Python-7
ООП. Типы методов класса. Понятие стека, очереди и дека.

Что такое ООП?

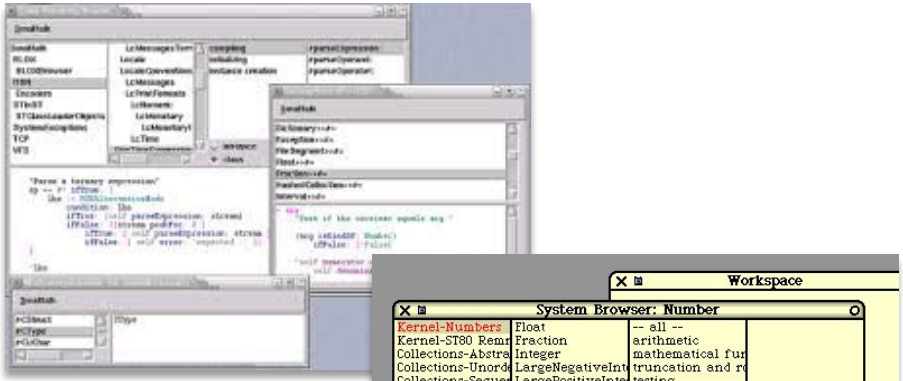
Объектно-ориентированное программирование (ООП) в Python предоставляет мощный и гибкий способ организации кода, позволяя создавать структурированные и повторно используемые компоненты. В основе ООП лежат классы и объекты, которые позволяют абстрагировать данные и их функциональность.

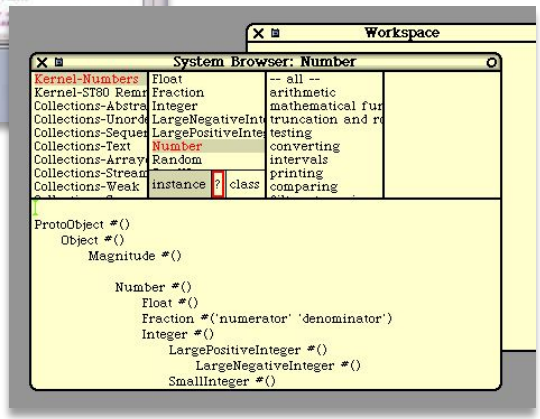
Таким образом, более общее определение **ООП** - это стиль программирования, который основывается на использовании объектов, то есть некоторых абстрактных сущностей, которые представляют некоторые объекты или концепции в реальном мире. И в языке программирования Python есть множество возможностей для ООП.

История ООП

1837: Лукас Ада сформулировал идеи алгебраического языка программирования.
1936: Алан Тьюринг предложил модель универсальной машины Тьюринга, которая легла в основу теории вычислений.
1943-1944: Энциклопедия Матем.Наук исследовала программы для ЭВМ.
1950-1960: Эра машинных ЯП, включая Fortran, Lisp, COBOL.
1970-е: Появление языков C и Pascal. Развитие идеи структурного программирования.
1980-е: Программирование на уровне объектов (ООП) стало широко распространенным. ЯП, такие как **Smalltalk**, C++, и Objective-C, интегрировали концепции ООП.
1990-е: Появление Java, Python, и C#, языков программирования, активно использующих ООП. Введение UML (Unified Modeling Language) для визуального представления структур и поведения систем.
2000-е: Развитие фреймворков и технологий, поддерживающих ООП. Важные шаги в сторону веб-разработки.

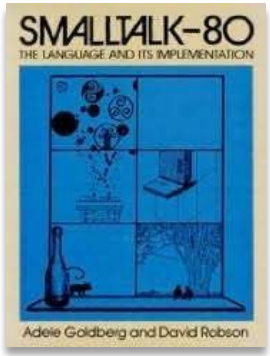
The First OOPs Code.





