codecs.IncrementalDecoder):
    codec = codec
```

Пакет `encodings` также содержит модуль `Lib ▶ encodings ▶ aliases.py`, в котором есть словарь `aliases`. Этот словарь назначает кодировкам в реестре альтернативные имена. Например, `utf8`, `utf-8` и `u8` являются синонимами для кодировки `utf_8`.

Модуль `codecs`

Модуль `codecs` обеспечивает преобразование данных с заданной кодировкой. Функцию кодирования или декодирования для конкретной кодировки можно получить вызовом `getencoder()` или `getdecoder()` соответственно:

```
>>> iso2022_jp_encoder = codecs.getencoder('iso2022_jp')
>>> iso2022_jp_encoder('\u3072\u3068') # hi-to
(b'\x1b$B$R$H\x1b(B', 2)
```

Функция кодирования возвращает двоичный результат и количество байтов в выходных данных в виде кортежа. `codecs` также реализует встроенную функцию `open()` для открытия файловых дескрипторов операционной системы.

Реализации кодеков

Реализация объекта Юникода (`Objects ▶ unicodeobject.c`) содержит следующие методы кодирования.

КОДЕК	КОДИРОВЩИК
ascii	PyUnicode_EncodeASCII()
latin1	PyUnicode_EncodeLatin1()
UTF7	PyUnicode_EncodeUTF7()
UTF8	PyUnicode_EncodeUTF8()
UTF16	PyUnicode_EncodeUTF16()
UTF32	PyUnicode_EncodeUTF32()
unicode_escape	PyUnicode_EncodeUnicodeEscape()
raw_unicode_escape	PyUnicode_EncodeRawUnicodeEscape()

Методы декодирования имеют похожие имена, но `Encode` заменяется на `Decode`.

Реализация других кодировок размещается в `Modules ▶ _codecs`, чтобы не загромождать реализацию основного объекта строки Юникода. Кодеки `unicode_escape` и `raw_unicode_escape` используются во внутренней работе CPython.

Внутренние кодеки

CPython включает ряд внутренних кодировок, уникальных для CPython; они используются функциями стандартной библиотеки, а также при работе с генерированием исходного кода. Эти кодировки могут использоваться с любым текстовым вводом или выводом.

КОДЕК	НАЗНАЧЕНИЕ
idna	Реализует RFC 3490
mbcs	Кодирует в соответствии с кодовой страницей ANSI (только на Windows)
raw_unicode_escape	Преобразует необработанные строковые литералы в исходном коде Python в строку
string_escape	Преобразует строковые литералы в исходном коде Python в строку
undefined	Пробует применить установленную по умолчанию системную кодировку

КОДЕК	НАЗНАЧЕНИЕ
unicode_escape	Преобразует строки в исходном коде Python в литерал Юникода
unicode_internal	Возвращает внутреннее представление CPython

Также есть несколько чисто двоичных кодировок, которые должны использоваться с `codecs.encode()` или `codecs.decode()` с входной байтовой строкой, как в следующем примере:

```
>>> codecs.encode(b'hello world', 'base64')
b'aGVsbG8gd29ybGQ=\n'
```

Список чисто двоичных кодировок:

КОДЕК	СИНОНИМЫ	НАЗНАЧЕНИЕ
base64_codec	base64, base-64	Преобразование в MIME base64
bz2_codec	bz2	Сжатие строки с использованием bz2
hex_codec	hex	Преобразование в шестнадцатеричное представление с двумя цифрами на байт
quopri_codec	quoted-printable	Преобразование операнда в MIME Quoted Printable
rot_13	rot13	Возвращение результата подстановочного шифрования Цезаря (сдвиг 13)
uu_codec	uu	Преобразование с использованием uuencode
zlib_codec	zip, zlib	Сжатие с использованием gzip

Пример

Слот типа `tp_richcompare` заполняется `PyUnicode_RichCompare()` в `PyUnicode_Type`. Эта функция сравнивает строки и может адаптироваться для использования оператора `~`. Поведение, которое мы реализуем, — сравнение двух строк без учета регистра символов.

Сначала добавьте дополнительный блок `case`, который будет проверять строки в левой и правой части на двоичную эквивалентность:

Objects ▶ unicodeobject.c, строка 11361

```
PyObject *
PyUnicode_RichCompare(PyObject *left, PyObject *right, int op)
{
    ...
    if (left == right) {
        switch (op) {
            case Py_EQ:
            case Py_LE:
            >>> case Py_ALE:
            case Py_GE:
                /* Стока равна самой себе */
                Py_RETURN_TRUE;
    }
}
```

Затем добавьте новый блок `else if` для оператора `Py_ALE`. В нем будут выполняться следующие действия:

1. Преобразование левой строки в новую строку в верхнем регистре.
2. Преобразование правой строки в новую строку в верхнем регистре.
3. Сравнение двух строк.
4. Разыменование обеих временных строк для освобождения памяти.
5. Возвращение результата.

Код должен выглядеть примерно так:

```
else if (op == Py_EQ || op == Py_NE) {
    ...
}
/* Добавьте следующие строки */
else if (op == Py_ALE){
    PyObject* upper_left = case_operation(left, do_upper);
    PyObject* upper_right = case_operation(right, do_upper);
    result = unicode_compare_eq(upper_left, upper_right);
    Py_DECREF(upper_left);
    Py_DECREF(upper_right);
    return PyBool_FromLong(result);
}
```

После перекомпиляции сравнение строк без учета регистра должно давать следующие результаты в REPL:

```
>>> "hello" ~= "HEllo"
True
>>> "hello?" ~= "hello"
False
```

СЛОВАРИ

Словари — быстрый и гибкий способ сопоставления. Они используются разработчиками для хранения и сопоставления данных, а также объектами Python для хранения свойств и методов.

Словари Python также используются для локальных и глобальных переменных, для именованных аргументов и многих других сценариев использования. Словари Python компактны, так как в хеш-таблицах хранятся только сопоставляемые значения. Алгоритм хеширования, который является частью всех неизменяемых встроенных типов, работает быстро. Именно он обеспечивает высокую скорость словарей Python.

Хеширование

Все неизменяемые встроенные типы предоставляют функцию хеширования. Эта функция определяется в слоте типа `tp_hash` или (для пользовательских типов) при помощи специального метода `__hash__()`. Хеш имеет такой же размер, как указатель (64 бита для 64-разрядных систем, 32 бита для 32-разрядных систем), но он не представляет адреса своего значения в памяти.

Результат хеширования, полученный для любого объекта Python, не должен изменяться на протяжении всего жизненного цикла объекта. Хеши двух неизменяемых экземпляров, содержащих одинаковые значения, должны быть равны:

```
>>> "hello".__hash__() == ("hel" + "lo").__hash__()
True
```

Коллизии хешей недопустимы. Два объекта с разными значениями не должны иметь одинаковые хеши.

Некоторые хеши очень просты, например, у типа Python `long`:

```
>>> (401).__hash__()
401
```

Для более длинных значений хеши усложняются:

```
>>> (40112312438979898989).__hash__()
2212283795829936375
```

Многие встроенные типы используют модуль Python `pyhash.c`, который предоставляет следующие вспомогательные функции хеширования:

- **Байты:** `_Py_HashBytes(const void*, Py_ssize_t)`
- **Double:** `_Py_HashDouble(double)`
- **Указатели:** `_Py_HashPointer(void*)`

Например, строки Юникода используют `_Py_HashBytes()` для хеширования байтовых данных строки:

```
>>> ("hello").__hash__()
4894421526362833592
```

Пользовательские классы могут определять функцию хеширования, реализуя `__hash__()`. Вместо того чтобы писать специальную реализацию хеш-функции, добавьте в класс уникальное свойство. Позаботьтесь о том, чтобы свойство было неизменяемым, сделав его доступным только для чтения, а затем хешируйте его встроенной функцией `hash()`:

```
class User:
    def __init__(self, id: int, name: str, address: str):
        self._id = id

    def __hash__(self):
        return hash(self._id)

    @property
    def id(self):
        return self._id
```

Теперь экземпляры этого класса могут хешироваться:

```
>>> bob = User(123884, "Bob Smith", "Townsville, QLD")
>>> hash(bob)
123884
```

Экземпляр может использоваться в качестве ключа словаря:

```
>>> sally = User(123823, "Sally Smith", "Cairns, QLD")
>>> near_reef = {bob: False, sally: True}
>>> near_reef[bob]
False
```

Множества удаляют дублирующиеся хеши этого экземпляра:

```
>>> {bob, bob}
{<__main__.User object at 0x10df244b0>}
```

Исходные файлы

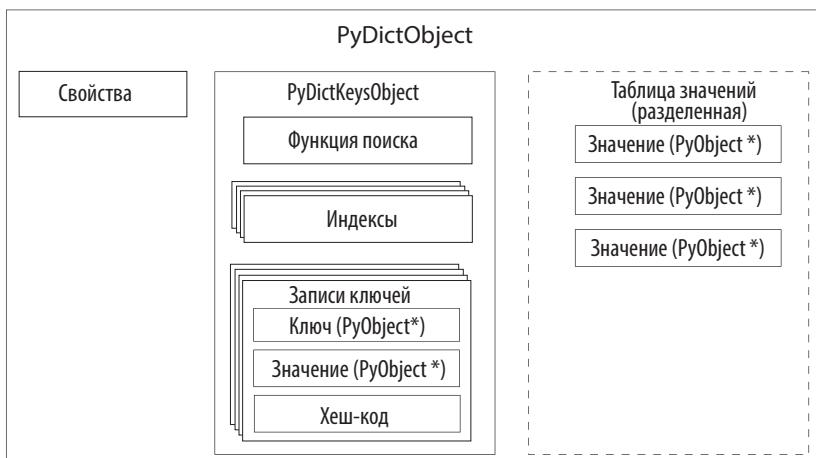
Ниже перечислены исходные файлы, относящиеся к словарям.

ФАЙЛ	НАЗНАЧЕНИЕ
Include ▶ dictobject.h	Определение API объекта словаря
Include ▶ cpython ▶ dictobject.h	Определение типов объекта словаря
Objects ▶ dictobject.c	Реализация объекта словаря
Objects ▶ dict-common.h	Определение элемента словаря и объектов ключей
Python ▶ pyhash.c	Внутренний алгоритм хеширования

Структура словаря

Объект словаря `PyDictObject` состоит из следующих элементов:

- Свойства объекта словаря: размер, метка версии, ключи и значения.
- Объект таблицы ключей словаря `PyDictKeysObject`, содержащий ключи и хеши всех элементов.



Объект `PyDictObject` содержит следующие свойства.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>ma_keys</code>	<code>PyDictKeysObject*</code>	Объект таблицы ключей словаря
<code>ma_used</code>	<code>Py_ssize_t</code>	Количество элементов в словаре
<code>ma_values</code>	<code>PyObject**</code>	Необязательный массив значений (см. примечание)
<code>ma_version_tag</code>	<code>uint64_t</code>	Номер версии словаря

ПРИМЕЧАНИЕ

Словари могут находиться в одном из двух состояний: разделенном или объединенном. В объединенном словаре указатели на значения словаря хранятся в таблице ключей.

В разделенном словаре значения хранятся в дополнительном свойстве `ma_values` в виде таблицы значений `PyObject*`.

Таблица ключей словаря `PyDictKeysObject` содержит следующие свойства.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>dk_entries</code>	<code>PyDictKeyEntry[]</code>	Массив записей ключей словаря
<code>dk_indices</code>	<code>char[]</code>	Хеш-таблица и соответствия для <code>dk_entries</code>
<code>dk_lookup</code>	<code>dict_lookup_func</code>	Функция поиска (см. следующий раздел)
<code>dk_nentries</code>	<code>Py_ssize_t</code>	Количество используемых элементов в таблице элементов
<code>dk_refcnt</code>	<code>Py_ssize_t</code>	Счетчик ссылок
<code>dk_size</code>	<code>Py_ssize_t</code>	Размер хеш-таблицы
<code>dk_usable</code>	<code>Py_ssize_t</code>	Количество пригодных для использования записей в таблице – если 0, размер словаря изменяется

Запись ключа словаря `PyDictKeyEntry` содержит следующие свойства.

ПОЛЕ	ТИП	НАЗНАЧЕНИЕ
<code>me_hash</code>	<code>Py_ssize_t</code>	Кешированный хеш <code>me_key</code>
<code>me_key</code>	<code>PyObject*</code> *	Указатель на объект ключа
<code>me_value</code>	<code>PyObject*</code> *	Указатель на объект значения (в объединенном состоянии)

Поиск

Для заданного объекта ключа существует обобщенная функция поиска `lookdict()`.

Поиск в словаре должен обеспечивать обработку трех возможных ситуаций:

1. В таблице ключей существует адрес ключа в памяти.
2. В таблице ключей существует хеш объекта.
3. Ключ не существует в словаре.

СМ. ТАКЖЕ

Функция поиска основана на материалах знаменитой книги Дональда Кнута «Искусство программирования» («The Art of Computer Programming»). Хеширование рассматривается в 4-м разделе 6-й главы.

Функция поиска работает по следующей схеме:

1. Получить хеш `ob`.
2. Найти хеш `ob` в ключах словаря и получить индекс `ix`.
3. Если значение `ix` пусто, вернуть `DKIX_EMPTY` (не найдено).
4. Получить запись ключа `ep` для заданного индекса.
5. Если значения ключей совпадают, то `ob` указывает на то же значение ключа. Вернуть результат.
6. Если хеши ключей совпадают, потому что объект `ob` разрешает то же значение хеша, что и `ep->me_mash`, вернуть результат.

ПРИМЕЧАНИЕ

`lookupdict()` – одна из нескольких **hot**-функций в исходном коде CPython:

Атрибут **hot** сообщает компилятору, что функция является горячей точкой скомпилированной программы. Такая функция оптимизируется более агрессивно, и на многих целевых платформах она помещается в специальный подраздел раздела `text`, чтобы все **hot**-функции располагались поблизости друг от друга, повышая эффективность пространственной локальности данных.

– Документация GCC, «Common Function Attributes»

Данная возможность характерна для компиляторов GNU C, но при компиляции с PGO¹ эта функция с большой вероятностью будет оптимизирована компилятором автоматически.

ВЫВОДЫ

Мы рассмотрели реализацию некоторых встроенных типов. Теперь вы готовы к исследованию других типов.

При изучении классов Python важно помнить, что существуют встроенные типы, написанные на C, и классы на Python или C, наследующиеся от этих типов.

В некоторых библиотеках типы тоже написаны на C (вместо того, чтобы наследоваться от встроенных). Один из примеров – NumPy, библиотека для работы с числовыми массивами. Тип `ndarray` написан на C, он чрезвычайно эффективен и производителен. В следующей главе мы рассмотрим классы и функции, определенные в стандартной библиотеке.

¹ Профильная оптимизация. — Примеч. пер.

Стандартная библиотека

Python всегда поставлялся с «батарейками в комплекте». Это означает, что стандартный дистрибутив CPython содержит библиотеки для работы с файлами, потоками, сетями, веб-сайтами, музыкой, клавиатурами, экранами, текстом и широким спектром утилит.

Некоторые «батарейки», поставляемые с CPython, можно сравнить с батарейками АА, которые подойдут почти для любого случая, как, например, модуль `collections` и модуль `sys`. Но другие больше похожи на маленькие батарейки для часов: никогда не знаешь, когда они могут понадобиться.

Стандартная библиотека CPython включает два типа модулей:

1. Модули, написанные на чистом Python.
2. Модули, написанные на С с обертками Python.

В этой главе будут рассмотрены оба типа.

МОДУЛИ PYTHON

Все модули, написанные на чистом Python, находятся в каталоге `Lib` исходного кода. Некоторые крупные модули могут содержать подмодули в подкаталогах (например, модуль `email`).

Например, существует такой простой модуль, как `colorsys`. Возможно, вы с ним еще не сталкивались. Он содержит всего сто строк Python-кода и служебные функции для преобразования цветовых шкал.

При установке дистрибутива Python из исходного кода модули стандартной библиотеки копируются из папки `Lib` в папку дистрибутива. Эта папка всегда

является частью пути при запуске Python, что позволяет вам импортировать модули, не беспокоясь о том, где они располагаются.

Импортирование и использование `colorsys` может выглядеть так:

```
>>> import colorsys
>>> colorsys
<module 'colorsys' from '/usr/shared/lib/python3.7/colorsys.py'>

>>> colorsys.rgb_to_hls(255,0,0)
(0.0, 127.5, -1.007905138339921)
```

Исходный код `rgb_to_hls()` находится в `Lib ▶ colorsys.py`:

```
# HLS: тон, яркость, насыщенность
# H: позиция в спектре
# L: яркость цвета
# S: насыщенность цвета

def rgb_to_hls(r, g, b):
    maxc = max(r, g, b)
    minc = min(r, g, b)
    # XXX Can optimize (maxc+minc) and (maxc-minc)
    l = (minc+maxc)/2.0
    if minc == maxc:
        return 0.0, 1, 0.0
    if l <= 0.5:
        s = (maxc-minc) / (maxc+minc)
    else:
        s = (maxc-minc) / (2.0-maxc-minc)
    rc = (maxc-r) / (maxc-minc)
    gc = (maxc-g) / (maxc-minc)
    bc = (maxc-b) / (maxc-minc)
    if r == maxc:
        h = bc-gc
    elif g == maxc:
        h = 2.0+rc-bc
    else:
        h = 4.0+gc-rc
    h = (h/6.0) % 1.0
    return h, l, s
```

В этой функции нет ничего особенного — вполне стандартный код Python. Так можно сказать обо всех модулях стандартной библиотеки, написанных на чистом Python. Все они содержат обычный код Python, хорошо структурированы и не особенно сложны для понимания.

Возможно, среди них вы даже найдете те, которые можно улучшить, или заметите ошибки. В таком случае можно внести изменения и включить их в дистрибутив Python. Рассмотрим эту возможность ближе к концу книги.

МОДУЛИ PYTHON И С

Остальные модули написаны на С или сочетают Python и С. Для компонента Python исходный код хранится в каталоге `Lib`, а для С — в каталоге `Modules`. Существуют два исключения:

1. Модуль `sys` находится в файле `Python ▶ sysmodule.c`.
2. Модуль `__builtins__` находится в файле `Python ▶ bltinmodule.c`.

Так как модуль `sys` тесно связан с интерпретатором и внутренними механизмами CPython, он хранится в каталоге `Python`. Он также помечен как «деталь реализации» CPython и отсутствует в других дистрибутивах.

Python выполняет `import * from __builtins__` при создании экземпляра интерпретатора, так что все встроенные функции, такие как `print()`, `chr()`, `format()` и т. д., — находятся в `Python ▶ bltinmodule.c`.

Вероятно, ваше изучение Python началось именно со встроенной функции `print()`.

Так что же происходит, когда вы вызываете `print("Hello, World")?`

Общая схема выглядит так:

1. Компилятор преобразует аргумент `"Hello, World"` из строковой константы в `PyUnicodeObject`.
2. `builtin_print()` выполняется с одним аргументом и NULL `kwnames`.
3. Переменной `file` присваивается `PyId_stdout`, системный обработчик стандартного вывода `stdout`.
4. Каждый аргумент передается `file`.
5. В `file` отправляется символ новой строки (`\n`).

Вот как это работает:

Python ▶ bltinmodule.c, строка 1828

```

static PyObject *
builtin_print(PyObject *self, PyObject *const *args,
             Py_ssize_t nargs, PyObject *kwnames)
{
    ...
    if (file == NULL || file == Py_None) {
        file = _PySys_GetObjectId(&PyId_stdout);
        ...
    }
    ...
    for (i = 0; i < nargs; i++) {
        if (i > 0) {
            if (sep == NULL)
                err = PyFile_WriteString(" ", file);
            else
                err = PyFile_WriteObject(sep, file,
                                         Py_PRINT_RAW);
            if (err)
                return NULL;
        }
        err = PyFile_WriteObject(args[i], file, Py_PRINT_RAW);
        if (err)
            return NULL;
    }
    if (end == NULL)
        err = PyFile_WriteString("\n", file);
    else
        err = PyFile_WriteObject(end, file, Py_PRINT_RAW);
    ...
    Py_RETURN_NONE;
}

```

Содержимое некоторых модулей, написанных на C, открывает доступ к функциям операционной системы. Так как исходный код CPython должен компилироваться для macOS, Windows, Linux и других операционных систем семейства *nix, существуют некоторые особые случаи.

Хорошим примером служит модуль `time`. Способ хранения времени в операционной системе Windows принципиально отличается от Linux и macOS. Это одна из причин, по которым точность функций часов зависит от операционной системы¹.

¹ https://docs.python.org/3/library/time.html#time.clock_gettime_ns.

В файле `Modules/timemodule.c` функции времени ОС для систем семейства Unix импортируются из `<sys/times.h>`:

```
#ifdef HAVE_SYS_TIMES_H
#include <sys/times.h>
#endif
...
#ifndef MS_WINDOWS
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include "pythread.h"
#endif /* MS_WINDOWS */
...
```

Далее в файле функция `time_process_time_ns()` объявляется как обертка для `_PyTime_GetProcessTimeWithInfo()`:

```
static PyObject *
time_process_time_ns(PyObject *self, PyObject *unused)
{
    _PyTime_t t;
    if (_PyTime_GetProcessTimeWithInfo(&t, NULL) < 0) {
        return NULL;
    }
    return _PyTime_AsNanosecondsObject(t);
}
```

`_PyTime_GetProcessTimeWithInfo()` реализуется в исходном коде несколькими разными способами, но лишь некоторые части компилируются в двоичный файл для модуля в зависимости от операционной системы. В системах Windows вызывается `GetProcessTimes()`, а в системах Unix — `clock_gettime()`.

Существуют и другие модули, имеющие несколько реализаций для одного API: модуль `threading`¹, модуль файловой системы и сетевые модули. Так как операционные системы обладают разным поведением, исходный код CPython реализует одно и то же поведение, настолько хорошо, насколько это возможно, и предоставляет целостный абстрагированный API.

¹ <https://realpython.com/intro-to-python-threading/>.

Набор тестов

CPython содержит надежный набор тестов для базового интерпретатора, инструментов, стандартной библиотеки и дистрибутивов для Windows, Linux и macOS. Они находятся в каталоге `Lib ▶ test` и написаны большей частью на Python.

Полный набор тестов находится в пакете Python, и его можно запустить с помощью только что скомпилированного интерпретатора Python.

ЗАПУСК НАБОРА ТЕСТОВ В WINDOWS

В Windows для запуска тестов используется скрипт `rt.bat` из папки PCBuild. В следующем примере тесты запускаются в быстром режиме для отладочной конфигурации на архитектуре x64:

```
> cd PCbuild
> rt.bat -q -d -x64

== CPython 3.9
== Windows-10-10.0.17134-SP0 little-endian
== cwd: C:\repos\cpython\build\test_python_2784
== CPU count: 2
== encodings: locale=cp1252, FS=utf-8
Run tests sequentially
0:00:00 [ 1/420] test_grammar
0:00:00 [ 2/420] test_OPCODE
0:00:00 [ 3/420] test_dict
0:00:00 [ 4/420] test_builtin
...
...
```

Чтобы выполнить набор регрессионных тестов для рабочей конфигурации, удалите флаг `-d` из командной строки.

ЗАПУСК НАБОРА ТЕСТОВ В LINUX ИЛИ MACOS

В Linux и macOS выполните команду `make test`, чтобы скомпилировать и запустить тесты:

```
$ make test
== CPython 3.9
== macOS-10.14.3-x86_64-i386-64bit little-endian
== cwd: /Users/anthonyshaw/cpython/build/test_python_23399
== CPU count: 4
== encodings: locale=UTF-8, FS=utf-8
0:00:00 load avg: 2.14 [ 1/420] test_opcodes passed
0:00:00 load avg: 2.14 [ 2/420] test_grammar passed
...
...
```

Также можно использовать путь к скомпилированному двоичному файлу `python` или `python.exe` с пакетом `test`:

```
$ ./python -m test
== CPython 3.9
== macOS-10.14.3-x86_64-i386-64bit little-endian
== cwd: /Users/anthonyshaw/cpython/build/test_python_23399
== CPU count: 4
== encodings: locale=UTF-8, FS=utf-8
0:00:00 load avg: 2.14 [ 1/420] test_opcodes passed
0:00:00 load avg: 2.14 [ 2/420] test_grammar passed
...
...
```

Дополнительные `make`-цели для тестирования:

ЦЕЛЬ	НАЗНАЧЕНИЕ
test	Выполнение основного набора регрессионных тестов
testall	Двукратное выполнение полного набора тестов: один раз без файлов <code>.rus</code> и один – с ними
quicktest	Выполнение набора более быстрых регрессионных тестов с исключением тестов, занимающих много времени
testuniversal	Выполнение набора тестов для обеих архитектур в универсальной сборке для OSX
coverage	Компиляция и запуск тестов с <code>gcov</code>
coverage-lcov	Создание HTML-отчетов о покрытии

ФЛАГИ ТЕСТИРОВАНИЯ

Для некоторых тестов нужно установить специальные флаги, иначе тесты пропускаются. Например, многим тестам IDLE требуется графический интерфейс (GUI).

Чтобы увидеть список наборов тестов в конфигурации, передайте флаг `--list-tests`:

```
$ ./python -m test --list-tests

test_grammar
test_OPCODES
test_dict
test_builtin
test_exceptions
...
```

ЗАПУСК КОНКРЕТНЫХ ТЕСТОВ

Чтобы запустить конкретные тесты, укажите набор тестов в первом аргументе.

Пример для Linux или macOS:

```
$ ./python -m test test_webbrowser

Run tests sequentially
0:00:00 load avg: 2.74 [1/1] test_webbrowser

== Tests result: SUCCESS ==

1 test OK.

Total duration: 117 ms
Tests result: SUCCESS
```

Пример для Windows:

```
> rt.bat -q -d -x64 test_webbrowser
```

Также вместе с результатом можно получить подробный список выполненных тестов. Для этого используется аргумент `-v`:

```
$ ./python -m test test_webbrowser -v

== CPython 3.9
== macOS-10.14.3-x86_64-i386-64bit little-endian
== cwd: /Users/anthonyshaw/cpython/build/test_python_24562
```

```
== CPU count: 4
== encodings: locale=UTF-8, FS=utf-8
Run tests sequentially
0:00:00 load avg: 2.36 [1/1] test_webbrowser
test_open (test.test_webbrowser.BackgroundBrowserCommandTest) ...ok
test_register (test.test_webbrowser.BrowserRegistrationTest) ...ok
test_register_default (test.test_webbrowser.BrowserRegistrationTest) ...ok
test_register_preferred (test.test_webbrowser.BrowserRegistrationTest) ...ok
test_open (test.test_webbrowser.ChromeCommandTest) ...ok
test_open_new (test.test_webbrowser.ChromeCommandTest) ...ok
...
test_open_with_autoraise_false (test.test_webbrowser.OperaCommandTest) ...ok
-----
Ran 34 tests in 0.056s

OK (skipped=2)

== Tests result: SUCCESS ==

1 test OK.

Total duration: 134 ms
Tests result: SUCCESS
```

Если вы захотите внести изменения в CPython, очень важно понимать, как использовать наборы тестов и проверять состояние скомпилированной версии. Прежде чем вносить изменения, запустите все тесты и убедитесь в том, что они проходят.

МОДУЛИ ТЕСТИРОВАНИЯ

Расширения С и модули Python можно импортировать и протестировать при помощи модуля `unittest`. Тесты группируются по модулям или пакетам.

Например, тесты для типа строки Юникода в Python находятся в файле `Lib ▶ test ▶ test_unicode.py`, а для пакета `asyncio` — в `Lib ▶ test ▶ test_asyncio`.

СМ. ТАКЖЕ

Если у вас еще нет опыта работы с `unittest` или тестированием в Python, обратитесь к статье «*Getting Started With Testing in Python*»¹ на сайте Real Python.

¹ <https://realpython.com/python-testing/>.

Перед вами фрагмент класса `UnicodeTest`:

```
class UnicodeTest(string_tests.CommonTest,
                  string_tests.MixinStrUnicodeUserStringTest,
                  string_tests.MixinStrUnicodeTest,
                  unittest.TestCase):
    ...
    def test_casifold(self):
        self.assertEqual('hello'.casifold(), 'hello')
        self.assertEqual('hELlo'.casifold(), 'hello')
        self.assertEqual('ß'.casifold(), 'ss')
        self.assertEqual('fi'.casifold(), 'fi')
```

Оператор «почти равно», реализованный нами для строк Юникода в предыдущих главах, можно расширить и добавить новый тестовый метод в класс `UnicodeTest`:

```
def test_almost_equals(self):
    self.assertTrue('hello' ~=' hello')
    self.assertTrue('hELlo' ~=' hello')
    self.assertFalse('hELlo!' ~=' hello')
```

Этот конкретный модуль тестирования можно запустить на Windows:

```
> rt.bat -q -d -x64 test_unicode
```

Также можно выполнить его на macOS или Linux:

```
$ ./python -m test test_unicode -v
```

ВСПОМОГАТЕЛЬНЫЕ СРЕДСТВА ТЕСТИРОВАНИЯ

Импортируя модуль `test.support.script_helper`, вы получаете доступ к некоторым вспомогательным функциям для тестирования среды выполнения Python:

- Функция `assert_python_ok(*args, **env_vars)` выполняет процесс Python с заданными аргументами и возвращает кортеж (*код возврата, stdout, stderr*).
- Функция `assert_python_failure(*args, **env_vars)` похожа на `assert_python_ok()`, но проверяет, завершилось ли выполнение неудачей.
- Функция `make_script(script_dir, script_basename, source)` создает в `script_dir` скрипт с именем `script_basename` и исходным кодом `source`,

после чего возвращает путь к этому скрипту. Ее удобно использовать в сочетании с `assert_python_ok()` или `assert_python_failure()`.

Если вы хотите создать тест, который будет пропускаться, если модуль не был построен, можно воспользоваться вспомогательной функцией `test.support.import_module()`. Она поднимет исключение `SkipTest` и даст сигнал программе запуска тестов пропустить этот пакет. Пример:

```
import test.support

_multiprocessing = test.support.import_module('_multiprocessing')

# Ваши тесты...
```

ВЫВОДЫ

История средств регрессионного тестирования Python насчитывает более двух десятилетий разработки тестов для необычных, редких случаев, а также для исправления ошибок и введения новой функциональности. За ее пределами остается большая часть стандартной библиотеки CPython с минимумом тестов или вообще без них. Если вы хотите поучаствовать в проекте CPython, то написание или расширение юнит-тестов может стать отличным началом.

Если вы захотите изменить любую часть CPython или добавить функциональность, то в качестве обновления вы можете написать свои тесты или расширить существующие.

Отладка

В CPython есть встроенный отладчик `pdb` для отладки приложений. `pdb` отлично подходит для отладки сбоев в приложениях Python, а также для написания тестов и просмотра локальных переменных.

Но когда дело доходит до CPython, нужен второй отладчик — с поддержкой С.

В этой главе вы узнаете:

- как подключить отладчик к интерпретатору CPython;
- как при помощи отладчика заглянуть внутрь работающего процесса CPython.

Существуют две разновидности отладчиков: консольные и визуальные. **Консольные отладчики** (такие, как `pdb`) предоставляют командную строку и специальные команды для анализа переменных и стека. **Визуальные отладчики** — приложения с графическим интерфейсом.

В этой главе рассматриваются следующие отладчики:

ОТЛАДЧИК	ТИП	ПЛАТФОРМА
LLDB	Консольный	macOS
GDB	Консольный	Linux
Отладчик Visual Studio	Визуальный	Windows
Отладчик CLion	Визуальный	Windows, macOS, Linux

ОБРАБОТЧИК СБОЕВ

Если в языке С приложение пытается читать или записывать данные в область памяти, в которую это делать не разрешено, происходит ошибка сегментации.

Выполняемый процесс немедленно останавливается, чтобы избежать сбоев в других приложениях. Ошибки сегментации также могут происходить при попытке чтения из памяти, не содержащей данных, или по некорректному указателю.

Когда в CPython происходит сбой сегментации, вы практически не получаете информации о произошедшем:

```
[1] 63476 segmentation fault ./python portscanner.py
```

В CPython встроен обработчик сбоев. Если запустить CPython с ключом `-X faulthandler` или `-X dev`, то вместо вывода системного сообщения об ошибке сегментации обработчик выведет список выполняемых потоков и трассировку стека Python до места возникновения ошибки:

```
Fatal Python error: Segmentation fault
Thread 0x0000000119021dc0 (most recent call first):
  File "/cpython/Lib/threading.py", line 1039 in _wait_for_tstate_lock
  File "/cpython/Lib/threading.py", line 1023 in join
  File "/cpython/portscanner.py", line 26 in main
  File "/cpython/portscanner.py", line 32 in <module>
[1] 63540 segmentation fault ./python -X dev portscanner.py
```

Данная возможность также пригодится при разработке и тестировании расширений С для CPython.

КОМПИЛЯЦИЯ ПОДДЕРЖКИ ОТЛАДКИ

Чтобы получить осмысленную информацию от отладчика, необходимо скомпилировать CPython с отладочными данными о символьических именах. Без символьических имен трассировка стека в сеансе отладки не будет содержать правильные имена функций, переменных или файлов.

Windows

Выполняя действия, описанные в разделе Windows главы «Компиляция CPython», убедитесь в том, что программа скомпилирована в отладочной конфигурации для получения отладочных символьических имен:

```
> build.bat -p x64 -c Debug
```

Не забудьте, что в отладочной конфигурации создается исполняемый файл `python_d.exe`, который должен использоваться для отладки.

macOS или Linux

В главе «Компиляция CPython» предлагалось запустить скрипт `./configure` с флагом `--with-pydebug`. Если флаг не был установлен, вернитесь и сделайте это. На этот раз будет сгенерирован правильный исполняемый файл и символические имена для отладки.

LLDB ДЛЯ MACOS

Отладчик LLDB входит в состав средств разработчика Xcode, поэтому он уже должен быть установлен на вашем компьютере.

Запустите LLDB и загрузите скомпилированный двоичный файл CPython как цель отладки:

```
$ lldb ./python.exe
(lldb) target create "./python.exe"
Current executable set to './python.exe' (x86_64).
```

Открывается приглашение, в котором можно вводить команды отладки.

Создание точек останова

Чтобы создать точку останова, выполните команду `break set` с указанием файла (пути относительно корня) и номера строки:

```
(lldb) break set --file Objects/floatobject.c --line 532
Breakpoint 1: where = python.exe'float_richcompare + 2276 at
floatobject.c:532:26, address = 0x000000010006a974
```

ПРИМЕЧАНИЕ

Также поддерживается сокращенная запись для установки точек останова: `(lldb) b Objects/floatobject.c:532`

Командой `break set` можно добавить несколько точек останова. Чтобы получить список текущих точек, введите команду `break list`:

```
(lldb) break list
Current breakpoints:
1: file = 'Objects/floatobject.c', line = 532, exact_match = 0, locations = 1
```

```
1.1: where = python.exe'float_richcompare + 2276 at floatobject.c:532:26,
      address = python.exe[...], unresolved, hit count = 0
```

Запуск CPython

Чтобы запустить CPython, используйте команду `process launch --` с параметрами командной строки, которые обычно используются для Python.

Чтобы запустить Python со строкой (например, `python -c "print(1)"`), введите следующую команду:

```
(lldb) process launch -- -c "print(1)"
```

Для запуска Python со скриптом используется команда:

```
(lldb) process launch -- my_script.py
```

Подключение к работающему интерпретатору CPython

Если у вас уже запущен интерпретатор CPython, вы можете подключить к нему отладчик.

В сеансе LLDB выполните команду `process attach --pid` с идентификатором процесса:

```
(lldb) process attach --pid 123
```

Идентификатор процесса можно получить в Activity Monitor или при помощи функции `os.getpid()` в Python.

Любые точки останова, установленные до или после этой точки, остановят процесс.

Работа с точками останова

Чтобы увидеть, как обрабатывается точка останова, установите ее на функции `float_richcompare()` из файла `Objects ▶ floatobject.c`.

Запустите процесс и сравните два значения с плавающей точкой оператором «почти равно», разработкой которого мы занимались в предыдущих главах:

```
(lldb) process launch -- -c "1.0~=1.1"
Process 64421 launched: '/cpython/python.exe' (x86_64)
Process 64421 stopped
* thread #1, queue = '...', stop reason = breakpoint 1.1
  frame #0: 0x000000010006a974 python.exe'float_richcompare(v=1.0,
w=1.1, op=6) at floatobject.c:532:26
  529 break;
  530 case Py_ALE: {
  531     double diff = fabs(i - j);
-> 532     const double rel_tol = 1e-9;
  533     const double abs_tol = 0.1;
  534     r = (((diff <= fabs(rel_tol * j)) ||
Target 0: (python.exe) stopped.
```

LLDB снова выдает приглашение. Для просмотра локальных переменных используется команда `v`:

```
(lldb) v
(PyObject *) v = 0x000000010111b370 1.0
(PyObject *) w = 0x000000010111b340 1.1
(int) op = 6
(double) i = 1
(double) j = 1.1000000000000001
(int) r = 0
(double) diff = 0.1000000000000009
(const double) rel_tol = 2.1256294105914498E-314
(const double) abs_tol = 0
```

Чтобы вычислить результат выражения на C, примените команду `expr` с любой допустимой командой C. При этом можно использовать переменные из области видимости. Например, чтобы вызвать `fabs(rel_tol)` и преобразовать результат в `double`, выполните следующую команду:

```
(lldb) expr (double)fabs(rel_tol)
(double) $1 = 2.1256294105914498E-314
```

Эта команда выводит переменную и присваивает ее идентификатору (`$1`). Вы можете повторно использовать этот идентификатор как временную переменную.

Также можно проанализировать экземпляры `PyObject`:

```
(lldb) expr v->ob_type->tp_name
(const char *) $6 = 0x000000010034fc26 "float"
```

Для получения трассировки из точки останова используется команда `bt`:

```
(lldb) bt
* thread #1, queue = '...', stop reason = breakpoint 1.1
* frame #0: ...
  python.exe'float_richcompare(...)' at floatobject.c:532:26
frame #1: ...
  python.exe'do_richcompare(...)' at object.c:796:15
frame #2: ...
  python.exe'PyObject_RichCompare(...)' at object.c:846:21
frame #3: ...
  python.exe'cmp_outcome(...)' at ceval.c:4998:16
```

Для выполнения с заходом в функции используйте команду `step` или `s`.

Для выполнения с обходом функций или перехода к следующему выражению используется команда `next` или `n`.

Чтобы продолжить выполнение, используйте команду `continue` или `c`.

Чтобы завершить сеанс, введите команду `quit` или `q`.

СМ. ТАКЖЕ

В учебной документации LLDB¹ приведен намного более подробный список команд.

Расширение cpython_lldb

LLDB поддерживает расширения, написанные на Python. Расширение с открытым кодом `cpython_lldb` выводит дополнительную информацию в сеансе LLDB для объектов CPython.

Чтобы установить его, выполните следующие команды:

```
$ mkdir -p ~/.lldb
$ cd ~/.lldb && git clone https://github.com/malor/cpython-lldb
$ echo "command script import ~/.lldb/cpython-lldb/cpython_lldb.py" \
>> ~/.lldbinit
$ chmod +x ~/.lldbinit
```

¹ <https://lldb.llvm.org/use/tutorial.html>.

Теперь при выводе переменных в LLDB справа будет дополнительная информация — числовое значение для целых чисел и чисел с плавающей точкой или текст для строк Юникода. На консоли LLDB также появляется дополнительная команда `py-bt`, которая выводит трассировку стека для кадров Python.

GDB

GDB — популярный отладчик для приложений C/C++, написанных на платформах Linux. Он также часто используется в команде ключевых разработчиков CPython.

При компиляции CPython генерируется скрипт `python-gdb.py`. Не запускайте его напрямую. GDB обнаружит и запустит его автоматически после завершения настройки.

Для настройки этого этапа отредактируйте файл `.gdbinit` в домашнем каталоге (`~/gdbinit`), добавив следующую строку, где `/path/to/checkout` — путь к извлечению рабочей версии cpython:

```
add-auto-load-safe-path /path/to/checkout
```

Чтобы запустить GDB, передайте в аргументе путь к скомпилиированному двоичному файлу CPython:

```
$ gdb ./python
```

GDB загружает информацию символьических имен для скомпилированного двоичного файла и открывает приглашение командной строки. GDB содержит набор встроенных команд, а расширения CPython добавляют к ним несколько дополнительных.

Создание точек останова

Чтобы поставить точку останова, используйте команду `b <файл>:<строка>`, прописав путь к исполняемому файлу:

```
(gdb) b Objects/floatobject.c:532
Breakpoint 1 at 0x10006a974: file Objects/floatobject.c, line 532.
```

Вы можете установить сколько угодно точек.

Запуск CPython

Чтобы запустить процесс, выполните команду `run` с аргументами для запуска интерпретатора Python.

Например, следующая команда передает при запуске строку:

```
(gdb) run -c "print(1)"
```

Чтобы запустить Python со скриптом, введите следующую команду:

```
(gdb) run my_script.py
```

Подключение к работающему интерпретатору CPython

Если у вас уже запущен интерпретатор CPython, вы можете подключить к нему отладчик.

В сеансе GDB выполните команду `attach` с идентификатором процесса:

```
(gdb) attach 123
```

Идентификатор процесса можно получить в Activity Monitor или при помощи функции `os.getpid()` в Python. Любые точки останова, установленные до или после этой точки, останавливают процесс.

Работа с точками останова

Когда GDB достигает точки останова, вы можете воспользоваться командой `print` или `p` для вывода переменной:

```
(gdb) p *(PyLongObject*)v
$1 = {ob_base = {ob_base = {ob_refcnt = 8, ob_type = ...}, ob_size = 1},
      ob_digit = {42}}
```

Для выполнения с заходом в функции используйте команду `step` или `s`.

Для выполнения с обходом функций используется `next` или `n`.

Расширение python-gdb

Расширение `python-gdb` загружает дополнительный набор команд в консоли GDB:

КОМАНДА	НАЗНАЧЕНИЕ
py-print	Поиск и вывод переменной Python
py-bt	Вывод трассировки стека Python
py-locals	Вывод результата locals()
py-up	Переход на один кадр Python вниз
py-down	Переход на один кадр Python вверх
py-list	Вывод исходного кода Python для текущего кадра

ОТЛАДЧИК VISUAL STUDIO

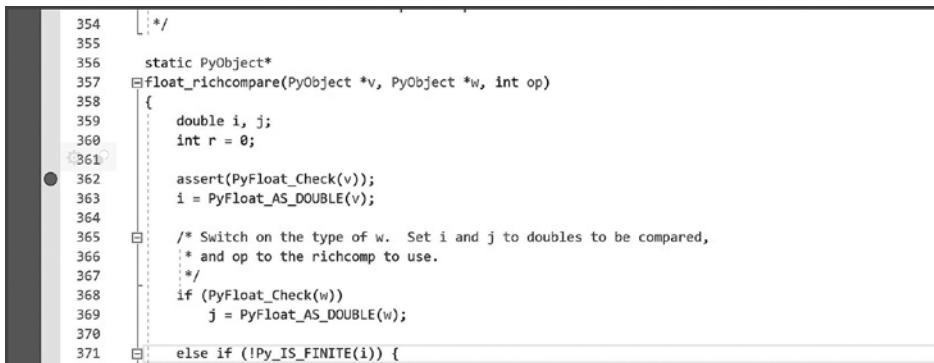
В поставку Microsoft Visual Studio входит визуальный отладчик. Это мощный инструмент, который поддерживает визуализацию стека кадров, список отслеживания, возможность вычисления выражений и т. д.

Чтобы использовать его, откройте Visual Studio и файл решения PCBuild ➤ pcbuild.sln.

Добавление точек останова

Чтобы добавить новую точку останова, перейдите к нужному файлу в окне решения, а затем щелкните по полю слева от номера строки.

Появившийся красный круг показывает, что в строке установлена точка останова:

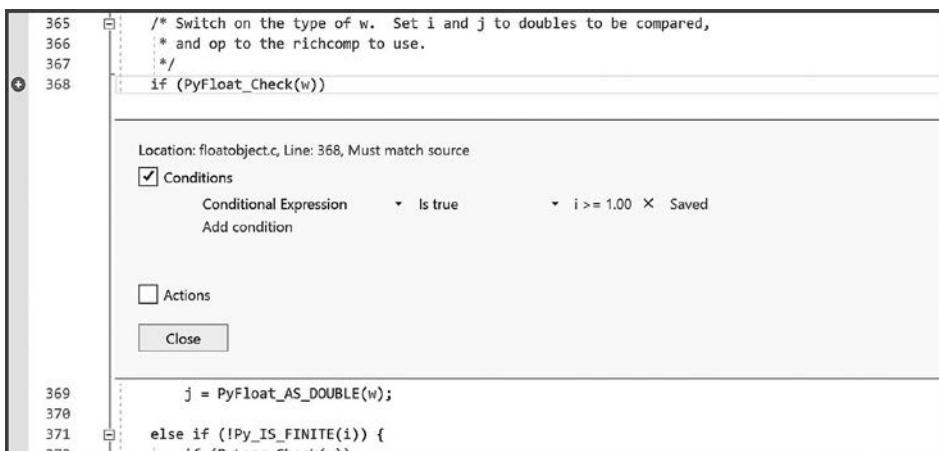


