100 вопросов для подготовки к собесу Python

Средний 75 мин 21K

Python*Регулярные выражения*ООП*Профессиональная литература*Интервью

Из песочницы

Доброго времени суток!

Представляю подборку из 100 вопросов с собесов на позицию джуна Python-разработчика. На Хабре есть неплохие статьи на тему подготовки к собеседованию и всё в таком духе, но прямо набора вопросов/ответов на понимание Python в формате чек-листа не встречал.

Для кого статья?

- для тех, кто думает, что знает Python как 5 пальцев — проверьте себя)
- кому предстоят собесы, и хочется ликвидировать слепые зоны
- кто начинает изучать Python — статья покажет, что в Python хватает сюрпризов

Вопросы есть ~~глупые~~ простые/сложные, теоретические/практические, в общем, это микс, призванный помочь вам лучше понимать свой рабочий инструмент.

В моем канале вы найдете много задач с собеседований и акутальных инструментов и гайдов для Python разработчиков.

Телеграм канал для тех, кто хочет изучить машинное обучение на Python - Нейронные сети, машинное обучение, python

Ну и плюс я тут запилил ролик с разбором этих вопросов, вот — https://youtu.be/zKkuWq0vIPE

Что ж, поехали!

Содержание:

32. Что такое контекстный менеджер?
33. Как сделать python-скрипт исполняемым в различных операционных системах?
34. Как сделать копию объекта? Как сделать глубокую копию объекта?
35. Опишите принцип работы сборщика мусора в python.
36. Как использовать глобальные переменные? Это хорошая идея?
37. Для чего в классе используется атрибут slots?
38. Какие пространства имен существуют в python?
39. Как реализуется управление памятью в python?
40. Что такое метаклассы и в каких случаях их следует использовать?
41. Зачем нужен pdb?
42. Каким будет результат следующего выражения?
43. Как создать класс без слова class?
44. Как перезагрузить импортированный модуль?
45. Напишите декоратор, который будет перехватывать ошибки и повторять функцию максимум N раз.
46. Каким будет результат следующего выражения?
47. Какие проблемы есть в python?
48. Когда будет выполнена ветка else в конструкции try…except…else?
49. Поддерживает ли python множественное наследование?
50. Как dict и set реализованы внутри? Какова сложность получения элемента? Сколько памяти потребляет каждая структура?
51. Что такое MRO? Как это работает?
52. Как аргументы передаются в функции: по значению или по ссылке?
53. С помощью каких инструментов можно выполнить статический анализ кода?
54. Что будет напечатано в результате выполнения следующего кода?
55. Что такое GIL? Почему GIL всё ещё существует?
56. Опишите процесс компиляции в python
57. Что такое дескрипторы? Есть ли разница между дескриптором и декоратором?
58. Почему всякий раз, когда python завершает работу, не освобождается вся память?
59. Что будет напечатано в результате выполнения следующего кода?
60. Что такое интернирование строк? Почему это есть в python?
61. Как упаковать бинарные зависимости?
62. Почему в python нет оптимизации хвостовой рекурсии? Как это реализовать?
63. Что такое wheels и eggs? В чём разница?
64. Как получить доступ к модулю, написанному на python из C и наоборот?
65. Как ускорить существующий код python?
66. Что такое pycache? Что такое файлы .pyc?
67. Что такое виртуальное окружение?
68. Python — это императивный или декларативный язык?
69. Что такое менеджер пакетов? Какие менеджеры пакетов вы знаете?
70. В чём преимущества массивов numpy по сравнению с (вложенными) списками python?
71. Вам нужно реализовать функцию, которая должна использовать статическую переменную. Вы не можете писать код вне функции и у вас нет информации о внешних переменных (вне вашей функции). Как это сделать?
72. Что будет напечатано в результате выполнения следующего кода?
73. Как имплементировать словарь с нуля?
74. Напишите однострочник, который будет подсчитывать количество заглавных букв в файле.
75. Что такое файлы .pth?
76. Какие функции из collections и itertools вы используете?
77. Что делает флаг PYTHONOPTIMIZE?
78. Какие переменные среды, влияющие на поведение интерпретатора python, вы знаете?
79. Что такое Cython? Что такое IronPython? Что такое PyPy? Почему они до сих пор существуют и зачем?
80. Как перевернуть генератор?
81. Приведите пример использования filter и reduce над итерируемым объектом.
82. Чем фреймворк отличается от библиотеки?
83. Расположите функции в порядке эффективности, объясните выбор.
84. Произошла утечка памяти в рабочем приложении. Как бы вы начали отладку?
85. В каких ситуациях возникает исключение NotImplementedError?
86. Что не так с этим кодом? Зачем это нужно?
87. Что такое магические методы (dunder-методы)?
88. Что такое monkey patching? Приведите пример использования.
89. Как работать с транзитивными зависимостями?
90. Когда использование Python является «правильным выбором» для проекта?
91. Что такое метод?
92. Есть ли в Python оператор switch-case?
93. Поддерживает ли Python регулярные выражения?
94. Напишите регулярное выражение, которое будет принимать идентификатор электронной почты. Используйте модуль re.

95. Как передать необязательные или ключевые параметры из одной функции в другую?
96. Как создать свой собственный пакет в Python?
97. Что такое функции высшего порядка?
98. Назовите модули в Python, связанные с файлами
99. В чем разница между NumPy и SciPy?
100. Что такое аксессоры, мутаторы, @property

1. Какие типы данных есть в python? На какие классы делятся?

Очень простой вопрос, но с чего-то же надо начать. В Python есть такие типы данных (*вообще, это классы в ООП, но ладно*):

- Числа: `int`, `float`, и `complex`.
- Строки: `str`.
- Списки: `list`.
- Кортежи: `tuple`.
- Словари: `dict`.
- Множества: `set`.
- Булевы значения: `bool`

Эти типы данных можно объединить в такие группы:

- Числовые типы данных: `int`, `float`, и `complex`.
- Строковые типы данных: `str`.
- Коллекции: `list`, `tuple`, `dict`, и `set`.
- Булевы типы данных: `bool`.

2. Что такое лямбда-функция? Какое у неё назначение?

Лямбда-функция (a.k.a *"анонимная функция"*) - это функция, которая определяется в одной строке кода без использования ключевого слова `def`. Она может быть использована вместо обычной функции, когда требуется быстрое определение небольшой функции. Как правило лямбда-функции одноразовые.

В Python лямбда-функция определяется с помощью ключевого слова `lambda`, за которым следует список аргументов через запятую, затем символ `:`, и наконец, тело функции.

Например, чтобы определить лямбда-функцию, которая удваивает свой аргумент, можно написать:

```
double = lambda x: x * 2
```

Лямбда-функции в основном используются в качестве аргументов функций высшего порядка, которые принимают другие функции в качестве аргументов. Также они могут использоваться для создания более читаемого и компактного кода.

Например, можно использовать лямбда-функцию вместо объявления обычной функции для преобразования списка:

```
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x**2, numbers))
```

Этот пример создает список квадратов чисел в списке `numbers` с помощью функции `map()`, принимающей лямбда-функцию в качестве аргумента.

Таким образом, лямбда-функция в Python позволяет определять небольшие функции быстро и использовать их в качестве аргументов для других функций. Вот ещё пару лямбд:

```
# применяем lambda к каждому элементу списка
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x**2, numbers))
print(squares)
```

```
# при помощи lambda фильтруем по чётности
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

```
# передаём lambda в качестве аргумента key для сортировки
mylist = ['111', '22', '3']
mylist = sorted(mylist, key=lambda x: len(x))
print(mylist)
```

3. Что такое docstring?

Docstring в Python - это строка документации, которая описывает, что делает функция, метод, модуль или класс Python. Данная строка располагается в начале определения объекта и используется для генерации документации автоматически. В других словах, docstring используется для создания описания API и содержит информацию о том, как использовать функцию или метод, какие аргументы они принимают и какие значения возвращают.

Например:

```
def add_numbers(a, b):
    """
    This function takes in two numbers and returns their sum
    """
    return a + b
```

В данном примере, docstring - это строка между тройными кавычками, после имени функции. Она описывает, что делает функция и как ее использовать.

Docstring является важным инструментом в Python разработке, так как важно документировать ваш код для себя и для других разработчиков. Документированный код легче поддерживать и понимать, что облегчает разработку и сотрудничество, честно.

```
def add_numbers(a, b):
    """
    This function takes in two numbers and returns their sum
    """
    return a + b

print(add_numbers.__doc__)
help(add_numbers)
```

4. Как получить документацию по атрибутам объекта?

В Python вы можете получить документацию по атрибутам объекта с помощью атрибута `doc`. Например, если у вас есть объект с атрибутом `attribute_name`, то вы можете получить его документацию так: `print(attribute_name.__doc__)`

Вы также можете использовать встроенную функцию `help()` для получения подробной информации о любом объекте, включая его атрибуты. Просто передайте объект в функцию `help()`, чтобы получить всю доступную документацию:
`help(attribute_name)`

Небольшое уточнение: `doc` отображает документацию для конкретного атрибута или метода. Если вы хотите получить общую документацию для объекта, вызовите `help()` без параметров (т.е. `help(object_name)`).
Например, если у вас есть класс с атрибутом attribute_name, вы можете получить его документацию следующим образом:

```
class MyClass:
    """This is the docstring for MyClass."""
    attribute_name = "value"

print(MyClass.attribute_name.__doc__)
```

Этот код выведет документацию для атрибута `attribute_name`, которая будет равна `None`, так как мы не определили документацию для него в классе. Теперь мы можем использовать функцию `help()` для получения документации для самого класса:
`help(MyClass)`
Это приведет к выводу всей доступной документации для `MyClass`, включая документацию для его атрибута `attribute_name`.

5. В чём разница между типами list и tuple?

В Python, список (`list`) и кортеж (`tuple`) являются двумя различными типами данных, которые предоставляют набор элементов в определенном порядке.

Основная разница между ними заключается в том, что список может быть изменен (мутабельный тип), а кортеж является неизменяемым (иммутабельным типом).

То есть, после создания кортежа, вы не можете изменять его содержимое, добавлять или удалять элементы. Это делает кортежи более эффективными по памяти и дает гарантию того, что их содержимое не будет изменено случайно в коде.

В то время как список может быть изменен, что было бы очень полезно, если вам нужна коллекция
элементов, которые вы можете изменять по ходу выполнения кода.

Различие в изменяемости во многом определяет отличие между списком и кортежем в скорости доступа к элементам. За счет того, что кортежи являются неизменным типом данных, они обрабатываются быстрее, чем списки.

Ну и да, для создания списка в Python используется квадратная скобка, а для создания кортежа используется круглая скобка. Вот примеры использования списков и кортежей:

```
my_list = [1, 2, 3, 4, 5]  # Это список
my_tuple = (1, 2, 3, 4, 5)  # Это кортеж
```

Список может быть изменен, например, можно добавить элемент в список:

```
my_list.append(6)
```

Но не можем добавить элемент в кортеж, так как он неизменяем:

```
my_tuple.append(6) # Эта строка вызовет ошибку
```

```
# в Jupyter Notebook можно замерить время так, будет видно замедление list
%time a = list(range(10**7))

%time b = tuple(range(10**7))
```

Кстати, забавный факт со списками — их можно делать бесконечно вложенными. Есть идеи, что возвращает код ниже?

```
a = [1, 2, 3, 4]
a.append(a)
print(a)
```

6. Может ли быть индекс списка отрицательным?

Да, индекс списка может быть отрицательным. В таком случае, отрицательное значение считается от конца списка, где `-1` соответствует последнему элементу, `-2` — предпоследнему элементу и так далее.

Например, чтобы получить последний элемент списка `my_list` в Python, можно использовать следующую команду:

```
last_element = my_list[-1]
```

Также можно использовать отрицательные значения для срезов (`slicing`) списка, например:

```
my_list[-3:] # вернет последние три элемента списка
my_list[:-2] # вернет все элементы списка, кроме последних двух
my_list[::-1] # вернет список в обратном порядке
```

Но следует учесть, что если индекс отрицательный и его абсолютное значение больше или равно длине списка, будет возбуждено исключение `IndexError` .

7. Что значит конструкция pass?

В Python, `pass` является пустым оператором. Он используется там, где синтаксически требуется оператор, но никаких действий выполнять не нужно.

Например, это может быть полезно при написании заглушки функции, которая будет реализована позже, или в цикле, который ничего не должен делать на данной итерации.

Пример использования конструкции `pass` :

```
def my_function():
    pass # заглушка для функции, которая будет реализована позже

for i in range(10):
    if i < 3:
        pass # ничего не делать на первых трёх итерациях
    else:
        print(i) # вывести значения на всех остальных итерациях
```

В обоих случаях pass играет роль пустого оператора, который не выполняет никаких действий, но позволяет синтаксически корректно описать код. Вообще, pass может использоваться в массе случаев.

К примеру, вот валидный код:

```
class MyClass:
    pass

for i in range(10):
    pass

def main():
    pass
```

```
myclass = MyClass()
myclass.name = 'Bob'  # мы можем даже создавать атрибуты экземпляра класса

main()
```

Кстати, в Python вместо `pass` можно писать и `...` (вот интересная статья на Хабр: Объект многоточие в Python)

8. Чем отличаются многопоточное и многопроцессорное приложение?

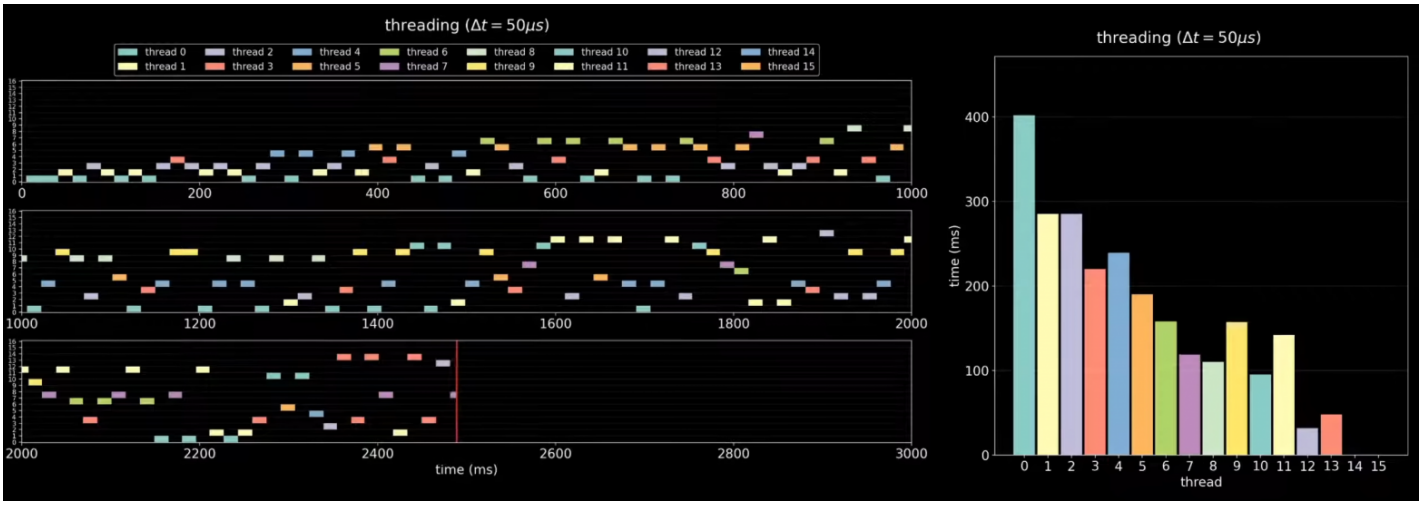
Многопоточное и многопроцессорное приложения отличаются друг от друга в том, как они используют ресурсы компьютера.

В многопроцессорных приложениях каждый процесс имеет свой собственный набор ресурсов, включая память, открытые файлы, сетевые соединения и другие системные ресурсы.
Многопроцессорность в Python может быть достигнута с помощью библиотек `multiprocessing` и `concurrent.futures` .

В многопоточных приложениях несколько потоков выполняются в рамках одного процесса, используя общие ресурсы. Это означает, что все потоки имеют доступ к общим данным.
Реализация многопоточности в Python выполняется за счет стандартной библиотеки `threading` .

При правильном использовании оба подхода могут ускорить выполнение программы и улучшить управляемость ею, однако многопоточное приложение может иметь проблемы с блокировками и условиями гонки при доступе к общим ресурсам. В многопроцессорных приложениях каждый процесс защищен от других процессов и обеспечивает более высокую степень изоляции.

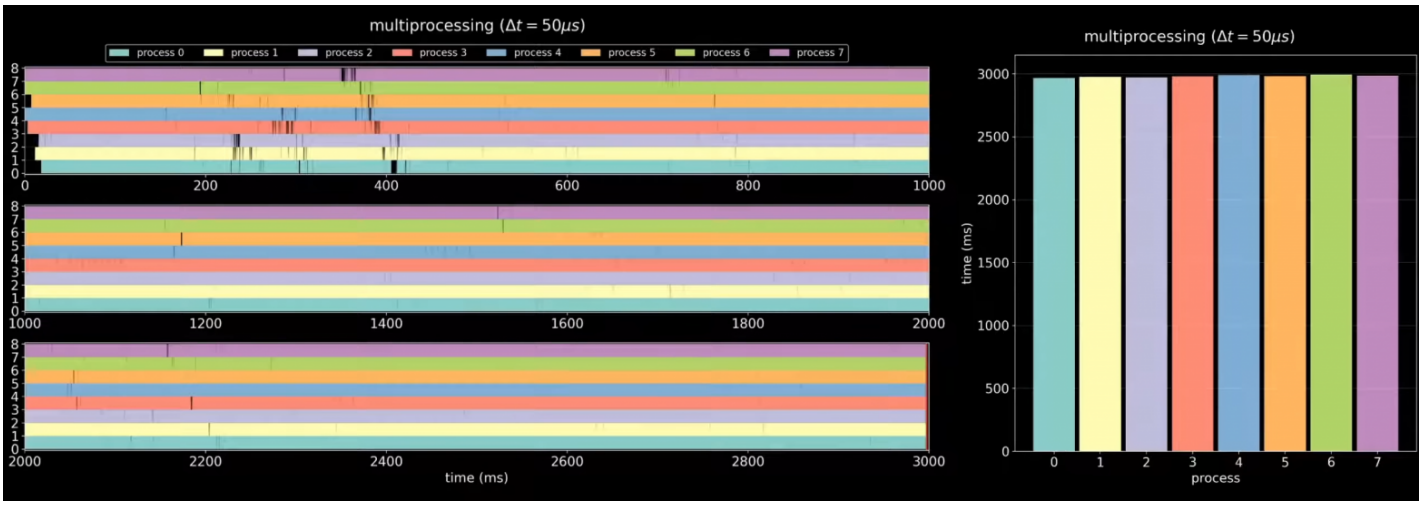
Так выглядит многопоточность, потоки конкурируют за доступ к ресурсам, памяти и др. в рамках одного процесса



img

Отличное видео с объяснением

А так выглядит многопроцессорность



img

Отличное видео, объясняющее многопроцессорность

9. Как просмотреть методы объекта?

Чтобы посмотреть все методы и атрибуты, связанные с определенным объектом в Python, можно использовать функцию `dir()`. Она принимает объект в виде аргумента и возвращает список имен всех атрибутов и методов объекта.

Например, если нужно увидеть все методы и атрибуты, связанные с объектом `my_list`, следующее:

