# Урок 1. Установка Пайтон, пайчарм, Pep8

1.1. Установка Пайчарм и Пайтон на комп.

Установили

2.1. Стандарты PEP

<https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html>

PEP8 - документ предлагает единый и общепринятый стиль написания программ на языке Python.

PEP8:

- Избегайте использования пробелов перед открывающей скобкой.

- После запятой нужен пробел

- Комментарии должны отделяться хотя бы двумя пробелами от кода (или на новой строке). должны начинаться с символа # и одного пробела.  
  
  
**Задание**

Вот код с простыми ошибками PEP8:

python

имя="Вася"

возраст=25

def сказать\_привет():

print("Привет!")

print(f"Меня зовут {имя}")

print(f"Мне {возраст} лет")

def сложить\_числа(a,b):

результат=a+b

return результат

сказать\_привет()

сумма=сложить\_числа(5,3)

print(f"Сумма: {сумма}")

Решение

**Для проверки кода на соответствие стандартам PEP 8**

bash

**«python -m black --check .»**

**"Проверить красоту кода без изменений"**

**Что делает:**

* 🔍 **Смотрит** все Python файлы в папке
* 📏 **Проверяет** отступы, пробелы, переносы строк
* ❌ **Сообщает** если код некрасивый
* ✅ **НЕ меняет** файлы (только проверяет)

**«python -m isort --check-only .»**

**"Проверить порядок импортов без изменений"**

**Что делает:**

* 📚 **Смотрит** все import строки
* 🔄 **Проверяет** что импорты отсортированы правильно:
  + Сначала встроенные модули (import os)
  + Потом сторонние (import requests)
  + Потом свои (from my\_module import something)
* ❌ **Сообщает** если импорты в беспорядке
* ✅ **НЕ меняет** файлы

**«python -m flake8 .»**

**"Найти ошибки и плохой стиль в коде"**

**Что ищет:**

* 🐛 **Ошибки** - синтаксис, несуществующие переменные
* 📝 **Стиль** - слишком длинные строки, неиспользуемые переменные
* 🎯 **Качество** - сложный код, потенциальные баги

Устанавливаем flake8

pip install flake8

Можно установить все 3 линтера сразу

pip install black flake8 isort

# Используй:

python -m flake8 .

# Тема 2. Git, Гитхаб, настройка подключения

## 2.1. Git

Git - система контроля версий.

Git - программа, которая отслеживает изменения в файлах и сохраняет их историю.

**Проверяем, есть ли на компе git**

git --version

если есть ответ с версие значит, есть

**Основные команды Git**

**1. git init - "Начать отслеживание" (создается репозиторий в папке с проектом)**

bash

# Создает репозиторий для папки с проектом

git init

**2. git add - "Добавить файлы" (подготовить файлы к сохранению)**

bash

# **Добавить один файл**

git add main.py

**# Добавить ВСЕ файлы**

git add .

**3. git commit - "Сохранить версию"**

bash

# Сохранить текущее состояние с комментарием

git commit -m "Мой первый автотест"

**4. git status - "Что изменилось?"**

bash

# Показывает какие файлы изменены

**1. git config --global user.name " George-GS"**

Устанавливает твоё имя для всех Git-репозиториев

**2. git config --global user.email ‘georgegarber11@gmail.com’**

Устанавливает твою почту для всех Git-репозиториев

**Команды для перехода по папкам**

1. Посмотреть где ты сейчас:

pwd

2. Посмотреть что в текущей папке:

dir

3. Перейти в другую папку:

cd полный путь\_к\_папке

git log – посмотреть историю комитов

git log --oneline (КОРОТКАЯ информация) только id и изменение

**Как вернуться к старому коммиту?**

git checkout <commit Id> (id берем из информации в git log oneline)

Отменить изменения до индексации (до того как сделали git add, просто когда внесли изменния в файл)

git checkout <file name>

Восстанавливает файл к состоянию последнего коммита

**Отмена индексации** изменений (после того как сделали git add, но еще не сделали комит)

git reset HEAD <file name>

Сами изменения не изменяются, просто индексация (git add) отменяется

**Отмена коммитов**

git revert HEAD

Создает НОВЫЙ коммит, который отменяет изменения последнего коммита

**git commit --amend —** это изменение последнего коммита

# Создать и переключиться на новую ветку

git switch -c новая\_ветка

Создать ветку, но остаться в текущей

git branch новая\_ветка

**git switch** — команда для переключения между ветками

# Переключиться на ветку master

git switch master

я создал новую ветку "ветка 1", сделал в ней изменение, закомитил их, переключился в мастер ветку, там этих изменений не будет.

