# Введение в Python

*Программа читает список сообщений и считает длинные. На одном примере она права, на другом молча ошибается. Как заметить это до запуска?*

Python можно установить на компьютер и запускать программы самостоятельно. В наших простых упражнениях Python работает прямо в браузере благодаря Pyodide. Ничего устанавливать не нужно: измените код и нажмите «Запустить». При первом запуске браузер загрузит Python; для этого нужен интернет.

## 1. Числа и переменные

Python можно использовать как калькулятор. Функция `print()` выводит результат в область под программой. Запустите пример, затем замените числа и запустите ещё раз.

### Калькулятор

Цель: Запустите пример и измените одно выражение.

```
print(2 + 3)
print(10 / 4)
print(2 ** 3)
```

Получатся три строки: `5`, `2.5` и `8`. Знаки `+`, `-`, `*` и `/` означают сложение, вычитание, умножение и деление. Две звёздочки `**` возводят число в степень. Умножение выполняется раньше сложения: `2 + 3 * 4` даёт 14. Скобки меняют порядок: `(2 + 3) * 4` даёт 20.

Для дробной части используют точку: `2.5`. Обычное деление `/` возвращает число с дробной частью, даже если остатка нет: `8 / 2` даёт `4.0`. Ещё две операции пригодятся для целых чисел:

```
print(17 // 5)  # 3: целая часть частного
print(17 % 5)   # 2: остаток от деления
```

Текст после `#` называют комментарием. Python его не выполняет; здесь комментарии показывают ожидаемый результат.

*Профи.* **Деление с остатком: почему `//` и `%` ходят парой**

Две операции связаны одним равенством: для любых целых `a` и `b` при `b != 0` выполняется `a == (a // b) * b + a % b`. Для 17 и 5: $3 \cdot 5 + 2 = 17$. Поэтому пара «частное и остаток» переводит минуты в часы: `125 // 60` даёт `2` часа, `125 % 60` — 5 минут, и $2 \cdot 60 + 5 = 125$.

Подвох появляется у отрицательных чисел. Python округляет частное вниз, к меньшему целому, а не к нулю: `-17 // 5` даёт `-4`, потому что $-4 < -3{,}4$. Тогда остаток обязан дополнить равенство: $-17 - (-4) \cdot 5 = 3$, и `-17 % 5` равно `3`. Остаток в Python всегда имеет знак делителя, а не делимого: при положительном делителе он лежит от нуля до `b - 1`. В языках C и Java те же выражения дали бы `-3` и `-2`.

Как поймать ошибку: если программа переводит разницу времён в «часы и минуты» и разница бывает отрицательной, проверьте один отрицательный пример вручную, подставив его в равенство выше. Полезная сторона того же правила: `(day + shift) % 7` переносит день недели через границу недели без единого `if`, и при отрицательном сдвиге тоже.

> В Python переменная `a` равна −13. Что напечатает `print(a % 7)`?

**Переменная** хранит значение под именем. Знак `=` присваивает значение: сначала Python вычисляет выражение справа, затем сохраняет результат под именем слева.

```
count = 3
count = count + 2
print(count * 4)  # 20
```

После первой строки `count` равен `3`, после второй — `5`. Выражение `count = count + 2` берёт старое значение и заменяет его новым. Это присваивание, поэтому его не нужно читать как математическое равенство.

Последняя строка печатает 20, но сама переменная по-прежнему равна 5. Чтобы изменить её ещё раз, понадобится присваивание, например `count = count * 4`. Имя чувствительно к регистру: `count` и `Count` — разные переменные. Внутри имени нельзя ставить пробелы; несколько слов обычно соединяют подчёркиванием: `message_count`.

**Пример**

Запишите на бумаге два столбца: выполненная строка и значение `count`. После первой строки будет 3, после второй 5. Вычисление внутри print не меняет count: в третьей строке в этом столбце снова будет 5.

> В Python сначала `count = 3`, затем `count = count + 3`. Что напечатает `print(count * 2)`?

## 2. Строки и списки

Кавычки меняют смысл записи. `3 + 4` даёт число 7, а `"3" + "4"` склеивает две строки и даёт текст `"34"`. Строкой можно хранить сообщение целиком: `message = "Встреча-`

емся в семь" . Кавычки ограничивают текст в программе и не входят в само сообщение. Если написать `print(message)` , вы увидите только его содержимое.

Целое число имеет тип `int` , число с дробной частью обычно имеет тип `float` , текст имеет тип `str` . **Тип** определяет допустимые действия. Умножение строки на целое повторяет её: `"да" * 2` даст `"дада"` . Складывать число со строкой без явного преобразования нельзя. Ошибка в таком месте означает, что программе передали значения несовместимых типов.

