ChatGPT

Продвинутый Python 3.10+: углублённое руководство

1. Области видимости переменных в Python

Модель LEGB (Local, Enclosing, Global, Built-in). В Python область видимости определяет видимость имени (переменной) в разных частях программы. LEGB – это акроним, обозначающий порядок поиска имен: **Local** (локальная область внутри функции), **Enclosing** (охватывающая, то есть область внешних функций для вложенных функций), **Global** (глобальная область модуля) и **Built-in** (встроенная область имён интерпретатора). При обращении к имени Python сначала ищет его в локальной области текущей функции, затем в областях содержащих функций (если это вложенная функция), потом в глобальных переменных модуля и, наконец, среди встроенных имен интерпретатора [1] [2]. Например, если переменная не определена в текущей функции, но определена во внешней функции, будет использовано значение из внешней (охватывающей) области, а если не найдена и там – поиск перейдёт на уровень модуля, и затем к встроенным именам (таким как `len`, `range` и т.д., определённым в модуле `builtins`) [3] [4]. Таким образом, LEGB описывает и уровни областей видимости, и порядок поиска по ним.

Разница между Scope и Namespace (область видимости и пространством имён). *Пространство имён* – это контейнёр, сопоставляющий имена объектов с самими объектами (реализуется обычным словарём). Например, атрибут `__dict__` у модуля или объекта хранит его пространство имён (имена переменных и их значения) [5] [6]. *Область видимости* же – это часть программы (например, блок кода функции, модуль, класс), в пределах которой определённые имена доступны без квалификации. Python реализует области видимости через пространства имён: так, локальная область видимости функции – это просто её локальный словарь переменных, глобальная область – словарь переменных модуля, а встроенная – словарь модуля `builtins` [5] [7]. Отличие в том, что *область видимости* определяет где в коде можно использовать имя, а *пространство имён* – как хранятся сами имена и объекты. Иными словами, scope – это правило поиска имени, а namespace – это конкретное место хранения сопоставления имен с объектами. Когда Python ищет имя, он последовательно просматривает связанные пространства имён в соответствии с текущей областью видимости (LEGB) [7] [8].

Локальная, глобальная и нелокальная переменные. Переменные, созданные внутри функции, находятся в локальной области видимости (Local). Переменные, созданные в основном теле модуля (вне функций), – глобальные для этого модуля. По умолчанию присваивание имени внутри функции создает или изменяет локальную переменную. Если же нужно в функции присвоить значение переменной из глобальной области или из области внешней функции, используются ключевые слова `global` и `nonlocal`. Ключевое слово `global` сообщает, что имя принадлежит глобальному пространству модуля, и присваивания будут происходить в глобальной области (при этом переменная должна существовать глобально, иначе будет создана глобально). Ключевое слово `nonlocal` аналогично сообщает, что имя находится во внешней (охватывающей) функции. `nonlocal` ищет ближайшую внешнюю область видимости (не глобальную) и связывает переменную с ней. Важное отличие: `nonlocal` не позволит искать имя вплоть до глобального уровня – только в объемлющих функциях. Если имя не найдено в объемлющих функциях, использование `nonlocal` вызовет ошибку. При компиляции функции

Python определяет, к какой области относится имя: если в текущем блоке (функции) для имени отсутствует объявление `global` / `nonlocal`, то присваивание создаёт новую локальную переменную [9]. Если объявлено `global`, то переменная считается глобальной и будет браться из модуля (или создана в модуле) [10]. Если объявлено `nonlocal`, переменная привязывается к существующему имени во внешней функции. Пример:

```
x = 5       # глобальная переменная
def outer():
    x = 1   # переменная в области outer (скрывает глобальную x)
    def inner():
        nonlocal x   # ссылается на x из outer, а не создаёт новую
        x += 1       # изменяем переменную outer.x
    inner()
    print(x)
outer()      # выведет 2, т.к. outer.x был увеличен во внутренней функции
print(x)     # выведет 5, глобальная x осталась неизменной
```

Без `nonlocal` присваивание `x += 1` внутри `inner()` трактовалось бы как создание локальной `x` в `inner`, не затрагивая `x` из `outer`, и попытка прочитать ее перед созданием привела бы к `UnboundLocalError`. Таким образом `global` и `nonlocal` позволяют явно указать, что переменная находится не в локальном scope, а вне его.

Функции `globals()`, `locals()` **и** `dir()`**.** Эти встроенные функции позволяют получить доступ к пространствам имён текущего выполнения. `globals()` возвращает словарь глобальных имен текущего модуля [11]. `locals()` возвращает словарь локальных переменных текущей области (функции) [12]. Например, внутри функции `locals()` даст словарь ее локальных переменных, а если вызвать его на уровне модуля – вернёт то же, что `globals()`. Учтите, что изменения, внесённые в словарь, возвращаемый `locals()`, в одних случаях могут влиять на реальные переменные, а в других – нет (в частности, внутри функции в CPython до версии 3.11 изменения в `locals()` не отражались на реальных локальных переменных). Функция `dir()` без аргумента возвращает список имен в текущей локальной области видимости [13]. Это удобно для интерактивного исследования – например, `dir()` на уровне модуля покажет все доступные имена (переменные, функции, импортированные модули и т.п.), кроме имен встроенных функций и исключений [14]. Если вызвать `dir(some_object)`, она возвращает список атрибутов данного объекта (используя метод `object.__dir__` или анализируя `__dict__` объекта) [15]. Пример:

```
import math
print(dir(math)[:5])   # первые несколько имён внутри модуля math
# ['__doc__', '__loader__', '__name__', '__package__', '__spec__', ...]
```

Идиома `if __name__ == "__main__":`**.** В каждом модуле Python есть специальная глобальная переменная `__name__`. Если модуль запускается как основная программа, то `__name__` устанавливается в строку `"__main__"`. Если модуль импортируется, то `__name__` будет равно имени модуля (например, `"os"`, `"sys"` и т.д.) [16]. Поэтому часто в конце скрипта пишут:

```python
if __name__ == "__main__":
    # выполнить некоторый код (тестирование или запуск функций)
    ...
```

Этот условный блок выполнится только когда файл запускается напрямую, но не при импорте этого файла как модуля в другую программу. Таким образом можно, например, разместить в блоке под `if __name__ == '__main__'` код, который тестирует функции модуля или выполняет основной сценарий, а при импорте этого модуля эти действия не будут происходить (потому что тогда `__name__` не равен `"__main__"`). Это распространённая практика для создания модулей, которые можно использовать повторно, но при самостоятельном запуске они демонстрируют свою работу. Подробнее: интерпретатор устанавливает `__name__ == "__main__"` для "точки входа" – первого запускаемого модуля [17], либо для интерактивного режима (где также `__name__` равно `"__main__"`).

Оператор присваивания выражения (морж-оператор) `:=` и его область видимости. Оператор `:=`, появившийся в Python 3.8, позволяет выполнять присваивание внутри выражений (например, внутри условий `if`, списковых включений и т.д.). Важно понимать, что с точки зрения области видимости переменные, которым присваивается значение через `:=`, ведут себя как обычные переменные присвоения. В частности, они подчиняются тем же правилам LEGB: если вы используете `:=` внутри функции, то без объявления `global` или `nonlocal` переменная станет локальной для этой функции [18]. Если до этого она была объявлена как global, то `:=` присвоит значение глобальной переменной [18]. Особенности появляются при использовании `:=` внутри генераторов и comprehensions (генераторные выражения, списковые включения и т.п.). Например, рассмотрим списковое включение:

```python
results = [(x, y, x/y) for x in data if (y := f(x)) > 0]
```

Здесь переменная `y` используется в условии `if`. Согласно спецификации, при использовании `:=` внутри включений **переменная привязывается не к локальной области самого включения, а к содержащей области** (например, к функции, где находится это включение, или к глобальному модулю, если включение на уровне модуля) [19]. Это означает, что после выполнения такого спискового включения переменная `y` будет доступна вне него (в окружающей области). В приведённом примере, если код находится на уровне модуля, после вычисления списка имя `y` останется определено в глобальном пространстве. Такое поведение задумано разработчиками: PEP 572 явно указывает, что переменная, присвоенная через `:=` в comprehension, становится именем в объемлющей области видимости [20] [19]. В результате, хотя переменные цикла в списковых включениях не "выходят" наружу (начиная с Python 3 они локальны для включения), переменные, присвоенные через морж-оператор, наоборот, сохраняются во внешней области. Следует иметь это в виду, чтобы не столкнуться с неожиданным "утечкой" переменных из comprehension. В других контекстах (например, в условии `if` или цикле `while`) оператор `:=` не создает новой области, а просто выполняет присваивание в текущей области, как эквивалентный оператор `=`. Пример использования `:=`:

```python
if (n := len(some_list)) > 10:
    print(f"Длина списка {n} слишком велика")
# Здесь переменная n будет доступна после if, т.к. она создана в окружающей области (например, глобально или в функции).
```

В этом примере `n` появляется после `if` – она находится в той же области, где и сам оператор `if` (например, глобально).

Атрибут `func.__code__.co_varnames`. У каждой функции в Python есть атрибут `__code__`, который представляет объект кода (bytecode) этой функции. У объекта `__code__` есть атрибуты, среди которых `co_varnames` – кортеж, содержащий имена локальных переменных и аргументов функции, а `co_names` – кортеж имен других использованных имен (например, глобальных или внешних) [21]. Проще говоря, `co_varnames` хранит имена всех переменных, определённых в **локальной области** функции (включая параметры), а `co_names` – имена, на которые функция ссылается, но которые не являются локальными (то есть потенциально глобальные или свободные переменные) [21]. Например:

```
a = 1
def f(b):
    c = a + b
    return c

print(f.__code__.co_varnames)  # ('b', 'c')
print(f.__code__.co_names)     # ('a',)
```

Здесь `co_varnames` вернул `('b','c')` – имена аргумента и локальной переменной `c`, а `co_names` – `('a',)`, поскольку `a` не локальная, а берётся из глобальной области [21]. Такое разделение позволяет интерпретатору эффективно различать локальные и глобальные переменные при выполнении.

