КАЛУЖСКИЙ ФИЛИАЛ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО
ОБРАЗОВАТЕЛЬНОГО УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ Н.Э. БАУМАНА
(национальный исследовательский университет)»

Факультет «Информатика и управление»

Кафедра «Программное обеспечение ЭВМ, информационные технологии»

Высокоуровневое программирование

Лекции №15-16.
«Принципы ООП. Классы в Python»

Калуга - 2020

Технология ООП

- ООП определяется как технология создания сложного программного обеспечения, которая основана на представлении программы в виде *совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств*.
- Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований. В частности, программирование, не основанное на иерархических отношениях, не относится к ООП, а называется программированием на основе абстрактных типов данных.

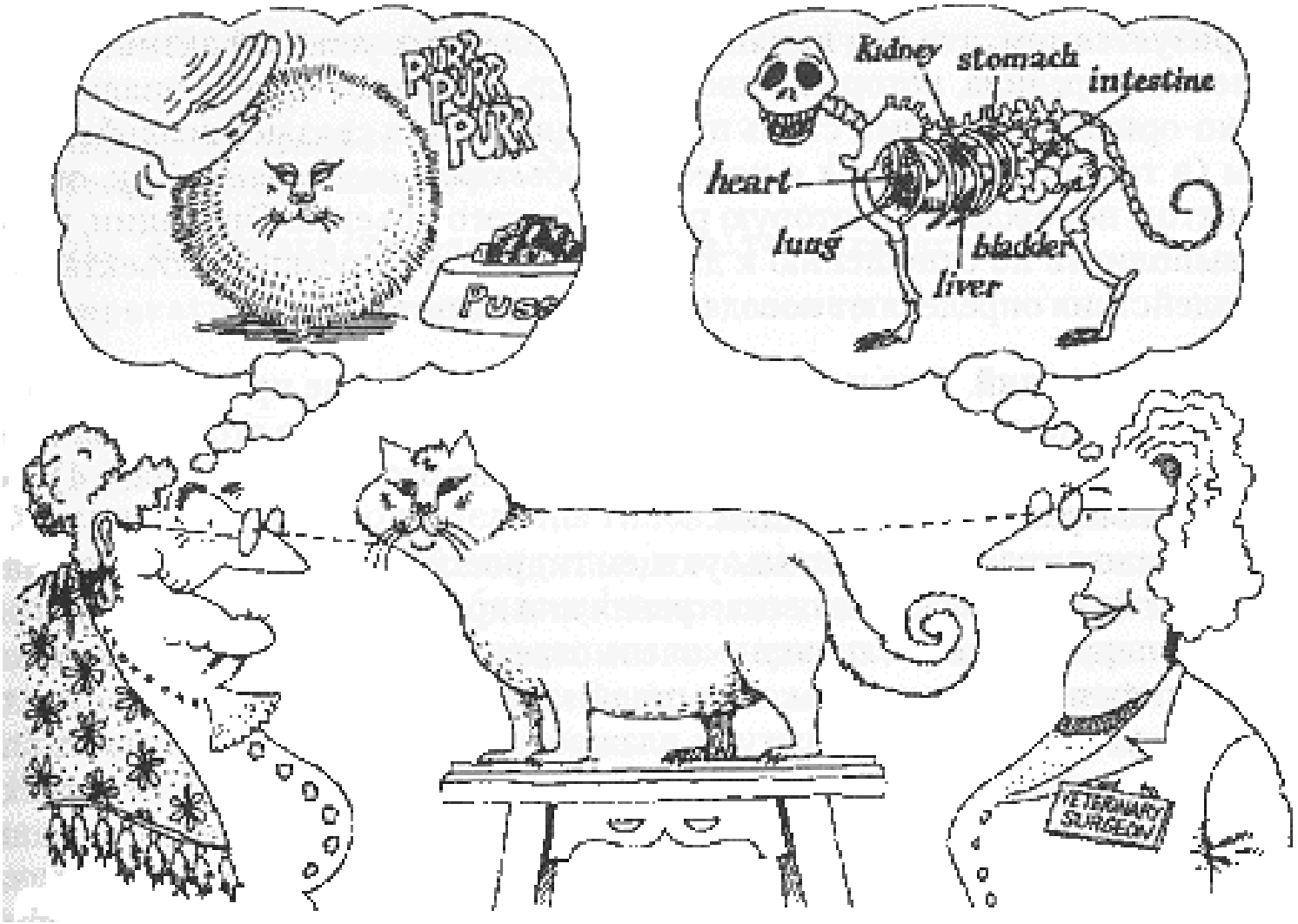
Технология ООП

- Базовым понятием объектно-ориентированного программирования является *объект*. Практически любой элемент решаемой задачи может быть представлен как объект. Например, собака, дерево, телефон. Иначе говоря, данный подход к программированию можно выразить следующим образом – всё является объектом.
- *Объект можно определить как осязаемую сущность, которая четко проявляет свое поведение.*

Принципы ООП

- Абстрагирование
- Ограничение доступа (инкапсуляция)
- Модульность
- Иерархичность
- Полиморфизм
- Типизация
- Параллелизм
- Устойчивость (сохраняемость)

***Абстракция* фокусируется на существенных с точки зрения наблюдателя характеристиках объекта.**



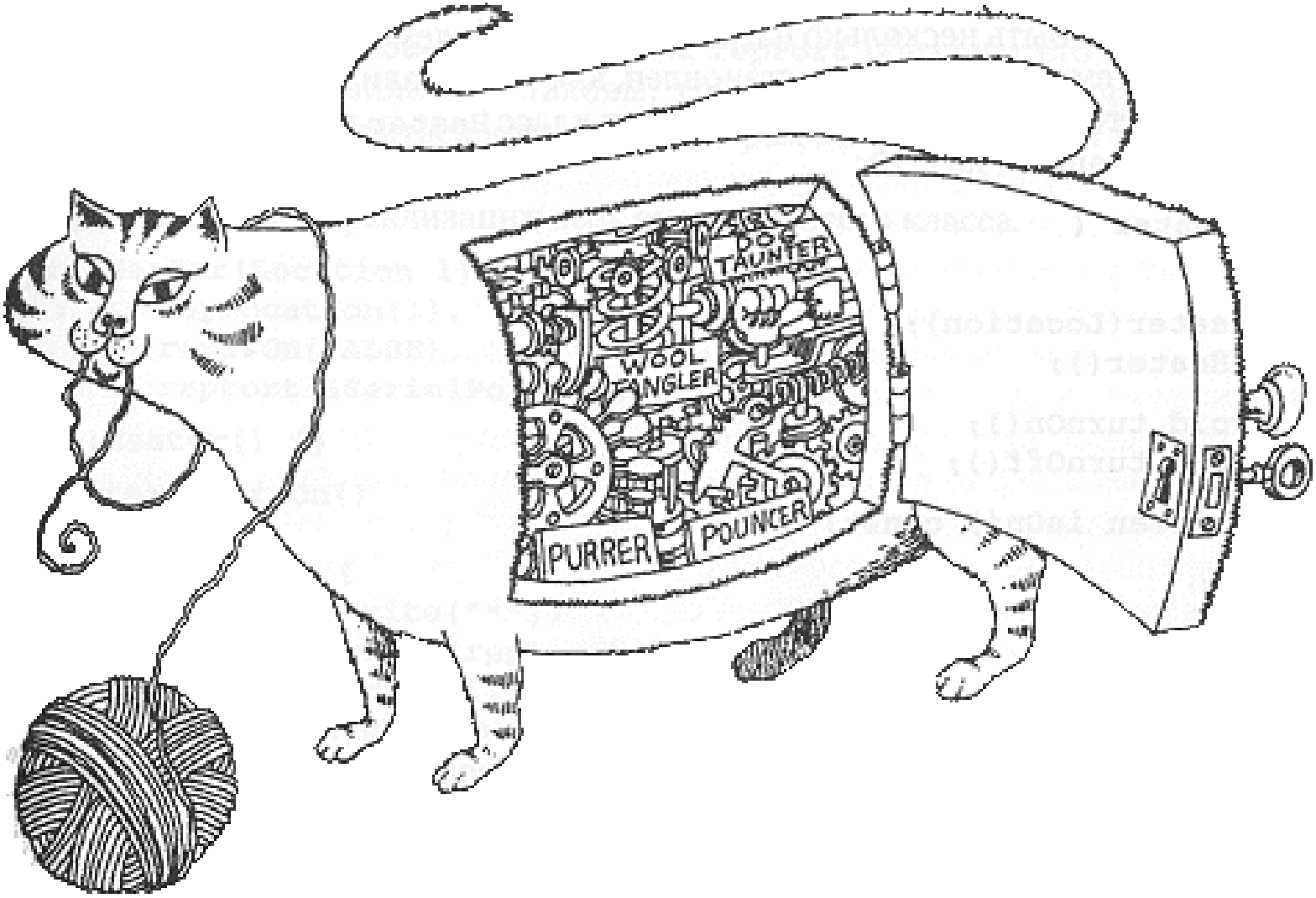
Абстрагирование

- *Абстрагирование* – процесс выделения абстракций в предметной области задачи. *Абстракция* – совокупность существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа. В соответствии с определением применяемая абстракция реального предмета существенно зависит от решаемой задачи. Современный уровень абстракции предполагает объединение всех свойств абстракции (как касающихся состояния анализируемого объекта, так и определяющих его поведение) в единую программную единицу некий абстрактный тип (*класс*).

