**Основы языка**

**1. Интерпретатор Python и его регистрозависимость**

* Python — это интерпретируемый язык программирования, что означает, что код выполняется построчно интерпретатором Python. Вам не нужно компилировать весь код перед его запуском.
* Python - регистрозависимый язык, поэтому выражения print и Print или PRINT представляют разные выражения. И если вместо метода print для вывода на консоль мы попробуем использовать метод Print:
* Print("Hello World")

то у нас ничего не получится.

**2. Отступы: сердце Python**

* В отличие от многих других языков программирования, Python использует отступы для определения блоков кода. Правильное использование отступов критически важно для работы вашей программы.

Отступы в Python используются, чтобы показать, какие части кода входят в состав других частей. Можно представить это как структуру вложенных коробок, где каждая коробка может содержать внутри себя меньшие коробки.

Если сравнивать код с рассказом, то отступы помогают понять, когда начинается новая "глава" или "подглава". Например, если у вас есть условие (это как заголовок главы в книге), то все действия, которые должны произойти при выполнении этого условия, записываются с отступом вправо (как текст под заголовком).

**Пример с отступами:**

if условие:

действие1

действие2

Здесь **действие1** и **действие2** выполнятся только если **условие** истинно, потому что они "вложены" внутрь условия благодаря отступам. Это как если бы внутри коробки "Если условие истинно" лежали маленькие коробочки "Действие1" и "Действие2".

**Пример без отступов (неправильно):**

if условие:

действие1

действие2

Здесь Python не сможет понять, что **действие1** и **действие2** являются частью условия, потому что отсутствуют отступы. Это как если бы вы выложили все коробки на одном уровне — тогда непонятно, какая коробка к какой относится.

**3. Вывод информации: функция print()**

* Функция **print()** используется для вывода информации пользователю. В качестве аргументов *функции* может быть передано любое количество *объектов*, разделенных запятой.
* Примеры использования:
* print("Hello world!")
* print("Hello world!", "something else", 5)
* **Что такое функция?** Можно представить функцию как маленькую программу внутри вашей программы, которая делает какую-то конкретную задачу.
* **Что такое аргументы?** Аргументы — это информация или данные, которые вы передаете в функцию, чтобы она что-то с ними сделала. В случае с **print(5)**, число 5 — это аргумент.

print("Hello world!")

\*\*print(5)\*\*

Здесь вы говорите компьютеру: "Выведи слова 'Здравствуй, мир!' на экран". Компьютер "печатает" этот текст там, где вы можете его увидеть — обычно в консоли или терминале.

* **print(5)** просит компьютер показать число 5.
* **print("Я учу Python")** просит компьютер показать текст "Я учу Python".

**4. Комментарии: общение с будущим вами и другими разработчиками**

* Комментарии в коде играют важную роль. Они не влияют на выполнение программы, но предоставляют важную информацию о том, что делает ваш код.
* Однострочный комментарий начинается с **#**, а всё после **#** в этой строке Python игнорирует.
* # Это комментарий
* print("Это не комментарий")
* Также они могут располагаться на той же строке, что и инструкции языка, после выполняемых инструкций:

print("Hello World") # Это тоже комментарий

* В блочных комментариях до и после текста комментария ставятся три одинарные кавычки: '''текст комментария'''. Например:

'''

И это, кстати,

Тоже комментарий

И его кол-во строк неограниченно

'''

print("Hello World")

**Переменные и типы данных: Строки, числа (целые и вещественные), булевы значения.**

Переменные и типы данных — это основы любого языка программирования, включая Python. Переменные можно представить как ярлыки, которые "приклеиваются" к данным, чтобы мы могли легко на них ссылаться. Тип данных определяет, какого рода информацию может хранить переменная и что с этой информацией можно делать.

**Переменные**

В Python создать переменную можно, просто присвоив ей значение. Например:

name = "Алиса"

age = 30

is\_student = True

Здесь **name**, **age**, и **is\_student** — это переменные, которые хранят данные разных типов.

Название переменной в Python должно начинаться с алфавитного символа или со знака подчеркивания и может содержать алфавитно-цифровые символы и знак подчеркивания. И кроме того, название переменной не должно совпадать с названием ключевых слов языка Python

В пайтоне применяется два типа наименования переменных: **camel case** и **underscore notation**.

