Теория по языкам программирования

Введение в программирование:

Императивное программирование

— парадигма программирования (стиль написания исходного кода компьютерной программы), для которой характерно следующее:

- в исходном коде программы записываются инструкции (команды);
- инструкции должны выполняться последовательно;
- данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями;
- данные, полученные при выполнении инструкции, могут записываться в память.

Особенности:

- Данная парадигма предполагает описания вычислений в форме инструкций, которые постепенно меняют состояние программы.
- В языках низкого уровня, например в ассемблерах, состояниями могут являться память, регистры и флаги. Инструкциями же выступают команды, которые поддерживаются целевым процессором.
- В языках более высокого уровня, например в Си, состоянием является исключительно память. Инструкции в таком случае могут быть более сложными, а также приводить к выделению и высвобождению памяти по мере своего функционирования.

В наиболее высокоуровневых языках (скажем, Python, при условии императивного программирования) состояние ограничивают переменные, а команды являются совокупностью процессов, которые в ассемблере расходовали бы огромное количество строк.

Основные понятия:

- Инструкция.
- Состояние.

Принципы императивного программирования

Описание расчета в императивном методе происходит на основе последовательности команд. В этом случае программирование предопределяет точную процедуру (алгоритм) решения задачи. При этом законченный компьютерный код имеет вид определенного количества переменных, которые, исходя из оценки условий, меняются в своем состоянии при помощи команд.

Виды:

1) Процедурное программирование

Процедурное программирование — программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка Выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, то есть значений исходных данных, в заключительное, то есть в результаты. Таким образом, с точки зрения программиста имеются программа и память, причём первая последовательно обновляет содержимое последней.

Процедурный язык программирования предоставляет возможность программисту определять каждый шаг в процессе решения задачи. Особенность таких языков программирования состоит в том, что задачи разбиваются на шаги и решаются шаг за шагом. Используя процедурный язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов. Важным шагом в развитии процедурного программирования стал переход к структурной парадигме.

2) Структурное программирование

Структурное программирование — парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков.
В соответствии с парадигмой, любая программа, которая строится без использования оператора goto (оператор безусловного перехода), состоит из трёх базовых управляющих конструкций: последовательность, ветвление, цикл; кроме того, используются подпрограммы. При этом разработка программы ведётся пошагово, методом «сверху вниз».

3) Модульное программирование
Модульное программирование — организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определённым правилам.
Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Аппаратно-зависимые подзадачи могут быть строго отделены от других подзадач, что улучшает мобильность создаваемых программ.
Модуль — последовательность логически связанных фрагментов, оформленных как отдельная часть программы.

4) Объектно ориентированное программирование
Объектно ориентированное программирование — методология или стиль программирования на основе описания типов/моделей предметной области и их взаимодействия, представленных порождением из прототипов или как экземпляры классов, которые образуют иерархию наследования.
Идеологически, ООП — подход к программированию как к моделированию информационных объектов, решающий на более высоком абстрактном уровне основную задачу структурного программирования — структурирование информации с точки зрения управляемости. Это позволяет управлять самим процессом моделирования и реализовывать крупные программные проекты.

Основные принципы структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

- абстракция для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счёте — контекстное понимание предмета, формализуемое в виде класса;
- инкапсуляция для быстрой и безопасной организации собственно иерархической управляемости: чтобы было достаточно простой команды «что делать», без одновременного уточнения как именно делать, так как это уже другой уровень управления;
- наследование для быстрой и безопасной организации родственных понятий: чтобы было достаточно на каждом иерархическом шаге учитывать только изменения, не дублируя всё остальное, учтённое на предыдущих шагах;
- полиморфизм для определения точки, в которой единое управление лучше распараллелить или наоборот — собрать воедино.

Абстракция — возможность определить характеристики (свойства и методы) объекта, которые полностью описывают его возможности.

Инкапсуляция — возможность скрыть реализацию объекта, предоставив пользователю некую спецификацию (интерфейс) взаимодействия с ним.

Наследование — возможность создания производных от родительского объекта, которые будут расширять или изменять свойства и поведение родителя.

Полиморфизм — возможность одному и тому же фрагменту кода работать с разными типами данных. Это происходит, когда объект может вести себя как другой объект.

5) Аспектно-ориентированное программирование
Аспектно-ориентированное программирование — парадигма программирования, основанная на идее разделения функциональности для улучшения разбиения программы на модули.
Существующие парадигмы программирования — процедурное, модульное, объектно-ориентированное программирование — предоставляют определённые способы для разделения и выделения функциональности: функции, модули, классы, но некоторую функциональность с помощью предложенных методов невозможно выделить в отдельные сущности. Такую функциональность называют сквозной, так как её реализация распределена по различным модулям программы. Сквозная функциональность приводит к рассредоточенному и запутанному коду, сложному для понимания и сопровождения.

Декларативное программирование

— парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается ожидаемый результат, а не способ его получения. Противоположностью декларативного является императивное программирование, при котором на том или ином уровне детализации требуется описание последовательности шагов для решения задачи. В качестве примеров декларативных языков обычно приводят HTML и SQL.
Декларативные программы не используют понятия состояния, в частности, не содержат переменных и операторов присваивания, обеспечивается ссылочная прозрачность.

Виды:
1) Функциональное программирование
Функциональное программирование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний (в значении, подобном таковому в теории автоматов). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например, как список.
Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).
На практике отличие математической функции от понятия «функции» в императивном программировании заключается в том, что императивные функции могут опираться не только на аргументы, но и на состояние внешних по отношению к функции переменных, а также иметь побочные эффекты и менять состояние внешних переменных. Таким образом, в императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, можно получить разные данные на выходе из-за влияния на функцию состояния переменных. А в функциональном языке при вызове функции с одними и теми же аргументами мы всегда получим одинаковый результат: выходные данные зависят только от входных. Это позволяет средам выполнения программ на функциональных языках кешировать результаты функций и вызывать их в порядке, не определяемом алгоритмом и распараллеливать их без каких-либо дополнительных действий со стороны программиста (что обеспечивают функции без побочных эффектов — чистые функции).