Абстрагирование

- *По-другому*: Упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, которая подчеркивает детали, существенные для рассмотрения и использования, и опускает те, которые на данный момент несущественны.
- Выбор правильного набора абстракций для заданной предметной области представляет собой *главную задачу объектно-ориентированного проектирования*.

Инкапсуляция ***скрывает детали реализации объекта.***



Инкапсуляция (ограничение доступа)

- *Ограничение доступа* – сокрытие отдельных элементов реализации абстракции, не затрагивающих существенных характеристик ее как целого.
- Необходимость ограничения доступа предполагает разграничение двух частей в описании абстракции:
  - интерфейс – совокупность доступных извне элементов реализации абстракции (основные характеристики состояния и поведения);
  - реализация – совокупность недоступных извне элементов реализации абстракции (внутренняя организация абстракции и механизмы реализации ее поведения).
- Ограничение доступа в ООП позволяет разработчику:
  - выполнять конструирование системы поэтапно, не отвлекаясь на особенности реализации используемых абстракций;
  - легко модифицировать реализацию отдельных объектов, что в правильно организованной системе не потребует изменения других объектов.

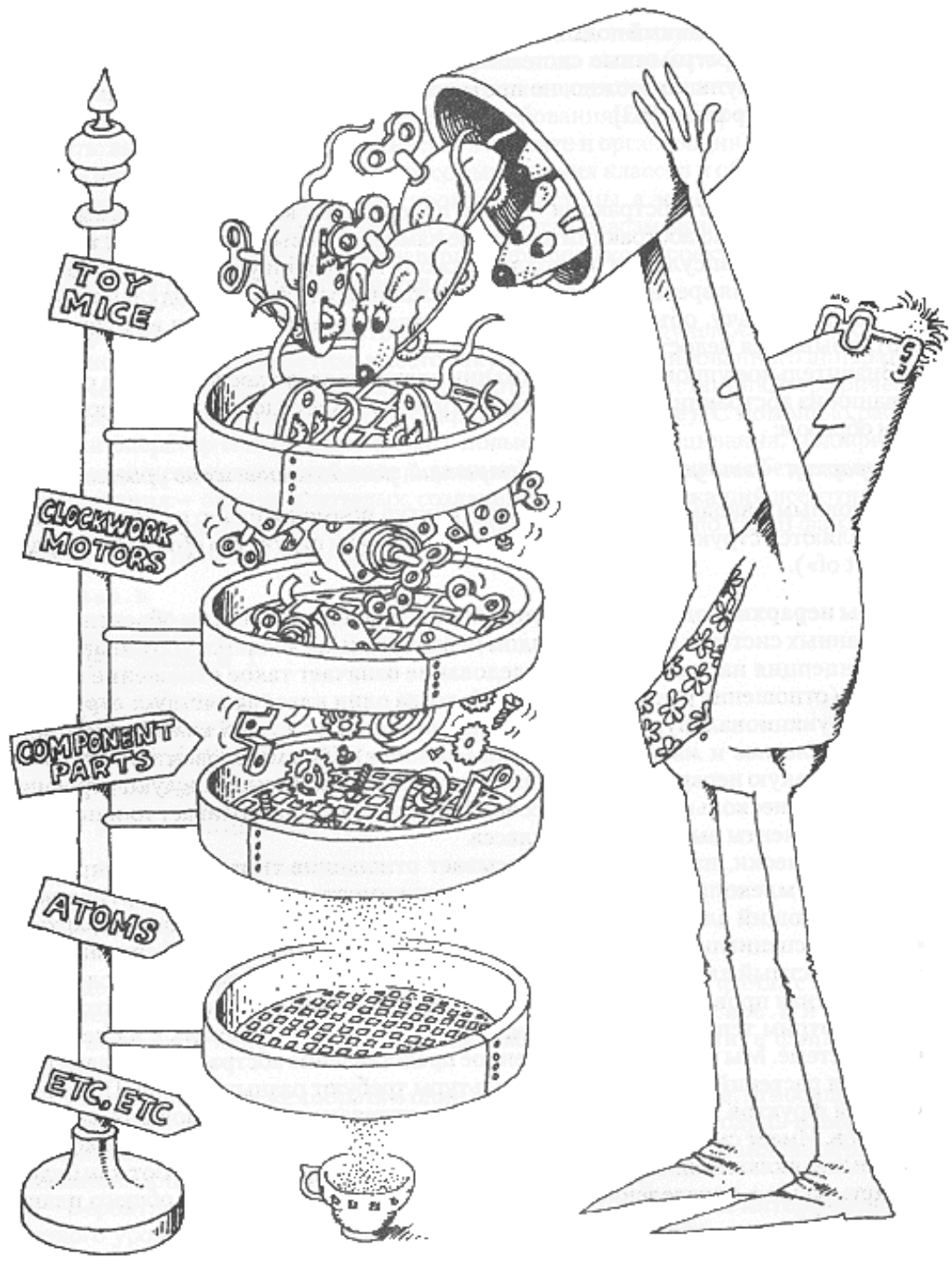
Инкапсуляция (ограничение доступа)

- Сочетание объединения всех свойств предмета (составляющих его состояния и поведения) в единую абстракцию и ограничения доступа к реализации этих свойств получило название *инкапсуляции*.
- Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством.
- Чаще всего инкапсуляция выполняется посредством скрытия информации, то есть маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта, и реализация его методов.
- Инкапсуляция, таким образом, определяет четкие границы между различными абстракциями. Возьмем для примера структуру растения: чтобы понять на верхнем уровне действие фотосинтеза, вполне допустимо игнорировать такие подробности, как функции корней растения или химию клеточных стенок.

Модульность позволяет хранить абстракции отдельно.

Модульность

- *Модульность* – принцип разработки программной системы, предполагающий реализацию ее в виде отдельных частей (модулей). При выполнении декомпозиции системы на модули желательно объединять логически связанные части, по возможности обеспечивая сокращение количества внешних связей между модулями. Принцип унаследован от модульного программирования, следование ему упрощает проектирование и отладку программы.
- В объектно-ориентированных языках классы и объекты составляют логическую структуру системы, они помещаются в модули, образующие физическую структуру системы. Это свойство становится особенно полезным, когда система состоит из многих сотен классов.



Абстракции образовывают иерархию.

Различают два вида иерархии:

«целое/часть»
«общее/частное»

Иерархичность

- *Иерархия* – ранжированная или упорядоченная система абстракций. Принцип иерархичности предполагает использование иерархии при разработке программных систем.
- По-другому: *Иерархия – это упорядочение абстракций, расположение их по уровням.*
- В ООП используются два вида иерархии.
- *Иерархия «целое/часть»* – показывает, что некоторые абстракции включены в рассматриваемую абстракцию как ее части, например, лампа состоит из цоколя, нити накаливания и колбы. Этот вариант иерархии используется в процессе разбиения системы на разных этапах проектирования (на логическом уровне – при декомпозиции предметной области на объекты, на физическом уровне – при декомпозиции системы на модули и при выделении отдельных процессов в мультипроцессной системе).
- *Иерархия «общее/частное»* – показывает, что некоторая абстракция является частным случаем другой абстракции, например, «обеденный стол – конкретный вид стола», а «столы – конкретный вид мебели». Используется при разработке структуры классов, когда сложные классы строятся на базе более простых путем добавления к ним новых характеристик и, возможно, уточнения имеющихся.

Наследование

- Один из важнейших механизмов ООП – *наследование* свойств в иерархии общее/частное.
- *Наследование* – такое соотношение между абстракциями, когда одна из них использует структурную или функциональную часть другой или нескольких других абстракций (соответственно простое и множественное наследование).

