ChatGPT

Углублённые концепции Python (уровень senior)

Ниже представлен подробный обзор продвинутых тем Python, актуальных для уровня Senior. Каждая тема раскрыта с пояснениями, примерами кода и ссылками на официальную документацию и авторитетные источники.

Области видимости и правило LEGB

В Python разрешение имён переменных подчиняется **правилу LEGB** — порядок поиска имени: Local → Enclosed → Global → Built-in [1]. При обращении к имени Python последовательно ищет его в: - **Локальной области (Local)** текущей функции или блока. - **Внешней области (Enclosing)** ближайших вложенных функций. - **Глобальной области (Global)** модуля (верхний уровень скрипта или пакета). - **Встроенной области (Built-in)**: встроенные имена (`len`, `print` и т.д.).

Например, если в функции нет локального определения имени, то Python проверит внешний уровень и т.д. Это иллюстрирует следующий пример:

```
x = 100                # глобальная область

def outer():
    x = 10             # область enclosed (замыкание)

    def inner():
        print(x)       # 'x' из внешней enclosing области (выведет 10)
    inner()

outer()
print(x)               # 'x' из глобальной области (выведет 100)
```

Использование `global` и `nonlocal` позволяет управлять областями видимости вручную [2] [3]. Ключевое слово `global X` внутри функции указывает, что имя `X` относится к глобальной области (модульной). Без `global` попытка присвоения создаст локальную переменную. Например:

```
count = 0           # глобальная переменная

def func():
    global count
    count += 1      # изменяет глобальную count
```

Ключевое слово `nonlocal Y` используется во вложенных функциях, чтобы ссылаться на переменную `Y` из внешней (но не глобальной) области [3]. Пример:

```
def make_counter():
    total = 0
    def inc():
        nonlocal total
        total += 1
        return total
    return inc

counter = make_counter()
print(counter())    # 1
print(counter())    # 2
```

В этом примере `total` хранится в области enclosing функции `make_counter`. Без `nonlocal`, `total += 1` в `inc()` создавал бы новую локальную `total`, что привело бы к ошибке.

Scope vs Namespace. Понятие *области видимости (scope)* — это контекст доступа к именам (например, область функции или модуля). *Пространство имен (namespace)* — это хеш-таблица (словарь), где имена ассоциируются с объектами. В Python пространства имен создаются для модулей, функций, классов и т.д. (например, `globals()`, `locals()`) [4]. Таким образом, область видимости определяет, какие пространства имен видимы, а пространства имен хранят сами привязки.

Функции `globals()` и `locals()` позволяют получить словари текущих глобальных и локальных имен соответственно. Например:

```
a = 10
def f(b):
    print("globals:", list(globals().keys()))
    print("locals:", list(locals().keys()))
    return a + b

f(5)
```

`dir()` без аргументов выводит список имен текущего пространства имен (модуля или интерактивного сеанса), а с объектом показывает его атрибуты. Например, `dir()` внутри функции покажет локальные имена, а `dir(module)` — атрибуты модуля.

`if __name__ == "__main__"`

Конструкция

```
if __name__ == "__main__":
    # код
```

используется для различения режима запуска скрипта. Когда файл выполняется как главный скрипт, его встроенная переменная `__name__` равна `"__main__"`. Если модуль импортирован,

`__name__` будет именем модуля. Таким образом, этот блок позволяет выполнять код только при прямом запуске файла, но не при импорте его в качестве модуля.

Компиляция Python-кода (py → pyc, байткод)

При импорте модуля Python автоматически компилирует код в байткод и сохраняет его в файле с расширением `.pyc` в каталоге `__pycache__`. Этот процесс облегчает последующую загрузку модуля. Компиляция включает лексический разбор, генерацию AST и преобразование в байткод. Байткод — это набор инструкций для виртуальной машины CPython. Модуль `dis` позволяет посмотреть байткод:

```
import dis
def f(x): return x*x
print(dis.dis(f))
```

Выполнение байткода производится интерпретатором CPython. При изменении исходного `.py` файла или при явной компиляции (например, `python -m compileall .`), автоматически обновляются файлы `.pyc`.