Компиляция и байткод (пример с `dis`). При импортации модуля или определении функции Python компилирует код в байт-код, который затем исполняется виртуальной машиной. Распределение переменных по областям видимости определяется на этапе компиляции. Например, если внутри функции переменная не объявлена как `global`, компилятор помечает ее как локальную, и все операции с ней станут быстрыми операциями работы с локальными переменными (LOAD_FAST, STORE_FAST). Глобальные же переменные или переменные внешних областей компилируются как более “дорогие” операции (LOAD_GLOBAL, LOAD_DEREF и т.д.). Рассмотрим функцию `f` из предыдущего примера и посмотрим её байткод:

```
import dis
def f(b):
    c = a + b
    return c

print(dis.code_info(f))
print(dis.dis(f))
```

Вывод (CPython 3.10) будет примерно таким:

```
Name:              f
Filename:          test.py
Argument count:    1
```

```
Locals:            b, c
... <прочие данные> ...

  2           0 LOAD_GLOBAL              0 (a)
              2 LOAD_FAST                0 (b)
              4 BINARY_ADD
              6 STORE_FAST               1 (c)
  3           8 LOAD_FAST                1 (c)
             10 RETURN_VALUE
```

В строке `0 LOAD_GLOBAL 0 (a)` видно, что переменная `a` загружается как глобальная (поскольку `a` не локальная для `f`, она находилась вне функции). Переменная `b` загружается как локальная (`LOAD_FAST 0 (b)`), а результат операции складывания сохраняется в локальную `c` (`STORE_FAST 1 (c)`). Далее `c` возвращается (`LOAD_FAST 1 (c)` и `RETURN_VALUE`). Таким образом, **локальные переменные компилируются как "FAST"**, а обращение к глобальным – через инструкцию `LOAD_GLOBAL` с индексом в кортеже `co_names`. Если бы внутри функции были обращение к переменной внешней функции, это было бы отражено инструкцией вроде `LOAD_DEREF` (для *free variable* – свободной переменной из замыкания). Использование модуля `dis` позволяет наглядно увидеть, как Python разграничил переменные по областям видимости на этапе компиляции.

2. Замыкания (Closures)

Что такое замыкание. *Замыкание* – это функция, которая **сохраняет** ссылку на переменные из охватывающей (внешней) области видимости, даже после того как внешняя функция завершила выполнение. В Python замыкания возникают, когда есть вложенная функция, использующая переменные из объемлющей функции, и объемлющая функция возвращает эту вложенную функцию. Вложенная функция **замыкает** (закрывает) значения переменных из внешней области, делая их доступными во время своего исполнения вне исходного контекста. Простыми словами, замыкание – это функция + привязанные значения переменных из окружающей среды.

Пример замыкания:

```python
def make_adder(x):
    def add(y):
        return x + y
    return add

add5 = make_adder(5)
print(add5(10))  # 15
print(add5(3))   # 8
```

Здесь `make_adder` возвращает функцию `add`, которая прибавляет к аргументу `y` некое сохранённое значение `x`. При вызове `make_adder(5)` создаётся замыкание, где `x=5` удерживается внутри функции `add`. Переменная `x` находится во внешней области относительно `add`, и хотя выполнение `make_adder` завершилось, `add5` по-прежнему "помнит" значение `x=5` – за счёт замыкания. Вызовы `add5` с разными `y` продолжают использовать закрытую переменную `x=5`.

Где хранятся переменные замыкания. Технически, когда функция имеет *свободные переменные* (free variables – те, что не являются ее локальными и не являются глобальными, а определены в объемлющей области), интерпретатор при создании функции создает для них специальные объекты – *ячейки* (cell objects). Эти ячейки содержат ссылки на реальные переменные во внешней функции. Атрибут функции `__closure__` как раз и хранит кортеж таких ячеек для замыкания [22]. Каждая ячейка (`cell`) ссылается на соответствующую переменную во внешней функции [22]. Когда внешняя функция завершает работу, переменные обычно бы удалились, но если на них есть ссылки в ячейках замыкания, они остаются доступны через эти ячейки. То есть, замыкание "удерживает" объекты переменных, не давая им быть сборщику мусора уничтоженными после выхода из внешней функции.

Свободные переменные (free variables). Так называют переменные, используемые внутри функции, но не являющиеся её локальными. В примере выше для `add(y)` свободной переменной является `x` (оно не определено внутри `add`, а берётся извне). Для замыкания важно, чтобы свободная переменная не была глобальной, а принадлежала некоторой внешней *не глобальной* области – обычно это переменная внешней функции (как `x` принадлежит `make_adder`). Свободные переменные записаны в атрибуте функции `__code__.co_freevars` – это кортеж имён таких переменных [23]. Например, у функции `add5.__code__.co_freevars` будет `('x',)`. Сами же значения хранятся в `__closure__`.

Как получить имя и значение переменной из замыкания. Используя упомянутые атрибуты: `func.__code__.co_freevars` содержит имена свободных переменных, а `func.__closure__` – кортеж ячеек с их значениями. У объекта cell есть атрибут `cell_contents`, в котором и лежит сохранённое значение. Продолжая пример:

```
print(add5.__code__.co_freevars)    # ('x',)
print(add5.__closure__[0].cell_contents)  # 5
```

Мы видим, что замыкание `add5` содержит одну свободную переменную `'x'`, а в соответствующей ячейке хранится значение `5`. Если бы было несколько свободных переменных, они бы располагались в одном порядке в `co_freevars` и в `__closure__`. Пример посложнее:

```
def outer(a, b):
    x = a + b
    def inner(c):
        return x + c
    return inner

func = outer(2, 3)
print(func.__code__.co_freevars)           # ('x',)
print(func.__closure__[0].cell_contents)   # 5 (т.е. a+b)
```

Здесь `inner` замкнуло переменную `x` из `outer`. После вызова `outer(2,3)` значение `x=5` сохранено в замыкании функции `func`. Заметьте, что свободные переменные – это те, которые не определены внутри функции и не являются глобальными; если бы `inner` использовала глобальную переменную, она не считалась бы free variable (Python разрешает глобальные переменные через механизм поиска в `globals()`).

Почему говорят о замыканиях и зачем они нужны. Замыкания позволяют создавать функции с состоянием – то есть, “памятью” о предыдущих вычислениях или о внешнем контексте, без использования глобальных переменных. Это мощная концепция функционального программирования. Например, с помощью замыкания мы можем настроить функцию с определёнными параметрами заранее (как `make_adder(5)` генерирует “добавлятель 5”). Другой случай – при создании декораторов: когда декоратор определяет функцию-обёртку, которая замыкает переменные (например, оригинальную функцию, какие-то настройки декоратора и т.д.). Такие обёртки удерживают контекст, необходимый для работы декоратора. Замыкания также часто используются для создания фабрик функций, функций-счётчиков, сохранения промежуточных данных между вызовами и т.д. В Python замыкания реализуются прозрачно – вы просто используете переменные из внешней области, и если функция возвращается, эти переменные сохраняются в `__closure__`. Понимание того, что происходит "под капотом", помогает, например, в отладке: можно проверять `func.__closure__` чтобы узнать, какие значения замкнуты, или понимать, почему переменная не подхватилась (например, если внутри функции вы присвоили значение имени, не объявив `nonlocal`, то оно перестаёт быть свободной и становится локальной, что часто приводит к ошибкам). Поэтому о замыканиях говорится во всех курсах продвинутого Python: это фундаментальный механизм, позволяющий функции сохранять контекст.

3. Импорт в Python

Файл `__init__.py` и переменная `__all__` в пакетах. Наличие файла `__init__.py` в каталоге делает его пакетом (package) – грубо говоря, сообщает Python, что этот каталог следует рассматривать как модуль-пакет, из которого можно импортировать субмодули. В Python 3.3+ для поддержки *namespace packages* файл `__init__.py` не обязателен, но в классических пакетах он обычно присутствует. Этот файл может быть пустым, либо выполнять инициализирующий код при импорте пакета (например, устанавливать определённые переменные). Кроме того, в `__init__.py` можно определять список `__all__` – это список строк, определяющий, какие подмодули или атрибуты будут экспортированы при выполнении `from package import *` [24]. Если `__all__` определён, то конструкция `from mypack import *` импортирует **только** названные в списке имена (например, имена подмодулей или объекты, явно импортируемые/определённые в `__init__.py`) [24] [25]. Если переменная `__all__` не определена, то поведение по умолчанию при `import *` для пакета состоит в том, что **не импортируется ничего, кроме самого пакета** (т.е. подмодули не подтягиваются автоматически) [26]. В таком случае `from package import *` лишь выполнит `__init__.py` (инициализирует пакет) и импортирует имена, явно присвоенные в `__init__.py`, но не загрузит автоматически субмодули пакета. Таким образом, `__all__` – механизм экспорта API пакета при звездочном импорте. Обычно считается хорошей практикой явно определять `__all__` в пакетах/модулях, если вы хотите контролировать, что считается публичным API.

Пример: предположим, в `mypack/__init__.py` написано `__all__ = ["module1", "module2"]`. Тогда `from mypack import *` импортирует подмодули `mypack.module1` и `mypack.module2` (и присвоит имена `module1`, `module2` в текущем пространстве) [25]. Если же `__all__` не указано, `from mypack import *` ничего из подмодулей не импортирует, хотя модули пакета можно импортировать явно по именам.

Соглашения об именах для “непубличных” сущностей. В Python нет строгого механизма ограничения видимости, но принято использовать соглашение: имена, начинающиеся с символа подчёркивания `_`, считаются *internal* или “не предназначенными для внешнего использования”. Это относится как к переменным и функциям внутри модуля, так и к самим модулям/пакетам.

Например, файл `_util.py` в пакете может обозначать, что это внутренний модуль, и прямое его использование вне пакета не предполагается. На уровне from-import тоже есть эффект: конструкция `from module import *` **не будет импортировать имена, начинающиеся с** `_` [27] (если только модуль сам явно не определил их в `__all__`). Таким образом, одинарное подчеркивание служит "мягким" модификатором приватности. Двойное подчёркивание в начале имени (`__name`) запускает механизм *name mangling* в классах, но для модулей такой паттерн обычно не используется (в модулях достаточно одного `_`). Важно помнить, что это лишь соглашение: вы всё равно можете импортировать или использовать `_`-имена, Python не запрещает (кроме описанного случая со `import *`). Но разумный программист расценит такое имя как не входящее в публичный API модуля.

