Лекция 1

Создание экземпляра класса:

- Вызывается метод __new__ класса Dog для создания нового объекта. Этот метод отвечает за выделение памяти для объекта.
- Созданный объект передается в метод __init__ этого класса для инициализации. Аргументы, переданные в вызове класса, передаются в метод __init__.
- Метод __init__ устанавливает начальные значения атрибутов объекта.
- Возвращается ссылка на вновь созданный и инициализированный объект.

self в Python - это ссылка на текущий экземпляр класса. Он используется для доступа к атрибутам и методам объекта изнутри класса. Он используется для:

1. **Доступ к атрибутам объекта**: С помощью `self` вы можете обращаться к атрибутам объекта внутри методов класса.
2. **Вызов других методов объекта**: `self` позволяет вызывать другие методы того же объекта внутри класса.

Методы, окруженные двойными нижними подчеркиваниями, например __init__, являются специальными методами в Python. Они имеют особое назначение и вызываются автоматически в определенных ситуациях

Приватность атрибутов:

- **Одно нижнее подчеркивание** (`_attribute`): Соглашение между программистами, что атрибут предназначен для внутреннего использования и не должен быть доступен вне класса. Однако он всё еще доступен и может быть изменен извне.
- **Два нижних подчеркивания** (`__attribute`): Приводит к искажению имени атрибута (name mangling), что затрудняет его доступ извне класса. Это ближе к приватным атрибутам в других языках программирования, но всё еще не является строгой приватностью.

Перегрузка (overloading) - это возможность в некоторых языках программирования определять несколько методов с одинаковым именем, но разными сигнатурами (например, разное количество или типы аргументов). При вызове перегруженного метода компилятор или интерпретатор выбирает подходящую версию метода на основе аргументов, переданных в вызове.

Особенности атрибутов класса:

- **Общие для всех экземпляров**: Значение атрибута класса одинаково для всех экземпляров класса. Если значение атрибута класса изменяется, это изменение отражается во всех экземплярах.

- **Доступ к атрибутам класса**: Атрибуты класса можно получить через имя класса или через экземпляр класса.

Статические методы объявляются с помощью декоратора `@staticmethod` и не имеют доступа ни к экземпляру класса, ни к самому классу. Они ведут себя как обычные функции, но принадлежат пространству имен класса. Статические методы используются, когда необходимо выполнить какое-то действие, не зависящее от конкретного экземпляра или класса.

Методы класса объявляются с помощью декоратора `@classmethod` и принимают в качестве первого аргумента ссылку на класс (`cls`). Они могут обращаться к атрибутам класса и вызывать другие методы класса, но не имеют доступа к атрибутам конкретного экземпляра. Методы класса используются, когда необходимо выполнить действие, связанное с самим классом, а не с его экземплярами.

Getter - это метод, который используется для получения значения атрибута объекта. Он позволяет читать значение приватного атрибута, не предоставляя прямой доступ к нему. Геттеры обеспечивают контролируемый доступ к данным и могут выполнять дополнительные действия, такие как форматирование или валидация, перед возвратом значения.

Setter - это метод, который используется для установки значения атрибута объекта. Он позволяет изменять значение приватного атрибута, предварительно выполнив проверки или преобразования. Сеттеры обеспечивают безопасное изменение состояния объекта и предотвращают некорректное использование атрибутов.

Deleter - это метод, который используется для удаления атрибута объекта. Он позволяет контролировать процесс удаления атрибута и выполнять дополнительные действия, такие как (освобождение ресурсов) или вывод сообщений.

Пример:

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

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

    @name.deleter
    def name(self):
        print("Deleting name...")
        del self._name
```

Лекция 2

Магические методы в Python — это специальные методы, имена которых начинаются и заканчиваются двойными подчеркиваниями (например, `__init__`, `__str__`, `__len__` и т.д.). Они также известны как специальные методы или dunder-методы (от double underscore). Магические методы позволяют вам определить поведение ваших классов в контексте встроенных операций и функций Python. С их помощью объекты пользовательских классов могут имитировать поведение встроенных типов данных и взаимодействовать с ключевыми аспектами языка, такими как арифметические операции, контекстное управление ресурсами, работа с атрибутами и многим другим.

Категория	Магические методы
Инициализация и удаление	`__init__(self, [...])`, `__del__(self)`
Представление	`__str__(self)`, `__repr__(self)`
Арифметические операции	`__add__(self, other)`, `__sub__(self, other)`, `__mul__(self, other)`, `__truediv__(self, other)`
Сравнения	`__eq__(self, other)`, `__ne__(self, other)`, `__lt__(self, other)`, `__le__(self, other)`, `__gt__(self, other)`, `__ge__(self, other)`
Работа с коллекциями	`__len__(self)`, `__getitem__(self, key)`, `__setitem__(self, key, value)`, `__delitem__(self, key)`
Итерация	`__iter__(self)`, `__next__(self)`
Контекстное управление	`__enter__(self)`, `__exit__(self, exc_type, exc_val, exc_tb)`

Декструкторы:

Деструктор в Python определяется методом `__del__`. Этот специальный метод вызывается, когда объект удаляется, то есть когда его **счетчик ссылок** достигает нуля.

Счетчик ссылок — это механизм, используемый Python для отслеживания количества ссылок на объект в памяти. Когда количество ссылок становится равным нулю, Python автоматически удаляет объект и освобождает занятую им память.

Метод `__del__` вызывается непосредственно перед тем, как сборщик мусора уничтожит объект и освободит выделенную для него память

Сильные ссылки — это самый обычный тип ссылок, когда один объект напрямую ссылается на другой. Счетчик ссылок объекта увеличивается на единицу каждый раз, когда создается новая сильная ссылка на него.

Циклические ссылки возникают, когда два или более объекта ссылаются друг на друга. Это может препятствовать автоматическому освобождению памяти, так как счетчики ссылок таких объектов не могут достигнуть нуля естественным образом. Python обнаруживает циклические ссылки и удаляет такие объекты, но это может произойти не сразу.

Представление объектов:

__repr__: Этот метод должен возвращать "официальное" строковое представление объекта, которое, по возможности, должно быть достаточно подробным, чтобы с его помощью можно было воссоздать данный объект. `__repr__` предназначен скорее для разработчиков, чем для конечных пользователей, и вызывается функцией `repr()` и внутри оболочек, таких как интерактивная консоль Python.

Метод `__str__` предназначен для создания "неофициального" или удобочитаемого строкового представления объекта, предназначенного для конечного пользователя. Этот метод вызывается функцией `str()` и встроенной функцией `print()`.

Метод `__bytes__` должен возвращать байтовое представление объекта. Этот метод вызывается функцией `bytes()` и может быть использован для получения бинарного представления объекта.

Метод `__format__` используется для определения пользовательского форматирования объекта. Это особенно полезно, когда объекты ваших классов используются в сочетании с функцией `format()` или методом `str.format()`. `format_spec` может содержать спецификации формата, которые указывают, как объект должен быть представлен

Сравнение объектов:

- **`__lt__(self, other)`**: Определяет поведение оператора сравнения "меньше" (<). Возвращает `True`, если объект `self` меньше `other`. [less than]
- **`__le__(self, other)`**: Определяет поведение оператора сравнения "меньше или равно" (<=). Возвращает `True`, если объект `self` меньше или равен `other`. [less than or equal]
- **`__eq__(self, other)`**: Определяет поведение оператора равенства (==). Возвращает `True`, если объекты `self` и `other` считаются равными. [equal]
- **`__ne__(self, other)`**: Определяет поведение оператора неравенства (`!`=). Возвращает `True`, если объекты `self` и `other` не равны. [not equal]
- **`__gt__(self, other)`**: Определяет поведение оператора сравнения "больше" (>). Возвращает `True`, если объект `self` больше `other`. [greater than]
- **`__ge__(self, other)`**: Определяет поведение оператора сравнения "больше или равно" (>=). Возвращает `True`, если объект `self` больше или равен `other`. [greater than or equal]

`__hash__(self)`: Этот метод вызывается функцией `hash(...)` и используется для получения хэш-суммы объекта. Хэш-сумма используется в структурах данных, основанных на хэшировании, таких как словари (`dict`) и множества (`set`). Чтобы объект мог использоваться в качестве ключа в словарях или добавляться в множества, он должен быть "хешируемым", то есть иметь стабильное значение хэша и поддерживать операцию сравнения на равенство.

`__bool__(self)`: Определяет поведение объекта в логическом контексте, например, при использовании в условиях `if`. Метод должен возвращать `True` или `False`. Если этот метод не определен, Python пытается вызвать метод `__len__` объекта: объект считается истинным, если его длина больше нуля, и ложным в противном случае. Определение `__bool__` полезно для классов, логическое значение которых не связано непосредственно с их длиной или для которых требуется специальная логика определения истинности.

Доступ к атрибутам:

`__getattr__(self, name)`

Этот метод вызывается, когда пытается быть прочитан атрибут, который не существует в объекте. Он позволяет определить пользовательскую логику для обработки таких случаев, например, для возврата значения по умолчанию или генерации исключения.

`__setattr__(self, name, value)`

Этот метод автоматически вызывается при попытке назначить значение атрибуту объекта, вне зависимости от того, существует атрибут или нет. Он позволяет определить пользовательскую логику для всех операций присваивания атрибутам объекта.

`__delattr__(self, name)`

Этот метод вызывается, когда атрибут объекта удаляется с помощью `del` или `delattr()`. Подобно `__setattr__`, он позволяет вмешаться в процесс удаления атрибута.

`__dir__(self)`

Метод `__dir__` вызывается функцией `dir()` и должен возвращать список атрибутов объекта. Это может быть полезно для определения, какие атрибуты объекта доступны для чтения.

Создание последовательностей:

`__len__(self)`

Этот метод должен возвращать количество элементов в последовательности. Вызывается встроенной функцией `len()`

`__getitem__(self, key)`

Вызывается при доступе к элементу последовательности по ключу или индексу. Должен возвращать значение элемента или выбрасывать соответствующее исключение.

`__setitem__(self, key, value)`

Вызывается при попытке присвоить значение элементу последовательности. Может выбрасывать исключения при неверном использовании.

`__delitem__(self, key)`

Вызывается при удалении элемента из последовательности с использованием `del`.

`__missing__(self, key)`

Этот метод обычно используется в словарях для обработки случаев, когда ключ не найден. Он не является частью стандартного API последовательностей, но может быть полезен для создания объектов, похожих на словари.

`__iter__(self)`

Возвращает итератор для последовательности, что позволяет объекту быть итерируемым в циклах `for` и в других контекстах.

`__reversed__(self)`

Возвращает итератор, проходящий элементы последовательности в обратном порядке

`__contains__(self, item)`

Проверяет, содержит ли последовательность указанный элемент, и вызывается при использовании операторов `in` и `not in`.

Арифметические операции:

Категория	Метод	Описание
Унарные операторы	`__neg__(self)`	Определяет поведение для отрицания (`-a`).
	`__pos__(self)`	Определяет поведение для унарного плюса (`+a`).
	`__abs__(self)`	Определяет поведение для `abs(...)` .
	`__invert__(self)`	Определяет поведение для инвертирования оператором `~` .
Обычные арифметические	`__add__(self, other)`	Сложение (`a + b`).
	`__sub__(self, other)`	Вычитание (`a - b`).
	`__mul__(self, other)`	Умножение (`a * b`).
	`__matmul__(self, other)`	Умножение матриц (`a @ b`).
	`__truediv__(self, other)`	Деление (`a / b`).
	`__floordiv__(self, other)`	Целочисленное деление (`a // b`).
	`__mod__(self, other)`	Остаток от деления (`a % b`).
	`__divmod__(self, other)`	Деление с остатком (`divmod(a, b)`).
	`__pow__(self, other[, modulo])`	Возведение в степень (`a ** b`).
	`__lshift__(self, other)`	Двоичный сдвиг влево (`a << b`).
	`__rshift__(self, other)`	Двоичный сдвиг вправо (`a >> b`).
	`__and__(self, other)`	Двоичное И (`a & b`).
	`__xor__(self, other)`	Исключающее ИЛИ (`a ^ b`).
	`__or__(self, other)`	Двоичное ИЛИ (`a
Составные присваивания	`__iadd__(self, other)`	Сложение с присваиванием (`a += b`).
	`__isub__(self, other)`	Вычитание с присваиванием (`a -= b`).
	`__imul__(self, other)`	Умножение с присваиванием (`a *= b`).
	`__imatmul__(self, other)`	Умножение матриц с присваиванием (`a @= b`).
	`__itruediv__(self, other)`	Деление с присваиванием (`a /= b`).
	`__ifloordiv__(self, other)`	Целочисленное деление с присваиванием (`a //= b`).
	`__imod__(self, other)`	Остаток от деления с присваиванием (`a %= b`).
	`__ipow__(self, other[, modulo])`	Возведение в степень с присваиванием (`a **= b`).
	`__ilshift__(self, other)`	Двоичный сдвиг влево с присваиванием (`a <<= b`).
	`__irshift__(self, other)`	Двоичный сдвиг вправо с присваиванием (`a >>= b`).
	`__iand__(self, other)`	Двоичное И с присваиванием (`a &= b`).
	`__ixor__(self, other)`	Исключающее ИЛИ с присваиванием (`a ^= b`).
	`__ior__(self, other)`	Двоичное ИЛИ с присваиванием (`a |= b`).
Преобразования типов	`__complex__(self)`	Преобразование типа в комплексное число.
	`__int__(self)`	Преобразование типа к `int` .
	`__float__(self)`	Преобразование типа к `float` .
	`__index__(self)`	Преобразование типа к `int` для использования в срезах.
	`__round__(self[, ndigits])`	Округление числа.
	`__trunc__(self)`	Округление числа в сторону нуля.
	`__floor__(self)`	Округление числа в меньшую сторону.
	`__ceil__(self)`	Округление числа в большую сторону.

Магический метод `__call__` в Python делает объекты класса "вызываемыми", аналогично функциям. Это означает, что вы можете использовать экземпляры класса как если бы они были функциями, вызывая их с помощью круглых скобок и передавая аргументы, если это необходимо.

Определение и использование `__call__`:

```
class CallableObject:
    def __call__(self, *args, **kwargs):
        print(f"Вызван __call__ с аргументами {args} и ключевыми словами {kwargs}"
)
```

Пример использования:

```
callable_object = CallableObject()
callable_object(1, 2, a=3, b=4)
```

`dataclass` — это декоратор, введенный в Python 3.7, предназначенный для упрощения создания классов, главным образом используемых для хранения данных. Он автоматически добавляет специальные методы, включая `__init__`, `__repr__`, `__eq__`, и, при необходимости, `__hash__`, в классы, что уменьшает объем шаблонного кода.

Основные особенности `dataclass`:

- **Автоматическое создание метода `__init__`**: Не нужно явно определять метод `__init__` для инициализации атрибутов класса.
- **Автоматическое создание метода `__repr__`**: Генерируется строковое представление объекта, что упрощает отладку и логирование.
- **Автоматическое создание методов `__eq__` и `__hash__`**: Упрощает сравнение экземпляров и использование класса в качестве ключей в словарях или элементов множеств.

ЛК 3

Наследование в объектно-ориентированном программировании создаёт отношение **"является"** между классами. Это ключевой принцип, позволяющий одному классу (**производному классу**, также известному как дочерний класс или подкласс) наследовать атрибуты и методы другого класса (**базового класса**, также называемого родительским классом или суперклассом).

- Преимущества наследования:
  - **Реализация общих методов** на уровне базового класса, чтобы избежать дублирования кода.
  - Возможность **расширения или модификации** функциональности базового класса в производных классах для выполнения более специфичных задач.

Принцип подстановки Лисков (LSP) является фундаментальным понятием в объектно-ориентированном программировании, обеспечивающим правильное использование наследования

Согласно LSP, **объекты производного класса** должны быть **взаимозаменяемы с объектами базового класса** без нарушения корректности работы программы. Это означает, что поведение производных классов не должно противоречить ожиданиям, основанным на поведении базовых классов.

Ключевые аспекты для соблюдения LSP:

- **Тщательное проектирование**: Необходимо внимательно спроектировать классы и их интерфейсы, чтобы обеспечить совместимость поведения между базовыми и производными классами.
- **Уточнение функциональности**: Производные классы должны расширять или уточнять функциональность базовых классов, не изменяя основных принципов их работы.
- **Сохранение ожидаемого поведения**: Изменения в производных классах не должны приводить к неожиданным результатам при использовании в контексте, предназначенном для базового класса.

Композиция — это ключевая концепция в объектно-ориентированном программировании (ООП), моделирующая отношение "**имеет**" между объектами. Это означает, что один класс (**композит**) может включать в себя один или несколько объектов других классов (**компонентов**), образуя тем самым более сложные структуры.

Как и в других объектно-ориентированных языках есть возможность вызывать методы суперкласса в вашем подклассе. В Python это реализуется через `super()`. Основной сценарий использования — это расширение функциональности унаследованного метода.

Абстрактные классы существуют для наследования, но не для создания экземпляров. Python предоставляет модуль `abc` для формального определения **абстрактных базовых классов**. Вы можете использовать нижние подчеркивания в начале названия класса, чтобы дать понять, что объекты этого класса создавать не следует.

Множественное наследование — это возможность класса наследоваться одновременно от нескольких базовых классов. Эта концепция позволяет одному классу объединить функциональность и свойства нескольких классов.

Множественное наследование часто критикуется и имеет плохую репутацию по нескольким причинам:

- **Сложность восприятия**: Когда класс наследуется от многих базовых классов, становится сложнее понимать его структуру и взаимодействие с базовыми классами.
- **Проблемы с разрешением конфликтов**: Если базовые классы имеют методы с одинаковыми именами, может возникнуть неоднозначность, какой метод будет использоваться.

Большинство современных языков программирования избегают множественного наследования, предпочитая подходы, которые упрощают проектирование и повышают читаемость кода:

- **Интерфейсы**: Вместо множественного наследования, языки поддерживают реализацию нескольких интерфейсов. Интерфейс определяет набор методов, которые класс должен реализовать, но не предоставляет реализацию этих методов.
- **Композиция**: Этот подход предпочтительнее множественного наследования, поскольку он позволяет классу иметь другие классы в

качестве полей, вместо того чтобы наследоваться от них, тем самым избегая сложностей множественного наследования. Мы его рассмотрим немного позже. А пока разберемся с множественным наследованием

Порядок разрешения методов (MRO, Method Resolution Order) определяет, как Python ищет унаследованные методы. Это особенно полезно при использовании функции `super()`, потому что MRO точно указывает, где и в каком порядке Python будет искать метод, вызываемый через `super()`.

У каждого класса есть атрибут `.__mro__`, который позволяет нам проверить порядок. Например:

```
Pyramid.__mro__
```

```
(__main__.Pyramid,
 __main__.Triangle,
 __main__.Square,
 __main__.Rectangle,
 object)
```

Композиция — это концепция объектно-ориентированного проектирования, моделирующая отношение "**имеет**". В композиции класс, называемый *композитным*, содержит объект или компонент другого класса. Другими словами, **композитный класс "имеет" компонент другого класса**.

Эта концепция позволяет композитным классам **повторно использовать реализацию** содержащихся в них компонентов. Композитный класс **не наследует интерфейс** компонентного класса, но может использовать его реализацию.

Связь между двумя классами в композиции считается **слабо связанной**, потому что изменения в классе компонента редко влияют на композитный класс, а изменения в композитном классе никогда не влияют на класс компонента.

Это обеспечивает **лучшую адаптивность к изменениям** и позволяет приложениям вводить новые требования, не затрагивая существующий код.

Сравнивая два конкурирующих программных решения, одно на основе **наследования**, и другое на основе **композиции**, решение на основе композиции обычно бывает **более гибким**.

Миксин - это класс, который предоставляет методы другим классам, но не рассматривается как базовый класс.

Миксин позволяет другим классам повторно использовать его интерфейс и реализацию без превращения в суперкласс. Он реализует уникальное поведение, которое вы можете агрегировать к другим несвязанным классам. Миксины аналогичны композиции, но создают более сильную связь.

Как работают классы Mixin

- **Миксины для повторного использования кода**: Миксины предлагают способ повторного использования кода между классами, которые не находятся в одной иерархии наследования.
- **Не предназначены для прямого инстанцирования**: Миксины предназначены для того, чтобы быть включенными в другие классы и не предназначены для прямого создания экземпляров.
- **Расширение функциональности**: Используя миксины, вы можете легко добавить дополнительную функциональность к существующим классам, не изменяя их иерархию наследования.

Преимущества использования миксинов

1. **Повторное использование кода**: Миксины позволяют избежать дублирования кода, предоставляя общую функциональность, которая может быть повторно использована в различных классах.
2. **Гибкость**: Они предоставляют гибкий способ добавления функциональности к классам без изменения их основной иерархии.
3. **Чистота дизайна**: Использование миксинов может помочь сохранить чистоту и организованность вашего кода, делая его более модульным и легким для понимания.

Класс `AsDictionaryMixin` предоставляет метод **`.to_dict()`**, который возвращает представление самого себя в виде словаря. Метод реализуется как генератор словаря, создающий отображение `prop` в `value` для каждого элемента в `self.__dict__.items()`, если `prop` не является внутренним.

С помощью метода **`._represent()`** проверяется указанное значение. Если значение является объектом, то метод проверяет, есть ли у него член **`.to_dict()`** и использует его для представления объекта. В противном случае возвращается строковое представление. Если значение не является объектом, то просто возвращается значение.

Вывод всего этого:

- **Используйте наследование**, когда объект "является" другим объектом. Обоснуйте это отношение в обоих направлениях (вспоминаем правило подстановки Лиска)
- **Используйте наследование**, чтобы предоставить функциональность миксина нескольким несвязанным классам.
- **Используйте композицию**, чтобы моделировать отношение "имеет", когда объект содержит другой объект.
- **Используйте композицию**, чтобы создавать компоненты, повторно используемые в различных классах.
- **Используйте композицию**, чтобы реализовать группы ролей, политик зарплат и тд, применяемых к другим классам.

- **Используйте композицию**, чтобы изменять поведение во время выполнения без изменения существующих классов.

4 ЛК:

Важность numpy:

- Оптимизированные вычисления
- Основа для других библиотек (SciPy, Pandas, scikit-learn)
- Однородность данных

Создание одномерных массивов

1. **Из Python списков:**
   - Можно создать массив, передав список в `np.array()`.
   - Пример: `arr = np.array([1, 2, 3, 4, 5])`.
2. **С использованием функций NumPy:**
   - `np.arange(start, stop, step)`: создаёт массив с заданным началом, концом и шагом.
   - Пример: `arr = np.arange(0, 10, 2)`.
3. **Случайные данные:**
   - `np.random.rand(number)`: генерирует массив заданной длины с случайными числами.
   - Пример: `random_arr = np.random.rand(5)`.

Размеры и изменение размеров

1. **Размер массива (shape):**
   - `shape` массива показывает его размер. Для одномерного массива это будет количество элементов.
   - Пример: `arr.shape` возвращает `(5,)` для массива из пяти элементов.
2. **Изменение формы массива:**
   - `reshape` используется для изменения формы массива. Хотя это чаще применяется к многомерным массивам, одномерный массив может быть преобразован в многомерный.
   - Примеры: `arr.reshape((5,1))` превращает одномерный массив в двумерный с одним столбцом.

Типы данных в массивах

1. **Определение типа данных:**
   - `dtype` возвращает тип данных элементов массива.
   - Пример: `arr.dtype` может вернуть `int64` для массива целых чисел.
2. **Приведение типов:**
   - Используйте `astype` для изменения типа данных массива.

- Пример: `arr.astype(float)` преобразует массив целых чисел в массив с плавающей точкой.

Индексация в массивах:

1. **Одиночные индексы:**
   - Доступ к конкретному элементу массива по его индексу.
   - Пример: `arr[2]` возвращает третий элемент массива `arr`.
2. **Срезы (Slices):**
   - Доступ к подмассиву с использованием срезов: `start:stop:step`.
   - Пример: `arr[1:5:2]` возвращает элементы с первого по четвёртый с шагом в два.
3. **Целочисленная индексация:**
   - Использование массивов индексов для доступа к элементам.
   - Пример: `arr[[1, 3, 4]]` возвращает массив, состоящий из второго, четвёртого и пятого элементов исходного массива.
4. **Логическая индексация (маски):**
   - Выбор элементов на основе условий.
   - Пример: `arr[arr > 2]` возвращает все элементы массива `arr`, которые больше двух.
5. **Индексация многомерных массивов:**
   - Для многомерных массивов индексация происходит по каждому измерению.
   - Пример: В двумерном массиве `arr2D` элемент `arr2D[1, 3]` обращается к элементу во второй строке и четвёртом столбце.

Срезы не делают копию, как в list

Последний элемент массива a[-1]

Задача 1

Создать numpy-массив, состоящий из первых четырех простых чисел, выведите его тип и размер:

[]