История ООП

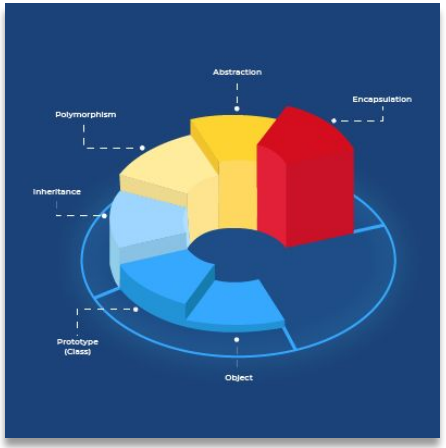
ООП возникло как развитие идей процедурного программирования, при котором данные и методы, их обрабатывающие, формально не связаны. Взгляд на программирование «под новым углом» в 80-х годах предложили **Алан Кэй** и **Дэн Ингаллс** в языке Smalltalk. Здесь понятие класса стало основообразующей идеей для всех остальных конструкций.




```
| s f c|
Transcript show: ' Enter a line: '.
s := stdin nextLine.
f := Bag new.
s do: [ :ch | ch isLetter
    ifTrue: [ f add: ch asLowercase]
].

1 to:  26 do: [ :i |
    c := (i+96) asCharacter.
    Transcript show: ((f occurrencesOf: c)  printString, ' ')
]
```

Цели использования ООП.

В Python объектно-ориентированное программирование (ООП) может использоваться для решения широкого круга задач: от простых сценариев до сложных систем. С ООП проще поддерживать и развивать продукт, ООП также можно использовать для создания полноценных модулей, библиотек или даже приложений.



Преимущества/недостатки ООП.

Плюсы (+)

- Модульность и Повторное Использование Кода
- Инкапсуляция/Наследование/Полиморфизм
- Структурированность кода
- Удобство отладки и сопровождения
- Масштабируемость и гибкость
- Разделение программы на независимые компоненты

Минусы(-)

- Часть программ требует больших ресурсов оборудования.
- Сложность написания кода.
- Возможно даже потеря производительности в некоторых программных реализациях.

SOLID принципы ООП.

SOLID - это набор принципов объектно-ориентированного программирования, разработанный для создания более гибких, расширяемых и поддерживаемых программных систем.

S: Один класс должен иметь только одну причину для изменения и каждый класс должен быть ответственен только за один аспект функциональности программы.

O: Программные сущности (классы, модули, функции) должны быть открыты для расширения, но закрыты для модификации.

L: (относится к наследованию) Подклассы должны сохранять поведение своих суперклассов.

I: Клиенты не должны зависеть от интерфейсов, которые они не используют.

D: Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. Программные сущности должны зависеть от абстракций, а не от конкретных реализаций.

Понятие класса.

Класс - это шаблон для создания объектов. В классе определяются атрибуты (переменные, которые хранят данные) и методы (функции, которые могут менять данные или выполнять различные операции).

Класс задается с использованием ключевого слова **class**, затем записывается **имя класса** (которое должно соответствовать стандартам оформления кода PEP8).

Например:

```
class Car:
    ....
class Work:
    ....
```

Понятие объекта класса.

Объект - это экземпляр класса. Когда вы создаете объект на основе класса, вы получаете экземпляр этого класса, который содержит все методы и атрибуты, определенные в классе.

Например:

```
class Person:
    ....
person1 = Person("John", 30)
```

Обобщение понятий класса и объекта.

Класс - это как чертеж или шаблон для создания чего-то нового. Давай представим, что у нас есть чертеж для изготовления машин. Этот чертеж (класс) определяет, как выглядит и как устроена машина, но сам по себе он не является машиной.

Объект - это конкретная машина, созданная по этому чертежу. Например, если мы используем чертеж машины "Мерседес" (класс "Машина"), то конкретная машина "Мерседес" (объект) будет создана на основе этого чертежа.

Таким образом, **класс** - это описание, как создать что-то, а **объект** - это конкретная вещь, созданная с использованием этого описания.

Атрибуты и методы класса.

Атрибуты (они же поля класса) - это данные, которые хранятся в объекте. (переменные,.. и др). Атрибуты можно разделить на **статические** (static) - которые объявляются в самом классе <не в INIT> и **динамические** (dynamic), которые объявляются в INIT.

Методы - это функции, которые определены в классе и могут менять данные объекта или выполнять другие операции. (вызов метода у объекта происходит через специальный символ точка (.))

object.method1(...)
object.method2(...)