**Camel case** подразумевает, что каждое новое подслово в наименовании переменной начинается с большой буквы. Например:

NewValue = 1

**Underscore notation** подразумевает, что подслова в наименовании переменной разделяются знаком подчеркивания. Например:

new\_value = 1

Определив переменную, мы можем использовать ее в нашем коде. Например, мы можем вывести ее содержимое на консоль с помощью встроенной функции **print,** как делали это раннее:

new\_value = 1

print(new\_value)

Отличительной особенностью переменной является то, что мы можем менять ее значение в течение работы программы:

Переменные и типы данных — это основы любого языка программирования, включая Python. Переменные можно представить как ярлыки, которые "приклеиваются" к данным, чтобы мы могли легко на них ссылаться. Тип данных определяет, какого рода информацию может хранить переменная и что с этой информацией можно делать.

**Типы данных**

**Строки (str, или string)**

Строки используются для хранения текста. В Python строку можно определить, заключив текст в кавычки (одинарные **'** или двойные **"**):

greeting = "Привет"

city = 'Москва'

Строки можно складывать (конкатенировать), используя оператор **+**, чтобы сформировать новую строку:

greeting = "Привет"

city = "Москва"

full\_greeting = greeting + ", " + city + "!"

print(full\_greeting) # Выведет: Привет, Москва!

Если же мы хотим определить многострочный текст, то такой текст заключается в тройные двойные или одинарные кавычки:

'''

Это комментарий

'''

text = '''Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

'''

print(text)

При использовании тройных одинарных кавычек не стоит путать их с комментариями: если текст в тройных одинарных кавычках присваивается переменной, то это строка, а не комментарий.

Вставка значений в строку:

Python позволяет встраивать в строку значения других переменных. Для этого внутри строки переменные размещаются в фигурных скобках {}, а перед всей строкой ставится символ **f**:

city = "Москва"

user = f"город - {city}"

print(user) # Выведет: город - Москва

**Числа**

**Целые числа (int или integer)**

Используются для представления целых чисел, могут быть как положительными, так и отрицательными:

age = 25

temperature = -5

**Вещественные числа (float)**

Числа с плавающей точкой (или просто дроби) используются для хранения десятичных значений:

height = 1.75

price = 299.99

**Булевы значения (bool или boolean)**

Булевы (логические) значения могут принимать одно из двух значений: **True** (истина) или **False** (ложь). Они часто используются для проверки условий:

is\_student = True

is\_sunny = False

**Значение None**

**None** в Python — это специальный тип данных, представляющий отсутствие значения. Оно часто используется для инициализации переменных, к которым будет присвоено значение позже, или для указания на отсутствие результата в функциях.

Примеры использования **None**:

result = None

**Работа с переменными и типами данных**

Вы можете проверить тип переменной, используя функцию **type()**:

greeting = 'greeting'

age = 18

height = 181.54

is\_student = false

print(type(greeting)) # Выведет: <class 'str'>

print(type(age)) # Выведет: <class 'int'>

print(type(height)) # Выведет: <class 'float'>

print(type(is\_student)) # Выведет: <class 'bool'>

**Операции с данными: Арифметические операции, операции сравнения, логические операции.**

**Операции с данными в Python**

После знакомства с переменными и типами данных, важно понять, как можно манипулировать этими данными. В Python есть различные операции для работы с данными: арифметические, сравнения и логические операции.

**Арифметические операции**

Арифметические операции позволяют выполнять математические расчеты с числами.

* **Сложение** (**+**): Складывает два числа.
* **Вычитание** (-): Вычитает одно число из другого.
* **Умножение** (\*): Умножает два числа.
* **Деление** (**/**): Делит одно число на другое, результат — всегда вещественное число.
* **Целочисленное деление** (**//**): Деление с округлением результата до ближайшего меньшего целого числа.
* **Остаток от деления** (**%**): Возвращает остаток от деления.
* **Возведение в степень** (\*\*): Возводит число в заданную степень.

Примеры:

summ = 10 + 5 # 15

difference = 10 - 5 # 5

product = 10 \* 5 # 50

quotient = 10 / 5 # 2.0

floor\_division = 10 // 3 # 3

remainder = 10 % 3 # 1

power = 10 \*\* 2 # 100

print("Привет, " + "мир!") # Выведет: Привет, мир!

print("повтор" \* 3) # Выведет: повторповторповтор

**Операции сравнения**

Операции сравнения используются для сравнения двух значений и возвращают логическое значение (**True** или **False**).

* **Равно** (**==**): Проверяет, равны ли два значения.
* **Не равно** (**!=**): Проверяет, не равны ли два значения.
* **Больше** (**>**), **Меньше** (**<**), **Больше или равно** (**>=**), **Меньше или равно** (**<=**): Сравнивают два значения.

