ТОМСКИЙ
ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

tpu.ru

Углубленный курс информатики

Лекция 7

Модули.

Введение в объектно-ориентированное программирование

21 марта 2022 г.

Долганов Игорь Михайлов,
к.т.н., доцент ОХИ ИШПР

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Содержание

1. Модули
   - Назначение модулей
   - Импортирование и атрибуты
   - Стандартные библиотечные модули
   - Создание модулей
   - Оператор `import`
   - Оператор `from`
   - Смешанные режимы использования: `__name__` и `__main__`
2. Введение в объектно-ориентированное программирование
   - Переменные класса и объекта
   - Иерархия наследования
   - Вызовы методов
   - Создание иерархии классов
   - Примеры создания классов

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Модули

Назначение модулей

- Модули предоставляют простой способ организации компонентов в систему, выступая в качестве изолированных пакетов переменных, которые известны как пространства имен.
- Все имена, определенные на верхнем уровне файла модуля, становятся атрибутами объекта импортированного модуля.
- Импортирование обеспечивает доступ к именам в глобальной области видимости модуля. Таким образом, глобальная область видимости файла модуля превращается в пространство имен объекта модуля, когда файл модуля импортируется.
- В конечном итоге модули Python позволяют связывать индивидуальные файлы в более крупную программную систему.

Назначение модулей

Многократное использование кода

- Модули дают возможность сохранять код в файлах на постоянной основе.
- В отличие от кода, набираемого в интерактивном режиме Python, который исчезает после выхода из него, код в файлах модулей постоянен – его можно перезагружать и повторно запускать столько раз, сколько нужно.
- Модули представляют собой место для определения имен, известных как *атрибуты*, на которые могут ссылаться многочисленные внешние клиенты.
- При правильном применении в результате поддерживается *модульная* конструкция программ, группирующая функциональность в многократно используемые единицы.

Назначение модулей

Разбиение пространства имен системы

- Модули также считаются организационной единицей наивысшего уровня в программах Python.
- Хотя по существу модули – всего лишь пакеты имен, они являются изолированными – Вы не сможете увидеть имя из другого файла, пока явно не импортируете этот файл.
- Во многом подобно локальным областям видимости функций такое решение помогает избежать конфликтов имен в программах.
- Абсолютно все «существует» в модуле, и запускаемый код, и создаваемые объекты всегда неявно заключаются в модули.
- По указанной причине модули представляют собой естественные инструменты для группирования компонентов системы.

Реализация разделяемых служб или данных

- С эксплуатационной точки зрения модули также удобны для реализации компонентов, которые разделяются в рамках системы и потому требуют только одной копии.
- Например, если необходимо предоставить глобальный объект, применяемый в нескольких функциях или файлах, тогда его можно реализовать в модуле, который затем будет импортироваться многими клиентами.

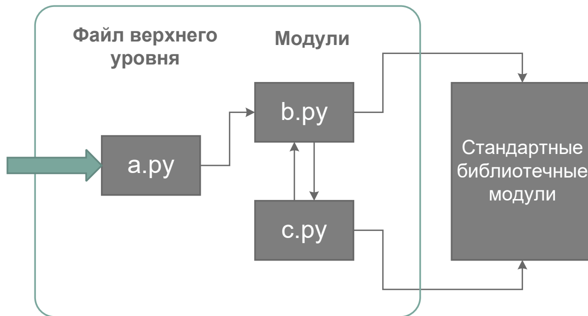
Назначение модулей

Структурирование программы в Python

- На самом базовом уровне программа Python состоит из текстовых файлов, содержащих *операторы* Python, с одним главным файлом верхнего уровня и нулем или большим количеством добавочных файлов, известных как *модули*.
- Файл верхнего уровня содержит главный поток управления программы – именно данный файл запускается для старта приложения.
- Файлы модулей являются библиотеками инструментов, используемых для сбора компонентов, которые применяются файлом верхнего уровня и возможно в других модулях.
- Несмотря на то что файлы модулей тоже относятся к файлам кода, они обычно ничего не делают, когда запускаются напрямую; взамен в них определяются инструменты, предназначенные для использования в других файлах.
- Файл импортирует модуль для получения доступа к определенным в нем инструментам, которые известны как его атрибуты – имена переменных, присоединяемые к таким объектам, как функции.