Сборка байткода и сохранение `.pyc` выполняется так, что при следующем импорте, если файл `.py` не изменялся (или маркер времени совпадает), Python загрузит готовый `.pyc` для ускорения. Подробнее см. официальную документацию и PEP [2].

Морж-оператор (`:=`) и `func.__code__.co_varnames`

Морж-оператор (`:=`)

Оператор присваивания выражения `:=` (так называемый **морж-оператор**) введён в Python 3.8. Он позволяет выполнить присваивание в составе выражения. Например:

```
if (n := len(my_list)) > 0:
    print(f"Список не пуст, длина {n}")
```

Здесь `n := len(my_list)` вычисляет длину списка и одновременно сохраняет её в переменную `n`. Такой приём сокращает код, когда нужно использовать результат функции сразу и проверить его. Другой пример — чтение вводимых данных:

```
while (line := input(">>> ")) != "exit":
    print("Введено:", line)
```

Здесь `line := input()` читает строку и сохраняет в `line`, после чего сравнение с `"exit"` выполняется в условии `while`. Морж-оператор помогает избегать повторного вызова функций и улучшает читаемость в некоторых ситуациях.

`func.__code__.co_varnames`

Каждая функция в Python имеет объект кода `__code__` с различными атрибутами, описывающими её параметры и локальные переменные. Атрибут `func.__code__.co_varnames` — кортеж имён всех локальных переменных функции (включая параметры) в том порядке, как они хранятся. Например:

```
def add(a, b):
    total = a + b
    return total

print(add.__code__.co_varnames)  # ('a', 'b', 'total')
```

Это может быть полезно для анализа функций на уровне байткода или динамического кода. Также существуют `co_argcount` (число позиционных аргументов), `co_consts` (константы функции) и другие поля объекта `__code__`.

Замыкания (Closures)

Замыканием (closure) называют вложенную функцию, которая “захватывает” переменные из внешней области. То есть внутренняя функция запоминает состояние внешней. Замыкание происходит, когда:

- В функции определена вложенная функция.
- Внутренняя функция ссылается на переменную из внешней функции, и при этом внешняя функция уже завершила работу.

Например:

```
def make_multiplier(factor):
    def multiply(x):
        return x * factor
    return multiply

double = make_multiplier(2)
print(double(10))  # 20
```

Здесь `multiply` — вложенная функция, использующая `factor` из объёма `make_multiplier`. После завершения `make_multiplier`, значение `factor` сохраняется в замыкании, и функция `double` помнит, что `factor = 2`.

Free variables (свободные переменные) — это имена, используемые внутри функции, но не определённые в её локальной области (и не во встроенных), то есть они “свободно” лежат в области enclosing. В примере выше переменная `factor` является свободной в `multiply`.

В CPython информация о замыкании хранится в атрибуте `__closure__` функции: это кортеж объектов `cell`, содержащих значения захваченных переменных. Имена этих переменных находятся в `func.__code__.co_freevars`. Например:

```
def make_adder(n):
    def adder(x):
        return x + n
    return adder

f = make_adder(5)
print(f.__code__.co_freevars)       # ('n',)
print(f.__closure__[0].cell_contents)  # 5
```

Здесь `f.__code__.co_freevars` показывает, что `n` — свободная переменная, а `f.__closure__[0].cell_contents` содержит её текущее значение (5).

Где находятся замыкания: Структура `PyFunctionObject` в CPython хранит кортеж `func_closure`, в котором лежат `PyCellObject` для каждой закрытой переменной. Фактически замыкание — это доступ к переменным из области исполнения вызова функции-замыкателя. Это используется, например, для сохранения состояния между вызовами (как в примере с накоплением среднего или суммой).