Пример статических атрибутов/динамических.

```python
# static
class TestClass:
    name = "Ivan"
    age = 30
    def __init__(self, name):
        pass
```

```python
# dynamic
class TestClass:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Конструктор класса. INIT.

Конструктор класса - это специальный метод, вызываемый при создании нового объекта. В Python конструктор обозначается как __init__. Он автоматически вызывается при каждом создании экземпляра класса.

Основная цель конструктора - инициализация атрибутов объекта. Это момент, когда вы устанавливаете начальные значения для свойств объекта, задаете его начальное состояние и выполняете другие подготовительные действия.

Пример:

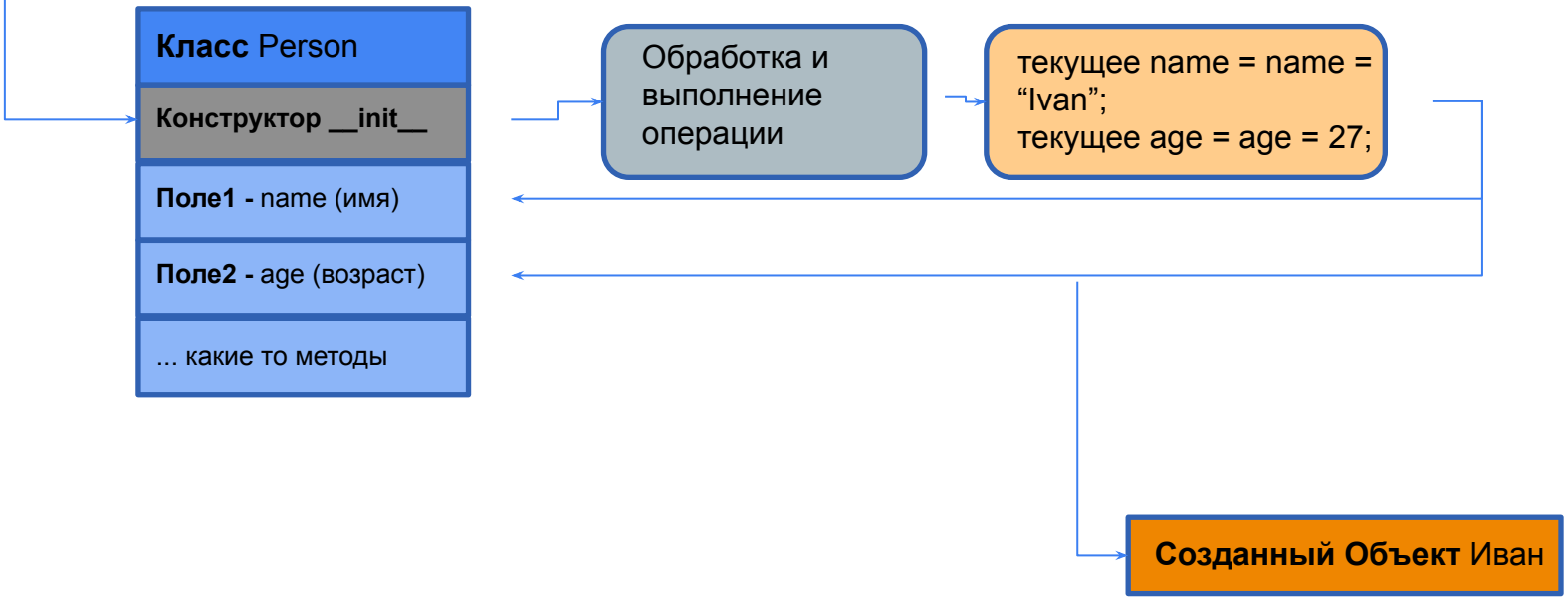
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Ключевое слово SELF.

слово self - это специальное ключевое слово в Python, которое используется внутри классов. Оно представляет экземпляр объекта, с которым в данный момент работает метод. Это как слово THIS - которое означает этот, текущий объект.

Работа конструктура (graph).

object_ivan = Person(“Ivan”, 27)



Виды конструкторов. INIT.

- Базовый конструктор INIT.

  (принимает параметры и инициализирует атрибуты объекта.)

  ```
  def __init__(self, param1, param2):
  ```

- Конструктор с параметрами по умолчанию.

  (Позволяет задать значения по умолчанию для параметров конструктора.)

  ```
  def __init__(self, param1=default1, param2=default2):
  ```

- Конструктор без параметров.

  (Создает объект с заданными атрибутами, но без явного указания параметров при создании.) `def __init__(self):`

- Конструктор с динамическим количеством параметров.

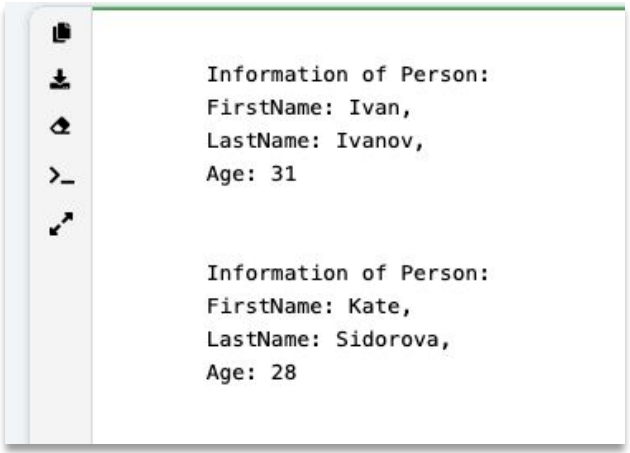
  (Позволяет передавать произвольное количество позиционных и именованных параметров.)