```
my_list = [1, 2, 3]
print(dir(my_list))
```

Это выведет список всех методов и атрибутов, которые можно использовать с объектом `my_list`.

10. Что такое *args и **kwargs в определении функции?

`*args` и `**kwargs` - это специальные параметры в Python, которые позволяют передавать переменное количество аргументов в функцию.

Параметр `*args` используется для передачи переменного количества аргументов без ключевого слова. Он представляет собой кортеж из всех дополнительных аргументов, переданных функции.

Параметр `**kwargs` используется для передачи переменного количества именованных аргументов. Он представляет собой словарь из всех дополнительных именованных аргументов, переданных функции.

Символ `*` и `**` могут использоваться в определении функций для указания переменного числа аргументов, которые могут быть переданы в функцию. Символ `*` перед именем параметра означает, что все позиционные аргументы, которые не были использованы при определении других параметров, будут собраны в кортеж, который можно будет использовать внутри функции. Такой параметр называется `*args`.

Например:

```
def my_fun(a, b, *args):
    print(a, b, args)

my_fun(1, 2, 3, 4, 5)
# 1 2 (3, 4, 5)
```

Символ `**` перед именем параметра означает, что все именованные аргументы, которые не были использованы при определении других параметров, будут собраны в
словарь, который можно будет использовать внутри функции. Такой параметр называется `**kwargs`.

Например:

```
def my_fun(a, b, **kwargs):
    print(a, b, kwargs)

my_fun(1, 2, x=3, y=4, z=5)
# 1 2 {'x': 3, 'y': 4, 'z': 5}
```

Краткий итог: использование `*args` и `**kwargs` позволяет создавать более гибкие функции, которые могут принимать любое количество аргументов. А именно: `*args` собирает все "лишние" аргументы в кортеж, а `**kwargs` собирает в словарь именованные аргументы.

11. Python полностью поддерживает ООП?

Да, Python является полностью объектно-ориентированной языком. Он поддерживает все основные принципы ООП: наследование, инкапсуляцию и полиморфизм.

В Python все объекты в явном виде являются экземплярами классов, и даже типы данных, такие как список или словарь, являются классами со своими методами и атрибутами.
Кроме того, Python поддерживает множественное наследование, который позволяет создавать новые классы, которые наследуют методы и атрибуты от нескольких родительских классов одновременно.

В целом, Python предоставляет множество инструментов для написания кода в объектно-ориентированном стиле, и это один из главных его преимуществ, особенно для написания крупных и сложных приложений.

12. Что такое globals() и locals()?

`globals()` и `locals()` - это встроенные функции в Python, которые возвращают словари глобальных и локальных переменных соответственно.

`globals()` возвращает словарь, содержащий все глобальные переменные, доступные в текущей области видимости, включая встроенные переменные.
`locals()` возвращает словарь, содержащий все локальные переменные, определенные в текущей области видимости. Это включает аргументы функции и переменные,
которым присвоено значение внутри функции.

Например, вот как можно использовать эти функции:

```
x = 5
y = 10

def my_func(z):
    a = 3
    print(globals()) # выводит все глобальные переменные
    print(locals()) # выводит все локальные переменные

my_func(7)
```

В этом примере функция `my_func()` принимает один аргумент и определяет две локальные переменные (`a` и `z`). Когда она вызывается, она выводит на экран словари
глобальных и локальных переменных.

13. Что хранится в атрибуте dict объекта?

Атрибут `dict` содержит словарь, который хранит атрибуты объекта в виде пар ключ-значение. Этот словарь заполняется значениями при создании объекта и может быть
изменен позже. Например, если у вас есть объект класса `Person` , и вы создаете его экземпляр `person1` , то вы можете добавить новый

атрибут `age` и присвоить ему значение `25` следующим образом:

```
class Person:
    def __init__(self, name):
        self.name = name

    def say_hello(self):
        print("Hello, my name is", self.name)

person1 = Person("Alice")
person1.age = 25
print(person1.__dict__)
```

Это выведет словарь, содержащий пару ключ-значение `{'name': 'Alice', 'age': 25}`.
Вы можете обратиться к любому атрибуту объекта, используя либо обычную запись `person1.name`, либо запись, использующую словарь python `person1.__dict__["name"]`.

```
class Person:
    def __init__(self, name):
        self.name = name

    def say_hello(self):
        print("Hello, my name is", self.name)
```

```
person1 = Person("Alice")
person1.age = 25
print(person1.__dict__)
class MyClass:
    pass

myclass = MyClass()
myclass.name = 'Steve'
myclass.age = 12
myclass.lastname = 'King'

print(myclass.__dict__)
```

14. Как проверить файл .py на синтаксические ошибки, не запуская его?

Утилита `py_compile`, позволит проверить файл `.py` на наличие синтаксических ошибок без его запуска.

Используется `py_compile` очевидно:

1. открываем командную строку/терминал.
2. переходим в каталог, содержащий файл `.py`, который вы хотите проверить
3. выполняем: `python -m py_compile yourfile.py` где `yourfile.py` - это имя файла, который вы хотите проверить.

Эта команда выполнит проверку файла и выведет описание любых синтаксических ошибок, которые были найдены, или пустой вывод, если ошибок нет.

Когда это может быть полезно? Например, если код большой и в процессе задействует много ресурсов, а нужно всего лишь удостовериться в его валидности.

15. Зачем в python используется ключевое слово self?

В Python ключевое слово `self` используется для обращения к текущему объекту класса. Оно передается как первый аргумент в методы класса и позволяет работать с атрибутами и методами объекта класса внутри этих методов.

К примеру, рассмотрим класс `Person`, который имеет атрибут `name` и метод `say_hello`:

```
class Person:
    def __init__(self, name):
        self.name = name

    def say_hello(self):
        print(f"Hello, my name is {self.name}")
```

Здесь мы можем обратиться к атрибуту `name` объекта класса `Person` с помощью ключевого слова `self`. Аналогично, мы можем вызвать метод `say_hello`, который также использует `self` для доступа к атрибуту `name`:

```
person = Person("Alice")
person.say_hello() # выведет "Hello, my name is Alice"
```

Подводя итог, `self` позволяет нам работать с конкретным экземпляром класса (именно с `"Alice"` или `"Bob"`), с атрибутами и методами этого экземпляра, не трогая другие.

16. Что такое декоратор? Как написать собственный?

Декоратор в Python - это функция, которая принимает другую функцию в качестве аргумента и расширяет ее функциональность без изменения ее кода. Декораторы могут использоваться для добавления логирования, проверки аутентификации, тайминга выполнения и ещё кучи полезных штук.

Вот пример создания декоратора:

```
def my_decorator(func):
    def wrapper():
        print("Дополнительный код, который исполняется перед вызовом функции")
        func()
        print("Дополнительный код, который исполняется после вызова функции")
    return wrapper

@my_decorator
def say_hello():
    print("Привет!")

say_hello()

# Дополнительный код, который исполняется перед вызовом функции
# Привет!
# Дополнительный код, который исполняется после вызова функции
```

Этот код создает декоратор `my_decorator`, который добавляет дополнительный код до и после выполнения функции `say_hello()`. Декоратор применяется к `say_hello()` с помощью синтаксиса `@my_decorator`.

Таким образом, написав свой собственный декоратор, вы можете расширить функциональность функций, не изменяя их исходный код.

Вот, к примеру, декоратор, который позволяет измерять время выполнения функции:

```
from time import time

def executiontime(func):
    def wrapper():
```

```
        start = time()
        func()
        end = time()
        print(f'Функция {func} выполнялась: {end - start} сек')
    return wrapper

@executiontime
def create_tuple():
    return tuple(range(10**7))

create_tuple()
```

Суть двумя словами: по сути декоратор принимает на вход другую функцию и позволяет её модифицировать снаружи, не меняя внутренней реализации самой функции. Кстати, один из полезнейших декораторов — `@njit()` из библиотеки `numba`, позволяет космически ускорить Python.

Ну и неплохая статья — 6 Python декораторов, которые значительно упростят ваш код

17. Что может быть ключом в словаре?

В Python ключом в словаре может быть любой неизменяемый объект, такой как число, строка или кортеж.
Например:

```
my_dict = {1: 'one', 'two': 2, (3, 4): 'three four'}
```

В этом примере ключами словаря являются число `1`, строка `'two'` и кортеж `(3, 4)`.

Однако, если вы попытаетесь использовать изменяемый объект, такой как список, как
ключ словаря, вы получите `TypeError`:

```
my_dict = {[1, 2]: 'one two'}
# this will raise a TypeError: unhashable type: 'list'
```

Также, если вы попытаетесь добавить два ключа в словарь с одинаковым хеш-кодом, то второй ключ перезапишет первый:

```
my_dict = {1: 'one', '1': 'one again'}
# {1: 'one again'}
```

Кстати, а что вернёт данный код?

```
a = {True : 'a', 1 : 'b', '1' : 'c', 1.0 : 'd'}
print(a[True])
```

18. В чём разница между пакетами и модулями?

Модуль - это файл, содержащий код Python, который может быть повторно использован в других программах.

Пакет - это директория, содержащая один или несколько модулей (или пакетов внутри пакетов), а также специальный файл `init.py`, который выполняется при импорте пакета. Он может содержать код, который инициализирует переменные, функции и классы, и становится доступным для использования внутри модулей, находящихся внутри этого пакета.

Таким образом, основная разница между модулем и пакетом заключается в том, что модуль - это файл с кодом, который можно использовать повторно, а пакет - это директория, которая может содержать один или несколько модулей. Код, находящийся в

файле `init.py`, может инициализировать переменные, функции и классы, что обеспечивает общую функциональность для всех модулей, находящихся внутри пакета.
Например, если у нас есть пакет `mypackage`, в нем может находится несколько модулей, таких как `module1.py`, `module2.py`. В файле `init.py` определяются функции и переменные, которые могут использоваться внутри `module1` и `module2`.

Некоторые примеры импорта:

```
import mymodule # импортируем модуль
from mypackage import mymodule # импортируем модуль из пакета
from mypackage.mymodule import myfunction # импортируем функцию из модуля в пакете
```

19. Для чего используется дандер-метод init?

Функция (дандер-метод, если точнее) `__init__` является конструктором класса, и она вызывается автоматически при создании нового экземпляра класса. `__init__` используется для инициализации атрибутов, которые будут принадлежать объектам, создаваемым с помощью класса.

Внутри функции `__init__` определяются атрибуты объекта, которые будут доступны через ссылку на экземпляр, на который ссылается переменная `self`.

Например, пусть каждый экземпляр класса `Person` создаётся с атрибутами `name` и `age`:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person1 = Person("John", 30)
person2 = Person("Alice", 25)
print(person1.name) # output: John
print(person2.age) # output: 25
```

Тут функция `init` инициализирует атрибуты `name` и `age` для каждого экземпляра, создаваемого с помощью класса `Person`.
Когда мы создаем новый объект, мы передаем эти аргументы в функцию `init`, чтобы инициализировать соответствующие атрибуты.

20. Что такое слайс(slice)?

Совсем банальный вопрос, но ладно.

Слайс (slice) - это способ извлечения определенной части последовательности (например, строки, списка, кортежа) с использованием индексации.

Синтаксис для создания слайса:

```
sequence[start:end:step]
```

где `start` - индекс, с которого начинается извлечение (включительно), `end` - индекс, на котором заканчивается извлечение (не включая его), и `step` - шаг для извлечения элементов (по умолчанию равен 1).

Обратите внимание, что если не указывать `start`, то по умолчанию он равен 0, а если не указывать `end`, то по умолчанию он равен длине последовательности.

Вот пример использования слайса для выбора подряд идущих элементов списка (`list`):

```
my_list = [0, 1, 2, 3, 4, 5]
my_slice = my_list[1:4] # выбираем элементы с индексами от 1 до 3 включительно
```

```
print(my_slice) # выведет [1, 2, 3]
```

В этом примере мы использовали слайс `my_list[1:4]` для выбора элементов списка с индексами от 1 до 3 включительно.

В общем, слайс/срез используется для того, чтобы взять какую-то подпоследовательность любого итерируемого объекта, будь то строка, список или кортеж; при этом мы можем указывать начало среза, конец и шаг.

21. Как проверить, что один кортеж содержит все элементы другого кортежа?

Для проверки того, содержит ли один кортеж все элементы другого кортежа в Python, можно воспользоваться встроенной функцией `all()`, передав ей выражение генератора
списков, которое проверяет наличие каждого элемента из второго кортежа в первом кортеже.

Например:

```
first_tuple = (1, 2, 3, 4, 5)
second_tuple = (2, 4, 5)

contains_all = all(elem in first_tuple for elem in second_tuple)
print(contains_all) # True
```

Этот код создает два кортежа `first_tuple` и `second_tuple` и затем использует генератор списка, чтобы проверить, содержит ли `first_tuple` все элементы из `second_tuple`.
Результат будет `True`, если все элементы второго кортежа содержатся в первом кортеже, и `False` в противном случае.

Есть и другой вариант для наборов неповторяющихся элементов. Можно использовать `issubset()`, то есть проверить, является ли одно множество подмножеством другого:

```
first_tuple = (1, 2, 3, 4, 5)
some_list = [2, 4, 5]

contains_all = set(some_list).issubset(set(first_tuple))
print(contains_all) # True
```

Этот код дает тот же результат, что и предыдущий пример, но при преобразовании в `set()` повторы теряются, подойдёт только для наборов уникальных элементов.

22. Почему пустой список нельзя использовать как аргумент по умолчанию?

Значения по умолчанию для аргументов функции вычисляются только 1 раз, когда функция определяется, а не каждый раз, когда она вызывается. Таким образом, если вы попытаетесь использовать изменяемый тип данных (например, список) как аргумент по умолчанию для функции, то каждый вызов функции, который изменяет это значение, также изменит значение по умолчанию для всех последующих вызовов функции. Это может привести к разным сюрпризам и неожиданным последствиям.

Пустой список - это изменяемый тип данных в Python, поэтому его использование в качестве аргумента по умолчанию не рекомендуется. Вместо этого лучше использовать `None` в качестве значения по умолчанию и создавать новый пустой список внутри функции, если требуется список.

Типо того:

```
def my_function(my_list=None):
    if my_list is None:
        my_list = []
        # do something with my_list
        pass
```

При такой реализации вы всегда можете быть уверены, что получаете новый объект списка при каждом вызове функции.

P.S. На самом деле есть обходной путь: если внутри функции входной аргумент (список) никак не меняется, то код будет абсолютно валидным.

Это реально работает:

```
def foo(var: int, checks: list[Callable] = []):
    for check in checks:
        check(var)
```

Но линтеру это может не понравиться, или кто-то может заставить функцию изменять список и всё сломается, да и не по феншую это.

23. Что такое @classmethod, @staticmethod, @property?

`@classmethod` , `@staticmethod` , and `@property` - это декораторы методов класса в языке Python.

`@classmethod` используется для создания методов, которые будут работать с классом в целом, а не с отдельным экземпляром. В качестве первого параметра этот метод принимает класс, а не экземпляр объекта, и часто используется для создания фабричных методов и методов, которые работают с класс-уровнем методов.

`@staticmethod` декоратор работает подобно `@classmethod` , но он не получает доступ к классу в качестве первого параметра.

