# **[Глава 2](#_Глава_2_ПЕРЕМЕННЫЕ,) ПЕРЕМЕННЫЕ, ТИПЫ ДАННЫХ И ОПЕРАТОРЫ**

# [**Глава 3**](#_Глава_3_УПРАВЛЯЮЩИЕ) **УПРАВЛЯЮЩИЕ СТРУКТУРЫ: УСЛОВНЫЕ ОПЕРАТОРЫ И ЦИКЛЫ**

# [**Глава 4**](#_Глава_4_ФУНКЦИИ) **ФУНКЦИИ И МОДУЛИ**

# [**Глава 5**](#_Глава_5_СТРУКТУРЫ) **СТРУКТУРЫ ДАННЫХ: CПИСКИ, КОРТЕЖИ И СЛОВАРИ**

# [**Глава 6**](#_Глава_6_ВВОД) **ВВОД И ВЫВОД**

# [**Глава 7**](#_Глава_7_ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ) **ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**

# **Глава 2 ПЕРЕМЕННЫЕ, ТИПЫ ДАННЫХ И ОПЕРАТОРЫ**

# **Переменная** — это, по сути, именованный контейнер, в котором хранится значение. Оно может быть числом, строкой или любым другим типом данных.

# **Типы данных**

# Python поддерживает несколько основных типов данных, включая числовые, строковые и булевы значения. Вот краткий обзор некоторых типов данных: 1. **Целочисленные значения**: целые числа, 1, 2, 3 и т. д. Они представлены с помощью типа данных **int**. int **в Python 3 —** безразмерный (арифметика произвольной точности). Жёсткого «максимума» нет — ограничен только памятью.

# 2. **Числа с плавающей точкой**: десятичные числа, 9,99 или 3,1415. Они представлены с помощью типа данных **float**, точность: **53 бита мантиссы** → примерно **15–17 значащих десятичных цифр**.

# 3. **Строка**: последовательность символов, например "робот". Они представлены с помощью типа данных **str**. Конкатенация: можно объединить две строки или более вместе с помощью оператора «плюс» (+). Можно найти длину строки с помощью функции len(). Индексирование: можно получить доступ к отдельным символам в строке с помощью индексации. Индексация в Python начинается с 0, это означает, что первый символ в строке имеет индекс 0. Нарезка: можно извлечь подстроку из строки с помощью нарезки (слайсин га). Нарезка позволяет указать диапазон индексов для извлечения из строки. substring = string4[0:5].

# 4. **Булевы значения: True или False**. Они представлены с помощью типа данных bool. В Python можно выполнять булевы операции с помощью следующих операторов:

# **Логическое И**: логический оператор **AND** (И) возвращает значение True, если оба операнда равны True.

# **Логическое ИЛИ**: логический оператор **OR** (ИЛИ) возвращает True, если хотя бы один операнд равен True.

# **Логическое НЕ**: логический оператор **NOT** (НЕ) возвращает противоположный операнд.

# Булевы значения также можно сравнивать с помощью операторов сравнения, таких как оператор равенства (==) или неравенства (!=).

# 5. **None**: специальный тип данных, который представляет отсутствие значения. Он часто используется для представления переменных, которые еще не инициализированы или не имеют значения.

# **Приведение типов**

# Приведение, или преобразование, типов — это процесс преобразования одного типа данных в другой. Например, можно преобразовать строку в целое число с помощью функции int() int5 = int(string5) или в строку string6 = str(int6).

# **Операторы сравнения**

# Операторы сравнения используются для сравнения двух значений и возвращают булево значение (True или False) на основе результата сравнения.

# 1. Равно (==): возвращает True, если два значения равны.

# 2. Не равно (!=): возвращает True, если два значения не равны.

# 3. Больше чем (>): возвращает True, если первое значение больше второго.

# 4. Меньше чем (<): возвращает True, если первое значение меньше второго.

# 5. Больше или равно (>=): возвращает True, если первое значение больше или равно второму.

# 6. Меньше или равно (<=): возвращает True, если первое значение меньше или равно второму.

# **Глава 3 УПРАВЛЯЮЩИЕ СТРУКТУРЫ: УСЛОВНЫЕ ОПЕРАТОРЫ И ЦИКЛЫ**

# **Условные операторы: if, elif и else**. Условные операторы используются для управления ходом выполнения в программах. Основной условный оператор в Python — if. Он проверяет условие и выполняет блок кода, если условие True.

# **Булевы выражения и операторы сравнения.** Булево выражение — это выражение, результатом которого является значение логического типа данных: True или False.

# 1. Равно (==): возвращает True, если два значения равны.

# 2. Не равно (!=): возвращает True, если два значения не равны.

# 3. Больше чем (>): возвращает True, если первое значение больше второго.

# 4. Меньше чем (<): возвращает True, если первое значение меньше второго.

# 5. Больше или равно (>=): возвращает True, если первое значение больше или равно второму.

# 6. Меньше или равно (<=): возвращает True, если первое значение меньше или равно второму.

# **Циклы: for и while**

# Циклы используются для многократного выполнения блока кода до тех пор, пока не будет выполнено определенное условие.

# **Цикл for** используется для итерации последовательности (например, списка, кортежа или строки) и выполнения блока кода для каждого элемента в последовательности.

# **Цикл while** используется для многократного выполнения блока кода до тех пор, пока определенное условие не будет равно True.

# В дополнение к циклам for и while Python также поддерживает **операторы управления циклами**, которые позволяют управлять потоком выполнения в циклах.

# **Есть три оператора управления циклами:**

# **break** — преждевременное завершение цикла;

# **continue** — пропускает текущую итерацию цикла и переходит к следующей итерации;

# **pass** — ничего не делает и используется в качестве заполнителя. Когда оператор pass встречается внутри цикла, он ничего не делает, и выполнение программы продолжается со следующей итерации.

# **Вложенные циклы и условия**

# Вложенные циклы и условия используются для создания более сложных условий и итераций по многомерным структурам данных. Вложенный цикл — это цикл внутри другого цикла, а вложенное условие — это условие внутри другого условия.

# **Глава 4 ФУНКЦИИ И МОДУЛИ**

# **Функция** — это блок многократно используемого кода, который выполняет определенную задачу и может быть вызван из других частей программы. Функция определяется с помощью ключевого слова def. Функции также могут возвращать значения с помощью ключевого слова return.

