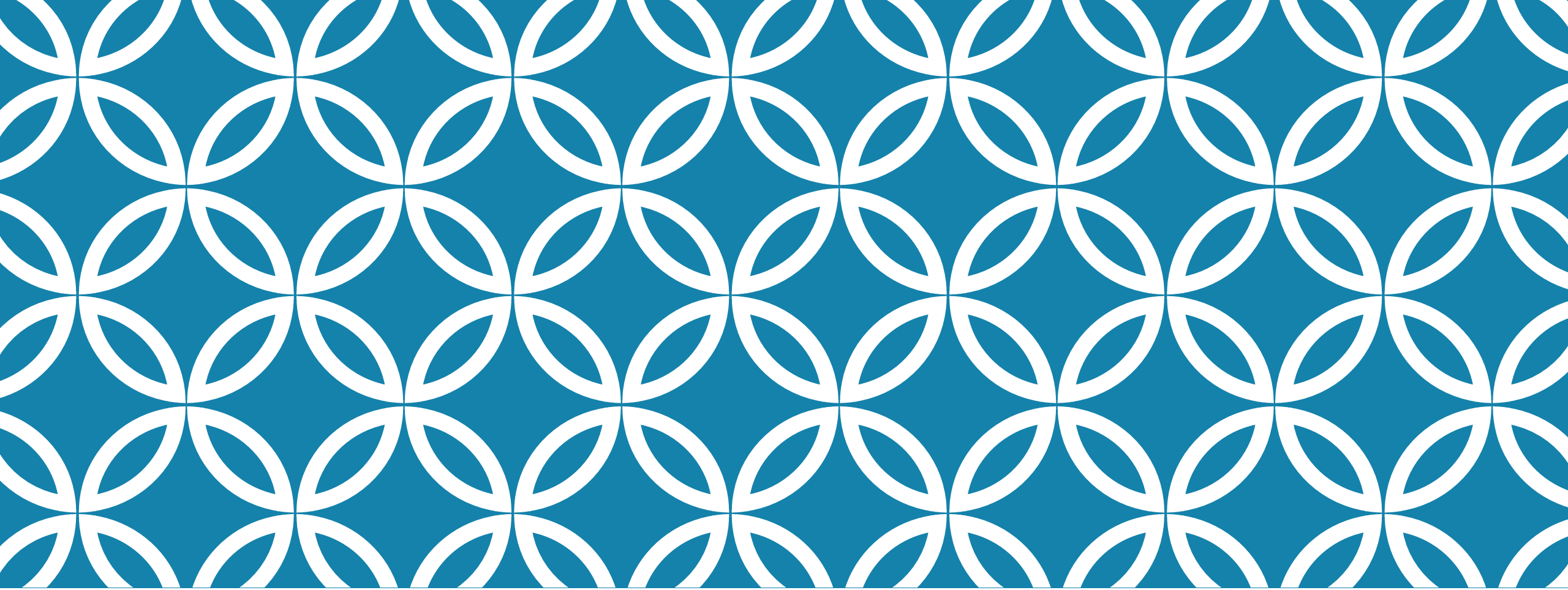


ООП

Классы

Объектно-ориентированный подход строится на следующих принципах:

1. Полиморфизм: в разных объектах одна и та же операция может выполнять различные функции. Слово «полиморфизм» имеет греческую природу и означает «имеющий многие формы». Простым примером полиморфизма может служить функция count(), выполняющая одинаковое действие для различных типов обьектов: 'abc'.count('a') и [1, 2, 'a'].count('a'). Оператор плюс полиморфичен при сложении чисел и при сложении строк.
2. Инкапсуляция: можно скрыть ненужные внутренние подробности работы объекта от окружающего мира. Это второй основной принцип абстракции. Он основан на использовании атрибутов внутри класса. Атрибуты могут иметь различные состояния в промежутках между вызовами методов класса, вследствие чего сам объект данного класса также получает различные состояния — state.

3. Наследование: можно создавать специализированные классы на основе базовых. Это позволяет нам избегать написания повторного кода.
4. Композиция: объект может быть составным и включать в себя другие объекты.