Импортирование и атрибуты

- На рисунке показана упрощенная структура программы Python, состоящей из трех файлов: `a.py`, `b.py` и `c.py`.
- Файл `a.py` выбран в качестве файла верхнего уровня; он будет простым текстовым файлом с операторами, которые при его запуске выполняются от начала до конца.
- Файлы `b.py` и `c.py` – это модули; они являются простыми текстовыми файлами, также содержащими операторы, но их операторы, как правило, не выполняются напрямую.
- Как объяснялось ранее, модули обычно импортируются другими файлами, которые заинтересованы в применении инструментов, определенных в модулях.



Импортирование и атрибуты

- Предположим, что в файле b.py определена функция по имени fun, предназначенная для внешнего применения.
- Файл b.py будет содержать оператор def языка Python для создания объекта функции, который позже можно запускать, передавая ему ноль или большее количество значений в круглых скобках после имени функции:

```
1 def fun(text): #Файл b.py
2     print(text, "fun")
3
```

- Теперь допустим, что в a.py желательно использовать fun. Для этой цели файл может содержать операторы Python следующего вида:

```
1 import b          #Файл a.py
2 b.fun("fancy")    #Выводит fancy fun
3
```

- Оператор import, предоставляет файлу a.py доступ ко всему тому, что определено кодом верхнего уровня в файле b.py.
- Любой файл способен импортировать инструменты из любого другого файла.
- Модули являются также наивысшим уровнем многократного использования кода в Python. Помещение кода компонентов в файлы модулей делает их полезными в Вашей первоначальной программе и в любых других программах, которые Вы можете написать позже.

Стандартные библиотечные модули

- В состав Python входит крупная коллекция служебных модулей, известная как *стандартная библиотека*.
- Эта коллекция, насчитывающая свыше 300 модулей, содержит независимую от платформы поддержку для распространенных задач программирования: интерфейсы к операционной системе, постоянство объектов, сопоставление текста с образцом, написание сценариев для работы с сетями и Интернетом, построение графических пользовательских интерфейсов и многие другие.
- Ни один из таких инструментов не является частью самого языка Python, но Вы можете их использовать путем импортирования подходящих модулей в любой стандартной копии Python.

Создание модулей

- Эквивалентный код с def потребовал бы временных имен функций (они могли бы конфликтовать с остальными именами) и определений функций за пределами контекста их планируемого применения (который может находиться на сотни строк дальше):
- Чтобы определить модуль, наберите в своем текстовом редакторе какой-нибудь код Python и сохраните его в текстовом файле с расширением .py; любой файл такого рода автоматически считается модулем Python.
- Все имена, присвоенные на верхнем уровне модуля, становятся его *атрибутами* (именами, ассоциированными с объектом модуля) и экспортируются для применения клиентами – переменные автоматически превращаются в атрибуты объекта модуля.
- Например, если Вы поместите следующий оператор def в файл `module1.py` и импортируете его, то создадите объект модуля с одним атрибутом – именем `printer`, которое окажется ссылкой на объект функции:

```
def printer(x):
    print(x)

```

Имена файлов модулей

- Имена файлов модулей должны заканчиваться суффиксом `.py`, если планируется их импортировать. Формально наличие `.py` необязательно для файлов верхнего уровня, которые будут запускаться, но не импортироваться, однако добавление суффикса `.py` во всех случаях делает типы файлов более очевидными и дает возможность импортировать любой файл в будущем.
- Из-за того, что имена модулей становятся именами переменных внутри программы Python (без суффикса `.py`), они также обязаны следовать обычным правилам именования переменных.
- Например, можно создать файл модуля по имени `if.py`, но его будет нельзя импортировать, т.к. `if` является зарезервированным словом – оператор `import if` приведет к синтаксической ошибке.
- Имена файлов модулей могут содержать только буквы, цифры и подчеркивания.