Примеры:

is\_equal = 5 == 5 # True

is\_not\_equal = 5 != 2 # True

is\_greater = 5 > 3 # True

is\_less = 5 < 3 # False

**Операции сравнения с None**

Для проверки значения на **None** обычно используется оператор сравнения **is** или **is not**, а не **==** или **!=**. Это обеспечивает точное сравнение с **None**

result = None

print(result is None) # True

print(result is not None) # False

**Логические операции**

Логические операции позволяют комбинировать логические значения и условия.

* **И** (**and**): Возвращает **True**, если оба условия истинны.
* **ИЛИ** (**or**): Возвращает **True**, если хотя бы одно из условий истинно.
* **НЕ** (**not**): Инвертирует логическое значение.

Примеры:

number\_one = 22

number\_two = 58

result = number\_one > 21 and number\_two == 58

print(result) # True

'''

В данном случае оператор and сравнивает результаты двух выражений:

age > 21 weight == 58

И если оба этих выражений возвращают True, то оператор and также возвращает True

'''

Так же мы можем делать различные операции внутри при передаче аргументов внутрь функции. Рассмотрим на примере print()

# Сложение

print("10 + 5 =", 10 + 5)

# Равно

print("5 == 5:", 5 == 5)

# И (AND)

print("True and False:", True and False)

* Больше примеров по логическим операциям
* '''Логическое И (AND)'''
* print(True and True) # Выведет: True
* print(True and False) # Выведет: False
* print(False and True) # Выведет: False
* print(False and False) # Выведет: False
* '''Логическое ИЛИ (OR)'''
* print(True or True) # Выведет: True
* print(True or False) # Выведет: True
* print(False or True) # Выведет: True
* print(False or False) # Выведет: False
* '''Логическое НЕ (NOT)'''
* print(not True) # Выведет: False
* print(not False) # Выведет: True
* '''Комбинирование логических операций'''
* # Комбинируем AND и OR
* print((True or False) and (False or False)) # Выведет: False

**ASSERT**

Ключевое слово assert в Python предназначено для проверки истинности определенных условий в коде. Если условие, проверяемое с помощью assert, ложно, программа автоматически прерывается

Определение и использование Определение: assert — это инструкция для проверки условий, которая помогает обеспечить корректность работы программы.

Использование: assert используется для автоматической проверки истинности условий. Если проверяемое условие истинно, программа продолжает свою работу. Если условие ложно, программа автоматически прерывается, что указывает на наличие ошибки. Синтаксис без сообщения об ошибке assert логическое выражение, которое проверяется на истинность **Пример** 1: Проверка, что число положительное:

number = 10

assert number > 0

'''Если number положительное, программа продолжит работу

иначе выполнение программы прервется

'''

**Пример** 2: Убедиться, что список не пуст

items = [1, 2, 3]

assert len(items) > 0

'''Здесь assert проверяет, что список items не пуст'''

**Пример** 3: Проверка истинности выражения

is\_valid = True

assert is\_valid

'''В этом примере assert используется для проверки,

что значение переменной is\_valid истинно'''

Зачем использовать В последствии когда мы будем писать автотесты, наши тесты смогут проходить либо успешно, либо “падать” (завершаться с ошибкой). Как раз таки именно эти проверки - наш основной инструмент для проверки логики.

**Коллекции данных: Списки, словари, кортежи, множества — создание, доступ к элементам, базовые операции.**

**Массивы в Python (Списки)**

Массивов в классическом понимании в Python нет, но есть списки, которые выполняют аналогичную функцию — упорядоченную коллекцию элементов.

Создание списка:

fruits = ["яблоко", "банан", "вишня"]

Основные методы работы со списками:

* **Добавление элемента**:
* fruits = ["яблоко", "банан", "вишня"]
* fruits.append("апельсин")
* print(fruits) #["яблоко", "банан", "вишня", "апельсин"]
* **Удаление элемента**:
* fruits = ["яблоко", "банан", "вишня"]
* fruits.remove("банан")
* print(fruits) #["яблоко", "вишня"]
* **Доступ к элементу** по индексу:
* fruits = ["яблоко", "банан", "вишня"]
* print(fruits[0]) # Выведет: яблоко
* **Доступ к элементу** по срезу индекса:

fruits = ["яблоко", "банан", "вишня"]

print(fruits[0:1]) # Выведет: яблоко

print(fruits[0:2]) # Выведет: яблоко, банан

print(fruits[0:3]) # Выведет: яблоко, банан, вишня

* **Изменение элемента**:
* fruits = ["яблоко", "банан", "вишня"]
* fruits[2] = "груша"
* print(fruits) #["яблоко", "банан", "груша"]
* Узнаем длину списка:
* fruits = ["яблоко", "банан", "вишня"]
* len\_list = len(fruits)
* print(len\_list) # выведет 3
* # можно сделать проще - сразу передав len(list) в виде аргумента в функцию print()
* print(len(fruits))