# **Параметры и аргументы функции.** В Python параметр функции — это переменная, которая определяется в заголовке функции и используется для получения входных значений при вызове функции. Аргумент — это значение, которое передается функции при ее вызове. Это может быть как переменная, так и литерал.

# **Позиционные аргументы.** Это - наиболее распространенный тип параметров — они определяются в заголовке функции без значения по умолчанию. Позиционные аргументы должны быть предоставлены в том же порядке, в каком определены в заголовке функции при вызове функции. def add(x, y).

# **Параметры ключевых слов.** Это - параметры, которые определены в заголовке функции со значением по умолчанию, они могут быть предоставлены в любом порядке при вызове функции.

# **Параметры переменной длины.** Это - параметры, которые позволяют функции принимать произвольное количество аргументов, как позиционных, так и ключевых. В Python есть два типа параметров переменной длины: \*args и \*\*kwargs.

# **\*args** – позиционные аргументы, собирает неименованные (позиционные) аргументы в кортеж. Используется, когда неизвестно, сколько аргументов передадут.

# **\*\*kwargs** – именованные аргументы (ключ=значение). Собирает именованные аргументы в словарь. Используется, когда передают переменное число ключевых аргументов.

# **Операторы возврата и возвращаемые значения.** В Python функция может возвращать значение с помощью ключевого слова return. Оператор return используется для выхода из функции и возврата значения вызывающей стороне. Когда выполняется оператор return, функция немедленно завершается, а значение, указанное в операторе return, передается вызывающей стороне. Функция также может возвращать несколько значений, возвращая кортеж.

# **Область действия функции и локальные переменные.** В Python переменные, определенные внутри функции, считаются локальными для этой функции, и доступ к ним возможен только внутри функции. Это называется областью видимости функции — она помогает предотвратить конфликты имен между переменными в разных частях программы.

# По умолчанию переменные, определенные вне функции, считаются глобальными переменными и к ним можно получить доступ из любого места программы.

# **Глобальные переменные и оператор global**

# Функция использует ключевое слово **global**, чтобы указать, что она хочет изменить глобальную переменную, а не создавать новую локальную переменную с тем же именем. Если вы попытаетесь получить доступ к глобальной переменной внутри функции без использования ключевого слова global, Python предположит, что вы пытаетесь создать новую локальную переменную с тем же именем.

# **Лямбда-функции**

# В Python **лямбда-функция** — это небольшая анонимная функция, которая может принимать любое количество аргументов, но иметь только одно выражение. Лямбда-функции полезны для написания коротких одноразовых функций, которые не требуют полного определения функции и могут использоваться везде, где требуется функция. Пример: add = lambda x, y: x + y

# В этом коде переменной add присвоена лямбда-функция, которая принимает два параметра, x и y, и возвращает их сумму с помощью оператора +.

# **Встроенные функции**

# Вот некоторые из наиболее часто используемых встроенных функций в Python:

# **len()** — возвращает количество элементов в списке, кортеже, строке или

# словаре;

# **range()** — возвращает последовательность чисел от начала до конца, с необязательным размером шага;

# **str()** — преобразует объект в строку;

# **int()** — преобразует строку или число с плавающей запятой в целое число;

# **float()** — преобразует строку или целое число в число с плавающей точкой;

# **type()** — возвращает тип объекта;

# **max()** — возвращает наибольший элемент в списке, кортеже или строке;

# **min()** — возвращает наименьший элемент в списке, кортеже или строке;

# **sum()** — возвращает сумму всех элементов в списке или кортеже;

# **sorted()** — возвращает отсортированный список элементов в списке, кортеже или строке.

# **Модули и оператор import**

# Python поставляется с рядом встроенных модулей, и вы также можете создавать свои собственные модули. Оператор import дает команду Python загрузить модуль и сделать его содержимое доступным для использования в программе.

# **Стандартная библиотека Python.** Стандартная библиотека Python — это набор модулей и пакетов. Они поставляются вместе с Python и обеспечивают широкий спектр функциональных возможностей для решения общих задач программирования.

# Вот некоторые из наиболее часто используемых модулей стандартной библиотеки:

# **os** — обеспечивает взаимодействие с операционной системой, например доступ к файлам и каталогам;

# **datetime** — предоставляет классы для работы с датами и временем;

# **re** — обеспечивает поддержку регулярных выражений;

# **urllib** — предоставляет функции для работы с URL и HTTP;

# **math** — предоставляет математические функции и константы;

# **random** — предоставляет функции для генерации случайных чисел и значений;

# **json** — предоставляет функции для работы с данными JSON;

# **csv** — предоставляет функции для работы с файлами CSV.

# **Глава 5 СТРУКТУРЫ ДАННЫХ: CПИСКИ, КОРТЕЖИ И СЛОВАРИ**

# **Структура данных** — это способ хранения и организации данных в вашей программе. Структуры данных позволяют работать с коллекциями данных (списки, кортежи и словари) и выполнять операции над этими коллекциями, например сортировку, фильтрацию и поиск.

# Вот некоторые из наиболее часто используемых структур данных в Python.

# **Список** — упорядоченная коллекция элементов, представленная квадратными скобками ([]).

# **Кортеж** — упорядоченная, неизменяемая коллекция элементов, представленная круглыми скобками (()).

# **Словарь** — неупорядоченная коллекция пар «ключ — значение», представленная фигурными скобками ({}).

# **Множество** — неупорядоченная коллекция уникальных элементов, представленная фигурными скобками ({}).

# **DataFrame —** табличная структура данных, доступная в библиотеке Python pandas. Это одна из наиболее важных структур данных для алгоритмов. Она используется для обработки классических структурированных данных.

# **Списковое включение**

# Списковое включение — это лаконичный способ создания нового списка путем преобразования или фильтрации существующего.

# Вот пример простого спискового включения, которое удваивает каждый элемент в списке чисел:

# numbers = [5, 4, 3, 2, 1]

# doubles = [num \* 2 for num in numbers]

# или отбор чётных чисел

# numbers = [5, 4, 3, 2, 1]

# odds = [num for num in numbers if num % 2 != 0]

# **Сортировка и поиск в структурах данных**

# В Python можно использовать встроенные функции для сортировки и поиска структур данных, таких как списки и словари. Для сортировки списка в Python используйте метод **sort()**. Он сортирует список на месте, то есть изменяет исходный список, а не создает новый, отсортированный.