`@property` декоратор используется для создания свойств объекта, которые можно получить и задать, но выглядят как обычные атрибуты объекта. Это позволяет управлять доступом к атрибутам объекта, установив условиями доступа и возможностью заложить дополнительную логику при чтении, установке или удалении атрибута.

Например, явное использование декораторов может выглядеть так:

```
class MyClass:
    def __init__(self, value):
        self._value = value

    @classmethod
    def from_string(cls, input_string):
        value = process_input_string(input_string)
        return cls(value)

    @staticmethod
    def process_input_string(input_string):
        # implementation details
        pass

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, new_value):
        if new_value < 0:
            raise ValueError("Value must be positive")
        self._value = new_value
```

Декорированные методы могут быть использованы для достижения различных целей, таких как доступ к класс-уровню, расширение функциональности объекта и управление доступом к атрибутам.

24. Что такое синхронный код?

Синхронный код - это код, который выполняется последовательно, один за другим, и блокирует выполнение других задач до его завершения. Это означает, что если у вас есть функция, которая занимает много времени на выполнение, и вы вызываете ее в основной программе, то выполнение программы заблокируется до завершения этой функции.

Примером синхронного кода в Python может служить следующий фрагмент, который содержит цикл while, обрабатывающий список элементов:

```
items = [1, 2, 3, 4, 5]

for item in items:
    print(item)
```

Здесь цикл `for` будет обрабатывать каждый элемент в списке `items` последовательно, один за другим, и не будет переходить к следующему элементу, пока не завершится обработка текущего элемента.

Выполнение синхронного кода может занять много времени и может вызвать проблемы с производительностью, особенно когда код выполняет блокирующие операции, такие как чтение и запись файлов, обращение к сети, или поиск значений в базе данных. Для решения этой проблемы в Python используют асинхронное программирование с использованием конструкций `async/await` и библиотеки `asyncio` . Они позволяют выполнять несколько задач асинхронно, не блокируя выполнение других задач, и добиваться более высокой производительности.

25. Что такое асинхронный код? Приведите пример.

Асинхронный код - это подход к написанию кода, который позволяет выполнять несколько задач одновременно в рамках одного процесса. Это достигается за счет использования асинхронных функций и корутин. В отличие от синхронного кода, который выполняет каждую задачу последовательно, синхронный код может запустить несколько задач "параллельно" и организовать их выполнение с помощью итераций и вызовов коллбеков.

Примером использования асинхронного кода является библиотека `asyncio` в Python.

Например, вот простой пример кода, который использует `asyncio` для запуска
нескольких задач одновременно и ожидания их завершения:

```
import asyncio

async def hello():
    await asyncio.sleep(1)
    print("Hello")

async def world():
    await asyncio.sleep(2)
    print("World")

async def main():
    await asyncio.gather(hello(), world())

if __name__ == '__main__':
    asyncio.run(main())
```

В этом примере мы определяем 3 асинхронные функции: `hello()` , `world()` и `main()` . Функции `hello()` и `world()` печатают соответствующие сообщения и ждут 1 и 2 секунды
соответственно.

Функция `main()` запускает эти две функции одновременно с помощью `asyncio.gather()` и ждет, пока они завершат свою работу. Затем мы запускаем
функцию `main()` с помощью `asyncio.run()` . В результате мы получим сообщения `"Hello"` и `"World"` , каждое через 1 и 2 секунды соответственно, при этом результаты двух
задач были получены почти одновременно.

26. Каким будет результат следующего выражения?

Уверен, все знают, эту фишку, но я рискнул и поместил этот вопрос.

```
-31 % 10
```

Результатом выражения `-31 % 10` будет `9`. Это происходит потому, что для отрицательных чисел оператор `%` возвращает остаток от деления первого числа на второе немного другим образом. `-31 % 10 = -3 - 1/10` и в ответ мы получим `10 - 1 = 9`

27. Для чего нужен метод id()?

Метод `id()` используется для получения уникального целочисленного идентификатора (адреса в памяти) объекта. Этот идентификатор может быть использован для сравнения объектов, поскольку два объекта будут иметь одинаковый идентификатор только в том случае, если это один и тот же объект в памяти.

Например, если у вас есть две переменные, которые ссылаются на один и тот же объект, то их идентификаторы будут равны:

```
a = [1, 2, 3]
b = a
print(id(a)) # выведет адрес в памяти объекта a
print(id(b)) # выведет адрес в памяти объекта b
```

Однако, если у вас есть две переменные, которые ссылаются на разные объекты, их идентификаторы будут отличаться:

```
a = [1, 2, 3]
b = [1, 2, 3]
print(id(a)) # выведет адрес в памяти объекта a
print(id(b)) # выведет адрес в памяти объекта b (отличный от идентификатора a)
```

Использование метода `id()` может быть полезно при отладке или проверке, какие переменные ссылаются на один и тот же объект. Однако, в общем случае, использование метода `id()` не рекомендуется, поскольку это может быть неэффективным при работе с большим количеством объектов в памяти.

28. Что такое итератор?

Итератор (Iterator) — это объект, который возвращает свои элементы по одному за раз.

Он должен иметь метод `next()`, который возвращает следующий элемент и вызывает исключение `StopIteration`, когда элементы закончились. Итератор также может быть написан с помощью генераторов.

Пример использования итератора в Python:

```
my_list = [1, 2, 3, 4, 5]
my_iterator = iter(my_list)  # Получаем итератор из списка

print(next(my_iterator)) # выведет 1
print(next(my_iterator)) # выведет 2
print(next(my_iterator)) # выведет 3
```

В этом примере мы создаем список и получаем из него итератор. Затем мы выводим элементы итератора с помощью функции `next()`, которая вызывает метод `next()`
объекта итератора. Каждый вызов функции `next()` выводит следующий элемент, пока не закончатся элементы списка, после чего будет вызвано исключение `StopIteration`.

Еще один способ создания итераторов в Python — использование генераторов. Генератор — это функция, которая возвращает итерируемый объект (такой, как список или кортеж). Вместо того, чтобы возвращать все элементы сразу, генератор возвращает элементы по одному по мере необходимости.

Например (спасибо поддержке юникода в питоне):

```
# Определяем генератор
def my_generator():
    yield 🦝
    yield 🦊
    yield 🐓

# Получаем итератор из генератора
my_iterator = my_generator()

# Выводим элементы итератора
print(next(my_iterator)) # выведет 🦝
print(next(my_iterator)) # выведет 🦊
print(next(my_iterator)) # выведет 🐓
```

А вот отличная статья по теме — 10 итераторов, о которых вы могли не знать

29. Что такое генератор? Чем отличается от итератора?

Генератор - это функция, которая использует ключевое слово `yield` для возврата итератора.

Генератор может быть использован для создания последовательности значений, которые генерируются в момент обращения к ним, что позволяет эффективно использовать память и ускоряет выполнение программы. Короче, генератор основан на тех самых "ленивых" (отложенных) вычислениях.

Отличие генератора от итератора заключается в том, что итератор используется для обхода коллекции (например, списка) до тех пор, пока все элементы не будут перебраны, а генератор используется для создания последовательности значений.

Итераторы также могут быть созданы как классы, которые реализуют методы `iter()` и `next()` , в то время как генераторы создаются при помощи функций и используют ключевое слово `yield` .

Пример использования генератора, который генерирует последовательность чисел от 0 до n включительно:

```
def my_generator(n):
    for i in range(n + 1):
        yield i

my_gen = my_generator(5)
for i in my_gen:
    print(i)
```

Этот код создаст объект генератора `my_gen` , который можно использовать для последовательного получения каждого из значений, произведенных генератором при помощи ключевого слова `yield` .

Если хочется подробностей про генераторы, вот отличная статья. А ещё Тимофей Хирьянов неплохо про это объясняет, можно заглянуть к нему на канал.

30. Для чего используется ключевое слово yield?

Ключевое слово `yield` используется для создания генераторов.

Генератор - это функция, которая может возвращать последовательность значений используя инструкции `yield` вместо `return` . При каждом вызове инструкции `yield` генератор возвращает значение, после чего сохраняет свое состояние и приостанавливает свое выполнение до следующего вызова.

Это позволяет генерировать последовательности значений без необходимости создания и хранения всех значений в памяти, что может быть особенно полезно при работе с большими объемами данных. Кроме того, генераторы являются итерируемыми и могут использоваться в циклах `for`.

31. Чем отличаются iter и next?

`iter` и `next` являются специальными методами в Python, которые обеспечивают поддержку итерации для объектов.

Метод `iter` возвращает объект, который может быть использован для итерации по элементам контейнера. Объект, возвращаемый `iter`, должен содержать метод `next`.
Метод `next` должен вернуть следующий элемент в итерации или вызвать исключение `StopIteration`, если элементов больше нет.

Таким образом, метод `iter` используется для создания итератора, а метод `next` используется для перехода к следующему элементу в итерации.
В общем случае, класс должен определять метод `iter`, который возвращает сам объект класса, и метод `next`, который определяет, какие элементы будут возвращены при итерации.

Например:

```
class MyIterator:
    def __init__(self, data):
        self.index = 0
        self.data = data

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.data):
            raise StopIteration
        result = self.data[self.index]
        self.index += 1
        return result
```

Метод `iter` возвращает сам объект, а метод `next` возвращает следующий элемент `data` каждый раз, когда вызывается.

32. Что такое контекстный менеджер?

Контекстный менеджер в Python - это объект, который определяет вход и выход из контекста с помощью методов `enter()` и `exit()`.

Контекстный менеджер может быть использован в блоке `with` для выполнения конкретных действий при входе и выходе из блока. Например, контекстный менеджер может устанавливать и закрывать соединение с базой данных, блокировать и разблокировать файлы или временно изменять настройки системы.

Вот простой пример, демонстрирующий использование контекстного менеджера для работы с файлом:

```
with open('file.txt', 'r') as f:
    data = f.read()
```

В этом примере `open()` возвращает контекстный менеджер `f`. Когда блок `with` начинается, вызывается метод `enter()` контекстного менеджера, который открывает файл.
Затем выполняется код в блоке, который использует `f` для чтения данных из файла. При завершении блока `with` вызывается метод `exit()` контекстного менеджера, который закрывает файл.

Контекстные менеджеры в Python используются для обращения с ресурсами, которые должны быть корректно открыты и закрыты, включая файлы, сетевые соединения, блокировки и базы данных. Кроме того, их можно использовать для временной модификации состояния системы или окружения в блоках `with`.

33. Как сделать python-скрипт исполняемым в различных операционных системах?

Для того чтобы сделать Python-скрипт исполняемым в различных операционных системах, можно воспользоваться утилитой `PyInstaller`, которая позволяет упаковать скрипт в исполняемый файл для Windows, Linux и macOS.

Чтобы установить `PyInstaller`, можно выполнить следующую команду в командной строке:
```
pip install pyinstaller
```

После установки PyInstaller необходимо перейти в директорию с Python-скриптом и запустить утилиту с соответствующими параметрами для создания исполняемого файла. Например:
```
pyinstaller myscript.py --onefile
```

Эта команда создаст единый исполняемый файл `myscript.exe` (для Windows) или `myscript` (для Linux/macOS), который можно запустить на соответствующих операционных системах.

Если нужно создать исполняемый файл с определенными параметрами, можно воспользоваться другими параметрами `PyInstaller`, такими как `--icon` для добавления иконки, `--name` для задания имени исполняемого файла и т.д.
Но стоит отметить, что `PyInstaller` не является универсальным решением и возможна потребность в использовании других инструментов в зависимости от конкретной задачи и требований к исполняемому файлу.

34. Как сделать копию объекта? Как сделать глубокую копию объекта?

Метод `copy()` создает поверхностную копию объекта, то есть создает новый объект, который содержит ссылки на те же объекты, что и исходный объект. Если вы измените какой-либо из этих объектов, изменения отразятся и на копии, и на исходном объекте.

Метод `deepcopy()` создает глубокую копию объекта, то есть создает новый объект, который содержит копии всех объектов, на которые ссылаются элементы исходного объекта. Если вы измените какой-либо из этих объектов, изменения не отразятся на копии или на исходном объекте.

Вот примеры использования этих методов:

```
import copy
# создание копии объекта
new_list = old_list.copy()
# создание глубокой копии объекта
new_list = copy.deepcopy(old_list)
```

где `old_list` - исходный список, а `new_list` - его копия.

Примечание: для выполнения глубокого копирования объектов, сами объекты также должны поддерживать копирование. Если объекты в ваших данных не поддерживают копирование, `deepcopy()` вернет исходный объект, а не его копию.

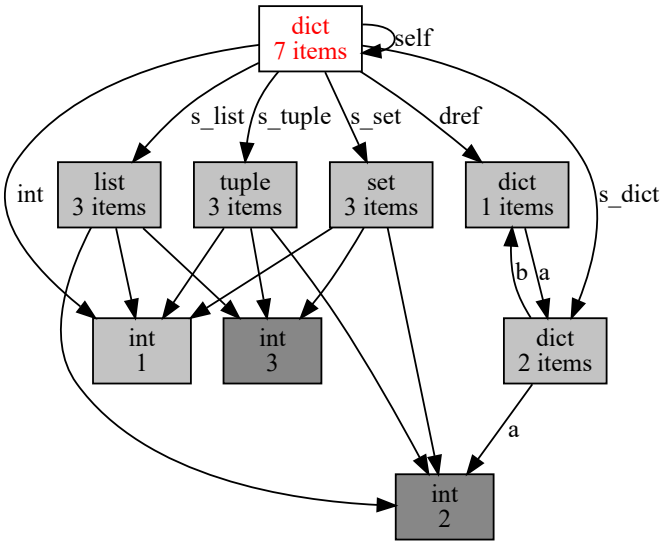
35. Опишите принцип работы сборщика мусора в python.

Python использует автоматическое управление памятью, что означает, что разработчику не нужно явно выделять или освобождать память в своем коде. Вместо этого в Python есть встроенный сборщик мусора, который автоматически управляет памятью для объектов, на которые больше нет ссылок.

Сборщик мусора запускается периодически и ищет объекты, на которые больше не ссылается ни одна переменная в коде. Затем эти объекты идентифицируются как мусор и удаляются из памяти. Сборщик мусора работает, отслеживая ссылки на объекты в памяти, используя механизм подсчета ссылок. Каждый раз, когда создается новая ссылка на объект, счетчик ссылок для этого объекта увеличивается. Точно так же, когда ссылка удаляется, счетчик ссылок уменьшается.

Однако одного подсчета ссылок недостаточно для обработки всех случаев управления памятью. В некоторых случаях могут быть циклические ссылки, когда два или более объекта ссылаются друг на друга и больше не нужны. Для обработки этих случаев сборщик мусора Python использует вторичный механизм, называемый «обнаружение циклов». Этот механизм периодически ищет циклические ссылки среди объектов, и если они найдены, он знает, что нужно удалить циклическую ссылку и освободить память.

Если интересно, визуализировать ссылки можно с помощью objgraph, выглядит как-то так:



image

В целом, сочетание подсчета ссылок и обнаружения циклов позволяет Python автоматически управлять памятью и обеспечивать очистку объектов, когда они больше не нужны. Это приводит к более эффективному использованию памяти и снижает риск нехватки памяти в приложениях, которые долго работают или интенсивно используют память.

Хорошая статья по теме — Всё, что нужно знать о сборщике мусора в Python

36. Как использовать глобальные переменные? Это хорошая идея?

Для использования глобальных переменных достаточно объявить их за пределами функций и классов.

Например:

```
# объявляем глобальную переменную
global_var = 42
def my_func():
    # можно использовать глобальную переменную
    global global_var
    print(global_var)

# вызываем функцию
my_func()
```

Однако, использование глобальных переменных — зло ~~не считается хорошей практикой программирования~~, так как это может привести к ошибкам при изменении значения переменной в разных частях программы. Вместо этого лучше использовать локальные переменные внутри функций или передавать значения между функциями через параметры и возвращаемые значения.

37. Для чего в классе используется атрибут slots?

Вот официальная документация по `__slots__` , а вот дополнительные разъяснения от одного из разработчиков официальной документации.

Атрибут `__slots__` в классе Python используется для оптимизации памяти и ускорения работы с объектами класса. Он позволяет явно указать, какие атрибуты объекта будут использоваться, а какие нет.

Когда вы определяете класс, Python создает для каждого экземпляра этого класса словарь, который содержит все его атрибуты. Это может быть выгодным в том случае, если у вас много различных атрибутов, но может привести к большому расходу памяти, если вы создаете много экземпляров класса с небольшим количеством атрибутов.

Атрибут `__slots__` позволяет определить, какие атрибуты должны быть на самом деле созданы для каждого экземпляра класса, и в какой момент их можно будет получить.

Если вы используете атрибут `__slots__`, Python уже не будет создавать словарь для каждого экземпляра класса, а будет использовать непосредственно массив атрибутов, что может ускорить работу программы и уменьшить использование памяти.