Если изменения удачные – могу объединить их в одну

# Объединить ветку\_1 в master

git switch master

git merge [имя ветки] # c какой веткой объединить ту ветку, в которой находимся

После мержа ветка ветка\_1 продолжит существовать. Просто изменения прокинуть в главную ветку.

git branch # увидишь все ветки

# Удалить ветку

git branch -d [имя ветки]

шпоргалка по GIT

<https://training.github.com/downloads/ru/github-git-cheat-sheet/>

## 2.2. Github

[**https://github.com/**](https://github.com/)

**мой логин George-GS**

**почта Gmail**

GitHub — это платформа для хостинга Git-репозиториев и совместной работы.

GitHub = "Облачное хранилище" для твоего кода + социальная сеть для разработчиков

**Git vs GitHub:**

Git — инструмент на твоём компьютере для контроля версий

GitHub — сайт в интернете где хранятся твои репозитории

📝 **Основные понятия GitHub:**

1. **Repository** (репозиторий)

Папка с твоим проектом на GitHub

Может быть public (все видят) или private (только ты)

2. **Clone** (клонирование)

bash

# Скопировать репозиторий с GitHub на компьютер

git clone https://github.com/username/repository-name.git

**3. Push (отправка)**

bash

# Отправить твои коммиты на GitHub

git push origin main

**4. Pull (загрузка на комп)**

bash

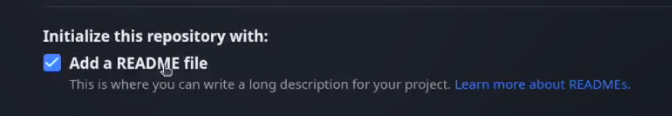
# Скачать изменения с GitHub

git pull origin main

**Твои первые шаги на GitHub:**

**1. Создать аккаунт на**[**github.com**](https://github.com/)

**2. Создать новый репозиторий**

* Нажать "New repository"
* Выбрать имя (например my-python-tests)
* Выбрать Public/Private (приватный сильно урезан по функциям)
* Поставить галку   
  

Чтобы инициализировать репозиторий, в нем должен быть хотя бы один файл

**Что такое add .gitignore:**

* .gitignore — это файл-инструкция для Git, в котором перечислено что **НЕЛЬЗЯ** добавлять в репозиторий.

**💡 Простая аналогия:**

* .gitignore**= "Чёрный список" для Git**
* "Не следи за этими файлами и папками"
* "Не добавляй их в коммиты"
* "Игнорируй их полностью"