Списки в python могут содержать в себе различные типы данных:

mixed\_list = ["Алиса", 30, 1.75, True, [1, 2, 3], {"key": "value"}, (4, 5, 6)]

print(mixed\_list) # Выведет ['Алиса', 30, 1.75, True, [1, 2, 3], {'key': 'value'}, (4, 5, 6)]

**Словари**

Словарь в Python — это неупорядоченная коллекция элементов, где каждый элемент хранится как пара ключ-значение.

Создание словаря:

person = {"имя": "Алиса", "возраст": 30}

Основные методы работы со словарями:

* **Добавление или изменение элемента**:
* person["город"] = "Москва"
* **Удаление элемента**:
* del person["возраст"]
* **Доступ к элементу** по ключу:
* print(person["имя"]) # Выведет: Алиса
* **Проверка наличия ключа**:
* print("город" in person) # Выведет: True

У словарей значение может быть практически любым типом данных, включая другие словари (объекты), списки (массивы), и переменные:

Словари внутри словаря

person = {

"имя": "Алиса",

"возраст": 30,

"адрес": {

"город": "Москва",

"улица": "Тверская"

}

}

# Доступ к данным во вложенном словаре

print(person["адрес"]["город"])

Списки внутри словаря

person = {

"имя": "Алиса",

"возраст": 30,

"хобби": ["путешествия", "фотография", "чтение"]

}

# Доступ к элементу списка внутри словаря

print(person["хобби"][1]) # Выведет: фотография

Переменные как значения в словаре

name = "Алиса"

age = 30

person = {

"имя": name,

"возраст": age

}

print(person["имя"]) # Выведет: Алиса

print(person["возраст"]) # Выведет: 30

Добавление и изменение элементов

* Добавление нового хобби в список "хобби":

person["хобби"].append("рисование")

print(person["хобби"]) # Выведет: ['путешествия', 'фотография', 'чтение', 'рисование']

* Изменение города в адресе:

person["адрес"]["город"] = "Санкт-Петербург"

print(person["адрес"]["город"]) # Выведет: Санкт-Петербург

Удаление элементов

* Удаление элемента из списка "хобби":

person["хобби"].remove("чтение")

print(person["хобби"]) # Выведет: ['путешествия', 'фотография', 'рисование']

* Удаление пары "ключ-значение" из словаря:

del person["возраст"]

print(person) # Выведет словарь без пары "возраст"

**Кортежи**

Кортеж — это неизменяемая упорядоченная коллекция элементов. Похож на список, но не позволяет изменять элементы после создания.

Создание кортежа:

coordinates = (10, 20, 30)

Работа с кортежами:

* **Доступ к элементу** по индексу:
* print(coordinates[1]) # Выведет: 20
* Кортежи **не поддерживают изменение элементов**, попытка это сделать вызовет ошибку.

**Изменяемые и неизменяемые типы данных**

В Python типы данных можно разделить на две категории: изменяемые и неизменяемые. Понимание разницы между ними критически важно, поскольку это влияет на то, как данные ведут себя при передаче в функции или при их изменении.

**Неизменяемые типы данных**

Неизменяемые типы данных не могут быть изменены после их создания. Это значит, что каждый раз, когда кажется, что вы меняете неизменяемый объект, на самом деле создается новый объект, а переменная теперь ссылается на него.

Основные неизменяемые типы данных в Python:

* Целые числа (**int**)
* Вещественные числа (**float**)
* Кортежи (**tuple**)
* Строки (**str**)
* Логические значения (**bool**)

**Изменяемые типы данных**

Изменяемые типы данных могут быть изменены после создания. Изменение содержимого объекта не приводит к созданию нового объекта; переменная продолжает ссылаться на тот же объект.

Основные изменяемые типы данных в Python:

* Списки (**list**)
* Словари (**dict**)
* Множества (**set**)

Давайте подробнее рассмотрим примеры с изменяемыми и неизменяемыми типами данных, используя функцию **id()** для демонстрации того, как изменяется (или не изменяется) идентификатор объекта при различных операциях.