```

354     */
355
356     static PyObject*
357     float_richcompare(PyObject *v, PyObject *w, int op)
358     {
359         double i, j;
360         int r = 0;
361
362         assert(PyFloat_Check(v));
363         i = PyFloat_AS_DOUBLE(v);
364
365         /* Switch on the type of w. Set i and j to doubles to be compared,
366          * and op to the richcomp to use.
367          */
368         if (PyFloat_Check(w))
369             j = PyFloat_AS_DOUBLE(w);
370
371         else if (!Py_IS_FINITE(i)) {

```

Если навести указатель мыши на красный круг, появляется изображение шестеренки. Нажмите на него, чтобы настроить условные точки останова. Добавьте одно или несколько условных выражений, которые должны проверяться до срабатывания точки:



Запуск отладчика

Выберите в меню команду Debug ▶ Start Debugger или нажмите F5.

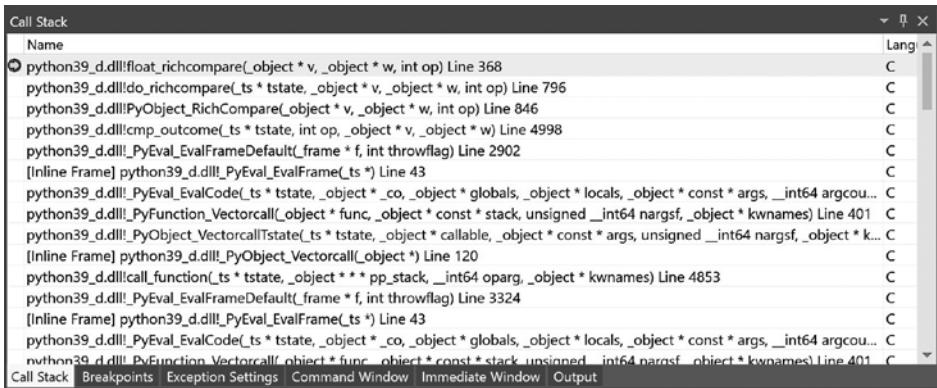
Visual Studio запускает новую среду выполнения Python и REPL.

Работа с точками останова

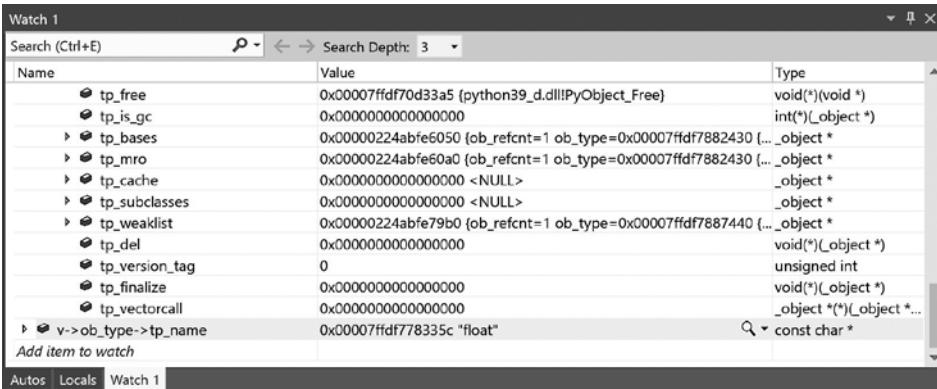
Когда программа достигает точки останова, вы получаете возможность выполнять код пошагово при помощи клавиш со стрелками или следующих горячих клавиш:

- **Выполнение с заходом в функции:** F11
- **Выполнение с обходом функций:** F10
- **Выход:** Shift + F11

Стек вызовов будет отображаться снизу. Выбирая кадры в стеке, можно изменять навигацию и просматривать переменные в других кадрах:



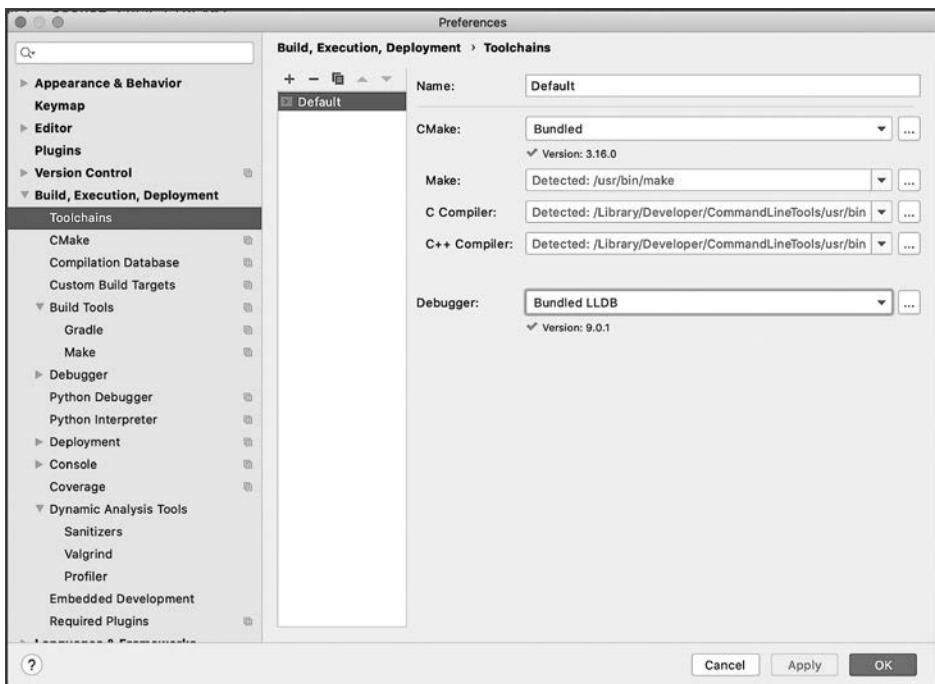
В редакторе кода можно выделить любую переменную или выражение, чтобы просмотреть их значение; также можно щелкнуть правой кнопкой мыши и выбрать Add Watch. Команда добавляет переменную в окно Watch, в котором для упрощения отладки можно быстро просмотреть, что хранит переменная:



ОТЛАДЧИК CLION

В поставку IDE CLion включен мощный визуальный отладчик. Он работает с LLDB в macOS и GDB в macOS, Windows и Linux.

Чтобы настроить его, выполните команду Preferences и выберите раздел Build, Execution, Deployment ▶ Toolchains:



В окне есть выпадающий список для выбора целевого отладчика. Выберите вариант, подходящий для вашей операционной системы:

- **macOS:** Bundled LLDB
- **Windows или Linux:** Bundled GDB

ВАЖНО

LLDB и GDB пользуются расширениями `cpython_lldb` и `python-gdb` соответственно. О том, как установить и включить эти расширения, рассказано ранее в этой главе, в разделах о LLDB и GDB.

Отладка make-приложений

В CLion 2020.2 можно скомпилировать и отладить любой проект на базе make-файлов, включая CPython.

Чтобы запустить отладку, выполните последовательность действий в разделе «Настройка JetBrains CLion» главы «Настройка среды разработки».

После выполнения этих действий появляется цель `Make Application`. Выберите в меню команду `Run ▶ Debug`, чтобы запустить процесс и приступить к отладке.

Также возможно подключить отладчик к работающему процессу CPython.

Подключение отладчика

Чтобы подключить отладчик CLion к работающему процессу CPython, выберите команду `Run ▶ Attach to Process`.

На экране появляется список выполняемых процессов. Найдите нужный процесс Python и выберите команду `Attach`. Начинается сеанс отладки.

ВАЖНО

Если у вас установлен плагин Python, то процесс Python будет отображаться сверху. Не выбирайте его!

Здесь используется отладчик Python, а не отладчик C:



Вместо этого прокрутите дальше вниз до списка Native и найдите нужный процесс Python.

Создание точек останова

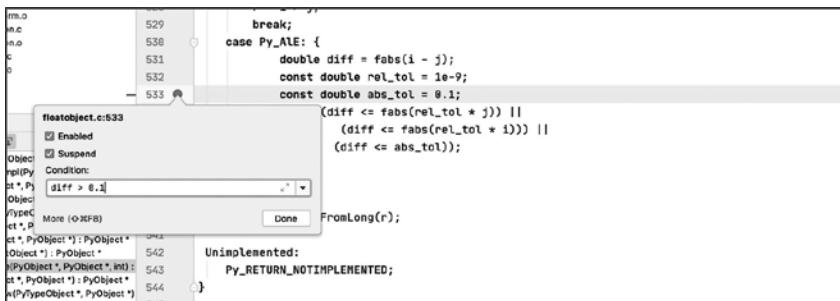
Чтобы поставить точку останова, перейдите к нужному файлу и строке, а затем щелкните по полю между номером строки и кодом. Появится красный кружок, показывающий, что точка поставлена:

```

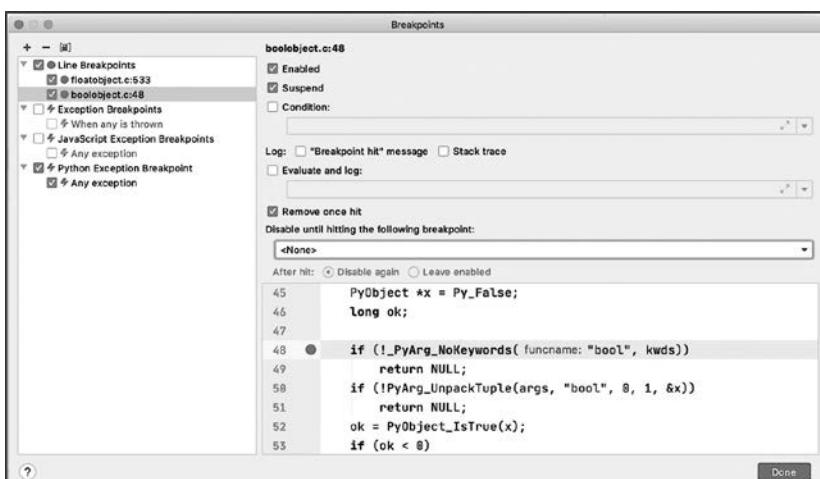
529         break;
530     case Py_EQ:
531         double diff = fabs(i - j);
532         const double rel_tol = 1e-9;
533         const double abs_tol = 0.1;
534         r = (((diff <= fabs(rel_tol * j)) ||
535               (diff <= fabs(rel_tol * i))) ||
536               (diff <= abs_tol));
537     }
538     break;
539 }
540 return PyBool_FromLong(r);
541
542 Unimplemented:
543     Py_RETURN_NOTIMPLEMENTED;
544 }
545

```

Щелкните правой кнопкой мыши по точке останова, чтобы определить условие:



Чтобы просмотреть все текущие точки останова и выполнить с ними нужные операции, выберите в меню команду Run ▶ View Breakpoints:

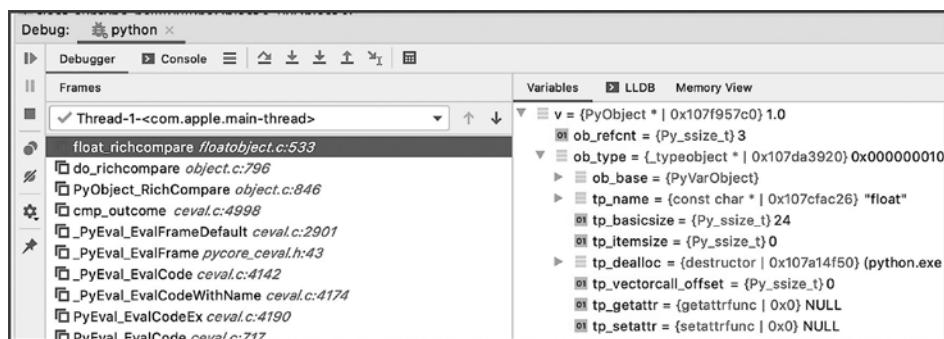


Точки останова можно включать и отключать: например, при достижении другой точки останова.

Работа с точками останова

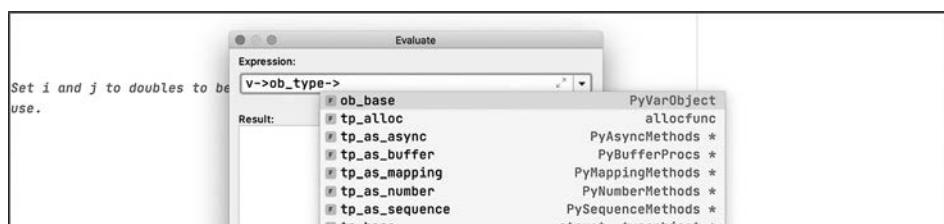
При достижении точки останова CLion открывает панель **Debug**. На ней отображается стек вызовов с информацией о том, где сработала точка. Вы можете переключаться между кадрами в стеке вызовов.

Далее в стеке идут локальные переменные. Вы можете раскрывать свойства указателей и структур типов; на панели выводятся значения простых типов:

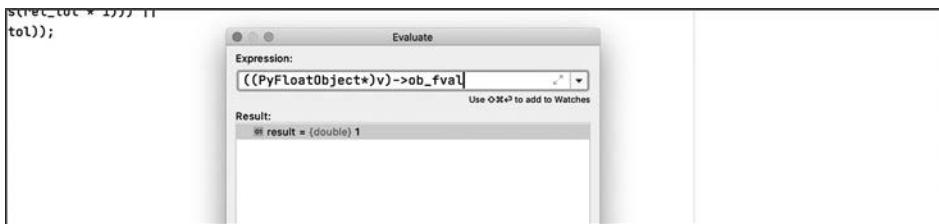


В точке останова можно вычислять выражения, чтобы получить больше информации о локальных переменных. Окно **Evaluate** открывается командой **Run** ▶ **Debugging Actions** ▶ **Evaluate Expression** или кнопкой в окне **Debug**.

Когда вы начинаете вводить выражение в окне **Evaluate**, механизм автозаполнения CLion предлагает возможные варианты свойств и типов:



Также поддерживается приведение типов выражений. Например, можно преобразовать `PyObject*` в любой существующий тип, скажем в `PyFloatObject*`:



ВЫВОДЫ

В этой главе вы узнали, как подготовить отладчик к использованию во всех основных операционных системах. Хотя исходная настройка занимает некоторое время, оно не будет потрачено напрасно. Возможность установки точек останова, анализа переменных и памяти наделит вас суперсилой, которая поможет в работе с расширением CPython, оптимизацией существующих частей кодовой базы или нахождением особенно коварных ошибок.

Бенчмаркинг, профилирование и трассировка

При внесении изменений в СPython необходимо следить за тем, чтобы эти изменения не имели отрицательного воздействия на производительность. Возможно, вам даже захочется изменить СPython так, чтобы производительность улучшилась.

В этой главе рассматриваются различные средства профилирования:

1. Использование модуля `timeit` для многократной проверки простых выражений Python и вычисления медианной скорости выполнения.
2. Выполнение `pyperformance` (набора тестов Python) для сравнения разных версий Python.
3. Анализ времени выполнения кадров с использованием `cProfile`.
4. Профилирование выполнения СPython при помощи датчиков.

Выбор решения зависит от типа задачи:

- **Бенчмарк** выдает среднее или медианное время выполнения фиксированного фрагмента кода, чтобы вы могли сравнить разные варианты.
- **Профайлер** строит граф вызовов с указанием времени выполнения, чтобы вы понимали, какая функция работает медленнее всего.

Профайлеры доступны на уровне С и на уровне Python. Если вы профилируете функцию, модуль или скрипт, написанные на Python, необходимо использовать профайлер Python. Если же профилируется модуль расширения

или модификация кода C в CPython, вам понадобится профайлер C или комбинация профайлеров C и Python.

Краткий перечень некоторых доступных инструментов:

ИНСТРУМЕНТ	КАТЕГОРИЯ	УРОВЕНЬ	ПОДДЕРЖКА ОС
timeit	Бенчмарк	Python	Все
pyperformance	Бенчмарк	Python	Все
cProfile	Профилирование	Python	Все
DTrace	Трассировка/профилирование	C	Linux, macOS

ВАЖНО

Прежде чем выполнять какие-либо тесты, лучше закрыть все приложения на компьютере, чтобы процессор был полностью выделен для бенчмарка.

ИСПОЛЬЗОВАНИЕ TIMEIT ДЛЯ МИКРОБЕНЧМАРКА

Набор тестов производительности Python тщательно проверяет общую оперативность среды выполнения CPython, делая множество итераций. Если вы хотите провести быстрое и простое сравнение конкретного фрагмента, используйте модуль `timeit`.

Чтобы применить `timeit` к короткому скрипту, запустите скомпилированную версию CPython с модулем `-m timeit` и заключите данный скрипт в кавычки:

```
$ ./python -m timeit -c "x=1; x+=1; x**x"
1000000 loops, best of 5: 258 nsec per loop
```

Чтобы уменьшить количество циклов, используйте флаг `-n`:

```
$ ./python -m timeit -n 1000 "x=1; x+=1; x**x"
1000 loops, best of 5: 227 nsec per loop
```

Пример использования timeit

В этой книге мы внесли изменения в тип `float`, добавив поддержку оператора «почти равно».

Попробуйте выполнить этот тест, чтобы оценить производительность сравнения двух чисел с плавающей точкой:

```
$ ./python -m timeit -n 1000 "x=1.0001; y=1.0000; x~=y"  
1000 loops, best of 5: 177 nsec per loop
```

Реализация этого сравнения содержится в функции `float_richcompare()`, внутри `Objects ▶ floatobject.c`:

Objects ▶ floatobject.c, строка 358

```
static PyObject*  
float_richcompare(PyObject *v, PyObject *w, int op)  
{  
    ...  
    case Py_EQ:  
        double diff = fabs(i - j);  
        double rel_tol = 1e-9;  
        double abs_tol = 0.1;  
        r = (((diff <= fabs(rel_tol * j)) ||  
              (diff <= fabs(rel_tol * i))) ||  
              (diff <= abs_tol));  
    }  
    break;  
}
```

Обратите внимание: значения `rel_tol` и `abs_tol` являются константами, но не имеют соответствующей пометки. Приведите их к следующему виду:

```
const double rel_tol = 1e-9;  
const double abs_tol = 0.1;
```

Теперь снова скомпилируйте CPython и заново выполните тест:

```
$ ./python -m timeit -n 1000 "x=1.0001; y=1.0000; x~=y"  
1000 loops, best of 5: 172 nsec per loop
```

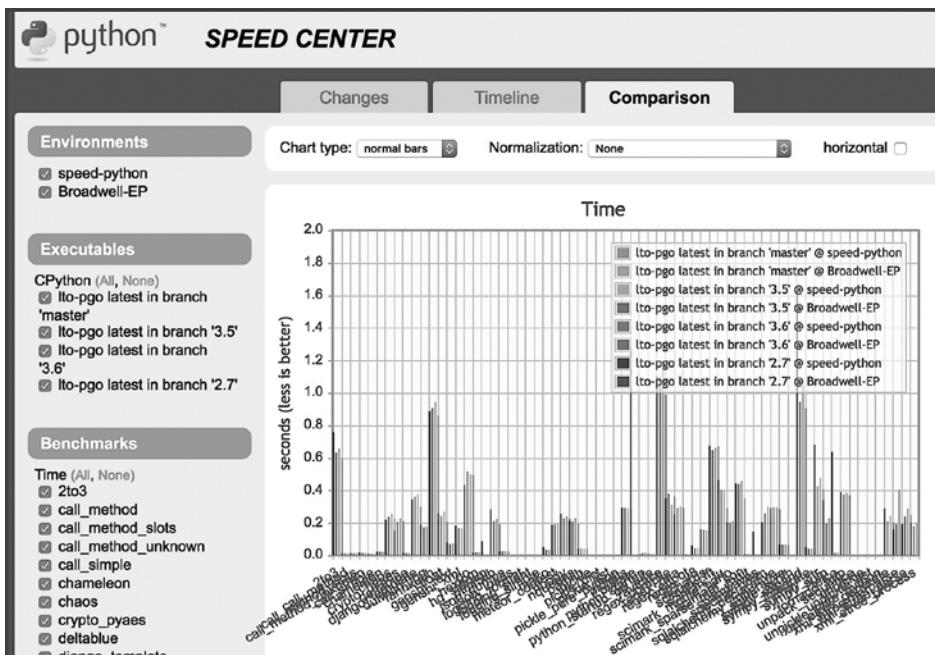
Наблюдается небольшое (от 1 до 5 процентов) улучшение производительности. Поэкспериментируйте с разными реализациями сравнения и посмотрите, удастся ли вам добиться еще большего прироста.

ИСПОЛЬЗОВАНИЕ НАБОРА ТЕСТОВ ПРОИЗВОДИТЕЛЬНОСТИ PYTHON

Набор тестов Python (Python Benchmark Suite) поможет оценить общую производительность Python. В него включены приложения Python, разработанные для тестирования нескольких аспектов среды выполнения Python при нагрузке.

Тесты производительности из набора написаны на чистом Python, что позволяет использовать их для тестирования разных сред выполнения, например PyPy и Jython. Кроме того, они совместимы с Python от 2.7 до последней версии.

Любые коммиты в главной ветви github.com/python/cpython тестируются с использованием инструментов бенчмарка, а результаты отправляются в Python Speed Center¹:



¹ <https://speed.python.org>.

В Speed Center можно параллельно сравнить различные коммиты и ветви. В бенчмарках применяется как профильная оптимизация, так и обычные сборки с фиксированной конфигурацией оборудования для обеспечения стабильности сравнений.

Набор тестов производительности можно установить из PyPI с использованием среды выполнения Python (не той, которую вы тестируете) в виртуальной среде:

```
(venv) $ pip install pyperformance
```

Затем необходимо создать файл конфигурации и выходной каталог для тестового профиля. Рекомендуется создать этот каталог за пределами вашей рабочей директории Git. Это также позволит вам извлечь из репозитория несколько разных версий.

Добавьте в файл конфигурации (например, `~/benchmarks/benchmark.cfg`) следующие строки:

cpython-book-samples ▶ 62 ▶ benchmark.cfg

```
[config]
# Путь к выходным файлам json
json_dir = ~/benchmarks/json

# Если True, скомпилировать CPython в отладочном режиме (без LTO и PGO),
# запустить тесты с --debug-single-sample и заблокировать отправку
#
# Используется для быстрого тестирования конфигурации
debug = False

[scm]
# Каталог с исходным кодом CPython (репозиторий Git)
repo_dir = ~/cpython

# Обновить репозиторий Git (git fetch)?
update = False

# Имя удаленного репозитория Git, используемое для создания
# ревизии ветви Git
git_remote = remotes/origin

[compile]
# Создать файлы в bench_dir:
bench_dir = ~/benchmarks/tmp

# Применять оптимизацию на стадии компоновки кода (LTO)?
lto = True
```

```
# Применять профильную оптимизацию (PGO)?
pgo = True

# Разделенный пробелами список пакетных библиотек
pkg_only =

# Установить Python? Если False, то Python запускается из каталога сборки
install = True

[run_benchmark]
# Выполнить "sudo python3 -m pyperf system tune" перед запуском тестов?
system_tune = True

# Параметр --benchmarks для 'pyperformance run'
benchmarks =

# Параметр --affinity для 'ruperf system tune' и 'pyperformance run'
affinity =

# Отправить сгенерированный файл JSON?
upload = False

# Конфигурация для отправки результатов на сайт Codespeed
[upload]
url =
environment =
executable =
project =

[compile_all]
# Список ветвей CPython в Git
branches = default 3.6 3.5 2.7

# Список ревизий для compile_all
[compile_all_revisions]
# Список 'sha1=' (ветвь по умолчанию: 'master') или 'sha1=branch',
# используемый командой "pyperformance compile_all"
```

Проведение бенчмарка

Когда файл конфигурации настроен, можно выполнить тесты:

```
$ pyperformance compile -U ~/benchmarks/benchmark.cfg HEAD
```

Команда компилирует CPython в заданном каталоге `repo_dir` и формирует выходной файл в формате JSON, помещая данные бенчмарка в каталог, заданный в файле конфигурации.

Сравнение результатов

В набор тестов производительности не входят инструменты для построения графов, поэтому если вы хотите сравнить результаты в формате JSON, в пределах виртуальной среды можно использовать приведенный ниже скрипт.

Сначала установите зависимости:

```
$ pip install seaborn pandas pyperformance
```

Затем создайте скрипт profile.py:

cpython-book-samples ▶ 62 ▶ profile.py

```
import argparse
from pathlib import Path
from perf._bench import BenchmarkSuite

import seaborn as sns
import pandas as pd

sns.set(style="whitegrid")

parser = argparse.ArgumentParser()
parser.add_argument("files", metavar="N", type=str, nargs="+",
                    help="files to compare")
args = parser.parse_args()

benchmark_names = []
records = []
first = True
for f in args.files:
    benchmark_suite = BenchmarkSuite.load(f)
    if first:
        # Инициализировать ключи словаря именами бенчмарков
        benchmark_names = benchmark_suite.get_benchmark_names()
        first = False
    bench_name = Path(benchmark_suite.filename).name
    for name in benchmark_names:
        try:
            benchmark = benchmark_suite.get_benchmark(name)
            if benchmark is not None:
                records.append({
                    "test": name,
                    "runtime": bench_name.replace(".json", ""),
                    "stdev": benchmark.stdev(),
                    "mean": benchmark.mean(),
                    "median": benchmark.median()
                })
        except KeyError:
```

```
# Дополнительный бенчмарк! Игнорировать
pass

df = pd.DataFrame(records)

for test in benchmark_names:
    g = sns.factorplot(
        x="runtime",
        y="mean",
        data=df[df["test"] == test],
        palette="YlGnBu_d",
        size=12,
        aspect=1,
        kind="bar")
    g.despine(left=True)
    g.savefig("png/{}-result.png".format(test))
```

Чтобы построить график, запустите этот скрипт из интерпретатора со сгенерированными JSON-файлами:

```
$ python profile.py ~/benchmarks/json/HEAD.json ...
```

Данная команда построит серию графов в подкаталоге `png/` для каждого выполняемого бенчмарка.

ПРОФИЛИРОВАНИЕ КОДА PYTHON С ИСПОЛЬЗОВАНИЕМ CPROFILE

В стандартную библиотеку включены два профайлеры для Python-кода:

1. `profile`: профайлер, написанный на чистом Python.
2. `cProfile`: более быстрый профайлер на С.

В большинстве случаев лучше использовать модуль `cProfile`.

`cProfile` подходит для анализа запущенных приложений и сбора детализированных профилей для выполняемых кадров. Отчет `cProfile` можно вывести в командной строке или сохранить его в файле `.pstat` для анализа во внешней программе.

В главе «Параллелизм и конкурентность» вы написали сканер портов на Python. Попробуйте профилировать это приложение в `cProfile`.

Чтобы запустить модуль `cProfile`, выполните команду `python` в командной строке с аргументом `-m cProfile`. Второй аргумент — выполняемый скрипт:

```
$ python -m cProfile portscanner_threads.py
Port 80 is open
Completed scan in 19.8901150226593 seconds
    6833 function calls (6787 primitive calls) in 19.971 seconds

    Ordered by: standard name

      ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
        2      0.000     0.000     0.000     0.000  ...


```

Выводимая таблица состоит из следующих столбцов.

СТОЛБЕЦ	НАЗНАЧЕНИЕ
ncalls	Количество вызовов
tottime	Общее время, проведенное в функции (за вычетом подфункций)
percall	Результат деления tottime на ncalls
cumtime	Общее время, проведенное в функции (включая подфункции)
percall	Результат деления cumtime на количество примитивных вызовов
filename:lineno(function)	Данные каждой функции

Также можно добавить аргумент `-s` и имя столбца, чтобы отсортировать вывод:

```
$ python -m cProfile -s tottime portscanner_threads.py
```

Эта команда сортирует вывод по общему времени, проведенному в каждой функции.

Экспортирование профилей

Вы можете снова запустить модуль cProfile с аргументом `-o`, чтобы указать путь к выходному файлу:

```
$ python -m cProfile -o out.pstat portscanner_threads.py
```

Команда создает файл `out.pstat`, который можно загрузить и проанализировать при помощи класса `Stats`¹ или внешней программы.

¹ <https://docs.python.org/3.9/library/profile.html#the-stats-class>.

Наглядное представление данных в SnakeViz

SnakeViz — бесплатный пакет Python для наглядного представления профильных данных в веб-браузерах.

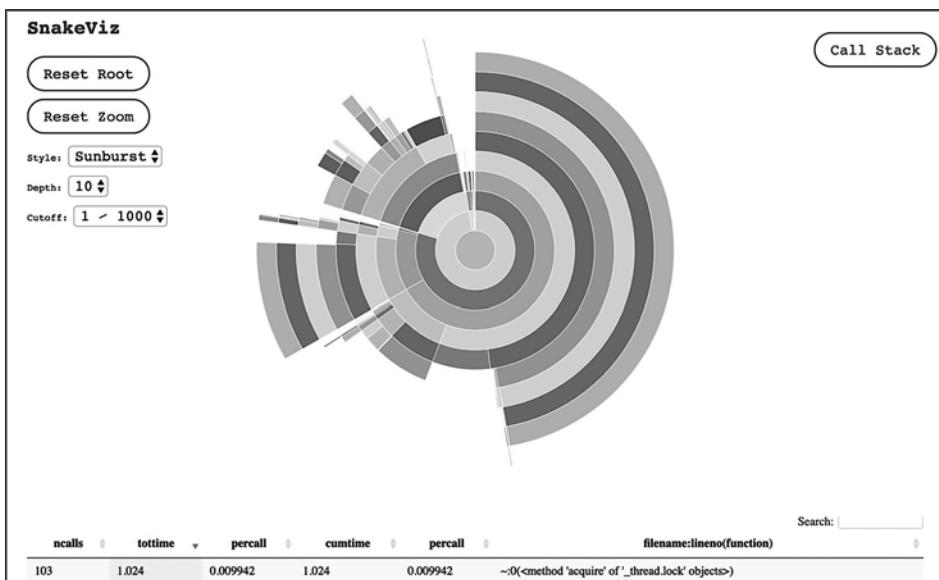
Для установки SnakeViz используйте pip:

```
$ python -m pip install snakeviz
```

Затем выполните `snakeviz` в командной строке с указанием пути к созданному файлу со статистикой:

```
$ python -m snakeviz out.pstat
```

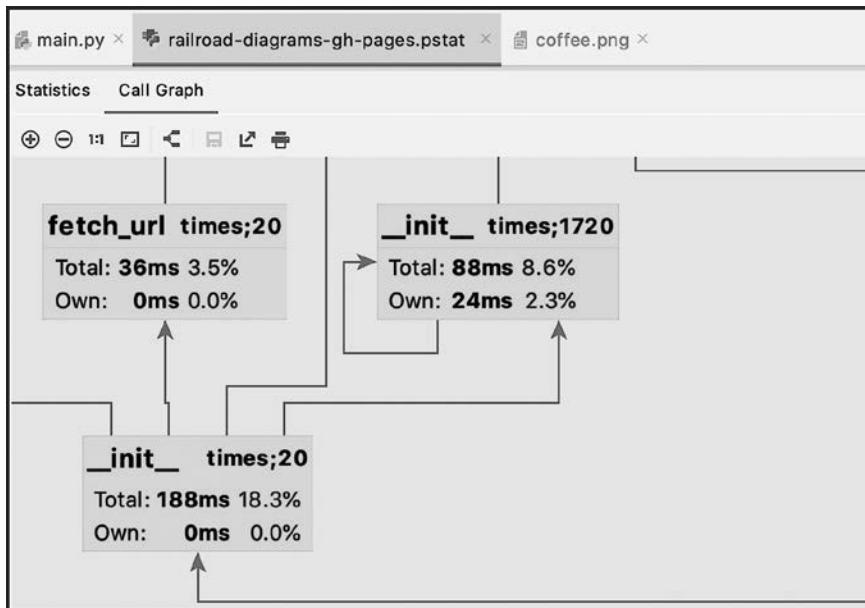
Команда открывает браузер, в котором можно исследовать и анализировать данные:



Визуализация данных в PyCharm

В PyCharm есть встроенный инструмент для запуска `cProfile` и наглядного представления результатов. Для этого необходимо настроить цель Python.

Чтобы запустить профайлер, выделите цель запуска и в меню выберите команду Run ▶ Profile (target). Команда выполнит цель запуска с `cProfile` и откроет окно визуализации с табличными данными и графиком вызовов:



ПРОФИЛИРОВАНИЕ КОДА С В DTRACE

Исходный код CPython содержит несколько маркеров для фреймворка трассировки DTrace. Dtrace выполняет скомпилированный двоичный файл С/C++, перехватывая и обрабатывая события в нем при помощи **датчиков** (probes).

Чтобы DTrace выдавал осмысленные данные, в скомпилированное приложение должны быть добавлены специальные маркеры. Они представляют события, возникающие во время выполнения. К маркерам можно присоединить произвольные данные для упрощения трассировки.

Например, функция вычисления кадра в Python ▶ `ceval.c` включает вызов `dtrace_function_entry()`:

```
if (PyDTrace_FUNCTION_ENTRY_ENABLED())
    dtrace_function_entry(f);
```

Маркер с именем `function_entry` в DTrace будет срабатывать при каждом входе в функцию.

В CPython существуют встроенные маркеры для:

- выполнения строк;
- входа в функцию и возвращения результата (выполнения кадра);
- запуска и завершения сборки мусора;
- запуска и завершения импортирования модулей;
- события аудита с хуками от `sys.audit()`.

Каждый из этих маркеров получает аргументы с дополнительной информацией. Например, в аргументах маркера `function_entry` передаются:

- Имя файла.
- Имя функции.
- Номер строки.

Статические аргументы маркеров определяются в официальной документации¹.

DTrace может выполнить файл скрипта, написанный на языке D, для запуска специального кода при срабатывании датчиков. Также можно фильтровать датчики по значениям их атрибутов.

Исходные файлы

Ниже перечислены исходные файлы, относящиеся к DTrace.

ФАЙЛ	НАЗНАЧЕНИЕ
Include ▶ pydtrace.h	Определение API для маркеров DTrace
Include ▶ pydtrace.d	Метаданные для провайдера Python, используемого DTrace
Include ▶ pydtrace_probes.h	Автоматически сгенерированные заголовки для обработки датчиков

¹ <https://realpython.com/cpython-static-markers/>.

Установка DTrace

DTrace предустановлен в macOS, а в Linux его можно установить пакетным менеджером.

Команда для систем на базе YUM:

```
$ yum install systemtap-sdt-devel
```

Команда для систем на базе APT:

```
$ apt-get install systemtap-sdt-dev
```

Компиляция поддержки DTrace

Поддержка DTrace должна быть скомпилирована в CPython. Это можно сделать при помощи скрипта `./configuration`.

Запустите `./configure` с теми же аргументами, которые использовались в главе «Компиляция CPython», и добавьте флаг `--with-dtrace`. Затем выполните команду `make clean && make`, чтобы заново построить двоичный файл.

Убедитесь в том, что программа конфигурации создала заголовок датчиков:

```
$ ls Include/pydtrace_probes.h  
Include/pydtrace_probes.h
```

ВАЖНО

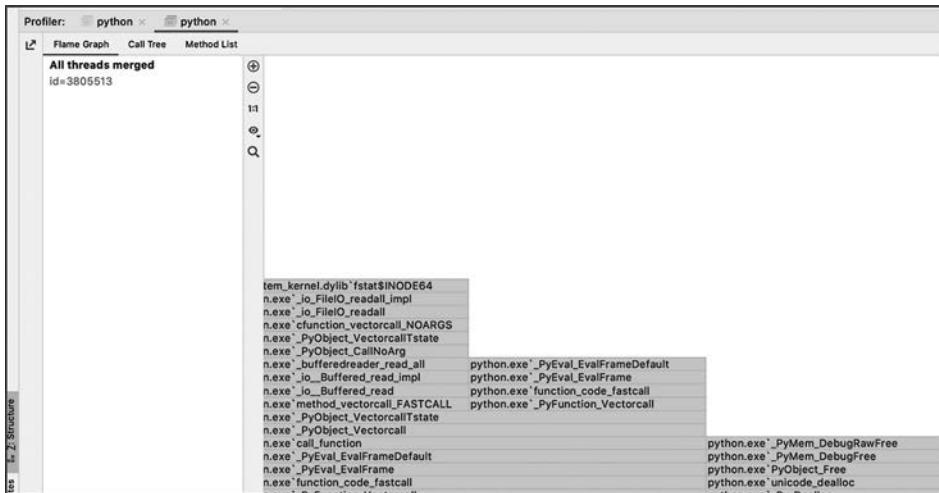
В новых версиях macOS установлена защита уровня ядра SIP (System Integrity Protection), мешающая работе DTrace. В примерах этой главы используются датчики CPython. Если вы захотите включить датчики `libc` или `syscall` для получения дополнительной информации, SIP необходимо отключить.

DTrace из CLion

IDE CLion поддерживает DTrace. Чтобы запустить трассировку, выполните команду `Run ▶ Attach Profiler to Process` и выберите запущенный процесс Python.

В окне профайлеров вам будет предложено запустить, а затем остановить сеанс трассировки. Когда трассировка будет завершена, вы получите

flame-график со стеками выполнения и временем вызова, деревом вызовов и списком методов:



Пример использования DTrace

В этой главе мы протестируем многопоточный сканер портов, созданный в главе «Параллелизм и конкурентность».

Создайте скрипт профилирования на D с именем `profile_compare.d`. Чтобы не выводить лишние данные при запуске интерпретатора, профайлер будет запускаться при входе в `portscanner_threads.py:main()`:

`cpython-book-samples > 62 > profile_compare.d`

```
#pragma D option quiet
self int indent;
python$target:::function-entry
/basename(copyinstr(arg0)) == "portscanner_threads.py"
  && copyinstr(arg1) == "main"/
{
    self->trace = 1;
    self->last = timestamp;
}

python$target:::function-entry
/self->trace/
{
    this->delta = (timestamp - self->last) / 1000;
    printf("%d\t%*s:", this->delta, 15, probename);
```

```

        printf("%*s", self->indent, "");
        printf("%s:%s:%d\n", basename(copyinstr(arg0)), copyinstr(arg1), arg2);
        self->indent++;
        self->last = timestamp;
    }

python$target:::function-return
/self->trace/
{
    this->delta = (timestamp - self->last) / 1000;
    self->indent--;
    printf("%d\t%*s:", this->delta, 15, probename);
    printf("%*s", self->indent, "");
    printf("%s:%s:%d\n", basename(copyinstr(arg0)), copyinstr(arg1), arg2);
    self->last = timestamp;
}
python$target:::function-return
/basename(copyinstr(arg0)) == "portscanner_threads.py"
&& copyinstr(arg1) == "main"/
{
    self->trace = 0;
}

```

При каждом выполнении функции скрипт выводит строку и измеряет промежуток времени между запуском и завершением функции.

При выполнении необходимо использовать аргумент скрипта `-s profile_compare` и аргумент команды `-c './python portscanner_threads.py'`:

```
$ sudo dtrace -s profile_compare.d -c './python portscanner_threads.py'
0 function-entry:portscanner_threads.py:main:16
28 function-entry: queue.py:__init__:33
18 function-entry: queue.py:__init__:205
29 function-return: queue.py:__init__:206
46 function-entry: threading.py:__init__:223
33 function-return: threading.py:__init__:245
27 function-entry: threading.py:__init__:223
26 function-return: threading.py:__init__:245
26 function-entry: threading.py:__init__:223
25 function-return: threading.py:__init__:245
```

ВАЖНО

Старые версии DTrace могут не поддерживать параметр `-c`. В таком случае вам придется выполнять DTrace и команду Python в разных оболочках.

В первом столбце будет выводиться время, прошедшее с последнего события в микросекундах, в следующих столбцах — имя события, имя файла и номер строки. При вложенных вызовах функций имя файла последовательно смещается вправо.

Выводы

В этой главе рассматривались способы бенчмаркинга, профилирования и трассировки с помощью разнообразных инструментов, разработанных для CPython. Правильно подобранный инструментарий поможет найти узкие места, сравнить производительность разных сборок и выявить возможности для оптимизации.

Что дальше?

В этой главе рассматриваются три возможных варианта применения информации, изложенной в книге:

1. Написание модулей расширения на С или С++.
2. Улучшение ваших приложений Python.
3. Участие в проекте CPython.

Начнем с написания модулей расширения на С или С++.

СОЗДАНИЕ РАСШИРЕНИЙ С ДЛЯ СРУТНОН

Функциональность Python можно расширять несколькими способами. Один из них — написание модулей Python на С или С++. Этот процесс может привести к улучшению производительности и упрощению доступа к библиотечным функциям С и системным функциям.

Если вы хотите написать модуль расширения С, вот вещи, к которым вам стоит вернуться:

- **Настройка компилятора С и компиляция модулей С** в главе «Компиляция CPython».
- **Настройка среды разработки для С** в главе «Настройка среды разработки».
- **Увеличение и уменьшение счетчика ссылок сгенерированных объектов** в разделе «Подсчет ссылок» главы «Управление памятью».
- **Описание PyObject* и его интерфейсов** в разделе «Типы объектов» главы «Объекты и типы».

- **Описание слотов типов и обращения к API типов Python из C** в разделе «Слоты типов» главы «Объекты и типы».
- **Добавление точек останова в исходные файлы C для модулей расширения и их отладка** в главе «Отладка».

СМ. ТАКЖЕ

Если у вас еще нет опыта написания модулей расширения C, обратитесь к статье «Building a C Extension Module¹» на сайте Real Python. В ней дается конкретный пример построения, компиляции и тестирования модуля расширения.

УЛУЧШЕНИЕ ПРИЛОЖЕНИЙ PYTHON

В книге мы рассмотрели несколько важных тем, которые помогут в улучшении ваших приложений:

- **Использование механизмов параллелизма и конкурентности для сокращения времени выполнения приложений** в главе «Параллелизм и конкурентность».
- **Настройка алгоритма сборки мусора для повышения эффективности работы с памятью в приложениях** в разделе «Сборка мусора» главы «Управление памятью».
- **Использование отладчиков для расширений C и обработки ошибок** в главе «Отладка».
- **Применение профайлеров для исследования времени выполнения вашего кода** в разделе «Профилирование кода Python с использованием cProfile» главы «Бенчмарк, профилирование и трассировка».
- **Анализ выполнения кадров для изучения и отладки сложных проблем** в разделе «Трассировка выполнения кадров» главы «Цикл вычисления».

¹ <https://realpython.com/build-python-c-extension-module>.

УЧАСТИЕ В ПРОЕКТЕ CPYTHON

За двенадцать месяцев вышло 12 дополнительных версий CPython, были сделаны сотни преобразований и отчетов об ошибках, а также тысячи коммитов в исходном коде.

CPython — один из самых крупных, динамичных и открытых программных проектов в мире. Знания, полученные в этой книге, помогут вам ориентироваться в нем и принимать участие в его совершенствовании.

Сообщество CPython охотно принимает новых участников. Но прежде чем предлагать какие-то улучшения или исправления, необходимо знать, с чего начать. Например:

1. УстраниТЬ ошибки, о которых сообщают разработчики на сайте bugs.python.org.
2. Исправить небольшие, хорошо описанные проблемы.

Рассмотрим каждый из этих вариантов более подробно.

Устранение ошибок

Все отчеты об ошибках и запросы на внесение изменений сначала отправляются на сайт bugs.python.org, также известный как ВРО. Этот веб-сайт является баг-трекером проекта CPython. Если вы захотите отправить pull request на GitHub, то вам сначала понадобится **номер ВРО** — код проблемы, созданный ВРО.

Для начала зарегистрируйтесь в качестве пользователя; для этого выберите команду **User ▶ Register** в меню слева.

Вид по умолчанию не особенно удобен. В нем отображаются как ошибки, выявленные пользователями, так и ошибки, обнаруженные ключевыми разработчиками, у которых обычно уже имеется решение.

Чтобы избежать этого, после входа в систему выберите команду **Your Queries ▶ Edit** в меню слева. На странице выведется список запросов индексов ошибок, которые вы можете добавить в закладки:

Query	
Patches	leave out
pending issues	leave out
serverhorror's Reports	leave out
Easy Tasks	leave out
Showstoppers	leave out
Latest issues	leave out
Release Blockers	include
Critical	leave out
py3k-open	leave out
Needs review	leave out
Crashers	leave out
Open Doc Bugs 2.6	leave in
Opened patches	leave out
Open documentation issues	leave out

Замените значение на `leave in`, чтобы добавить эти запросы в меню Your Queries.

Вот несколько запросов, которые я считаю полезными:

- **Easy Documentation Issues:** улучшения документации, которые еще не сделаны.
- **Easy Tasks:** задачи, которые были определены как подходящие для начинающих.
- **Recently Created:** недавно созданные ошибки.
- **Reports Without Replies:** сообщения об ошибках, которые остались без ответа.
- **Unread:** непрочитанные сообщения об ошибках.
- **50 Latest Issues:** 50 проблем, которые были недавно обновлены.

Далее вы можете следовать руководству «*Triaging an Issue*»¹, в котором описана последняя версия процесса публикации комментариев об ошибках.

Отправка pull request с решением проблемы

Определившись с проблемой, можно переходить к созданию исправления и отправке его в проект CPython. При этом используется следующая схема:

1. Убедитесь в том, что у вас имеется номер ВРО.
2. Создайте ветвь в вашем форке CPython. За описанием процесса загрузки исходного кода обращайтесь к главе «Загрузка исходного кода CPython».

¹ <https://devguide.python.org/triaging/>.

3. Создайте тест для воспроизведения проблемы. Последовательность действий описана в разделе «Модули тестирования» главы «Набор тестов».
4. Внесите изменение, следуя принципам руководств по стилю PEP 7 и PEP 8.
5. Выполните набор регрессионных тестов и убедитесь в том, что все тесты проходят успешно. Регрессионные тесты будут автоматически выполняться на GitHub при отправке pull request, но лучше сначала проверить локально. За описанием процедуры обращайтесь к главе «Набор тестов».
6. Сохраните коммит и отправьте свои изменения на GitHub.
7. Перейдите по адресу github.com/python/cpython и создайте pull request для своей ветви.

Позже запрос будет проанализирован одной из команд отладчиков и закреплен за каким-то ключевым разработчиком (или командой) для рецензирования.

Как упоминалось ранее, проекту CPython требуется больше участников. Между отправкой изменения и его рецензированием может пройти час, неделя или несколько месяцев. Не огорчайтесь, если не получите немедленной реакции. Многие ключевые разработчики работают на добровольной основе, и часто рецензирование или объединение pull-запросов происходит партиями.

ВАЖНО

Важно, чтобы один запрос решал только одну проблему. Если в ходе написания исправления вы обнаружите еще одну проблему, отметьте ее и отправьте во втором запросе.

Чтобы интеграция ваших изменений происходила быстрее, желательно включить хорошее объяснение проблемы, сути и способа решения.

Другие варианты поддержки

Кроме исправления ошибок, существуют и другие виды улучшений, которые можно внести в проект CPython:

- Во многих функциях и модулях стандартной библиотеки отсутствуют модульные тесты. Напишите тесты и отправьте их в проект.
- Многие функции стандартной библиотеки не имеют актуальной документации. Обновите документацию и отправьте изменения.

ДАЛЬНЕЙШЕЕ ОБУЧЕНИЕ

Одна из самых сильных сторон Python – сообщество. Знаете людей, изучающих Python? Помогите им! Если вы хотите убедиться в том, что вы действительно усвоили какую-то концепцию, попробуйте объяснить ее другим.

Продолжайте свое путешествие в мир Python, посетив наш сайт realpython.com.

Еженедельные советы для Python-разработчиков

Хотите каждую неделю получать порцию рекомендаций, которые улучшают производительность и оптимизируют ваш рабочий процесс? Хорошие новости: специально для таких Python-разработчиков, как вы, мы ведем еженедельную бесплатную рассылку по электронной почте.

Рассылка не сводится к материалам типа «вот вам список популярных статей». Мы стараемся делиться как минимум одной оригинальной мыслью в неделю в формате очерка (краткого).

Если вас это заинтересует, перейдите по адресу realpython.com/newsletter и введите адрес электронной почты в форме регистрации. До скорой встречи!

Библиотека видеокурсов Real Python

Большая (и постоянно растущая) подборка учебников Python и учебных материалов поможет вам достичь уровня всесторонне развитого питониста. Новые материалы публикуются еженедельно, и вы всегда найдете что-то интересное для повышения вашей квалификации.

- **Осваивайте практически значимые навыки программирования на языке Python:** созданием, отбором и контролем наших учебных материалов занимается целое сообщество опытных разработчиков. На сайте Real Python вы найдете ресурсы, заслуживающие доверия, которые пригодятся вам при совершенствовании мастерства программирования на Python.

- **Знакомьтесь с другими питонистами:** присоединяйтесь к чату сообщества Real Python и еженедельным Q&A сессиям, чтобы познакомиться с командой Real Python и с другими неофитами. Получайте ответы на свои вопросы, относящиеся к Python, обсуждайте вопросы программирования и построения карьеры или просто проводите с нами время у этого виртуального кофейного автомата.
- **Пройдите интерактивные тесты и траектории обучения:** реальные задачи по кодированию, интерактивные тесты и программы обучения, ориентированные на получение необходимых навыков, позволят вам оценить ваш текущий уровень и потренироваться в применении новых знаний.
- **Отслеживайте ход обучения:** помечайте уроки как завершенные или «в процессе», занимайтесь в наиболее подходящем для вас темпе. Создавайте закладки на самых интересных уроках и просматривайте их позднее, чтобы запомнить надолго.
- **Получайте сертификаты об окончании курсов:** по окончании каждого курса вы получаете сертификат, который можно предъявить (в электронном или печатном виде). Включайте сертификаты в свой проектный портфель, резюме на LinkedIn или на других веб-сайтах, чтобы показать миру, что вы продвинутый питонист.
- **Идите в ногу со временем:** поддерживайте свои рабочие навыки на должном уровне и не отставайте от технологических новинок. Мы постоянно выпускаем новые учебные материалы только для зарегистрированных участников, а также регулярно обновляем контент.

За информацией о доступных курсах обращайтесь на realpython.com/courses.

Приложение. Введение в С для Python- программистов

Приложение предназначено для опытных Python-разработчиков, желающих освоить азы языка C и научиться пользоваться им, работая с исходным кодом CPython. Предполагается, что читатель понимает синтаксис Python хотя бы на среднем уровне.

Язык C не особенно сложен, и то, как он применяется в CPython, можно свести к небольшому набору синтаксических правил. Вы довольно быстро научитесь понимать код, однако умение писать эффективные программы на C приходит намного позднее. Данное руководство преследует лишь первую цель.

Начнем с первого отличия между Python и C, с которым вы сталкиваетесь практически немедленно, — препроцессора C.

ПРЕПРОЦЕССОР С

Препроцессор обрабатывает исходные файлы до того, как они будут обработаны компилятором. Он обладает весьма ограниченными возможностями, но эти возможности приносят огромную пользу при построении программ на C.

Препроцессор создает новый файл, который, собственно, и будет реально обрабатываться компилятором. Все команды препроцессора размещаются в начале строки, а первым непробельным символом в них является знак #.

Главная цель препроцессора — замена текста в исходном файле, но он также может выполнять простейшие условные конструкции при помощи `#if` и других операторов.

Начнем с самой распространенной директивы препроцессора: `#include`.

#include

`#include` подставляет содержимое одного файла в текущий исходный файл. В директиве `#include` нет ничего сложного: она читает файл из файловой системы, обрабатывает его препроцессором и помещает результаты в выходной файл. Это происходит рекурсивно для каждой директивы `#include`. Например, если заглянуть в файл `Modules/_multiprocessing/semaphore.c`, то в начале этого файла обнаруживается следующая строка:

```
#include "multiprocessing.h"
```

Она приказывает препроцессору взять все содержимое `multiprocessing.h` и подставить его в выходной файл в соответствующей позиции.

Команда `#include` существует в двух разных формах. В одной имя включаемого файла заключается в двойные кавычки (" "), а в другой — в угловые скобки (<>). Эти формы различаются тем, по какому пути выполняется поиск файла в файловой системе.

Если имя файла заключено в угловые скобки <>, то препроцессор ограничивается системными файлами. Если же название файла в кавычках, препроцессор сначала ищет файл в локальном каталоге и только потом переходит к системным каталогам.

#define

Директива `#define` позволяет выполнять простую замену текста, а также сочетается с директивами `#if`, о которых будет рассказано ниже.

На простейшем уровне `#define` определяет новое символическое имя, которое заменяется текстовой строкой в выводе препроцессора.

Так, в файле `semaphore.c` присутствует следующая строка:

```
#define SEM_FAILED NULL
```

Она приказывает препроцессору заменить каждый экземпляр `SEM_FAILED` ниже этой точки литеральной строкой `NULL`, прежде чем код будет передан компилятору.

Директивы `#define` также могут получать параметры, как в следующей версии `SEM_CREATE` для Windows:

```
#define SEM_CREATE(name, val, max) CreateSemaphore(NULL, val, max, NULL)
```

В данном случае препроцессор ожидает, что `SEM_CREATE()` будет выглядеть как вызов функции и получит три параметра. Обычно подстановки такого рода называются **макросами**. Текст трех параметров напрямую подставляется в выходной код.

Например, в строке 460 файла `semaphore.c` макрос `SEM_CREATE` используется следующим образом:

```
handle = SEM_CREATE(name, value, max);
```

При компиляции на Windows происходит расширение макроса, в результате чего строка будет выглядеть так:

```
handle = CreateSemaphore(NULL, value, max, NULL);
```

Как будет показано ниже, этот макрос по-разному определяется в разных операционных системах.

#undef

Директива стирает все предшествующие определения препроцессора из `#define`. Это позволяет ограничить действие `#define` частью файла.

#if

Препроцессор также поддерживает условные конструкции, которые позволяют включать или исключать фрагменты текста в зависимости от определенных условий. Условные команды закрываются директивой `#endif`. Для более точной настройки можно также использовать `#elif` и `#else`.

В исходном коде CPython встречаются три основные разновидности `#if`:

1. `#ifdef <макрос>` включает следующий блок текста, если указанный макрос определен. Также эта конструкция иногда записывается в виде `#if defined(<макрос>)`.
2. `#ifndef <макрос>` включает следующий блок текста, если указанный макрос **не** определен.
3. `#if <макрос>` включает следующий блок текста, если макрос определен **и** его вычисление дает результат `True`.

Обратите внимание: для описания того, что включается или исключается из файла, используется термин «текст», а не «код». Препроцессор ничего не знает о синтаксисе C, и его не интересует, что собой представляет заданный текст.

#pragma

Директивы `#pragma` содержат инструкции или рекомендации для компилятора. Как правило, при чтении кода на них можно не обращать внимания, так как обычно они влияют на компиляцию кода, а не на его выполнение.

#error

Наконец, директива `#error` выводит сообщение и заставляет препроцессор прервать обработку. При чтении исходного кода CPython эти директивы также можно смело игнорировать.

БАЗОВЫЙ СИНТАКСИС С

Этот раздел не охватывает все аспекты языка C, и я не ожидаю, что вы научитесь писать на нем код. Основное внимание здесь уделяется тем частям C, которые могут показаться незнакомыми или непонятными для Python-разработчиков.

Общие сведения

В отличие от Python, пробельные символы (whitespace) не важны для компилятора C. Компилятор не обратит внимания на то, что одна команда разбита

на несколько строк или же вся программа состоит из одной очень длинной строки. Это объясняется тем, что во всех командах и блоках используются разделители.

Конечно, существуют очень конкретные правила для парсера, но в общем случае для понимания исходного кода CPython достаточно знать, что каждая команда завершается точкой с запятой (;), а все блоки кода заключаются в фигурные скобки ({}).

У этого правила есть исключение: если блок состоит только из одной команды, то фигурные скобки можно опустить.

Все переменные в C должны быть объявлены, то есть в программе должна присутствовать одна команда, в которой указывается тип этой переменной. Обратите внимание: в отличие от Python, тип данных, хранящихся в одной переменной, меняться не может.

Рассмотрим несколько примеров:

```
/* Комментарии размещаются между символами "косая черта-звездочка" */
/* и "звездочка-косая черта" */
/* Такие комментарии могут занимать несколько строк -
   а значит, эта часть все еще является комментарием */

// Комментарии также могут следовать за двумя косыми чертами
// Такие комментарии могут идти только до конца строки, поэтому новые
// строки также должны начинаться с двойной косой черты (//)

int x = 0; // Объявляет x типа 'int' и инициализирует его со значением 0

if (x == 0) {
    // Это блок кода
    int y = 1; // Имя переменной у действительно только до закрывающей скобки }
    // Другие команды
    printf("x is %d y is %d\n", x, y);
}