**🚀 Что добавляют в .gitignore:**

**1. Системные файлы:**

* .idea/ - настройки PyCharm

**2. Временные файлы:**

* \_\_pycache\_\_/ - кеш Python
* \*.log - логи приложений

**3. Конфиденциальные данные:**

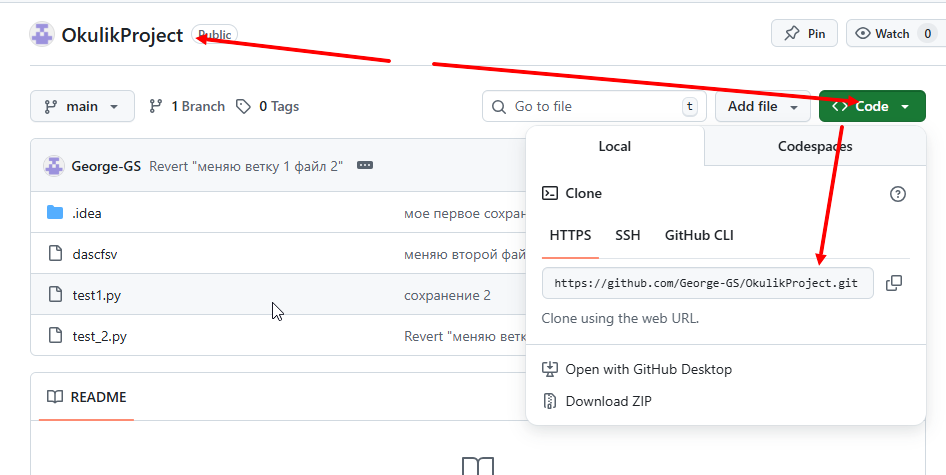
* config.py с паролями
* Файлы с персональными данными

### 3. Связать локальный репозиторий с GitHub

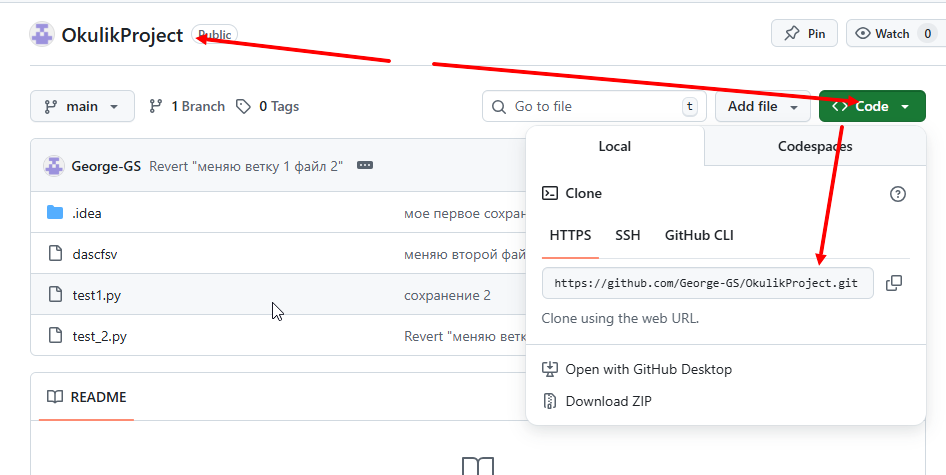
Bash

1. Перейти в терминале в нужную папку
2. git clone <https://github.com/George-GS/OkulikProject.git>

и вводим ссылку из гитхаба



**Второй способ**

1. *# Добавить удалённый репозиторий*
2. git remote add origin <https://github.com/твой-логин/имя-репозитория.git>  
    
3. *# Отправить код на GitHub*
4. git push -u origin main (но если главная ветка называется master, то в конце слово мастер, а не мэйн)

**ВЫШЕ ОПИСАН СПОСОБ ЧЕРЕЗ HTTPS, НО ПРАВЛЬНО СВЯЗЫВАТЬ ЧЕРЕЗ SSH (потому что безопаснее? Не надо вводить пароли как в HTTPS, ну и некоторые работодатели требуют, что только SSH)**

### 🚀 Настройка SSH для GitHub

**Способ 1.**

bash

1. ssh-keygen -t ed25519 -C "???????@gmail.com"

* **-** На вопрос Enter file in which to save the key → **Enter**
* На вопрос Enter passphrase → **Enter** (два раза)

1. Переходим в путь, который был указан в терминале в шаге 1  
   открываем файл с id\_ed\_\_\_519.pub через блокнот и копруем значение
2. Заходим в настрйоки в гитхабе
3. Вклдка ssh
4. New ssh
5. Вводим значение ключа ()
6. bash вводим (клонируем)   
   git clone [git@github.com:George-GS/OkulikProject.git](mailto:git@github.com:George-GS/OkulikProject.git)

**Пайчарм – файл – open**

**Выбираем папку**

**КОНЕЦ**

**Шаг 1: Генерируем SSH ключ**

bash

ssh-keygen -t ed25519 -C "georgegarber11@gmail.com"

* На вопрос Enter file in which to save the key → **Enter**
* На вопрос Enter passphrase → **Enter** (два раза)