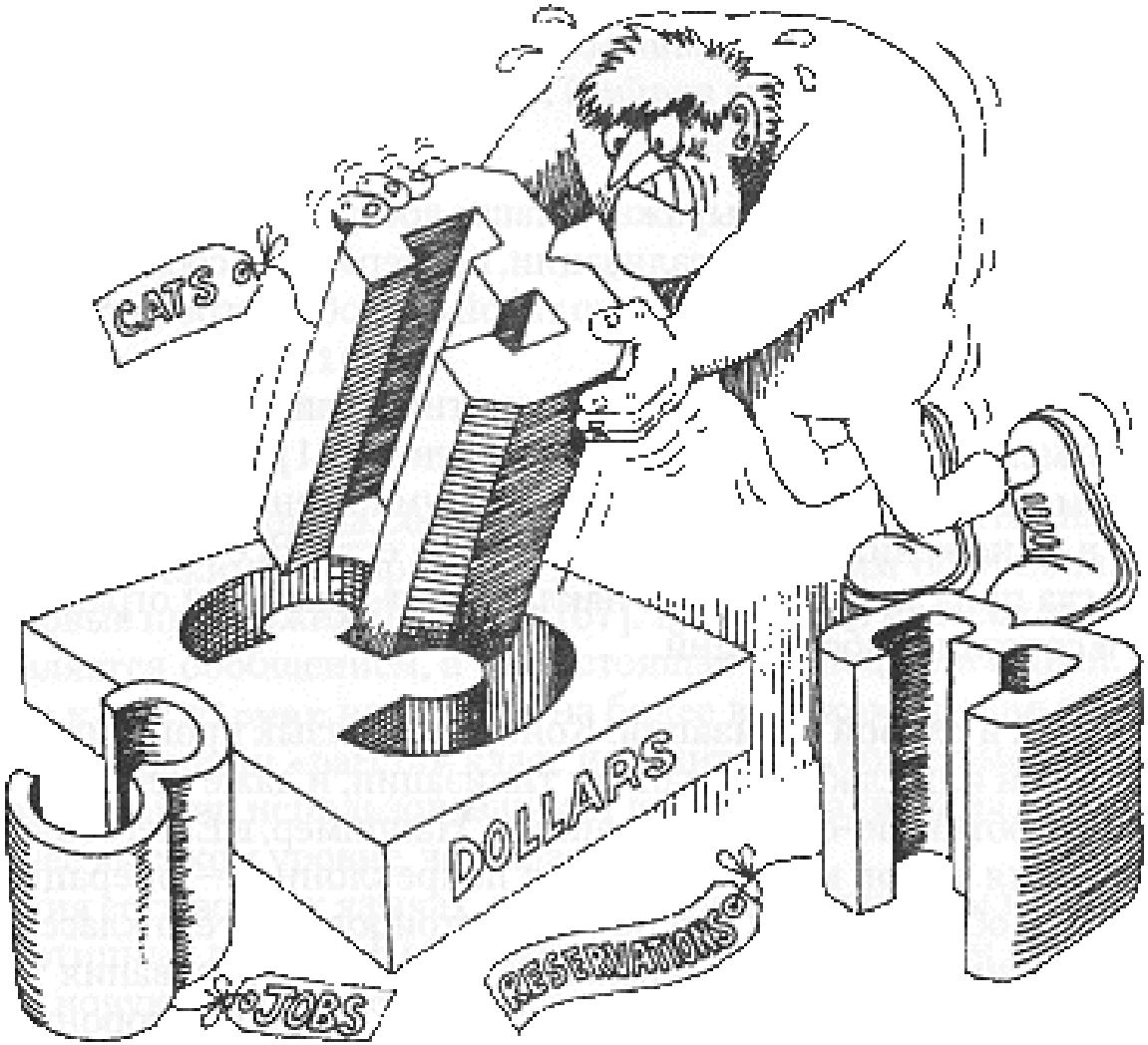
Агрегация, композиция

- Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём по умолчанию, агрегацией называют *агрегацию по ссылке*, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.

- ***Композиция*** — более строгий вариант агрегации. Известна также как агрегация по значению.

- Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

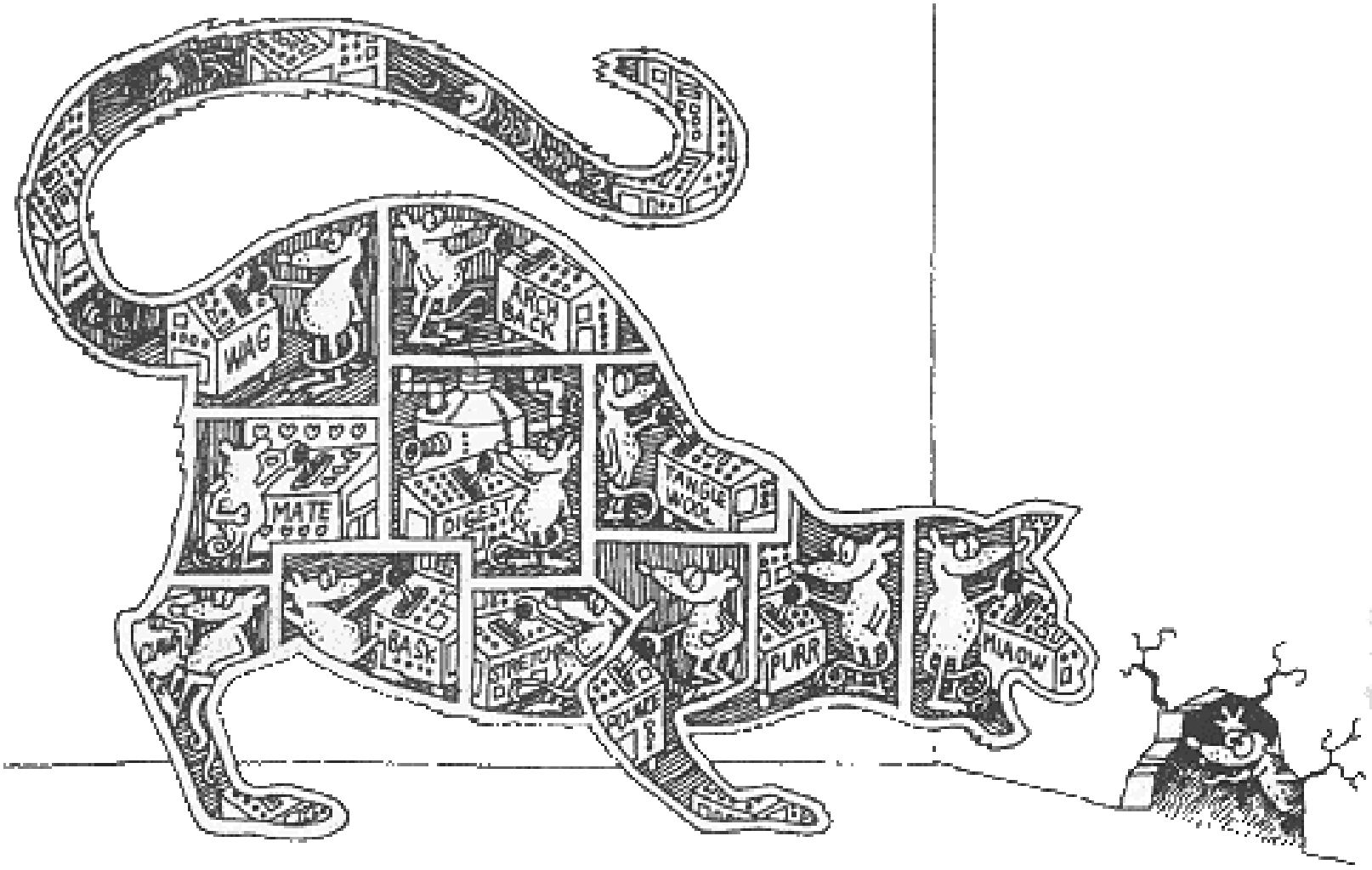
Строгая типизация предотвращает смешивание абстракций.



Типизация

- Типизация – ограничение, накладываемое на свойства объектов и препятствующее взаимозаменяемости абстракций различных типов (или сильно сужающее возможность такой замены). В языках с жесткой типизацией для каждого программного объекта (переменной, подпрограммы, параметра и т. д.) объявляется тип, который определяет множество операций над соответствующим программным объектом. Рассматриваемые далее языки программирования на основе Паскаля используют строгую, а на основе С –среднюю степень типизации.
- Использование принципа типизации обеспечивает:
  - раннее обнаружение ошибок, связанных с недопустимыми операциями над программными объектами (ошибки обнаруживаются на этапе компиляции программы при проверке допустимости выполнения данной операции над программным объектом);
  - упрощение документирования;
  - возможность генерации более эффективного кода.