```
arr = np.array([2, 3, 5, 7])

print(arr)

print(arr.dtype)

print(type(arr))

print(arr.shape)

print(arr.nbytes)
```

np.arange

Функция `np.arange` в NumPy работает аналогично встроенной функции Python `range`, но с расширенными возможностями, включая поддержку чисел с плавающей точкой.

Функция np.linspace в NumPy представляет собой мощный инструмент для создания последовательностей чисел. Она позволяет генерировать равномерно распределенные числа в заданном диапазоне, где как начальная, так и конечная точки включаются в последовательность.

Ключевые особенности `np.linspace`:

1. **Включение начальной и конечной точки:** В отличие от `np.arange`, `np.linspace` гарантированно включает в себя как начальное, так и конечное значение в последовательность. Это делает `np.linspace` идеальным выбором для задач, где необходимо точно определить границы диапазона.
2. **Задание количества элементов:** Вместо указания шага между элементами, как в `np.arange`, в `np.linspace` задается общее количество точек, которое должно быть сгенерировано. Это обеспечивает равномерное распределение значений в диапазоне.
3. **Применение в научных вычислениях:** `np.linspace` часто используется в научных вычислениях, например, при создании осей для графиков или при генерации точек для аппроксимации функций.

Задача 2: создать и вывести последовательность чисел от 10 до 32 с постоянным шагом, длина последовательности -- 12. Чему равен шаг?

```
a = np.linspace(10, 32, 12)
print(a)
print(a[1] - a[0])
```

Операции с массивами:

А.Элементарные математические операции

1. **Арифметические операции:**
    - Выполнение операций сложения, вычитания, умножения и деления над массивами.
    - Пример: `arr + 10` увеличивает каждый элемент массива `arr` на 10.
2. **Тригонометрические функции:**
    - Применение тригонометрических функций, таких как `np.sin`, `np.cos`, `np.tan` к элементам массива.
    - Пример: `np.sin(arr)` возвращает массив синусов элементов массива `arr`.
3. **Логарифмические и экспоненциальные функции:**
    - Использование функций `np.log`, `np.exp` для вычисления логарифмов и экспонент.
    - Пример: `np.exp(arr)` возвращает массив, содержащий экспоненты элементов `arr`.

Б. Статистические операции

1. **Основные статистические показатели:**
   - Вычисление среднего (`np.mean`), медианы (`np.median`), стандартного отклонения (`np.std`).
   - Пример: `np.mean(arr)` вычисляет среднее значение элементов массива `arr`.
2. **Минимум и максимум:**
   - Определение минимального (`np.min`) и максимального (`np.max`) значений в массиве.
   - Пример: `np.max(arr)` возвращает максимальный элемент в массиве `arr`.

В. Логические операции

1. **Сравнения:**
   - Выполнение элементарных сравнений, таких как равенство, не равенство, больше, меньше.
   - Пример: `arr > 5` возвращает массив булевых значений, показывающих, какие элементы `arr` больше 5.
2. **Логические функции:**
   - Использование функций `np.any` и `np.all` для проверки условий на всем массиве.
   - Пример: `np.any(arr > 0)` проверяет, есть ли в массиве хотя бы один элемент больше 0.

Г. Манипуляции с массивами

1. **Сортировка:**
   - Сортировка элементов массива функцией `np.sort`.
   - Пример: `np.sort(arr)` возвращает отсортированную копию массива `arr`.
2. **Конкатенация и разделение:**
   - Объединение нескольких массивов в один (`np.concatenate`) и разделение массива на несколько частей (`np.split`).
   - Пример: `np.concatenate([arr1, arr2])` объединяет `arr1` и `arr2` в один массив.

Д. Векторизованные операции

1. **Применение функций:**
   - Векторизация позволяет применять функции к каждому элементу массива без явного использования циклов.
   - Пример: `np.vectorize(lambda x: x ** 2)(arr)` возводит каждый элемент массива в квадрат.

В NumPy существует категория важных функций, известных как универсальные функции или `ufunc`. Эти функции предназначены для эффективного поэлементного выполнения операций на массивах.

Ключевые особенности универсальных функций:

1. **Поэлементное выполнение:** `ufunc` обрабатывают каждый элемент массива независимо, что делает их идеальными для выполнения математических и логических операций.
2. **Поддержка различных операций:** Включают в себя широкий спектр операций – от арифметических (сложение, умножение) до более сложных функций (тригонометрические, логарифмические, экспоненциальные).
3. **Векторизация вычислений:** Использование `ufunc` позволяет избежать написания явных циклов в коде, векторизуя вычисления и тем самым повышая эффективность.
4. **Встроенная поддержка широковещательных (broadcasting) операций:** Это означает, что `ufunc` могут обрабатывать массивы разных размеров и форм, автоматически расширяя их для выполнения операций.

Примеры универсальных функций:

- **Арифметические функции:** `np.add`, `np.subtract`, `np.multiply`, `np.divide`.
- **Тригонометрические функции:** `np.sin`, `np.cos`, `np.tan`.
- **Сравнительные функции:** `np.greater`, `np.less`, `np.equal`.
- **Статистические функции:** `np.mean`, `np.median`, `np.std`.

Кванторы всеобщности и существования.