Например, если у вас есть класс `Person` с атрибутами `name` и `age`, вы можете определить `__slots__` следующим образом:

```
class Person:
    __slots__ = ['name', 'age']
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Таким образом, каждый экземпляр класса `Person` будет содержать только атрибуты `name` и `age`, и никакие другие атрибуты не будут созданы.

38. Какие пространства имен существуют в python?

Пространство имен — это совокупность определенных в настоящий момент символических имен и информации об объектах, на которые они ссылаются.

Python имеет множество встроенных пространств имен. Некоторые из них включают:

- `builtins` : содержит встроенные функции и типы, которые доступны в любой области видимости по умолчанию.
- `main` : это специальное пространство имен, которое содержит определения, которые были выполнены на верхнем уровне скрипта или интерактивной оболочки Python.
- `name` : это атрибут, который содержит имя текущего модуля. Если модуль импортирован, то значение name будет именем модуля. Если модуль запускается как скрипт, то значение name будет "main".
- `globals()` : это функция, которая возвращает словарь, содержащий все имена в глобальной области видимости.
- `locals()` : это функция, которая возвращает словарь, содержащий все имена в локальной области видимости.

Это далеко не полный список, но это некоторые из наиболее распространенных пространств имен в Python.

39. Как реализуется управление памятью в python?

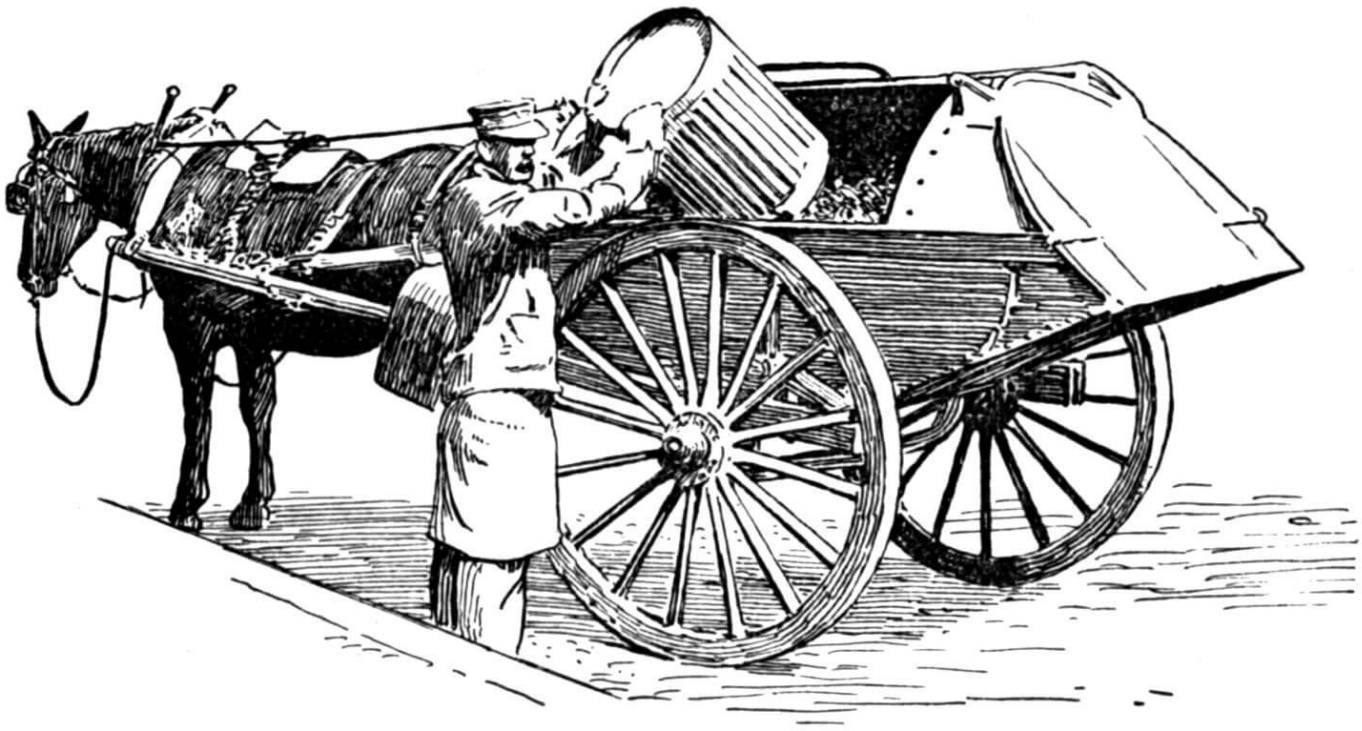


FIG. 109. — A GARBAGE COLLECTOR.

img

Управление памятью осуществляется автоматически с помощью механизма сборки мусора (`Garbage collector`).

Когда объект в Python больше не нужен (например, после того как на него уже нет ссылок), он помечается как `garbage` (мусор), после чего он будет автоматически удален при следующем запуске сборщика мусора.

Используется метод подсчета ссылок для отслеживания того, когда объект уже не нужен, и этот объект должен быть освобожден. Кроме того, Python также использует циклический сборщик мусора (`Cycle detector`), который может определить и удалить объекты, на которые ссылается другой объект, на который уже нет ссылок.

Сборка мусора в Python использует алгоритм под названием `reference counting` , который подсчитывает количество ссылок на каждый объект в памяти. Когда количество ссылок на объект становится равным нулю, он помечается как мусор и память автоматически освобождается.

В Python также реализованы другие алгоритмы сборки мусора, такие как `generational garbage collection` , который разбивает объекты на несколько "поколений" и собирает мусор с различной частотой в зависимости от поколения, в котором они находятся, но `reference counting` является основой управления памятью в Python.

Модуль `gc` в Python также предлагает дополнительный функционал для управления памятью. Например, метод `gc.collect()` позволяет сделать принудительную сборку мусора.

40. Что такое метаклассы и в каких случаях их следует использовать?

Метаклассы - это классы, которые определяют поведение других классов. Они используются для изменения способа, которым Python создает и обрабатывает классы.

Метаклассы могут быть полезны в следующих случаях:

- При необходимости динамического изменения поведения класса, например, если вы хотите добавить или удалить атрибут или метод класса во время выполнения программы.
- При создании классов из данных, которые не заранее известны. Например, вы можете создавать классы на основе определенных условий во время выполнения программы.
- Для создания фреймворков и библиотек, которые нужно настраивать под конкретные требования и при этом сохранить простоту интерфейса.

- Они также могут использоваться для создания классов с определенными свойствами, например, классов, которые автоматически регистрируются в библиотеке или классов, которые автоматически сериализуются и десериализуются для совместимости с другими системами.

Пример использования метакласса для добавления атрибута к классу:

```
class MyMeta(type):
    def __new__(cls, name, bases, dct):
        dct['my_attribute'] = 42
        return super(MyMeta, cls).__new__(cls, name, bases, dct)

class MyClass(metaclass=MyMeta):
    pass

print(MyClass.my_attribute)
```

В этом примере создается метакласс `MyMeta`, который добавляет атрибут `my_attribute` к любому классу, который использует данный метакласс для своего создания. Затем создается класс `MyClass`, который использует метакласс `MyMeta`. При вызове `print(MyClass.my_attribute)` выводится значение 42, так как этот атрибут был добавлен в момент создания класса.

Про метаклассы совершенно замечательно рассказывал Григорий Петров на Moscow Python — "Простой Python": ложь, большая ложь и метаклассы.

41. Зачем нужен pdb?

![image_2023-12-18_13-24-22](C:\Users\Admin\Downloads\Telegram Desktop\image_2023-12-18_13-24-22.png)

`pdb` - это интерактивный отладчик для Python, с помощью которого можно перемещаться по коду во время запуска вашей программы, смотреть и изменять значения переменных, построчно навигироваться по коду (в том числе углубляться во вложенности кода), назначать брейкпоинты и все прочие операции присущие отладчику.

Модуль `pdb` предоставляет интерфейс командной строки, который можно использовать для взаимодействия с кодом Python во время его выполнения. Вы можете войти в режим `pdb` в своей программе Python, вставив следующую строку кода там, где вы хотите остановить отладчик:

```
import pdb;
pdb.set_trace()
```

Когда интерпретатор дойдет до этой строки, он приостановится, и можно использовать команды `pdb` для проверки состояния вашей программы. Таким образом, `pdb` — это полезный инструмент для отладки кода Python, поскольку он позволяет в интерактивном режиме проверять состояние кода и выявлять проблемы.

Много полезного про pdb можно почерпнуть тут — Профилирование и отладка Python

42. Каким будет результат следующего выражения?

Супер простой вопрос, но, согласитесь, ошибиться возможно.

```
[0, 1][10:]
```

Выражение `[0, 1][10:]` возвращает пустой список `[]`, так как срез `[10:]` означает извлечение элементов начиная с индекса `10` и до конца списка `[0, 1]`, но таких элементов нет.

43. Как создать класс без слова class?

Класс можно создать без использования ключевого слова `class`, используя типы `type` или `metaclass`.

Например, следующий код определяет класс `MyClass` без использования ключевого слова `class`:

```
MyClass = type('MyClass', (), {'x': 42, 'foo': lambda self: self.x})
```

Этот код эквивалентен определению класса с использованием ключевого слова `class`:

```
class MyClass:
    x = 42
    def foo(self):
        return self.x
```

Оба определения класса эквивалентны и создают объект класса `MyClass`. Но, честно, использование ключевого слова `class` читабельнее, хотя где-нибудь может пригодиться и `type` с `metaclass`.

44. Как перезагрузить импортированный модуль?

Чтобы перезагрузить импортированный модуль в Python, вы можете использовать функцию `reload()` из модуля `importlib`.

Вот как это сделать:

```
from importlib import reload
import module_name
reload(module_name)
```

Замените `module_name` на фактическое имя модуля, который вы хотите перезагрузить.
Это может быть полезно при разработке и тестировании модулей, но не рекомендуется использовать в производственном коде без серьезных причин.

45. Напишите декоратор, который будет перехватывать ошибки и повторять функцию максимум N раз.

Вот пример декоратора на Python, который будет перехватывать ошибки и повторять функцию максимум N раз:

```
import functools
def retry(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        max_retries = 3
        for i in range(max_retries):
            try:
                result = func(*args, **kwargs)
                return result
            except Exception as e:
                print(f'Error occurred: {e}. Retrying ({i+1}/{max_retries})...')
        raise Exception(f'Function {func.__name__} failed after {max_retries} attempts.')
    return wrapper
```

Для использования декоратора вам нужно добавить `@retry` перед определением функции, которую вы хотите обернуть:

```
@retry
def my_function(arg1, arg2):
    pass
```

В этом примере функция `my_function` будет повторно вызываться до трех раз в случае ошибки, до тех пор, пока она не выполнится успешно. Если после трех попыток
функция не выполнится успешно, вы получите исключение.

Вы можете настроить параметр `max_retries` , чтобы изменить количество попыток или добавить его как аргумент декоратора, чтобы делать эти настройки динамически.

46. Каким будет результат следующего выражения?

```
len(' '.join(list(map(str, [0, 1]))))
```

Внутреннее выражение `map(str, [[0], [1]])` преобразует целые значения 0 и 1 в строки, в результате чего получается `['0', '1']` .

Затем функция списка преобразует этот итератор в список.
Метод соединения соединяет элементы списка пробелом, в результате чего получается строка `"0 1"` .

Наконец, функция `len` возвращает длину этой строки, которая равна 3. Вот и всё.

47. Какие проблемы есть в python?

Python, как и любой язык программирования, имеет свой набор потенциальных проблем и ограничений.

Вот некоторые из распространенных проблем, с которыми сталкиваются разработчики при работе с Python:

- **Глобальная блокировка интерпретатора (GIL)** — это механизм в реализации Python на CPython, который предотвращает одновременное выполнение кода Python
  несколькими потоками. В некоторых случаях это может ограничить производительность задач, связанных с процессором.
- **Управление пакетами и зависимостями**. Управление сторонними пакетами и зависимостями в Python иногда может быть сложным, особенно для крупных проектов
  или в сложных средах.
- **Производительность**. Хотя Python обычно считается быстрым языком, он не может быть оптимальным выбором для задач, требующих высокой производительности,
  таких как машинное обучение или научные вычисления.
- **Типизация и статический анализ**. Python — это язык с динамической типизацией, что может затруднить обнаружение определенных типов ошибок во время
  компиляции.
- **Управление памятью**: автоматическое управление памятью в Python может в некоторых случаях привести к утечке памяти или неэффективному использованию
  памяти.
- **Документация**: Хотя сообщество Python уделяет большое внимание документации, некоторые пакеты или библиотеки могут иметь неполную или устаревшую
  документацию, что может затруднить их эффективное использование.

Но не стоит грустить, ~~и переходить на C~~, многие из этих проблем не уникальны для Python, и часто существуют обходные пути или решения. К тому же Python имеет большое и активное сообщество пользователей и разработчиков, которые постоянно работают над улучшением языка и решением этих и других проблем.

А вот неплохая статья, посвящённая проблемам в Python — Я люблю питон, и вот почему он меня бесит

Или Python — это медленно. Почему?

48. Когда будет выполнена ветка else в конструкции try…except…else?

Ветка `else` в конструкции `try ... except ... else` будет выполнена только в том случае, если исключения не было возбуждено в блоке `try` .

Если в блоке `try` произошло исключение, то выполнение программы переходит к соответствующему блоку `except`, и ветка `else` пропускается. Если блок `except` не указан, то исключение будет возбуждено дальше, а программа завершится с сообщением об ошибке.

```
a, b = map(int, input().split())
try:
    print(a / b)
except:
    print('Ошибка')
else:
    print('Ошибки не было')  # это выводится для любых чисел a, b, если b != 0
```

49. Поддерживает ли python множественное наследование?

Да, Python поддерживает множественное наследование. Это означает, что класс может наследовать функциональность от нескольких предков, путем указания их имен в скобках при определении класса.

Например:

```
class MyBaseClass1:
    pass

class MyBaseClass2:
    pass

class MyDerivedClass(MyBaseClass1, MyBaseClass2):
    pass
```

В этом случае `MyDerivedClass` является подклассом `MyBaseClass1` и `MyBaseClass2`, и поэтому наследует их функциональность. Класс `MyDerivedClass` может использовать методы и атрибуты, определенные в `MyBaseClass1` и `MyBaseClass2`.

Существует несколько способов объявления класса, который наследует от нескольких родительских классов, но один из распространенных способов - это просто указать несколько родительских классов в скобках при определении класса-потомка.

Следующий код определяет класс `MyClass`, который наследует от классов `Parent1` и `Parent2`:

```
class Parent1:
    def method1(self):
        print("This is a method from Parent1")

class Parent2:
    def method2(self):
        print("This is a method from Parent2")

class MyClass(Parent1, Parent2):
    pass

obj = MyClass()
obj.method1() # outputs "This is a method from Parent1"
obj.method2() # outputs "This is a method from Parent2"
```

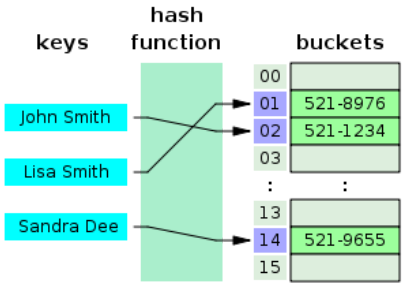
Приведенный выше код создает `MyClass`, который наследует свойства и методы как от класса `Parent1`, так и от класса `Parent2`. Вы можете вызвать методы как от `Parent1`, так и от `Parent2` через объект `MyClass`.

50. Как dict и set реализованы внутри? Какова сложность получения элемента? Сколько памяти потребляет каждая структура?

`Dict` и `Set` реализованы в виде хэш-таблицы.

Хэш-таблица - это структура данных, которая использует хэш-функцию для преобразования ключа в индекс в массиве, где хранятся значения. Затем элемент добавляется в массив по соответствующему индексу.

Так работает хэш-таблица:



img

Сложность получения элемента в `Dict` и `Set` в наилучшем случае составляет `O(1)`, поскольку элемент может быть получен просто с помощью хэш-функции в качестве индекса массива. Однако в худшем случае, когда возникают хэш-коллизии, сложность может вырасти до `O(n)`, где `n` - количество элементов в таблице.

Ну и сложность операций добавления, удаления и поиска элементов в `Set` и `Dict` также составляет `O(1)` в наилучшем случае и `O(n)` в худшем случае.

51. Что такое MRO? Как это работает?

`MRO (Method Resolution Order)` - это порядок разрешения методов, который используется в языке программирования Python при наследовании классов.

Когда вызывается метод на экземпляре класса, Python ищет этот метод в самом классе, а затем в его родительских классах в порядке, определенном в `MRO`. Таким образом, `MRO` управляет тем, как Python ищет методы, которые были унаследованы из нескольких родительских классов.

Порядок `MRO` может быть определен несколькими способами, но в общем случае `MRO` определяется с помощью алгоритма `C3`, который гарантирует, что порядок разрешения методов будет соблюдать локальный порядок наследования каждого класса и не создавать циклов в определении этого порядка.
Например, если класс `A` наследуется от классов `B` и `C`, а класс `B` наследуется от класса `D`, а класс `C` наследуется от класса `E`, то `MRO` для класса `A` будет определен как `[A, B, D, C, E, object]`.

Это означает, что если существует метод, определенный в классе `A` и в одном из его родительских классов, то метод из класса `A` будет вызван, а не из его родительских классов.

52. Как аргументы передаются в функции: по значению или по ссылке?

В Python аргументы передаются по ссылке на объект. Это означает, что когда вы передаете объект в качестве аргумента функции, функция получает ссылку на этот объект, а не его копию. Если вы модифицируете объект внутри функции, эти изменения будут отражены и вне функции, так как обе переменные (внутри и вне функции) ссылаются на один и тот же объект в памяти. Однако, если внутри функции вы присваиваете новое значение аргументу, это не изменит значение переменной, которую вы использовали при вызове функции, потому что эта переменная по-прежнему ссылается на тот же объект в памяти.
Например:

```
def increment(x):
    x += 1
    return x

y = 10
print(increment(y)) # Output: 11
print(y) # Output: 10
```

Здесь модификации `x` внутри функции не влияют на значение переменной `y`, так как теперь `x` ссылается на новый объект в памяти (увеличенное значение на 1), но `y` по-прежнему ссылается на старый объект (изначальное значение 10).

При работе со изменяемыми объектами (например, списками), модификация объекта внутри функции будет отражаться вне функции. Например:

```
def modify_list(lst):
    lst.append(4)

my_list = [1, 2, 3]
modify_list(my_list)
print(my_list) # Output: [1, 2, 3, 4]
```

Здесь модификации списка `lst` в функции `modify_list` отражаются и на переменной `my_list`, так как обе переменные ссылаются на один и тот же список в памяти.

53. С помощью каких инструментов можно выполнить статический анализ кода?

Для статического анализа кода есть несколько инструментов:

- `Pylint` - это инструмент, который анализирует исходный код на соответствие PEP8, а также предупреждает о потенциальных ошибках в коде.
- `Flake8` - это комбинированный инструмент, который объединяет в себе Pylint, PyFlakes и множество других правил, обеспечивающих соответствие стиля написания кода и обнаруживающих ошибки в исходном коде.
- `Mypy` - это статический типизатор для Python, который позволяет находить ошибки в типах переменных в исходном коде.
- `Bandit` - это инструмент для поиска уязвимостей в исходном коде Python.
- `Black` - это инструмент для автоматического форматирования кода Python, который придерживается только одного стиля написания кода.
- `Pycodestyle` — это простая консольная утилита для анализа кода Python, а именно для проверки кода на соответствие PEP8. Один из старейших анализаторов кода, до 2016 года носил название pep8, но был переименован по просьбе создателя языка Python Гвидо ван Россума.
- `Vulture` — это небольшая утилита для поиска “мертвого” кода в программах Python. Она использует модуль ast стандартной библиотеки и создает абстрактные синтаксические деревья для всех файлов исходного кода в проекте. Далее осуществляется поиск всех объектов, которые были определены, но не используются. Vulture полезно применять для очистки и нахождения ошибок в больших базовых кодах.

Эти инструменты могут улучшить качество кода, облегчить его чтение и поддержку и всё такое. Ну и помогут избежать ошибок, связанных с типами переменных, например.

Подробнее о статическом анализе кода в целом можно почитать тут: Внедряйте статический анализ в процесс

54. Что будет напечатано в результате выполнения следующего кода?

```
import sys
arr_1 = []
arr_2 = arr_1
print(sys.getrefcount(arr_1))
```

В результате выполнения данного кода будет напечатано число, равное количеству ссылок на объект `arr_1`, которые существуют в настоящий момент времени. Так как мы создаем две переменные, `arr_1` и `arr_2`, которые ссылаются на один и тот же пустой список `[]`, то количество ссылок на него будет равно 2.

Поэтому в результате выполнения данного кода будет напечатано число 2. Эта величина может быть немного больше, чем ожидается, из-за внутренней оптимизации CPython, которая добавляет временные ссылки на объекты.

55. Что такое GIL? Почему GIL всё ещё существует?

`GIL (Global Interpreter Lock)` - это механизм в интерпретаторе CPython , который гарантирует, что только один поток исполнения может выполнять байт-код Python в любой момент времени. Это было добавлено в Python для обеспечения безопасности потоков в многопоточной среде и для упрощения реализации интерпретатора. `GIL` всё ещё существует, потому что он является важной частью интерпретатора CPython и его логики работы с потоками.

Однако, недавние версии Python имеют некоторые механизмы для обхода ограничений `GIL` , такие как использование многопроцессных вычислений вместо многопоточных и использование асинхронного
программирования. Кроме того, есть и другие реализации языка Python, такие как Jython и IronPython, которые не используют `GIL` .

Короче, вопрос насколько `GIL` ограничивает производительность Python сейчас является спорным и холиварным.

Подробнее о GIL можно почитать тут: Python — это медленно. Почему?

56. Опишите процесс компиляции в python.

Python — это интерпретируемый язык, а это значит, что он не требует компиляции, как C или C++. Вместо этого интерпретатор Python читает и выполняет исходный код напрямую.

Однако Python использует форму компиляции, называемую компиляцией байт-кода. Когда сценарий Python запускается в первый раз, интерпретатор компилирует его в байтовый код, представляющий собой низкоуровневое представление исходного кода. Затем этот байт-код выполняется виртуальной машиной Python (PVM), которая представляет собой интерпретатор, который считывает байт-код и выполняет его.

Байт-код хранится в каталоге `pycache` с расширением `.pyc` . Python проверяет, есть ли у файла `.py` уже соответствующий файл `.pyc` , и, если файл `.pyc` старше файла `.py` , он компилирует файл `.py` в новый файл `.pyc` .

Таким образом, процесс «компиляции» в Python включает интерпретатор, который компилирует исходный код в байтовый код, который затем выполняется PVM. Однако этот процесс происходит автоматически и за кулисами, без необходимости пользователю явно вызывать отдельный шаг компиляции.

Кстати, процесс превращения Python-кода в байт-код превосходно и доходчиво расписан на в этом ролике, посмотрите

57. Что такое дескрипторы? Есть ли разница между дескриптором и декоратором?

Дескрипторы - это объекты Python, которые определяют, как другие объекты должны вести себя при доступе к атрибуту. Дескрипторы могут использоваться для
реализации протоколов, таких как протокол доступа к атрибутам, протокол дескрипторов и протокол методов.

Декораторы - это функции Python, которые принимают другую функцию в качестве аргумента и возвращают новую функцию. Декораторы обычно используются для
изменения поведения функции без изменения ее исходного кода.

Разница между дескриптором и декоратором заключается в том, что дескрипторы используются для определения поведения атрибутов объекта, в то время как декораторы
используются для изменения поведения функций. Однако, декораторы могут использоваться для реализации протоколов дескрипторов.

Например, декоратор `@property` можно использовать для создания дескриптора доступа к атрибутам. Он преобразует метод класса в дескриптор, который позволяет получать, устанавливать и удалять значение атрибута как обычный атрибут объекта.

58. Почему всякий раз, когда python завершает работу, не освобождается вся память?

Python использует автоматическое управление памятью с помощью механизма сборки мусора, который освобождает память, занятую объектами, которые больше не
используются в программе. Однако, до того как механизм сборки мусора может освободить память объекта, все ссылки на этот объект должны быть удалены. Если в
программе остаются ссылки на объекты, которые больше не нужны, то эти объекты не будут удалены до окончания работы приложения.

Также может случиться, что размер объектов, которые использует программа, слишком велик для доступной оперативной памяти. В этом случае операционная система

может начать использовать файл подкачки, что может замедлить работу программы.

Если вы столкнулись с проблемой утечки памяти, то можно воспользоваться инструментами, такими как `memory_profiler` для Python, которые помогут выявить места, где
память не освобождается, и найти способы ее оптимизации.

59. Что будет напечатано в результате выполнения следующего кода?

```
class Variable:
    def __init__(self, name, value):
        self._name = name
        self._value = value

    @property
    def value(self):
        print(self._name, 'GET', self._value)
        return self._value

    @value.setter
    def value(self, value):
        print(self._name, 'SET', self._value)
        self._value = value


var_1 = Variable('var_1', 'val_1')
var_2 = Variable('var_2', 'val_2')
var_1.value, var_2.value = var_2.value, var_1.value
```

При выполнении этого кода будет выведено следующее:

```
var_2 GET val_2
var_1 GET val_1
var_2 SET val_1
var_1 SET val_2
```

В этом коде определяется класс `Variable` со свойствами `name` и `value` . Метод `@property` используется для определения свойства значения, которое можно прочитать с помощью `getter` (функция, используемая для получения значения свойства) и установить новое значение с помощью `setter` (функция, используемая для установки нового значения свойства).

Затем создаются два экземпляра класса, и значения их свойств `value` меняются по очереди с помощью кортежа.
При каждом вызове метода `value` класса `Variable` выводится сообщение о том, что происходит (`GET` - когда значение свойства читается, `SET` - когда устанавливается новое значение свойства).

60. Что такое интернирование строк? Почему это есть в python?

Интернирование строк - это процесс, при котором две или более строковые переменные, содержащие одинаковое значение, ссылаются на один и тот же объект в памяти.

В Python интернирование строк происходит автоматически при создании строковых констант в исходном коде программы. Это означает, что если две или более строковые константы содержат одинаковое значение, они будут ссылаются на один и тот же объект в памяти.

Интернирование строк применяется для оптимизации использования памяти и ускорения выполнения программы. Поскольку операция сравнения двух строк, ссылающихся на один и тот же объект в памяти, выполняется быстрее, чем сравнение двух строк, которые хранятся в разных объектах в памяти.

В Python интернирование строк применяется для строковых констант, которые состоят из символов ASCII и имеют длину не более 20 символов. Это объясняется тем, что длинные строки могут занимать слишком много места в памяти, что может привести к проблемам производительности.

Интернирование строк является одним из многих способов оптимизации производительности, доступных в Python. Оно позволяет ускорить выполнение программы за счет сокращения использования памяти и оптимизации операций сравнения строк.

Пример кода, который демонстрирует интернирование строк в Python:

```
a = 'hello'
b = 'hello'
print(a is b) # True, потому что обе переменные ссылаются на один и тот же объект в памяти

c = 'hello world'
d = 'hello world'
print(c is d) # False, потому что строка "hello world" длиннее 5 символов и не является интернированной

e = '_123'
f = '_123'
print(e is f) # True, потому что строка содержит только цифры и символ '_'
```

61. Как упаковать бинарные зависимости?

Для упаковки бинарных зависимостей в проект следует использовать менеджеры пакетов. Для Python наиболее распространены `pip` и `conda`.

Пример для Python с использованием `pip`:

- Установите необходимые библиотеки и зависимости в проекте:
  `pip install requests numpy pandas`
- Создайте файл requirements.txt с полным списком зависимостей:

```
requests
numpy
pandas
```

- Упакуйте зависимости в архив:
  `pip freeze > requirements.txt`
- Можно передать файл `requirements.txt` другим пользователям вашего проекта, которые могут установить все зависимости одной командой:
  `pip install -r requirements.txt`

Для упаковки бинарных зависимостей можно использовать инструмент `wheel`. Wheel-файлы - это zip-архивы, содержащие установочные файлы для Python-пакетов, и могут содержать бинарные расширения (например, скомпилированные модули C), которые необходимо собрать и установить на целевой машине.

Для создания wheel-файла для Python-пакета можно использовать команду `pip wheel`. Например, если есть файл с требованиями `requirements.txt`, содержащий список зависимостей вашего проекта, можете создать wheel-файлы для всех зависимостей с помощью следующей команды:
`pip wheel -r requirements.txt`
Вы также можете установить wheel-файлы с помощью `pip install`, указав имя файла:
`pip install mypackage-1.0.0-py3-none-any.whl`

Таким образом, вы можете создавать и распространять бинарные зависимости в виде wheel-файлов и использовать их при установке пакетов на других устройствах.

62. Почему в python нет оптимизации хвостовой рекурсии? Как это реализовать?

В Python хвостовая рекурсия не оптимизируется автоматически, поскольку она может привести к переполнению стека вызовов. В связи с этим, используется итеративный подход для написания функций, которые могут быть написаны с использованием хвостовой рекурсии в других языках.

Вы можете использовать декоратор `sys.setrecursionlimit()` для установки максимальной глубины стека вызовов. Однако это не рекомендуется, поскольку установка слишком большого лимита может привести к проблемам с производительностью, а слишком маленький лимит - к ошибкам переполнения стека вызовов.

Вот пример того, как можно установить максимальную глубину стека вызовов до 4000:

```
import sys
sys.setrecursionlimit(4000)
```

Вы также можете изменить код функции, чтобы использовать итеративный подход вместо хвостовой рекурсии. Один пример такого изменения может выглядеть следующим образом:

```
def factorial(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

это вместо использования рекурсивного подхода с вызовом `factorial(n-1)` внутри функции `factorial(n)` .

Изменение рекурсивно написанной функции на итеративный код не всегда легко, но может существенно повысить производительность и устранить проблемы с переполнением стека вызовов.

63. Что такое wheels и eggs? В чём разница?

В Python `wheels` и `eggs` - это форматы пакетов для установки и дистрибуции пакетов с помощью утилиты управления пакетами `pip` .

`Eggs` был первоначально разработан как формат дистрибуции пакетов для Python, но был заменен `wheels` . В отличие от `wheels` , `eggs` могут содержать `.pyc` файлы, что может привести к проблемам при установке на другой платформе или версии Python.

`Wheels` - это новый формат дистрибуции пакетов, который был введен в Python 2.7. Он поддерживается большинством пакетов на PyPI и имеет множество преимуществ, например:

- Он не содержит `.pyc` файлов, что снижает вероятность конфликтов.
- Он легко переносится между платформами и версиями Python.
- Он поддерживает сжатие библиотек и упрощает установку требований.

В целом, wheels считается более продвинутой и предпочтительной формой дистрибуции пакетов в Python.

Вообще, инфраструктура системы пакетов Python — довольно холиварная тема, подробнее тут: Python на колёсах

64. Как получить доступ к модулю, написанному на python из C и наоборот?

Для того чтобы получить доступ к модулю, написанному на Python из C, можно использовать библиотеку Python/C API, которая позволяет вызывать Python функции и работать с объектами Python из C программы. Для того чтобы получить доступ к модулю, сначала нужно получить указатель на объект модуля с помощью функции `PyImport_ImportModule()` . Затем можно получить указатель на функции или объекты модуля с помощью функции `PyObject_GetAttrString()` .

Например, вот пример кода на C, который вызывает функцию "hello" из модуля "example" на Python:

```
#include <Python.h>
int main() {
    Py_Initialize();
    PyObject* module = PyImport_ImportModule("example");
    PyObject* func = PyObject_GetAttrString(module, "hello");
```

```
    PyObject* result = PyObject_CallObject(func, NULL);
    printf("Result: %s\n", PyUnicode_AsUTF8(result));
    Py_DECREF(func);
    Py_DECREF(module);
    Py_DECREF(result);
    Py_Finalize();
    return 0;
}
```

Аналогичным образом можно вызвать функции из библиотек, написанных на C из Python, используя библиотеку `ctypes`. Например, вот пример кода на Python, который вызывает функцию `sqrt` из библиотеки `math`:

```
from ctypes import cdll
libm = cdll.LoadLibrary('libm.so')
print(libm.sqrt(4.0))
```

Здесь мы загружаем библиотеку `libm.so` (которая содержит функцию `sqrt`) и вызываем её с помощью атрибута `dot-notation`

65. Как ускорить существующий код python?

img

Чтобы ускорить существующий код на Python, можно использовать несколько подходов:

- **Векторизация**: векторизация позволяет оптимизировать код, который выполняет большое количество операций над массивами данных, например, использование библиотеки NumPy.
- **Выбор правильных структур данных**: выбор правильных структур данных и алгоритмов может значительно ускорить выполнение кода. Например, использование словарей может быть более эффективным, чем использование списков.
- **Компиляция**: компиляция Python-кода в байт-код или в машинный код может ускорить выполнение кода. Для этого можно использовать Cython, Nuitka или PyPy.
- **Многопоточность**: использование многопоточности может ускорить выполнение задач, которые можно разделить на несколько независимых частей.
- **Параллелизм**: параллельное выполнение задач на нескольких ядрах процессора может ускорить выполнение кода.
- **Оптимизация**: такие инструменты, как cProfile и line_profiler, могут помочь оптимизировать код, выявляя узкие места в его выполнении и предоставляя информацию о времени выполнения каждой строки кода.

Компромиссы: если выполнение кода нельзя ускорить до приемлемого уровня, можно рассмотреть возможность использования компромиссов, например, уменьшить
количество данных, обрабатываемых кодом, или упростить логику выполнения задачи.

66. Что такое pycache? Что такое файлы .pyc?

В Python, когда вы запускаете программу, интерпретатор сначала компилирует ее в байт-код и сохраняет в папке `pycache` Это делается для того, чтобы в следующий раз выполнить программу быстрее, поскольку байт-код можно напрямую загрузить в память, а не приходится компилировать заново. Файлы байт-кода имеют расширение `.pyc` и обычно хранятся в подкаталоге каталога, содержащего соответствующие файлы `.py` .

Каталог pycache автоматически создается интерпретатором Python и используется для хранения скомпилированных файлов байт-кода. Каталог содержит скомпилированные версии импортированных сценариев Python, а также любые модули, импортированные этими сценариями. Этот каталог обычно находится в том же каталоге, что и файлы `.py` , но может также находиться во временном каталоге системы, если исходный каталог доступен только для чтения. Как правило, вам не нужно напрямую взаимодействовать с каталогом pycache или файлами `.pyc` в нем, поскольку они автоматически управляются интерпретатором Python. Однако вы можете удалить файлы `.pyc` , если хотите заставить интерпретатор перекомпилировать соответствующие скрипты Python.