Функция `__import__`**.** Это встроенная функция Python, которая является низкоуровневой основой импорта. Когда вы пишете `import math`, это эквивалентно вызову `__import__("math")` (с дополнительными параметрами, передаваемыми интерпретатором). В общем случае синтаксис `__import__(name, globals=None, locals=None, fromlist=(), level=0)` соответствует семантике оператора `import` [28]. Обычно программисты **не используют** `__import__` **напрямую**, потому что есть более удобные и безопасные способы динамического импорта (например, модуль `importlib`). Кроме того, прямой вызов `__import__` несколько громоздок и легко сделать ошибку с его параметрами. Тем не менее, знать о нём полезно: через `__import__` можно, например, импортировать модуль по строковому имени, заданному в runtime. Пример: `m = __import__("os")` загрузит модуль `os` (и присвоит его также в `sys.modules`). Если хотим эквивалент `from package import submodule`, надо использовать параметр `fromlist`: например, `pkg = __import__('mypack', globals(), locals(), ['submod'], 0)`. Обычно вместо этого предпочитают функцию `importlib.import_module`:

```
import importlib
module_name = "os"
os_module = importlib.import_module(module_name)
```

В большинстве случаев рекомендуется именно `importlib.import_module` (которая внутри вызывает `__import__`, но предоставляет более простой интерфейс) [29]. Кстати, PEP 328 и документация отмечают, что непосредственное использование `__import__` считается низкоуровневым и **нежелательным**, предпочтительнее `importlib` [29].

Абсолютные и относительные импорты. *Абсолютный импорт* – когда указывается полный путь до модуля относительно корня пространств имён Python. Например, если у вас структура проекта имеет пакет `mypack` с модулем `utils.py`, то абсолютный импорт внутри другого модуля: `import mypack.utils` или `from mypack import utils`. *Относительные импорты* используют синтаксис с точками, чтобы импортировать относительно текущего модуля/пакета. Одна точка `.` означает текущий пакет, две – родительский пакет и т.д. Например, внутри пакета `mypack` в модуле `core.py` вы можете написать `from . import utils` (импорт модуля `utils` из того же пакета) или `from .subpack import mod` (импорт из под-пакета текущего пакета). Две точки: `from .. import base` – подняться на уровень выше. Относительные импорты позволяют организовывать модули в пакетах без необходимости писать полные имена пакетов, но они работают только внутри пакета (нельзя использовать относительный импорт в скрипте, запущенном напрямую – только в модуле, являющемся частью пакета). Пример: структура:

```
mypack/
    __init__.py
    core.py
    utils.py
    subpack/
        __init__.py
        mod.py
```

В `core.py` можно сделать `from . import utils` (импортирует `mypack.utils`). В `subpack/mod.py` можно сделать `from .. import utils` (импортирует модуль `mypack.utils` из родительского пакета). Абсолютно же это было бы `from mypack import utils`. Начиная с Python 3, **все импорты внутри пакета по умолчанию считаются абсолютными**, если не использовать явный `.` или не настроить `__package__`. В Python 2 допускались двусмысленные ситуации, поэтому сейчас лучше явно выбирать относительный импорт, если нужно.

Где располагаются установленные библиотеки (site-packages и др.). Когда мы устанавливаем пакеты через pip или иной менеджер, они обычно помещаются в специальный каталог, входящий в `sys.path`. В CPython стандартные пути включают директорию стандартной библиотеки и директории *site-packages*. Например, для системного Python на Linux это обычно что-то вроде `/usr/lib/python3.X/site-packages/` (для X.Y версии Python) [30] [31]. В среде virtualenv или Conda – свои каталоги. Узнать, какие пути используются для поиска модулей, можно посмотрев `sys.path` – это список строк директорий, которые обходятся при импорте [32]. Туда входят и папки site-packages, и, например, текущая рабочая директория (обычно первым элементом `""` или `"."` – текущий путь). `site-packages` – стандартное название папки для сторонних (устанавливаемых) пакетов. Также могут быть пути для сторонних пакетов OS (например, `/usr/local/lib/python3.X/dist-packages` и т.п.). Важно: Python формирует `sys.path` при запуске, включая директории из переменной окружения `PYTHONPATH` (если есть), стандартные пути установки и, для запускаемого скрипта, добавляет директорию скрипта (или текущую директорию, если запущен интерактивно). Например, при выполнении `python -m module` текущий рабочий каталог добавляется в начало `sys.path` [33].

Вы можете проверить, где установлен конкретный модуль, посмотрев его атрибут `__file__`. Например:

```
import requests
print(requests.__file__)
```

может показать путь в site-packages, где лежит `requests`. Некоторые модули – встроенные (built-in, написаны на C и встроены в интерпретатор) – у них атрибут `__file__` отсутствует, так как они не загружаются с файловой системы.

Словарь `sys.modules`. Python кэширует импортированные модули в словаре `sys.modules`. Это глобальный кэш: ключи – имена модулей, значения – объекты модулей. Каждый новый импорт сначала проверяет, нет ли модуля в `sys.modules` (и если есть, обычно берёт его оттуда, не загружая второй раз) [34]. Благодаря этому повторные `import` одного и того же модуля не выполняют повторно весь код модуля, а просто получают ссылку на уже загруженный модуль. `sys.modules` можно использовать для различных трюков: например, можно подменить там запись, чтобы изменить модуль, или удалить запись, чтобы заставить Python заново загрузить

модуль при следующем импорте. Однако обычно напрямую редактировать `sys.modules` не рекомендуется без очень веских причин. Для перезагрузки модулей лучше использовать специальные функции (см. ниже). Но понимать `sys.modules` полезно: например, при динамическом импорте, если модуль уже загружен, `importlib.import_module` все равно вернет объект из `sys.modules`. Также, если вы модифицировали код файла модуля и хотите обновить модуль в запущенном интерпретаторе, нужно либо удалить его из `sys.modules` и сделать import, либо воспользоваться `importlib.reload`. Отметим, что если вы хотите пройтись по всем загруженным модулям, можно итерироваться по `sys.modules`, однако он может меняться во время работы программы (особенно в многопоточных сценариях), поэтому для надежности лучше итерировать по копии списка ключей [35].

Пример использования `importlib.reload`. Модуль `importlib` предлагает функцию `reload(module)`, позволяющую перезагрузить уже импортированный модуль. Например:

```
import importlib, mymodule
# ... внести изменения в код mymodule.py (например, в ходе отладки)
importlib.reload(mymodule)
```

Вызов `reload` выполнит повторно код модуля (снова прочитает файл `.py`) и обновит объект модуля. Все ссылки на объекты внутри модуля (например, полученные ранее классы, функции) при этом не обновляются автоматически, только сам объект модуля и его атрибуты будут перезагружены. `reload` полезен в интерактивных сессиях (например, Jupyter/Colab), чтобы применить изменения без перезапуска интерпретатора.

Функция `importlib.import_module(name)` позволяет импортировать модуль по строковому имени. Она аналогична `__import__`, но проще в использовании. Например, `import_module('math')` вернет объект модуля `math`. Можно также импортировать вложенный модуль, указав полное имя через точку, например `import_module('mypack.mymodule')`. Эта функция особенно удобна, когда имя модуля вычисляется динамически.

Файлы и магия `%%writefile` в Jupyter/Colab. В среде Google Colab или Jupyter Notebooks нередко используют *команду магии* `%%writefile filename.py` для записи содержимого ячейки в файл. Это просто удобный способ создать модуль в файловой системе из ячейки ноутбука. После выполнения такой ячейки появится файл `filename.py` в текущей рабочей директории (которую можно узнать через `!pwd` либо `import os; print(os.getcwd())`). Затем этот файл можно импортировать обычным образом: `import filename` (без `.py`). Будьте внимательны: если вы изменили код и выполнили снова `%%writefile` (перезаписали файл), интерпретатор не узнает автоматически об изменениях. Вам либо нужно перезапустить kernel, либо использовать `importlib.reload` для модуля, либо (нежелательно) удалить запись из `sys.modules`. При первом импорте после создания файла модуль загрузится как обычно.

Связь между импортом и `sys.modules`. Когда вы делаете `import some_module`, Python создаёт новый объект модуля (типа `module`), выполняет код из `some_module.py` внутри него (инициализируя атрибуты) и затем сохраняет этот объект в `sys.modules` под ключом `"some_module"`. Если импортируете модуль второй раз, Python увидит запись в `sys.modules` и не будет выполнять код снова, а просто вернёт уже загруженный модуль. Именно поэтому `reload` необходимо, когда вы хотите, чтобы код выполнился заново. Если вы динамически формируете модуль (например, создаёте новый module объект вручную или копирует уже

загруженный), можно поместить его в `sys.modules` – тогда при импорте по имени вы получите именно его. Это довольно продвинутый трюк, используемый, например, при создании псевдомодулей (модуль, написанный на Python, который подменяет себя на объект написанный на C и т.п.). В обычной практике достаточно помнить: **каждый загруженный модуль хранится в** `sys.modules`, и если что-то "не импортируется как ожидается", стоит проверить не находится ли модуль уже там, но под другим именем или пустой.

Версия модуля. Часто возникает задача узнать версию установленного пакета/модуля. По соглашению, многие библиотеки определяют в своём модуле переменную `__version__`, содержащую версию (строкой). Например, `import numpy; print(numpy.__version__)`. Однако это не унифицировано, и некоторые могут не иметь `__version__` или хранить версию в другом месте (например, `pandas.__version__` есть, а у некоторых модулей надо смотреть атрибут `version` или константу в коде). В Python 3.8+ можно использовать стандартный способ: модуль `importlib.metadata`. В нём есть функция `version("package-name")`. Например:

```
from importlib import metadata
print(metadata.version('numpy'))
```

выведет строку версии установленного пакета numpy (например, `'1.23.4'`). Этот способ получает информацию из метаданных установленных пакетов (та же, что выводит `pip show`). Если пакет не найден, будет исключение `metadata.PackageNotFoundError`. Ещё вариант – утилита командной строки: `pip show module_name` или `pip list`. Но программно `importlib.metadata.version` наиболее прямой и поддерживаемый. Также обратите внимание: *версии встроенных модулей* (например, `sys` или `math`) – они не имеют понятия "версии", т.к. поставляются с Python (их версия соответствует версии Python). Для Python-пакетов же, устанавливаемых отдельно, версии обычно определяются.

Специальная переменная `__name__` **в модулях.** Мы уже обсудили её для `__main__`, но стоит повторить: каждый модуль при загрузке получает атрибут `__name__` – обычно равный полному имени модуля. Например, если модуль `math` написан на C и встроен, у него `math.__name__ == "math"`. Для пакета `mypack.submod` будет `"mypack.submod"`. Внутри модуля это значение можно использовать, чтобы понять, кто ты (обычно нужно лишь для сравнения с `"__main__"`). Также у каждого модуля есть атрибут `__file__` (если модуль загружен с файла) – путь к файлу, из которого модуль загружен, и `__package__` – имя пакетного пространства (для пакетов оно равно имени пакета, а для модулей внутри пакета – имени пакета, а не всего модуля). `__doc__` хранит строку документации модуля (если в начале файла есть строковый литерал, он становится `__doc__`). `__annotations__` – словарь для аннотаций модульных переменных (о нём далее). Эти специальные атрибуты позволяют introspection – например, можно программно выяснить, где лежит модуль через `mod.__file__`, или прочитать его docstring через `mod.__doc__`.