  ```
  def __init__(self, *args, **kwargs):
  ```

Пример класса Person.

```python
class Person:
    def __init__(self, firstname, lastname, age):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age

    def info(self):
        print(f"""
        Information of Person:
        FirstName: {self.firstname},
        LastName: {self.lastname},
        Age: {self.age}
        """)

object_person1 = Person("Ivan", "Ivanov", 31)
object_person2 = Person("Kate", "Sidorova", 28)
object_person1.info()
object_person2.info()
```

Результат выполнения кода:



```
Information of Person:
FirstName: Ivan,
LastName: Ivanov,
Age: 31


Information of Person:
FirstName: Kate,
LastName: Sidorova,
Age: 28
```

Деструктор. DEL.

Деструктор - метод класса, который определяет поведение при уничтожении объекта (т.е служит для уничтожения объекта). Однако, стоит отметить, что использование **__del__** не всегда является хорошей практикой, и в большинстве случаев его использование не требуется.

Вызов: Деструктор вызывается автоматически при сборке мусора (garbage collection) для объекта. Сборка мусора может быть неопределенной, поэтому не рекомендуется полагаться на точность времени вызова **__del__**.

Использование: **__del__** используется для выполнения операций по очистке, освобождению ресурсов или других действий перед уничтожением объекта.
Например, закрытие файлов, освобождение памяти, разрыв соединений, и т.д.

Пример __del__.

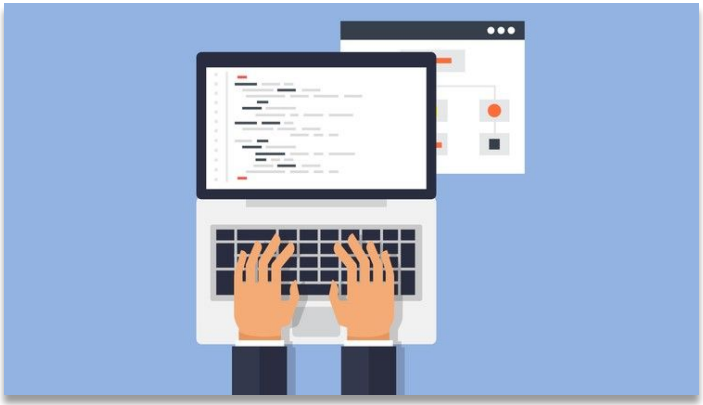
```python
class TestClass:
    def __init__(self, name):
        self.name = name

    def __del__(self):
        print(f"Object {self.name} is being destroyed.")

obj1 = TestClass("Obj1")
obj2 = TestClass("Obj2")
# При завершении программы (или явном вызове
# сборщика мусора) __del__ будет вызван
```

Основные концепции ООП.

Некоторые из основных принципов ООП в Python включают **наследование**, **инкапсуляцию** и **полиморфизм**.

Наследование позволяет создавать новые классы на основе уже существующих классов.
Инкапсуляция означает, что данные класса являются приватными, что означает, что они не могут быть изменены снаружи класса.
Полиморфизм позволяет использовать методы и функции классов с одинаковыми именами, но с разными параметрами.

**Абстрагирование - выделение сущности, игнорируя лишние детали.*



Наследование.

Наследование - это механизм, при котором новый класс создается на основе существующего класса. Новый класс наследует свойства и методы от родительского класса и может добавлять свои собственные свойства и методы. Наследование позволяет создавать новые классы на основе уже существующих, что упрощает их создание и снижает затраты на разработку.

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

    def speak(self):
        print("")

class Dog(Animal):
    def speak(self):
        print("woof")
```

Класс Dog наследует свойства и методы класса Animal, и добавляет свой собственный метод speak, который переопределяет метод speak класса Animal.

Полиморфизм.

Полиморфизм - это способность объектов разных классов использовать одни и те же методы или функции. Полиморфизм позволяет работать с объектами различных классов, не зная их конкретного типа.