# numbers = [1, 2, 5, 7, 2, 3, 4, 9, 6]

# numbers.sort()

# **Для поиска элемента в списке** можно использовать ключевое слово in. Оно возвращает True, если элемент находится в списке, и False в противном случае.

# Вот пример поиска элемента в списке:

# numbers = [5, 4, 3, 2, 1]

# if 3 in numbers:

# print("3 is in the list")

# else:

# print("3 is not in the list")

# **Продвинутые методы работы со структурами данных**

# **Копирование и клонирование структур данных**

# В Python можно **копировать и клонировать** структуры данных для создания новых независимых копий исходных данных. Копирование и клонирование полезны, когда нужно изменить структуру данных, не затрагивая исходные данные.

# Чтобы создать копию списка в Python, можно использовать метод **copy()**.

# Чтобы создать клон списка, используйте оператор нарезки (:). Он создает новый список с теми же элементами, что и в исходном.

# **Копирование и клонирование** похожи, но у них есть несколько важных различий. **При копировании** создается новый объект, который является неглубокой (поверхностной) копией исходной структуры данных. Неглубокая копия означает, что новый объект имеет свою собственную область памяти, но разделяет те же внутренние объекты (например, вложенные списки или словари), что и исходная структура данных. Если какие-либо из внутренних объектов являются изменяемыми (например, списки), изменения этих объектов будут отражены как в исходной структуре данных, так и в копии.

# **При клонировании** создается новый объект, который является глубокой копией исходной структуры данных. Глубокая копия означает, что новый объект имеет свою собственную область памяти и свои собственные внутренние объекты. Изменения внутренних объектов нового объекта не повлияют на исходную структуру данных, и наоборот.

# **Нарезка списка и расширенные нарезки.** Вы можете нарезать список, чтобы извлечь его часть. При нарезке списка создается новый список, содержащий только выбранные элементы. Нарезка, или слайсирование, — это мощная техника, которую можно использовать для раз личных манипуляций со списками. Для нарезки списка используется оператор нарезки (:) с начальным и конечным индексами нарезки.

# Пример:

# my\_list = [5, 4, 3, 2, 1] sliced\_list = my\_list[1:4]

# В дополнение к базовой нарезке Python также поддерживает расширенную нарезку, которая позволяет указать значение шага для нарезки. Значение шага определяет, сколько элементов будет включено в срез.

# Вот пример расширенной нарезки:

# my\_list = [5, 4, 3, 2, 1] sliced\_list = my\_list[1:4:2]

# **Множественные входные последовательности в списковых включениях**

# Одна из особенностей списковых включений — возможность использования нескольких входных последовательностей, что позволяет объединять элементы из нескольких списков в одном списке.

# Чтобы использовать несколько входных последовательностей в списке, можно перечислить их друг за другом внутри квадратных скобок, разделяя запятыми.

# Пример:

# numbers = [1, 2, 3]

# colors = ['green', 'yellow', 'red']

# pairs = [(n, c) for n in numbers for c in colors]

# Этот код создаст новый список pairs, содержащий все возможные пары чисел и цветов.

# **Словарные включения**

# Помимо спискового включения Python также поддерживает словарное включение (генератор словарей), что позволяет создавать новые словари из существующих в краткой и выразительной форме. Синтаксис словаря похож на синтаксис списка, но вместо одного элемента используется пара «ключ — значение».

# Пример:

# my\_dict = {'x': 1, 'y': 2, 'z': 3}

# new\_dict = {k: v \* v for k, v in my\_dict.items() if v > 1}

# Этот код создаст новый словарь new\_dict, который будет содержать квадраты значений из my\_dict, но только для тех элементов, где значение больше 1.

# Словарное включение можно также использовать для создания словарей из других типов последовательностей. Например, вы можете создать словарь из списка кортежей.

# **Множества и операции с ними**

# Множества можно использовать для выполнения различных операций над коллекциями данных, например для нахождения пересечения или разности между двумя множествами.

# Вы можете выполнять различные операции над множествами, такие как объединение, пересечение и разность. Вот краткий обзор некоторых наиболее распространенных операций с множествами в Python.

# **Объединение**

# Возвращает множество, которое содержит все элементы из обоих множеств, без дубликатов. Для выполнения объединения используйте оператор | или метод **union()**.

# set1 = {3, 2, 1}

# set2 = {5, 4, 3}

# union\_set = set1 | set2

# # или

# union\_set = set1.union(set2)

# # Результат: {1, 2, 3, 4, 5}

# **Пересечение**

# Возвращает множество, содержащее только те элементы, которые есть в обоих множествах. Для выполнения пересечения используйте оператор & или метод **intersection()**.

# set1 = {3, 2, 1}

# set2 = {5, 4, 3}

# intersection\_set = set1 & set2

# # или

# intersection\_set = set1.intersection(set2)

# # Результат: {3}

# **Разность**

# Возвращает множество, содержащее элементы, которые есть в первом множестве, но которых нет во втором. Для выполнения разницы используйте оператор - или метод **difference()**.

# set1 = {3, 2, 1}

# set2 = {5, 4, 3}

# difference\_set = set1 - set2

# # или

# difference\_set = set1.difference(set2)

# # Результат: {1, 2}

# **Симметричная разность**

# Возвращает множество, содержащее элементы, которые есть в любом из множеств, но не в обоих. Для выполнения симметричной разности используйте оператор ^ или метод **symmetric\_difference()**. set1 = {3, 2, 1}

# set2 = {5, 4, 3}

# symmetric\_difference\_set = set1 ^ set2

# # или

# symmetric\_difference\_set = set1.symmetric\_difference(set2)

# # Результат: {1, 2, 4, 5}

# Помимо этих операций, множества также поддерживают другие методы, на пример add(), remove() и discard(), которые позволяют добавлять или удалять элементы из множества.

# **Продвинутые методы сортировки**

# Python предоставляет несколько встроенных функций сортировки, включая **sorted()**, которые можно использовать для сортировки списков, кортежей и других коллекций элементов в порядке возрастания или убывания.