7. Структура проекта (модулей и пакетов)

(Пропущена нумерация 4-6, продолжаем как указано в запросе, предполагая, что 7 – следующий раздел.)

Назначение `__init__.py` **в пакете.** Как отмечалось, `__init__.py` делает папку пакетом. Он может выполнять инициализацию (например, импортировать удобства). Например, иногда в

`__init__.py` разработчики сами импортируют часть подмодулей, чтобы упростить доступ к ним. Но злоупотреблять этим не стоит – лучше явно импортировать то, что нужно, или пользоваться `__all__`. В небольших проектах `__init__.py` может быть пустым; в больших – может настраивать логирование, проверять версию, инициализировать субпакеты.

Переменная `__all__` в пакетах и модулях. Мы разобрали её роль для пакетов (в `__init__.py`). В обычных модулях (файлах) `__all__` тоже можно определить – тогда `from module import *` будет импортировать только имена из этого списка, а не все глобальные объекты модуля. Это можно использовать, чтобы спрятать какие-то глобальные переменные/вспомогательные функции: поместите в `__all__` только имена “публичных” объектов. Так `import *` станет чуть безопаснее (но всё равно его применять надо с осторожностью).

Документация модуля `__doc__`. Если в начале файла модуля поставить строку или тройные кавычки с текстом, это будет **докстринг модуля**. Он доступен как атрибут `module.__doc__`. Например, в модуле:

```
"""Модуль для демонстрации docstring."""
# ...код модуля...
```

теперь `module.__doc__` вернет эту строку. Стандартные инструменты (help(), pydoc) выводят `__doc__` для модуля при запросе справки по нему. Докстринг следует писать коротко, объясняя назначение модуля.

Аннотации в модуле `__annotations__`. В Python 3 появилось понятие аннотаций переменных (PEP 526). Это касается и глобальных переменных модуля. Если в модуле вы делаете аннотированное присваивание, например:

```
count: int = 0
```

то Python сохраняет аннотацию `{"count": int}` в словаре `__annotations__` модуля [36] (при этом значение переменной `count` равно 0, а аннотация типа – int). Аналогично с функциями: их аннотации параметров и возвращаемого значения хранятся в `function.__annotations__`. Модульный `__annotations__` – просто словарь, который вы можете использовать для introspection или для библиотек (например, dataclass может считывать аннотации). Если в модуле нет аннотированных переменных, `__annotations__` может отсутствовать или быть пустым. Обратите внимание: аннотации не влияют на выполнение программы (они для проверки типов, документации и т.д.). Кроме того, существует специальная переменная `__annotations__` и на уровне класса (для атрибутов класса, объявленных с типами).

Поиск пути к модулю во время выполнения. Чаще всего достаточно посмотреть атрибут `module.__file__` – он хранит путь к файлу, откуда модуль был загружен (абсолютный путь, либо относительный, но обычно Python старается сохранить абсолютный) [37]. Например:

```
import json
print(json.__file__)
# /usr/lib/python3.10/json/__init__.py
```

Таким образом, мы узнали, где находится модуль `json`. У пакетов `__file__` обычно указывает на файл `__init__.py` внутри пакета. У динамически загруженных расширений (например, `.so` или `.pyd` файлов) `__file__` укажет на них. Если модуль является встроенным (built-in, написан на C и встроен в сам Python, как `math` или `sys`), атрибут `__file__` у него отсутствует – вместо этого есть атрибут `__spec__`, где `origin` может быть `"built-in"` или аналогичный маркер. Но в общем случае для "обычных" модулей `__file__` – надёжный способ получить путь.

Если модуль загружен из zip-архива или другого нестандартного источника, `__file__` может иметь особый формат (например, `myarchive.zip/mymodule.py`). Но для локальных файлов – это путь в файловой системе. Также есть `module.__spec__` (объект спецификации загрузки модуля), где можно найти `origin` (происхождение) и `loader`.

Еще один случай: у пакетов есть атрибут `__path__` – это список путей, где искать подпакеты. Обычно он содержит одну директорию – путь к пакету. Но у *namespace packages* `__path__` может быть списком нескольких директорий (когда один логический пакет объединён из нескольких физических мест). Однако для большинства задач достаточно `__file__` (для модулей) и `__path__` (для пакетов) – их можно напечатать и посмотреть.

8. Внутреннее устройство Python (объектная модель)

Python – язык с динамической типизацией, но под капотом у CPython (стандартной реализации) всё является структурами на C. В частности, каждый объект в CPython представлен структурой `PyObject` или её расширением.

PyObject, PyTypeObject и базовые типы. В **CPython** все объекты (числа, строки, пользоват. объекты, функции и т.д.) начинаются с структуры `PyObject` (или `PyVarObject` для объектов переменной длины). `PyObject` включает в себя как минимум два поля: счетчик ссылок `ob_refcnt` и указатель на объект типа `ob_type` [38]. То есть у каждого объекта есть ссыльность (сколько ссылок на него держится) и указатель на структуру типа, которая описывает, что это за объект и как с ним работать. Тип объекта – тоже объект! В CPython типы (классы) представлены структурой `PyTypeObject` [39]. Это большая структура (десятки полей), содержащая указатели на функции, реализующие поведение типа: размер объекта, методы, операции, флаги и пр. Глобально в интерпретаторе определены ключевые экземпляры `PyTypeObject` для всех built-in типов. Например, `PyLong_Type` – структура типа для всех целых (`int`), `PyList_Type` – для списков, и т.д. [40]. Среди них есть особенно важные: `PyType_Type` – это **тип объекта "type"**, то есть метакласс всех классов, и `PyBaseObject_Type` – базовый тип всех классов (соответствует встроенному классу `object`) [41] [42].

Связи между ними можно представить так: встроенный класс `object` в Python (то, что вы получаете, когда пишете `class A: pass`, не указывая родителя – он наследует от `object`) соответствует `PyBaseObject_Type` на уровне C [43]. Этот `PyBaseObject_Type` сам является экземпляром (`ob_type`) `PyType_Type` [44]. А `PyType_Type` – это структура типа для самого класса `type` (в Python это встроенный метакласс). Удивительный факт: `PyType_Type` является экземпляром самого себя. То есть в Python `type(type) is type` – класс `type` является экземпляром самого себя (это делает объектную модель замкнутой) [45]. В терминах C: глобальная переменная `PyType_Type` (структура) представляет класс `type` и имеет свой `ob_type` указывающий на себя же [45]. Этот трюк позволяет описать идею "все классы являются объектами, и у них есть свой класс (метакласс)". Визуально это можно представить так:

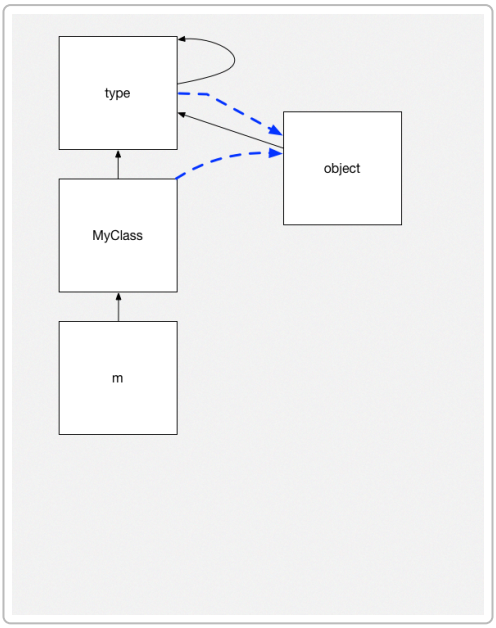

Схема отношения объектов и классов в Python: квадратами обозначены объекты (со своими типами), синие пунктирные стрелки – отношение "наследует от", черные стрелки – "является экземпляром". Класс `MyClass` *наследует от* `object` *(синие стрелки к* `object`*),* `m` *– экземпляр* `MyClass` *(черная стрелка вверх). Класс* `MyClass` *сам является экземпляром* `type` *(черная стрелка к* `type`*). Класс* `type` *наследует от* `object` *(синие стрелки) и является экземпляром себя (черная стрелка закольцована). Базовый* `object` *является экземпляром* `type`*.* [46] [47]

На схеме показано: `m` – объект пользовательского класса `MyClass`. Его тип (`__class__`) – `MyClass`. У `MyClass` тип – `type` (потому что это класс, а все классы – экземпляры метакласса `type`). `MyClass` наследует от `object` (пустой класс неявно наследует от `object`, то есть `MyClass.__bases__ == (object,)`). Класс `type` тоже наследует от `object` (в Python `type.__bases__ == (object,)`), то есть `object` – вершина иерархии классов (ни от кого не наследует, `object.__bases__ == ()`). При этом `object` – экземпляр `type` (как и все классы). Получается замкнутая структура: `object` <- наследование – `type` (потому что `type.__bases__` включает `object`), а `type` <- инстанцирование – `type` (он сам себе экземпляр), и `object` <- инстанцирование – `type`. Благодаря этому любой объект в системе имеет цепочку: у экземпляра есть класс (пользовательский или встроенный), у класса – метакласс (обычно `type`), у `type` – метакласс он сам, и `type` наследует от `object`. Все дороги так или иначе ведут к `object` и `type`.

С точки зрения CPython: `PyBaseObject_Type` и `PyType_Type` – глобальные структуры. `PyBaseObject_Type.tp_base == NULL` (у object нет базы), а `PyType_Type.tp_base == PyBaseObject_Type` (у type базовый класс – object). Поле `ob_type` у `PyBaseObject_Type` указывает на `PyType_Type` (т.е. класс object является экземпляром type) [47]. А `PyType_Type.ob_type` указывает на себя (type как экземпляр сам себя) [45]. Эти связи устанавливаются при инициализации интерпретатора.

PyTypeObject и поля слотов. `PyTypeObject` содержит много полей, часто называемых *slot* – это указатели на функции C, реализующие, например, операции сложения, атрибуты, методы и т.д. (Например, поле `tp_as_number` указывает на таблицу, где есть указатели на функцию реализации `nb_add` – операции `+` для данного типа, если она поддерживается). В исходниках CPython `Include/object.h` и `Include/typemodules` определены все поля. Размер

`PyTypeObject` довольно большой (сотни байт) [48], но для каждого конкретного типа многие поля могут быть нулевыми (не используются). Статически определённые типы (например, `PyLong_Type`) обычно инициализируются структурным литералом в C (в CPython 3.x эти объекты создаются и инициализируются при старте). Есть также *heap types* – типы, создаваемые динамически, например, через выражение `class` в Python (они тоже имеют тип `PyType_Type`, но создаются во время выполнения, и их `PyTypeObject` находится в куче). Но принципы одинаковы.