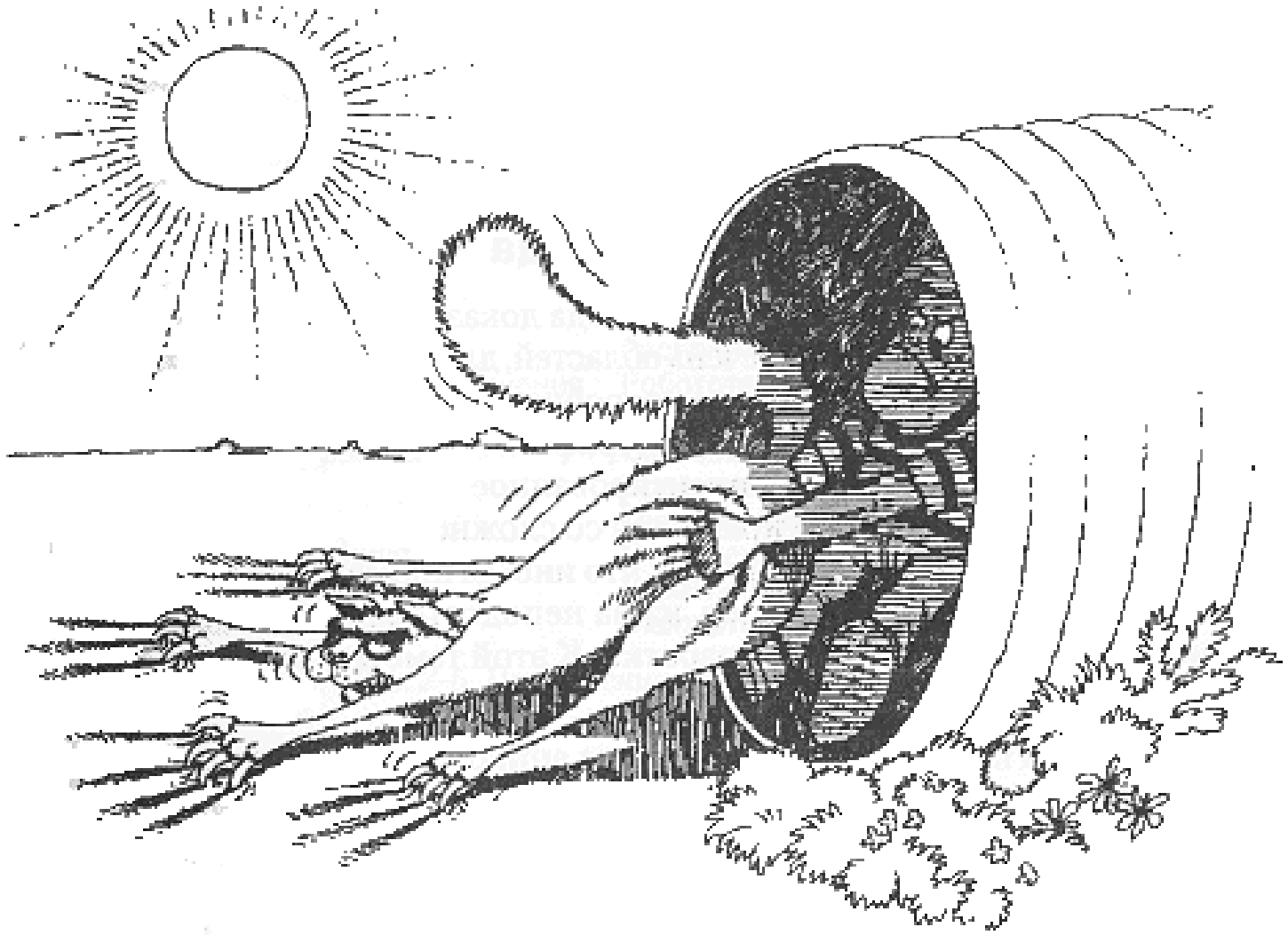
Параллелизм ***позволяет различным объектам действовать одновременно.***



Параллелизм

- *Параллелизм* – свойство нескольких абстракций одновременно находиться в активном состоянии, т.е. выполнять некоторые операции.
- Существует целый ряд задач, решение которых требует одновременного выполнения некоторых последовательностей действий. К таким задачам, например, относятся задачи автоматического управления несколькими процессами.
- Реальный параллелизм достигается только при реализации задач такого типа на многопроцессорных системах, когда имеется возможность выполнения каждого процесса отдельным процессором. Системы с одним процессором имитируют параллелизм за счет разделения времени процессора между задачами управления различными процессами.

***Сохраняемость* поддерживает состояние и класс объекта в пространстве и во времени.**



Устойчивость

- *Устойчивость* – свойство абстракции существовать во времени независимо от процесса, породившего данный программный объект, и/или в пространстве, перемещаясь из адресного пространства, в котором он был создан.
- Различают:
  - *временные объекты*, хранящие промежуточные результаты некоторых действий, например вычислений;
  - *локальные объекты*, существующие внутри подпрограмм, время жизни которых исчисляется от вызова подпрограммы до ее завершения;
  - *глобальные объекты*, существующие пока программа загружена в память;
  - *сохраняемые объекты*, данные которых хранятся в файлах внешней памяти между сеансами работы программы.

Главные признаки объектно-ориентированной модели

- **Абстрагирование**
- **Ограничение доступа (инкапсуляция)**
- **Модульность**
- **Иерархичность**
- **Полиморфизм**
- Типизация
- Параллелизм
- Устойчивость (сохраняемость)

Основы ООП

- **Класс** (class) - элемент программы, который описывает какой-то тип данных. Класс описывает шаблон для создания объектов, как правило, указывает переменные этого объекта и действия, которые можно выполнять применимо к объекту.
- **Экземпляр класса** (instance) - объект, который является представителем класса.
- **Метод** (method) - функция, которая определена внутри класса и описывает какое-то действие, которое поддерживает класс
- **Переменная экземпляра** (instance variable, а иногда и instance attribute) - данные, которые относятся к объекту
- **Переменная класса** (class variable) - данные, которые относятся к классу и разделяются всеми экземплярами класса
- **Атрибут экземпляра** (instance attribute) - переменные и методы, которые относятся к объектам (экземплярам) созданным на основании класса. У каждого объекта есть своя копия атрибутов.

Создание класса

```
1  class MyClass:
2      'Краткое описание класса (необязательно)'
3      pass
```

- Имена классов: в Python принято писать имена классов в формате CamelCase.
- Для создания экземпляра класса, надо вызвать класс:

```
1  cl1 = MyClass()
2  print(cl1)
3  print(cl1.__doc__)
```

```
<__main__.MyClass object at 0x055C12F8>
Краткое описание класса (необязательно)
```

- В теле класса допускается объявление атрибутов, методов и конструктора.

Наполнение класса

Атрибут:

- Атрибут – это элемент класса. Например, у прямоугольника таких 2: ширина (`width`) и высота (`height`).

Метод:

- Метод класса напоминает классическую функцию, но на самом деле – это функция класса. Для использования ее необходимо вызывать через объект.
- Первый параметр метода всегда `self` (ключевое слово, которое ссылается на сам класс).

Наполнение класса

Конструктор:

- Конструктор – уникальный метод класса, который называется `__init__`.
- Первый параметр конструктора во всех случаях `self` (ключевое слово, которое ссылается на сам класс).
- Конструктор нужен для создания объекта.
- Конструктор передает значения аргументов свойствам создаваемого объекта.
- В одном классе всегда только один конструктор.
- Если класс определяется не конструктором, Python предположит, что он наследует конструктор родительского класса.

Класс Rectangle

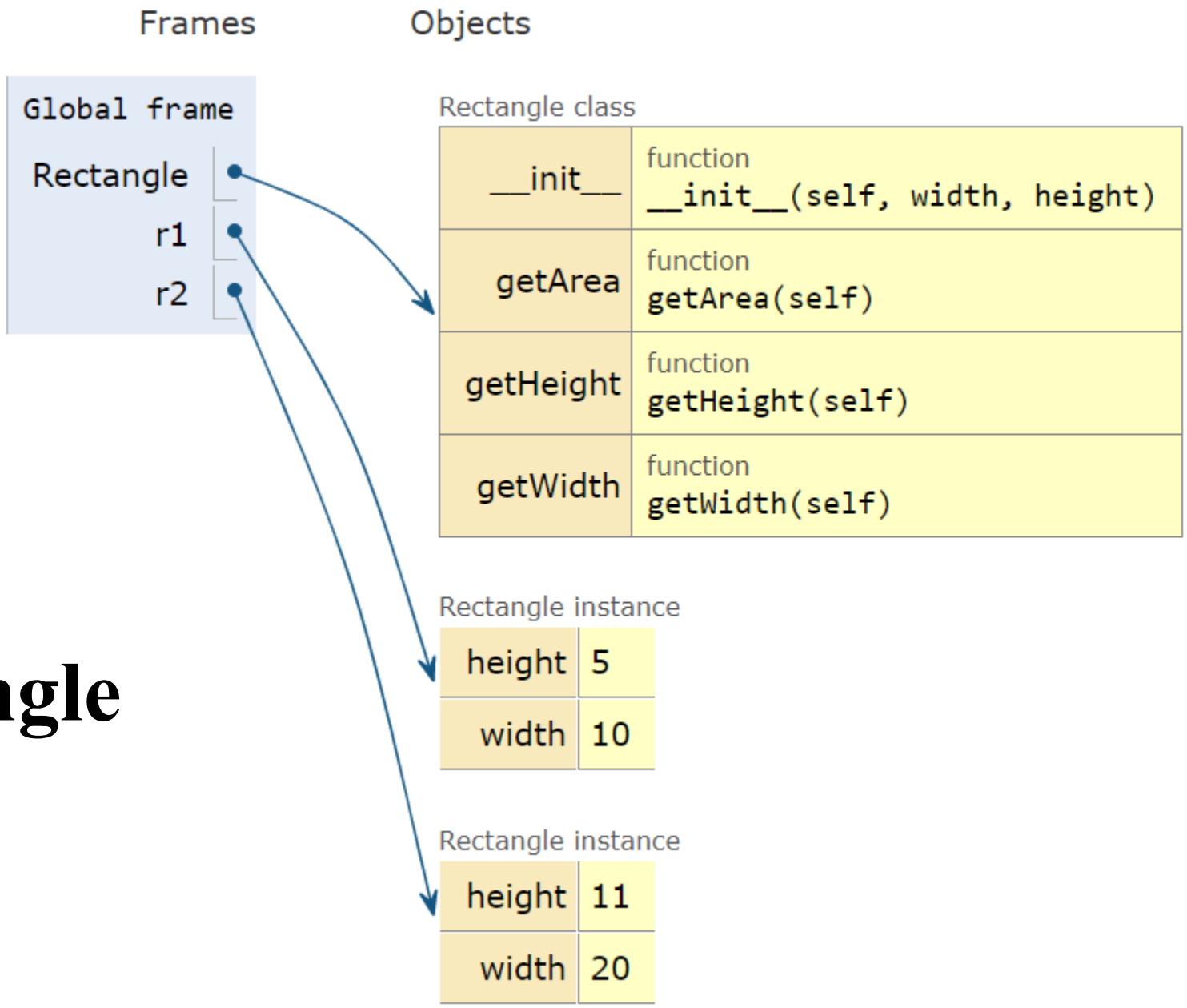
```
1  class Rectangle :
2      'Класс Rectangle'
3      # Способ создания объекта (конструктор)
4      def __init__(self, width, height):
5          self.width= width
6          self.height = height
7
8      def getWidth(self):
9          return self.width
10
11     def getHeight(self):
12         return self.height
13
14     # Метод расчета площади.
15     def getArea(self):
16         return self.width * self.height
```