Файлы `.pyc` - это скомпилированные байт-коды Python, которые создаются при импорте модулей. Когда вы импортируете модуль в Python, интерпретатор компилирует его и создает файл `.pyc` , который содержит байт-коды для модуля. Этот файл будет использоваться для ускорения повторных импортов модуля, так как он может быть загружен вместо повторной компиляции каждый раз.

Кроме того, файлы `.pyc` также могут использоваться для распространения скомпилированных версий модулей или приложений. Они представляют собой скомпилированные версии исходных файлов Python, которые можно предоставить пользователям без необходимости предоставления исходного кода.

Важно отметить, что файлы `.pyc` являются специфичными для версии Python, так что файлы, созданные для одной версии Python, не будут работать с другой версией.

67. Что такое виртуальное окружение?

Виртуальное окружение - это механизм, который позволяет создавать изолированные окружения для установки и использования пакетов Python. Это полезно, когда вам нужно установить определенную версию пакета или когда вам нужно иметь одновременный доступ к разным версиям библиотек в зависимости от проекта.

Создание виртуального окружения позволяет изолировать зависимости проекта от системных зависимостей и других проектов, работающих на той же машине. Это помогает избежать конфликтов зависимостей, что может привести к ошибкам и сбоям. Вы можете создать виртуальное окружение Python с помощью модуля `venv` , который поставляется в стандартной библиотеке Python.

Например, вы можете создать виртуальное окружение в текущей директории, выполнив следующую команду в терминале:
`python3 -m venv myenv`
где `myenv` - имя виртуального окружения.

После создания виртуального окружения вы можете активировать его, выполнив команду (для Unix-системы):
`source myenv/bin/activate`
или (для Windows):
`myenv\Scripts\activate`

После активации виртуального окружения вы можете устанавливать и использовать пакеты Python без влияния на глобальное окружение вашего компьютера.

68. Python — это императивный или декларативный язык?

Python является императивным языком программирования. В императивном программировании программист составляет последовательность команд, которые выполняются компьютером. Python также поддерживает некоторые функциональные и объектно-ориентированные концепции программирования, однако основной подход в языке является императивный.

"Императивный язык" это термин, который относится к классу языков программирования, использующих прямые команды для управления компьютером, в отличие от
декларативных языков. В императивных языках программист явно описывает действия, которые нужно выполнить компьютеру, а не просто описывает желаемый результат. Примеры императивных языков программирования это Java, C, C++, Python и JavaScript.

Декларативный язык - это язык программирования, который назначает техническую реализацию системы или программы для достижения определенной цели, но не указывает конкретных шагов для ее выполнения. Вместо этого вы определяете, какая информация должна быть обработана, а система сама определяет, как решить эту проблему. Примерами декларативных языков являются SQL для работы с базами данных и HTML для создания веб-страниц. Такие языки обычно используются в случаях, когда важнее задать желаемый результат, чем указать, как добиться этого результата.

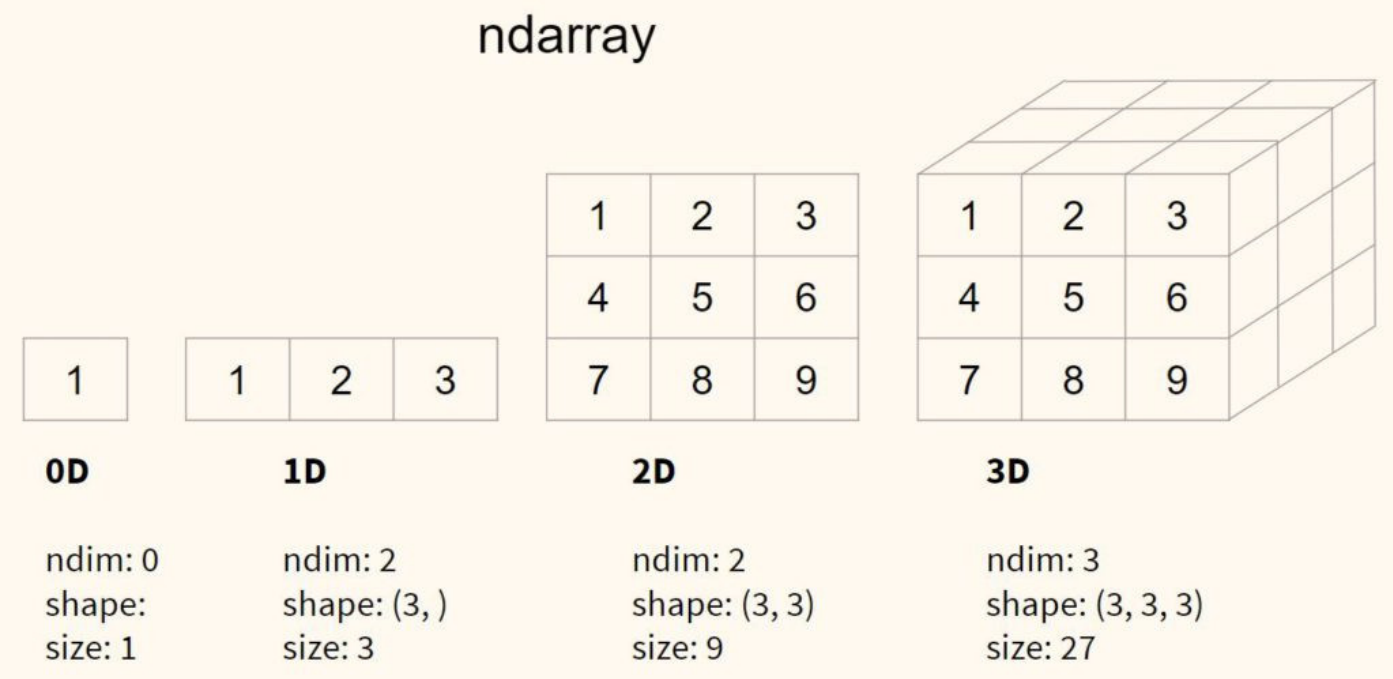
69. Что такое менеджер пакетов? Какие менеджеры пакетов вы знаете?

Менеджер пакетов - это инструмент, который позволяет управлять установкой, обновлением и удалением библиотек и зависимостей в проектах на языке Python.

Встречайте, вот наиболее популярные менеджеры пакетов Python:

- `pip` - это стандартный менеджер пакетов Python. Он позволяет устанавливать пакеты из Python Package Index (PyPI) и других источников, а также управлять зависимостями проекта.
- `conda` - это менеджер пакетов и среда управления, который позволяет управлять пакетами и зависимостями для проектов на Python, а также для других языков программирования и платформ.
- `easy_install` - инструмент для установки и управления пакетами Python, который был стандартным до выпуска Python 3. Используется редко в настоящее время.
- `poetry` - новый менеджер пакетов, предназначенный для замены в некоторой степени pip и virtualenv.

70. В чём преимущества массивов numpy по сравнению с (вложенными) списками python?



Основное преимущество массивов NumPy перед списками Python заключается в том, что NumPy использует более оптимизированную память и имеет более эффективные методы работы с массивами (из-за реализации на C), что делает его подходящим выбором для работы с большими объемами данных и научных вычислений.

Например, с NumPy вы можете выполнять бродкастинг (broadcasting), матричные операции и другие векторизованные вычисления с более высокой производительностью, чем при использовании вложенных списков.

Некоторые из основных преимуществ NumPy:

- Более оптимизированная память, что позволяет NumPy работать быстрее с большим объемом данных
- Встроенные методы для выполнения арифметических операций, таких как сумма и произведение, которые могут работать сразу над всеми элементами массивов.
- Возможность выполнять матричные операции и другие векторизованные вычисления.
- Простой синтаксис для выполнения операций над массивами.
- Возможность конвертировать массивы NumPy в другие формы данных, такие как списки Python или таблицы Pandas.

Если вы работаете с массивами данных, над которыми нужно выполнять научные вычисления, то использование NumPy будет более предпочтительным вариантом, чем использование списков Python.

71. Вам нужно реализовать функцию, которая должна использовать статическую переменную. Вы не можете писать код вне функции и у вас нет информации о внешних переменных (вне вашей функции). Как это сделать?

Вам нужно использовать замыкание. Замыкание - это функция, которая сохраняет ссылку на переменные из своей внешней области видимости, даже когда эта область видимости больше не существует. Это позволяет функции работать с переменной, которая является статической, даже если она была определена вне функции.

Вот пример использования замыкания для создания функции, которая использует статическую переменную:

```
def my_function():
    static_var = 0
    def inner_function():
        nonlocal static_var
        static_var += 1
        return static_var
    return inner_function

# создаем объект функции, который использует статическую переменную
f = my_function()

# вызываем функцию несколько раз, чтобы увидеть изменение значения статической переменной
print(f()) # выводит 1
print(f()) # выводит 2
print(f()) # выводит 3
```

Этот код определяет функцию `my_function`, которая содержит внутри себя функцию `inner_function`, которая использует статическую переменную `static_var`. Каждый раз, когда `inner_function` вызывается через `f()`, значение `static_var` увеличивается на единицу и возвращается новое значение. Таким образом, каждый вызов `f()` возвращает увеличенное значение статической переменной.

Важно, чтобы вы использовали ключевое слово `nonlocal`, чтобы объявить `static_var` как статическую переменную внутри `inner_function`, иначе Python будет считать ее локальной переменной и создает новую переменную каждый раз, когда `inner_function` вызывается.

72. Что будет напечатано в результате выполнения следующего кода?

```
def f_g():
    yield 43
    return 66

print(f_g())
```

Результат выполнения кода будет объект генератора (`generator object`). Когда мы вызываем функцию с `yield`, то это создает генератор, который возвращает объект-итератор. Так как `print(f_g())` вызывает только генератор, а не запускает его выполнение, то мы получим объект-итератор в качестве результата, а не значение, возвращенное посредством `yield` или `return`.

Если мы хотим получить значение из генератора, мы должны использовать ключевое слово `next`, чтобы продвинуть генератор на следующее значение или использовать цикл `for` для извлечения всех значений из итератора. Вот пример вызова генератора с помощью цикла `for`:

```
def f_g():
    yield 43
    return 66

for i in f_g():
    print(i)
```

Этот код выведет только 43, потому что выполнение генератора останавливается после первого вызова `yield`.

73. Как имплементировать словарь с нуля?

Для реализации словаря можно использовать хэш-таблицу. Хэш-таблица - это структура данных, которая обеспечивает эффективный поиск, вставку и удаление элементов. Ключи преобразуются в индексы с помощью хэш-функции, и значения хранятся в соответствующих ячейках памяти.
Например, можно создать класс, который будет имитировать словарь:

```
class MyDictionary:
    def __init__(self):
        self.size = 10 # размер таблицы
        self.keys = [None] * self.size
        self.values = [None] * self.size

    def __setitem__(self, key, value):
        index = hash(key) % self.size # вычисляем индекс
        self.keys[index] = key
        self.values[index] = value

    def __getitem__(self, key):
        index = hash(key) % self.size
        return self.values[index]
```

Теперь можно создавать экземпляры этого класса и использовать их, как обычный словарь:

```
d = MyDictionary()
d['apple'] = 'red'
d['banana'] = 'yellow'
print(d['apple']) # выведет 'red'
print(d['banana']) # выведет 'yellow'
```

Это простой пример, и на практике словари в Python имеют более сложную реализацию, чтобы обеспечить высокую производительность и эффективность использования памяти.

74. Напишите однострочник, который будет подсчитывать количество заглавных букв в файле.

Для подсчета количества заглавных букв в файле можно использовать следующий однострочник:

```
num_uppercase = sum(1 for line in open('filename.txt') for character in line if character.isupper())
```

В этом однострочнике мы открываем файл `'filename.txt'` и пробегаемся по всем его строкам и символам в каждой строке. Для каждого символа, который является заглавной буквой метод `isupper()` возвращает `True`, и мы добавляем 1 к счетчику с помощью функции `sum()`. В конце, `num_uppercase` будет содержать количество заглавных букв в файле.

Однострочники — это вообще очень приятная питонячая радость, есть даже целые книги (рекомендую, легко гуглится `Кристиан Майер однострочники в Python filetype:pdf`).

75. Что такое файлы .pth?

Файлы с расширением `.pth` - это файлы, которые могут быть использованы для добавления директорий в путь поиска модулей Python. Директивы `.pth` выполняются при запуске интерпретатора Python и добавляют определенные каталоги в переменную `sys.path`. Это удобно, когда нужно импортировать модули из нестандартных директорий без необходимости переноса файлов в директории по умолчанию. Использование директив `.pth` достаточно распространено в мире Python и они встречаются в различных средах разработки и фреймворках, таких как PyTorch.

Файлы `.pth` могут быть также использованы злоумышленниками для внедрения вредоносного кода в систему Python, так как они могут изменять список каталогов, в которых выполняется поиск модулей Python. Поэтому необходимо быть внимательными при работе с такими файлами и использовать только те файлы `.pth`, которые вы знаете и доверяете.

76. Какие функции из collections и itertools вы используете?

В модулях `collections` и `itertools` в Python есть множество полезных функций, которые могут использоваться в различных задачах. Некоторые из наиболее часто используемых функций включают:

`defaultdict`: это удобный способ создания словаря с заданным значением по умолчанию для любого ключа, который еще не был добавлен в словарь.

`Counter`: это удобный способ подсчета количества встречаемых элементов в списке или другом итерируемом объекте. Он возвращает объект, который можно использовать как словарь, где ключами являются элементы, а значения - количество их вхождений.

`namedtuple`: можно создать именованный кортеж с заданными полями, что может быть удобно для работы с данными, которые имеют структуру, но не требуют создания класса.

`itertools.chain`: позволяет конкатенировать несколько итерируемых объектов в единый итератор.

`itertools.groupby`: позволяет группировать элементы итерируемого объекта по заданному ключу.

`itertools.combinations` и `itertools.permutations`: генерируют все различные комбинации или перестановки элементов из заданного множества.

```
from collections import defaultdict
d = defaultdict(int)
print(d['apple'])

d = defaultdict(list)
print(d['apple'])

d = defaultdict(set)
print(d['apple'])

# вывод:
# 0
# []
# set()
```

```
from collections import Counter
cnt = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
print(cnt)
print(dict(cnt))

# вывод:
# Counter({'blue': 3, 'red': 2, 'green': 1})
# {'red': 2, 'blue': 3, 'green': 1}
```

```
from collections import namedtuple

Point = namedtuple("Point", "x y")
print(issubclass(Point, tuple))

point = Point(2, 4)
print(point)

print(point.x)
print(point.y)
```

```
print(point[0])
print(point[1])

# вывод:
# True
# Point(x=2, y=4)
# 2
# 4
# 2
# 4
```

```
from itertools import chain

chained = chain('ab', [33])
print(next(chained))
print(next(chained))
print(next(chained))
# вывод:
# a
# b
# 33

for i in chain('1', [1, 2, 3], {3, 4}):
    print(i)
# вывод:
# 1
# 1
# 2
# 3
# 3
# 4

for i in chain('1', [1, {2, (5, '6')}, 3], {3, 4}):
    print(i)
# вывод:
# 1
# 1
# {2, (5, '6')}
# 3
# 3
# 4
```

```
from itertools import groupby

grouper = lambda item: item['country']

data = [
    {'city': 'Москва', 'country': 'Россия'},
    {'city': 'Новосибирск', 'country': 'Россия'},
    {'city': 'Пекин', 'country': 'Китай'},
]

for key, group in groupby(data, key=grouper):
    print(key, *group)

# Россия {'city': 'Москва', 'country': 'Россия'} {'city': 'Новосибирск', 'country': 'Россия'}
# Китай {'city': 'Пекин', 'country': 'Китай'}
```

```
from itertools import combinations

print(list(combinations('123', 2)))
```

```
# [('1', '2'), ('1', '3'), ('2', '3')]
```

```
from itertools import permutations

print(list(permutations('123', 2)))

# [('1', '2'), ('1', '3'), ('2', '1'), ('2', '3'), ('3', '1'), ('3', '2')]
```

77. Что делает флаг PYTHONOPTIMIZE?

Флаг `-O` или `PYTHONOPTIMIZE` в Python используется для оптимизации скомпилированного кода, что может привести к ускорению выполнения программы. Этот флаг удаляет отладочную информацию, отключает `asset checks`, `asserts` и отладочные проверки.

Стандартная оптимизация `-O` удаляет `docstrings` из скомпилированного `byte-code`, а также удаляет `assert statements`. С флагом `-OO` удаляются все `docstrings` в модулю (включая те, которые не соответствуют многострочным строкам) и также удаляются `assert statements`.

Запуск интерпретатора Python с флагом `-O` может уменьшить размер скомпилированного кода и сократить потребление памяти, что может привести к ускорению работы программы. Однако, для большинства приложений, эта оптимизация может не иметь значимого влияния на производительность.

Например, для запуска скрипта с флагом `-O`, можно использовать следующую команду в командной строке:
`python -O my_script.py`

78. Какие переменные среды, влияющие на поведение интерпретатора python, вы знаете?

Несколько известных переменных среды, влияющих на поведение интерпретатора Python:

`PYTHONPATH` - определяет список каталогов, в которых интерпретатор Python будет искать модули.