PyObject и PyVarObject. `PyObject` – базовая структура, как уже сказано, содержит счетчик ссылок и указатель на тип [38]. Макрос `Py_INCREF` увеличивает счетчик, `Py_DECREF` уменьшает и при достижении 0 удаляет объект. `PyVarObject` расширяет `PyObject` добавлением поля `ob_size` (размер переменной части) [49]. Это используется для последовательностей: списков, строк, кортежей и т.д., где нужно хранить длину. Многие API-функции принимают `PyObject*` – то есть указатель на начало структуры объекта. Через него можно получить тип (`Py_TYPE(obj)` – макрос для `obj->ob_type`), а далее обращаться к полям типа или, сделав каст к конкретному структуре, к его полям.

Почему важно понимать внутреннее устройство. При обычном использовании Python вам не нужно заботиться об этих деталях – интерпретатор всё делает сам. Но для продвинутого понимания полезно знать, что класс в Python – тоже объект, поэтому его можно передавать, изменять атрибуты, динамически создавать новые методы, и эти изменения отражаются на всех экземплярах, потому что экземпляры хранят ссылку на объект класса. Также понимание того, что у каждого объекта есть ссылочный счетчик, объясняет необходимость осторожности с циклическими ссылками и работу сборщика мусора (хотя сейчас CPython помимо reference counting имеет и cycle GC). Кроме того, зная, что внутри `list` или `dict` – C-структуры, вы можете лучше оценивать сложность операций (хотя для этого обычно достаточно знать алгоритмические оценки, но иногда стоит понимать, что, например, при увеличении списка может произойти реаллокация).

Для любопытных: исходный код CPython весьма информативен. Например, на GitHub можно найти определения `PyTypeObject` и глобальных `PyType_Type` и `PyBaseObject_Type` (они там помечены комментариями *built-in 'type'* и *built-in 'object'* [43]). Таким образом, внутренняя объектная модель Python весьма последовательна и, несмотря на динамичность, определяется вполне конкретными структурами и соотношениями между ними.

9. Численные типы в Python (int, float, complex)

Создание и хранение целых чисел (int). В Python 3 целые числа имеют неограниченную точность (big integers). Это реализовано с помощью структуры `PyLongObject`, которая представляет собой массив "цифр" в некоторой базе, хранящий абсолютное значение числа, и отдельный знак. В CPython база хранится в виде типа `digit` (обычно 30 бит на каждую "цифру" на платформах с 32-битными словами, либо 15 бит на 16-битных – детали можно найти в `longobject.c`). Таким образом, число произвольной длины занимает пропорционально памяти: например, число порядка 2^30 помещается в одном "слоте", 2^60 – в двух, и т.д. Все операции реализованы на уровне C (с функциями для сложения, умножения, деления больших чисел). В официальной документации указано: "Все целые реализованы как объекты типа long произвольной точности" [50], а `PyLongObject` – это структура, представляющая целое число [51]. Для удобства и производительности, CPython кеширует небольшие целые числа. **Кеш малых int:** текущее выполнение CPython по умолчанию кеширует целые от -5 до 256 включительно [52]. Эти объекты создаются заранее и при использовании числа в этом диапазоне интерпретатор просто

даёт ссылку на уже существующий объект. Например, выражение `a = 256; b = 256` приведёт к тому, что `a` и `b` ссылаются на один и тот же объект (а `257` уже будет разными объектами для разных появлений, если не сохраняется в переменную). Это сделано, потому что такие маленькие числа часто используются (счётчики, индексы и т.п.), и выгодно не создавать их заново. Деталь: границы кеша (-5 и 256) исторически выбраны; их можно поменять, пересобрав Python, но обычно нет смысла. Кеширование малых чисел – реализация детали CPython, в других реализациях Python может быть иначе, но знать про это полезно (например, поэтому `256 is 256` возвращает True, а `257 is 257` может вернуть False, т.к. это уже разные объекты).

Пример:

```
x = 500
y = 500
print(x is y)  # False, скорее всего (не кешируется 500)
a = 50
b = 50
print(a is b)  # True (малые числа кешируются)
```

Хотя следует помнить, что оператор `is` не предназначен для сравнения значений чисел – этот пример только иллюстрирует кэш.

В исходном коде CPython реализация big int находится в файлах `Objects/longobject.c` и `Include/cpython/longobject.h`. Там определён тип `PyLongObject` и множество функций: например, `PyLong_FromLong`, `PyLong_AsLong` и т.д. – для преобразования, и реализация операций. Если число помещается в фит в размер машинного слова, CPython всё равно обращается к нему через эти структуры. Это влияет разве что на скорость: например, сложение двух маленьких int – очень быстрая операция на C (прямо 64-битные инты складываются), но как только числа становятся больше 2^30, уже идёт работа с массивом "цифр". Однако для программиста Python всё прозрачно: можно складывать огромные числа, просто это будет потреблять больше процессорного времени.

Вещественные числа с плавающей запятой (float). Встроенный тип `float` – это *двойная точность 64-бит* по стандарту IEEE 754 (binary64). В CPython ему соответствует структура `PyFloatObject`, содержащая поле типа C `double` для значения [53] [54]. То есть Python `float` прямо хранит `double` из C. Все операции над float (сложение, умножение и т.д.) в CPython делегируются к соответствующим операциям над C `double`. Это значит, что `float` Python имеет ограниченную точность ~15 десятичных цифр и определённые особенности (ошибка округления, специальное значение `inf`, `nan`, наследуемые от IEEE 754). Для большинства целей это “обычные” вещественные числа двойной точности. Класс `float` в Python – неизменяемый, как и int.

Важно отметить: Python не дает неявно более высокой точности для float, он полагается на C `double`. Поэтому, например, 0.1 + 0.2 == 0.3 будет False (из-за двоичной природы представления). Если нужна произвольная точность для десятичных дробей, можно использовать `decimal.Decimal` из стандартной библиотеки. Но встроенный `float` – это фиксированный 64-бит.

В документации C-API указано: `PyFloatObject` представляет Python float [55], а объект `PyFloat_Type` – сам тип. Для создания float есть функции `PyFloat_FromDouble` и др. [56], а для

извлечения – `PyFloat_AsDouble` [57]. В общем, `float` Python очень прямолинеен относительно реализации – он отражает платформенный тип двойной точности.

Комплексные числа (complex). Python поддерживает комплексные числа как встроенный тип (литералы, например `3+4j`). Внутренняя реализация – структура `PyComplexObject`. Документация говорит: "Этот subtype PyObject представляет комплексное число" [58]. В CPython комплексное число хранит два `double` – действительную и мнимую части. В коде это сделано через C-структуру `Py_complex` (с маленькой буквы), которая содержит два поля `double real` и `double imag` [59]. А `PyComplexObject` включает в себя `PyObject_HEAD` и поле `cval` типа `Py_complex` (то есть два двойных). Все операции с комплексными в CPython (сложение, умножение, и т.д.) реализованы в `Objects/complexobject.c`, обычно просто выполняя формулы с `double`. Таким образом, Python `complex` – пара 64-битных float. Например:

```
z = 3.0 + 4.5j
print(z.real, z.imag)  # 3.0 4.5
```

Внутри `z` хранится `3.0` и `4.5` как `double`. Как и float, это ограниченная точность по 15 десятичных цифр на каждую часть.

В C-API есть функции: `PyComplex_FromDoubles(double real, double imag)` – создаёт новый комплекс [60]. `PyComplex_RealAsDouble(obj)` и `PyComplex_ImagAsDouble(obj)` возвращают соответствующие части [61] [62]. С точки зрения Python, комплексные – неизменяемые (immutable). Операции создают новые объекты. Например, `z1 + z2` создаст новый `PyComplexObject` с суммой.

Исходный код CPython и численные типы. Он распределён так: - `Include/longobject.h` (или часть в `Include/cpython/longobject.h`) – объявления для int. - `Objects/longobject.c` – реализация int. - `Include/floatobject.h` / `Objects/floatobject.c` – float. - `Include/complexobject.h` / `Objects/complexobject.c` – complex.

Можно посмотреть, например, в `floatobject.h`: определение `PyFloatObject` и константы (там же, например, есть `PyFloat_AsDouble` макроопределение) [55].

Выводы: - `int` в Python – это произвольной длины целое (арифметика целая без переполнения), но требующее больше памяти и времени для больших чисел. - `float` – 64-бит с плавающей запятой (IEEE 754 double). - `complex` – две 64-битных составляющих.

Это соответствует математической модели Python: целые неограничены, а вещественные и комплексные – ограничены двойной точностью, что надо учитывать (особенно ошибки округления).

Официальная дока C-API прямо говорит, что `PyLongObject` представляет целое, `PyFloatObject` – float, `PyComplexObject` – комплексное [51] [54] [63].

Для полноты: Python также имеет bool (подтип int, в CPython `PyBoolObject` – просто 0 или 1), и тип Decimal в библиотеке (не встроен, реализован на Python), а также Fractions (рациональные). Но в рамках встроенных численных типов – три основных класса: int, float, complex.

Обращаясь к исходникам, можно увидеть, например, в `object.h`:

```
PyAPI_DATA(PyTypeObject) PyLong_Type;    /* built-in 'int' */
PyAPI_DATA(PyTypeObject) PyFloat_Type;   /* built-in 'float' */
PyAPI_DATA(PyTypeObject) PyComplex_Type; /* built-in 'complex' */
```

И далее определения структур. Это подтверждает соответствие.

В заключение, знание устройства числовых типов может пригодиться при оптимизации (например, понимать, что цикл суммирования 100 млн. маленьких ints – ок, а 100 млн. очень больших ints – значительно медленнее), а также при решении задач высокой точности (когда Python автоматически переключается на длинную арифметику).

Вопросы и ответы

1. Области видимости переменных
Вопрос: Что такое LEGB и как Python ищет переменные по этой модели?
Ответ: LEGB – это акроним уровней областей видимости: Local, Enclosing, Global, Built-in. При обращении к имени внутри функции Python сначала смотрит в **локальной** области данной функции. Если не найдено – ищет во **внешних (охватывающих)** функциях (для вложенных функций, последовательно от внутренней к внешней). Затем – в **глобальной** области текущего модуля. И наконец – во **встроенных именах** интерпретатора (модуль `builtins`) [1] [2]. Например, имя встроенной функции `len` будет найдено только на этапе Built-in, если вы не переопределили `len` локально или глобально. Такая последовательность поиска гарантирует, что локальные переменные перекрывают глобальные с тем же именем и т.д.