2) Логическое программирование
Логическое программирование — парадигма программирования, основанная на математической логике — программы в ней задаются в форме логических утверждений и правил вывода. Наиболее известный язык логического программирования — Пролог.

Основные принципы программирования: компилируемые и интерпретируемые языки

Компилируемый язык программирования:

Определение: В компилируемых языках программы пишутся на исходном языке (например, C++, Java) и затем компилируются в машинный код, который может быть выполнен на целевой аппаратуре.

Свойства: Компилируемые языки часто обладают высокой производительностью и эффективностью выполнения программ. Компиляция происходит один раз перед запуском программы.

Преимущества: Благодаря компиляции программы работают быстрее, так как машинный код уже оптимизирован для конкретной аппаратуры. Кроме того, компилированные программы могут быть распространены без необходимости предоставления исходного кода.

Недостатки: Программы на компилируемых языках требуют перекомпиляции при изменениях исходного кода. Компиляция может занять значительное время, особенно для больших проектов.

Интерпретируемый язык программирования:

Определение: В интерпретируемых языках программы выполняются построчно интерпретатором без предварительной компиляции в машинный код.

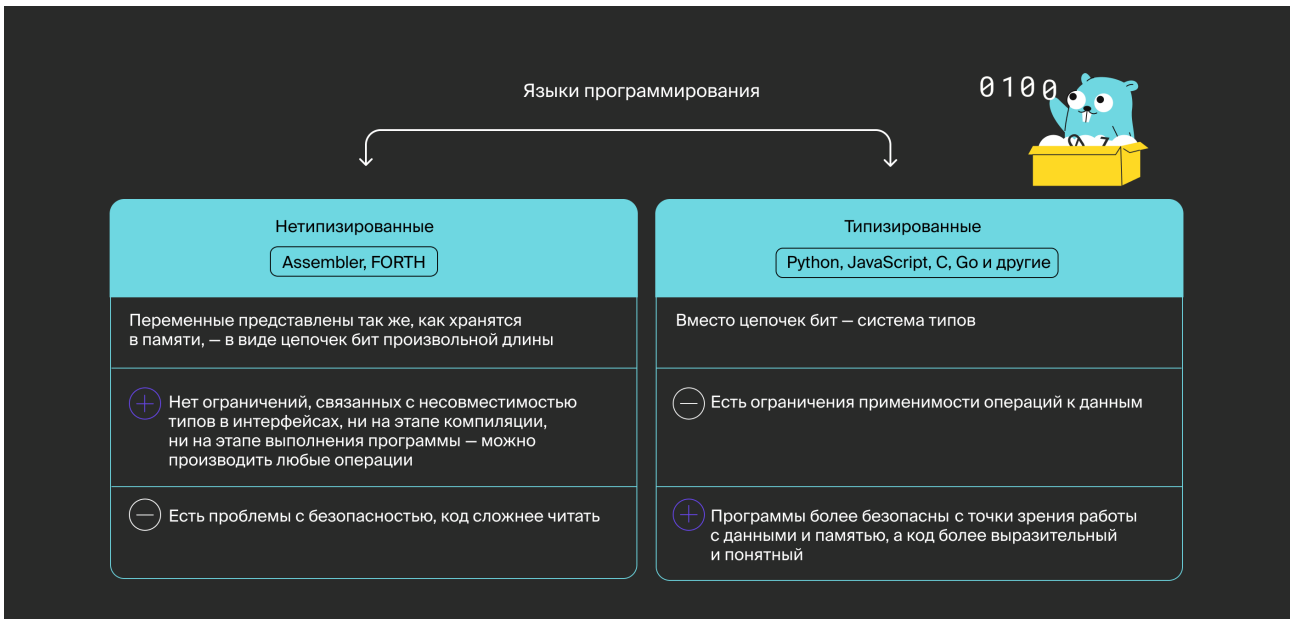
Свойства: Интерпретируемые языки обычно более гибкие и удобные для разработки, так как изменения в коде могут быть видны сразу же после внесения.

Преимущества: Интерпретация позволяет быстрее запускать и тестировать программы, а также удобно использовать в обучении и прототипировании.

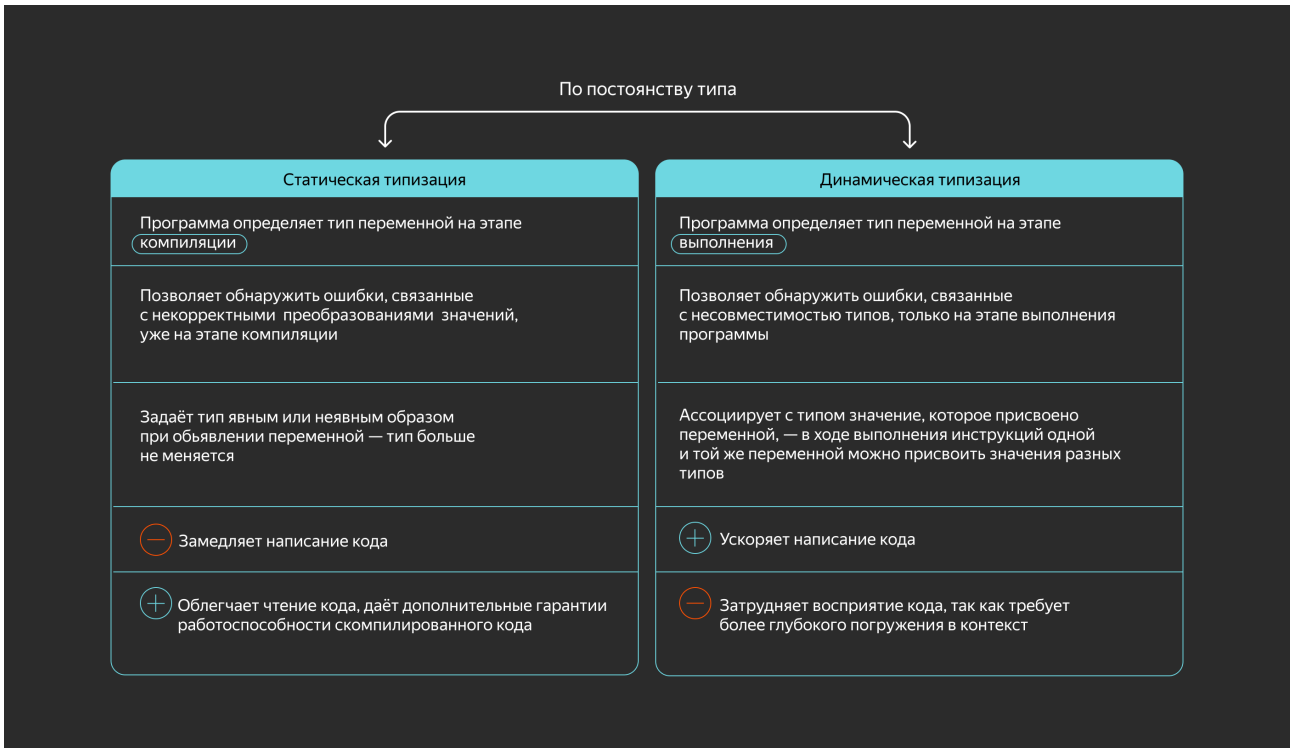
Недостатки: Интерпретация может снижать производительность программы из-за дополнительных накладных расходов на интерпретатор. Интерпретируемые программы могут быть менее эффективными по сравнению с компилированными.