**Шаг 2: Запускаем SSH-агент (в PowerShell)**

bash

Start-Service ssh-agent

**Шаг 3: Добавляем ключ в агент**

bash

ssh-add C:\Users\gs.shabarov\.ssh\id\_ed25519

**Шаг 4: Копируем публичный ключ**

bash

cat C:\Users\gs.shabarov\.ssh\id\_ed25519.pub

**СКОПИРУЙ ВСЮ ВЫВОДИМУЮ СТРОКУ** (начинается с ssh-ed25519)

**Шаг 5: Добавляем ключ в GitHub**

1. Открой [github.com/settings/keys](https://github.com/settings/keys)
2. **New SSH key**
3. **Title**: Мой компьютер Windows
4. **Key**: вставь скопированную строку
5. **Add SSH key**

**Шаг 6: Проверяем подключение**

bash

ssh -T git@github.com

1. **Должен быть ответ:**

text

Hi George-GS! You've successfully authenticated...

**💡 После настройки SSH:**

**Для СУЩЕСТВУЮЩЕГО репозитория:**

bash

*# Переключаем на SSH*

git remote set-url origin git@github.com:George-GS/OkulikProject.git

**Для НОВОГО клонирования:**

bash

git clone git@github.com:George-GS/OkulikProject.git

################################

В Windows: нужно сначала сменить диск D:, потом cd

bash

# Сначала переключись на диск D:

D:

# Потом перейди в папку PyCharm

cd D:\PyCharm

######################################################

Как выполнять домашнее задание:

ДЗ №3 – «создай файлик с 1 буквой внутри»

1) иду на свой комп, переходим в ту папку (синхронизированную) с проектом

2) для каждого дз нужно создать ветку, в которой мы его будет выполнять

- создаем ветку из ветки main

git checkout -b 'GS\_homework\_3'

3) В общей папке Homework – создаем свою папку «Shabarov», и в ней для каждого дз создаем отдельную папку Homawork\_1 и т.д.

4) создаем файлик .py

5) git status

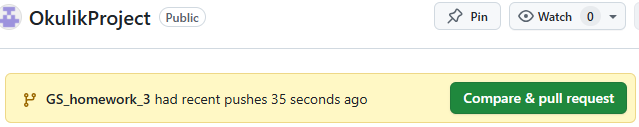
6) git add .  
6.5 git commit –m ‘Сделал ДЗ4’

7) git push origin vetka3 (видимо пушим в новую ветку)  
- если мы пушим по указанной ветку первый раз, то просто git push – будет ошибка, так как ветка в гитхабе и в локальном репозитории не связаны

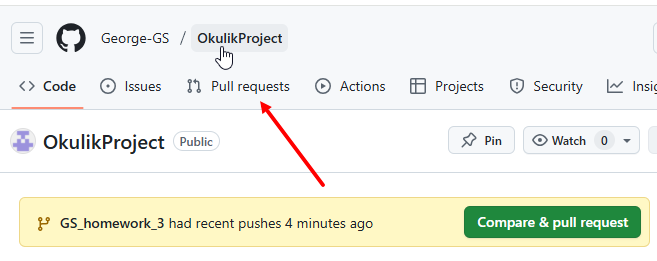
- ‘git push’ можно пистаь тотолько если ранее уже пушились по ветке

- а если не связаны ветки - всегда указывать git push origin vetka3

8) переходим в гитхаб – там появляется уведомление, что мы запушили изменения, не хотим ли сделать пул реквест

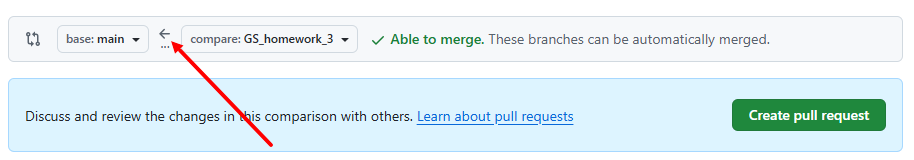


9) Но это сообщение не всегда отображается сразу, поэтому можно обойти и можно перейти на вкладку пул реквест