Полный пример:

```
def outer():
    text = "Hello"
    def inner():
        print(text)
    return inner

fn = outer()
fn()  # Выведет "Hello" — внутренней функции доступен text из enclosing scope
```

Чтобы вручную “достать” замыкание, можно прочитать `func.__code__.co_freevars` для имён и `func.__closure__` для значений. Если функция не захватывает ничего, то `__closure__` равно `None`.

Замыкания полезны для создания фабрик функций и сохранения состояния без использования глобальных переменных [5]. Однако нужно учитывать, что замыкание сохраняет **ссылку** на переменную, а не копию. Это влияет на поведение, если внешняя переменная изменяется после создания функции.

Пример, почему обсуждают замыкания: они позволяют инкапсулировать данные в функции, но могут неожиданно “держать в памяти” объекты и влиять на сборку мусора.

Импорт и структура модулей

`__init__.py`, `__all__`, публичные и приватные сущности

Каталог с файлом `__init__.py` считается пакетом Python. При импорте пакета выполняется код в `__init__.py`, который может задавать, какие подмодули и атрибуты будут доступны.

Параметр `__all__` внутри модуля или пакета управляет тем, что импортируется при `from package import *`. Если `__all__` определён, то `*` импортирует только имена из списка. Например:

```
# package/__init__.py
__all__ = ['mod1', 'func2']
from .mod1 import foo
from .mod2 import bar as func2
```

Публичными считаются имена без ведущего подчёркивания, приватные — начинающиеся с `_` (одиночное или двойное). По конвенции при `from module import *` будут импортированы только публичные имена (или указанные в `__all__`) [6]. Имена с одним подчёркиванием считаются неофициально приватными и обычно не импортируются таким способом.

Динамический импорт: `__import__`, `importlib`

Функция `__import__('module_name')` позволяет импортировать модуль по строке. Часто более удобно использовать `importlib.import_module()`:

```
import importlib
mod = importlib.import_module('os.path')
```

`importlib.reload(module)` перезагружает уже импортированный модуль, что полезно при отладке, когда нужно обновить определения функций без перезапуска интерпретатора.

Абсолютные и относительные импорты

По умолчанию импорты являются абсолютными: `import package.module`. При использовании относительных импортов в пакетах используют нотацию `.`:

```
# изнутри пакета
from . import sibling_module    # текущий пакет
from ..subpackage import submod  # родительский пакет
```

Важно понимать `sys.path` — список директорий, где Python ищет модули. Обычно это текущая директория, каталоги стандартной библиотеки и `site-packages` для установленных пакетов [7]. Подключенные библиотеки лежат в каталогах, указанных в `sys.path`, чаще всего это `.../pythonX.Y/site-packages/`.

`sys.modules` и кэширование

При импорте Python сохраняет загруженные модули в словаре `sys.modules`. Это кэш, чтобы при повторном импорте тот же объект не загружался заново. Можно просмотреть содержимое `sys.modules`:

```
import sys, math
print(sys.modules.get('math'))   # <module 'math' (built-in)>
```

Если нужно принудительно заново выполнить код модуля, используют `importlib.reload`:

```
import mymodule, importlib
# Изменили mymodule.py...
importlib.reload(mymodule)  # загрузит обновлённый код
```

Местоположение и структура подключённых библиотек

Узнать, откуда загружен модуль, можно через атрибут `module.__file__` (для модулей в файлах) или `module.__path__` (для пакетов). Пример:

```
import os
print(os.__file__)      # путь к файлу os.py или .so
```

Это позволяет убедиться, что используется нужная версия библиотеки. Некоторые пакеты могут иметь несколько уровней (каталог с **init**.py, подпакеты и файлы).