Класс Rectangle

```
1  r1 = Rectangle(10,5)
2  r2 = Rectangle(20,11)
3
4  print("r1.width = ", r1.width)
5  print("r1.height = ", r1.height)
6  print("r1.getWidth() = ", r1.getWidth())
7  print("r1.getArea() = ", r1.getArea())
8
9  print("-----------------")
10
11 print("r2.width = ", r2.width)
12 print("r2.height = ", r2.height)
13 print("r2.getWidth() = ", r2.getWidth())
14 print("r2.getArea() = ", r2.getArea())
```

```
r1.width =  10
r1.height =  5
r1.getWidth() =  10
r1.getArea() =  50
-----------------
r2.width =  20
r2.height =  11
r2.getWidth() =  20
r2.getArea() =  220
```

Класс Rectangle



Конструктор с аргументами по умолчанию

- В других языках программирования конструкторов может быть несколько. В Python – только один. Но этот язык разрешает задавать значение по умолчанию.
- Все требуемые аргументы нужно указывать до аргументов со значениями по умолчанию.

Конструктор с аргументами по умолчанию

```
class Person:
    # Параметры возраста и пола имеют значение по умолчанию.
    def __init__(self, name, age=1, gender="Male"):
        self.name = name
        self.age = age
        self.gender= gender

    def showInfo(self):
        print("Name: ", self.name)
        print("Age: ", self.age)
        print("Gender: ", self.gender)
```

Конструктор с аргументами по умолчанию

```
p1 = Person("Mary", 21, "Female")
p1.showInfo()
print(" --------------- ")

# возраст по умолчанию, пол.
p2 = Person( "Katy" )
p2.showInfo()

print(" --------------- ")

# Пол по умолчанию.
p3 = Person("Michael", 37)
p3.showInfo()
```

```
Name:  Mary
Age:  21
Gender:  Female
 ---------------
Name:  Katy
Age:  1
Gender:  Male
 ---------------
Name:  Michael
Age:  37
Gender:  Male
```

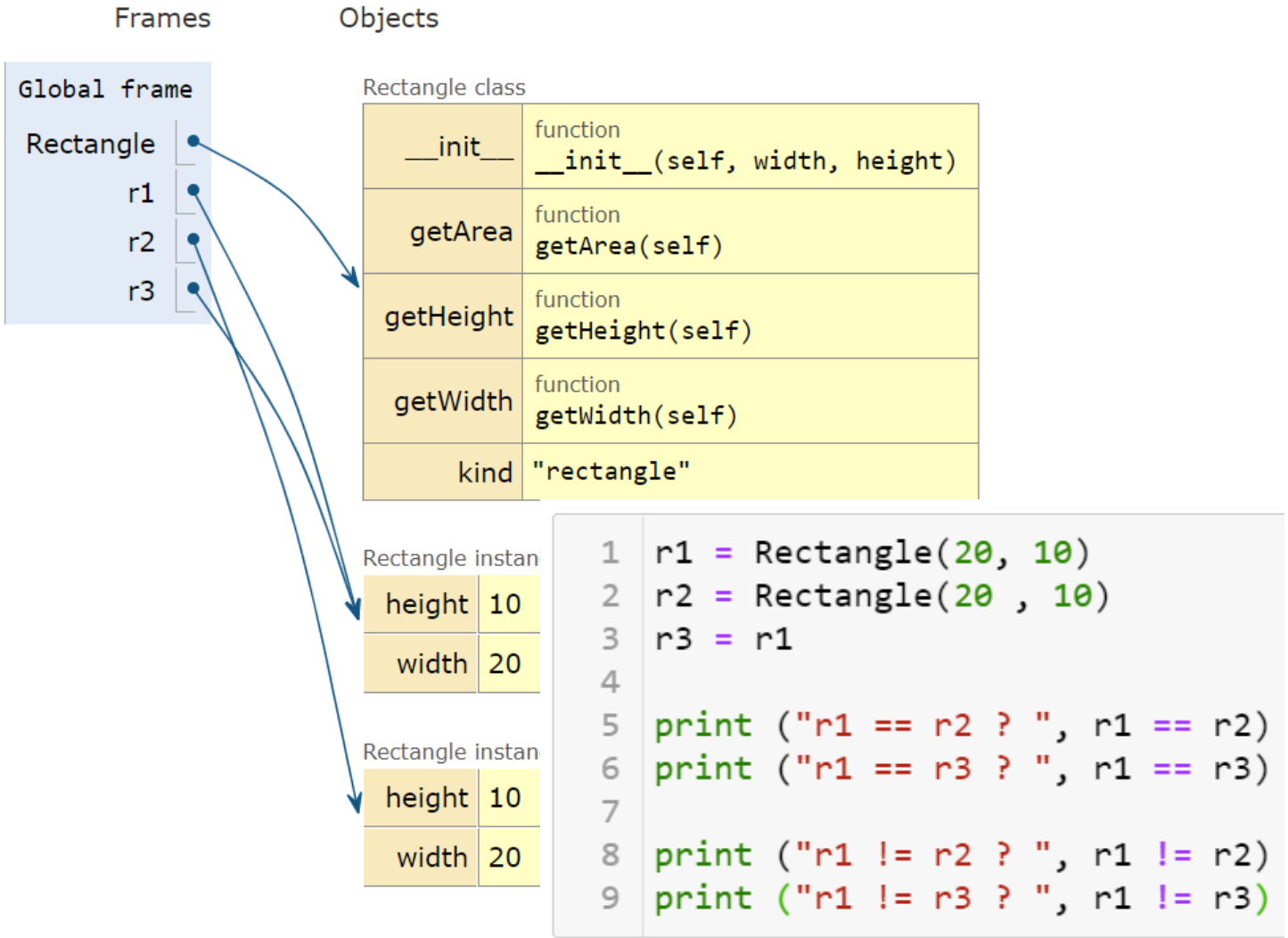
Сравнение объектов

- Оператор == нужен, чтобы узнать, ссылаются ли два объекта на одно и то же место в памяти. Он вернет `True`, если это так. Оператор `!=` вернет `True`, если сравнить 2 объекта, которые ссылаются на разные места в памяти.

```
1  r1 = Rectangle(20, 10)
2  r2 = Rectangle(20 , 10)
3  r3 = r1
4
5  print ("r1 == r2 ? ", r1 == r2)
6  print ("r1 == r3 ? ", r1 == r3)
7
8  print ("r1 != r2 ? ", r1 != r2)
9  print ("r1 != r3 ? ", r1 != r3)
```

```
r1 == r2 ?  False
r1 == r3 ?  True
r1 != r2 ?  True
r1 != r3 ?  False
```