# Один из приемов сортировки — параметр key в функции **sorted()**. Параметр key позволяет указать функцию, которая будет использоваться для определения порядка сортировки элементов в коллекции. Например, если у вас есть список словарей и вы хотите отсортировать список по определенному ключу в каждом словаре, используйте параметр key для указания функции key. Пример:

# my\_list = [{'name': 'Sergey', 'age': 22}, {'name': 'Peter', 'age': 42},

# {'name': 'Ivan', 'age': 32}]

# sorted\_list = sorted(my\_list, key=lambda x: x['age'])

# # Результат: [{'name': 'Sergey', 'age': 22}, {'name': 'Ivan', 'age': 32},

# {'name': 'Peter', 'age': 42}]

# Другой продвинутый метод сортировки в Python — метод **sort()**, который можно использовать для сортировки списков по месту. Он позволяет указать пользовательскую функцию сравнения, которая будет использоваться для определения порядка сортировки элементов в списке. Пример:

# my\_list = ['Peter', 'robot', 'Ivan', 'Sergey']

# my\_list.sort(key=lambda x: x.lower())

# # Результат: ['Ivan', 'Peter', 'robot', 'Sergey']

# **sort() –** это метод списка. Изменяет сам список.

# **sorted() –** это встроенная в Python функция. Работает с любыми коллекциями (списки, кортежи, словари, множества, строки). Создает новый отсортированный список, оригинал остается без изменения.

# **map() -** функция **map(func, iterable)** берёт **каждый элемент** из списка (или другого итерируемого объекта) и применяет к нему функцию func. Результат — новый объект (итератор), где каждый элемент уже преобразован.

# **filter()** - функция **filter(func, iterable)** оставляет только те элементы из последовательности, для которых функция func возвращает True.

# То есть map() **преобразует** элементы, а filter() **отбирает** элементы.

# **reduce()** - функция **reduce(func, iterable)** из модуля functools:

# берёт **первые два элемента** последовательности, применяет к ним функцию func, результат объединяет со следующим элементом, и так до конца, пока не останется одно значение. То есть reduce **сворачивает** список в одно число/строку/объект.

# **range()** - встроенная функция Python для генерации последовательностей чисел. Чаще всего используется в циклах for.

# range(start, stop, step)

# start — с какого числа начинать (по умолчанию 0); stop — до какого числа генерировать (не включая stop); step — шаг изменения (по умолчанию 1)

# **append()** — это функция, которая добавляет элемент в конец уже существующего кортежа.

# **Работа с вложенными структурами данных**

# Один из распространенных методов работы с вложенными структурами данных в Python — индексирование. Оно позволяет получить доступ к отдельным элементам во вложенной структуре данных, указав их позицию или ключ.

# Пример:

# my\_list = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]

# element = my\_list[0][2]

# # Результат: 7

# **DataFrame**

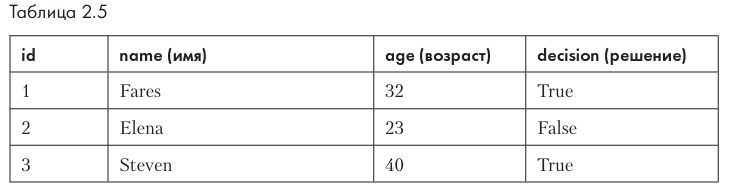
# может быть создан с помощью следующего кода:

# >>> import pandas as pd

# >>> df = pd.DataFrame([ ... ['1', 'Fares', 32, True], ... ['2', 'Elena', 23, False], ... ['3', 'Steven', 40, True]])

# >>> df.columns = ['id', 'name', 'age', 'decision']

# >>> df



Обратите внимание, что в данном коде df.column — это список, в котором со держатся имена столбцов. DataFrame используются и в других популярных языках и фреймвор ках для реализации табличной структуры данных.

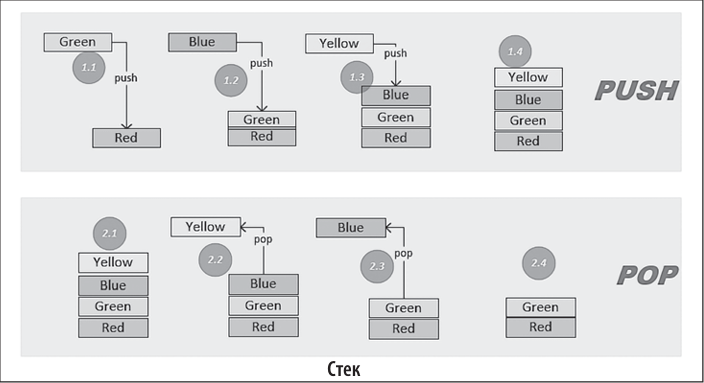
# **Стек**

Линейная структура данных для хранения одномерного списка. Элементы могут обрабатываться по принципу LIFO (LastIn, FirstOut: «последним пришел — первым ушел») либо FILO (FirstIn, LastOut: «первым пришел — последним ушел». Новые элементы могут добавляться и удаляться только с одного конца списка. LIFO и FILO — это просто разные названия одного принципа.

# isEmpty. Возвращает true, если стек пуст;

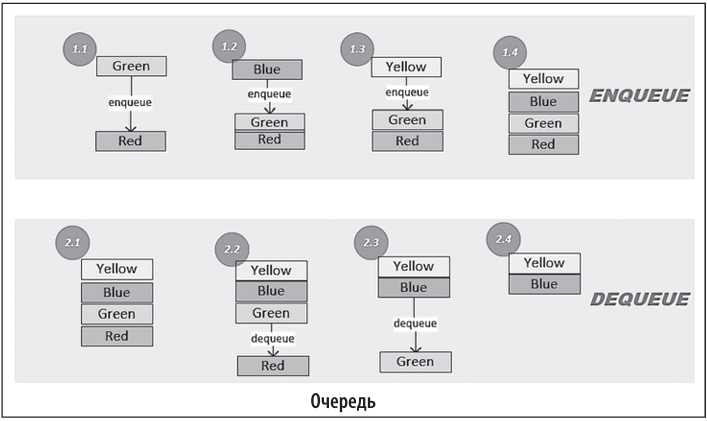
# push. Добавляет новый элемент;