Интерактивный интерпретатор и Google Colab

Интерактивный Python обычно означает запуск интерпретатора без немедленного выполнения скрипта (режим REPL). В Google Colab интерфейс уже представляет собой интерактивную среду: каждая ячейка — это интерактивный блок кода на Python. Чтобы запустить интерактивный сеанс внутри Colab, можно:

- Просто ввести и выполнить Python-код в ячейках ноутбука (Colab по умолчанию предоставляет Python-интерпретатор).
- Для доступа к системной оболочке можно использовать `!python` или `%%bash`. Например, выполнить в ячейке:

  ```
  !python - <<'PYCODE'
  print("Интерактивный режим Python")
  x = 5
  print(x*x)
  PYCODE
  ```

  Это запустит код в отдельном сеансе Python.
- Также можно ввести `!python` в ячейке, чтобы попасть в REPL (Ctrl+C — выход). Например:

  ```
  !python
  ```

  После чего в появившемся выводе можно писать команды вручную. Однако переменные этого режима не будут сохраняться в Python-ядре ноутбука.

На практике в Colab обычно работают прямо в ячейках, которые и так интерактивны. Специальных magics (`%python`) нет по умолчанию (как `%bash`), но можно воспользоваться `%%bash` или `!python`. Также у Colab есть меню *"Инструменты → Открыть терминал"*, где можно запустить систему с `python`.

Модуль `sys`

Модуль `sys` предоставляет информацию и функции, взаимодействующие с интерпретатором [8].

- `sys.argv` — список аргументов командной строки; `argv[0]` — имя скрипта (либо `'-c'`, если передавали код через `-c`). Полезно для парсинга аргументов. Пример: `import sys; print(sys.argv)`.
- `sys.exit([code])` — завершает выполнение интерпретатора, выбрасывая исключение `SystemExit`. Обычно `sys.exit()` с ненулевым кодом означает ошибочный выход. Это аналог *exit code* процесса.
- `sys.path` — список строк путей, где Python ищет модули при импорте [7]. Сюда входят текущая директория, пути из `PYTHONPATH`, а также каталоги стандартной библиотеки и `site-packages`.
- `sys.version` — строка с версией интерпретатора и информацией о сборке [9]. `sys.version_info` — кортеж (namedtuple) версий: `(major, minor, micro, ...)`.
- `sys.platform` — строка с именем платформы (например, `'linux'`, `'darwin'`, `'win32'`) [10]. Позволяет определить ОС в коде.
- `sys.getsizeof(obj)` возвращает размер объекта в байтах **с учётом** накладных расходов GC [11]. `obj.__sizeof__()` возвращает *базовый* размер без GC-накладных. Так, `sys.getsizeof()` вызывает `obj.__sizeof__()` и добавляет дополнительные байты для сбора мусора [11]. Пример разницы:

```
import sys
lst = [1, 2, 3]
print(lst.__sizeof__(), sys.getsizeof(lst))
```

Вывод может быть, например, `56 80` (где 56 — базовый объект, 80 — с учётом GC). - `sys.getrefcount(obj)` — возвращает счётчик ссылок на объект **+1**, так как вызов функции сам добавляет временную ссылку [12]. Например:

```
import sys
a = []
print(sys.getrefcount(a))  # например, 2
```

Здесь 2 означает одну нашу ссылку `a` и одну временную внутри `getrefcount`. Заметьте, что у «вечнозелёных» (singleton) объектов (например, `None`, `True`, малые целые) могут быть очень большие счётчики, не отражающие реальное число ссылок [12]. - Другие полезные атрибуты `sys`: `sys.maxsize` (максимальный размер int-подобного), `sys.platform` (платформа), `sys.version_info` (версия), `sys.executable` (путь до интерпретатора), `sys.modules` (список загруженных модулей) и др.

Пример использования `sys`:

```
import sys
print("Python", sys.version_info)
print("OS:", sys.platform)
print("Script args:", sys.argv)
```

Эти данные помогают адаптировать поведение программы в зависимости от окружения.