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "woof"

class Cat(Animal):
    def speak(self):
        return "meow"

def animal_speak(animal):
    print(animal.speak())

dog = Dog("Buddy")
cat = Cat("Fluffy")

animal_speak(dog)
animal_speak(cat)
```

Функция animal_speak работает с объектами разных классов, но вызывает один и тот же метод speak, что является примером полиморфизма.

Разница между Наследованием и Полиморфизмом.

Разница между наследованием и полиморфизмом заключается в их целях. Наследование используется для повторного использования кода и создания новых классов на основе существующих, а полиморфизм используется для работы с объектами разных классов, но обладающих одинаковым интерфейсом методов.

Инкапсуляция

Инкапсуляция означает, что данные объекта скрыты от прямого доступа извне класса. Атрибуты класса могут быть обозначены как приватные (недоступные извне) с использованием двойного подчеркивания: **self.__private_attribute**. Поэтому для доступа и изменения атрибутов используют СЕТТЕРЫ/ГЕТТЕРЫ.

```python
class Person:
    def __init__(self, name, age):
        self.__name = name  # приватное свойство
        self.__age = age    # приватное свойство

    # геттер для имени
    def get_name(self):
        return self.__name

    # геттер для возраста
    def get_age(self):
        return self.__age

    # сеттер для имени
    def set_name(self, name):
        self.__name = name

    # сеттер для возраста
    def set_age(self, age):
        self.__age = age

# Создаем объект класса Person
person = Person("John", 25)

# Используем геттеры для получения значений приватных свойств
print(person.get_name())  # John
print(person.get_age())   # 25

# Используем сеттеры для изменения значений приватных свойств
person.set_name("Jack")
person.set_age(30)

# Проверяем, что значения были изменены
print(person.get_name())  # Jack
print(person.get_age())   # 30
```

*Абстрагирование.

Абстрагирование в программировании - это процесс выделения ключевых характеристик объекта или процесса, игнорируя детали, которые не важны для текущего контекста. Это позволяет разработчикам работать с высокоуровневыми концепциями, не вдаваясь в подробности реализации.

Абстрагирование является одним из основных принципов объектно-ориентированного программирования (ООП) и способствует созданию более чистого и гибкого кода.

Абстрактные классы и методы.

Абстрактные классы и методы - в Python это механизм для создания классов, у которых есть один или несколько абстрактных методов (без реализации). Эти классы предоставляют шаблоны для подклассов, которые должны реализовать абстрактные методы. **Абстрактные классы** создаются с использованием модуля abc. **Абстрактные методы** объявляются с использованием декоратора @abstractmethod.

Модуль abc: *from abc import ABC, abstractmethod*

***ABC** - это базовый класс, который позволяет определить абстрактные методы.*
***abstractmethod** - это декоратор, который используется для объявления абстрактного метода в абстрактном классе. Он указывает, что подклассы должны предоставить реализацию этого метода.*

(о том что такое модули/библиотеки мы уже совсем скоро узнаем)

Пример (abc).

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

circle = Circle(5)
rectangle = Rectangle(4, 6)
print(f"Circle Area: {circle.area()}")
print(f"Rectangle Area: {rectangle.area()}")
```

абстрактный класс так как в скобках ABC

абстрактный метод (area) который нужно будет переопределять для каждого последующего подкласса.

переопределения абстрактного метода в подклассах Circle и Rectangle. (которые унаследованы от класса Shape)

Public/Private Атрибуты (поля)

Public (публичные) атрибуты в Python доступны из любой части программы и из любого объекта класса. Атрибуты, которые не начинаются с подчеркивания (_), считаются public.

```
class Test:
    def __init__(self):
        self.public_attr = "Public Attribute"

obj = Test()
print(obj.public_attr)  # можем получить доступ к public атрибуту
```

Private атрибуты в Python предназначены для использования только внутри класса, они не доступны напрямую извне. Атрибуты, которые начинаются с двойного подчеркивания (__).*__Test пишем для получения доступа к атрибуту.* Имя private атрибута преобразуется в форму _ClassName__attribute.

```
class Test:
    def __init__(self):
        self.__private_attr = "Private Attribute"

obj = Test()
print(obj._Test__private_attr)
```

Понятие <<заглушки>> или <<защищенная>>

В контексте ООП в Python, одиночное подчеркивание (_), часто называемое "заглушкой" (single underscore), обычно используется для обозначения переменных или методов, которые являются "внутренними" или "локальными" внутри класса. Однако это не имеет строгого значения и не влияет на доступность переменных извне класса.