# pop. Возвращает элемент, добавленный последним, и удаляет его.



# **Очередь**

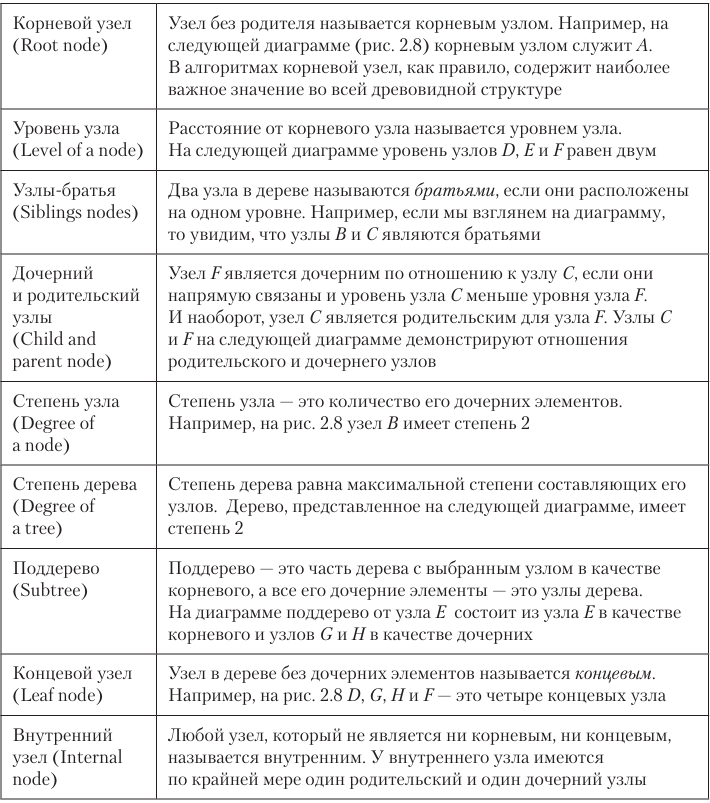
# Как и стек хранит n элементов в одномерной структуре. Элементы добавляются и удаляются по принципу FIFO (FirstIn, FirstOut: «первым пришел — первым ушел»). Когда элементы удаляются из начала, операция удаление из очереди - dequeue. Когда элементы добавляются в конец, операция постановка в очередь - enqueue.



# **Дерево**

# Это иерархическая структура данных, что делает ее особенно полезной при разработке алгоритмов. Мы используем деревья везде, где требуются иерархические отношения между элементами данных. Каждое дерево имеет конечный набор узлов, так что в нем есть начальный эле мент данных, называемый корнем (root) и набор узлов, соединенных между собой ветвями (branches).

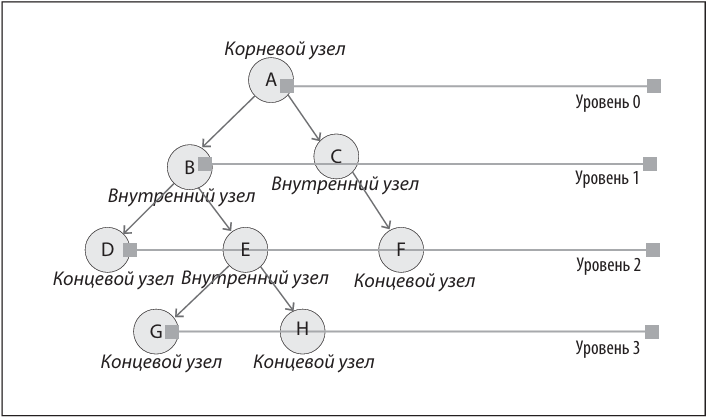
# Терминология. Рассмотрим некоторые термины, связанные с древовидной структурой данных:



# **Типы деревьев**

# Существует несколько типов деревьев:

# **Двоичное дерево (binary tree)**. Если степень дерева равна двум, оно называется двоичным. Например, дерево, показанное на следующей диаграмме, является двоичным, поскольку имеет степень 2

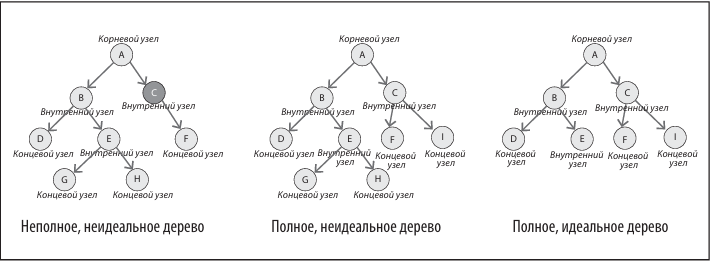


Обратите внимание, что дерево на рис. 2.8 имеет четыре уровня и восемь узлов.

# **Полное дерево (full tree).** Это дерево, в котором все узлы имеют одинаковую степень, которая равна степени дерева.

# **Идеальное дерево (perfect tree).** Это особый тип полного дерева, у которого все конечные узлы расположены на одном уровне.

# **Упорядоченное дерево (ordered tree).** Если дочерние элементы узла организованы в определенном порядке согласно установленным критериям, дерево называется упорядоченным. Дерево, например, может быть упорядочено слева направо в порядке возрастания. Таким образом, значение узлов одного уровня будет увеличиваться при движении слева направо.



# **Глава 6 ВВОД И ВЫВОД**

# Ввод и вывод (I/O) — это фундаментальные понятия в программировании, которые позволяют взаимодействовать с внешним миром путем чтения и записи данных. В Python есть несколько способов выполнения операций ввода, и вывода, включая стандартный ввод и вывод (**stdin/stdout**), ввод и вывод файлов, а также ввод и вывод по сети. В Python можно выполнять стандартные операции ввода и вывода с помощью функций input() и print() соответственно. Функция input() предназначена для чтения строки от пользователя или из другой программы, а функция print() — для записи на консоль или в другую программу. Важно отметить, что input() всегда возвращает строку, даже если пользователь вводит число или другой тип данных.

# **Ввод и вывод файлов**

# Файловый ввод и вывод (часто сокращенно называемый file I/O) является важной формой ввода и вывода в Python. Файловый ввод-вывод позволяет читать данные из файлов и записывать данные в файлы на вашем компьютере. В Python операции ввода-вывода файлов можно выполнять с помощью функции open(), чтобы открыть файл для чтения, используйте следующий синтаксис:

# file = open("filename.txt", "r") второй аргумент, "r", указывает, что мы хотим открыть файл в режиме только для чтения. Второй аргумент, "w", указывает, что мы хотим открыть файл в режиме только для записи. Второй аргумент, "a", указывает, что мы хотим открыть файл в режиме добавления. Это означает, что мы можем записывать новые данные в конец файла, не перезаписывая существующие данные в файле.