Работа с модулями и атрибутами

- `builtins` — модуль, содержащий все стандартные встроенные имена (например, `len`, `list`, `Exception` и т.д.) [6]. Обычно Python автоматически обеспечивает доступ ко встроенным через `__builtins__`. Атрибут `__builtins__` (с буквой *s*) присутствует в глобальном пространстве любого модуля. Это либо сам модуль `builtins`, либо его словарь атрибутов [6]. Пользователям редко требуется напрямую обращаться к `builtins`, но его можно импортировать:

```
import builtins
print(builtins.len([1,2,3]))  # 3
```

- `__builtin__` (без s) — название встроенного модуля в Python 2; в Python 3 используется `builtins`. `__builtins__` (с s) — внутренняя ссылка на этот модуль (смотрите примечание в [56†L38-L46] и [60†L19-L28]). Для переносимости кода лучше использовать `import builtins` (или `import __builtin__` в Py2).

- **Версия модуля.** Модули могут содержать атрибут `__version__`, указывающий версию (но не все). Например, `numpy.__version__`. Чтобы узнать версию установленного пакета, можно также использовать `importlib.metadata` (или `pkg_resources`):

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

Однако это зависит от наличия метаданных в установленных пакетах. Обратить внимание: `module.__version__` — не стандартный атрибут, но часто встречается в API библиотек.

- `__name__` — имя модуля. Для основного скрипта `__name__ == '__main__'`. Для импортированных модулей `__name__` — имя модуля (без `.py`). Это удобно, чтобы определить контекст работы.

- `__doc__` — строка документации модуля (может быть пустой). Можно посмотреть с помощью `print(module.__doc__)` или функции `help(module)`.

- `__annotations__` — словарь аннотаций (например, типовых подсказок) на уровне модуля. Обычно пуст, если нет глобальных аннотаций.

- Некоторые служебные атрибуты модуля: `module.__file__` (путь к файлу), `module.__path__` (для пакетов), `module.__package__` (имя пакета) и т.д.

Структура проекта и пакетов

При организации более сложных проектов обычно создают пакеты (каталоги с `__init__.py`). Структура может быть такой:

```
project/
  setup.py
  src/
    mypackage/
      __init__.py
      module1.py
      module2.py
      subpackage/
        __init__.py
        submod.py
```

`__init__.py` определяет, какие подмодули видимы при импорте. Часто он импортирует публичные сущности или просто оставляется пустым. Через `__all__` внутри `__init__.py` или модулей можно контролировать `from ... import *`.

При запуске кода можно узнать, откуда загружается модуль:

```
import mypackage
print(mypackage.__file__)   # путь к __init__.py пакета
```

В пакете `__init__.py` можно определить `__all__`, чтобы скрыть внутренние модули:

```
# mypackage/__init__.py
__all__ = ['module1']
from . import module1
```

Тогда `from mypackage import *` импортирует только `module1`. При этом переменные без `_` обычно считаются публичными.

Если нужно узнать расположение любых модулей во время выполнения, можно посмотреть `sys.path` или использовать:

```
import importlib.util, os
spec = importlib.util.find_spec('numpy')
print(os.path.dirname(spec.origin))  # каталог установки numpy
```

Низкоуровневая структура объектов CPython

В CPython все объекты имеют тип `PyObject*` в реализации. Структура `PyObject` (через макрос `PyObject_HEAD`) содержит вначале два поля: счётчик ссылок и указатель на

PyTypeObject (тип объекта) [13]. То есть любая переменная в Python указывает на некий `PyObject*`, у которого есть поля `ob_refcnt` и `ob_type`. Удобные макросы: `Py_REFCNT(obj)` и `Py_TYPE(obj)` позволяют получить эти поля [13].