Нажимаем new pull request – мы отправляем запрос на то, что изменения в моей ветке, были добавлены в ветку main (но это только запрос, еще не само слияние)

10) выбираем из какой ветки и в какую ветку закидывает изменения – выбираем из ветки homework 3 в main (там есть стрелочка)



11) нажимаем create pull request

- переименуем название PR в «Shabarov HW3»

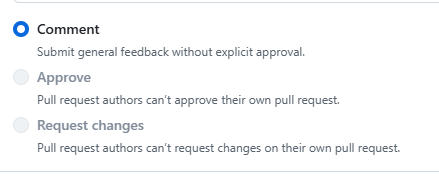
-если хотим добавляем комментарий

12) снова нажимаем create pull request

13) если это лично ваш репозитори, то у вас прсото появится кнопка Merge PR

14) Но если это чужой – то вряд ли так будет (скорее всего будет у того лица, кто отвечает за этот репозиторий), в этом случае нужно ждать апрува (одобрения мержа)

Если ты смотриim чужой пул реквест, то будет 3 вараинтта что с ним делать:



- если нужно будет что-то исправить то нужно выбрать - третий вариан

- если все ок и мержим – то второй

**Pull Request**

Pull Request (PR) — это запрос на объединение изменений из одной ветки в другую

Что такое Pull Request простыми словами:

PR = "Эй, посмотрите на мои изменения и если всё ок - добавьте их в основной код!"

**Как это работает на практике:**

**1. Ты работаешь в отдельной ветке**

bash

git switch -c my\_new\_feature

*# делаешь изменения...*

git add .

git commit -m "Добавил новую функцию"

git push origin my\_new\_feature

**2. Создаёшь Pull Request на GitHub**

1. Заходишь на страницу репозитория на GitHub
2. Видишь предложение "Compare & pull request"
3. Заполняешь описание что изменил
4. Нажимаешь "Create pull request"

**3. Кто-то проверяет твой код**

* Смотрит изменения
* Оставляет комментарии
* Может попросить исправить что-то

**4. Если всё ок - мержат в основную ветку**

* Нажимается кнопка **"Merge pull request"**

**🎯 Зачем это нужно:**

**Для командной работы:**

* **Code Review** - другие разработчики проверяют твой код
* **Обсуждение** - можно обсуждать изменения до мержа
* **Тестирование** - можно запустить автотесты перед мержем

**Даже если работаешь один:**

* **История изменений** - видно что и когда менялось
* **Визуальный обзор** - удобно смотреть разницу между ветками
* **CI/CD** - можно настроить автоматические проверки

Вот мы сделали ДЗ 3, надо сделать ДЗ 4:

1. Проверяем, на какой ветке мы находимся – git status (результат, находимся в ветке Дз 3)
2. Переключаемся на ветку main, так как все новые ветки мы создаем от главное ветки - git switch main
3. Получаем данные главной ветки с облачного репозитория на локальный – git pull
4. Создаем и переключаемся новую ветку «Дз 4» - git switch –c 'GS\_Homework\_4'
5. Создаем в своей папке как обычно новую папку «homework\_4'
6. Создаем новый файл с дз, в нем выполняем дз
7. сохранить всё что сейчас есть в папке для комита - git add .
8. комитим изменения - git commit -m 'Сделал дз 4 задача1'

Пока не хочу пушить, потому что впереди еще нужно сделать Задачу№2

В этот момент мне приходит комментарий по ДЗ 3, что все неправильно, нужно пределать

1. переключаюсь в пайчарме на ветку ДЗ 3 – git switch GS\_homework\_3
2. исправляю код в фале дз
3. далее опять как обычно добавляем изменения, комитим, пушим

- git add .

- git commit -m ‘Переделал дз 4 задача1'

- git push origin GS\_Homework\_4

1. При этом последний комит попадет в тот же пул реквест, новый создавать не надо
2. Ну и все Проверяющий мержит наши изменения
3. Конец!