Объектно-ориентированный подход хорош там, где проект подразумевает долгосрочное развитие, состоит из большого количества библиотек и внутренних связей.

Наиболее важные особенности классов в питоне:

1. Множественное наследование.
2. Производный класс может переопределить любые методы базовых классов.
3. В любом месте можно вызвать метод с тем же именем базового класса.
4. Все атрибуты класса в питоне по умолчанию являются public, т.е. доступны отовсюду; все методы — виртуальные, т.е. перегружают базовые.

Python полностью объектно-ориентирован, то есть вы можете определять свои собственные классы, наследовать новые классы от своих или встроенных классов, и создавать экземпляры классов, которые уже определили.

Определить класс в Python просто. Также как и в случае с функциями, раздельное объявление интерфейса не требуется. Вы просто определяете класс и начинаете программировать. Определение класса в Python начинается с зарезервированного слова class, за которым следует имя (идентификатор) класса.

Формально, это все, что необходимо, в случае, когда класс не должен быть унаследован от другого класса.

```
class PapayaWhip:
    pass
```

1. Определенный выше класс имеет имя PapayaWhip и не наследует никакой другой класс. В именах классов каждое слово обычно пишется с большой буквы, но это не требование, а лишь соглашение.
2. Каждая строка в определении класса имеет отступ, также как и в случае с функциями, оператором условного перехода if, циклом for или любым другим блоком кода. Первая строка без отступа находится вне блока class.

Класс PapayaWhip не содержит определений методов или атрибутов, но с точки зрения синтаксиса, тело класса не может оставаться пустым. В таких случаях используется оператор pass.

В языке Python pass — зарезервированное слово, которое говорит интерпретатору: «идем дальше, здесь ничего нет». Это оператор не делающий ровным счетом ничего, но тем не менее являющийся удобным решением, когда вам нужно сделать заглушку для функции или класса.

Выражение *pass* в языке Python аналог пустого множества или фигурных скобок в языках Java или C++.

Многие классы наследуются от других классов, но не этот. Многие классы определяют свои методы, но не этот. Класс в Python не обязан иметь ничего, кроме имени. В частности, людям знакомым с C++ может показаться странным, что у класса в Python отсутствуют в явном виде конструктор и деструктор.

Несмотря на то, что это не является обязательным, класс в Python может иметь нечто, похожее на конструктор: метод *init*().

```
class ClassName:
  """Необязательная строка документации класса"""
  class_suite
```

У класса есть строка документации, к которой можно получить доступ через
ClassName.__doc__
class_suite состоит из частей класса, атрибутов данных и функции.

ПРИМЕР СОЗДАНИЯ КЛАССА НА PYTHON

```
class Employee:
   """Базовый класс для всех сотрудников"""
   emp_count = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.emp_count += 1

   def display_count(self):
      print('Всего сотрудников: %d' % Employee.empCount)

   def display_employee(self):
      print('Имя: {}. Зарплата: {}'.format(self.name, self.salary))
```

Переменная emp_count — переменная класса, значение которой разделяется между экземплярами этого класса. Получить доступ к этой переменной можно через Employee.emp_count из класса или за его пределами.

Первый метод __init__() — специальный метод, который называют конструктором класса или методом инициализации. Его вызывает Python при создании нового экземпляра этого класса.

Объявляйте другие методы класса, как обычные функции, за исключением того, что первый аргумент для каждого метода self. Python добавляет аргумент self в список для вас; и тогда вам не нужно включать его при вызове этих методов.