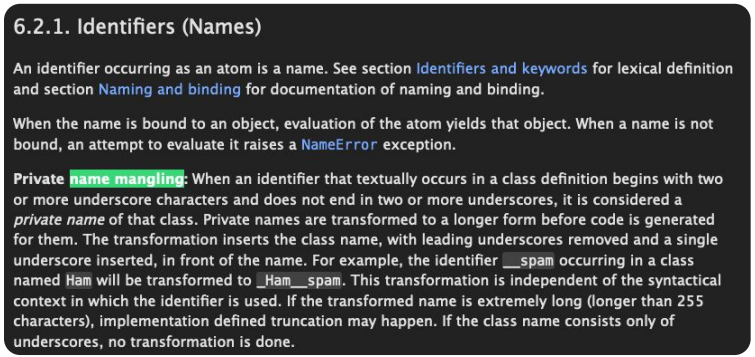
```python
class MyClass:
    def __init__(self):
        self._internal_attr = "Internal Attribute"

    def _internal_method(self):
        print("Internal Method")

obj = MyClass()
print(obj._internal_attr)  # можем получить доступ к "внутреннему" атрибуту
obj._internal_method()     # можем вызвать "внутренний" метод
```

Public/Private Методы (функции)

В Python нет явного ключевого слова для определения "публичных" или "приватных" методов, как в некоторых других языках программирования. Однако, есть соглашение по использованию двойного подчеркивания в начале имени метода, чтобы обозначить его как "приватный". Это соглашение называется "name mangling" (переименование имен) и позволяет избежать случайных конфликтов имен. Остальные методы без двойного подчеркивания можно считать публичными (public).

```python
class Test:
    def __init__(self):
        self.public_method()

    def public_method(self):
        print("This is a public method")

    def _private_method(self):
        print("This is a private method")

obj = Test()
obj.public_method()   # Обращение к публичному методу
obj._private_method()  # Обращение к "приватному" методу (не рекомендуется)
```

Соглашение NM



6.2.1. Identifiers (Names)

An identifier occurring as an atom is a name. See section Identifiers and keywords for lexical definition and section Naming and binding for documentation of naming and binding.

When the name is bound to an object, evaluation of the atom yields that object. When a name is not bound, an attempt to evaluate it raises a `NameError` exception.

Private name mangling: When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a *private name* of that class. Private names are transformed to a longer form before code is generated for them. The transformation inserts the class name, with leading underscores removed and a single underscore inserted, in front of the name. For example, the identifier `__spam` occurring in a class named `Ham` will be transformed to `_Ham__spam`. This transformation is independent of the syntactical context in which the identifier is used. If the transformed name is extremely long (longer than 255 characters), implementation defined truncation may happen. If the class name consists only of underscores, no transformation is done.

https://docs.python.org/3/reference/expressions.html

Основные типы методов класса. Статические. Классовые.

Статические методы и методы класса - это особые методы в объектно-ориентированном программировании в языке Python.

Статические методы должны определяться с помощью аннотации @staticmethod перед их определением внутри класса. Они могут вызываться с использованием имени класса вместо экземпляра класса, и обычно они не имеют доступа к атрибутам класса или экземпляра. Статические методы обычно используются как вспомогательные методы, которые не зависят от состояния объекта.

Методы класса должны определяться с помощью аннотации @classmethod перед их определением внутри класса. Они могут получать аргумент cls вместо self, и имеют доступ к атрибутам класса, а не экземпляра. Методы класса используются для связывания данных с классом, а не с экземпляром.

```
class Test:
    @staticmethod
    def static_method():
        print("This is a static method")

Test.static_method()
```

```
class Test:
    class_variable = "Class variable"

    @classmethod
    def class_method(cls):
        print(f"This is a class method. Class variable: {cls.class_variable}")

Test.class_method()
```

Стороние <<магические>> методы класса.

- **__str__()** - метод, который представляет объект в виде строки при использовании функции str();
- **__len__()** - метод, который возвращает количество элементов в объекте;
- **__getitem__()** и **__setitem__()** - методы, которые позволяют получить или установить значение для элемента по индексу.

В целом, методы классов в Python помогают удобно организовывать и работать с объектами данного класса, а также предоставляют удобный интерфейс для взаимодействия с ними.