Система типов

Типизация в языках программирования
Данные в компьютерах хранятся в виде бинарных последовательностей нулей и единиц. Почему используют именно двоичное представление? Причина — в универсальности: любые данные, будь то переменные или инструкции выполняемых программ, можно представить в бинарном виде. Это позволяет проектировать устройства хранения, для которых не важна природа самих данных.
Однако при написании кода программисты работают не с последовательностями бит, а с другими представлениями данных — такими как «символ», «строка» или «число». Эти представления называют типами.
Языки программирования делят на типизированные (typed) и нетипизированные (untyped), в зависимости от поддерживаемой ими системы типов.



По постоянству типа различают языки со статической и динамической типизацией. Постоянство типа указывает на то, в какой момент программа определяет тип переменной.



По тому, приводится ли значение переменной автоматически к нужному типу на стадии исполнения, различают языки со слабой и сильной типизацией. Сильная или слабая типизация характеризует отношение языка к неявным преобразованиям типов в различных контекстах. Чем сильнее типизация, тем меньше в языке автоматических и неявных преобразований.

Например, в условных конструкциях if языка Perl условием могут служить не только булевы переменные и логические выражения, но также строки и числовые переменные. В этом случае пустая строка, строка "0" или целочисленный и вещественный нули будут интерпретированы как false, а непустая строка или числовое значение, отличное от нуля, — как true. Это слабая типизация.
Явной и неявной типизацией называют способность компилятора выводить переменную из контекста. Явно типизированные языки отличаются тем, что тип новых переменных, функций и их аргументов нужно задавать явно. В языках с неявной типизацией это не всегда необходимо, потому что тип может вывести компилятор.
Система типов в языках программирования реализована по-разному. Рассмотрим характеристики типизации на примере самых популярных.

Python — динамическая и сильная типизация
В языке Python тип переменной связан с её значением и определяется на этапе выполнения программы.
Например, у переменной, значение которой представлено в виде последовательности символов в одинарных кавычках, тип — str, строка. Если следующей инструкцией присвоить ей целочисленное значение, тип поменяется на int.

```
a = 'yandex'
a = 5
```

Обратите внимание: при динамической типизации не указывают явно тип переменной, когда её объявляют.

Также в Python к строке применима конкатенация посредством операнда +. Полученная строка состоит из склеенных строк-операндов в порядке их представления в выражении:
Скопировать код
PYTHON

```
a = 'yandex'
b = 'practicum'
dot = '.'
result = a + dot + b # "yandex.practicum"
```

Но если вы захотите прибавить к строке число и получить на выходе исходную строку с приклеенным к ней числом в начале или в конце (в зависимости от порядка операндов), интерпретатор выдаст ошибку.

```
a = 'NO'
b = 10
res = a + b # ошибка: can only concatenate str (not "int") to str
```

Причина в том, что операция конкатенации работает исключительно со строками. Если бы Python автоматически преобразовывал типы перед сложением, ошибки бы не было. Этого не происходит, потому что типизация в Python характеризуется как сильная.

```
res = a + str(b)
```

При сильной типизации преобразования значений переменных не происходят автоматически — их нужно производить в коде явным образом.

JavaScript — динамическая и слабая типизация
В JavaScript тоже используется динамическая типизация, то есть тип переменной определяется её значением и может меняться в процессе выполнения программы:

```
var a = 'yandex'
a = 5
```

Как и в Python, здесь есть конкатенация строк. Однако в JavaScript возможна такая запись:

```
var a = "yandex"
var b = 5
var res = a + b
```

JavaScript автоматически, то есть неявным образом, преобразует тип переменной b в строку, прежде чем выполнить конкатенацию. Это свойство характеризует JavaScript как язык со слабой типизацией.