СОЗДАНИЕ ЭКЗЕМПЛЯРОВ КЛАССА

Чтобы создать экземпляры классов, нужно вызвать класс с использованием его имени и передать аргументы, которые принимает метод __init__.

```
# Это создаст первый объект класса Employee
emp1 = Employee("Андрей", 2000)
# Это создаст второй объект класса Employee
emp2 = Employee("Мария", 5000)
```

Получите доступ к атрибутам класса, используя оператор «.» после объекта класса. Доступ к классу можно получить используя имя переменой класса:

```
emp1.display_employee()
emp2.display_employee()
print("Всего сотрудников: %d" % Employee.emp_count)
```

```
class Employee:
    """Базовый класс для всех сотрудников"""
    emp_count = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.emp_count += 1

    def display_count(self):
        print('Всего сотрудников: %d' % Employee.emp_count)

    def display_employee(self):
        print('Имя: {}. Зарплата: {}'.format(self.name, self.salary))


# Это создаст первый объект класса Employee
emp1 = Employee("Андрей", 2000)
# Это создаст второй объект класса Employee
emp2 = Employee("Мария", 5000)
emp1.display_employee()
emp2.display_employee()
print("Всего сотрудников: %d" % Employee.emp_count)
```

При выполнении этого кода, мы получаем следующий результат:

Имя: Андрей. Зарплата: 2000
Имя: Мария. Зарплата: 5000
Всего сотрудников: 2

Можно добавлять, удалять или изменять атрибуты классов и объектов в любой момент.

```
emp1.age = 7  # Добавит атрибут 'age'
emp1.age = 8  # Изменит атрибут 'age'
del emp1.age  # Удалит атрибут 'age'
```

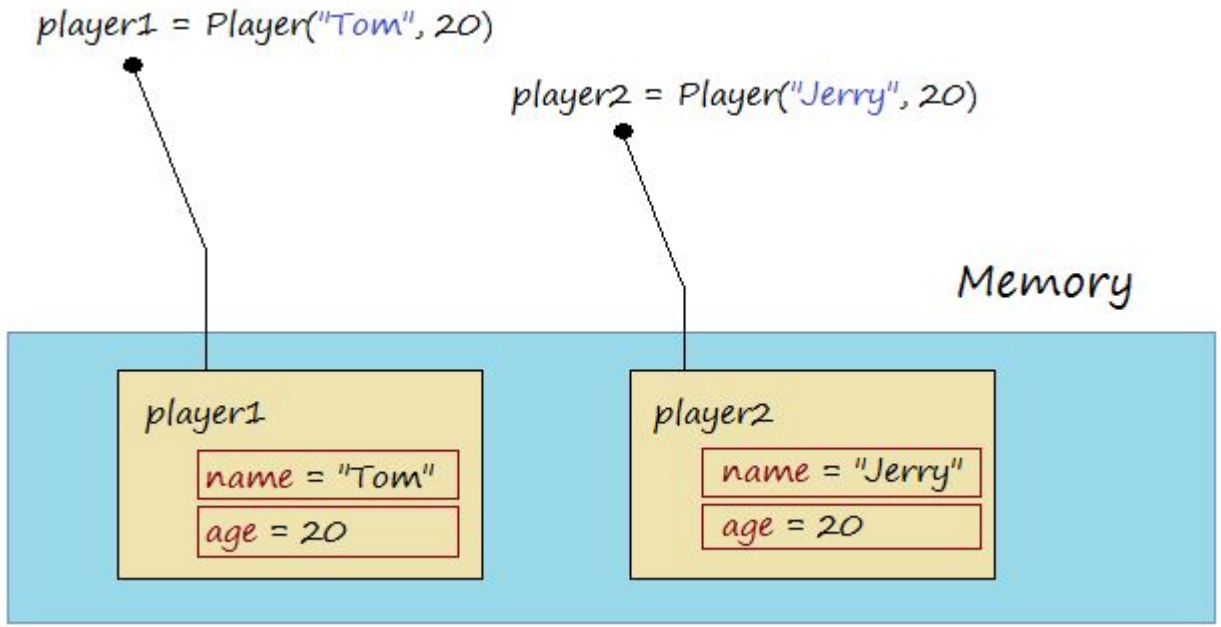
АТРИБУТЫ

В Python есть два похожих понятия, которые на самом деле отличаются:

1. Атрибуты
2. Переменные класса

```
class Player:
    # Переменная класса
    minAge  = 18
    maxAge = 50

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

Объекты, созданные одним и тем же классом, будут занимать разные места в памяти, а их атрибуты с «одинаковыми именами» — ссылаться на разные адреса. Например:



```
from player import Player
player1 = Player("Tom", 20)
player2 = Player("Jerry", 20)

print("player1.name = ", player1.name)
print("player1.age = ", player1.age)
print("player2.name = ", player2.name)
print("player2.age = ", player2.age)
print(" ------------ ")
print("Assign new value to player1.age = 21 ")

# Присвойте новое значение атрибуту возраста player1.
player1.age = 21
print("player1.name = ", player1.name)
print("player1.age = ", player1.age)
print("player2.name = ", player2.name)
print("player2.age = ", player2.age)
```

```
>>>
player1.name =  Tom
player1.age =  20
player2.name =  Jerry
player2.age =  20
 ------------
Assign new value to player1.age = 21
player1.name =  Tom
player1.age =  21
player2.name =  Jerry
player2.age =  20
>>>
```

Ln: 97 Col: 4

АТРИБУТЫ ФУНКЦИИ

Обычно получать доступ к атрибутам объекта можно с помощью оператора «точка» (например, player1.name). Но Python умеет делать это и с помощью функции.

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

```
from player import Player

player1 = Player("Tom", 20)
# getattr(obj, name[, default])
print("getattr(player1,'name') = " , getattr(player1,"name"))

print("setattr(player1,'age', 21): ")
# setattr(obj, name, value)
setattr(player1,"age", 21)
print("player1.age = ", player1.age)
```

```
# Проверка, что player1 имеет атрибут 'address'
hasAddress =  hasattr(player1, "address")
print("hasattr(player1, 'address') ? ", hasAddress)

# Создать атрибут 'address' для объекта 'player1'
print("Create attribute 'address' for object 'player1'")
setattr(player1, 'address', "USA")
print("player1.address = ", player1.address)

# Удалить атрибут 'address'.
delattr(player1, "address")
```

Вывод:

getattr(player1,'name') = Tom
setattr(player1,'age', 21):
player1.age = 21
hasattr(player1, 'address') ? False
Create attribute 'address' for object 'player1'
player1.address = USA

ВСТРОЕННЫЕ АТРИБУТЫ КЛАССА

Объекты класса — дочерние элементы по отношению к атрибутам самого языка Python. Таким образом они заимствуют некоторые атрибуты:

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.

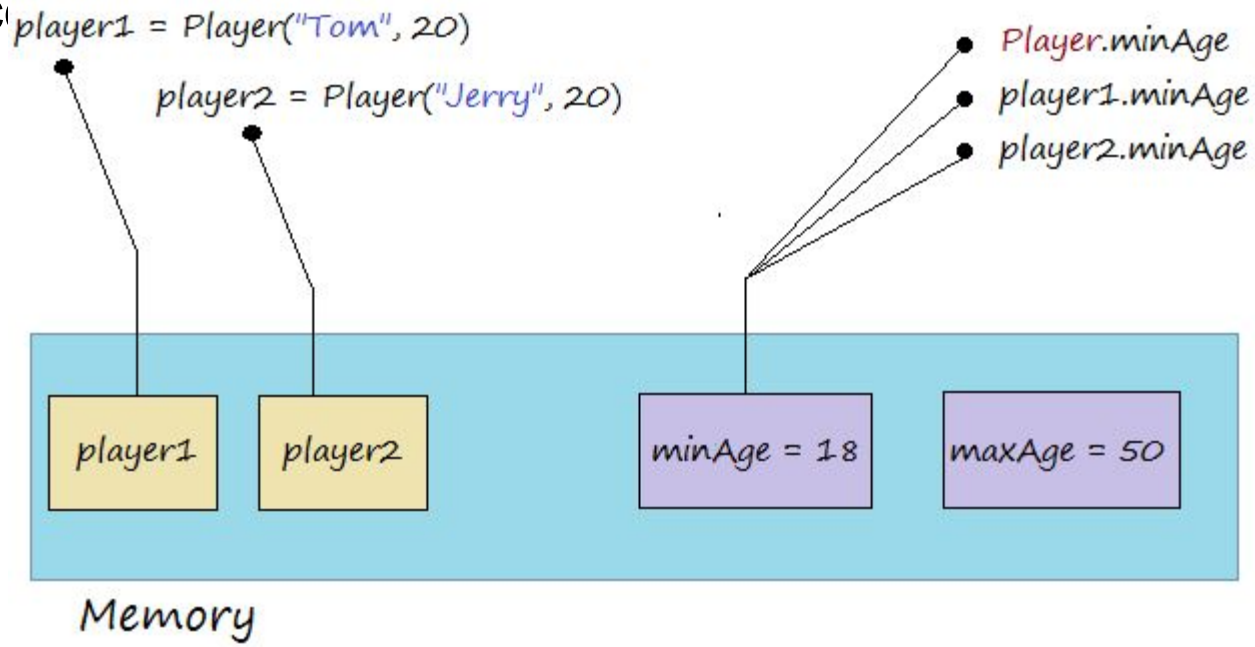
```
class Customer:
    'Это класс Customer'
    def __init__(self, name, phone, address):
        self.name = name
        self.phone = phone
        self.address = address


