**### 1. "Утиная" типизация**

Утиная типизация (или "типизация по поведению") основана на том, что тип объекта определяется не его фактическим типом, а тем, какие операции над ним можно выполнять. Например, в Python, если у объекта есть метод quack(), то считается, что это "утка", даже если он не является экземпляром класса "Утка".

\*\*Пример:\*\*

```

class Duck:

def quack(self):

return "Quack!"

class Person:

def quack(self):

return "I'm quacking like a duck!"

def make\_it\_quack(duck):

print(duck.quack())

make\_it\_quack(Duck()) # Утка

make\_it\_quack(Person()) # Человек "квакает"

```

**### 2. Абстрактные типы данных и классы. Сравнение**

Абстрактные типы данных (АТД) — это теоретическая концепция, которая описывает тип данных в терминах его логического поведения, тогда как классы в ООП — это их реализация.

- \*\*АТД\*\*: Описывает операции, доступные для типа, например, стек (push, pop).

- \*\*Классы\*\*: Это конкретные реализации этих типов данных в коде, например, класс Stack, который реализует операции для работы со стеком.

\*\*Пример АТД:\*\*

- Стек: push, pop, peek – все операции определяются, но как они реализованы, не имеет значения.

**### 3. Внутренние (вложенные) классы**

Вложенные классы объявлены внутри другого класса и могут использоваться для организации кода. Они могут обращаться к атрибутам и методам внешнего класса.

\*\*Пример:\*\*

```

class Outer:

class Inner:

def inner\_method(self):

return "I'm an inner method!"

outer\_instance = Outer()

inner\_instance = outer\_instance.Inner()

print(inner\_instance.inner\_method())

```

**### 4. Генерация исключений**

В Python исключения обрабатываются с использованием конструкций try, except, finally. Их генерация позволяет сообщить о неверных действиях или предсказать исключительные ситуации.

\*\*Пример:\*\*

```

try:

x = 1 / 0

except ZeroDivisionError:

print("Вы не можете делить на ноль.")

finally:

print("Это выполнится в любом случае.")

```

**### 5. Инкапсуляция и области видимости**

Инкапсуляция скрывает внутренние детали реализации объекта и защищает его состояние. Python использует соглашения о наименовании для определения области видимости классов и атрибутов.

- \*\*Public\*\* (доступный): атрибуты/методы без подчеркивания (self.attr).

- \*\*Protected\*\* (защищенный): начинаются с одного подчеркивания (self.\_attr).

- \*\*Private\*\* (закрытый): начинаются с двух подчеркиваний (self.\_\_attr).

**### 6. История OOP, классификация и архитектура**

Объектно-ориентированное программирование возникло в 1960-х и 1970-х годах с языком Simula. Основные концепции включают:

- \*\*Классы и объекты\*\*: Основные строительные блоки OOP.

- \*\*Наследование\*\*: Позволяет создавать новые классы на основе существующих.

- \*\*Полиморфизм\*\*: Позволяет разным классам предоставлять одинаковый интерфейс.

**### 7. Источники данных и компоненты доступа к базам данных**

Компоненты доступа к базам данных управляют соединениями, запросами и транзакциями. Например, в Python можно использовать библиотеки, такие как SQLite или SQLAlchemy, для взаимодействия с различными базами данных.

**### 8. Классы для работы с датой и временем**

Модуль datetime в Python предоставляет классы для работы с датами и временем, включая datetime, date, time, timedelta.

\*\*Пример:\*\*

```

from datetime import datetime

now = datetime.now()

print("Текущая дата и время:", now)

```

**### 9. Классы для работы с файлами и каталогами**

Python предлагает функционал для работы с файлами через встроенные функции и модули, такие как os и pathlib. Эти модули позволяют выполнять операции с файлами и директориями.

\*\*Пример:\*\*

```

from pathlib import Path

# Создать новый каталог

Path("new\_directory").mkdir(parents=True, exist\_ok=True)

```

**### 10. Конструкторы и деструкторы**

Конструкторы и деструкторы — специальные методы классов.

- \*\*Конструктор\*\* (\_\_init\_\_): вызывается при создании объекта.

- \*\*Деструктор\*\* (\_\_del\_\_): вызывается при удалении объекта.

\*\*Пример:\*\*

```

class MyClass:

def \_\_init\_\_(self):

print("Конструктор вызван!")

def \_\_del\_\_(self):

print("Деструктор вызван!")

obj = MyClass()

del obj

```

**### 11. Множественное наследование и MRO**

Множественное наследование позволяет классу наследовать от более чем одного родительского класса. Метод разрешения порядка (MRO) определяет порядок, в котором классы проверяются при поиске методов.