Оператор `import`

- В примере имя `module1` служит двум разным целям – оно идентифицирует внешний файл, подлежащий загрузке, и оно становится переменной в сценарии, которая ссылается на объект модуля после того, как файл загружен:

```
>>> import module1   #Модуль как единое целое (один или несколько)

>>> module1.printer("Hello world!")   #Уточнить, чтобы получить имена
Hello world!

```

- В операторе `import` просто указывается одно или несколько имен модулей для загрузки, разделенные запятыми. Так как оператор `import` дает имя, которое ссылается на полный объект модуля, мы обязаны задавать имя модуля, чтобы извлечь его атрибуты (например, `module1.printer`).

Оператор from

- Напротив, поскольку оператор from копирует специфические имена из одного файла в другую область видимости, он дает возможность применять скопированные имена в сценарии напрямую, не уточняя их именем модуля (например, printer):

```
>>> from module1 import printer   #Импортировать объект (один или
                                  #несколько)

>>> printer("Hello world!")       #Уточнение не требуется
Hello world!

```

- Такая форма from позволяет указывать одно или несколько имен для копирования, разделенных запятыми.
- Здесь оператор from имеет такой же эффект, как в предыдущем примере, но из-за того, что импортированное имя копируется в область видимости, где находится from, использование этого имени в сценарии сопряжено с меньшим объемом набора – мы можем работать с именем напрямую, не задавая включающий модуль.

Оператор from *

- В следующем примере применяется особая форма оператора from: когда вместо специфических имен используется *, мы получаем копии *всех имен*, присвоенных на верхнем уровне указанного модуля.
- Затем мы снова можем задействовать в своем сценарии скопированное имя printer, не уточняя его именем модуля:

```
>>> from module1 import *   #Копировать все переменные
>>> printer("Hello world!")
Hello world!

```

- Формально операторы import и from вызывают ту же самую операцию импортирования; форма from * просто добавляет дополнительный шаг, который копирует все имена из модуля в импортирующую область видимости.
- Поскольку оператор from делает местоположение переменной менее явным и понятным (для читателя кода имя не настолько выразительно, как модуль.имя), некоторые пользователи Python рекомендуют отдавать предпочтение оператору import, а не from.
- Форма from module import * на самом деле *способна* искажать пространства имен и делать имена трудными для понимания, главным образом, когда применяется к более чем одному файлу – в данном случае невозможно выяснить, из какого модуля поступило имя, не считая поиска во внешних файлах исходного кода. В действительности форма from * сворачивает одно пространство имен внутри другого и потому сводит на нет саму цель модулей, заключающуюся в разбиении на пространства имен.

Расширение as для операторов import и from

- Со временем операторы import и from были расширены, чтобы позволить назначать импортированному имени другое имя в сценарии:

```
import modulename as name  #Использовать name вместо modulename

```

- Это расширение часто применяется с целью предоставления кратких псевдонимов для более длинных имен и устранения конфликтов имен, когда в сценарии уже используется имя, которое иначе было бы перезаписано обычным оператором:

```
import reallylongname as name
from module1 import utility as util1
from module2 import utility as util2

name.func()
util1()
util2()

```

- Если в новом выпуске библиотеки модуль или инструмент, широко используемый в вашем коде, получает новое имя, тогда Вы при импортировании всего лишь назначаете ему прежнее имя и предотвращаете нарушение работоспособности имеющегося кода:

```
import newname as oldname
from library import newname as oldname