john = Customer("John",1234567, "USA")

print ("john.__dict__ = ", john.__dict__)
print ("john.__doc__ = ", john.__doc__)
print ("john.__class__ = ", john.__class__)
print ("john.__class__.__name__ = ", john.__class__.__name__)
print ("john.__module__ = ", john.__module__)
```

Вывод:

john.__dict__ = {'name': 'John', 'phone': 1234567, 'address': 'USA'}
john.__doc__ = Это класс Customer
john.__class__ = <class '__main__.Customer'>
john.__class__.__name__ = Customer
john.__module__ = __main__

ПЕРЕМЕННЫЕ КЛАССА

Для получения доступа к переменной класса лучше использовать имя класса, а не объект. Это поможет не путать «переменную класса» и атрибуты. У каждой переменной класса есть свой адрес в памяти. И он доступен вс



```
from player import Player

player1 = Player("Tom", 20)
player2 = Player("Jerry", 20)
# Доступ через имя класса.
print ("Player.minAge = ", Player.minAge)
# Доступ через объект.
print("player1.minAge = ", player1.minAge)
print("player2.minAge = ", player2.minAge)

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

print("Assign new value to minAge via class name, and print..")

# Новое значение minAge через имя класса
Player.minAge = 19

print("Player.minAge = ", Player.minAge)
print("player1.minAge = ", player1.minAge)
print("player2.minAge = ", player2.minAge)
```

Вывод:

Player.minAge = 18
player1.minAge = 18
player2.minAge = 18

Assign new value to minAge via class name, and print..
Player.minAge = 19
player1.minAge = 19
player2.minAge = 19

КОНСТРУКТОР КЛАССА МЕТОД INIT

В следующем примере демонстрируется инициализация класса Fib, с помощью метода *init*().

```
class Fib:
'''iterator that yields numbers
   in the Fibonacci sequence'''
    def __init__(self, max):
        …
```

1. Классы, по аналогии с модулями и функциями могут (и должны) иметь строки документации (docstrings).
2. Метод init() вызывается сразу же после создания экземпляра класса.

Было бы заманчиво, но формально неверно, считать его «конструктором» класса.

Заманчиво, потому что он напоминает конструктор класса в языке C++: внешне (общепринято, что метод init() должен быть первым методом, определенным для класса), и в действии (это первый блок кода, исполняемый в контексте только что созданного экземпляра класса).

Неверно, потому что на момент вызова init() объект уже фактически является созданным, и вы можете оперировать корректной ссылкой на него (self)

Первым аргументов любого метода класса, включая метод init(), всегда является ссылка на текущий экземпляр класса. Принято называть этот аргумент self. Этот аргумент выполняет роль зарезервированного слова this в C++ или Java, но, тем не менее, в Python self не является зарезервированным. Несмотря на то, что это всего лишь соглашение, пожалуйста не называйте этот аргумент как либо еще.
В случае метода init(), self ссылается на только что созданный объект; в остальных методах — на экземпляр, метод которого был вызван. И, хотя вам необходимо явно указывать self при определении метода, при вызове этого не требуется; Python добавит его для вас автоматически.

СОЗДАНИЕ ЭКЗЕМПЛЯРОВ

Для создания нового экземпляра класса в Python нужно вызвать класс, как если бы он был функцией, передав необходимые аргументы для метода init(). В качестве возвращаемого значения мы получим только что созданный объект.