`PYTHONDONTWRITEBYTECODE` - если установлено в любое ненулевое значение, интерпретатор Python не будет создавать файлы .pyc для скомпилированного байт-кода.

`PYTHONSTARTUP` - определяет путь к файлу, который содержит инициализационный код Python, он выполняется в начале каждой сессии интерпретатора.

`PYTHONIOENCODING` - задает кодировку, которую интерпретатор Python должен использовать для обработки ввода / вывода.

`PYTHONLEGACYWINDOWSSTDIO` - если установлено в любое ненулевое значение, указывает интерпретатору Python использовать режим Windows для ввода-вывода вместо UNIX-стиля.

В зависимости от операционной системы, может быть и другие переменные среды, которые влияют на поведение интерпретатора Python. Чтобы увидеть все переменные среды, которые влияют на вашу систему, вы можете использовать команду `"env"` в терминале, если вы используете UNIX-подобную систему, или команду `"set"` в командной строке Windows.

Эти альтернативные реализации продолжают существовать, поскольку каждая из них предлагает уникальные функции и преимущества по сравнению со стандартной реализацией Python (CPython). Например, Cython может обеспечить значительное повышение производительности по сравнению со стандартным кодом Python, а IronPython позволяет коду Python легко взаимодействовать с другими приложениями .NET. PyPy также может обеспечить значительное повышение производительности по сравнению со стандартным кодом Python, особенно при работе с задачами, требующими большого количества вычислений. В целом эти альтернативные реализации Python расширяют функциональные возможности языка и предоставляют больше возможностей разработчикам, решившим использовать Python в своих проектах.

79. Что такое Cython? Что такое IronPython? Что такое PyPy? Почему они до сих пор существуют и зачем?

Cython - это язык программирования, нацеленный на увеличение производительности Python-кода. Cython позволяет использовать возможности языка Python и C/C++ для эффективного написания расширений модулей на языке Python. Он позволяет вам писать код на

Python, который доступен из C/C++, и наоборот. Cython обеспечивает скорость выполнения, сравнимую со скоростью выполнения на языке C/C++, при этом сохраняя простоту и удобство использования языка Python. Cython compiler компилирует исходный код в C/C++ и затем переводит его в машинный код, что дает быстрый доступ к низкоуровневым ресурсам операционной системы, таким как память и ввод-вывод. Cython также предоставляет возможность использовать дополнительные функции, такие как статическая типизация и параллельное программирование, для дополнительного увеличения производительности.

IronPython - это реализация языка программирования Python, которая работает в контексте платформы .NET. IronPython предоставляет возможность использовать Python в качестве языка .NET. Он может использоваться для написания .NET-приложений, а также для расширения приложений, написанных на других языках .NET. IronPython является открытым и свободно распространяемым программным обеспечением.

PyPy — это высокопроизводительная реализация языка программирования Python. Он был создан с целью предоставления более быстрой и эффективной альтернативы стандартному интерпретатору CPython. PyPy включает компилятор Just-In-Time (JIT), который может оптимизировать выполнение кода Python во время выполнения, что может привести к значительному повышению производительности по сравнению с CPython, особенно для определенных типов рабочих нагрузок. PyPy также поддерживает многие из тех же функций и модулей, что и CPython, включая объектно-ориентированное программирование, динамическую типизацию и стандартную библиотеку Python.

80. Как перевернуть генератор?

Можно перевернуть генератор в Python, используя функцию `reversed()`. Вот пример, который демонстрирует это:

```
my_list = [1, 2, 3, 4, 5]
my_generator = (x**2 for x in my_list)

for item in reversed(list(my_generator)):
    print(item)
```

В этом примере мы используем функцию `reversed()` вместе с функцией `list()`, чтобы создать обратный список элементов, сгенерированных генератором. Затем мы используем этот список с циклом `for` для перебора элементов в обратном порядке. Если вы работаете с большими наборами данных, может быть полезно использовать обратное итерирование без использования `list()`, чтобы избежать создания полной копии.

Вот пример, который демонстрирует это:

```
my_list = [1, 2, 3, 4, 5]
my_generator = (x**2 for x in my_list)

for item in reversed(tuple(my_generator)):
    print(item)
```

Здесь мы используем функцию `reversed()` вместе с функцией `tuple()` для обратного итерирования через генератор без создания полной копии.

81. Приведите пример использования filter и reduce над итерируемым объектом.

Пример использования `filter()` и `reduce()` над итерируемым объектом в Python:

```
from functools import reduce

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Пример использования filter() для отфильтровывания четных чисел
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers) # выводит [2, 4, 6, 8, 10]

# Пример использования reduce() для нахождения суммы чисел от 1 до 10
```

```
sum_of_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_of_numbers) # выводит 55
```

В этом примере мы использовали `filter()` для отбора только четных чисел в списке `numbers`, и `reduce()` для нахождения суммы всех чисел в списке от 1 до 10.

`filter()` принимает два аргумента - функцию-предикат и итерируемый объект. Он возвращает новый итератор, содержащий только те элементы итерируемого объекта,
которые удовлетворяют условиям, заданным функцией-предикатом.

`reduce()` также принимает два аргумента - функцию и итерируемый объект. Он выполняет функцию на каждой паре элементов из итерируемого объекта, образуя
редуцированное значение, которое в конечном итоге становится результатом функции.

В примере мы использовали `reduce()` для нахождения суммы всех чисел в
итерируемом объекте.

82. Чем фреймворк отличается от библиотеки?

Фреймворк и библиотека - это два разных подхода к организации кода, которые используются для упрощения разработки программного обеспечения.

Библиотека представляет собой коллекцию модулей или функций, предназначенных для использования другими приложениями. Она содержит набор готовых решений для различных задач и обеспечивает функциональность, которую можно использовать в своём приложении. Пользователь сам выбирает, какие модули или функции использовать, и какую логику реализовывать самостоятельно.

Фреймворк представляет собой интегрированный набор компонентов и инструментов, который предоставляет готовое решение для решения определенной задачи. Его основная цель - упростить разработку приложений, обеспечивая заранее заданную структуру и логику работы. В отличие от библиотеки, фреймворк накладывает определенные ограничения на структуру, логику и процесс разработки приложения, но при этом предоставляет готовый инструментарий для работы.

В целом, библиотека дает большую свободу в выборе логики и реализации приложения, но требует больше написания кода. Фреймворк же облегчает начало разработки и создает более унифицированный код, но может ограничивать возможности программиста по изменению поведения и структуры приложения.

83. Расположите функции в порядке эффективности, объясните выбор.

```
def f1(arr):
    l1 = sorted(arr)
    l2 = [i for i in l1 if i < .5]
    return [i * i for i in l2]

def f2(arr):
    l1 = [i for i in arr if i < .5]
    l2 = sorted(l1)
    return [i * i for i in l2]

def f3(arr):
    l1 = [i * i for i in arr]
    l2 = sorted(l1)
    return [i for i in l1 if i < (.5 * .5)]
```

Наиболее эффективной функцией из трех предоставленных, вероятно, будет `f2`. Это связано с тем, что он избегает сортировки всего списка, вместо этого сортируется только меньший предварительно отфильтрованный список. Вот почему.

`f1` сортирует весь список с помощью функции `sorted`, которая имеет временную сложность `O(n log n)`, где `n` — длина входного списка. После сортировки он отфильтровывает все элементы, большие или равные `0.5`, и вычисляет квадраты оставшихся элементов.

Фильтрация списка занимает время `O(n)`, а окончательное вычисление занимает время `O(m)`, где `m` — длина отфильтрованного списка. Следовательно, общая временная сложность этой функции равна `O(n log n + n + m)`.

`f2` сначала фильтрует входной список, чтобы включить только элементы меньше `0.5`, что занимает `O(n)` времени. Затем он сортирует этот отфильтрованный список с помощью функции `sorted`, которая имеет временную сложность `O(m log m)`, где `m` — длина отфильтрованного списка. Наконец, он вычисляет квадраты отсортированных элементов. Вычисление квадратов занимает `O(m)` времени. Поэтому, общая временная сложность этой функции составляет `O(n + m log m + m)`

`f3` вычисляет квадраты всех элементов во входном списке, что занимает `O(n)` времени. Затем он сортирует список в квадрате с помощью функции `sorted`, которая
имеет временную сложность `O(n log n)`. Наконец, он отфильтровывает все элементы, большие или равные `0.25`, что занимает время `O(n)`. Таким образом, общая
временная сложность этой функции равна `O(n log n)`.

Таким образом, `f2` имеет наилучшую временную сложность, поскольку сортирует наименьший список, который является только отфильтрованным.

Дисклеймер: это может быть несущественным в небольших списках, и всегда ключевым фактором является бенчмаркинг.

```
def f1(arr):
    l1 = sorted(arr)
    l2 = [i for i in l1 if i < .5]
    return [i * i for i in l2]

def f2(arr):
    l1 = [i for i in arr if i < .5]
    l2 = sorted(l1)
    return [i * i for i in l2]

def f3(arr):
    l1 = [i * i for i in arr]
    l2 = sorted(l1)
    return [i for i in l1 if i < (.5 * .5)]

# в Jupyter можно замерить время при помощи %time
arr = list(range(10**7))
%time a = f1(arr)
%time b = f2(arr)
%time c = f3(arr)
```

84. Произошла утечка памяти в рабочем приложении. Как бы вы начали отладку?

Для отладки утечек памяти в Python можно использовать инструменты, такие как `Memory Profiler` или `objgraph`. Вот пример использования `Memory Profiler` для отслеживания утечек памяти:

- Установите `Memory Profiler` с помощью `pip`: `pip install memory-profiler`
- Используйте декоратор `@profile` перед функцией, которая может вызывать утечки памяти.

```
from memory_profiler import profile
@profile
def my_func():
    # Some code that may cause a memory leak
```

Запустите вашу программу с помощью команды `python -m memory_profiler my_script.py`. Будет выведен подробный отчет о том, сколько памяти используется в каждой строке программы, а также общее использование памяти и любые утечки.
Также можно использовать `objgraph` для визуализации объектов, которые находятся в оперативной памяти и могут вызывать утечки. Вот пример:

```
import objgraph
my_list = [1, 2, 3]
objgraph.show_refs([my_list], filename='my_list.png')
```

Этот код создаст изображение `my_list.png`, на котором будут показаны все объекты, на которые ссылается `my_list`, а также все объекты, которые ссылается на них. Это может помочь вам понять, какие объекты держат ссылки на ваши объекты и могут вызывать утечки памяти.

85. В каких ситуациях возникает исключение NotImplementedError?

Исключение `NotImplementedError` возникает, когда метод или функция должны быть реализованы в подклассе, но не были реализованы. Это может произойти, когда родительский класс определяет метод, но не реализует его сам, а оставляет это для подклассов. В этом случае, если подкласс не реализует метод, он будет вызывать исключение `NotImplementedError`. Это может быть полезно для отладки, чтобы убедиться, что все необходимые методы реализованы в подклассах. Это также может возникнуть в других ситуациях, например, если вы пытаетесь использовать неопределенную функцию или метод.

86. Что не так с этим кодом? Зачем это нужно?

```
if __debug__:
    assert False, ("error")
```

Этот код вызывает ошибку утверждения `assert` с сообщением `"error"`, если `__debug__` равен `True`.
`__debug__` - это встроенная переменная Python, которая является истинной, если к интерактивной консоли или скрипту был присоединен флаг оптимизации `-O`. Для типичных скриптов в режиме отладки эта переменная равна `True`. Если оптимизация включена, то интерпретатор Python игнорирует все операторы утверждения `assert`, поэтому этот код не вызовет ошибку в optimized mode.
Такой код может быть использован для проверки инвариантов в программе или для отладки кода. Если утверждение не выполняется и вызывается `AssertionError`, это означает, что в программе произошло что-то непредвиденное, что нарушило заданное утверждение, и программа остановится с сообщением об ошибке.

87. Что такое магические методы (dunder-методы)?

Магические методы, также известные как "dunder" (double underscore) методы в Python, это специальные методы, которые начинаются и заканчиваются двойным подчеркиванием. Они позволяют определить, как объекты этого класса будут вести себя в различных контекстах, например, при использовании операторов Python, таких как `+, -, *, /` и т.д., при вызове функций и методов, при сериализации и многое другое.

Некоторые примеры магических методов в Python включают:

- `init` : инициализирует новый экземпляр объекта
- `str` : определяет, как объект будет представлен в строковом формате
- `add` : определяет, что происходит при использовании оператора `+`
- `len` : определяет, как объект будет представлен при вызове функции `len()`
- `getitem` : позволяет получать доступ к элементам объекта, как к элементам списка

Магические методы могут быть очень полезными при создании пользовательских классов в Python, так как они позволяют управлять поведением объектов в различных контекстах и создавать более понятный и гибкий код.

```
# иллюстрация "утиной типизации": мы можем измерять длину у всего, где есть метод len()

class MyClass:
    def __len__(self):
        print('длина - это смотря как посмотреть')
        return 0
```

```
myclass = MyClass()
print(len(myclass))
```

88. Что такое monkey patching? Приведите пример использования.

Monkey patching - это техника изменения поведения кода во время выполнения путем динамической замены или добавления методов или атрибутов в существующем объекте. По сути, мы просто переопределяем метод за пределами объявления класса. Это может быть полезно, когда изменения не могут быть внесены в существующий код, и требует минимальных изменений в существующем коде.

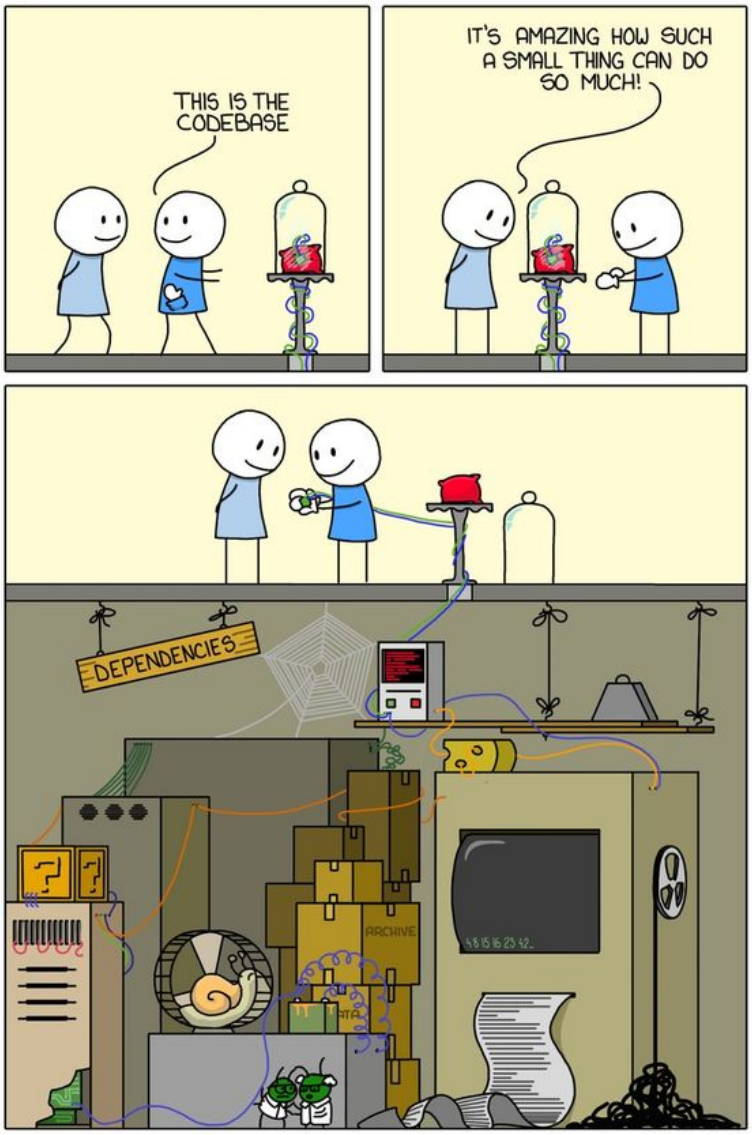
Например, можно добавить новый метод в класс в runtime, который наследуется от базового класса:

```
class MyBaseClass:
    def my_method(self):
        print('Hello from MyBaseClass')

def monkey_patch():
    def new_method(self):
        print('Hello from new_method')
    MyBaseClass.my_method = new_method

monkey_patch()
obj = MyBaseClass()
obj.my_method()  # выведет "Hello from new_method"
```

В этом примере мы добавляем новый метод `new_method()` в класс `MyBaseClass` , используя функцию `monkey_patch()` . После этого, вызов метода `obj.my_method()` выведет строку `'Hello from new_method'`

Важно учитывать, что использование monkey patching может усложнить отладку и поддержку в будущем, поэтому следует использовать эту технику с осторожностью и только при необходимости.

89. Как работать с транзитивными зависимостями?



img

Для работы с транзитивными зависимостями можно использовать систему управления зависимостями, например, `pipenv`, `poetry` или `pip`. Эти системы позволяют устанавливать зависимости и их транзитивные зависимости, а также контролировать версии зависимостей. Например, при использовании `pipenv` для установки и работы с зависимостями можно использовать следующие команды:

`pipenv install <имя пакета>`.
Эта команда установит пакет и его транзитивные зависимости и создаст файл Pipfile с перечнем зависимостей и версиями.

`pipenv shell`.
Эта команда позволит активировать виртуальное окружение, в котором установлены зависимости.