Тип значения можно узнать с помощью `type` . Преобразование `int("12")` даст число 12, а `str(12)` — строку `"12"` . Запись `int("кот")` вызовет `ValueError` : этот текст нельзя прочитать как целое число.

```
print(type(12))     # <class 'int'>
print(type(2.5))    # <class 'float'>
print(type("12"))   # <class 'str'>
print(int("12") + 3)  # 15
```

*Эпик.* **Когда 0,1 + 0,2 не равно 0,3**

Правило «Python считает точно» верно для целых: `10 ** 20 + 1` хранится без потерь. Для `float` оно ломается: `print(0.1 + 0.2 == 0.3)` даёт `False` , а `print(0.1 + 0.2)` покажет `0.30000000000000004` . Число хранится в двоичной системе, и 0,1 в ней — бесконечная дробь, как 1/3 в десятичной. Записывается ближайшее представимое число, а ошибка округления накапливается при сложениях: `sum([0.1] * 10)` даёт `0.9999999999999999` , а не `1.0` .

Оценка порядка: одна операция вносит относительную ошибку около $10^{-16}$; после миллиона сложений в худшем случае накопится $10^{-10}$ — для среднего времени поездки незаметно, для сравнения `==` фатально. Поэтому дробные числа не сравнивают на равенство, а проверяют близость: `abs(a - b) < 1e-9` или `math.isclose(a, b)` . Контрпример к привычке «отброшу дробную часть и сравню»: `int(0.9999999999999999)` равен `0` , хотя до единицы не хватает одной десятиквинтиллионной.

Перенос: ту же ошибку легко сделать в условии цикла. Цикл «прибавлять 0,1, пока сумма не равна 1» никогда не остановится; цикл «прибавлять, пока сумма меньше 1» остановится, но сделает на один шаг больше, чем вы ожидали.

Переменная `total` в Python — сумма дробных чисел, которая по смыслу должна равняться единице. Какая проверка надёжно это подтвердит?

- `total == 1.0`
- `str(total) == "1"`
- `abs(total - 1) < 1e-9`
- `int(total) == 1`

Для нескольких длин создадим **список**: `lengths = [12, 30, 18, 40]`. Элементы записываются через запятую в квадратных скобках. Позицию элемента называют индексом. Первый индекс равен нулю, поэтому `lengths[0]` даёт 12, а `lengths[2]` даёт 18. Индекс `-1` обращается к последнему элементу: это 40.

Срез `lengths[1:3]` возвращает новый список `[30, 18]`: начало включается, правая граница исключается. `len(lengths)` считает элементы, `sum(lengths)` складывает их. Поэтому средняя длина равна `sum(lengths) / len(lengths)`, то есть 25.0. Запись `lengths.append(22)` добавит ещё одну длину в конец. После неё размер списка изменится, а среднее потребуется вычислить заново.

```
lengths = [12, 30, 18, 40]
print(lengths[0])       # 12
print(lengths[1:3])     # [30, 18]
print(len(lengths))     # 4
print(sum(lengths))     # 100
```

В Python есть список `[5, 8, 13, 21, 34]`. Какова сумма элементов среза `[1:4]` этого списка?

**Заметка**

У списка из четырёх элементов нет элемента с индексом 4. Последний положительный индекс равен 3. Сообщение `IndexError` предлагает проверить границы списка.

## 3. Условия и циклы

Сравнение `length > 20` проверяет условие. Результат имеет логический тип: `True`, если условие выполнено, или `False`, если нет. Для проверки равенства используются два знака: `==`. Один знак `=` сохраняет значение. Например, `length == 20` сравнивает длину с двадцатью, а `length = 20` записывает двадцать в переменную.

Условная команда `if` выполняет вложенные строки только при истинном условии. После условия ставится двоеточие; вложенные строки сдвигаются вправо на четыре пробела. Ветка `else` задаёт действие для остальных случаев. Строгое сравнение `>` не включает сам порог: длина двадцать не больше двадцати.

```
length = 30
if length > 20:
    print("Длинное сообщение")
else:
    print("Короткое сообщение")
```

Цикл `for` выполняет одни и те же строки для каждого элемента списка. Переменная `length` по очереди получает значения 12, 30, 18 и 40. Вложенные строки выполняются с этим значением, затем начинается следующая итерация.

```
lengths = [12, 30, 18, 40]
count = 0
for length in lengths:
    if length > 20:
        count = count + 1
print(count)  # 2
```

Счётчик увеличится для 30 и 40. Строка `print(count)` стоит без отступа и выполнится один раз после цикла. Если поставить её внутрь цикла, получится несколько строк вывода. В опыте ниже меняйте порог: при пороге 30 останется одно подходящее число, при 40 — ни одного.