```
np.any(c == 0.), np.all(c == 0.)
```

Куммулятивные суммы:

```
print(b)
print(b.cumsum())
```

```
[ 1.  6. 11. 16. 21.]
[ 1.  7. 18. 34. 55.]
```

Объединение массивов в одно целое

```
c = np.hstack((a, b, 5*b))
print(a)
print(b)
c
```

```
[-2  0  1  5  6 10 18]
[ 1.  6. 11. 16. 21.]
array([ -2.,   0.,   1.,   5.,   6.,  10.,  18.,   1.,   6.,  11.,  16.,
        21.,   5.,  30.,  55.,  80., 105.])
```

Расщепление массива по индексам

```
x1, x2, x3, x4 = np.hsplit(a, [3, 5, 6])
print(a)
print(x1)
print(x2)
print(x3)
print(x4)
```

```
[-2  0  1  5  6 10 18]
[-2  0  1]
[5 6]
[10]
[18]
```

Манипулияции с элементами:

- **`np.append(arr, values)`**: Создает новый массив, объединяя исходный массив `arr` с `values`. Например, если `arr` содержит [1, 2, 3], а `values` - 4, результатом будет новый массив [1, 2, 3, 4].
- **`np.delete(arr, obj)`**: Возвращает новый массив, в котором удалены элементы из `arr`, указанные в `obj`. Например, `np.delete([1, 2, 3, 4], 1)` вернет [1, 3, 4], удаляя элемент на позиции 1.
- **`np.insert(arr, index, values)`**: Возвращает новый массив, созданный путем вставки `values` в `arr` на указанном `index`. Например, `np.insert([1, 2, 3], 1, 5)` даст [1, 5, 2, 3].

Задание 3:

- Создать массив чисел от -4π до 4π, количество точек 100
- Посчитать сумму поэлементных квадратов синуса и косинуса для данного массива
- С помощью `np.all` проверить, что все элементы равны единице.

```
x = np.linspace(-4*np.pi, 4*np.pi, 100)

np.all((np.sin(x)**2 + np.cos(x)**2).round() == 1)
```

```
True
```

```
(np.sin(x)**2 + np.cos(x)**2)
```

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

```
print(x.size)
```

```
100
```

Двумерные массивы:

1. Создание и основные свойства

- **Инициализация:**
  - Из списков списков: `arr2d = np.array([[1, 2, 3], [4, 5, 6]])`.

- С использованием функций NumPy: `np.zeros((2, 3))`, `np.ones((2, 3))`, `np.full((2, 3), 7)`.

- **Атрибуты:**
  - `shape`: возвращает кортеж с размерами массива. Для матрицы 2x3 `shape` будет `(2, 3)`.
  - `dtype`: тип данных элементов массива.
  - `size`: общее количество элементов в массиве.

2. Индексация и срезы

- **Доступ к элементам:**
  - Доступ к отдельному элементу: `arr2d[0, 1]`.
  - Срезы: `arr2d[:, 1]` возвращает второй столбец.
- **Модификация массива:**
  - Изменение элемента: `arr2d[0, 1] = 10`.
  - Изменение строки/столбца: `arr2d[1, :] = [7, 8, 9]`.

3. Операции над двумерными массивами

- **Арифметические операции:**
  - Производятся поэлементно: `arr2d + 2`, `arr2d * arr2d`.
- **Статистические вычисления:**
  - Методы `mean`, `sum`, `std` и др.: `arr2d.mean(axis=0)` вычисляет среднее по столбцам.
- **Линейная алгебра:**
  - Методы `np.dot`, `np.linalg.det`, `np.linalg.inv` для матричных операций.

4. Изменение формы и размера

- **Изменение формы:**
  - `reshape`: `arr2d.reshape((3, 2))`.
  - `ravel` или `flatten`: преобразование в одномерный массив.
- **Конкатенация и разбиение:**
  - `np.concatenate`, `np.vstack`, `np.hstack`.
  - `np.split`, `np.vsplit`, `np.hsplit`.

По индексу можно обращаться несколькими способами:

```
[ ]  a[1][1], a[1,1]
```

```
(4, 4)
```

Задание 4: Создать квадратную матрицу размера 8, на главной диаг. арифметическая прогрессия с шагом 3 (начиная с 3), а на побочной -1, остальные элементы 0.

```
[ ]  a = np.diag(np.arange(3, 27, 3)) - np.eye(8)[::-1]
     print(a)
```

```
[[ 3.  0.  0.  0.  0.  0.  0. -1.]
 [ 0.  6.  0.  0.  0.  0. -1.  0.]
 [ 0.  0.  9.  0.  0. -1.  0.  0.]
 [ 0.  0.  0. 12. -1.  0.  0.  0.]
 [ 0.  0.  0. -1. 15.  0.  0.  0.]
 [ 0.  0. -1.  0.  0. 18.  0.  0.]
 [ 0. -1.  0.  0.  0.  0. 21.  0.]
 [-1.  0.  0.  0.  0.  0.  0. 24.]]
```

```
print(a*b, '\n') # поэлементное умножение
print(a @ b, '\n') # матричное умножение
print(a.dot(b)) # матричное
```

```
[[10.  5.  5.  5.  5.]
 [ 5. 10.  5.  5.  5.]
 [ 5.  5. 10.  5.  5.]
 [ 5.  5.  5. 10.  5.]
 [ 5.  5.  5.  5. 10.]]

[[30. 30. 30. 30. 30.]
 [30. 30. 30. 30. 30.]
 [30. 30. 30. 30. 30.]
 [30. 30. 30. 30. 30.]
 [30. 30. 30. 30. 30.]]