**Неизменяемые типы данных: int и str**

**Целые числа (int)**

a = 5

print(id(a)) # Например, выводит 140703205164864

a += 2

print(id(a)) # Выводит другой идентификатор, например, 140703205164928

**Строки (str)**

name = "Alice"

print(id(name)) # Например, выводит 4321429136

name += " Bob"

print(id(name)) # Выводит другой идентификатор, например, 4321456784

В обоих примерах, когда мы "изменяем" значение переменной, на самом деле создается новый объект, и переменная ссылается уже на него, а не на исходный объект.

**Изменяемые типы данных: list и dict**

**Списки (list)**

numbers = [1, 2, 3]

print(id(numbers)) # Например, выводит 4321463456

numbers.append(4)

print(id(numbers)) # Выводит тот же идентификатор, например, 4321463456

Вот пример, демонстрирующий, как изменение изменяемого объекта (например, списка) через одну переменную может неожиданно отразиться на другой переменной, которая ссылается на тот же объект. Это связано с тем, что обе переменные на самом деле указывают на один и тот же объект в памяти.

**Пример**

# Изначально создаем список

original\_list = [1, 2, 3]

print("Исходный список:", original\_list) # Выведет: [1, 2, 3]

# Создаем вторую переменную, которая ссылается на тот же список

reference = original\_list

print("Вторая ссылка на список:", reference) # Выведет: [1, 2, 3]

# Добавляем элемент в список через вторую переменную

reference.append(4)

print("Второй список, который ссылается на первый:", original\_list) # Выведет: [1, 2, 3, 4]

print("Измененный через вторую переменную исходный список:", original\_list) # Выведет: [1, 2, 3, 4]

# Проверяем идентификаторы объектов

print("ID исходного списка:", id(original\_list)) # id: 139950223614208

print("ID второй ссылки на список:", id(another\_reference)) # id: 139950223614208

В этом примере **original\_list** и **another\_reference** обе указывают на один и тот же список в памяти. Поэтому, когда мы изменяем список через **another\_reference**, изменения отражаются и на **original\_list**.

**Для избегания подобный случаев используем метод .copy() - он создаст новый объект в новой ячейки памяти, и тогда каждая ссылка будет ссылаться на разные id в ячейках памяти:**

original = [5, 4, 3]

copy\_from\_original = original.copy()

copy\_from\_original.append(6)

print("наш первый список", original) # Выведет: наш первый список [5, 4, 3]

print('список через метод .copy()', copy\_from\_original) # Выведет: список через метод .copy() [5, 4, 3, 6]

print("id памяти для первого списка: ", id(original)) # id памяти для первого списка: 2133566731072

print("id памяти для скопированного списка: ", id(copy\_from\_original)) # id памяти для скопированного списка: 2133566694464

**Словари (dict)**

person = {"name": "Alice", "age": 30}

print(id(person)) # Выводит 4321527680

person["age"] = 31

print(id(person)) # Выводит тот же идентификатор 4321527680

# Добавление нового ключа-значения также не изменяет id

person["city"] = "New York"

print(id(person)) # Все еще 4321527680

**Попытка изменения неизменяемого объекта**

# Кортежи (`tuple`) являются неизменяемыми

coordinates = (10, 20)

print(id(coordinates)) # Например, 4321587648

# Попытка изменить кортеж вызовет ошибку

coordinates[0] = 15 # Выводит 'tuple' object does not support item assignment

Для вывода  текста  можно использовать встроенную функцию **print()**,

**\n** заменяется  на символ перевода строки

**\t** заменяется на символ табуляции

 Можно выводить строки не через пробел, а через произвольный разделитель, для чего используется необязательный аргумент **sep**

**Если не писать sep напрямую (как мы это делали раньше), то он будет автоматически заменен на sep=" "**

Чтобы **print**не переводил строку (чтобы каждый следующий **print**не выводил текст с новой строки) можно использовать необязательный параметр функции - **end**

 input() считывает **одну строку текста** из консоли и подставляет ее там, где она была вызвана

Функция **input()** может принимать необязательный аргумент - строку, которая будет выведена перед тем, как считывать строку из консоли

print("Приветствуем вас,", input("Назовись, путешественник"))

Назовись, путешественник

Приветствуем вас, <введенное имя>

**input**может вызываться либо **без аргументов**:

input()

ничего не выводит, но считывает введенную пользователем строку

Либо **с одним аргументом**, который будет выведен при запуске функции

input('Enter your name')

вначале выведет строку:

Enter your name

после чего считает введенную пользователем строку

Вызвать **input()** с двумя или более аргументами не получится:

input('первый аргумент', 'второй аргумент')

вызовет ошибку:

TypeError: input expected at most 1 argument, got 2