# После открытия файла используйте метод read() объекта file для чтения данных из файла или метод write() для записи данных в файл. Когда вы закончили работу с файлом, важно закрыть файл, чтобы освободить системные ресурсы и обеспечить сохранение всех изменений. В Python закрыть файл можно, вызвав метод close() объекта file.

# **Работа с двоичными файлами**

# Двоичные файлы — это файлы, содержащие нетекстовые данные: изображения, аудио или исполняемые файлы программ. При работе с двоичными файлами всегда нужно открывать файл в двоичном режиме, используя спецификатор режима "b". Например, чтобы открыть двоичный файл myfile.bin в режиме чтения, необходимо использовать следующий код:

# with open("myfile.bin", "rb") as f:

# data = f.read()

# print(data)

# **Обработка ошибок ввода-вывода при работе с файлами**

# Для обработки ошибок ввода-вывода при работе с файлами можно использовать встроенные в Python механизмы обработки ошибок, такие как блоки **try-except**.

# **Глава 7 ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**

# Объектно-ориентированное программирование (ООП) — это парадигма программирования, основанная на концепции «объектов», которые могут содержать данные и код для манипулирования этими данными. В ООП программа состоит из набора объектов, которые взаимодействуют друг с другом для выполнения задач. Ключевая идея ООП заключается в том, что объекты можно использовать для моделирования реальных сущностей или концепций.

# **Классы и объекты**

# В Python классы используются для определения свойств и поведения объектов. Класс — это схема объекта, определяющая его атрибуты (данные) и методы (функции).

# Вот пример определения класса в Python:

# class Person:

# def \_\_init\_\_(self, name, age):

# self.name = name

# self.age = age

# def greet(self):

# print("Hello, my name is", self.name, "and I am", self.age, "years old.").

# В этом примере мы определяем класс Person, который имеет два атрибута (имя и возраст) и один метод (приветствие). Метод \_\_init\_\_ — это специальный метод, называемый конструктором, который вызывается при создании нового экземпляра класса. Конструктор принимает два параметра (имя и возраст) и устанавливает соответствующие атрибуты объекта.

# **Атрибуты и методы класса**

# В Python класс может иметь как атрибуты, так и методы. Атрибуты класса являются общими для всех экземпляров класса, в то время как атрибуты экземпляра уникальны для каждого экземпляра. Аналогично, методы класса являются общими для всех экземпляров класса, а методы экземпляра уникальны для каждого экземпляра.

# **Атрибуты экземпляра и методы**

# В Python атрибуты экземпляра уникальны для каждого экземпляра класса. К ним можно получить доступ и изменить их с помощью точечной нотации, и они определяются в методе \_\_init\_\_ класса.

# **Конструкторы и деструкторы**

# В Python конструктор — это специальный метод, который вызывается при создании объекта из класса. Конструктор отвечает за установку начальных значений атрибутов объекта. Метод конструктора называется **\_\_init\_\_** и принимает параметр **self** (который ссылается на создаваемый объект) и любые другие параметры, необходимые для инициализации атрибутов объекта.

# В дополнение к конструктору Python также предоставляет специальный метод — деструктор, который вызывается при уничтожении объекта или сборке мусора. Метод деструктора называется **\_\_del\_\_**, он принимает параметр self.

# **Наследование**

# Наследование — это мощная возможность ООП, которая позволяет определить новый класс на основе существующего. Новый класс наследует все атрибуты и методы существующего класса, а также может добавлять новые атрибуты и методы. Существующий класс называется базовым, а новый класс — производным. Производный класс может переопределять атрибуты и методы базового класса или добавлять новые. В Python вы можете создать производный класс, указав базовый класс в круглых скобках после имени класса.

# В этом примере класс Animal — базовый, а Dog и Cat — производные. Классы Dog и Cat переопределяют метод speak класса Animal и добавляют собственную реализацию.

# Помимо переопределения методов, производный класс может вызывать методы базового класса с помощью функции super().

# **Переопределение метода**

# Переопределение метода — это свойство наследования, которое позволяет предоставить новую реализацию метода в производном классе, отменяющую реализацию в базовом классе. В Python переопределение метода — это простое определение метода в производном классе с тем же именем, что и метод в базовом классе:

# class Animal:

# def speak(self):

# print("The animal makes a sound")

# class Dog(Animal):

# def speak(self):

# print("The dog barks")

# class Cat(Animal):

# def speak(self):

# print("The cat meows")

# В этом примере класс Animal имеет метод speak, который выводит общее сообщение. Классы Dog и Cat переопределяют метод speak со своей собственной реализацией.

# **Полиморфизм**

# Полиморфизм — это способность объекта принимать различные формы. В ООП полиморфизм позволяет писать код, который может работать с объектами разных классов, если они имеют общий интерфейс или базовый класс.

# В Python полиморфизм достигается за счет **переопределения и перегрузки методов**. **Переопределение методов**, как обсуждалось ранее, позволяет предоставить новую реализацию метода в производном классе, которая отменяет реализацию в базовом классе. **Перегрузка методов** позволяет определить несколько методов с одинаковым именем, но разными параметрами.

# Пример перегрузки методов:

# class Math:

# def add(self, a, b):

# return a + b

# def add(self, a, b, c):

# return a + b + c

# **Абстрактные классы и интерфейсы**

# В Python **абстрактный** **класс** — это класс, который не может быть инстанцирован. Он существует только для того, чтобы быть подклассом других классов. Абстрактный класс может определять абстрактные методы, то есть те, которые не имеют реализации в абстрактном классе, но должны быть реализованы любым конкретным подклассом.

# Чтобы определить абстрактный класс в Python, можно использовать модуль abc, который расшифровывается как Abstract Base Classes. Пример:

# from abc import ABC, abstractmethod

# class Shape(ABC):

# @abstractmethod

# def area(self):

# pass

# @abstractmethod

# def perimeter(self):

# pass

# В этом примере класс Shape — абстрактный класс, который определяет два абстрактных метода: площадь и периметр. Любой конкретный подкласс класса Shape должен реализовать эти методы.