`pipenv install --dev <имя пакета>`.
Эта команда установит пакет в качестве зависимости разработки.

`pipenv uninstall <имя пакета>`.
Эта команда удалит пакет и его транзитивные зависимости.

Также можно использовать файлы `requirements.txt` или `setup.py` для установки зависимостей и их транзитивных зависимостей, сорри за тавтологию.

90. Когда использование Python является «правильным выбором» для проекта?

Использование Python может быть правильным выбором для проекта в следующих случаях:

- Когда нужен быстрый прототип или быстрое решение, которое будет работать достаточно быстро без оптимизации производительности.
- Когда нужен простой и понятный синтаксис языка программирования, который позволит быстрее писать код и делать его более читабельным.
- Когда нужен доступ к большому количеству сторонних библиотек и фреймворков в области машинного обучения, науки о данных, веб-разработки и многих других областях.
- Когда необходимо использование «кляузы batteries included», определяющей ысокоуровневый язык программирования с широким спектром интегрированных библиотек и модулей.

Однако Python может не быть оптимальным выбором для тех приложений, где требуется высокая производительность или многоуровневая безопасность. В этих случаях может быть предпочтительнее использование языков, таких как C++, Java, или C#. Ну или Rust, не зря же он у нас самый любимый язык).

91. Что такое метод?

Методы в Python - это функции, определенные внутри класса, которые могут быть вызваны на экземпляре этого класса или на самом классе. Методы предоставляют способ для объектов класса взаимодействовать с данными, хранящимися внутри объекта, а также для выполнения действий, которые связаны с этими данными.
Например, если у вас есть класс `Person` с атрибутами `name` и `age`, атрибут `name` будет хранить имя объекта `Person`, а атрибут `age` будет хранить возраст. Вы можете определить методы, такие как `get_name` и `get_age`, которые могут быть вызваны на экземпляре класса для получения значения хранящихся в атрибутах `name` и `age` соответственно.

Вот пример определения класса Person с методами `get_name` и `get_age`:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def get_name(self):
        return self.name

    def get_age(self):
        return self.age
```

Здесь метод `init` - это конструктор класса, который инициализирует атрибуты `name` и `age`, а методы `get_name` и `get_age` предоставляют доступ к их значениям.

Кстати, `__init__` — это один из дандер-методов, о том, как работать с ними неплохо описано тут: Руководство по магическим методам в Питоне

92. Есть ли в Python оператор switch-case?

В Python нет прямого оператора `switch-case`, как в других языках программирования. Однако, начиная с версии Python 3.10, появилась возможность использовать оператор `match-case`, который является аналогом `switch-case` в других языках. Он позволяет проверять значения на соответствие определенным шаблонам и выполнять соответствующее действие в зависимости от того, какой шаблон соответствует значению.

Пример использования оператора match-case:

```
def process_value(value):
    match value:
        case 1:
            print("Value is 1")
        case 2 | 3 | 4:
            print("Value is 2 or 3 or 4")
        case _:
            print("Value is not 1 or 2 or 3 or 4")
```

```
process_value(1) # output: Value is 1
process_value(3) # output: Value is not 1 or 2
```

Оператор `match-case` доступен только в версии Python 3.10 и выше, поэтому если вы используете более старую версию Python, то нужно воспользоваться другими способами для решения задачи, например, использовать условные выражения `if-elif-else` или словари.

93. Поддерживает ли Python регулярные выражения?

> ...если вы решили проблему при помощи регулярных выражений — теперь у вас две проблемы ©

Да, Python поддерживает использование регулярных выражений (regex). В стандартной библиотеке Python имеется модуль `re`, который предоставляет множество функций для работы с регулярными выражениями. Этот модуль позволяет выполнять различные операции, такие как поиск, замена, разбиение текста на подстроки и проверку совпадений с шаблоном регулярного выражения.

Вот, кстати, основные компоненты регулярных выражений, лично я довольно часто ими пользуюсь:

- `\w` - соответствует всем символам "слов". Символы слов являются буквенно-цифровыми (a-z, A-Z символы и подчеркивание).
- `\W` - соответствует символам "не слов". Все, кроме буквенно-цифровых символов и подчеркивания.
- `\d` - соответствует символам "цифр". Любая цифра от 0 до 9.
- `\D` - символы "не цифр". Все, кроме с 0 до 9.
- `\s` - символы пробела, в т.ч. символы табуляции и разрывы строк.
- `\S` - всё, кроме пробелов.
- `.` - любой символ, кроме разрыва строки.
- `[A-Z]` - символы в диапазоне; например, `[A-E]` будет соответствовать A, B, C, D и E.
- `[ABC]` - символы в заданном наборе; например, `[AMT]` будет соответствовать только A, M и T.
- `[^ABC]` - символы, отсутствующие в заданном наборе. Например, `[^A-E]` будет соответствовать всем символам, кроме A, B, C, D и E.
- `+` - одно или несколько вхождений предыдущего символа. Например, `\w+` вернет **ABD12D** в виде единственного соответствия вместо шести разных совпадений.
- `*` - ноль или более вхождений предыдущего символа. Например, `b\w*` соответствует полужирным частям в фразе **b**, **bat**, **bajhdsfbfjhbe**. В целом, он соответствует нулю или более символам "слова" после "b".
- `{m, n}` - не менее m и не более n вхождений предыдущего символа. `{m,}` будет соответствовать не менее m вхождений, и верхнего предела для совпадения не будет. `{k}` будет соответствовать точно k вхождениям предыдущего символа.
- `?` - ноль или одно вхождение предыдущего символа. Например, это может быть полезно при поиска двух вариантов написания для одной и той же работы. Например, `/behaviou?r/` будет соответствовать как **behavior**, так и **behaviour**.
- `|` - соответствует выражению до или после "pipe" символа. Например, `/se(a|e)/` соответствует как see, так и sea.
- `^` - ищет регулярное выражение в начале текста или в начале каждой строки, если включен многострочный флаг.
- `$` - ищет регулярное выражение в конце текста или в конце каждой строки, если включен многострочный флаг.
- `\b` - предыдущий символ соответствует, только если это граница слова.
- `\B` - предыдущий символ соответствует только в том случае, если граница слова отсутствует.
- `(ABC)` - это сгруппирует несколько символов вместе и запомнит подстроку, соответствующую им, для последующего использования. Это называется скобочной группой.
- `(?:ABC)` - это также объединяет несколько символов вместе, но не запоминает совпадение. Это незапоминаемая скобочная группа.
- `\d+(?=ABC)` - это будет соответствовать символу(-ам), предшествующему `(?=ABC)`, только если за ним следует `ABC`. Часть `ABC` не будет включена в массив совпадений. Часть `\d` - это всего лишь пример. Это может быть любая строка регулярного выражения.
- `\d+(?!ABC)` - это будет соответствовать символу(-ам), предшествующему `(?!ABC)`, только если за ним *не* следует `ABC`. Часть `ABC` не будет включена в массив совпадений. Часть `\d` - это всего лишь пример. Это может быть любая строка регулярного

выражения.

Для работы с регулярными выражениями в Python обычно используются строковые литералы с префиксом `r` (raw string), которые позволяют использовать специальные символы без экранирования. Например, регулярное выражение для поиска слов, начинающихся на `"a"` и заканчивающихся на `"b"`, может быть записано так:

```
import re
text = "apple and banana are fruits, but apricot is not"
pattern = r"\ba\w*b\b"
matches = re.findall(pattern, text)
print(matches)  # output: ['apple', 'apricot']
```

Здесь функция `re.findall()` выполняет поиск всех совпадений с шаблоном регулярного выражения `pattern` в строке `text` и возвращает список найденных подстрок.

94. Напишите регулярное выражение, которое будет принимать идентификатор электронной почты. Используйте модуль re.

Для написания регулярок в Python используется модуль `re`.

```
import re

email_regex = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

def is_valid_email(email):
    if re.match(email_regex, email):
        return True
    else:
        return False
```

В этом коде мы создаем регулярное выражение `email_regex`, которое проверяет, соответствует ли переданный идентификатор электронной почты заданному формату. Затем мы используем функцию `re.match()` для сравнения переданного идентификатора электронной почты с регулярным выражением. Если совпадение найдено, мы возвращаем `True`, в противном случае `False`. Например, вызов `is_valid_email('example@mail.com')` вернет `True`, а
вызов `is_valid_email('not_valid_email')` вернет `False`.

https://regex101.com/ — отличный сайт по работе с регулярками, кстати)

95. Как передать необязательные или ключевые параметры из одной функции в другую?

В Python для передачи необязательных параметров в функцию используется синтаксис со знаком звездочки (`*`) и двойной звездочки (`**`). Вот пример:

```
def my_function(required_arg, *args, **kwargs):
    print(required_arg)
    if args:
        print(args)
    if kwargs:
        print(kwargs)

my_function('Hello, world!', 2, 3, 4, my_keyword='some_value')
```

В этом примере `required_arg` - обязательный аргумент функции `my_function`. После этого первого аргумента мы указали символ звездочки (`*`), чтобы пометить все
следующие аргументы как необязательные. В примере, это `args`, который преобразуется в кортеж. Далее, мы указали символ двойной звездочки (`**`), чтобы пометить все
следующие аргументы как необязательные с ключами. Это параметр `kwargs`, который преобразуется в словарь.

В вызове `my_function` , мы передаем обязательный аргумент `'Hello, world!'` , аргументы `args - 2, 3, 4` , и ключевой параметр `my_keyword` со значением `'some_value'` в kwargs.
Таким образом, эта функция может принимать переменное количество аргументов, как позиционных, так и именованных.

96. Как создать свой собственный пакет в Python?

Для создания своего собственного пакета в Python нужно выполнить следующие шаги:

- Создать директорию с именем вашего пакета.
- Внутри директории создать файл `init.py` , который будет пустым, но он необходим, чтобы Python распознал эту директорию как пакет.
- Создать необходимые модули и скрипты внутри директории вашего пакета.
- Определить файл `setup.py` с метаданными вашего пакета и его зависимостями, например:

```
from setuptools import setup, find_packages
setup(
    name='mypackage',
    version='1.0',
    packages=find_packages(),
    install_requires=['numpy', 'scipy']
    )
```

- Создать дистрибутив вашего пакета, выполнив команду `python setup.py sdist`
- Установить свой пакет с помощью pip, выполнив команду `pip install dist/mypackage-1.0.tar.gz.`

После этого вы можете использовать свой пакет в своих проектах или опубликовать его на Python Package Index (PyPI) для использования другими людьми.

97. Что такое функции высшего порядка?

Функции высшего порядка - это функции, которые могут принимать другие функции в качестве аргументов или возвращать функции в качестве результата. Это является важным концептом в функциональном программировании и в целом упрощает написание кода, делая его более простым и модульным (если не перегибать).

В Python встроены несколько функций высшего порядка, таких как `map()` , `filter()` и `reduce()` .
Функция `map()` применяет заданную функцию к каждому элементу итерируемого объекта и возвращает итератор с результатами.
Функция `filter()` применяет заданную функцию к каждому элементу итерируемого объекта и возвращает итератор с элементами, для которых функция вернула True.
Функция `reduce()` объединяет элементы итерируемого объекта в одно значение, используя заданную функцию.

Пример использования `map()` :

```
def square(x):
    return x ** 2

numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)
print(list(squared_numbers))  # [1, 4, 9, 16, 25]
```

Пример использования `filter()` :

```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4, 5]
```

```
even_numbers = filter(is_even, numbers)
print(list(even_numbers)) # [2, 4]
```

Пример использования `reduce()`:

```
from functools import reduce
def add(x, y):
    return x + y

numbers = [1, 2, 3, 4, 5]
sum_of_numbers = reduce(add, numbers)
print(sum_of_numbers) # 15
```

98. Назовите модули в Python, связанные с файлами

Некоторые модули, связанные с файлами в Python:

- `os` — предоставляет функции для работы с операционной системой, включая операции с файлами, такие как создание, удаление и перемещение файлов.
- `sys` — предоставляет функции для работы с системными аргументами командной строки, включая передачу параметров через консоль.
- `pathlib` — предоставляет классы для удобной работы с путями к файлам и директориям.
- `io` — предоставляет классы для работы с текстовыми и бинарными потоками ввода-вывода.
- `shutil` — предоставляет функции для работы с файловой системой, включая операции с файлами, такие как копирование, перемещение и удаление файлов.
- `glob` - позволяет осуществлять поиск файлов по шаблону

Ну и подборка некоторых полезных функций из этих библиотек:

```
import os
os.getcwd()  # показать текущий каталог

os.path.exists('sample_data')  # проверка, существует ли каталог 'sample_data'

os.mkdir('test_dir')  # создаём каталог 'test_dir'

os.listdir('sample_data')  # смотрим содержимое каталога 'sample_data'
```

```
import os
from glob import glob  # список всех .csv в 'sample_data'
all_csv = list(glob(os.path.join('sample_data', '*.csv')))
print(all_csv)
```

```
import shutil  # копируем 'sample_data/README.md' в 'test_dir'
shutil.copy(
    os.path.join('sample_data', 'README.md'),
    os.path.join('test_dir')
)
```

99. В чем разница между NumPy и SciPy?

NumPy и SciPy - это две отдельные библиотеки для Python, которые используются для научных вычислений и работы с массивами данных.

NumPy - это библиотека для работы с многомерными массивами данных, включая матрицы, и предоставляет широкий набор функций для быстрой операции с массивами и векторами. Она часто используется в математических вычислениях, научной обработке данных, машинном обучении и других областях науки и техники.

SciPy - это библиотека для научных вычислений и анализа данных, основанная на NumPy. Она включает множество модулей для работы с различными задачами, такими как оптимизация, интеграция, обработка изображений, статистика, алгебра и другие научные и инженерные задачи.

Таким образом, хотя NumPy используется для основных операций на многомерных массивах и матрицах, SciPy используется для решения более сложных задач научных вычислений, таких как оптимизация, интеграция и обработка изображений.

Некоторые задачи, где может использоваться NumPy:

- Матричные операции и операции линейной алгебры
- Обработка изображения и видео
- Обработка звука и аудио-файлов
- Модули для статистики и машинного обучения, такие как scikit-learn

Некоторые задачи, где может использоваться SciPy:

- Решение систем нелинейных уравнений и оптимизация
- Численное интегрирование и дифференцирование
- Оптимизация функций
- Работа с линейными алгебраическими системами
- Анализ спектральных данных
- Моделирование физических систем и оптимизация их параметров
- Работа с сигналами и изображениями

100. Что такое аксессоры, мутаторы, @property?

`@property` - это декоратор, который позволяет создать метод класса, который может быть использован как атрибут объекта. `@property` можно использовать для создания доступа чтения (геттера) и записи (сеттера) для членов класса. Метод, помеченный как `@property`, может быть доступен как поле класса, без вызова его как функции. Это упрощает код и облегчает чтение и понимание объектного кода.

Аксессоры и мутаторы - это стили префиксов методов, применяемых для чтения и записи значений параметров. Аксессор, также известный как метод доступа или геттер, используется для доступа к значению членов класса, а мутатор, также известный как метод изменения или сеттер, используется для изменения значения членов класса.

Значение `@property` заключается в том, что оно автоматически генерирует геттер и сеттер для члена класса одновременно при использовании этого декоратора. Это
упрощает работу с данными и может сократить объем кода.

Вот простой пример использования `@property`:

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
```

```
        self._name = value

person = Person("John")
print(person.name) # John
person.name = "Mike"
print(person.name) # Mike
```

В этом примере мы создали класс `Person` с приватным полем `_name`, и использовали декоратор `@property` для создания геттера и сеттера для этого поля. Мы можем
получить доступ к значению `_name`, используя свойство `name` объекта, и изменить его значение, используя сеттер, как будто это обычное поле класса.

Итак, суть: для каждой переменной экземпляра класса метод getter (аксессор) возвращает ее значение, а метод setter (мутатор) устанавливает/обновляет ее значение. Для определения геттера и сеттера можно использовать `@property`.

Вот и всё, мы сделали это! Надеюсь, было полезно)

Кстати, вот эти статьи и ресурсы тоже помогут освежить понимание Python перед собесом:

- Полный список вопросов с собеседований по Python для дата-сайентистов.
- python вопросы с собеседований телеграм канал
- Ядро планеты Python. Интерактивный учебник
- Полный список вопросов с собеседований по Python для дата-сайентистов и инженеров
- Обширный обзор собеседований по Python. Советы и подсказки
- Вопросы не мальчика, а джуна. 22 вопроса работодателю на собеседовании на позицию «Middle Python-разработчик»
- Шпаргалка для технического собеседования
- Нейронные сети, машинное обучение, python
- Каверзные вопросы по Python
- 10 проектов с кодом на Python для начинающих

Ну и нетленная классика, полистайте хотя бы по диагонали, легко гуглится:

- Билл Любанович «Простой Python. Современный стиль программирования»
- Дэн Бейдер «Чистый Python. Тонкости программирования для профи»
- Бретт Слаткин «Секреты Python: 59 рекомендаций по написанию эффективного кода»
- Марк Лутц «Изучаем Python»
- Лучано Ромальо «Python. К вершинам мастерства»
- Дэвид Бизли «Python. Книга рецептов»

Если хочется ещё больше полезного контента по Python — welcome в тг `@data_analysis_ml`