```
>>> import fibonacci2
>>> fib = fibonacci2.Fib(100)
>>> fib
<fibonacci2.Fib object at 0x00DB8810>
>>> fib.__class__
<class 'fibonacci2.Fib'>
>>> fib.__doc__
'iterator that yields numbers in the Fibonacci sequence'
```

1. Вы создаете новый экземпляр класса Fib (определенный в модуле fibonacci2) и присваиваете только что созданный объект переменной fib. Единственный переданный аргумент, 100, соответствует именованному аргументу max, в методе init() класса Fib.
2. fib теперь является экземпляром класса Fib.
3. Каждый экземпляр класса имеет встроенный атрибут class, который указывает на класс объекта. Java программисты могут быть знакомы с классом Class, который содержит методы getName() и getSuperclass(), используемые для получения информации об объекте. В Python, метаданные такого рода доступны через соответствующие атрибуты, но используемая идея та же самая.
4. Вы можете получить строку документации (docstring) класса, по аналогии с функцией и модулем. Все экземпляры класса имеют одну и ту же строку документации.

Для создания нового экземпляра класса в Python, просто вызовите класс, как если бы он был функцией, явные операторы, как например new в C++ или Java, в языке Python отсутствуют.

ПЕРЕМЕННЫЕ ЭКЗЕМПЛЯРА

Перейдем к следующей строчке:

```
class Fib:
    def __init__(self, max):
        self.max = max
```

1. Что такое self.max? Это переменная экземпляра. Она не имеет ничего общего с переменной max, которую мы передали в метод init() в качестве аргумента. self.max является «глобальной» для всего экземпляра. Это значит, что вы можете обратиться к ней из других методов.

```
class Fib:
    def __init__(self, max):
        self.max = max

    def __next__(self):
        …
        if fib > self.max:
            …
```

2. self.max определена в методе __init__…
3. …и использована в методе __next__.

Переменные экземпляра связаны только с одним экземпляром класса. Например, если вы создадите два экземпляра класса Fib с разными максимальными значениями, каждый из них будет помнить только свое собственное значение.

```
>>> import fibonacci2
>>> fib1 = fibonacci2.Fib(100)
>>> fib2 = fibonacci2.Fib(200)
>>> fib1.max
100
>>> fib2.max
200
```

Фактически, класс — это пользовательский тип данных. Простейшая модель определения класса выглядит следующим образом:

class имя:

- инструкция1
- инструкция…

Класс состоит из объявления (инструкция class), имени класса и тела класса, которое содержит атрибуты и методы.
Для того чтобы создать объект класса необходимо воспользоваться следующим синтаксисом:
имя_объекта = имя_класса()

Класс может содержать атрибуты и методы. Ниже представлен класс, содержащий атрибуты color (цвет), width (ширина), height (высота).

class Figure:

- color = “green”
- width = 100
- height = 100

Доступ к атрибуту класса можно получить следующим образом.

имя_объекта.атрибут

```
fig1 = Figure()
print(fig1.color)
```

Добавим к нашему классу метод. Метод – это функция внутри класса. Например, нашему классу Figure, можно добавить метод, считающий площадь прямоугольника.
Для того, чтобы метод в классе «знал», с каким объектом он работает (это нужно для того, чтобы получить доступ к атрибутам: ширина (width) и высота (height)), первым аргументом ему следует передать параметр self, через который он может получить доступ к своим данным.