# **Интерфейс** похож на абстрактный класс тем, что определяет набор методов, которые должны быть реализованы любым классом, реализующим интерфейс. Но в Python интерфейсы не являются отдельной конструкцией, как в некоторых других языках. Интерфейс в Python — это класс, который определяет набор методов, не предоставляя никакой реализации.

# **Инкапсуляция и сокрытие данных**

# **Инкапсуляция** — это практика сокрытия внутренней работы класса от внешнего мира. При этом обеспечивается публичный интерфейс, который может быть использован для взаимодействия с классом. Сокрытие данных — это практика ограничения доступа к внутренним данным класса только теми методами, которые должны манипулировать этими данными. В Python инкапсуляция и сокрытие данных могут быть достигнуты за счет использования модификаторов доступа. В Python нет встроенных модификаторов доступа, как в некоторых других языках программирования, таких как private или protected. Вместо этого он полагается на соглашения об именовании, чтобы указать предполагаемый уровень доступа для определенного метода или атрибута.

# В Python обычно используются следующие соглашения об именовании.

# **\_attribute** — одинарное подчеркивание указывает на то, что атрибут или метод должен считаться приватным. Но это лишь условность, и на самом деле она не предотвращает доступ к атрибуту или методу извне класса.

# **\_\_attribute** — двойное подчеркивание приводит к тому, что имя атрибута будет искажено интерпретатором Python, что затрудняет доступ к нему извне класса. Например, атрибут \_\_foo в классе MyClass будет храниться в словаре экземпляров как \_MyClass\_\_foo. **\_\_method()** — двойное подчеркивание в имени метода приводит к тому, что метод сортируется по именам, как атрибуты. Это полезно для предотвращения коллизий имен в подклассах.

# Пример инкапсуляции и сокрытия данных в классе Python:

# class BankAccount:

# def \_\_init\_\_(self, balance):

# self.\_balance = balance

# def deposit(self, amount):

# self.\_balance += amount

# def withdraw(self, amount):

# if amount > self.\_balance:

# raise ValueError("Insufficient funds")

# self.\_balance -= amount

# def get\_balance(self):

# return self.\_balance

# В этом примере класс BankAccount имеет приватный атрибут \_balance, который доступен только изнутри класса. Методы deposit и withdraw могут исполь зоваться для изменения баланса, а метод get\_balance предоставляет доступ к балансу только для чтения.

# **Абстракция данных и инкапсуляция данных**

# Абстракция данных относится к процессу определения основных характеристик объекта или класса при игнорировании деталей реализации. Обычно это достигается путем определения набора абстрактных методов или свойств, которые представляют интерфейс класса, без указания того, как эти методы будут реализованы.

# **Модификаторы частного доступа**

# Атрибуты и методы, которые должны быть приватными (то есть доступными только внутри класса), должны иметь префикс с двумя знаками подчеркивания и суффикс с одним знаком подчеркивания (например, \_\_private\_attribute\_). Это приводит к тому, что имя атрибута или метода «смешивается» с именем класса и случайным хешем, что затрудняет доступ к нему извне класса. Тем не менее к этим атрибутам и методам все еще можно получить доступ, используя рефлексию (reflection) или другие продвинутые техники.

# **Перегрузка операторов**

# Для перегрузки операторов вы определяете специальный метод в классе, окруженный двойным подчеркиванием и имеющий то же имя, что и оператор, который вы хотите перегрузить. Например, чтобы перегрузить оператор +, вы определяете метод \_\_add\_\_ в своем классе.

# Пример класса, который перегружает оператор +:

# class ComplexNumber:

# def \_\_init\_\_(self, real, imaginary):

# self.real = real

# self.imaginary = imaginary

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

# return ComplexNumber(self.real + other.real, self.imaginary +

# other.imaginary)

# def \_\_str\_\_(self):

# return f"{self.real} + {self.imaginary}i"

# Мы перегружаем оператор +, определяя метод \_\_add\_\_, который принимает два экземпляра ComplexNumber в качестве аргументов и возвращает новый экземпляр ComplexNumber, представляющий сумму двух экземпляров. Мы также определяем метод \_\_str\_\_, который преобразовывает экземпляр класса в строку для печати или других целей. Символ "f" перед открывающей кавычкой указывает Python рассматривать строку как f-строку, что позволяет оценивать выражения внутри фигурных скобок и вставлять их в строку.

# **Класс и статические методы, и переменные**

# Методы класса — это методы, которые привязаны к самому классу, а не к его экземпляру. Чтобы определить метод класса в Python, используйте декоратор **@classmethod** в определении метода. Первый аргумент метода класса называется cls и относится к самому классу, а не к его экземпляру. Аргумент cls можно использовать для доступа к переменным уровня класса и вызова других методов класса.

# Чтобы определить статический метод в Python, используйте декоратор **@staticmethod** в определении метода. В отличие от методов класса, статические методы не принимают никаких аргументов экземпляра или класса. Они определяются в самом классе, но не имеют доступа к переменным уровня класса или другим методам класса.

# class MyClass:

# @staticmethod

# def static\_method():

# print("This is a static method")

# **Обработка исключений в ООП**

# В ООП обработка исключений используется для перехвата и обработки исключений, возникающих в коде. Когда возникает исключение, вы можете создать объект исключения и перехватить его с помощью блока **try-except**.

# **Сборка мусора и управление памятью**

# Python использует алгоритм подсчета ссылок для отслеживания количества ссылок на объект. Когда количество ссылок на объект достигает нуля, объект удаляется, а его память освобождается.

# **Продвинутые темы в ООП**

# **Декораторы и метаклассы** — это две продвинутые концепции в Python, тесно связанные с ООП. **Декоратор** — это функция, которая принимает на вход другую функцию и возвращает на выходе новую функцию. Новая функция может изменять поведение исходной функции без изменения ее исходного кода. Декораторы используются для добавления дополнительных возможностей в функции или классы, или для изменения их поведения во время выполнения.

# **Метаклассы**

# Метакласс — это класс, который определяет поведение других классов. В Python метакласс — это просто класс, который наследуется от класса типа. Метаклассы используются для динамического создания классов во время выполнения программы или для изменения поведения существующих классов.