Вопрос: Чем отличается *область видимости* от *пространства имён*?
Ответ: *Область видимости (scope)* – это контекст в коде, где определённые имена доступны без квалификатора. Например, внутри функции – её локальный scope. *Пространство имён (namespace)* – это конкретное хранение соответствия имя→объект (реализовано словарём). Связаны они так: каждая область видимости реализуется одним или несколькими пространствами имён. Например, глобальный scope – это namespace модуля (его `__dict__`), локальный scope функции – её локальный словарь (доступный через `locals()`), built-in scope – namespace модуля `builtins` [5] [7]. Scope определяет *правила поиска* имени (в каких namespaces смотреть и в каком порядке), а namespace просто содержит сами имена.

Вопрос: Для чего служат ключевые слова `global` и `nonlocal`?
Ответ: Они нужны, чтобы присваивать значения переменным из внешних областей. `global X` внутри функции означает, что все упоминания `X` в этой функции относятся к глобальной переменной модуля [10]. Без этого присваивание `X = ...` создало бы локальную `X`. `nonlocal Y` используется во вложенной функции и указывает, что `Y` – не локальная, а находится в охватывающей функции (не глобальной) [10]. Это позволяет модифицировать переменную, определённую во внешней функции, внутри внутренней. Без `nonlocal` попытка присвоить вызовет создание новой локальной переменной и потерю доступа к внешней.

Вопрос: Почему возникает ошибка `UnboundLocalError`, когда мы пытаемся изменить переменную из внешней области без объявления `nonlocal`?
Ответ: Потому что в момент компиляции функция видит присваивание имени и считает его локальным. Например:

```
x = 5
def f():
    print(x)  # пытаемся напечатать внешнюю x
    x = 3     # присваивание – Python решает, что x локальная
```

При компиляции `f` узнаёт о присваивании `x` и помечает `x` как локальную. А строка `print(x)` тогда ссылается на локальную `x`, которая ещё не была присвоена к тому моменту – отсюда ошибка о свободной переменной. Правильное решение: либо удалить присваивание (тогда `x` считается внешней и будет взята глобальная), либо явно объявить `global x` (если хотим менять глобальную), либо, в случае вложенных функций, `nonlocal` для изменения внешней не-глобальной переменной.

2. Замыкания
Вопрос: Что такое замыкание в Python?
Ответ: Замыкание – это функция, которая сохраняет ссылки на переменные из своей внешней (охватывающей) области видимости, даже когда та функция, где переменные определены, уже завершила выполнение. Т.е. внутренняя функция “замыкает” значения внешних переменных. В Python это реализуется, когда внутри функции определена другая функция и используется переменная из внешней функции, а затем внутренняя функция возвращается наружу. Она будет нести с собой “замкнутые” переменные.

Вопрос: Как узнать, какие переменные замкнуты внутри функции?
Ответ: У функции есть атрибут `__code__.co_freevars` – это кортеж имён свободных переменных (тех самых, которые замкнуты). А атрибут `__closure__` содержит кортеж объектов-ячееек, где хранятся значения этих переменных [22]. Например:

```
def outer():
    x = 10
    def inner():
        return x + 5
    return inner

fn = outer()
print(fn.__code__.co_freevars)      # ('x',) – имя замкнутой переменной
print(fn.__closure__[0].cell_contents)  # 10 – значение замкнутой переменной
x
```

Если переменных несколько, они по порядку соответствуют кортежу.

Вопрос: Почему после выхода из внешней функции переменные не удаляются, если на них есть замыкание?
Ответ: Потому что на них остаются активные ссылки – через те самые объекты `cell` во внутренней функции. С точки зрения сборщика мусора, переменная (точнее, объект) не может быть очищена, пока на неё есть хотя бы одна ссылка. В случае замыкания внутренняя функция хранит ссылки (через `__closure__`), поэтому объекты остаются живыми. Именно поэтому мы можем вызывать `inner` уже после завершения `outer`, и `inner` по-прежнему "помнит" значение `x`.

3. Импорт

Вопрос: Зачем нужен файл `__init__.py` в каталоге и что будет, если его нет?
Ответ: Файл `__init__.py` обозначает, что директория – это пакет. В Python до 3.3 без него пакет бы не распознавался (импорт из папки без `__init__.py` не работал). Сейчас возможны *namespace packages* без `__init__.py`, но классически его присутствие остаётся желательным. Если `__init__.py` отсутствует, вы всё равно можете импортировать вложенные модули, если указали полный путь, но сама папка может не создаваться как объект-пакет (в Python 3 это частично решено). В целом, лучше всегда иметь `__init__.py` (может быть пустой), чтобы чётко обозначить пакет.

Вопрос: Как работает переменная `__all__` и влияет ли она на обычные импорты?
Ответ: Переменная `__all__` – это список имён (строк), она определяет, что будет импортировано при `from module import *` [24]. Если `__all__` есть, то только указанные имена попадут в текущий namespace. Если её нет, то пакет при `import *` ничего не импортирует автоматически из подмодулей (а модуль импортирует все имена, не начинающиеся с `_`, из своего пространства) [26]. На **обычный импорт** (например, `import module` или `from module import name`) `__all__` никак не влияет – он используется *только* для звездочного импорта.

Вопрос: Что означает, если имя модуля или переменной начинается с подчёркивания? Импортируются ли такие объекты?
Ответ: Начало с `_` по соглашению означает “не предназначено для внешнего использования” (приватная часть). При выполнении `from module import *` такие имена **пропускаются** (не импортируются) [27]. Но их можно импортировать поимённо, например `from module import _hidden` – Python это не запретит. В самих модулях часто используют `_` для функций или переменных, которые являются служебными. Модуль `module` может сам переопределить это поведение через `__all__`, включив даже подчёркивания, но если он этого не сделал, `import *` фильтрует их. Одним словом: одинарное подчёркивание – сигнал “не трогать снаружи”, Python уважает это при `*`-импорте, а во всех других случаях – это лишь соглашение, разработчик сам должен избегать.

Вопрос: Как программно импортировать модуль, имя которого известно только в виде строки?
Ответ: Можно использовать функцию `importlib.import_module("name_of_module")` из библиотеки `importlib`. Например:

```
mod_name = "math"
mod = importlib.import_module(mod_name)
```

Теперь `mod` – это модуль `math`. Если нужно импортировать из пакета, можно указать полное имя (например `"mypack.mymod"`). Старый способ – вызвать `__import__("name")`, но это менее удобно (требуется управлять параметрами `globals`, `fromlist` и т.д.), и официально рекомендуется `importlib`.

Вопрос: Чем отличаются абсолютный и относительный импорт?
Ответ: Абсолютный импорт указывает модуль или пакет, начиная с корневого уровня. Например, `import package.module` – абсолютный путь. Относительный – использует точку (.) для текущего пакета, две точки (..) для родительского и т.д. Например, внутри пакета `mypack` в модуле `a.py` можно написать `from . import b` (импортировать модуль b из того же пакета) или `from .. import utils` (из родительского пакета). Относительные импорты работают

только внутри пакетов. Если попробовать их в скрипте не-пакете (или запустить модуль напрямую), получите `ImportError`. Поэтому относительный синтаксис применим, когда у вас структура пакетов, и вы хотите явно ссылаться на соседа или родителя, без указания полного имени.

4. Интерактивный интерпретатор и Google Colab
Вопрос: Можно ли внутри Google Colab (Jupyter) открыть настоящий интерактивный REPL, как в терминале, чтобы пошагово вводить команды вручную?
Ответ: По умолчанию Colab – это ноутбук, где вы запускаете ячейки. Обычного REPL, который бы останавливался и ждал ввода в том же окне, нет. Тем не менее, вы можете запустить shell или python-интерпретатор в output, но управлять им неудобно. Например, выполнив в ячейке `! python -i`, вы попытаетесь открыть интерактивный режим, но Colab не даст вам нормально ввести команды – он "повиснет", ожидая ввода, которого некуда вводить (ноутбук не предназначен для такого режима) [64]. Есть хитрый способ: запуск `!jupyter console --existing` мог бы подключиться к текущему kernel, но в Colab он не работает (блокируется по причине GIL) [64]. Поэтому практический ответ: **нет, Colab не предоставляет полноценного интерактивного prompt внутри себя** – он сам по сути работает, как REPL по ячейкам. Вы можете просто создавать новые ячейки и выполнять код, это и есть способ интерактивной работы. Если очень нужно, можно подключиться локально через `colab.connect_to_host` или использовать `%shell` (или `%%bash`) magic для выполнения команд shell, но полноценного python-взаимодействия как в терминале не получится.

Вопрос: Как считать ввод от пользователя в Colab (если, например, использовать функцию `input()`)?
Ответ: Несмотря на отсутствие консольного REPL, команда `input()` в Colab работает – при её вызове выводится приглашение "Введите..." и появляется текстовое поле, где пользователь может ввести строку. После ввода и нажатия Enter выполнение ячейки продолжится, и `input()` вернёт введённую строку. То есть Colab поддерживает ввод, просто не в виде терминала, а через специальные виджеты. Это применимо и к Jupyter Notebook в целом.

5. Модуль sys
Вопрос: Что содержит `sys.argv`?
Ответ: Это список строковых аргументов командной строки, переданных скрипту. `sys.argv[0]` – имя запускаемого скрипта (или путь к нему) [65]. Если интерпретатор запущен интерактивно или без файла, `sys.argv[0]` может быть пустой строкой. Элементы `sys.argv[1:]` – это те аргументы, которые вы указали после имени скрипта при запуске. Например, `python script.py foo bar` приведёт к `sys.argv == ["script.py", "foo", "bar"]`. В Google Colab и Jupyter `sys.argv` тоже определён, но там обычно не передаются аргументы (кроме технических).

Вопрос: Как получить путь к исполняемому файлу Python (интерпретатору)?
Ответ: В `sys` есть переменная `executable` – она содержит полный путь до исполняемого файла Python [66]. Например, `sys.executable` может вернуть `/usr/bin/python3.10`. Это полезно, например, чтобы вызвать субпроцесс с тем же интерпретатором. Если Python не смог определить путь, `sys.executable` может быть пустым или None, но в стандартных случаях он задан.