\*\*Пример:\*\*

```

class A:

pass

class B(A):

pass

class C(A):

pass

class D(B, C):

pass

print(D.mro()) # Показывает порядок разрешения

```

**### 12. Обработка событий клавиатуры и мыши**

В GUI-приложениях, таких как Tkinter, публикуется событие (например, нажатие клавиши или щелчок мыши), которое можно обработать.

\*\*Пример:\*\*

```

import tkinter as tk

def on\_key\_press(event):

print(f'Нажата клавиша: {event.char}')

root = tk.Tk()

root.bind('<KeyPress>', on\_key\_press)

root.mainloop()

```

**### 13. Объектная модель программы**

Объектная модель программы описывает, как объекты и классы взаимодействуют. Объект — это экземпляр класса, атрибуты — это данные в объектах, а свойства определяют доступ к этим данным.

**### 14. Оконные классы**

Оконные классы управляют графическим интерфейсом. Примером является класс Tk в Tkinter, который создает главное окно приложения.

**### 15. Основные подходы к разработке ПО**

Основные подходы включают:

- Процессный подход, ориентированный на задачи.

- Объектно-ориентированный подход, ориентированный на объекты.

**### 16. Перегрузка операций. Операция сложения**

Перегрузка операторов позволяет изменять стандартное поведение операторов для пользовательских классов.

\*\*Пример:\*\*

```

class Vector:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(1, 2)

v2 = Vector(3, 4)

v3 = v1 + v2 # Использует перегрузку +

```

**### 17. Перегрузка операций. Операция сравнения**

Перегрузка операторов позволяет изменять поведение операторов сравнения для классов.

\*\*Пример:\*\*

```

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_eq\_\_(self, other):

return self.x == other.x and self.y == other.y

p1 = Point(1, 2)

p2 = Point(1, 2)

print(p1 == p2) # True

```

**### 18. Полиморфизм: назначение и семантика**

Полиморфизм позволяет использовать методы с одинаковыми именами для разных типов объектов, что упрощает код и увеличивает его гибкость.

**### 19. Принципы ООП**

Принципы ООП включают:

- \*\*Инкапсуляция\*\*: Скрытие состояния объектов.

- \*\*Наследование\*\*: Позволяет создавать новые классы на основе существующих.

- \*\*Полиморфизм\*\*: Позволяет использовать один интерфейс для разных объектов.

- \*\*Абстракция\*\*: Освобождает ненужные детали и предоставляет только важную информацию.

**### 20. Рефлексия и интроспекция**

Рефлексия позволяет программе изучать свою структуру во время выполнения. Интроспекция — это способность объекта анализировать его собственные атрибуты, методы и типы.

**### 21. Сериализация и десериализация. Pickle**

Сериализация — это процесс преобразования объектов в байты, а десериализация — обратно. Модуль pickle позволяет сохранять и загружать объекты.

\*\*Пример:\*\*

```

import pickle

data = {'key': 'value'}

serialized\_data = pickle.dumps(data)

deserialized\_data = pickle.loads(serialized\_data)

```

**### 22. Сериализация и десериализация. Файлы json**

JSON — текстовый формат данных, часто используемый для обмена данными. Python предоставляет библиотеку json для работы с JSON.

\*\*Пример:\*\*

```

import json

data = {'key': 'value'}

json\_data = json.dumps(data) # Сериализация

data\_loaded = json.loads(json\_data) # Десериализация

```

**### 23. Статический (Ad hoc) полиморфизм**

Статический полиморфизм (или Ad hoc полиморфизм) реализуется через перегрузку функций, позволяя одной функции обрабатывать разные типы статических данных. Например, в Python это может быть достигнуто через аргументы функции.

**### 24. Типовые компоненты для обслуживания баз данных**

Типовые компоненты включают библиотеки и модули для взаимодействия с базами данных, такие как SQLAlchemy, pandas с SQL, и различные ORM (Object-Relational Mapping) инструменты.

**### 25. UML. Диаграмма классов**

Диаграммы классов UML отображают классы системы, их атрибуты, методы и отношения между ними. Они помогают визуализировать архитектуру системы.

**### 26. UML. Диаграмма прецедентов**

Диаграммы прецедентов иллюстрируют взаимодействие между пользователями (актерами) и системой, показывая, какие функции (прецеденты) доступны пользователям.