Важно!!! Почему мы для каждого ДЗ создаем нвоую ветку, потому что если бы мы пушили находясь в ветке main, то изменения прокидывались в гитхаб напрямую в главную ветку, без пул реквеста и кодревью. Так делать неправильно!!!

**Когда ты работаешь напрямую в**main**и пушишь:**

* Изменения сразу попадают в main на GitHub
* **Без Pull Request** - потому что ты работаешь в основной ветке
* **Без код-ревью** - потому что это прямой коммит

**Проблемы прямого коммита в main:**

* ❌ Нет код-ревью (никто не проверил твой код)
* ❌ Риск сломать основной код
* ❌ Нет истории обсуждения изменений

Правильный workflow на будущее:

Всегда работать в отдельной branches:

# Урок 3. Базовый python

### Идентификаторы

**Идентификаторы** в Python это имена используемые для обозначения переменной, функции, класса, модуля или другого объекта.

**Правила для написания идентификаторов:**

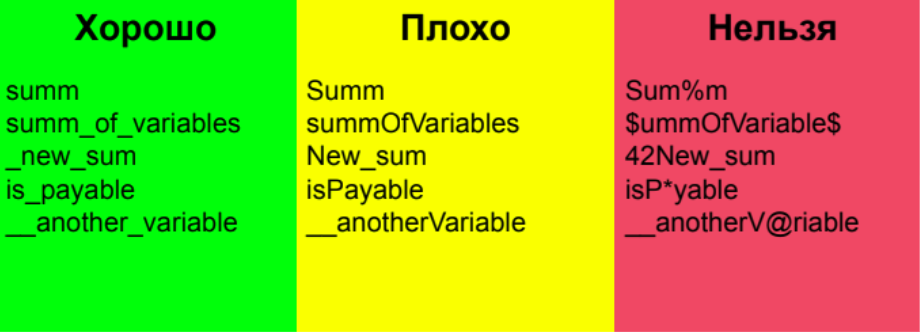
Идентификаторы могут быть комбинацией букв в нижнем регистре (a - z) или верхнего регистра (A - Z) или цифр (0 - 9) или подчеркивание \_. Например: myClass, var\_1 и print\_this\_to\_screen.

Идентификатор не может начинаться с цифры. 1variable неправильно, но variable1 правильно.

Ключевые слова не могут использоваться как идентификаторы.

Нельзя использовать специальные символы !, @, #, $, %.

Идентификаторы могут быть любой длины.

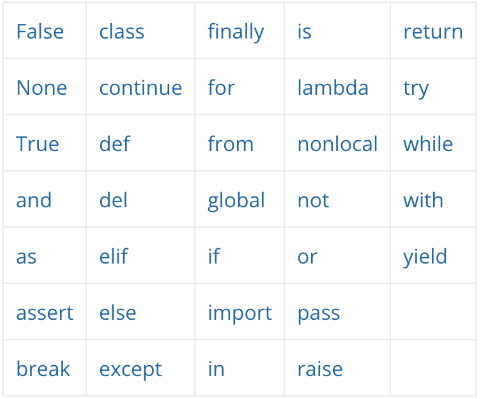


**Зарезервированные слова**

Мы не можем использовать ключевые слова как имена переменных, имена функций или какой-нибудь любой другой идентификатор.

Они используются, для определения синтаксиса и структуры в языке Python. Ключевые слова регистрозависимые.

Все слова за исключением True, False и None в маленьком регистре.



Python – регистрозависимый язык. Это означает, что `Variable` и `variable` не одно и тоже.

Называйте идентификаторы осмысленно.

Несколько слов могут быть разделены подчеркиванием this\_is\_a\_long\_variable.

`snake\_case` (undercore notation) - стандарт для именования всего кроме классов

### ## Основные типы данных

1. int - integer - целые числа (4)

2. float - числа с плавающей точкой (4.2)

3. str - string - строки ('My cool text')

4. bool - boolean - логические значения: True или False

Это не все типы данных, а только основные.

print(42)

print(42.5)

print('text')

print(True)

print(False)

**Пустота:**

python

* ничего = None *# NoneType (отсутствие значения)*

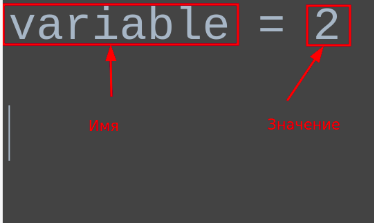
**## Переменная**

Именованный участок памяти, обладающий некоторым типом.