Вопрос: Что делает `sys.exit()`? Чем отличается вызов `sys.exit(0)` от `sys.exit(1)`?
Ответ: `sys.exit()` возбуждает исключение `SystemExit`, которое останавливает интерпретатор (если не перехвачено). Вы можете передать код возврата: `sys.exit(0)`

означает завершить программу с кодом 0 (успех), `sys.exit(1)` – с кодом 1 (общая ошибка) [67]. Можно передать и строку, например `sys.exit("Ошибка")` – тогда Python выведет эту строку на stderr и завершится с кодом 1 [68]. Внутри `exit()` просто бросает `SystemExit`; при работе в REPL (например, Jupyter) это исключение перехватывается окружением, чтобы не убить kernel, но в скрипте оно завершает программу.

Вопрос: Для чего используются `sys.getsizeof(obj)` и метод объекта `__sizeof__()`? Почему они могут давать разные результаты?
Ответ: Метод `obj.__sizeof__()` должен возвращать базовый размер объекта `obj` в байтах, без учета накладных расходов сборщика мусора [69]. А функция `sys.getsizeof(obj)` возвращает размер, **включая** дополнительный overhead. В CPython `getsizeof` вызывает `obj.__sizeof__()`, и если объект участвует в garbage collector, добавляет несколько байт (например, размер поля GC head) [70]. Поэтому часто `sys.getsizeof` чуть больше. Например, для пустого списка `[].__sizeof__()` может дать 40 (байт), а `sys.getsizeof([])` – 56, т.к. добавлены накладные ~16 байт на GC [69]. Важно понимать, что эти функции не показывают *полный* размер структуры данных, если она содержит ссылки на другие объекты. Например, `sys.getsizeof([1,2,3])` вернёт размер самого списка (управляющей структуры), но не включит размер элементов. Для глубокого расчёта нужно рекурсивно обходить. Также `sys.getsizeof` не работает в PyPy, где нет прямого доступа к размеру.

Вопрос: Почему `sys.getrefcount(obj)` почти всегда возвращает число на 1 больше, чем ожидается?
Ответ: Функция `getrefcount` возвращает **текущее количество ссылок** на объект, но когда вы передаёте объект в эту функцию, внутри создаётся **временная ссылка** на аргумент (на время вызова) [71]. Из-за этой дополнительной ссылки результат на единицу больше. Например:

```
import sys; a = []
print(sys.getrefcount(a))  # выведет, скажем, 2 (ожидали 1)
```

Одна ссылка – это переменная `a` сама, вторая – временная на время передачи аргумента в `getrefcount`. После вызова эта ссылка освобождается. Поэтому такое поведение нормально. Кроме того, у некоторых объектов (например, небольшие целые, interned строки) могут быть глобальные ссылки, поэтому их refcount может быть неочевидно большим.

6. Модули и importlib
Вопрос: В чём разница между модулем `builtins`, модулем `__builtin__` и переменной `__builtins__`?
Ответ: В Python 3 основным модулем, содержащим встроенные объекты, является `builtins` (с буквой *s*). В Python 2 он назывался `__builtin__` (без s). В Python 3 имя `__builtin__` остаётся как алиас для совместимости, но обычно не используется. Переменная `__builtins__` (с s) автоматически присутствует в глобальном словаре любого модуля – она ссылается на словарь встроенных имен (обычно на модуль `builtins` или его `__dict__`). Если вы выполните `globals().get("__builtins__")`, то получите либо модуль builtins, либо его dict (это детали реализации, но обычно модуль). Эта переменная нужна интерпретатору, чтобы при поиске built-in имен в данном модуле обращаться к ней. В общем, программисту напрямую редко нужно лезть в `__builtins__`. Иногда в sandbox окружениях его урезают. **Итого**: используйте модуль `builtins` для доступа к встроенным (например, `builtins.open`), `__builtins__` – служебная переменная, а `__builtin__` актуальна только для старых версий Python.

Вопрос: Как перезагрузить модуль во время выполнения, если он уже импортирован?
Ответ: Использовать функцию `importlib.reload(module)`. Перед этим модуль должен быть импортирован (у вас должен быть объект модуля). Например:

```
import importlib, mymodule
# ... изменили файл mymodule.py ...
importlib.reload(mymodule)
```

Это заново выполнит код модуля `mymodule` и обновит его содержимое. Обратите внимание: объекты, полученные *до* перезагрузки (например, из `from mymodule import X`) не изменятся автоматически. Нужно либо снова их импортировать, либо обращаться как `mymodule.X`. Также, если при первой загрузке модуль установил какие-то объекты в другие модули, `reload` их не зачистит. Иногда безопаснее полностью удалить модуль из `sys.modules` и заново `import`, но это может вызвать проблемы, если на модуль были ссылки. `importlib.reload` – штатный и рекомендованный способ.

Вопрос: Что делает функция `importlib.import_module` и чем отличается от обычного `import`?
Ответ: `import_module(name)` динамически импортирует модуль по имени, переданному строкой. Она возвращает объект модуля. Например, `import_module("math")` эквивалентно `import math` (но возвращает объект). Отличий в конечном результате нет – оба используют системный механизм импорта (внутри `import_module` всё равно вызывает `__import__`). Разница в том, что `import_module` – функция, её можно вызывать в runtime с переменной. Обычный `import` – это статический синтаксис. Также `import_module` легко импортирует модуль внутри пакета, если дать полное имя. Например, `importlib.import_module('os.path')` вернёт подмодуль `os.path`. Обычным import пришлось бы писать `from os import path`.

Вопрос: При работе в Jupyter я написал код, создал файл с помощью `%%writefile`, но модификации файла не отражаются при повторном импорте. Почему и как исправить?
Ответ: Когда вы первый раз импортируете модуль, Python загружает его и сохраняет в `sys.modules`. Повторный `import` того же модуля ничего не делает (возвращает кэш). Поэтому, если вы изменили исходник файла, нужно либо перезапустить kernel, либо использовать `importlib.reload`. В Jupyter (и Colab) есть `%load_ext autoreload` опция, но по умолчанию она не активна. Итак, если вы перезаписали файл `mymod.py` и хотите обновить модуль, вызовите `importlib.reload(mymod)`. При этом убедитесь, что `mymod` уже был импортирован (reload требует объект). Если ещё не импортировали, сделайте обычный import.

Вопрос: Как узнать версию установленного пакета через код?
Ответ: Первый вариант – проверить атрибут `__version__` у модуля (если он есть). Многие пакеты его имеют. Например:

```
import numpy
print(numpy.__version__)
```

Второй, более общий вариант – использовать `importlib.metadata`:

```
from importlib import metadata
print(metadata.version('numpy'))
```

Этот способ читает метаданные установленного пакета, поэтому работает даже если пакет не импортирован (и даже если имя модуля != имя пакета в pip). В случае, если пакет не найден, выбросится исключение. Также можно получить больше инфо через `metadata.metadata('package')`. Если вы работаете, например, в Conda, можно вызвать `!pip show package` в Jupyter, но программно лучше через importlib.metadata.

Вопрос: Что содержит специальная переменная `__name__` внутри модуля?
Ответ: Она содержит имя модуля. Если модуль импортирован, это его полное имя (например, `'mypack.mymodule'`). Если файл запущен как скрипт, `__name__` установится в `"__main__"` [16]. Именно поэтому проверка `if __name__ == "__main__":` позволяет понять, выполняется ли модуль напрямую. Для пакетов (файла `__init__.py`) `__name__` равно имени пакета. В интерактивной консоли `__name__` пространства выполненных команд – тоже `"__main__"`.

7. Структура проекта
Вопрос: Что происходит при импорте пакета? Выполняется ли код в `__init__.py`?
Ответ: Да. Когда вы делаете `import package`, Python создаёт новый модуль с именем `package` и выполняет код из `package/__init__.py`. Только после этого импорт считается успешным. Все объекты, которые `__init__.py` определил (или импортировал) становятся атрибутами пакета. Например, если в `__init__.py` написано `x = 5`, то после `import package` вы можете делать `package.x` (равно 5). Если `__init__.py` пустой, то пакет будет просто “пустым” контейнером без атрибутов (но подмодули всё равно можно импортировать по отдельности).

Вопрос: Как, находясь в модуле пакета, импортировать соседний модуль?
Ответ: Либо абсолютным импортом: `from package import sibling_module`, либо относительным: `from . import sibling_module`. Второй вариант короче и привязывает к текущему пакету, но работает только если модуль действительно импортируется как часть пакета. Если вы запустите модуль напрямую, относительный импорт не сработает (пакетный контекст не определён).

Вопрос: Для чего используется переменная `__annotations__` в модуле?
Ответ: В неё Python сохраняет аннотации типов для глобальных переменных. Если в модуле написать `size: int = 0`, то в `__annotations__` будет запись `{"size": <class 'int'>}`. Это полезно для инструментов, проводящих анализ типов. Также модуль `typing` и сторонние библиотеки могут использовать эти аннотации. В остальном на выполнение программы они не влияют.

Вопрос: Как узнать путь к файлу модуля?
Ответ: У объекта модуля есть атрибут `__file__`, содержащий путь к файлу, из которого модуль загружен [36]. Например:

```
import os
print(os.__file__)
```

выведет путь к файлу `os.py` (или к `os/__init__.py`, т.к. `os` – пакет). Если у модуля нет файла (например, встроенный модуль), атрибут может отсутствовать. Пакеты имеют `__file__` (путь к `__init__.py`) и `__path__` (список путей, где искать подмодули).

8. Внутреннее устройство Python

Вопрос: Правда ли, что в Python "всё является объектом"? Что собой представляют классы и типы с точки зрения реализации?

Ответ: Да, в Python всё – объекты, включая функции, классы, типы и даже модули. Реализовано это так: все объекты имеют структуру `PyObject` с указателем на свой тип [38]. У обычных данных (чисел, списков) тип – это встроенные классы (`int`, `list` и т.п.), а у классов (то есть у объектов типа `type`) тип – это *метакласс*, обычно `type`. Встроенный класс `type` является экземпляром себя самого [45] (метакласс самого себя). Базовый класс `object` – предок всех классов – является экземпляром `type` [47]. Внутри CPython есть глобальные структуры `PyTypeObject` для всех базовых типов. Например, `PyLong_Type` соответствует `<class 'int'>`, `PyList_Type` – `<class 'list'>`. Есть `PyType_Type` – это `<class 'type'>` и `PyBaseObject_Type` – `<class 'object'>` [43]. Связи: `PyType_Type` имеет в `tp_base` ссылку на `PyBaseObject_Type` (т.е. `type` наследует `object`) [47], `PyBaseObject_Type` имеет `ob_type` = `PyType_Type` (т.е. `object` – экземпляр `type`) [47], и `PyType_Type.ob_type` указывает на себя [45]. Благодаря этому класс `type` замыкает объектную систему.