## Проверьте свой прогноз




Проверяем length > 20. Счётчик: 0 → 1 → 1 → 2. Результат: 2.

---

Карточка окрашивается, если длина строго больше порога. Под карточками показано изменение счётчика.

*Интерактивный эксперимент — в электронной версии урока.*

При пороге 20 подходят длины 30 и 40. Счётчик после четырёх сравнений меняется так: 0, 1, 1, 2. При пороге 30 останется только длина 40. Условие применяется отдельно

к каждому элементу, поэтому изменение порога способно исключить сразу несколько элементов или не изменить результат вовсе.

> Счётчик начинают с нуля. Цикл перебирает `[7, 14, 21, 28, 35]` и увеличивает его на один для каждого числа строго больше 20. Чему равен итоговый счётчик?

**Подумай**

При пороге 19 и пороге 20 выбранные длины одинаковы. Между 18 и 30 в списке нет новых значений. Отсюда и целый диапазон порогов с одним ответом.

## 4. Функции

Повторяющийся расчёт можно оформить как **функцию**. Команда `def` задаёт имя и входные значения в круглых скобках. Входы называют аргументами. Строка `return` возвращает результат тому месту программы, где функция была вызвана.

```
def square(x):
    return x * x

print(square(3))  # 9
print(square(5))  # 25
```

Строка `def square(x):` определяет функцию; расчёт начнётся только при её вызове. В первом вызове вместо `x` подставляется `3`, во втором — `5`. Возвращённое число можно сохранить: `area = square(5)`. Если функция закончилась без `return`, результатом будет `None`, специальное значение «нет значения».

В упражнении функция получает список длин и порог. Счётчик начинается с нуля внутри функции: каждый вызов считает заново. Цикл берёт очередную длину; условие решает, прибавлять ли единицу. Строка `return` стоит после цикла, на уровне его начала. Если сдвинуть её внутрь цикла, функция завершится уже на первом элементе.

Исправьте строку с пометкой `TODO`. Сначала проверьте обычный список, затем пустой и случай, когда все длины равны порогу. Пустой список не выполняет тело цикла ни разу, поэтому правильный результат равен нулю. Для списка `[20, 20]` результат тоже должен быть нулём: условие использует строгое сравнение.

### Посчитать длинные сообщения

Цель: Допишите функцию: считать длины строго больше порога.

```
def count_long(lengths, threshold):
    count = 0
    for length in lengths:
        if length > threshold:
            count = count + 0  # TODO: учтите подходящий элемент
    return count

print(f"Длинных сообщений: {count_long([12, 30, 18, 40], 20)}")
```

**Ссылки**

Учебник Python показывает условия, циклы и определение функций. Сначала разберите наши короткие примеры; справочник пригодится для уточнения синтаксиса.

*Профи.* **Таблица состояний: проверить функцию без запуска**

Функцию с циклом удобно проверять таблицей: столбец на каждую переменную, строка на каждую итерацию. Для `count_long([12, 30, 18, 40], 20)` строки такие: до цикла `count = 0`; после 12 — `0`; после 30 — `1`; после 18 — `1`; после 40 — `2`; `return 2`.

Таблица показывает три типичные ошибки до запуска. Если `count = 0` стоит внутри цикла, в каждой строке счётчик начинается заново и итог равен единице или нулю. Если `return` сдвинут внутрь цикла, таблица обрывается после первой строки. Если в условии `>=` вместо `>`, строка с длиной, равной порогу, прибавляет единицу.

Четвёртая ошибка тоньше: в функции два накопителя, и один из них делится не на то число. Среднее длинных сообщений равно сумме длинных, делённой на их количество, а не на длину всего списка. На списке `[30, 40]` обе формулы дают 35, и ошибка не видна; на `[12, 30, 18, 40]` ответы 35 и 17,5 различаются.

Проверочный пример обязан различать правильную и ошибочную формулы: если на нём обе дают одно число, пример бесполезен. Допишите функцию ниже и проверьте её на пустом списке: без защиты от деления на ноль там возник бы `ZeroDivisionError`.

### Среднее длинных сообщений

Цель: Исправьте знаменатель: делите на число подходящих длин.

```
def mean_long(lengths, threshold):
    total = 0
    count = 0
    for length in lengths:
        if length > threshold:
            total = total + length
            count = count + 1
    if count == 0:
        return 0
    return total / len(lengths)  # TODO: делите на число подходящих

print(mean_long([12, 30, 18, 40], 20))  # ожидается 35.0
```

## 5. Ошибки и повторения

Сообщение об ошибке помогает понять, где выполнение остановилось. Начните с последней строки: там указан тип ошибки и причина. `NameError` часто означает опечатку в имени, например `lenghts` вместо `lengths`. `IndexError` указывает на выход за границы. При `SyntaxError` проверьте кавычки, скобки и двоеточия рядом с отмеченной строкой.