Таким образом, в Python у переменной всегда есть:

Имя (название)

Значение



Переменная - это элемент, который хранит в себе какие либо данные.

Если мы создали переменную, то она никогда не бывает абсолютно пустой. В ней может храниться, например, пустая строка. Тогда переменная хранит строку, которая пустая. И хранит ее до тех пор, пока программа не завершит работу.

empty\_str = ''

**### Объявление переменной**

Переменные в Python не типизированные. Это значит, что при создании переменной не нужно указывать какой тип данных в ней будет храниться. Более того, переменную не нужно объявлять заранее, до того как присвоить ей какие-либо данные. Объявление переменной происходит простым присвоением ей значения:

a = 45

print(2 + a)

После присвоения, переменная сама знает какой тип данных в ней хранится. У переменной всегда есть тип (конечно, если значение не None). Но этот тип НЕ неизменнен. Вы всегда можете переназначить содержимое переменной значением любого типа

b = None

**📦 3. ПЕРЕМЕННЫЕ**

**Создание:**

python

имя = "Вася" *# создал переменную*

возраст = 25 *# ещё переменная*

**Изменение:**

python

имя = "Петя" *# изменил значение*

число = 10

число = число + 5 *# теперь 15*

**Множественное присвоение:**

python

x, y, z = 1, 2, 3 *# x=1, y=2, z=3*

a = b = c = 0 *# все три = 0*

**5. ДИНАМИЧЕСКАЯ ТИПИЗАЦИЯ**

python

* переменная = 10 *# сейчас int*
* переменная = "текст" *# теперь str (автоматически)*

**6. ПРОВЕРКА ТИПОВ**

python

* type(переменная) *# <class 'int'>*
* isinstance(переменная, str) *# False*

**## Операторы**



**4. ОПЕРАТОРЫ**

**Арифметические:**

python

5 + 2 *# 7 → сложение*

5 - 2 *# 3 → вычитание*

5 \* 2 *# 10 → умножение*

5 / 2 *# 2.5 → деление*

5 // 2 *# 2 → целочисленное деление*

5 % 2 *# 1 → остаток от деления*

5 \*\* 2 *# 25 → степень*

**Сравнения:**

python

5 == 5 *# True → равно*

5 != 3 *# True → не равно*

5 > 3 *# True → больше*

5 < 3 *# False → меньше*

5 >= 5 *# True → больше или равно*

5 <= 3 *# False → меньше или равно*

**Логические:**

**and / or / not**

python

True and True *# True → И (оба True)*

True or False *# True → ИЛИ (хотя бы один True)*

not True *# False → НЕ (инвертирует)*

**Присваивания:**

python

x = 5 *# просто присвоить*

x += 3 *# x = x + 3 → 8*

x -= 2 *# x = x - 2 → 6*

x \*= 2 *# x = x \* 2 → 12*

**### Операторы сравнения**

Все эти операторы используются для того чтобы сравнивать какие-либо значения. Результатом такого сравнения будет True или False

**Операторы принадлежности**

print('e' **in** 'hello')

**Оператор идентичности (is)**

c = 'text'

d = с или d = 'text'

print(c **is** d) # True

d = 256

c = 256

print(c is d) # True

d = 257

c = 257

print(c is d) # False

Причина: Python кэширует только числа от -5 до 256

**### Порядок операций**

Если в выражении встречается больше, чем один оператор, то порядок вычислений зависит от правил старшинства (rules of precedence). Для математических операций, Python следует математическим соглашениям. Аббревиатура PEMDAS является простым способом для запоминания правил:

1. Parentheses - Скобки

2. Exponentiation - Возведение в степень

3. Multiplication - Умножение

4. Division - Деление

5. Addition - Сложение

6. Subtraction - Вычитания