Сравнение объектов



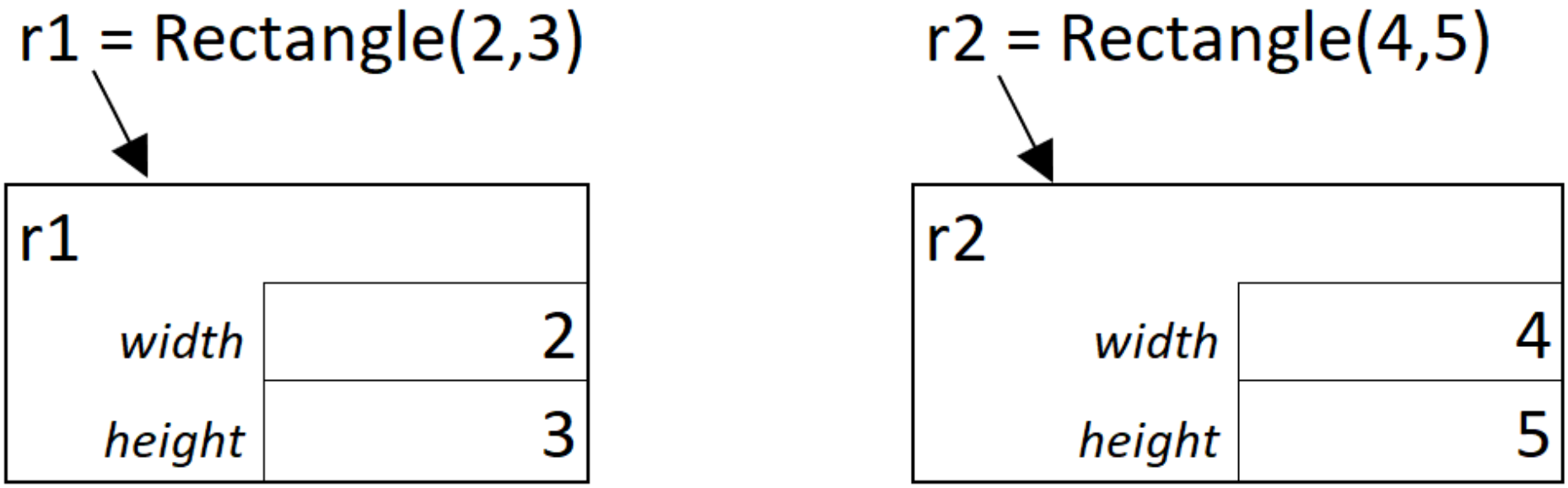

```
1  r1 = Rectangle(20, 10)
2  r2 = Rectangle(20 , 10)
3  r3 = r1
4
5  print ("r1 == r2 ? ", r1 == r2)
6  print ("r1 == r3 ? ", r1 == r3)
7
8  print ("r1 != r2 ? ", r1 != r2)
9  print ("r1 != r3 ? ", r1 != r3)
```

```
r1 == r2 ?  False
r1 == r3 ?  True
r1 != r2 ?  True
r1 != r3 ?  False
```

Атрибуты

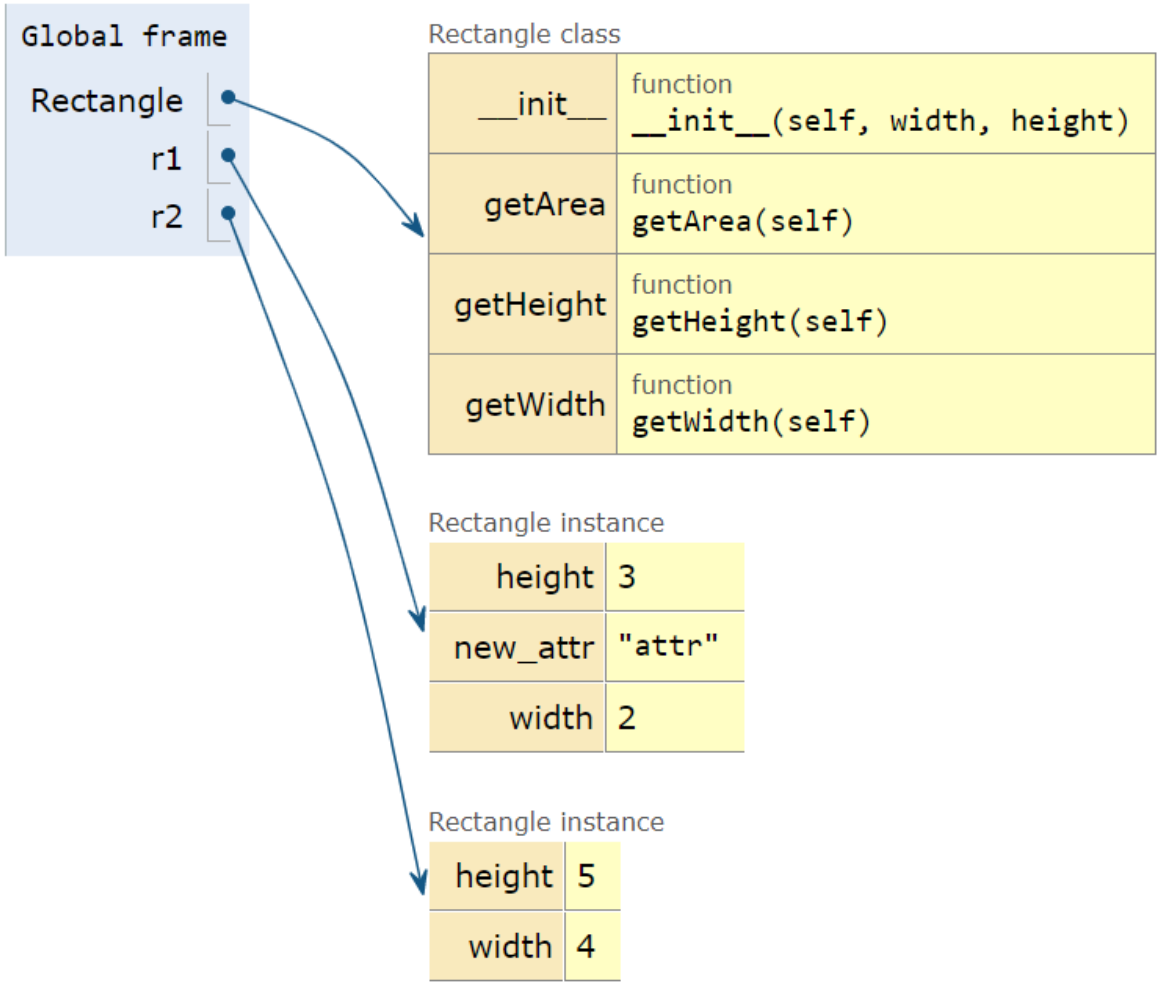
- В Python есть два похожих понятия, которые на самом деле отличаются:
  - **Атрибуты**
  - **Переменные класса**
- Объекты, созданные одним и тем же классом, будут занимать разные места в памяти, а их атрибуты с «одинаковыми именами» - ссылаться на разные адреса.

Атрибуты


r1 = Rectangle(2,3)
r1
width
2
height
3
r2 = Rectangle(4,5)
r2
width
4
height
5

Атрибуты

- Python умеет создавать новые атрибуты для уже существующих объектов.




```
18 r1 = Rectangle(2,3)
19 r2 = Rectangle(4,5)
20 r1.new_attr = "attr"
21 print(r2.new_attr)
```

```
---------------------------
AttributeError
<ipython-input-31-92524e3cf
     19 r2 = Rectangle(4,5)
     20 r1.new_attr = "attr
---> 21 print(r2.new_attr)

AttributeError: 'Rectangle' object has no attribute 'new_attr'
```

Встроенные функции для доступа к аттрибутам

Функция	Описание
getattr (obj, name[,default])	Возвращает значение атрибута или значение по умолчанию, если первое не было указано
hasattr (obj, name)	Проверяет атрибут объекта — был ли он передан аргументом «name»
setattr (obj, name, value)	Задает значение атрибута. Если атрибута не существует, создает его
delattr (obj, name)	Удаляет атрибут

Встроенные атрибуты класса

Атрибут	Описание
__dict__	Предоставляет данные о классе коротко и доступно, в виде словаря
__doc__	Возвращает строку с описанием класса, или None, если значение не определено
__class__	Возвращает объект, содержащий информацию о классе с массой полезных атрибутов, включая атрибут __name__
__module__	Возвращает имя «модуля» класса или __main__, если класс определен в выполняемом модуле.

