Слайд №1. Полиморфизм и наследование.

Слайд №2. Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Слайд №3. Полиморфизм в операциях. Для целочисленных типов данных оператор «+» используется для выполнения арифметической операции сложения. Подобным образом оператор «+» в строках используется для конкатенации.

Слайд №4. Полиморфные функции. Различные типы данных, такие как строка, список, кортеж, множество и словарь могут работать с функцией len(). Однако, мы можем увидеть, что она возвращает специфичную для каждого типа данных информацию.

Слайд №5. **Полиморфизм в методах класса.** Здесь мы создали два класса Cat и Dog. У них похожая структура и они имеют методы с одними и теми же именами info() и make_sound().

Однако, заметьте, что мы не создавали общего класса-родителя и не соединяли классы вместе каким-либо другим способом. Даже если мы можем упаковать два разных объекта в кортеж и итерировать по нему, мы будем использовать общую переменную animal. Это возможно благодаря полиморфизму.

Слайд №6. Статические методы в Python – по-сути обычные функции, помещенные в класс для удобства и находящиеся в пространстве имен этого класса. Это может быть какой-то вспомогательный код. Обозначается при помощи **@staticmethod** в строке перед методом.

Вообще, если в теле метода не используется self, то есть ссылка на конкретный объект, следует задуматься, чтобы сделать метод статическим. Если такой метод необходим только для обеспечения внутренних механизмов работы класса, то возможно его не только надо объявить статическим, но и скрыть от доступа из вне.

Слайд №7.

Слайд №8.

Слайд №9. Наследование в ООП. Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Класс, от которого производится наследование, называется **базовым или родительским.** Новый класс – **потомком, наследником или производным классом.**

Слайд №10.

Слайд №11. Множественное наследование — это возможность класса иметь более одного родительского класса.

При множественном наследовании дочерний класс наследует все свойства родительских классов. Синтаксис множественного наследования очень похож на синтаксис обычного наследования.

Слайд №12.

Слайд №13. Многоуровневое наследование. Мы также можем наследовать класс от уже наследуемого. Это называется многоуровневым наследованием. Оно может иметь сколько угодно уровней.

В многоуровневом наследовании свойства родительского класса и наследуемого от него класса передаются новому наследуемому классу.

Слайд №14. Порядок разрешения методов MRO. Все классы в Python наследуются от класса object. Это базовый класс языка.

И поэтому технически все классы, встроенные или определенные пользователем, являются наследуемыми, а все объекты — экземплярами класса object.

По порядку разрешения методов любой указанный атрибут сначала ищется в объявленном классе. Если его там нет, поиск продолжается в родительских классах на максимальную глубину слева направо без прохода по одному классу дважды.

MRO должен сохранять локальный порядок старшинства, а также обеспечивать монотонность. Он гарантирует, что класс всегда будет появляться до родителей. В случае нескольких родителей порядок будет таким же, как у кортежей в базовом классе.

MRO класса можно просмотреть в атрибуте __mro__ или с помощью метода mro(). Вызов атрибута возвращает кортеж, а вызов метода — список.

Слайд №15. Использование схемы для вызова метода из класса.

Слайд №16.

```
class X: ...
class Y(X): ...
class A(X, Y): ...
```

Здесь класс **X** наследуется дважды, и куда мы его не поместили в цепочке MRO, он либо нарушит правило старшинства (A -> X -> Y -> object), либо порядка наследования (A -> Y -> X -> object).

Слайд №17. Но что, если мы хотим что-то изменить из этого функционала? Например, добавить работнику через конструктор, новый атрибут, который будет хранить компанию, где он работает или изменить реализацию метода display_info. Python позволяет переопределить функционал базового класса.

Слайд №18. Здесь в классе Employee добавляется новый атрибут - self.company, который хранит компания работника. Соответственно метод __init__() принимает три параметра: второй для установки имени и третий для установки компании. Но если в базом классе определен конструктор с помощью метода __init__, и мы хотим в производном классе изменить логику конструктора, то в конструкторе производного класса мы должны вызвать конструктор базового класса. То есть в конструкторе Employee надо вызвать конструктор класса Person.

Для обращения к базовому классу используется выражение super(). Так, в конструкторе Employee выполняется вызов:

Это выражение будет представлять вызов конструктора класса Person, в который передается имя работника. И это логично. Ведь имя работника устанавливается именно в конструкторе класса Person. В самом конструкторе Employee лишь устанавливаем свойство company.

Кроме того, в классе Employee переопределяется метод display_info() - в него добавляется вывод компании работника. Причем мы могли определить этот метод следующим образом:

Слайд №19. Здесь мы можем увидеть, что такие методы как __str__(), которые не были переопределены в дочерних классах, используются из родительского класса.

Благодаря полиморфизму интерпретатор питона автоматически распознаёт, что метод fact() для объекта a(класса Square) переопределён. И использует тот, который определён в дочернем классе.

С другой стороны, так как метод fact() для объекта b не переопределён, то используется метод с таким именем из родительского класса(Shape).

Слайд №20. Фиксированность**.** После указания __slots__ добавление новых атрибутов в экземпляр класса, кроме уже указанных, невозможно

Также при использовании *__slots__* пропадает возможность получить словарь *__dict__* с атрибутами

Используемая коллекция для хранения имён переменных в __slots__ позволяет ускорить работу программы по сравнению с используемым по умолчанию словарём (*__dict__*).

Слайд №21. Экземпляр класса, использующий *__slots__* снижает количество используемой памяти,
так как содержит только пространство имён объекта.

Таким образом использовать *__slots__* важно в тех случаях, когда:

- Есть необходимость в фиксированном количестве используемых имён переменных в объектах.
- Необходимо ускорение работы программы.
- Имеются ограничения по объему используемой памяти.

__sizeof__ Возвращает размер объекта в байтах.