[[30. 30. 30. 30. 30.]
 [30. 30. 30. 30. 30.]
 [30. 30. 30. 30. 30.]
 [30. 30. 30. 30. 30.]
 [30. 30. 30. 30. 30.]]
```

Задание 5:

- Отобразить вектор размера 100, в котором вычеркивается **x**, если **x** --- составное (т. е. не является простым)

```
[ ]  is_prime = np.ones(100, dtype=bool)
```

```
[ ]  is_prime[:2] = False
```

```
[ ]  n_max = int(np.sqrt(len(is_prime)))
     for i in range(2, n_max):
         is_prime[2 * i: :i] = False # начинаем с 2i с шагом i

     print(is_prime)
```

```
[False False  True  True False  True False  True False False False  True
 False  True False False False  True False  True False False False  True
 False False False False False  True False  True False False False False
 False  True False False False  True False  True False False False  True
 False False False False False  True False False False False False  True
 False  True False False False False False  True False False False  True
 False  True False False False False False  True False False False  True
 False False False False False  True False False False False False False
 False  True False False]
```

Маски

```
a = np.arange(20)
print(a % 3 == 0)
print(a[a % 3 == 0])
```

```
[ True False False  True False False  True False False  True False False
  True False False  True False False  True False]
[ 0  3  6  9 12 15 18]
```

След

```
b = np.diag(a[a >= 10])
print(b)
print('-'*5)
print(np.trace(b))
```

```
[[10  0  0  0  0  0  0  0  0  0]
 [ 0 11  0  0  0  0  0  0  0  0]
 [ 0  0 12  0  0  0  0  0  0  0]
 [ 0  0  0 13  0  0  0  0  0  0]
 [ 0  0  0  0 14  0  0  0  0  0]
 [ 0  0  0  0  0 15  0  0  0  0]
 [ 0  0  0  0  0  0 16  0  0  0]
 [ 0  0  0  0  0  0  0 17  0  0]
 [ 0  0  0  0  0  0  0  0 18  0]
 [ 0  0  0  0  0  0  0  0  0 19]]
-----
145
```

Определение тензора

Тензор в контексте NumPy - это многомерный массив. Мы можем рассматривать тензоры как обобщение векторов и матриц на более высокие размерности. В то время как вектор - это одномерный массив, а матрица - двумерный, тензоры могут иметь три, четыре, пять или даже более измерений.

```
# Создание трехмерного массива
tensor = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

Работа с Размерностями

Размерности тензора важны, так как они определяют его форму и структуру. Мы можем узнать размерность массива с помощью атрибута `.shape`:

```
print(tensor.shape)  # Выведет форму тензора
```

Индексация и Срезы

Индексация в многомерных массивах следует тому же принципу, что и в одно- или двумерных. Однако, с увеличением размерности, возможности индексации становятся более мощными:

```
# Получение определенного элемента
element = tensor[0, 1, 2]  # Вернет элемент в первой матрице, второй строке, треть
ем столбце

# Срезы
slice = tensor[:, 1, :]  # Вернет все элементы второй строки из каждой матрицы
```

Операции с тензорами

Основные арифметические операции (сложение, вычитание, умножение, деление) применимы и к тензорам. Операции между тензорами выполняются поэлементно. Важно, чтобы формы тензоров были совместимы:

```
# Поэлементное умножение тензоров
result = tensor * tensor
```

Применение функций к тензорам

Многие функции NumPy могут быть применены к тензорам, например, для вычисления суммы по определенной оси или для вычисления среднего:

```
# Сумма элементов по первой оси
sum_along_axis = np.sum(tensor, axis=0)
```

```
[ ]  # Определитель
     np.linalg.det(a)
```

```
4.0
```

```
[ ]  # обратная матрица
     b = np.linalg.inv(a)
     print(b)
```

```
[[ 0.75 -0.25]
 [-0.5   0.5 ]]
```

```
[ ]  print(a.dot(b))
     print(b.dot(a))
```

```
[[1. 0.]
 [0. 1.]]
[[1. 0.]
 [0. 1.]]
```

Решение НЛУ.

$$A \cdot x = v$$

```
a = np.array([[2, 1], [2, 3]])
print(a)
v = np.array([5, -10])
x = np.linalg.solve(a, v)
print(x)
# проверка
print(a.dot(x))
```

```
[[2 1]
 [2 3]]
[ 6.25 -7.5 ]
[  5. -10.]
```

Найдем собственные вектора матрицы A.

$$A \cdot x = \lambda \cdot x$$

```
l, u = np.linalg.eig(a)
print(l)
print(u)

print()
# проверка
print('------', end='\n\n')

l, u = np.linalg.eig(a.T)
print(l)
print(u)
```

```
[1. 4.]
[[-0.70710678 -0.4472136 ]
 [ 0.70710678 -0.89442719]]

------

[1. 4.]
[[-0.89442719 -0.70710678]
 [ 0.4472136  -0.70710678]]
```

ЛК 5

Pandas – это библиотека для языка программирования Python, предназначенная для анализа и обработки данных.

- **Простота использования:** Pandas превращает сложные задачи обработки данных в несколько строк кода, делая анализ данных более доступным.

- **Мощные аналитические инструменты:** Она предоставляет широкий набор инструментов для очистки, преобразования, и анализа данных.
- **Интеграция с другими библиотеками:** Pandas хорошо интегрируется с другими популярными библиотеками для машинного обучения

Примеры применения Pandas в реальном мире:

1. **Финансовый анализ:** Анализ временных рядов для акций, создание финансовых моделей.
2. **Наука о данных:** Предварительная обработка и очистка данных перед построением моделей машинного обучения.
3. **Бизнес-аналитика:** Создание отчетов и анализ KPI.
4. **Академические исследования:** Обработка и анализ экспериментальных данных.