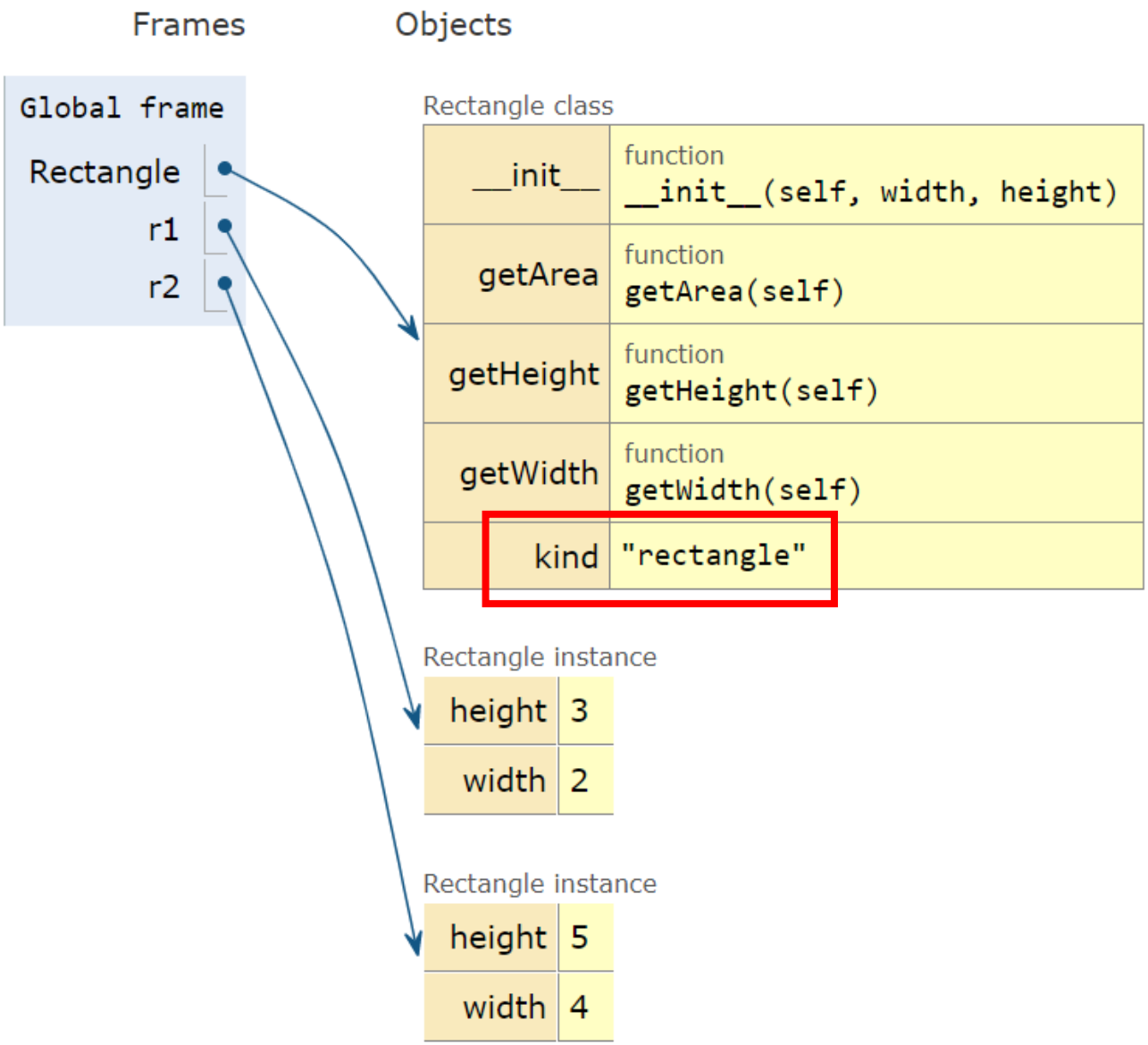
Переменные класса

- Переменные класса в Python — это то же самое, что Field в других языках, таких как Java или C#. Получить к ним доступ можно только с помощью имени класса или объекта.
- Для получения доступа к переменной класса лучше все-таки использовать имя класса, а не объект. Это поможет не путать «переменную класса» и атрибуты.
- **Переменные экземпляра класса** предназначены для данных, уникальных для каждого экземпляра класса, а **переменные класса** (атрибуты данных класса, аналог статических полей класса в C++) - для атрибутов и методов, общих для всех экземпляров класса.

Переменные класса

```
1  class Rectangle :
2      kind = 'rectangle'
3      'Класс Rectangle'
4      # Способ создания объекта (конструктор)
5      def __init__(self, width, height):
6          self.__width= width
7          self.__height = height
8
9      def getWidth(self):
10         return self.width
11
12     def getHeight(self):
13         return self.height
14
15     # Метод расчета площади.
16     def getArea(self):
17         return self.width * self.height
18
19 r1 = Rectangle(2,3)
20 r2 = Rectangle(4,5)
```

Переменные класса



Составляющие класса или объекта

- В Python присутствует функция dir, которая выводит список всех методов, атрибутов и переменных класса или объекта.

```
1 print(dir(Rectangle))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex_
_', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'getArea', 'getHeight', 'getWidth']
```

```
1 print(dir(r1))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex_
_', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'getArea', 'getHeight', 'getWidth',
'height', 'new_attr', 'width']
```

Режимы доступа

- При создании класса Rectangle были созданные обычные атрибуты width и height, которые можно изменять из вне, что противоречит принципу инкапсуляции.

```
1  r1.width = 20
2  r2.height = 50
3  print(r1.__dict__)
4  print(r2.__dict__)
```

```
{'width': 20, 'height': 3}
{'width': 4, 'height': 50}
```

Режимы доступа

- Чтобы программист не мог произвольным образом задавать атрибуты их следует «закрывать» от вмешательства извне. В Python возможны следующие варианты доступа к данным:

- <имя атрибута> (без одного или двух подчеркиваний вначале) – публичное свойство (**public**);
- _<имя атрибута> (с одним подчеркиванием) – режим доступа **protected** (можно обращаться только внутри класса и во всех его дочерних классах)
- __<имя атрибута> (с двумя подчеркиваниями) – режим доступа **private** (можно обращаться только внутри класса).

Режимы доступа

- Создадим два новых закрытых атрибута в классе Rectangle

```
1  class Rectangle :
2      kind = 'rectangle'
3      'Класс Rectangle'
4      # Способ создания объекта (конструктор)
5      def __init__(self, width, height):
6          self.__width= width
7          self.__height = height
8
9      def setWidth(self, w):
10         self.__width = w
11
12     def setHeight(self, h):
13         self.__height = h
14
15     def getWidth(self):
16         return self.__width
17
18     def getHeight(self):
19         return self.__height
20
21     # Метод расчета площади.
22     def getArea(self):
23         return self.width * self.height
```

Режимы доступа

- При выводе атрибутов класса мы увидим:

```
1 print(r1.__dict__)
2 print(r2.__dict__)
```

```
{'_Rectangle__width': 2, '_Rectangle__height': 3}
{'_Rectangle__width': 4, '_Rectangle__height': 5}
```

- Однако при привычном обращении к атрибутам класса через точку создаются новые атрибуты.

```
1 r1.width = 20
2 r2.height = 50
3 print(r1.__dict__)
4 print(r2.__dict__)
```

```
{'_Rectangle__width': 2, '_Rectangle__height': 3, 'width': 20}
{'_Rectangle__width': 4, '_Rectangle__height': 5, 'height': 50}
```

Режимы доступа

- Теперь напрямую к атрибуту извне не представляется возможным:

```
1 print(r1.__width)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-53-4eff55138225> in <module>
----> 1 print(r1.__width)

AttributeError: 'Rectangle' object has no attribute '__width'
```

Сеттеры и геттеры

- Теперь для переопределения закрытых атрибутов класса объявляют специальные методы, которые, обычно, начинаются с префикса **set** (что означает задать, установить) и далее, какое-либо имя.

```
1  class Rectangle :
2      kind = 'rectangle'
3      'Класс Rectangle'
4      # Способ создания объекта (конструктор)
5      def __init__(self, width, height):
6          self.__width= width
7          self.__height = height
8
9      def setWidth(self, w):
10         self.__width = w
11
12     def setHeight(self, h):
13         self.__height = h
14
15     def getWidth(self):
16         return self.__width
17
18     def getHeight(self):
19         return self.__height
20
21     # Метод расчета площади.
22     def getArea(self):
23         return self.width * self.height
```

Геттеры и сеттеры

```
1 r1.setWidth(20)
2 r1.setHeight(30)
3 print(r1.__dict__)
```

```
{'_Rectangle__width': 20, '_Rectangle__height': 30}
```

- Для чтения закрытых атрибутов используются методы с префиксом get. В нашем классе Rectangle они были реализованы ранее (см. предыдущий слайд)
- Такие методы в ООП называются сеттерами и геттерами. Их назначение не только передавать значения между приватными атрибутами класса, но и проверять их корректность. Например, в нашем случае мы могли бы проверить на этапе получения новых значений width и height, являются ли они положительными.