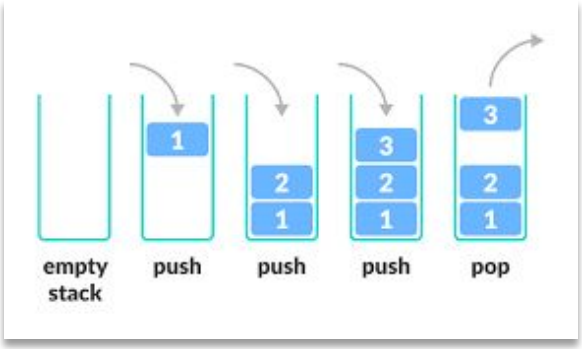
https://rszalski.github.io/magicmethods/

```
class MyList:
    def __init__(self):
        self.elements = []

    def add(self, element):
        self.elements.append(element)

    def __str__(self):
        return str(self.elements)

    def __len__(self):
        return len(self.elements)

    def __getitem__(self, index):
        return self.elements[index]

    def __setitem__(self, index, value):
        self.elements[index] = value
```

Понятие стека, очереди и дека.

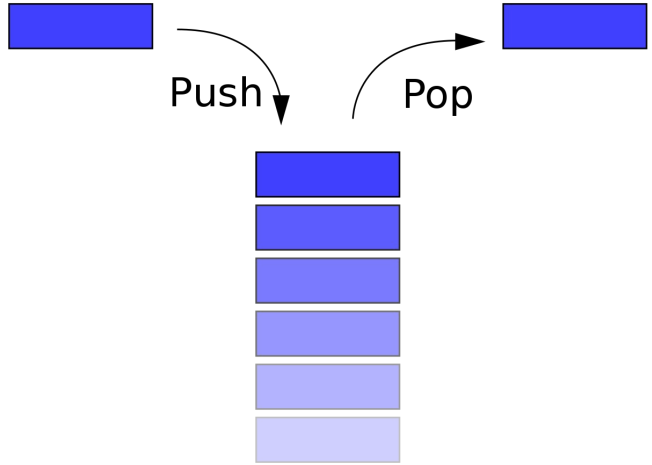
Стек, очередь и дека – это структуры данных, которые используются для организации последовательности элементов.

Стек представляет собой структуру данных, в которой элементы добавляются и удаляются только в одном конце – вершине стека. Последний элемент, добавленный в стек, будет первым элементом, который будет удален из стека. Это называется «принципом последнего вошел – первый вышел» (LIFO).



Операции со стеком.

- **Push** (Положить): Добавляет элемент на вершину стека.
- **Pop** (Извлечь): Удаляет элемент с вершины стека.
- **Peek** (Посмотреть): Возвращает значение верхнего элемента, не удаляя его.
- **IsEmpty** (Проверка на пустоту): Проверяет, пуст ли стек.
- **IsFull** (Проверка на полноту): Проверяет, полон ли стек (в случае фиксированного размера).
- **Size** (Размер стека): Возвращает количество элементов в стеке.



Алгоритмы операций PUSH/POP <stack>.

PUSH:

1. Проверка переполнения (если стек имеет ограниченный размер)
2. Увеличение указателя вершины стека
3. Запись элемента в вершину стека

Если использовать списки (list), то подобные действия делает метод append.

POP:

1. Проверка наличия элементов (если стек не пуст).
2. Чтение элемента с вершины стека.
3. Уменьшение указателя вершины стека.
4. Возвращение извлеченного элемента.

Алгоритмы операций PEEK/SIZE/ISEMPTY <stack>.

PEEK:

1. Проверка наличия элементов (если стек не пуст).
2. Чтение верхнего элемента стека.
3. Возвращение прочитанного элемента.

SIZE:

1. (здесь все просто) Возвращение размера стека.

ISEMPTY:

1. Проверка размера стека.
2. Возвращение результата проверки (True, если стек пуст, иначе False).

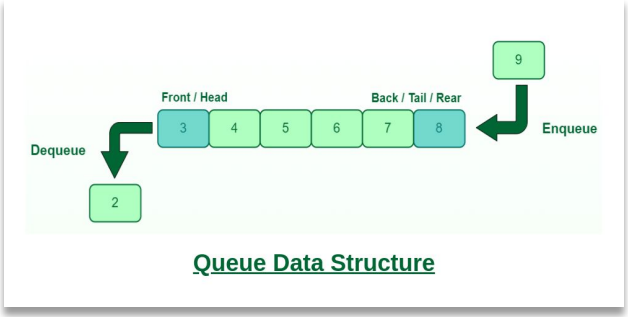
Пример создания стека.

```python
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("Pop from an empty stack")

    def peek(self):
        if not self.is_empty():
            pass

    def size(self):
        return len(self.items)
```

```python
stack = Stack()
print("Is empty:", stack.is_empty())  # True
stack.push(1)
stack.push(2)
stack.push(3)
print("Size:", stack.size())  # 3
print("Pop:", stack.pop())  # 3
print("Size after pop:", stack.size())  # 2
print("Peek:", stack.peek())
```