Бывает, что программа завершается без ошибки, но считает неверно. Тогда уменьшите пример до нескольких значений и запишите промежуточные результаты. Если счётчик всякий раз начинается с нуля, проверьте, не попала ли его инициализация внутрь цикла. Если функция вернула `None`, проверьте, выполняется ли `return`. Печать результата и возвращение результата выполняют разные задачи.

Для повторения действия заданное число раз используют `range`. Вызов `range(2, 5)` даёт последовательно 2, 3 и 4; правая граница снова не включается. Цикл по этому диапазону выполнит тело три раза. Здесь действует то же правило границы, которое уже встречалось в срезах.

Для проверки цикла временно добавьте `print(length, count)` внутрь него. Вы увидите текущий элемент и счётчик после каждого прохода. Затем уберите этот вывод, чтобы программа снова печатала только итог.

*Эпик.* **Сколько стоит цикл в цикле**

Поиск повторов «в лоб» — вложенный цикл: для каждого элемента пройти по всем элементам списка. Тело внутреннего цикла выполняется $n \cdot n$ раз. Чистый Python делает порядка $10^7$ простых итераций в секунду. Для списка из 1000 длин это $10^6$ сравнений — десятая доля секунды. Для 100 000 сообщений — $10^{10}$, около четверти часа; для миллиона — $10^{12}$, больше суток. Процессор вдвое быстрее здесь не спасёт: рост квадратичный, и увеличение входа в десять раз даёт стократное время.

Оценку стоит делать до написания кода, а не после того, как программа «зависла». Противоположный крайний случай тоже полезен: если список пуст, вложенный цикл не выполнится ни разу, и функция обязана вернуть осмысленный ответ без ошибки.

Перенос: та же задача за один проход решается множеством `set`. Идём по списку; если элемент уже есть в множестве — повтор найден, иначе добавляем. Проверка принадлежности множеству занимает почти постоянное вре-

**История**

Python создавался как язык с читаемым синтаксисом; его название связано с Monty Python. История в официальном FAQ.

**Заметка**

Комментарии после `#` поясняют программу человеку. Python их не выполняет. Комментарий `# TODO` отмечает строку для вашей правки, а не отдельную команду языка.

мя, и миллион элементов обрабатывается за секунду. Выбор структуры данных, а не скорость набора кода, определяет, дождётесь ли вы ответа.

> Программа на Python перебирает список из 5 тысяч чисел вложенным циклом: для каждого элемента внешнего цикла внутренний проходит по всем элементам того же списка. Сколько миллионов раз выполнится тело внутреннего цикла?

`name = value` сохраняет значение, `print(name)` печатает его.

`if` выбирает действие по условию, `for` повторяет его для элементов списка.

`def` определяет функцию, `return` возвращает результат вызова.

## Где продолжить

Здесь разобраны только азы Python. Для самостоятельного написания программ понадобится больше практики: ввод данных, строки и словари, работа с файлами, разбиение программы на функции. Выберите один из русскоязычных курсов и решайте задачи по порядку:

- Д. П. Кириенко: программирование на Python — курс школы № 179 на «Информатикс».
- «Поколение Python»: курс для начинающих — последовательное введение в язык с упражнениями на Stepik.
- Программирование на Python, Институт биоинформатики — вводный курс с видео и задачами, решения которых проверяются автоматически.

*Где это встретится дальше*

NumPy выполняет вычисления над числовыми массивами; pandas работает с таблицами. Продолжение: уроки «Массивы и вычисления в NumPy» и «Разведочный анализ данных в Python». Примеры на английском: массивы в D2L и обработка таблиц в D2L.

**Домашняя работа**

1. Напишите на Python функцию суммы положительных чисел. Проверьте `[-2, 3, 5]` (ответ 8) и пустой список (0).
2. В Python получите первые три элемента списка `[4, 7, 9, 12]` срезом.

3. ★ Функция Python должна считать длины строго больше порога. Найдите ошибку и предложите вход, на котором она заметна.

```
def count_long(lengths, threshold):
    count = 0
    for length in lengths:
        if length > threshold:
            count += 1
        return count
```

## Источники

- IDS Skoltech: введение в программирование (https://ids.skoltech.ru/seminars/seminar01/01-programming-basics.html).
- Python: числа, строки и списки (https://docs.python.org/3/tutorial/introduction.html).
- Python: условия, циклы и функции (https://docs.python.org/3/tutorial/controlflow.html).

## Повторить эту тему