`PyTypeObject` — структура, описывающая тип объекта (его методы, операции, размеры и т.д.) [14]. Когда мы определяем класс на Python, создаётся соответствующий `PyTypeObject` (тип-объект). `PyTypeObject` содержит, например, `tp_name` (имя типа), `tp_basicsize` (базовый размер) и указатели на функции-реализации операций (доступ к атрибутам, арифметика, конструктор и т.п.) [15] [16]. Практически каждый встроенный тип описывается в CPython своим PyTypeObject (например, `PyLong_Type` для `int`, `PyList_Type` для `list`).

`PyType_Type` — метакласс, то есть тип для всех типов. Проще говоря, каждый объект типа является инстанцией *типа*, а сам тип (класс) — это объект типа `PyType_Type`. Аналог в Python: `type` — это метакласс. Например, `type(int) is type` соответствует тому, что тип `int` сам имеет тип `type`.

`PyBaseObject_Type` — базовый тип всех объектов (как класс `object` в Python) [17]. Все типы наследуют по цепочке от него. В документации указано: “`PyBaseObject_Type` — базовый класс всех объектов, то же, что `object` в Python” [17].

Соотношения: - У любого объекта `o` на C-поверхности `Py_TYPE(o)` указывает на его тип (PyTypeObject*). - У типа (PyTypeObject) поле `ob_type` (или эквивалент) обычно указывает на `PyType_Type`. То есть `Py_TYPE(PyLong_Type) == &PyType_Type`. - `PyType_Type` сам является объектом типа `PyType_Type` (самоворождающийся тип), и его поле `tp_base` обычно `&PyBaseObject_Type` (т.е. наследует от `object`).

Структуру PyObject и PyTypeObject можно представить схематически:

```
typedef struct {
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
} PyObject;

typedef struct _typeobject {
    PyVarObject ob_base;    // включает PyObject + ob_size для хранения
    const char *tp_name;
    Py_ssize_t tp_basicsize, tp_itemsize;
    destructor tp_dealloc;
    // ... множество полей (методы, операции и т.д.) ...
    struct _typeobject *tp_base;
    // ...
} PyTypeObject;
```

Это упрощённое представление. Все объекты, включая типы, имеют базовую часть `PyObject_HEAD`. Подробная документация CPython описывает `PyTypeObject` и связанные макросы [13] [14].

Численные типы Python

Python предоставляет несколько числовых типов:

- `int` **(целые)**: в CPython реализованы как произвольной длины. Структура `PyLongObject` хранит целое число в виде массива «цифр» в базе (обычно 2^30) [18]. Это позволяет иметь очень большие числа. При этом с повышением разряда (число растёт) добавляются новые цифры (и память). Важная оптимизация CPython: **кэш малых целых** от -5 до 256 [19]. Это значит, что эти числа создаются один раз при запуске интерпретатора, и при последующем использовании повторно возвращается указатель на уже существующий объект. Это экономит время и память. Например:

```
a = 100
b = 100
print(id(a) == id(b))  # True, т.к. 100 в кэше
c = 1000
d = 1000
print(id(c) == id(d))  # False (скорее всего), т.к. 1000 не кэшируется
```

Документация CPython прямо говорит: «актуальная реализация хранит массив объектов для целых от -5 до 256; при создании int в этом диапазоне вы фактически получаете существующий объект» [19]. Подробнее о внутреннем виде: `PyLongObject` содержит `ob_size` (число цифр) и `ob_digit[i]` — сами цифры в base 2^30 [18].

- `bool` **(булев)**: в Python это подкласс `int`. Существует ровно два объекта `True` и `False`, которые являются экземплярами типа `bool`. На C-уровне `PyBool_Type` — это отдельный тип, но объекты `Py_True` и `Py_False` являются «вечными» (не уничтожаются) [20]. Операции с `True`/`False` ведут себя как 1/0. Например, `True + True == 2`. Тип `bool` в документации CPython описан как «подкласс целого, единственные два объекта `Py_True` и `Py_False`» [20].