1. Series

Описание:

- `Series` – это одномерный массив с метками, способный хранить данные любого типа (целые числа, строки, плавающие числа, Python-объекты и т.д.).
- Каждый элемент в `Series` имеет уникальную метку, называемую индексом.

```python
import pandas as pd # Стандартное сокращение для pandas. Всегда используйте его!

some_list = [1, 3, 5, None, 6, 8]
ser_1 = pd.Series(some_list)
ser_1
```

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

```python
# Так же можно в явном виде указать индексы, чтобы потом было более удобно обращаться к элементам
ind = ['1st day', '2nd day', '3rd day', '4th day', '5rd day', '6th day']

ser_2 = pd.Series(some_list, index=ind)
ser_2
```

```
1st day    1.0
2nd day    3.0
3rd day    5.0
4th day    NaN
5rd day    6.0
6th day    8.0
dtype: float64
```

Индексирование

так же, как и с обычными списками

```python
print(ser_3[0])

print('-----------')

print(ser_3[1:3])

print('-----------')

print(ser_3[::-1])
```

Индексирование pd.Series по условиям

```python
date_range = pd.date_range('20240101', periods=10)
ser_4 = pd.Series(list(range(10)), index=date_range, name='First_days')
ser_4
```

```
2024-01-01    0
2024-01-02    1
2024-01-03    2
2024-01-04    3
2024-01-05    4
2024-01-06    5
2024-01-07    6
2024-01-08    7
2024-01-09    8
2024-01-10    9
Freq: D, Name: First_days, dtype: int64
```

```python
ser_4 > 0.5
```

```
2024-01-01    False
2024-01-02     True
2024-01-03     True
2024-01-04     True
2024-01-05     True
2024-01-06     True
2024-01-07     True
2024-01-08     True
2024-01-09     True
```

1. **Сортировка по значениям**

Сортировка по значениям в `pd.Series` выполняется с помощью метода `sort_values()`.

Пример использования:

```
[ ]  import pandas as pd

     # Создаем Series
     series = pd.Series([3, 1, 4, 1, 5, 9, 2])

     # Сортируем по значениям
     series.sort_values()
```

```
5    9
4    5
2    4
0    3
6    2
1    1
3    1
dtype: int64
```

Ключевые параметры `sort_values()`:

- `ascending`: Логический флаг для сортировки по возрастанию (`True`) или по убыванию (`False`).
- `inplace`: Если `True`, сортировка будет выполнена на месте и изменит исходный объект `Series`.

2. **Сортировка по индексам**

Для сортировки по индексам используется метод `sort_index()`.

Пример использования:

```
# Создаем Series с неупорядоченными индексами
series = pd.Series([3, 11, 4], index=['b', 'a', 'c'])

# Сортируем по индексам
series.sort_index()
```

```
a    11
b     3
c     4
dtype: int64
```

1. **Модификация данных**

- **Присваивание значений:** Можно изменить значения в `Series` по индексам, что более гибко, чем в стандартном списке Python.

```
series = pd.Series([1, 2, 3])
series[0] = 10  # Замена значения
series
```

```
0    10
1     2
2     3
dtype: int64
```

- **Добавление данных:** В `Series` легко добавить новые элементы.

```
series = pd.Series([1, 2, 3])
series.loc['new_index'] = 4  # Добавление нового элемента
series
```

```
0            1
1            2
2            3
new_index    4
dtype: int64
```

- **Удаление данных:** Удаление элементов также удобно и может выполняться по индексу.

```
series = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
series.drop('b', inplace=True)  # Удаление элемента
series
```

```
a    1
b    2
c    3
dtype: int64
a    1
c    3
dtype: int64
```

5. Обработка отсутствующих данных

- `Series` упрощает работу с отсутствующими данными (NaN).

```
series = pd.Series([1, 2, None])
series = series.fillna(0)  # Заполнение отсутствующих значений нулями
series
```

```
0    1.0
1    2.0
2    0.0
dtype: float64
```

2. DataFrame

Описание:

- `DataFrame` – это двумерная, изменяемая по размеру, структура данных с метками для строк и столбцов. По сути, это табличные данные.
- `DataFrame` можно представить как набор объектов `Series`, собранных вместе.

Пример:

```
[ ] # Dataframe можно составить из словаря. Ключ будет соответствовать колонке
    some_dict = {'one': pd.Series([1,2,3], index=['a','b','c']),
                 'two': pd.Series([1,2,3,4], index=['a','b','c','d']),
                 'three': pd.Series([5,6,7,8], index=['a','b','c','d'])}
    df = pd.DataFrame(some_dict)
    df
```

	one	two	three
a	1.0	1.0	5.0
b	2.0	NaN	6.0
c	3.0	3.0	7.0
d	NaN	4.0	8.0
z	NaN	2.0	NaN

```
#Альтернативно, из списка списков с аргументом columns

some_array = [[1, 1, 5], [2, 2, 6], [3 ,3, 7], [None, 4, 8]]
df = pd.DataFrame(some_array, index=['a', 'b', 'c', 'd'], columns=['one', 'two', 'three'])
df
```

	one	two	three
a	1.0	1	5
b	2.0	2	6
c	3.0	3	7
d	NaN	4	8

```
df.values
```

```
array([[ 1.,  1.,  5.],
       [ 2.,  2.,  6.],
       [ 3.,  3.,  7.],
       [nan,  4.,  8.]])
```

```
df.columns
```

```
Index(['one', 'two', 'three'], dtype='object')
```

В dataframe не работает обращение по элементам. Но можно срезами

1. **Чтение данных из CSV файла:** `pd.read_csv`

CSV (Comma-Separated Values) — это простой текстовый формат, предназначенный для представления табличных данных. Каждая строка файла — это одна строка таблицы, а столбцы разделяются запятыми или другими разделителями.

Основное использование:

```
import pandas as pd

df = pd.read_csv('path/to/your/file.csv')
```

Задание 1:

Опишите данный датасет: какое расределение женщин/мужчин в нем? Сколько пассажиров ехало в каждом классе? Какой средний/минимальный/максимальный возраст пассажиров?

```
[ ] print(titanic_passengers['age'].min())
    print(titanic_passengers['age'].max())
    print(titanic_passengers['age'].mean())
```

```
0.1667
80.0
29.8811345124283
```

Задание 2

Сгруппируйте записи по классам пассажиров, в каждой группе посчитайте средний воз

```
titanic_passengers.groupby('pclass', as_index=False)['age'].mean()
```

	pclass	age
0	1.0	39.159918
1	2.0	29.506705
2	3.0	24.816367

Задание 3.

Сколько всего выживших пассажиров? Выживших пассажиров по каждому из полов? Постройте матрицу корреляций факта выживания, пола и возраста.

```
[ ] titanic_passengers['survived'].sum()
```

```
500.0
```

```
titanic_passengers.groupby('sex', as_index=False)['survived'].sum()
```

```
   sex  survived
```

Дальше какой-то треш идет. Всем удачи.