```

Смешанные режимы использования: __name__ и __main__

- Следующий прием, связанный с модулями, позволяет Вам импортировать файл как модуль и запускать его как автономную программу; он широко применяется в Python-программах.
- Каждый модуль имеет встроенный атрибут по имени __name__, который Python автоматически создает и присваивает следующим образом:
  - Если файл запускается как программа верхнего уровня, тогда во время старта атрибут __name__ устанавливается в строку "__main__".
  - Если взамен файл импортируется, то атрибут __name__ устанавливается в имя модуля, как известно его пользователям.
- Результатом будет то, что модуль может проверять собственный атрибут __name__ для выяснения, запущен он или импортирован. Например, пусть создан следующий файл модуля по имени runme.py, экспортирующий единственную функцию tester:

```
1  def tester():
2  print("It's Christmas in Heaven...")
3
4  if __name__ == "__main__":  #Выполняется только когда запущен
5  tester()  #Не когда импортирован
6
```

Смешанные режимы использования: `__name__` и `__main__`

- В модуле определена функция для импортирования и использования клиентами обычным образом:

```
>>> import runme
>>> runme.tester()
It"s Christmas in Heaven...

```

- Но модуль в самом конце содержит код, который настроен на автоматический вызов функции `tester`, когда данный файл запускается как программа:

```
c:\code> python runme.py
It"s Christmas in Heaven...

```

- В сущности, переменная `__name__` модуля служит флагом режима использования, позволяющим его коду быть задействованным как импортируемая библиотека и как сценарий верхнего уровня.
- Несмотря на простоту, Вы увидите, что такая привязка применяется в большинстве программных файлов Python, с которыми Вам, возможно, придется сталкиваться в реальном мире – для тестирования и для двойного использования.

ТОМСКИЙ
ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

Введение в объектно-ориентированное программирование

Введение в объектно-ориентированное программирование

- **Классы** и **объекты** – это два основных аспекта объектно-ориентированного программирования. Класс создаёт новый тип, а объекты являются экземплярами класса.
- **Класс** – это способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния, а также правила для взаимодействия с данной сущностью (контракт).

Пример

Класс будет отображать сущность – автомобиль. Атрибутами класса будут являться двигатель, подвеска, кузов, четыре колеса и т.д. Методами класса будет «открыть дверь», «нажать на педаль газа», а также «закачать порцию бензина из бензобака в двигатель». Первые два метода доступны для выполнения другим классам (в частности, классу «Водитель»). Последний описывает взаимодействия внутри класса и не доступен пользователю.

```
1  class Person:
2      def go(self, where="nowhere"):
3          print(where)
4
5
6  class Person(object):
7      pass
8
```

Введение в объектно-ориентированное программирование

- **Объект** (**экземпляр**) – это отдельный представитель класса, имеющий конкретное состояние и поведение, полностью определяемое классом.
  Объект имеет конкретные значения атрибутов и методы, работающие с этими значениями на основе правил, заданных в классе. В приведенном примере, если класс – это некоторый абстрактный автомобиль из «мира идей», то объект – это конкретный автомобиль, стоящий под окнами.

```
petr = Person()
ivan = Person()

print(petr)

```

- **Инкапсуляция** – свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе. Обычно инкапсуляцию отождествляют с сокрытием данных.
- **Наследование** – свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс – потомком, наследником, дочерним или производным классом.

Введение в объектно-ориентированное программирование

self

- Методы класса имеют одно отличие от обычных функций: они должны иметь дополнительно имя, добавляемое к началу списка параметров. Однако, при вызове метода никакого значения этому параметру присваивать не нужно – его укажет Python.
- Эта переменная указывает на сам объект экземпляра класса, и называется `self`.

Конструктор

Конструктор класса – специальный блок инструкций, вызываемый при создании объекта. В Python это метод `__init__`.