# **Множественное наследование**

# Множественное наследование позволяет классу наследоваться (перенимать функциональность) от нескольких родительских классов. В Python множественное наследование поддерживается по умолчанию, это означает, что класс может наследоваться от двух или более родительских классов одно временно.

# class А:

# def method\_a(self):

# print("method\_a from class A")

# class B:

# def method\_b(self):

# print("method\_b from class B")

# class C(A, B):

# def method\_c(self):

# print("method\_c from class C")

# **Миксины и компоновщик**

# Миксины и компоновщик — это два паттерна проектирования, которые могут быть использованы в объектно-ориентированном программировании для расширения функциональности классов без использования наследования. **Миксин** не предназначен для использования сам по себе, он нужен для смешивания с другим классом. В Python миксины реализованы как обычные классы, определяющие методы или атрибуты, которые могут быть добавлены к другим классам.

# class LoggingMixin:

# def log(self, message):

# print(f"{self.\_\_class\_\_.\_\_name\_\_}: {message}")

# class MyClass(LoggingMixin):

# def do\_something(self):

# self.log("doing something")

# **Компоновщик**

# Компоновщик — это паттерн проектирования, который предполагает создание нового класса, содержащего экземпляр другого класса в качестве переменной члена. Затем новый class может использовать методы и атрибуты содержащегося в нем класса для реализации своего собственного поведения.

# class Engine:

# def start(self):

# print("engine started")

# class Car:

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

# self.engine = Engine()

# def start(self):

# self.engine.start()

# print("car started")

# **Порядок разрешения методов (MRO)**

# Порядок разрешения методов (method resolution order, MRO) — это концепция ООП, которая определяет порядок поиска методов в иерархии классов. В Python каждый класс имеет MRO, который представляет собой линейное упорядочивание классов-предков, включая его самого. MRO используется для определения того, какую реализацию метода использовать при вызове метода на экземпляре класса. Когда метод вызывается на экземпляре, Python ищет метод в классе экземпляра, затем в его родительском классе и так далее, пока не найдет реализацию метода. Алгоритм MRO в Python известен как алгоритм C3.

# **Утиная типизация и EAFP** В Python часто используется принцип «утиной типизации», обеспечивающий большую гибкость и менее ограничительное программирование. Идея заключается в том, что если объект ведет себя как утка, то это и есть утка, даже если на самом деле это не утка. Другими словами, при работе с объектами программист должен сосредоточиться на поведении объекта, а не на его типе или классе.

# **Monkey patch и динамические классы**

# Monkey patch (обезьяний патч) и динамические классы — это продвинутые техники программирования в Python, которые позволяют изменять поведение объекта или добавлять новую функциональность во время выполнения про граммы. Эти техники обычно используются для расширения функциональности существующего модуля или библиотеки, или для исправления ошибок в стороннем коде без необходимости изменять исходный код.

# **Monkey patch** подразумевает изменение существующего объекта или модуля во время выполнения путем замены или добавления методов, или атрибутов. Это может быть полезно при работе с устаревшим кодом, который вы не можете изменить, или когда нужно изменить поведение библиотеки без необходимости создавать и поддерживать собственную версию.

# **Динамические классы** — это классы, которые создаются во время выполнения программы с помощью метаклассов или функции type(). Это может быть полезно для создания объектов, которые имеют то же поведение, что и существующий объект, но с дополнительной или измененной функциональностью.

# Пример обезьяньего патча класса во время выполнения:

# class MyClass:

# def say\_hello(self):

# print("Hello, Master!")

# def monkey\_patch(self):

# print("Monkey patching the say\_hello method")

# MyClass.say\_hello = monkey\_patch

# obj = MyClass()

# obj.say\_hello() # вывод: Monkey patching the say\_hello method

# **Пример создания динамического класса во время выполнения с помощью функции type():**

# def make\_class(name, base, attrs):

# return type(name, base, attrs)

# MyClass = make\_class("MyClass", (object,), {"say\_hello": lambda self:

# print("Hello, Master!")})

# obj = MyClass()

# obj.say\_hello() # вывод: Hello, Master!

# **Фабрики классов и метапрограммирование**

# Метапрограммирование — это практика написания кода, который пишет код. Она позволяет создавать программы, которые могут изменять свое поведение, часто динамически, во время выполнения. Благодаря своей динамической и гибкой природе, Python — мощный язык для метапрограммирования. Один из распространенных приемов метапрограммирования — использование фабрик классов. Фабрика классов — это функция, которая создает и возвращает новый класс. Она позволяет создавать новые классы с динамическими атрибутами и методами.

# Пример простой фабрики классов:

# def make\_class(name, \*args, \*\*kwargs):

# class NewClass:

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

# pass

# def \_\_str\_\_(self):

# return name

# return NewClass

# В этом примере make\_class() — это функция, которая принимает аргумент имени и возвращает новый класс с этим именем. Класс имеет простой конструктор, который не принимает никаких аргументов, а также метод \_\_str\_\_(), который возвращает имя класса.

# Чтобы использовать фабрику классов, вызовем ее с аргументом name:

# MyClass = make\_class('MyClass')

# print(MyClass()) # вывод: MyClass

# Этот код создает новый класс MyClass с методом \_\_str\_\_(), который возвращает имя класса.

# **Рефлексия и интроспекция** — это возможности ООП, которые позволяют разработчикам проверять объекты и манипулировать ими во время выполнения программы. Рефлексия относится к способности программы проверять свою структуру и поведение во время выполнения, а благодаря интроспекции программа проверяет поведение и свойства объектов во время выполнения. В Python встроена поддержка рефлексии и интроспекции, что позволяет легко изучать объекты и классы, а также обеспечивать динамическое поведение во время выполнения программы. Один из ключевых механизмов в Python для интроспекции — функция dir(). Она возвращает список всех атрибутов объекта, включая как атрибуты, определенные классом объекта, так и любые атрибуты, которые были динамически добавлены во время выполнения. Можно также использовать функцию type() для проверки типа объекта и функцию isinstance() — для проверки того, является ли объект экземпляром этого класса.

# Python также предоставляет ряд встроенных атрибутов, которые можно ис пользовать для рефлексии, например атрибут \_\_name\_\_, который возвращает имя класса или функции, и атрибут \_\_doc\_\_, который возвращает строку документации класса или функции.