C — статическая и слабая, явная типизация
В языке C, в отличие от JS и Python, типизация статическая. Тип переменной определяется на этапе компиляции, а не на этапе выполнения программы. При статической типизации конструкция объявления переменной обычно содержит явное указание типа:

```
int a;
a = 42;
```

В некоторых случаях язык C автоматически преобразует типы переменных в выражениях, избавляя от необходимости делать это явно:

```
int a = 2;
char b = 'b';
int c = a + b; // 100, так как 'b' в ASCII представлен числом 98
```

Go — статическая и сильная, неявная типизация
В Go тип переменной определяется на этапе компиляции и синтаксически может быть опущен при её объявлении, если сразу инициализировать переменную значением.
В Go сильная типизация: все преобразования нужно производить явным образом. Использовать значения разных типов в выражениях запрещено.
Есть несколько способов объявить переменную в коде. Самый простой:

```
var a int
```

После ключевого слова var идёт имя переменной, по которому можно обращаться к ней в последующих инструкциях. Затем следует тип переменной — в данном случае встроенный тип int.
Значение присваивается переменной оператором = :

```
var a int
a = 5
```

Аналогичным образом значение одной переменной можно присвоить другой:

```
var a, b int
a = 5
b = a // b == 5
```

Обратите внимание: можно объявлять переменные одного типа в одной строке, просто перечислив их названия через запятую после ключевого слова var.
Также есть возможность множественного присвоения:

```
var a, b int
a, b = 5, 10 // значения присваиваются по порядку: a == 5 и b == 10
```

В Go можно делать swap значений переменных без аллокаций в одну строку:
Скопировать код
GO

```
var a, b int
a, b = 5, 10 // a == 5, b == 10
a, b = b, a // swap: a == 10, b == 5
```

Неявная типизация
В Go также есть конструкции для неявной типизации, когда при объявлении переменных тип опускается. В этом случае тип переменной или константы определяется инициализирующим значением. Это не делает Go языком с динамической типизацией, так как тип переменной, объявленной таким образом, по-прежнему устанавливается на этапе компиляции.
Пример:

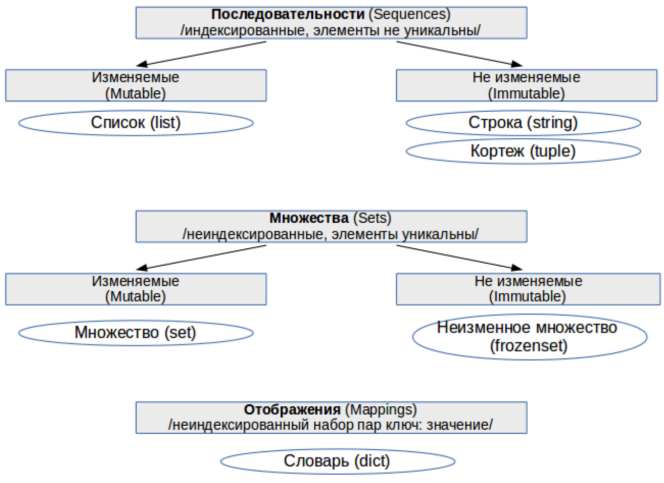
```
var intVar = 10 // переменной intVar компилятор присвоит тип int
```

PYTHON

Класс языка:
1) Императивный
2) Функциональный
3) Объектно-ориентированный

Тип исполнения:
1) Интерпретируемый
2) Компилируемый в байт-код

Кодировка по умолчанию:
UTF-8



Базовые типы данных в Python:
1) Числовые типы:
- **int:** целые числа, например 5, -10, 1000.
- **float:** числа с плавающей точкой, например 3.14, -0.001, 2.5.
- **complex:** комплексные числа, например 3+4j, -2-6j.

2) Строки (Strings):
- Последовательности символов, заключенные в кавычки (одинарные, двойные или тройные). Например: "Hello, World!", 'Python', """Многострочная строка""".

3) Списки (Lists):
- Упорядоченные изменяемые коллекции объектов. Могут содержать элементы различных типов. Например: [1, 2, 3], ['a', 'b', 'c'], [1, 'hello', True].

4) Кортежи (Tuples):
- Упорядоченные неизменяемые коллекции объектов. Объявляются с помощью круглых скобок. Например: (1, 2, 3), ('apple', 'banana', 'cherry').

5) Множества (Sets):
- Неупорядоченные коллекции уникальных элементов. Объявляются с помощью фигурных скобок. Например: 1, 2, 3, 'apple', 'banana', 'cherry'.

6) Словари (Dictionaries):
- Коллекции пар ключ-значение. Ключи должны быть уникальными и неизменяемыми. Объявляются с помощью фигурных скобок и двоеточия. Например: 'name': 'Alice', 'age': 30.

7) Логический тип (Boolean):
- Принимает значения True или False. Используется для условных операторов и логических выражений.

8) None:
- Тип данных, представляющий отсутствие значения или ничего. Используется для обозначения отсутствия результата или переменной без значения.

Базовые конструкции Python:
- Ввод и вывод данных. Операции с числами, строками. Форматирование
- Условный оператор
- Циклы
- Вложенные циклы

Таблица 2.1.3 - Элементы и блоки в Python

№	Наименование блока	Описание
1	Последовательность (инструкция)	Любое атомарное действие, например, присваивание
2	Ветвление (условие)	Выполнение инструкций в зависимости от определенного условия
3	Цикл	Многократное исполнение набора инструкций
4	Подпрограмма (процедура/функция)	Часть компьютерной программы, содержащая описание определенного набора инструкций, которая может быть многократно вызвана из разных частей программы. Может содержать (1)-(3)
5	Класс	Абстрактный тип данных в объектно-ориентированном программировании, задающий общее поведение для группы объектов; модель объекта. Может содержать (1)-(4)
6	Модуль	Функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом или поименованной непрерывной ее части. Может содержать (1)-(5)
7	Пакет	Логически законченная совокупность модулей как единое целое

Коллекции и работа с памятью
- Строки, кортежи *(неизменяемый массив)*, списки
- Множества, словари
- Списочные выражения. Модель памяти для типов языка Python
- Встроенные возможности по работе с коллекциями
- Потоковый ввод/вывод. Работа с текстовыми файлами. JSON

(Поток ввода (sys.stdin) — это специальный объект в программе, куда попадает весь текст, который ввёл пользователь. Потоком его называют потому, что данные хранятся в нем до тех пор, пока программа их не прочитала. Таким образом, данные поступают в программу и временно сохраняются в потоке ввода (sys.stdin), а программа может забрать их оттуда, например, с помощью встроенной функции input(). В момент прочтения, данные пропадают из потока ввода, так как он хранит их до тех пор, пока они не будут прочитаны. Поток ввода (sys.stdin) — является итератором, который невозможно перезапустить. Как и любой итератор, он может двигаться только вперёд. Как только данные прочитаны, они удаляются из потока ввода безвозвратно.)