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


person = Person(name="Petr", surname="Petrov")
print(f"Hi, {person.name} {person.surname}")
for person in [Person("Petr", "Petrov"), Person("Masha", "Petrova")]:
    print(f"Hi, {person.name} {person.surname}")

```

Переменные класса и объекта

- Поля можно воспринимать как обычные переменные, заключённые в **пространствах имён** классов и объектов. Их имена действительны только в контексте (пространстве имен) этих классов или объектов.
- **Переменные класса** разделяемы – доступ к ним могут получать все экземпляры этого класса. Переменная класса существует только одна, поэтому когда любой из объектов изменяет переменную класса, это изменение отразится и во всех остальных экземплярах класса.
- **Переменные объекта** принадлежат каждому отдельному экземпляру класса. В этом случае у каждого объекта есть своя собственная копия поля, т.е. не разделяемая с другими такими же полями в других экземплярах. Доступ к полям объекта осуществляется через переменную `self`.

Переменные класса и объекта

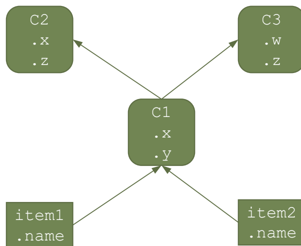

```
class Robot(object):
    population = 0

    def __init__(self, name):
        self.name = name
        print("  **Инициализация %s**" % self.name)
        Robot.population += 1

    def say_hi(self):
        print("Приветствую! Мои хозяева называют меня %s." % self.name)

    @staticmethod
    def how_many():
        print("У нас %d роботов." % Robot.population)

droid1 = Robot("R2-D2")
droid1.say_hi()

Robot.how_many()

droid2 = Robot("C-3PO")
droid2.say_hi()
Robot.how_many()
```

Иерархия наследования

- Извлечение атрибутов – это просто поиск в дереве.
- Термин *наследование* применяется из-за того, что объекты ниже в дереве наследуют атрибуты, присоединенные к объектам выше в дереве.
- По мере того, как поиск продолжается снизу вверх, связанные в дерево объекты в некотором смысле представляют собой объединение всех атрибутов, определенных во всех родителях в дереве на всем пути вверх.
- В Python все происходит буквально: мы действительно строим дерево связанных объектов с помощью кода, а интерпретатор во время выполнения на самом деле поднимается по такому дереву в поисках атрибутов каждый раз, когда встречается выражение `объект.атрибут`.

Иерархия наследования

На рисунке изображено дерево из пяти объектов, помеченных переменными, которые имеют присоединенные атрибуты, готовые для поиска.



- Дерево связывает вместе три объекта классов (`C1`, `C2` и `C3`) и два объекта экземпляров (прямоугольники `item1` и `item2`), образуя дерево поиска в иерархии наследования.
- Классы генерируют экземпляры. Атрибуты классов обеспечивают поведение (данные и функции), которое наследуется всеми экземплярами, сгенерированными из них (например, функция для расчета заработной платы сотрудника на основе оклада и отработанных часов).
- Экземпляры представляют конкретные элементы в предметной области программы. Атрибуты экземпляров хранят данные, которые варьируются для каждого отдельного объекта (скажем, номер карточки социального страхования сотрудника).

Иерархия наследования

- С точки зрения деревьев поиска экземпляр наследует атрибуты от своего класса, а класс наследует атрибуты от всех классов выше в дереве.
- Классы, расположенные более высоко в дереве (наподобие `C2` и `C3`), мы обычно называем *суперклассами*; классы, находящиеся ниже в дереве (вроде `C1`) известны как *подклассы*.
- Суперклассы обеспечивают поведение, разделяемое всеми их подклассами, но поскольку поиск направлен снизу вверх, подклассы могут переопределять поведение, определенное их суперклассами, за счет переопределения имен суперклассов ниже в дереве.
- Предположим, что мы построили дерево, приведенное на слайде 27, и затем написали:

```
1  item2.w
2
```

- Код сразу же обращается к наследованию. Поскольку он представляет собой выражение `объект.атрибут`, инициируется поиск в дереве, показанном на слайде 27 – Python будет искать атрибут `w` в `item2` и выше.
- В частности, он будет проводить поиск внутри связанных объектов в следующем порядке: `item2`, `C1`, `C2`, `C3` и остановится при нахождении первого присоединенного атрибута `w` (или сообщит об ошибке, если `w` не удалось отыскать).

Иерархия наследования

- В данном случае атрибут `w` не будет найден до тех пор, пока не пройдет поиск в C3, потому что он присутствует только в этомобъекте.
- Другими словами, благодаря автоматическому поиску `item2.w` распознается как `C3.w`.
- В терминах ООП экземпляр `item2` «наследует» атрибут `w` от C3.
- В конечном итоге два экземпляра наследуют от своих классов четыре атрибута: `w`, `x`, `y` и `z`. Ссылки на другие атрибуты будут вызывать проход по другим путям в дереве.

  Вот несколько примеров:
  - Для `item1.x` и `item2.x` атрибут `x` обнаруживается в C1 и поиск останавливается, т.к. C1 располагается в дереве ниже, чем C2.
  - Для `item1.y` и `item2.y` атрибут `y` обнаруживается в C1, поскольку это единственное место, где присутствует `y`.
  - Для `item1.z` и `item2.z` атрибут `z` обнаруживается в C2, потому что C2 располагается в дереве левее, чем C3.
  - Для `item2.name` атрибут `name` обнаруживается в `item2` без подъема по дереву.
- В предыдущем списке первый элемент является, вероятно, наиболее важным – поскольку класс C1 переопределяет атрибут `x` ниже в дереве, он фактически замещает его версию, находящуюся выше в C2.

Вызовы методов

- На предыдущих слайдах мы видели, что ссылка на атрибут `item2.w` в примере дерева классов была оттранслирована в `C3.w` посредством процедуры поиска внутри иерархии наследования в Python. Тем не менее, столь же важно понимать, что происходит, когда мы пытаемся вызывать методы – функции, присоединенные к классам в качестве атрибутов.
- Если ссылка `item2.w` представляет собой вызов *функции*, тогда в действительности она означает «вызвать функцию `C3.w` для обработки `item2`». То есть Python будет автоматически отображать вызов `item2.w()` на вызов `C3.w(item2)`, передавая унаследованной функции экземпляр в первом аргументе.
- Фактически всякий раз, когда вызывается функция, подобным образом присоединенная к классу, всегда подразумевается экземпляр класса.
- Подразумеваемый объект или контекст отчасти является причиной того, что мы называем это объектно-ориентированной моделью – при выполнении операции всегда имеется подчиненный объект.
- В более реалистичном примере мы могли бы вызывать метод повышения по имени `giveRaise`, присоединенный в виде атрибута к классу сотрудника `Employee`; такой вызов не имеет смысла, если только он не уточнен объектом сотрудника, в отношении которого должно быть применено повышение.

Создание иерархии классов

- Каждый оператор `class` генерирует новый объект класса.
- При каждом обращении к классу он генерирует новый объект экземпляра.
- Экземпляры автоматически связываются с классами, из которых они были созданы.
- Классы автоматически связываются со своими суперклассами в соответствии со способом их перечисления внутри круглых скобок в строке заголовка `class`; порядок слева направо здесь дает порядок в дереве.
- Для того, чтобы построить дерево, показанное на слайде 27 мы могли бы запустить представленный далее код Python. Подобно определениям функций код классов обычно помещается в файлы модулей и выполняется во время импортирования (для краткости внутренности операторов `class` опущены):