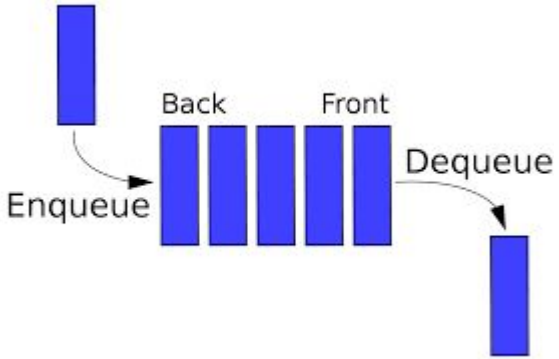
Структура Очередь.

Очередь – это структура данных, в которой элементы добавляются в конец и удаляются из начала. Первый элемент, добавленный в очередь, будет первым элементом, который будет удален из очереди. Это называется «принципом первым вошел – первый вышел» (FIFO).



Операции над очередью.

- **Enqueue** (Поместить в очередь): Добавляет элемент в конец очереди.
- **Dequeue** (Извлечь из очереди): Удаляет элемент из начала очереди.
- **Front** (Первый элемент): Возвращает значение элемента в начале очереди, не удаляя его.
- **IsEmpty** (Проверка на пустоту): Проверяет, пуста ли очередь.
- **IsFull** (Проверка на полноту): Проверяет, полна ли очередь (в случае фиксированного размера).
- **Size** (Размер очереди): Возвращает количество элементов в очереди.

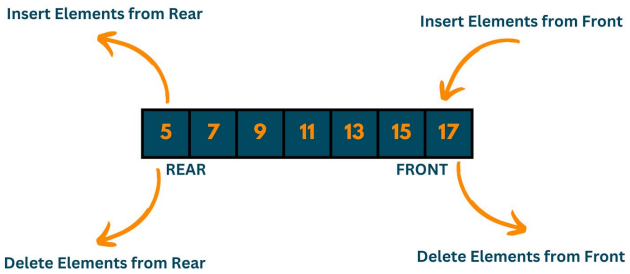


Пример создания очереди.

```python
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("Dequeue from an empty queue")

    def size(self):
        return len(self.items)

queue = Queue()
print("Is empty:", queue.is_empty())  # True
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print("Size:", queue.size())  # 3
print("Dequeue:", queue.dequeue())  # 1
print("Size after dequeue:", queue.size())  # 2
print("Is empty after dequeue:", queue.is_empty())  # False
```

Структура ДЕК.

Дека (двусторонняя очередь) – это структура данных, в которой элементы могут быть добавлены или удалены как с начала, так и с конца очереди. Дека позволяет реализовать очередь или стек в зависимости от того, какие операции выполняются в деке.



Операции над деком.

- **InsertFront** (Вставить в начало): Добавляет элемент в начало дека.
- **InsertRear** (Вставить в конец): Добавляет элемент в конец дека.
- **DeleteFront** (Удалить из начала): Удаляет элемент из начала дека.
- **DeleteRear** (Удалить из конца): Удаляет элемент из конца дека.
- **GetFront** (Получить первый элемент): Возвращает значение элемента в начале дека, не удаляя его.
- **GetRear** (Получить последний элемент): Возвращает значение элемента в конце дека, не удаляя его.
- **IsEmpty** (Проверка на пустоту): Проверяет, пуст ли дек.
- **IsFull** (Проверка на полноту): Проверяет, полон ли дек (в случае фиксированного размера).
- **Size** (Размер дека): Возвращает количество элементов в деке.

Пример создания дека.

```
class Deque:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def add_front(self, item):
        self.items.insert(0, item)

    def add_rear(self, item):
        self.items.append(item)

    def remove_front(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            raise IndexError("Remove front from an empty deque")

    def remove_rear(self):
        if not self.is_empty():
            pass

    def size(self):
        return len(self.items)
```

```
deque = Deque()
print("Is empty:", deque.is_empty())  # True
deque.add_rear(1)
deque.add_rear(2)
deque.add_front(3)
print("Size:", deque.size())  # 3
print("Remove front:", deque.remove_front())  # 3
print("Size after remove front:", deque.size())  # 2
print("Remove rear:", deque.remove_rear())  # 2
```