Функции и их особенности в Python
- Функции. Области видимости. Передача параметров в функции
- Позиционные и именованные аргументы. Функции высших порядков. Лямбда-функции
- Рекурсия. Декораторы. Генераторы

Позиционные аргументы: Аргументы функции, которые передаются по порядку, в соответствии с их позицией в определении функции. При вызове функции значения аргументов должны быть переданы в том же порядке, в котором они определены.

Именованные аргументы: Аргументы функции, которые передаются с явным указанием имени параметра при вызове функции. При использовании именованных аргументов порядок передачи не имеет значения.

Пример:

```
def greet(name, age):
    print(f"Привет, {name}! Тебе {age} лет.")

# Позиционные аргументы
greet("Анна", 25)

# Именованные аргументы
greet(age=30, name="Петр")
```

Функции высших порядков - это функции, которые могут принимать другие функции в качестве аргументов или возвращать функции как результат. Они позволяют абстрагировать общие операции и писать более компактный и гибкий код.

Пример:

```
def apply_operation(operation, x, y):
    return operation(x, y)

def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

result1 = apply_operation(add, 3, 4)
result2 = apply_operation(multiply, 3, 4)
```

```
print(result1)  # Вывод: 7
print(result2)  # Вывод: 12
```

Лямбда-функции (или анонимные функции) в Python - это способ определения коротких функций без необходимости использования ключевого слова def. Они могут содержать только одно выражение и обычно используются там, где требуется краткость.

Пример:

```
add = lambda x, y: x + y
result = add(3, 4)
print(result)  # Вывод: 7
```

Рекурсия - это процесс, при котором функция вызывает саму себя внутри своего тела. Рекурсивные функции могут быть мощным инструментом для решения задач, которые могут быть разделены на более мелкие подзадачи того же типа.

Пример рекурсивной функции, вычисляющей факториал числа:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print(result)  # Вывод: 120
```

Важно при использовании рекурсии учитывать условие выхода из рекурсии, чтобы избежать бесконечного цикла.

Декораторы в Python - это специальный вид функций, которые позволяют изменять поведение других функций без изменения их собственного кода. Декораторы принимают функцию как аргумент, выполняют некоторую обертку или дополнительную логику, а затем возвращают функцию.

Пример декоратора, выводящего сообщение до и после выполнения функции:

```
def my_decorator(func):
    def wrapper():
        print("До выполнения функции")
        func()
        print("После выполнения функции")
    return wrapper

@my_decorator
def say_hello():
    print("Привет!")

say_hello()
```

Генераторы в Python - это специальный вид функций, которые возвращают последовательность значений по одному при каждом вызове. Они позволяют создавать итераторы без необходимости хранить все значения в памяти сразу.

Пример генератора, возвращающего квадраты чисел от 1 до n:

```
def squares(n):
    for i in range(1, n+1):
        yield i**2

for num in squares(5):
    print(num)
```

Генераторы удобны для работы с большими объемами данных или для создания бесконечных последовательностей. Они позволяют экономить память и улучшают производительность программы.

Объектно-ориентированное программирование

- Объектная модель Python. Классы, поля и методы
- Волшебные методы, переопределение методов. Наследование
- Модель исключений Python. Try, except, else, finally. Модули

1. Классы, поля и методы:
   - В Python классы являются основным механизмом для создания объектов. Класс определяет структуру объекта, включая его поля (атрибуты) и методы (функции, принадлежащие объекту).
   - Пример определения класса Person с полями name и age, а также методом greet:

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

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

2. Волшебные методы, переопределение методов:
   - В Python существуют "магические" (волшебные) методы, которые позволяют переопределить стандартное поведение операций для объектов определенного класса. Например, метод __str__ для представления объекта в виде строки.
   - Пример переопределения метода __str__ для класса Person:

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

    def __str__(self):
        return f"{self.name} ({self.age} years old)"
```

3. Наследование:
   - Наследование в Python позволяет создавать новый класс на основе уже существующего (родительского) класса. Новый класс (потомок) наследует атрибуты и методы родительского класса.
   - Пример создания класса Student, наследующего от класса Person:

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

4. Модель исключений Python:
- В Python исключения используются для обработки ошибок и исключительных ситуаций. Конструкции try, except, else и finally позволяют управлять потоком программы при возникновении исключений.
- Пример использования конструкции try-except для обработки деления на ноль:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Division by zero is not allowed.")
else:
    print("Division successful.")
finally:
    print("End of program.")
```

Библиотеки для получения и обработки данных

• Модули math и numpy

Модуль math в Python предоставляет функции для выполнения математических операций, таких как вычисление тригонометрических функций, логарифмов, округление чисел и т. д.

Модуль numpy предоставляет мощные инструменты для работы с массивами и матрицами в Python. Он позволяет выполнять операции линейной алгебры, генерировать случайные числа, работать с многомерными массивами и многое другое.

• Модуль pandas

Модуль pandas является мощным инструментом для обработки и анализа данных в Python. Он предоставляет структуры данных, такие как DataFrame, которые удобно использовать для работы с табличными данными.

• Модуль requests

Модуль requests позволяет отправлять HTTP-запросы к веб-серверам и получать ответы. Он упрощает работу с HTTP-запросами и позволяет взаимодействовать с внешними API.

Многопоточность в Python

Многопоточность в Python представляет собой возможность создания и управления несколькими потоками выполнения в рамках одного процесса. Основные концепции многопоточности в Python включают:

1) Потоки (threads): Потоки в Python позволяют выполнять несколько задач параллельно в рамках одного процесса. Потоки в Python реализованы с помощью модуля threading. Каждый поток имеет свой собственный стек вызовов, но разделяет память и ресурсы с другими потоками в рамках процесса.

2) Глобальная блокировка интерпретатора (Global Interpreter Lock, GIL): GIL является механизмом синхронизации, который обеспечивает потокобезопасность в CPython (стандартная реализация Python). GIL ограничивает выполнение только одного потока Python кода за раз, что может ограничивать эффективное использование многопоточности для CPU-интенсивных задач.

3) Модуль queue: Модуль queue в Python используется для безопасного обмена данными между потоками. Он предоставляет классы Queue, LifoQueue и PriorityQueue для организации очередей данных, которые могут быть использованы для синхронизации доступа к общим данным.

4) Семафоры, мьютексы и условные переменные: В Python также доступны различные механизмы синхронизации, такие как семафоры (Semaphore), мьютексы (Lock) и условные переменные (Condition), которые помогают контролировать доступ к общим данным и предотвращать состояния гонки.

Процесс работы многопоточности в Python заключается в создании нескольких потоков выполнения, которые могут выполняться параллельно и обмениваться данными. Для создания потока выполнения можно использовать класс Thread из модуля threading, например: thread = threading.Thread(target=someFunction). После создания потока, его можно запустить с помощью метода start().

При работе с многопоточностью в Python важно учитывать особенности GIL, избегать гонок данных и правильно использовать механизмы синхронизации для безопасного доступа к общим данным. Понимание этих концепций поможет разработчикам создавать эффективные и надежные параллельные программы на языке Python.

GO

Go — это компилируемый язык программирования со строгой статической типизацией, сборщиком мусора и встроенным менеджером пакетов. Он разработан с упором на многопоточное программирование.

С точки зрения ООП:

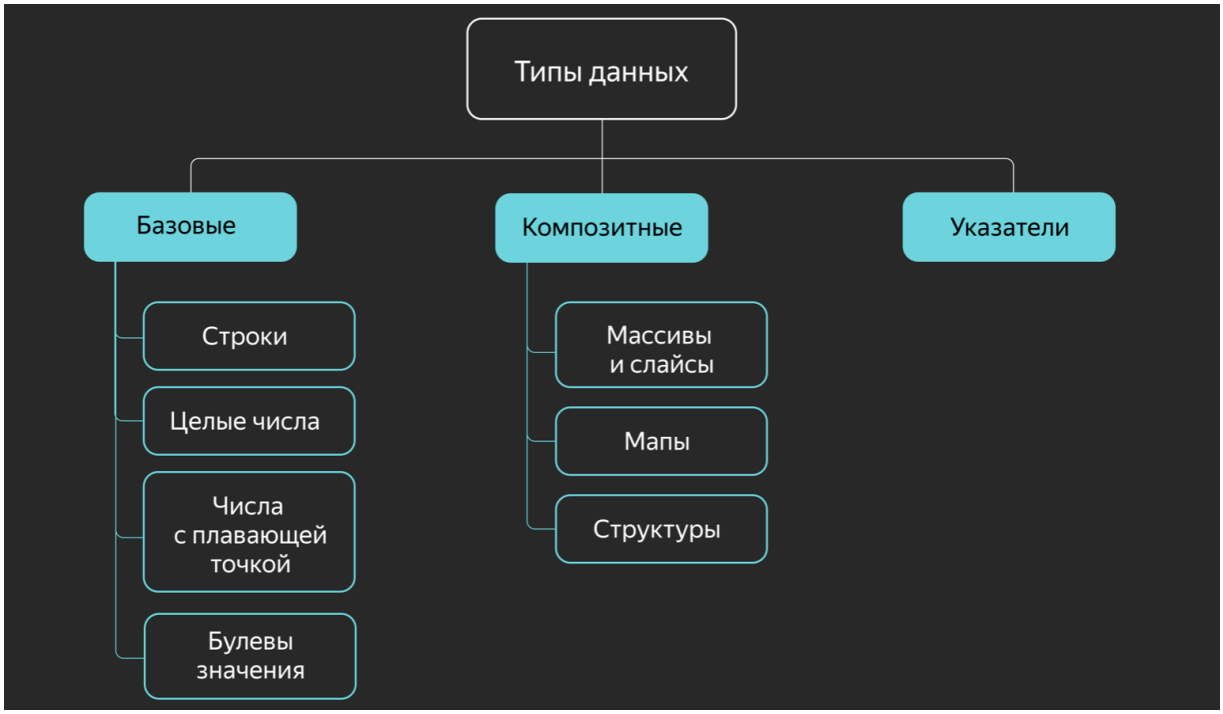
- **Абстракция** — В Go нет классов, но структуры с методами служат им неплохой заменой.
- **Инкапсуляция** — Go даёт возможность задать область видимости (публичные/приватные) методов структур и позволяет спрятать реализацию.
- **Наследование** — К сожалению, Go не реализует в полной мере механизм наследования, но есть встраивание — можно создавать типы на основе существующих.
- **Полиморфизм** — В Go нет полиморфизма в классическом понимании, однако похожие действия можно реализовать с помощью интерфейсов. Интерфейс определяет список методов, которые должен реализовывать тип, чтобы удовлетворять данному интерфейсу. Это ослабляет строгую типизацию и позволяет передавать в параметрах разные типы данных, поддерживающие один и тот же интерфейс.

С точки зрения функционального программирования:

- **Функции высшего порядка** — функции, которые могут в аргументах принимать другие функции и возвращать функции в качестве результата. В Go функции рассматриваются как значения и могут передаваться в другие функции, возвращаясь в виде результата.
- **Замыкания**. Go позволяет определять и использовать функции, которые ссылаются на переменные своей родительской функции.
- **Чистые функции**. В Go можно определять функции, которые зависят только от входящих аргументов и не влияют на глобальное состояние.
- **Рекурсия**. Как и в большинстве языков, в Go можно применять рекурсивные вызовы функций.
- **Ленивые вычисления**. В Go нет поддержки ленивых (отложенных) вычислений.
- **Иммутабельность переменных**. В Go переменные могут изменять своё значение, поэтому иммутабельность (неизменяемость) переменных отсутствует.

Выводы:
Видно, что Go полностью не реализует парадигмы объектно-ориентированного и функционального программирования, но частично это компенсируется похожими возможностями. Поэтому Go считается мультипарадигмальным языком программирования.

Базовые типы в Go



1.
Целочисленные типы:
- int: знаковый целочисленный тип, размер зависит от архитектуры (32 или 64 бита).
- int8, int16, int32, int64: знаковые целочисленные типы фиксированного размера.
- uint: беззнаковый целочисленный тип, размер зависит от архитектуры.
- uint8, uint16, uint32, uint64: беззнаковые целочисленные типы фиксированного размера.

2. Плавающая точка:
- float32: 32-битное число с плавающей точкой одинарной точности.
- float64: 64-битное число с плавающей точкой двойной точности.

3. Строки:
- string: последовательность символов Unicode.

4. Булев тип:
- bool: логический тип, принимает значения true или false.

5. Составные типы данных:
- array: фиксированный массив элементов одного типа.
- slice: динамический массив, представляет собой сегмент массива.
- map: ассоциативный массив (словарь), хранящий пары ключ-значение.
- struct: пользовательский тип данных, объединяющий различные поля под одним именем.

6. Указатели:
- pointer: переменная, содержащая адрес в памяти другой переменной.

7. Функции:
- Функции в Go являются типами данных и могут быть переданы как аргументы и возвращаемые значения.

8. Интерфейсы:

- Интерфейсы в Go определяют набор методов, которые должен реализовать тип данных для удовлетворения интерфейса.

9. Комплексные числа:
   - complex64: комплексное число с 32-битными компонентами.
   - complex128: комплексное число с 64-битными компонентами.

10. Байт и руны:
    - byte: синоним для типа uint8, представляющий байт данных.
    - rune: синоним для типа int32, представляющий Unicode кодовую точку.

Основные моменты:

1) Переменные и константы в Go:

- Переменные: Переменные в Go объявляются с использованием ключевого слова var и могут содержать различные типы данных. Пример объявления переменной:

```
var age int = 25
```

- Константы: Константы в Go объявляются с использованием ключевого слова const и должны быть инициализированы при объявлении. Пример объявления константы:

```
const pi = 3.14159
```

2) Область видимости в Go:

- Область видимости в Go определяет, где переменная может быть доступна в программе. Переменные могут быть объявлены с областью видимости на уровне пакета или на уровне функции.

3) Операторы ветвления в Go:

- Операторы ветвления в Go включают if, else if и else. Пример использования оператора if:

```
if age >= 18 {
    fmt.Println("Человек совершеннолетний")
} else {
    fmt.Println("Человек несовершеннолетний")
}
```

4) Циклы в Go:

- В Go используются циклы for, for range, break и continue для управления повторяющимися задачами. Пример использования цикла for:

```
for i := 0; i < 5; i++ {
    fmt.Println(i)
}
```

5) Указатели в Go:

- Указатели в Go представляют адреса памяти переменных. Они позволяют передавать переменные по ссылке. Пример объявления указателя:

```
var x int = 10
var ptr *int = &x
```

6) Массивы и слайсы в Go:

- Массивы: Массивы в Go представляют фиксированное количество элементов одного типа. Они объявляются с использованием [ ] и размером массива. Пример объявления массива:

```
var numbers [5]int
```

- Слайсы: Слайсы в Go являются динамическими массивами переменной длины. Они создаются без указания размера. Пример объявления слайса:

```
var numbers []int
```

7) Мапы в Go:

- Мапы (отображения) в Go представляют коллекцию пар ключ-значение. Они объявляются с использованием ключевого слова map. Пример объявления мапы:

```
var person = map[string]string{"name": "Alice", "city": "New York"}
```

8) Структуры в Go:

- Структуры в Go позволяют объединять различные типы данных в одну единицу. Они объявляются с использованием ключевого слова type и struct. Пример объявления структуры:

```
type Person struct {
    Name string
    Age  int
}
```

9) Функции в Go:

- Функции в Go объявляются с использованием ключевого слова func. Они могут принимать параметры и возвращать значения. Пример объявления функции:

```
func add(a, b int) int {
    return a + b
}
```

10) Операторы отложенного вызова в Go:

- Оператор defer в Go используется для отложенного выполнения функции до завершения текущей функции. Это может быть полезно для управления ресурсами. Пример использования оператора defer:

```
func main() {
    defer fmt.Println("Это будет выполнено в конце функции main")
    fmt.Println("Привет, мир!")
}
```

11) Пакеты и импорты в Go:

- Пакеты в Go являются коллекциями функций и объявлений, которые могут быть использованы в других пакетах. Импорт пакетов происходит с использованием ключевого слова import. Пример импорта пакета:

```
import "fmt"
```

12) Связь пакетов и файловой системы в Go:

- В Go каждый пакет обычно хранится в отдельной директории, а имя директории совпадает с именем пакета. Файлы внутри директории с пакетом должны иметь одинаковое имя и одинаковую декларацию пакета.

13) Модули в Go:

- Модули в Go представляют собой коллекцию пакетов, которые разделяют общий модульный путь. Модули помогают управлять зависимостями и версиями пакетов.

14) Внешние зависимости в Go:

- Внешние зависимости в Go управляются с помощью модулей и файлов go.mod. Разработчики могут добавлять и управлять внешними зависимостями с помощью инструментов управления модулями.

15) Методы в Go:

- Методы в Go представляют функции, связанные с определенным типом данных (структура или базовый тип). Они объявляются как функции, но с указанием получателя (типа). Пример объявления метода:

```
func (p Person) greet() {
    fmt.Println("Привет, меня зовут", p.Name)
}
```

16) Эмбеддинг в Go:

- Эмбеддинг в Go позволяет встраивать одну структуру в другую, чтобы унаследовать ее поля и методы. Это позволяет создавать иерархии типов и повторно использовать код. Пример эмбеддинга:

```
type Person struct {
    Name string
    Age  int
}

type Employee struct {
    Person
    Company string
}
```

17) Концепция интерфейсов и их реализация в Go:

- В Go интерфейсы определяют набор методов, которые должен реализовать конкретный тип данных. Реализация интерфейса происходит неявно, если тип имеет все методы, указанные в интерфейсе.

18) Интерфейсы в стандартной библиотеке Go:

- Стандартная библиотека Go содержит множество интерфейсов, которые используются для работы с различными функциональными возможностями языка. Например, интерфейс io.Reader и io.Writer для чтения и записи данных.

19) Пустой интерфейс и приведение типов в Go:

- Пустой интерфейс в Go представлен интерфейсом interface{} и может хранить значения любого типа. Приведение типов (type assertion) позволяет извлекать конкретный тип из пустого интерфейса. Пример:

```
var i interface{} = 42
num := i.(int)
```

20) Рефлексия в Go:

- Рефлексия в Go позволяет программам анализировать свою структуру во время выполнения. С помощью пакета reflect можно получать информацию о типах данных, вызывать методы и изменять значения.

21) Тип error в Go:

- Тип error в Go представляет собой интерфейс с методом Error() string, который используется для обработки ошибок. Функции могут возвращать ошибку, которая может быть проверена и обработана.

22) Функции panic и recover в Go:

- Функция panic вызывает аварийное завершение программы, а функция recover используется для восстановления контроля после паники. Они используются для управления критическими ситуациями.

23) Юнит-тесты и покрытие кода в Go:

- В Go юнит-тесты создаются с использованием пакета testing. Покрытие кода можно измерить с помощью утилиты go test -cover, которая показывает процент покрытия кода тестами.

24) Интерфейсы в тестировании в Go:

- В тестировании в Go интерфейсы часто используются для создания моков (заглушек), которые помогают изолировать тестируемый код от зависимостей. Интерфейсы позволяют легко заменять реальные объекты на заглушки при тестировании.

Многопоточность в Go

Многопоточность в Go (Golang) является одним из ключевых элементов языка, который позволяет эффективно использовать многоядерные процессоры и улучшить производительность приложений. Для понимания многопоточности в Go важно знать следующие основные концепции:

1) Горутины (goroutines): Горутины представляют собой легковесные потоки выполнения в Go. Они являются абстракцией над потоками операционной системы и позволяют создавать тысячи горутин в рамках одного процесса. Горутины обладают малым

потреблением памяти и накладными расходами, что делает их эффективными для параллельного выполнения задач.

2) Каналы (channels): Каналы в Go используются для обмена данными между горутинами. Они представляют собой типизированные каналы, через которые можно отправлять и принимать значения. Каналы обеспечивают безопасный и синхронизированный доступ к данным между горутинами.

3) Синхронизация (synchronization): В Go существует несколько механизмов синхронизации, таких как WaitGroups, Mutexes и RWMutexes, которые позволяют контролировать доступ к общим данным и предотвращать состояние гонки (race condition).

4) Select statement: Select statement в Go используется для выбора из нескольких операций ввода-вывода или коммуникации через каналы. Он позволяет организовать выборочное чтение или запись данных из нескольких каналов, что упрощает управление горутинами.

Процесс работы многопоточности в Go заключается в создании горутин, которые выполняют асинхронные задачи параллельно. Горутины могут быть запущены с помощью ключевого слова go, например: go someFunction(). После запуска горутины, она выполняется параллельно с другими горутинами и может обмениваться данными через каналы.

Для эффективного использования многопоточности в Go необходимо учитывать особенности синхронизации доступа к общим данным, избегать состояний гонки и правильно использовать каналы для обмена информацией между горутинами. Понимание этих концепций поможет разработчикам создавать эффективные и надежные параллельные программы на языке Go.

С точки зрения разработки:

Python:

Плюсы:
1) Простота и читаемость кода: Python имеет простой и понятный синтаксис, что делает его легким для изучения и использования.
2) Большое количество библиотек и фреймворков: Python имеет обширное сообщество разработчиков, что привело к созданию множества библиотек для различных задач (например, NumPy, Pandas, Django).
3) Мультипарадигменность: Python поддерживает различные стили программирования, такие как процедурное, объектно-ориентированное и функциональное программирование.
4) Подходит для быстрого прототипирования: Python позволяет быстро разрабатывать прототипы приложений благодаря своей простоте и высокому уровню абстракции.

Минусы:
1) Низкая производительность: Python является интерпретируемым языком, что может привести к медленной работе программ, особенно при выполнении вычислительно сложных задач.
2) Глобальная блокировка интерпретатора (GIL): GIL в Python ограничивает возможности многопоточности и параллельного выполнения кода.
3) Не подходит для некоторых типов приложений: Из-за низкой производительности, Python может быть не самым подходящим выбором для высоконагруженных систем или приложений, требующих быстрого выполнения.

Go (Golang):

Плюсы:
1) Высокая производительность: Go является компилируемым языком программирования, что обеспечивает высокую скорость выполнения программ.
2) Поддержка параллелизма: Go имеет встроенную поддержку параллелизма и многопоточности, что позволяет эффективно использовать многоядерные процессоры.

3) Простота и эффективность: Go предлагает простой и эффективный способ написания кода, что ускоряет разработку приложений.
4) Статическая типизация: Статическая типизация в Go помогает обнаруживать ошибки на этапе компиляции, что повышает надежность программ.

Минусы:
1) Относительно небольшое сообщество и экосистема: В сравнении с Python, сообщество разработчиков на Go не такое обширное, что может затруднить поиск решений или библиотек для определенных задач.
2) Меньшее количество библиотек и фреймворков: Несмотря на то, что Go имеет некоторые популярные библиотеки, выбор инструментов для разработки может быть ограничен.
3) Менее подходит для быстрого прототипирования: Из-за строгой статической типизации и некоторых особенностей языка, Go может потребовать больше времени на написание кода и создание прототипов.