Вопрос: Что содержится в структуре PyObject, и почему у каждого объекта Python есть "счётчик ссылок"?

Ответ: В `PyObject` два основных поля: счетчик ссылок (тип `Py_ssize_t`, хранит число) и указатель на тип (структуру `PyTypeObject*`) [38]. Счетчик ссылок – механизм управления памятью: каждый раз, когда объект присваивается новой переменной или включается в контейнер, refcount увеличивается, а когда удаляется ссылка – уменьшается. Когда refcount становится 0, объект уничтожается. CPython использует подсчёт ссылок, дополняя его периодическим сбором циклических ссылок. Поэтому у каждого объекта есть этот счетчик. Разработчик Python обычно его не видит напрямую (кроме случаев с `sys.getrefcount`), но понимание помогает: например, ясно, почему создание циклов (объект ссылается на себя через контейнер) требует отдельного сборщика – refcount там не обнулится автоматически.

Вопрос: Зачем нужен PyTypeObject и что хранится в нём?

Ответ: `PyTypeObject` – структура C, описывающая Python-класс (тип). В ней хранится множество полей: имя типа, размер в памяти экземпляров, указатели на функции – методы, операции, конструкторы, деструктор, флаги. Например, есть поле `tp_new` – функция C для выделения новой структуры объекта, `tp_init` – инициализация (если используется), таблицы числовых операций (`tp_as_number`), последовательностей (`tp_as_sequence`), отображений (`tp_as_mapping`), и т.д. Поля `tp_base` и `tp_bases` хранят информацию о наследовании. Когда вы создаёте в Python класс, CPython под капотом заполняет `PyTypeObject`. Для встроенных типов эти структуры заданы статически. Итог: `PyTypeObject` – "всё о типе": как создать объект, как уничтожить, как напечатать (`tp_repr`), какие у него методы (`tp_methods` – массив структур PyMethodDef) и т.д. [39]. Обычному программисту не нужно знать все поля, но если пишете расширение на C, придётся заполнить такую структуру, чтобы определить новый тип.

Вопрос: Что представляют собой `PyType_Type` и `PyBaseObject_Type`?

Ответ: Это глобальные экземпляры `PyTypeObject`. `PyType_Type` – это структура типа для объекта `type` (метакласс). То есть `PyType_Type` описывает, как устроены все объекты-типы. В Python он соответствует классу `type`. `PyBaseObject_Type` – структура типа для базового

`object`. То есть `PyBaseObject_Type` описывает, как устроен базовый класс, от которого наследуют все остальные. В Python он соответствует классу `object`. В исходниках они объявлены примерно так:

```
PyTypeObject PyType_Type; /* built-in 'type' */
PyTypeObject PyBaseObject_Type; /* built-in 'object' */
```

И при инициализации интерпретатора эти структуры связываются друг с другом (как мы обсуждали выше).

9. Численные типы

Вопрос: Почему целые числа в Python не переполняются? Как они хранятся?
Ответ: Потому что Python `int` – это “большое целое” произвольной длины. Вместо фиксированного разряда, числа занимают столько памяти, сколько нужно для его представления. Внутренне число хранится как массив “цифр” в основании 2^{30} (или 2^{15} на некоторых системах) [50], с признаком знака. При операциях, если результат требует больше памяти, Python автоматически расширяет массив. Поэтому выражение вроде `2**1000` выдаёт огромное число точно. Цена – операции с очень большими числами работают медленнее, чем с маленькими, и занимают больше памяти. Но с точки зрения программиста – целое “неограниченной точности”.

Вопрос: Что такое кеширование малых целых чисел в Python?
Ответ: CPython ради оптимизации хранит заранее объекты int для чисел от -5 до 256 [52]. При создании int в этом диапазоне возвращается уже существующий объект. Это ускоряет работу с этими часто используемыми числами и экономит память (например, все единицы – это один и тот же объект 1). Поэтому поведение оператора `is` может быть неожиданным для этих чисел: `a=256; b=256; a is b` вернёт True (оба указывают на один объект из кеша), а с 257 может быть False (создались разные объекты). Но следует помнить, что это внутренняя оптимизация: на корректность программы не влияет, а сравнивать числа надо через `==`.

Вопрос: Как хранятся числа с плавающей запятой (float) в Python?
Ответ: Точный ответ: как 64-битные `double` в формате IEEE 754 [54]. Python `float` напрямую соответствует типу `double` языка C на данной платформе. Это ~15-16 значащих десятичных цифр, порядка 10^{-308} до 10^{308} диапазон, имеются специальные значения nan, inf. Поэтому `0.1 + 0.2` не совсем равно `0.3` из-за двоичного представления. `float` – неизменяемый тип; его операции (сложение, умножение и т.п.) выполняются на уровне процессора с плавающей точкой.

Вопрос: А как реализованы комплексные числа?
Ответ: `complex` состоит из двух `double` – вещественной и мнимой частей [59]. Внутри CPython определена структура `Py_complex` (с полями `real` и `imag` типа double) и структура `PyComplexObject`, которая содержит `PyObject_HEAD` и `Py_complex cval`. Операции над комплексными (сложение, умножение) – просто комбинация операций над двумя double по формулам. Например, сложение $(a+bi)+(c+di) = (a+c) + (b+d)i$ – Python делает две операции сложения double. То есть точность комплексного – та же, что у double (порядка 15 значащих цифр на каждую часть).

Вопрос: Где в исходниках Python можно посмотреть реализацию целых, вещественных, комплексных?
Ответ: В репозитории CPython: - `Objects/longobject.c` – реализация int (PyLongObject), -

`Objects/floatobject.c` – реализация float (PyFloatObject), - `Objects/complexobject.c` – реализация complex (PyComplexObject). Также соответствующие заголовочные файлы в `Include/`. Например, в `Include/longobject.h` описано, что `PyLongObject` – произвольной длины целое, а в комментарии указано про кеш -5..256 и т.д. [52]. Там же макросы для работы с “digit”. Хотя проще прочитать документацию: она говорит: “All integers are implemented as long integer objects of arbitrary size” [50], `PyFloatObject` – “this subtype of PyObject represents a Python floating-point number (C double)” [54], `PyComplexObject` – аналогично для комплексных [63]. Если посмотреть `floatobject.c`, можно найти реализацию операций, например функцию `float_add` и т.д.
Для более высокого уровня: документация Python (не C API) тоже упоминает, что int – произвольной длины (убрали понятие long vs int, всё int), float – IEEE 754 double, complex – две double.

---

[1] [2] [3] [4] [5] [6] Python Scope & the LEGB Rule: Resolving Names in Your Code – Real Python
https://realpython.com/python-scope-legb-rule/

[7] [8] What is the relationship between scope and namespaces in Python? - Software Engineering Stack Exchange
https://softwareengineering.stackexchange.com/questions/273302/what-is-the-relationship-between-scope-and-namespaces-in-python

[9] [10] 7. Simple statements — Python 3.13.3 documentation
https://docs.python.org/3/reference/simple_stmts.html

[11] [12] [13] [14] [15] [28] [29] Built-in Functions — Python 3.13.3 documentation
https://docs.python.org/3/library/functions.html

[16] [17] __main__ — Top-level code environment — Python 3.13.3 documentation
https://docs.python.org/3/library/__main__.html

[18] The Walrus Operator: Python's Assignment Expressions – Real Python
https://realpython.com/python-walrus-operator/

[19] PEP 572 – Assignment Expressions | peps.python.org
https://peps.python.org/pep-0572/

[20] Walrus operator variable in list-comprehension leaks into global namespace - Python Help - Discussions on Python.org
https://discuss.python.org/t/walrus-operator-variable-in-list-comprehension-leaks-into-global-namespace/5969

[21] python - What is co_names? - Stack Overflow
https://stackoverflow.com/questions/45147260/what-is-co-names

[22] python - What exactly is contained within a obj.__closure__? - Stack Overflow
https://stackoverflow.com/questions/14413946/what-exactly-is-contained-within-a-obj-closure

[23] A Practical Guide to Python Closures By Examples - Python Tutorial
https://www.pythontutorial.net/advanced-python/python-closures/

[24] [25] [26] 6. Modules — Python 3.13.3 documentation
https://docs.python.org/3/tutorial/modules.html

[27] Why are python imports renamed with leading underscores - Stack Overflow
https://stackoverflow.com/questions/47138594/why-are-python-imports-renamed-with-leading-underscores

30 31 32 33 34 35 65 66 67 68 71 sys — System-specific parameters and functions — Python 3.13.3 documentation
https://docs.python.org/3/library/sys.html

36 __file__ (A Special variable) in Python | GeeksforGeeks
https://www.geeksforgeeks.org/__file__-a-special-variable-in-python/

37 Is module __file__ attribute absolute or relative? - Stack Overflow
https://stackoverflow.com/questions/7116889/is-module-file-attribute-absolute-or-relative

38 49 Common Object Structures — Python 3.13.3 documentation
https://docs.python.org/3/c-api/structures.html

39 48 Type Object Structures — Python 3.13.3 documentation
https://docs.python.org/3/c-api/typeobj.html

40 50 51 52 Integer Objects — Python 3.13.3 documentation
https://docs.python.org/3/c-api/long.html

41 43 44 cpython/Include/object.h at main - GitHub
https://github.com/python/cpython/blob/main/Include/object.h

42 /usr/include/python3.9/object.h
https://llvm.org/reports/scan-build/report-object.h-_Py_IS_TYPE-1-d75c7d.html

45 46 47 Python's objects and classes — a visual guide — Reuven Lerner
https://lerner.co.il/2015/10/18/pythons-objects-and-classes-a-visual-guide/

53 55 Floating Point Objects — Python 3.9.21 documentation
https://docs.python.org/3.9/c-api/float.html

54 56 57 Floating-Point Objects — Python 3.13.3 documentation
https://docs.python.org/3/c-api/float.html

58 59 60 61 62 63 Complex Number Objects — Python 3.13.3 documentation
https://docs.python.org/3/c-api/complex.html

64 How to open an interactive Python prompt in Google Colab? - Stack Overflow
https://stackoverflow.com/questions/58104847/how-to-open-an-interactive-python-prompt-in-google-colab

69 python - difference between __sizeof__ and sys.getsizeof() - Stack Overflow
https://stackoverflow.com/questions/55427689/difference-between-sizeof-and-sys-getsizeof

70 Difference between __sizeof__() and getsizeof() method – Python | GeeksforGeeks
https://www.geeksforgeeks.org/difference-between-__sizeof__-and-getsizeof-method-python/