```
1  class C2: ...   #Создание объектов классов
2  class C3: ...
3  class C1(C2, C3): ...   #Наследование от суперклассов (в указанном порядке)
4  item1 = C1()   #Создание объектов экземпляров
5  item2 = C2()   #Связывание с его классом
6
```

- Здесь мы создаем три объекта классов посредством трех операторов `class` и два объекта экземпляров за счет двукратного обращения к классу `C1`, как если бы он был функцией. Экземпляры запоминают класс, из которого были созданы, а класс `C1` запоминает свои перечисленные суперклассы.

Создание иерархии классов

- Классы обеспечивают поведение для своих экземпляров с помощью функций методов, которые мы создаем за счет написания кода операторов def внутри операторов class.
- Поскольку такие вложенные операторы def присваивают имена внутри класса, они в итоге присоединяют к объекту класса атрибуты, которые будут наследоваться всеми экземплярами и подклассами:

```
class C2: ... #Создание объектов суперклассов
class C3: ...
class C1(C2, C3): #Создание и связывание класса C1
  def setname(self, who): #Присваивание name: C1.setname
    self.name = who #self является либо item1, либо item2
item1 = C1() #Создание двух экземпляров
item2 = C1()
item1.setname("john") #Установка item1.name в "john"
item2.setname("james") #Установка item2.name в "james"
print(item1.name) #Выводит john
```

- Ничего уникального в плане синтаксиса def в этом контексте нет. Когда оператор def появляется внутри class, он известен как *метод* и автоматически получает особый первый аргумент, называемый self, который позволяет обращаться к обрабатываемому экземпляру.
- Любые значения, которые Вы передаете методу самостоятельно, отправляются аргументам, следующим после self (здесь who).

Примеры создания классов

- Определим класс по имени `FirstClass`, выполнив оператор `class` в интерактивном режиме.

```
1  >>> class FirstClass:                  #Определяем объект класса
2  ...:     def setdata(self, value):     #Определяем методы класса
3  ...:         self.data = value         #self - это экземпляр
4  ...:     def display(self):
5  ...:         print(self.data)          #self.data: для каждого экземпляра
6
```

- Как и все составные операторы, оператор `class` начинается со строки заголовка с именем класса, после чего следует тело с одним или несколькими вложенными операторами, (обычно) набранными с отступом. В приведенном примере вложенными операторами являются `def`; они определяют функции, которые реализуют поведение класса, предназначенное для экспортирования.
- В примере операторы `def` присваивают объекты функций именам `setdata` и `display` в области видимости оператора `class`, а потому генерируют атрибуты, присоединяемые к классу – `FirstClass.setdata` и `FirstClass.display`.
- В действительности любое имя, присвоенное на верхнем уровне вложенного блока класса, становится атрибутом этого класса.

Примеры создания классов

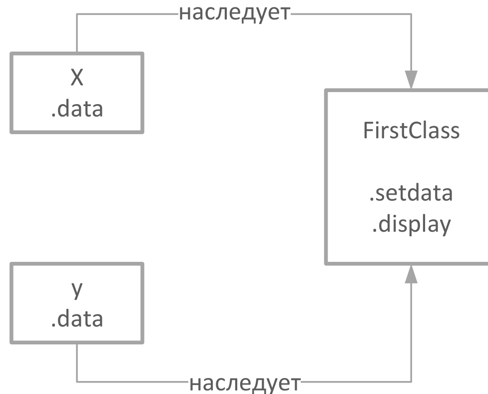
- Функции внутри класса, как правило, называются методами. Они создаются посредством операторов `def` и поддерживают все, что Вам уже известно о функциях (т.е. могут иметь стандартные значения аргументов, возвращать значения, выдавать элементы по запросу и т.п.).
- Но первый аргумент в функции метода при ее вызове автоматически получает подразумеваемый объект экземпляра – объект, на котором произведен вызов. Создадим пару экземпляров, чтобы посмотреть, как все работает:

```
>>> x = FirstClass()   #Создаем два экземпляра
>>> y = FirstClass()   #Каждый представляет новое пространство имен

```

- *Обращаясь* к классу таким способом (обратите внимание на круглые скобки), мы генерируем объекты экземпляров, представляющие собой просто пространства имен, которые имеют доступ к атрибутам своих классов.
- В этой точке мы имеем три объекта: два экземпляра и класс.

Примеры создания классов

- В действительности мы располагаем тремя связанными пространствами имен, как иллюстрируется на рисунке. В терминах ООП мы говорим, что экземпляр `x` «является» `FirstClass`, равно как и `y` – они оба наследуют имена, присоединенные к классу.



Примеры создания классов

- Два экземпляра начинают свое существование как пустые, но имеют ссылки на класс, из которого они были сгенерированы. Если мы уточним экземпляр с помощью имени атрибута, который находится в объекте класса, тогда Python извлечет имя из класса посредством поиска в иерархии наследования (при условии, что он также не присутствует в экземпляре):

```
1  >>> x = setdata("Sir Lancelot")  #Вызвать методы: self - это x
2  >>> y = setdata(2.71828)    #Выполняется First.Class.setdata(y, 2.71828)
3
```

- Оба экземпляра класса `FirstClass` – `x` и `y` не имеют собственного атрибута `setdata`, поэтому чтобы найти его, Python следует по ссылке из экземпляра в класс.
- Таким образом, наследование происходит во время уточнения атрибутов и предусматривает лишь поиск имен в связанных объектах.
- В методе `setdata` класса `FirstClass` передаваемое значение присваивается `self.data`.
- Внутри метода `self` (имя, по соглашению назначаемое крайнему слева аргументу) автоматически ссылается на обрабатываемый экземпляр (`x` или `y`), так что присваивания сохраняют значения в пространствах имен экземпляров, а не класса; подобным образом создавались имена `data` на рисунке (слайд 35).

Примеры создания классов

- Поскольку классы способны генерировать множество экземпляров, методы должны с помощью аргумента `self` получать обрабатываемый экземпляр. Вызвав метод `display` класса для вывода `self.data`, мы заметим, что он отличается для каждого экземпляра; с другой стороны, само имя `display` одинаковое в `x` и `y`, т.к. оно наследуется от класса:

```
>>> x.display()   #self.data отличается в каждом экземпляре
Sir Lancelot

>>> y.display()   #Выполняется FirstClass.display(y)
2.71828

```

- В каждом экземпляре мы сохраняем в атрибуте `data` объекты разных типов – строку и число с плавающей точкой. Для атрибутов экземпляров не предусмотрено каких-либо объявлений; они появляются при первом присваивании значений.
- В качестве еще одного способа оценить, насколько динамична эта модель, можно изменить атрибуты в самом классе, присваивая `self` в методах, или *за пределами* класса путем присваивания явному объекту экземпляра:

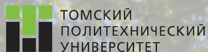
```
>>> x.data = "King Arthur"   #Можно получать/устанавливать атрибуты

>>> x.display()              #в том числе за пределами класса
King Arthur

```

Примеры создания классов

- Разрешено генерировать совершенно новый атрибут в пространстве имен экземпляра, присваивая значение его имени за пределами функций методов класса:

```
1  >>> x.anothername = "Pendragon" #Можно присваивать новые атрибуты
2
```

- Такой оператор присоединит к объекту экземпляра `x` новый атрибут по имени `anothername`, который может использоваться или нет любым методом класса.
- Классы обычно создают все атрибуты экземпляра путем присваивания аргумента `self`, но они не обязаны поступать так – программы могут извлекать, изменять или создавать атрибуты для любых объектов, ссылками на которые они располагают.
- Однако, как правило, нет смысла добавлять данные, которыми класс не в состоянии пользоваться.



Контакты

Долганов Игорь Михайлович,
к.т.н., доцент ОХИ ИШПР

Учебный корпус №2, ауд. 136

dolganovim@tpu.ru

+7-960-978-43-07

Благодарю за внимание!