```
class Figure:
    color = "green"
    width = 100
    height = 100
    def square(self):
        return self.width * self.height
```

Для того, чтобы иметь возможность задать цвет, длину и ширину прямоугольника при его создании, добавим к классу Figure следующий конструктор:

```
class Figure:
    def __init__(self, color="green", width=100, height=100):
        self.color = color
        self.width = width
        self.height = height
    def square(self):
        return self.width * self.height
```

```
fig1 = Figure()
print(fig1.color)
print(fig1.square())
fig1 = Figure("yellow", 23, 34)
print(fig1.color)
print(fig1.square())
```

НАСЛЕДОВАНИЕ

В организации наследования участвуют как минимум два класса: класс родитель и класс потомок. При этом возможно множественное наследование, в этом случае у класса потомка есть несколько родителей. Не все языки программирования поддерживают множественное наследование, но в Python можно его использовать.

Синтаксически создание класса с указанием его родителя/ей выглядит так:

```
class имя_класса(имя_родителя1, [имя_родителя2,…, имя_родителя_n])
```

```
class Figure:
    def __init__(self, color):
        self.color = color
    def get_color(self):
        return self.color
```

```
class Rectangle(Figure):
    def __init__(self, color, width=100, height=100):
        super().__init__(color)
        self.width = width
        self.height = height
    def square(self):
        return self.width*self.height
```

Функция super() в Python позволяет явно ссылаться на родительский класс.

```
fig1 = Rectangle("blue")
print(fig1.get_color())
print(fig1.square())
fig2 = Rectangle("red", 25, 70)
print(fig2.get_color())
print(fig2.square())
```

МЕТОДЫ __STR__, __REPR__

Специальные методы __str__ и __repr__ отвечают за строковое представления объекта. При этом используются они в разных местах.
Рассмотрим пример класса IPAddress, который отвечает за представление IPv4 адреса:

```
class IPAddress:
    def __init__(self, ip):
        self.ip = ip
```

После создания экземпляров класса, у них есть строковое представление по умолчанию, которое выглядит так (этот же вывод отображается при использовании print):

```
ip1 = IPAddress('10.1.1.1')
ip2 = IPAddress('10.2.2.2')

str(ip1)
'<__main__.IPAddress object at 0xb4e4e76c>'
str(ip2)
'<__main__.IPAddress object at 0xb1bd376c>'
```

К сожалению, это представление не очень информативно. И было бы лучше, если бы отображалась информация о том, какой именно адрес представляет этот экземпляр. За отображение информации при применении функции str, отвечает специальный метод __str__ - как аргумент метод ожидает только экземпляр и должен возвращать строку

```
class IPAddress:
    def __init__(self, ip):
        self.ip = ip
    def __str__(self):
        return f"IPAddress: {self.ip}"

ip1 = IPAddress('10.1.1.1')
ip2 = IPAddress('10.2.2.2')

str(ip1)
'IPAddress: 10.1.1.1'
str(ip2)
'IPAddress: 10.2.2.2'
```

Второе строковое представление, которое используется в объектах Python, отображается при использовании функции repr, а также при добавлении объектов в контейнеры типа списков:

```
ip_addresses = [ip1, ip2]

ip_addresses
[<__main__.IPAddress at 0xb4e40c8c>, <__main__.IPAddress at 0xb1bc46ac>]

repr(ip1)
'<__main__.IPAddress object at 0xb4e40c8c>'
```

За это отображение отвечает метод __repr__, он тоже должен возвращать строку, но при этом принято, чтобы метод возвращал строку, скопировав которую, можно получить экземпляр класса:

```
class IPAddress:
    def __init__(self, ip):
        self.ip = ip
    def __str__(self):
        return f"IPAddress: {self.ip}"
    def __repr__(self):
        return f"IPAddress('{self.ip}')"
```

```
ip1 = IPAddress('10.1.1.1')
ip2 = IPAddress('10.2.2.2')

ip_addresses = [ip1, ip2]

ip_addresses
[IPAddress('10.1.1.1'), IPAddress('10.2.2.2')]

repr(ip1)
"IPAddress('10.1.1.1')"
```

ЗАДАЧИ

Описать класс, представляющий треугольник. Предусмотреть методы для вычисления площади и периметра.

Написать класс «Калькулятор» с основными арифметическими действиями.

Написать класс «Студенты», в который входят фамилия, номер группы, оценка. Создать массив из пяти элементов такого типа.