// В односрочных блоках фигурные скобки не нужны
if (x == 13)
    printf("x is 13!\n");
printf("past the if block\n");
```

Как правило, код CPython очень четко отформатирован, а в пределах модуля обычно применяется единый стиль.

Операторы if

В С оператор `if` работает в целом так же, как в Python: если условие истинно, то выполняется следующий за ним блок. Синтаксис `else` и `elseif` должен быть знаком большинству Python-программистов. Следует помнить, что операторам `if` в C `endif` не нужен, потому что блоки ограничиваются фигурными скобками {}.

В С существует сокращенная запись для коротких команд `if ... else` — так называемый **тернарный оператор**:

```
условие ? true_результат : false_результат
```

Например, такой оператор встречается в файле `semaphore.c`, где он используется для определения макроса `SEM_CLOSE()` на Windows:

```
#define SEM_CLOSE(sem) (CloseHandle(sem) ? 0 : -1)
```

Возвращаемое значение этого макроса будет равно 0, если функция `CloseHandle()` возвращает `true`, или -1 в противном случае.

ПРИМЕЧАНИЕ

Переменные логического типа поддерживаются и используются в разных частях исходного кода CPython, но они не являются частью языка. С интерпретирует бинарные условия по простому правилу: 0 или NULL интерпретируется как `false`, а все остальное — как `true`.

Операторы switch

В отличие от Python, С также поддерживает команду `switch`. Этую команду можно рассматривать как сокращенную запись для расширенных цепочек `if ... elseif`. Следующий пример взят из файла `semaphore.c`:

```
switch (WaitForSingleObjectEx(handle, 0, FALSE)) {
    case WAIT_OBJECT_0:
        if (!ReleaseSemaphore(handle, 1, &previous))
            return MP_STANDARD_ERROR;
        *value = previous + 1;
```

```
    return 0;
case WAIT_TIMEOUT:
    *value = 0;
    return 0;
default:
    return MP_STANDARD_ERROR;
}
```

Эта команда осуществляет выбор в зависимости от возвращаемого значения `WaitForSingleObjectEx()`. Если значение равно `WAIT_OBJECT_0`, то выполняется первый блок. При значении `WAIT_TIMEOUT` выполняется второй блок, а все остальные значения перехватываются блоком `default`.

Обратите внимание: проверяемое значение (в данном случае значение, возвращаемое `WaitForSingleObjectEx()`) должно быть целочисленного или перечисляемого типа, а в каждой секции `case` должна быть указана константа.

Циклы

В C существует три разновидности циклических конструкций:

1. Циклы `for`.
2. Циклы `while`.
3. Циклы `do ... while`.

Синтаксис циклов `for` заметно отличается от Python:

```
for (<инициализация>; <условие>; <приращение>) {
    <многократно выполняемый код>
}
```

Кроме многократно выполняемого кода цикл `for` содержит три блока:

1. Блок `<инициализация>` выполняется ровно один раз при запуске цикла. Обычно он используется для присваивания начального значения счетчику цикла (и, возможно, объявления счетчика цикла).
2. Блок `<приращение>` выполняется после каждого прохождения основного блока цикла. Традиционно в нем увеличивается счетчик цикла.
3. Наконец, блок `<условие>`, который выполняется после блока `<приращение>`. Вычисляется возвращаемое значение, и цикл прерывается, если условие возвращает `false`.

Пример из файла Modules/sha512module.c:

```
for (i = 0; i < 8; ++i) {
    S[i] = sha_info->digest[i];
}
```

Цикл выполняется 8 раз, с увеличением *i* от 0 до 7, и завершается, когда при проверке условия окажется, что значение *i* достигло 8.

Циклы `while` практически идентичны своим аналогам в Python, а синтаксис `do ... while` несколько отличается. Условие цикла `do ... while` проверяется только после того, как цикл будет выполнен в первый раз.

Примеры использования циклов `for` и `while` в кодовой базе CPython встречаются достаточно часто, но цикл `do ... while` не используется.

Функции

По синтаксису функций язык C похож на Python, не считая того, что в C типы возвращаемого значения и параметров должны быть заданы явно.

Функция в C выглядит так:

```
<возвращаемый_тип> имя_функции(<параметры>) {
    <тело_функции>
}
```

Возвращаемым типом может быть любой действительный тип C, включая встроенные типы (такие, как `int` и `double`), а также специальные типы вроде `PyObject`, как в следующем примере из `semaphore.c`:

```
static PyObject *
semlock_release(SemLockObject *self, PyObject *args)
{
    <команды тела функции>
}
```

Здесь вы видите несколько особенностей, характерных для C. Прежде всего напомню, что пробельные символы игнорируются. В исходном коде CPython возвращаемый функцией тип очень часто указывается в строке над остальной частью объявления функции (фрагмент `PyObject *`). Смысл символа `*` будет описан позднее, а пока достаточно сказать, что к функциям и переменным могут применяться различные модификаторы. `static` — один из таких модификаторов.

Работой модификаторов управляют довольно сложные правила. Например, смысл модификатора `static` в данном случае полностью отличается от того, что он означал бы при размещении перед объявлением переменной.

К счастью, когда вы пытаетесь прочитать и понять исходный код CPython, модификаторы обычно можно игнорировать.

Список параметров для функций представляет собой переменные, разделенные запятыми, как и в Python. И снова C требует указания типов всех параметров, так что `SemLockObject *self` сообщает, что первый параметр содержит указатель на `SemLockObject` и называется `self`. Все параметры в C являются позиционными.

А теперь разберемся, что же означает «указатель» в этой команде.

Добавим немного контекста: параметры передаются функциям C по значению. Это означает, что функция работает с копией значения, а не с исходным значением в вызывающей функции. Чтобы обойти это ограничение, функциям часто передается адрес данных, чтобы эти данные могли изменяться функцией.

Эти адреса называются **указателями** и обладают типом. Так, `int *` — указатель на целое число, и этот тип отличается от `double *` — указателя на число с плавающей точкой с двойной точностью.

Указатели

Как упоминалось ранее, указатели представляют собой переменные, которые хранят адрес значения. Они часто используются в C, как показано в следующем примере:

```
static PyObject *
semlock_release(SemLockObject *self, PyObject *args)
{
    <команды тела функции>
}
```

Здесь параметр `self` содержит адрес значения `SemLockObject` (то есть указатель на него). Также обратите внимание на то, что функция возвращает указатель на значение `PyObject`.

В C существует специальное значение `NULL`; оно сообщает, что указатель ни на что не указывает. В исходном коде CPython присваивание `NULL` и проверка указателей на `NULL` встречаются достаточно часто. Такие проверки важны,

потому что ограничения на возможные значения указателей минимальны, а обращение к адресу памяти, не принадлежащему вашей программе, может привести к ее странному поведению.

С другой стороны, если вы попытаетесь обратиться к памяти по адресу NULL, программа немедленно завершится. Может показаться, что это слишком радикальное решение, но обычно проще искать ошибку памяти, проверяя указатель на NULL, чем проверяя, что некоторый адрес в памяти был изменен.

Строки

В языке С нет строкового типа. Существует условное соглашение, на базе которого написаны многие функции стандартной библиотеки, но реального типа не существует. Строки в С хранятся в виде массивов `char` (для ASCII) или `wchar` (для Юникода); каждый из этих типов содержит один символ. Строки помечаются **нуль-терминатором**, то есть значением 0, которое обычно обозначается в коде как `\0`.

Основные строковые операции, такие как `strlen()`, полагаются на расположение нуль-терминаатора, отмечающего конец строки.

Так как строки представляют собой массивы значений, их нельзя сравнивать или копировать напрямую. Для выполнения этих операций в стандартной библиотеке имеются функции `strcpy()` и `strcmp()` (и их аналоги для `wchar`).

Структуры

В последнем разделе нашего мини-обзора С рассмотрим возможность создания новых типов в С. Ключевое слово `struct` позволяет сгруппировать набор разных типов данных в новый нестандартный тип данных — структуру:

```
struct <имя_структурь> {
    <тип> <имя_поля>;
    <тип> <имя_поля>;
    ...
};
```

Следующий фрагмент из файла `Modules/arraymodule.c` демонстрирует объявление структуры:

```
struct arraydescr {
    char typecode;
```

```
    int itemsize;
    ...
};
```

В примере создается новый тип данных с именем `arraydescr`, который состоит из нескольких полей. Первые два поля — `char typecode` и `int itemsize`.

Структуры часто используются как часть `typedef`, предоставляющего удобный синоним для имени. В приведенном выше примере все переменные нового типа должны объявляться с полным именем `struct arraydescr x;`.

Часто встречается синтаксис следующего вида:

```
typedef struct {
    PyObject_HEAD
    SEM_HANDLE handle;
    unsigned long last_tid;
    int count;
    int maxvalue;
    int kind;
    char *name;
} SemLockObject;
```

Этот пример создает новый нестандартный тип структуры и присваивает ему имя `SemLockObject`. Для объявления переменной этого типа можно использовать синоним `SemLockObject x;`.

ВЫВОДЫ

На этом краткий обзор синтаксиса С завершается. Хотя такое описание нельзя назвать даже поверхностным, этого достаточно для чтения и понимания исходного кода CPython.

Благодарности

Спасибо моей жене Верити за поддержку и терпение. Без нее эта книга никогда бы не появилась на свет.

Спасибо всем, кто поддерживал меня в этом путешествии.

Энтони Шоу

Также хотим поблагодарить читателей ранних версий книги за полученную от них полезную информацию:

Юрген Гмач (Jürgen Gmach), ES Александр (Alexander), Пэттон Брэдфорд (Patton Bradford), Михал Портес (Michał Porteš), Сэм Робертс (Sam Roberts), Вишну Шрикумар (Vishnu Sreekumar), Матиас Хъяртстрём (Mathias Hjärtström), Сорен Вебер (Søren Weber), Арт (Art), Мэри Честер-Кэдуэлл (Mary Chester-Kadwell), Йонахан Рейхельт Гьертсен (Jonathan Reichelt Gjertsen), Андрей Ферриян (Andrey Ferriyan), Гийом (Guillaume), Майка Лайл (Micah Lyle), Роберт Уиллхофт (Robert Willhoft), Хуан Мануэль Гимено (Juan Manuel Gimeno), Блажей Михалик (Błażej Michalik), RWA, Дэйв (Dave), Лайонел (Lionel), Паси (Pasi), Тэд (Thad), Стив Хилл (Steve Hill), Маурисио (Mauricio), Р. Уэйн (R. Wayne), Карлос (Carlos), Мэри (Mary), Антон Зайнев (Anton Zayniev), aleks, Линдси Джон Арендс (Lindsay John Arendse), Винсент Пулайо (Vincent Poulailleau), Кристиан Хеттледж (Christian Hettlage), Фелипе «Биду» Родригес (Felipe «Bidu» Rodrigues), Франсуа (Francois), Юджин Латэм (Eugene Latham), Джордан Роуланд (Jordan Rowland), Дженн Д. (Jenn D.), Эйнджел (Angel), Мауро Фиакко (Mauro Fiacco), Роландас (Rolandas), Радек (Radek), Питер (Peter), milos, Ханс Дэвидссон (Hans Davidsson), Бернат Габор (Bernat Gabor), Флориан Далиц (Florian Dahlitz), Андерс Богнес (Anders Bogsnes), Шмуэл Каменски (Shmuel Kamensky), Мэтт Кларк (Matt Clarke), Джош Дейнер (Josh

Deiner), Орен Вольф (Oren Wolfe), Р. Уэйн Аренц (R. Wayne Arenz), emily spahn, Эрик Рейнджер (Eric Ranger), Дэйв Грюнвальд (Dave Grunwald), bob desinger, Роберт (Robert), Питер Макдональд (Peter McDonald), Аллен Хуан (Allen Huang), Се Ен Пак (Seyoung Park), Юджин (Eugene), Картик (Kartik), Вегард Стикбакке (Vegard Stikbakke), Мэтт Янг (Matt Young), Мартин Берг Питерсен (Martin Berg Petersen), Джек Камье (Jack Camier), Кейити Кобаяси (Keiichi Kobayashi), Джюлиус Шварц (Julius Schwartz), Люк (Luk), Кристиан (Christian), Аксель Вуатье (Axel Voitier), Александр (Aleksandr), Хавьер Новоя Катаньо (Javier Novoa Cataño), travis, Найям Съед (Najam Syed), Себастиан Нельс (Sebastian Nehls), И Вэй (Yi Wei), Бренден (Branden), paolo, Джим Вудворд (Jim Woodward), Хууб ван Тинен (Huub van Thienen), Эдуард Даурте (Edward Duarte), Рэй (Ray), Айвен (Ivan), Крис Герриш (Chris Gerrish), Спенсер (Spencer), Володимир (Volodymyr), Роб Пинкerton (Rob Pinkerton), Бен Кэмпбелл (Ben Campbell), Франциск (Francesc), Крис Смит (Chris Smith), Джон Видерхирн (John Wiederhирн), Джон Пек (Jon Peck), Бо Сенъяр (Beau Senyard), Реми МЕВЕР (Rémi MEVAERE), Карлос С. Анд (Carlos S Ande), Абхинав Упадхаяй (Abhinav Upadhyay), Чарльз Вегжин (Charles Wegrzyn), Ярослав Незвал (Yaroslav Nezval), Бен Хокли (Ben Hockley), Марин Мусо (Marin Muso), Джон Буссолетти (John Bussolletti), Джонатон (Jonathon), Керби Джеффрард (Kerby Geffrard), Эндрю Монталенти (Andrew Montalenti), Матеуш Ставярски (Mateusz Stawiarski), Эванс Сумаоро (Evance Soumaoro), Флетчер Грэм (Fletcher Graham), Андре Роберж (André Roberge), Дэниел Хао (Daniel Hao), Кимиа (Kimia). Если мы забыли упомянуть ваше имя, пожалуйста, знайте, что мы крайне признательны за вашу помощь. Спасибо всем!

Энтони Шоу

Внутри СРУТНОН: гид по интерпретатору Python

Перевел с английского Е. Матвеев

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Питиримов</i>
Ведущий редактор	<i>Е. Строганова</i>
Литературный редактор	<i>А. Алимова</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, М. Молчанова</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 01.2023. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 30.11.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 1000. Заказ 0000.

Дэн Бейдер

ЧИСТЫЙ PYTHON. ТОНКОСТИ ПРОГРАММИРОВАНИЯ ДЛЯ ПРОФИ



Изучение всех возможностей Python — сложная задача, а с этой книгой вы сможете сосредоточиться на практических навыках, которые действительно важны. Раскопайте «скрытое золото» в стандартной библиотеке Python и начните писать чистый код уже сегодня.

- Если у вас есть опыт работы со старыми версиями Python, вы сможете ускорить работу с современными шаблонами и функциями, представленными на Python 3;
- Если вы работали с другими языками программирования и хотите перейти на Python, то найдете практические советы, необходимые для того, чтобы стать эффективным питонщиком;
- Если вы хотите научиться писать чистый код, то найдете здесь самые интересные примеры и малоизвестные трюки.

КУПИТЬ