Создание объектов-свойств

- Однако, пользоваться на практике напрямую геттерами и сеттерами бывает не всегда удобно. Большего изящества кода можно добиться, используя так называемые объекты-свойства (property).
- На основе геттеров и сеттеров

```
def __setW(self, w):
    self.__width = w

def __getW(self):
    return self.__width
```

Определим свойство Width с помощью класса Property

```
Width = property(__getW, __setW)
```

Создание объектов-свойств

- Пользоваться свойствами удобнее, нежели вызывать специальные методы:

```
23 r1 = Rectangle(1, 2)
24 print(r1.Width)
25 r1.Width = 100
26 print(r1.__dict__)
```

```
1
{'_Rectangle__width': 100, '_Rectangle__height': 2}
```

Создание объектов-свойств

- В сеттер добавим ограничение с помощью закрытой функции __checkValue:

```
def __checkValue(x):
    if isinstance(x, int) or isinstance(x, float):
        return True
    return False

def __setW(self, w):
    if Rectangle.__checkValue(w):
        self.__width = w
    else:
        raise ValueError
```

- Теперь при попытке присвоить неправильное значение возникает исключение:

```
31 r1 = Rectangle(1, 2)
32 print(r1.Width)
33 try:
34     r1.Width = "100"
35     print(r1.__dict__)
36 except ValueError:
37         print("Неверный формат данных")
```

Простое наследование

- Общий принцип наследования в ООП: мы можем взять некий класс и на его основе создать новый, дочерний, изменив и расширив функционал базового класса.
- В Python 3 любой создаваемый класс автоматически является дочерним по отношению к базовому object:

```
1  class Point:
2      def __init__(self, x = 0, y = 0):
3          self.__x = x
4          self.__y = y
```

```
1  print(issubclass(Point, object))
```

```
True
```

Простое наследование

- Создавать новые объекты на базе существующих намного проще и быстрее, нежели создавать их с нуля.
- Допустим мы имеем класс Rectangle:

```
1  class Rectangle :
2      'Класс Rectangle'
3      # Способ создания объекта (конструктор)
4      def __init__(self, width, height):
5          self.__width= width
6          self.__height = height
7
8      def __str__(self):
9          return f"прямоугольник:\nширина: {self.__width}, \nвысота: {self.__height}."
10
11 r = Rectangle(1,2)
12 print(str(r))
```

```
прямоугольник:
ширина: 1,
высота: 2.
```

Простое наследование

- Создать класс Box будет быстрее и проще, используя класс Rectangle, т.е. используя простое наследование (родитель ← дочерний класс):

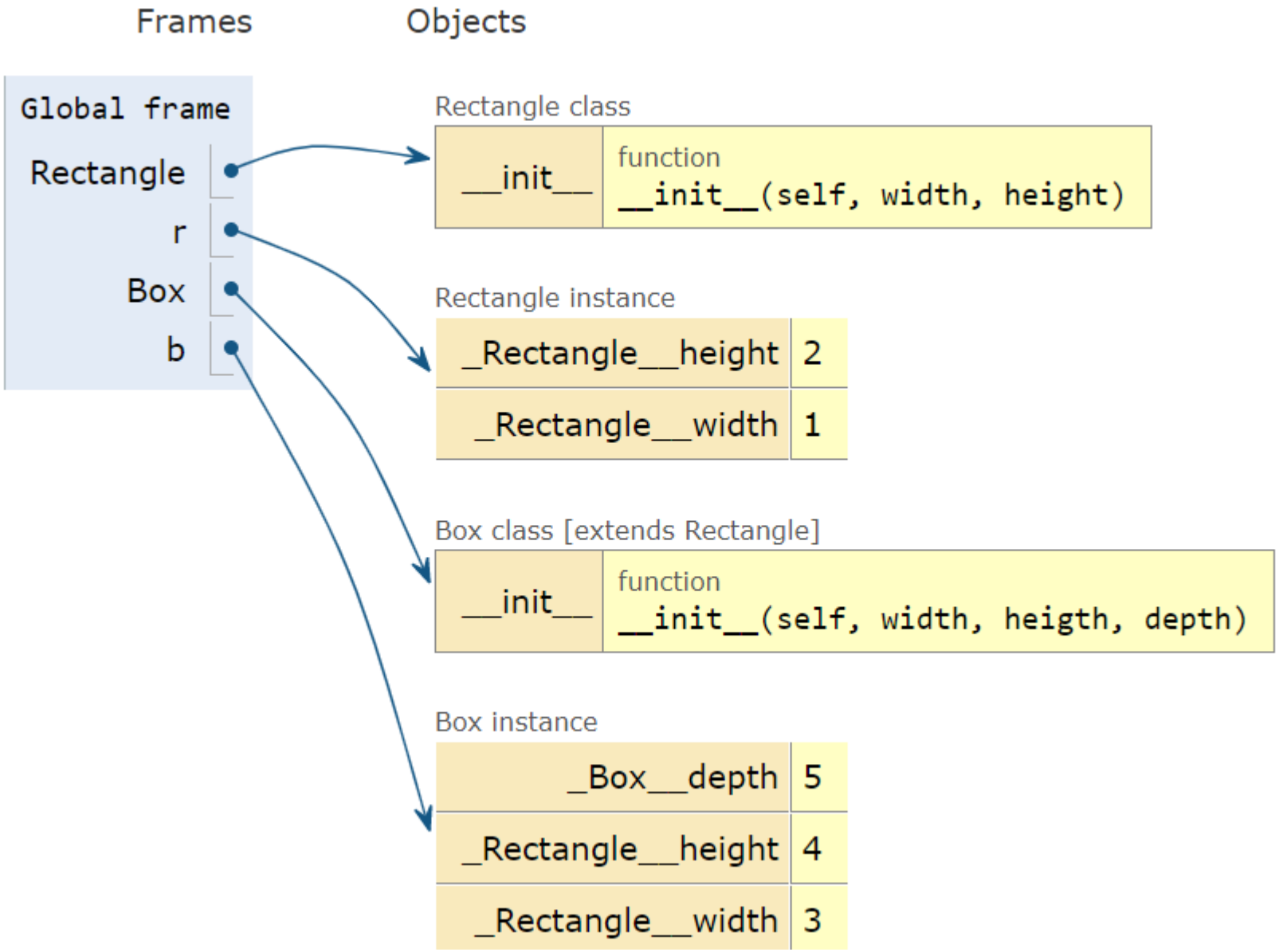
```
1  class Rectangle :
2      'Класс Rectangle'
3      # Способ создания объекта (конструктор)
4      def __init__(self, width, height):
5          self.__width= width
6          self.__height = height
7
8  r = Rectangle(1,2)
9  print(r.__dict__)
```

```
{'_Rectangle__width': 1, '_Rectangle__height': 2}
```

```
1  class Box(Rectangle):
2      def __init__(self, width, heigth, depth):
3          super().__init__(width, heigth)
4          self.__depth = depth
5
6  b = Box(3,4,5)
7  print(b.__dict__)
```

```
{'_Rectangle__width': 3, '_Rectangle__height': 4, '_Box__depth': 5}
```

Простой полиморфизм



Простое наследование

- class A:

  # определение конструктора родительского класса

  …

- class B(A):

  # определение конструктора дочернего класса

  …

- вместо явного указания имени базового класса, следует вызвать специальную функцию **super()**, которая в правильном порядке будет перебирать вышестоящие классы

Полиморфизм

- *Полиморфизм* – это возможность работы с совершенно разными объектами языка Python единым образом.
- Создадим список геометрических объектов и выведем их содержимое на экран:

```
1 lst = [Rectangle(1,2), Box(3,4,5), Rectangle(6,7)]
2 for item in lst:
3     print(str(item))
```

```
прямоугольник:
ширина: 1
высота: 2

параллелепипед:
ширина: 3
высота: 4
глубина: 5

прямоугольник:
ширина: 6
высота: 7
```

Полиморфизм

- Для этого необходимо предусмотреть определение str() в базовом класса и переопределение его в классе-потомке.

```
1  class Rectangle :
2      'Класс Rectangle'
3      # Способ создания объекта (конструктор)
4      def __init__(self, width, height):
5          self.__width= width
6          self.__height = height
7
8      def __str__(self):
9          return f"прямоугольник:\nширина: {self.__width} \
10         \nвысота: {self.__height}\n"
11
12 r = Rectangle(1,2)
13 print(str(r))
```

```
прямоугольник:
ширина: 1
высота: 2
```

Полиморфизм

- При переопределении метода в классе-потомке при обращении к закрытым атрибутам класса-родителя нужно обращаться следующим образом: object._className__attrName

```
class Box(Rectangle):
    # Переопределение конструктора дочернего класса
    def __init__(self, width, heigth, depth):
        super().__init__(width, heigth)
        self.__depth = depth

    def __str__(self):
        return f"параллелепипед:\nширина: {self._Rectangle__width} \
        \nвысота: {self._Rectangle__height}\nглубина: {self.__depth}\n"

b = Box(3,4,5)
print(str(b))
```

```
параллелепипед:
ширина: 3
высота: 4
глубина: 5
```

Полиморфизм

- Благодаря такому средству разработки программ Python «понимает» объект какого класса хранится в коллекции и соответственно из этого класса берет метод str().
- Это и есть полиморфизм в действии, когда к разным объектам происходит обращение по одному и тому же имени метода и на выходе получаем разное поведение этой функции.