- `float` **(с плавающей запятой)**: реализован как объект `PyFloatObject`, содержащий двойную точность (`double`). То есть стандарт IEEE 754 double. Этого достаточно примерно для 15–17 десятичных знаков. В отличие от int, не предполагается большое точное хранение. Примеры:

```
f = 3.14
print(type(f))  # <class 'float'>
print(f.as_integer_ratio())  # (157, 50)
```

Документация CPython указывает, что `PyFloatObject` представляет собой число двойной точности и `PyFloat_Type` соответствует Python-типу `float` [21].

- `complex` **(комплексные)**: состоит из двух чисел с плавающей запятой (два поля `double` — вещественная и мнимая части) [22] [23]. На C-уровне есть структура `PyComplexObject`, содержащая макрос `PyObject_HEAD` и поле `Py_complex cval` (структура с двумя `double`) [22] [23]. Функции для работы позволяют создавать комплексные:

`PyComplex_FromDoubles(real, imag)`. В Python: `z = 1.0 + 2.0j` даёт комплексное число.

Во всех перечисленных случаях новые объекты создаются специальными C-функциями (например, `PyLong_FromLong()`, `PyFloat_FromDouble()` и т.д.). В CPython часть операций оптимизирована (например, кэш целых, фриватные функции `_Py_c_sum` для комплексных [24]).

Таким образом, численные типы устроены «под капотом» на C разными структурами, а на уровне Python видно поведение: неограниченные целые (`int`), бинарные флаги (`bool`), двойная точность (`float`), комплексные. Дополнительную информацию можно найти в исходниках CPython (файлы *longobject.c*, *floatobject.c*, *complexobject.h*), а документация упоминает основные детали (кэш малых int [19], наследование bool от int [20], представление float и complex).

Заключение

В этом обзоре рассмотрены ключевые аспекты продвинутого Python: области видимости и модификаторы `global`/`nonlocal`, механизм замыканий и свободных переменных, тонкости импорта и структуры пакетов, особенности работы интерактивных сред (включая Google Colab), модуль `sys` и его возможности, метаполезные особенности модулей (`builtins`, атрибуты) и низкоуровневая организация объектов в CPython, а также устройство численных типов. Каждый пример снабжён пояснениями, а факты подкреплены авторитетными источниками [4] [19] [20]. Это должно помочь глубоко разобраться в интересующих темах и уверенно ответить на вопросы уровня Senior.

---

[1] [4] Scope Resolution in Python | LEGB Rule | GeeksforGeeks
https://www.geeksforgeeks.org/scope-resolution-in-python-legb-rule/

[2] [3] 7. Simple statements — Python 3.13.3 documentation
https://docs.python.org/3/reference/simple_stmts.html

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

[6] builtins — Built-in objects — Python 3.13.3 documentation
https://docs.python.org/3/library/builtins.html

[7] [8] [9] [10] [11] [12] sys — System-specific parameters and functions — Python 3.13.3 documentation
https://docs.python.org/3/library/sys.html

[13] [17] Common Object Structures — Python 3.13.3 documentation
https://docs.python.org/3/c-api/structures.html

[14] [15] [16] Type Object Structures — Python 3.13.3 documentation
https://docs.python.org/3/c-api/typeobj.html

[18] Python Caches Integers | Codementor
https://www.codementor.io/@arpitbhayani/python-caches-integers-16jih595jk

[19] Integer Objects — Python 3.13.3 documentation
https://docs.python.org/3/c-api/long.html

[20] Boolean Objects — Python 3.13.3 documentation
https://docs.python.org/3/c-api/bool.html

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

22 23 Include/complexobject.h - external/github.com/python/cpython - Git at Google
https://chromium.googlesource.com/external/github.com/python/cpython/+/refs/tags/v3.9.16/Include/complexobject.h

24 Complex Number Objects — Python 3.13.3 documentation
https://docs.python.org/3/c-api/complex.html